• Prentice Hall PTR

UNIX

网络编程（第1卷）

套接口 API 和 X/Open 传输接口 API

[美] W.Richard Stevens 著
施振川 周利民 孙宏晖 等译
杨继张 审校

第 2 版

清华大学出版社

前 言

简介

网络编程即编写通过计算机网络与其他程序进行通信的这类程序。相互通信的网络程序中,一方称为客户程序(client),另一方称为服务器程序(server)。大多数操作系统提供预先编译好的网络程序,例如 TCP/IP 世界中常见的 Web 客户程序(浏览器)和 Web 服务器程序,以及 FTP 和 Telnet 的客户和服务器程序,不过本书叙述如何编写自己的网络程序。

我们使用应用程序编程接口(application programming interface)即 API 编写网络程序。本书将叙述两种网络编程 API:

1. 套接口(sockets),有时称为"Berkeley 套接口",因为它源自 Berkeley Unix。
2. XTI(X/Open 传输接口),它是对 AT&T 开发的传输层接口(TLI)经少量修改的产物。

书中所有例子均取自 Unix 操作系统,不过网络编程所需的基础知识和概念在很大程度上并不依赖于操作系统。这些例子都基于 TCP/IP 协议族,既有 IP 版本 4,又有版本 6。

编写网络程序要求人们了解底层的操作系统和网络协议。本书是作者在这两个领域的其他四本书的基础上书写的,它们的名称在文中缩写如下:

- APUE:*Advanced Programming in the UNIX Environment*[Stevens 1992]
- TCPv1:*TCP/IP Illustrated, Volume 1*[Stevens 1994]
- TCPv2:*TCP/IP Illustrated, Volume 2*[Wright and Stevens 1995]
- TCPv3:*TCP/IP Illustrated, Volume 3*[Stevens 1996]

本书(英文名称为 UNIX Network Programming)第二版仍包含 Unix 和 TCP/IP 协议的有关内容,但同时又给出不少其他四本书的参考点,这样有兴趣的读者就可获得各个主题更为详细的信息。TCPv2 这本书更是如此,因为它叙述并提供了套接口 API 中网络编程函数(socket,bind,connect 等等)在 4.4BSD 上的真正实现。要是人们理解某个特性的实现,他们在应用程序中使用这个特性将更为明智。

本书与第一版的差别

本书第二版是对第一版的完全重写。这些修改基于作者 1990 年至 1996 年期间大约每月一次讲授这些内容的反馈意见,以及同期跟踪某些 Usenet 新闻组的结果,通过这种跟踪能发现经常被误解的那些主题。下面是新版本中所做的主要改动:

- 新版本中所有例子都使用 ANSI C。
- 老版本的第 6 章("Berkeley 套接口编程")和第 8 章("库例程")已经扩充成总共 25 章。从词数上统计这部分内容扩充了七倍,这也许是从第一版到第二版本的最重大的改动。第一版第 6 章的许多单节已扩充成完整一章,其中增加了更多的例子。

- 老版本第 6 章中的 TCP 和 UDP 部分现已被分开，首先讨论的是 TCP 函数以及一个完整的 TCP 客户-服务器程序例子，接着讨论的是 UDP 函数以及一个完整的 UDP 客户-服务器程序例子。以 connect 函数为例，这么做对于初学者来说要比描述其所有细节更易理解，毕竟在 TCP 和 UDP 中它有不同的语义。
- 老版本的第 7 章（“系统 V 传输层接口编程”）已经扩充成总共 7 章。另外我们讨论的是更新的 XTI API，而不是它所替代的 TLI API。
- 老版本的第 2 章（“Unix 模型”）现已删去。这一章提供约 75 页的 Unix 系统概貌。在 1990 年这一章是必要的，因为那时市面上几乎没有充分叙述基本的 Unix 编程接口的书，更不用说叙述 Berkeley 与系统 V 实现之间的差异了。然而，今天的更多读者对 Unix 已有基本的认识，因此诸如进程 ID、保密字文件、目录和组 ID 等概念不必重述。（对于期望了解 Unix 编程更多细节的读者来说，我的 APUE 一书是这些内容的 700 页扩充。）

  老版本第 2 章中一些高级主题在新版本中仍讨论到，不过它们是分散在各自用到的地方讨论的。例如，在给出第一个并发服务器程序时（4.8 节），我们讨论 fork 函数。在叙述并发服务器如何处理 SIGCHLD 信号时（5.9 节），我们说明 Posix 信号处理的许多其他特性（如僵尸进程、被中断的系统调用等等。）
- 本书尽可能描述 Posix 接口。（我们在 1.10 节中具体介绍标准中的 Posix 一族。）这不仅包括基本 Unix 函数（进程控制、信号等等）的 Posix.1 标准，而且包括套接口和 XTI 网络 API 的即将出台的 Posix.1g 标准，以及线程的 1996 Posix.1 标准。

  在描述诸如 socket 和 connect 等函数时，“系统调用”的称谓改用“函数”代替。这里沿用了 Posix 的一个约定，即系统调用与库函数的区别是一个实现上的细节，通常与程序员无关。
- 老版本的第 4 章（“网络入门”）和第 5 章（“通信协议”）现已被替换成涵盖 IP 版本 4（IPv4）和版本 6（IPv6）的附录 A 以及涵盖 TCP 和 UDP 的第 2 章。这些新内容的焦点在于网络程序员确实会碰到的那些网络协议问题。IPv6 的内容也包括在内，尽管它的实现才刚刚开始出现，然而在本书的生命期内它也许会成为占支配地位的网络协议。

  说实话，我在讲授网络编程时发现，所有的网络编程问题中约有 80%跟网络编程本身是不相关的。也就是说，这些问题不是针对诸如 accept 和 select 等 API 函数的，而是因缺乏对底层网络协议的理解而引起的。例如，我发现一旦学生理解了 TCP 的三路握手和四分组连接终止序列，许多网络编程问题也就迎刃而解了。

  老版本中关于 XNS、SNA、NetBIOS、OSI 协议以及 UUCP 的章节现已删去，因为这些专属协议在 20 世纪 90 年代初期与 TCP/IP 协议相比已显得黯然失色。（UUCP 仍然流行，而且也不是专属的，不过从网络编程角度看就它的使用而言也没有什么可说的。）
- 第二版包含下列新的主题：
  - IPv4 和 IPv6 的互操作性（第 10 章）
  - 协议无关的名字转换（第 11 章）
  - 路由套接口（第 17 章）

- 多播(第 19 章)
- 线程(第 23 章)
- IP 选项(第 24 章)
- 数据链路访问(第 26 章)
- 客户-服务器程序其他设计方法(第 27 章)
- 虚拟网络与隧道通路(附录 B)
- 网络程序调试技术(附录 C)

遗憾的是,对本书第一版内容所做的扩充如此之多,从而无法放在单独一本书中。因此,这套《UNIX 网络编程》系列丛书至少还有另外两卷也在写作计划之中。

- 卷 2 的副标题或许是《IPC:进程间通信》(英文名称为 IPC: Interprocess-Communication),它由老版本的第 3 章扩充而来,同时涵盖 1996 Posix.1 实时 IPC 机制。
- 卷 3 的副标题或许是《应用程序》(英文名称为 Applications),它是对老版本中第 9 至第 18 章的扩充。

尽管大多数网络应用程序将在卷 3 中介绍,但是少量特殊的应用程序还要在本书中介绍,它们是:Ping,Traceroute 和 inetd。

读者

本书既可用作网络编程的指导书,也可作为有经验程序员的参考书。当用作指导书或网络编程入门班的教材时,重点应放在第 2 部分"基本套接口编程"(第 3 章~第 9 章),再跟一些感兴趣的其他主题。第 2 部分的内容包括 TCP 和 UDP 的基本套接口函数、I/O 复用、套接口选项以及基本的名字与地址转换。第 1 章所有读者都必须阅读,特别是 1.4 节,它描述了一些全文都用到的包裹函数。第 2 章及附录 A 读者应根据自己的背景知识选读。第 3 部分"高级套接口编程"的大多数章节可彼此独立地阅读。

为便于用作参考书,本书提供了全文索引,在附录 G、H 上给出了所有的函数和结构的具体讲述所在的页码。为帮助那些以随意顺序阅读各主题的读者,全文提供了大量的对相关主题的参考点。

尽管套接口 API 已成为网络编程的既成事实标准,XTI API 也仍在使用,有时是用在非 TCP/IP 的协议族上。虽然第 4 部分讨论 XTI 的篇幅比第 2 部分和第 3 部分讨论套接口的篇幅小得多,但是在套接口 API 讨论中所叙述的概念同样也适用于 XTI API。例如,不论使用哪一个 API(套接口或 XTI),有关非阻塞 I/O、广播、多播、信号驱动 I/O、带外数据和线程的使用都有相同的概念。其实,许多网络编程问题基本类似,与程序是使用套接口 API 还是使用 XTI API 编写的无关,很少有使用这个 API 能干而使用那个 API 却干不了的事。一句话,概念相同,差别仅在函数名和参数而已。

源码和勘误表获取

本书中出现的所有例子的源码都可从我的主页获取,其 URL 地址列在本前言的末尾。学习网络编程的最好方法是使用这些程序,然后修改并改进它们。只有真正编写这种形式的代码,才能加深对概念的理解并提高编程技巧。各章最后提供了大量的习题,附录 E 给出了

其中大多数的解答。

本书最新的勘误表也可从我的网页获取。[①]

W. Richard Stevens
rstevens@kohala.com
http://www.kohala.com/~rstevens
1997 年 9 月于亚利桑那州 Tucson

① 译者注：中译本已根据最后修改日期为 1999 年 1 月 31 日的最新勘误表(打印在宽行打印纸上共有 12 页)作过订正。

目 录

前言 …… (1)

第 1 部分 简介和 TCP/IP

第 1 章 简介 …… (1)

1.1 概述 …… (1)
1.2 一个简单的时间/日期客户程序 …… (4)
1.3 协议无关性 …… (8)
1.4 错误处理:包裹函数 …… (9)
1.5 一个简单的时间/日期服务器程序 …… (10)
1.6 书中客户-服务器程序例子索引表 …… (13)
1.7 OSI 模型 …… (15)
1.8 BSD 网络支持历史 …… (16)
1.9 测试用网络及主机 …… (16)
1.10 Unix 标准 …… (20)
1.11 64 位体系结构 …… (22)
1.12 小结 …… (23)
1.13 习题 …… (24)

第 2 章 传输层:TCP 和 UDP …… (25)

2.1 概述 …… (25)
2.2 总图 …… (25)
2.3 UDP:用户数据报协议 …… (27)
2.4 TCP:传输控制协议 …… (28)
2.5 TCP 连接的建立和终止 …… (29)
2.6 TIME_WAIT 状态 …… (35)
2.7 端口号 …… (36)
2.8 TCP 端口号与并发服务器 …… (38)
2.9 缓冲区大小及限制 …… (39)
2.10 标准因特网服务 …… (44)
2.11 常见因特网应用程序的协议使用 …… (45)
2.12 小结 …… (46)
2.13 习题 …… (46)

第2部分 基本套接口编程

第3章 套接口编程简介 …… (47)

3.1 概述 …… (47)
3.2 套接口地址结构 …… (47)
3.3 值-结果参数 …… (52)
3.4 字节排序函数 …… (54)
3.5 字节操纵函数 …… (57)
3.6 inet_aton、inet_addr 和 inet_ntoa 函数 …… (58)
3.7 inet_pton 和 inet_ntop 函数 …… (59)
3.8 sock_ntop 和相关函数 …… (62)
3.9 readn、writen 和 readline 函数 …… (64)
3.10 isfdtype 函数 …… (67)
3.11 小结 …… (68)
3.12 习题 …… (69)

第4章 基本 TCP 套接口编程 …… (70)

4.1 概述 …… (70)
4.2 socket 函数 …… (70)
4.3 connect 函数 …… (73)
4.4 bind 函数 …… (75)
4.5 listen 函数 …… (77)
4.6 accept 函数 …… (83)
4.7 fork 和 exec 函数 …… (85)
4.8 并发服务器 …… (87)
4.9 close 函数 …… (89)
4.10 getsockname 和 getpeername 函数 …… (90)
4.11 小结 …… (92)
4.12 习题 …… (92)

第5章 TCP 客户-服务器程序例子 …… (93)

5.1 概述 …… (93)
5.2 TCP 回射服务器程序:main 函数 …… (94)
5.3 TCP 回射服务器程序:str_echo 函数 …… (95)
5.4 TCP 回射客户程序:main 函数 …… (95)
5.5 TCP 回射客户程序:str_cli 函数 …… (96)
5.6 正常启动 …… (97)
5.7 正常终止 …… (98)
5.8 Posix 信号处理 …… (99)
5.9 处理 SIGCHLD 信号 …… (102)
5.10 wait 和 waitpid 函数 …… (105)

5.11 accept 返回前连接夭折 …… (108)
5.12 服务器进程终止 …… (110)
5.13 SIGPIPE 信号 …… (111)
5.14 服务器主机崩溃 …… (113)
5.15 服务器主机崩溃后重启 …… (113)
5.16 服务器主机关机 …… (114)
5.17 TCP 程序例子小结 …… (114)
5.18 数据格式 …… (116)
5.19 小结 …… (118)
5.20 习题 …… (119)

第 6 章 I/O 复用:select 和 poll 函数 …… (121)

6.1 概述 …… (121)
6.2 I/O 模型 …… (121)
6.3 select 函数 …… (126)
6.4 str_cli 函数(修订版) …… (131)
6.5 批量输入 …… (133)
6.6 shutdown 函数 …… (135)
6.7 str_cli 函数(再修订版) …… (137)
6.8 TCP 回射服务器程序(修订版) …… (138)
6.9 pselect 函数 …… (143)
6.10 poll 函数 …… (144)
6.11 TCP 回射服务器程序(再修订版) …… (146)
6.12 小结 …… (149)
6.13 习题 …… (149)

第 7 章 套接口选项 …… (151)

7.1 概述 …… (151)
7.2 getsockopt 和 setsockopt 函数 …… (151)
7.3 检查选项是否受支持并获取缺省值 …… (153)
7.4 套接口状态 …… (156)
7.5 基本套接口选项 …… (156)
7.6 IPv4 套接口选项 …… (169)
7.7 ICMPv6 套接口选项 …… (170)
7.8 IPv6 套接口选项 …… (171)
7.9 TCP 套接口选项 …… (172)
7.10 fcntl 函数 …… (175)
7.11 小结 …… (178)
7.12 习题 …… (178)

第 8 章 基本 UDP 套接口编程 …… (180)

8.1 概述 …… (180)

8.2 recvfrom 和 sendto 函数 …… (180)
8.3 UDP 回射服务器程序:main 函数 …… (182)
8.4 UDP 回射服务器程序:dg_echo 函数 …… (182)
8.5 UDP 回射客户程序:main 函数 …… (184)
8.6 UDP 回射客户程序:dg_cli 函数 …… (185)
8.7 数据报的丢失 …… (185)
8.8 验证接收到的响应 …… (186)
8.9 服务器进程未运行 …… (188)
8.10 UDP 程序例子小结 …… (189)
8.11 UDP 的 connect 函数 …… (191)
8.12 dg_cli 函数(修订版) …… (194)
8.13 UDP 缺乏流量控制 …… (195)
8.14 UDP 中外出接口的确定 …… (199)
8.15 使用 select 函数的 TCP 和 UDP 回射服务器程序 …… (200)
8.16 小结 …… (202)
8.17 习题 …… (202)

第 9 章 基本名字与地址转换 …… (204)

9.1 概述 …… (204)
9.2 域名系统 …… (204)
9.3 gethostbyname 函数 …… (207)
9.4 RES_USE_INET6 解析器选项 …… (211)
9.5 gethostbyname2 函数与 IPv6 支持 …… (212)
9.6 gethostbyaddr 函数 …… (214)
9.7 uname 函数 …… (215)
9.8 gethostname 函数 …… (216)
9.9 getservbyname 和 getservbyport 函数 …… (216)
9.10 其他网络相关信息 …… (219)
9.11 小结 …… (220)
9.12 习题 …… (221)

第 3 部分 高级套接口编程

第 10 章 IPv4 和 IPv6 的互操作性 …… (222)

10.1 概述 …… (222)
10.2 IPv4 客户与 IPv6 服务器 …… (222)
10.3 IPv6 客户与 IPv4 服务器 …… (226)
10.4 IPv6 地址测试宏 …… (227)
10.5 IPV6_ADDRFORM 套接口选项 …… (228)
10.6 源代码可移植性 …… (230)
10.7 小结 …… (230)
10.8 习题 …… (231)

第 11 章 高级名字与地址转换 …… (232)

11.1 概述 …… (232)
11.2 getaddrinfo 函数 …… (232)
11.3 gai_strerror 函数 …… (236)
11.4 freeaddrinfo 函数 …… (237)
11.5 getaddrinfo 函数:IPv6 和 UNIX 域 …… (238)
11.6 getaddrinfo 函数:例子 …… (240)
11.7 host_serv 函数 …… (241)
11.8 tcp_connect 函数 …… (242)
11.9 tcp_listen 函数 …… (245)
11.10 udp_client 函数 …… (250)
11.11 udp_connect 函数 …… (252)
11.12 udp_server 函数 …… (253)
11.13 getnameinfo 函数 …… (255)
11.14 可重入函数 …… (256)
11.15 gethostbyname_r 和 gethostbyaddr_r 函数 …… (259)
11.16 getaddrinfo 和 getnameinfo 函数的实现 …… (261)
11.17 小结 …… (281)
11.18 习题 …… (282)

第 12 章 守护进程和 inetd 超级服务器 …… (283)

12.1 概述 …… (283)
12.2 syslogd 守护进程 …… (284)
12.3 syslog 函数 …… (284)
12.4 daemon_init 函数 …… (287)
12.5 inetd 守护进程 …… (290)
12.6 daemon_inetd 函数 …… (294)
12.7 小结 …… (296)
12.8 习题 …… (297)

第 13 章 高级 I/O 函数 …… (298)

13.1 概述 …… (298)
13.2 套接口超时 …… (298)
13.3 recv 和 send 函数 …… (302)
13.4 readv 和 writev 函数 …… (304)
13.5 recvmsg 和 sendmsg 函数 …… (305)
13.6 辅助数据 …… (309)
13.7 排队的数据量 …… (312)
13.8 套接口与标准 I/O …… (312)
13.9 T/TCP:事务 TCP …… (315)
13.10 小结 …… (317)

13.11 习题 …… (317)

第 14 章 Unix 域协议 …… (318)

14.1 概述 …… (318)
14.2 Unix 域套接口地址结构 …… (318)
14.3 socketpair 函数 …… (321)
14.4 套接口函数 …… (321)
14.5 Unix 域字节流客户-服务器程序 …… (322)
14.6 Unix 域数据报客户-服务器程序 …… (323)
14.7 描述字传递 …… (325)
14.8 接收发送者的凭证 …… (332)
14.9 小结 …… (336)
14.10 习题 …… (336)

第 15 章 非阻塞 I/O …… (338)

15.1 概述 …… (338)
15.2 非阻塞读和写:str_cli 函数(修订版) …… (339)
15.3 非阻塞 connect …… (348)
15.4 非阻塞 connect:日期/时间客户程序 …… (349)
15.5 非阻塞 connect:Web 客户程序 …… (352)
15.6 非阻塞 accept …… (360)
15.7 小结 …… (361)
15.8 习题 …… (362)

第 16 章 ioctl 操作 …… (363)

16.1 概述 …… (363)
16.2 ioctl 函数 …… (363)
16.3 套接口操作 …… (364)
16.4 文件操作 …… (365)
16.5 接口配置 …… (366)
16.6 get_ifi_info 函数 …… (367)
16.7 接口操作 …… (375)
16.8 ARP 高速缓存操作 …… (376)
16.9 路由表操作 …… (378)
16.10 小结 …… (378)
16.11 习题 …… (379)

第 17 章 路由套接口 …… (380)

17.1 概述 …… (380)
17.2 数据链路套接口地址结构 …… (380)
17.3 读和写 …… (381)
17.4 sysctl 操作 …… (388)

17.5 get_ifi_info 函数 …… (392)
17.6 接口名和索引函数 …… (395)
17.7 小结 …… (399)
17.8 习题 …… (400)

第 18 章 广播 …… (401)

18.1 概述 …… (401)
18.2 广播地址 …… (402)
18.3 单播和广播的比较 …… (403)
18.4 使用广播的 dg_cli 函数 …… (406)
18.5 竞争状态 …… (409)
18.6 小结 …… (416)
18.7 习题 …… (416)

第 19 章 多播 …… (417)

19.1 概述 …… (417)
19.2 多播地址 …… (417)
19.3 局域网上多播和广播的比较 …… (420)
19.4 广域网上的多播 …… (422)
19.5 多播套接口选项 …… (424)
19.6 mcast_join 和相关函数 …… (427)
19.7 使用多播的 dg_cli 函数 …… (430)
19.8 接收 MBone 会话声明 …… (431)
19.9 发送和接收 …… (434)
19.10 SNTP:简单网络时间协议 …… (436)
19.11 SNTP(续) …… (440)
19.12 小结 …… (451)
19.13 习题 …… (452)

第 20 章 高级 UDP 套接口编程 …… (454)

20.1 概述 …… (454)
20.2 接收标志、目的 IP 地址和接口索引 …… (454)
20.3 数据报截断 …… (461)
20.4 何时使用 UDP 而不是 TCP …… (461)
20.5 给 UDP 应用程序增加可靠性 …… (463)
20.6 捆绑接口地址 …… (472)
20.7 并发 UDP 服务器 …… (476)
20.8 IPv6 分组信息 …… (478)
20.9 小结 …… (480)
20.10 习题 …… (481)

第 21 章 带外数据 …… (482)

21.1 概述 …… (482)

21.2 TCP 带外数据 …… (482)
21.3 sockatmark 函数 …… (488)
21.4 TCP 带外数据小结 …… (494)
21.5 客户-服务器心博函数 …… (495)
21.6 小结 …… (499)
21.7 习题 …… (499)

第 22 章 信号驱动 I/O …… (501)

22.1 概述 …… (501)
22.2 套接口上的信号驱动 I/O …… (501)
22.3 使用 SIGIO 的 UDP 回射服务器程序 …… (503)
22.4 小结 …… (508)
22.5 习题 …… (509)

第 23 章 线程 …… (510)

23.1 概述 …… (510)
23.2 基本线程函数:创建和终止 …… (511)
23.3 使用线程的 str_cli 函数 …… (513)
23.4 使用线程的 TCP 回射服务器程序 …… (515)
23.5 线程特定数据 …… (519)
23.6 Web 客户与同时连接 …… (526)
23.7 互斥锁 …… (529)
23.8 条件变量 …… (532)
23.9 Web 客户与同时连接(续) …… (535)
23.10 小结 …… (537)
23.11 习题 …… (537)

第 24 章 IP 选项 …… (539)

24.1 概述 …… (539)
24.2 IPv4 选项 …… (539)
24.3 IP 源路径选项 …… (540)
24.4 IPv6 扩展头部 …… (547)
24.5 IPv6 步跳选项和目的选项 …… (548)
24.6 IPv6 路由头部 …… (551)
24.7 IPv6 粘附选项 …… (554)
24.8 小结 …… (555)
24.9 习题 …… (555)

第 25 章 原始套接口 …… (557)

25.1 概述 …… (557)
25.2 原始套接口创建 …… (557)
25.3 原始套接口输出 …… (558)

25.4 原始套接口输入 …… (559)
25.5 Ping 程序 …… (561)
25.6 Traceroute 程序 …… (572)
25.7 一个 ICMP 消息守护进程 …… (583)
25.8 小结 …… (597)
25.9 习题 …… (597)

第 26 章 数据链路访问 …… (599)

26.1 概述 …… (599)
26.2 BPF:BSD 分组过滤器 …… (599)
26.3 DLPI:数据链路提供者接口 …… (601)
26.4 Linux:SOCK_PACKET …… (602)
26.5 libpcap:分组捕获函数库 …… (603)
26.6 检查 UDP 的校验和字段 …… (603)
26.7 小结 …… (619)
26.8 习题 …… (619)

第 27 章 客户-服务器程序其他设计方法 …… (620)

27.1 概述 …… (620)
27.2 TCP 客户程序其他设计方法 …… (622)
27.3 TCP 测试用客户程序 …… (623)
27.4 TCP 迭代服务器程序 …… (624)
27.5 TCP 并发服务器程序,每个客户一个子进程 …… (624)
27.6 TCP 预先派生子进程服务器程序,accept 无上锁保护 …… (627)
27.7 TCP 预先派生子进程服务器程序,accept 使用文件锁保护 …… (633)
27.8 TCP 预先派生子进程服务器程序,accept 使用线程互斥锁保护 …… (636)
27.9 TCP 预先派生子进程服务器程序,传递描述字 …… (637)
27.10 TCP 并发服务器程序,每个客户一个线程 …… (642)
27.11 TCP 预先创建线程服务器程序,每个线程各自 accept …… (643)
27.12 TCP 预先创建线程服务器程序,主线程统一 accept …… (645)
27.13 小结 …… (648)
27.14 习题 …… (649)

第 4 部分 XTI:X/Open 传输接口编程

第 28 章 XTI:TCP 客户程序 …… (650)

28.1 概述 …… (650)
28.2 t_open 函数 …… (651)
28.3 t_error 和 t_strerror 函数 …… (654)
28.4 netbuf 结构和 XTI 结构 …… (655)
28.5 t_bind 函数 …… (656)
28.6 t_connect 函数 …… (658)

28.7 t_rcv 和 t_snd 函数 …… (658)
28.8 t_look 函数 …… (660)
28.9 t_sndrel 和 t_rcvrel 函数 …… (661)
28.10 t_snddis 和 t_rcvdis 函数 …… (662)
28.11 XTI TCP 时间/日期客户程序 …… (663)
28.12 xti_rdwr 函数 …… (666)
28.13 小结 …… (667)
28.14 习题 …… (667)

第 29 章 XTI:名字与地址函数 …… (668)

29.1 概述 …… (668)
29.2 /etc/netconfig 文件与 netconfig 函数 …… (668)
29.3 NETPATH 环境变量与 netpath 函数 …… (670)
29.4 netdir 函数 …… (671)
29.5 t_alloc 和 t_free 函数 …… (673)
29.6 t_getprotaddr 函数 …… (675)
29.7 xti_ntop 函数 …… (675)
29.8 tcp_connect 函数 …… (676)
29.9 小结 …… (680)
29.10 习题 …… (680)

第 30 章 XTI:TCP 服务器程序 …… (681)

30.1 概述 …… (681)
30.2 t_listen 函数 …… (682)
30.3 tcp_listen 函数 …… (683)
30.4 t_accept 函数 …… (685)
30.5 xti_accept 函数 …… (686)
30.6 简单的时间/日期服务器程序 …… (687)
30.7 多个待处理连接 …… (689)
30.8 xti_accept 函数(修订版) …… (691)
30.9 小结 …… (698)
30.10 习题 …… (698)

第 31 章 XTI:UDP 客户和服务器程序 …… (700)

31.1 概述 …… (700)
31.2 t_rcvudata 和 t_sndudata 函数 …… (700)
31.3 udp_client 函数 …… (700)
31.4 t_rcvuderr 函数:异步错误 …… (704)
31.5 udp_server 函数 …… (706)
31.6 分片读取数据报 …… (708)
31.7 小结 …… (710)

第 32 章 XTI 选项 …… (711)

32.1 概述 …… (711)
32.2 t_opthdr 结构 …… (713)
32.3 XTI 选项 …… (714)
32.4 t_optmgmt 函数 …… (717)
32.5 检查选项是否受支持并获取缺省值 …… (718)
32.6 获取和设置 XTI 选项 …… (721)
32.7 小结 …… (724)

第 33 章 流 …… (725)

33.1 概述 …… (725)
33.2 概貌 …… (725)
33.3 getmsg 和 putmsg 函数 …… (729)
33.4 getpmsg 和 putpmsg 函数 …… (730)
33.5 ioctl 函数 …… (731)
33.6 TPI:传输提供者接口 …… (731)
33.7 小结 …… (740)
33.8 习题 …… (740)

第 34 章 XTI:其他函数 …… (741)

34.1 概述 …… (741)
34.2 非阻塞 I/O …… (741)
34.3 t_rcvconnect 函数 …… (741)
34.4 t_getinfo 函数 …… (742)
34.5 t_getstate 函数 …… (742)
34.6 t_sync 函数 …… (743)
34.7 t_unbind 函数 …… (745)
34.8 t_rcvv 和 t_rcvvudata 函数 …… (745)
34.9 t_sndv 和 t_sndvudata 函数 …… (746)
34.10 t_rcvreldata 和 t_sndreldata 函数 …… (746)
34.11 信号驱动 I/O …… (747)
34.12 带外数据 …… (748)
34.13 回馈传输提供者 …… (752)
34.14 小结 …… (753)

第 5 部分 附 录

附录 A IPv4、IPv6、ICMPv4 和 ICMPv6 …… (754)

A.1 概述 …… (754)
A.2 IPv4 头部 …… (754)

A.3 IPv6 头部 …… (755)
A.4 IPv4 地址 …… (758)
A.5 IPv6 地址 …… (762)
A.6 ICMPv4 和 ICMPv6:网际控制消息协议 …… (766)

附录 B 虚拟网络 …… (769)

B.1 概述 …… (769)
B.2 MBone …… (769)
B.3 6bone …… (771)

附录 C 调试技术 …… (773)

C.1 系统调用跟踪 …… (773)
C.2 标准因特网服务 …… (777)
C.3 sock 程序 …… (778)
C.4 小测试程序 …… (780)
C.5 tcpdump 程序 …… (782)
C.6 netstat 程序 …… (782)
C.7 lsof 程序 …… (783)

附录 D 杂凑的源代码 …… (784)

D.1 unp.h 头文件 …… (784)
D.2 config.h 头文件 …… (787)
D.3 unpxti.h 头文件 …… (789)
D.4 标准错误处理函数 …… (791)

附录 E 部分习题解答 …… (794)

附录 F 参考文献 …… (828)

附录 G 函数和宏定义索引表 …… (837)

附录 H 结构定义索引表 …… (840)

附录 I 中英文对照词汇表 …… (842)

第1部分 简介和TCP/IP

第1章 简 介

1.1 概 述

大多数网络应用系统可分成两部分：客户(client)和服务器(server)。[①] 图1.1画出了这两部分及它们之间的通信链路。

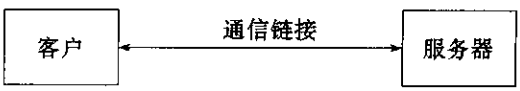


图1.1 网络应用系统：客户和服务器

大多数读者都已熟悉很多客户与服务器的实例：如Web浏览器(客户)和Web服务器的通信；FTP客户从FTP服务器获取文件；Telnet客户通过远程主机上的Telnet服务器提供远程登录手段。

通常客户一次只与一个服务器通信，不过以使用Web浏览器为例，我们也许在10分钟内就可以与许多不同的Web服务器通信。从服务器的角度来看，一个服务器同时与多个客户通信并不稀奇，见图1.2。以后我们将介绍几种让一个服务器同时处理多个客户请求的方法。

客户与服务器之间的通信涉及网络通信协议。本书的焦点是TCP/IP协议族，也称为网际协议族。例如Web客户与服务器之间的通信就使用TCP协议。而TCP又使用IP协议，IP则使用某种形式的数据链路层通信。举例来说，如果客户与服务器处于同一个以太网，我们就有图1.3的通信层次。

虽然客户与服务器之间的通信使用应用协议，传输层内的通信使用TCP协议，如此等等，但实际上，客户与服务器间的信息流在某一端是向下通过协议栈，而在另一端则是向上通过协议栈。

---

① 译者注：本书通篇频繁使用客户(client)和服务器(server)这两个术语。实际上它们的具体含义随上下文而不同，有时指静态的源程序或可执行程序(客户程序和服务器程序)，有时指动态进程(客户进程和服务器进程)，有时又指运行进程的主机(客户主机和服务器主机)。在不引起混淆的前提下，我们简单地称客户进程为客户，称服务器进程为服务器。同样应用(application)这个术语的具体含义也随上下文而变化，有时指程序(应用程序)，有时指进程(应用进程)，有时作为名词性修饰词译为应用。本书有时把同处应用层的客户和服务器对也用应用表示，我们称之为网络应用系统。程序(program)与进程(process)是在系统调用exec上衔接的。exec既可以由shell隐式调用(直接输入命令行执行程序属于这种情况)，也可以在用户程序中显式调用。我们称exec调用后运行中的程序为进程，它继承exec调用者的进程ID，而这个调用进程本身又可能是此前某个fork系统调用的结果。程序和进程的密切关系使得两者有时相互渗透使用，不易区分。

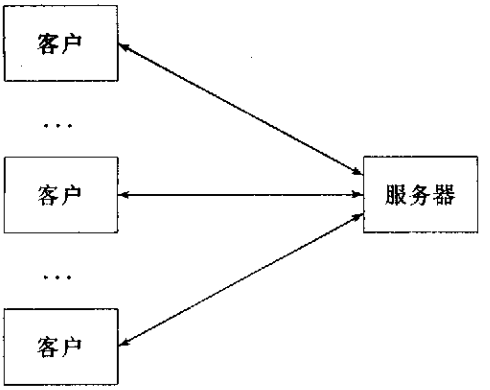


图 1.2 一个服务器同时处理多个客户的请求

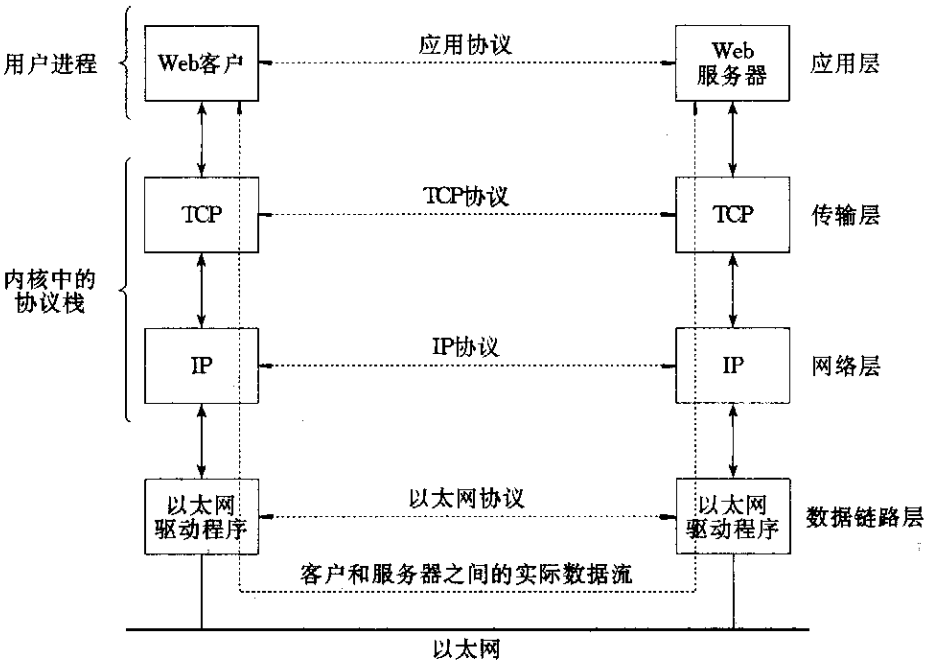


图 1.3 客户与服务器使用 TCP 协议在同一个以太网中通信

值得注意的是,客户与服务器是典型的用户进程,而 TCP 和 IP 协议则通常是系统内核协议栈的一部分。我们在图 1.3 右边标出了四层协议。

本书不只阐述 TCP 和 IP 协议,有些客户和服务器还使用 UDP 协议替代 TCP,第 2 章将详细介绍这两种协议。我们一直说的"IP"协议早在 20 世纪 80 年代早期就已使用,但它的正式名称是 IP 版本 4(IP version 4,即 IPv4)。IP 版本 6(IP version 6,即 IPv6)在 90 年代中期发展起来,将来可能取代 IPv4。本书编写时 IPv6 的最初实现已能获取,因此文中讨论的网络应用程序的开发不仅使用 IPv4,而且使用 IPv6。附录 A 比较了 IPv4 和 IPv6,介绍了 IPv6 最新的补充内容以及正文中会碰到的其他协议。

客户与服务器无需如图 1.3 所示处于同一个局域网。相反,图 1.4 给出了处于不同局域网中的客户与服务器,这两个局域网是使用路由器(router)连接到广域网的。

路由器是广域网的架构设备。今天，最大的广域网是因特网(Internet)。不过很多公司使用自己的广域网，而这些私有的广域网既可以连接也可以不连接到因特网。② 本章以下部分将概述多个主题，它们在以后章节中还要详细介绍。我们从一个完整使用 TCP 协议的简单客户程序开始，它用到了贯穿全书的许多函数调用和概念。这个客户程序仅在 IPv4 上运行，不过我们会指出在 IPv6 上需要修改的地方。较好的方法是编写独立于协议的客户和服务器程序，这在第 11 章中讨论。本章还介绍一个完整的 TCP 服务器程序，可与上述客户程序一起工作。

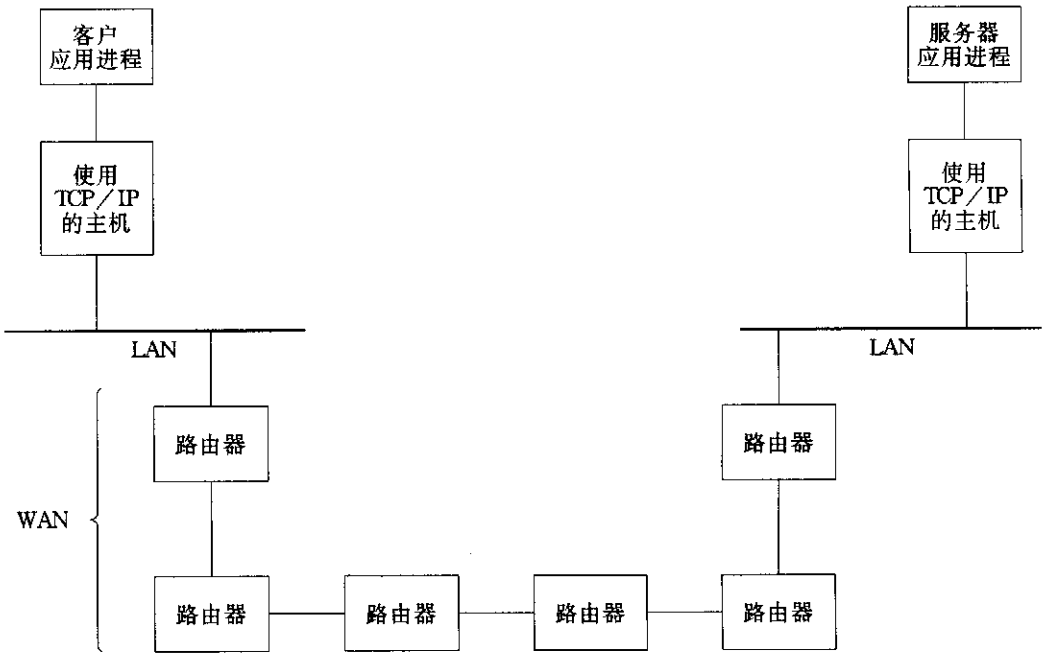


图 1.4 处于不同局域网的客户和服务器主机通过广域网连接

为了简化所有代码，对于本书中要调用的大部分系统函数，我们都定义了包裹函数，它们用于检查错误、输出适当消息及当出错时终止程序运行。我们还给出了本书在后面的许多例子要用到的测试网络、主机、路由器以及主机名、IP 地址和操作系统。

当今介绍的 Unix 时经常使用 Posix 一词，它是一种为多数厂商采用的标准。我们将介绍 Posix 的历史以及它对 API 的影响，再介绍标准化领域的其他主角。

② 译者注： internet 一词有多种含义。一是因特网(the Internet)，它是个专有名词，特指从 ARPANET 发展来的联结全世界的大型互联网。二是互联网(internets)，凡是采用 TCP/IP 协议的网络都可称为互联网，因特网就是互联网之一。三是作为名词性修饰词，这时应根据情况分别译成“因特网”、“互联网”或“网际”。例如“Internet Protocol”译成“网际协议”(注意：“Internet Protocol”是“internet protocol”名词专有化的结果)，“Internet Society”译成“因特网学会”。

1.2 一个简单的时间/日期客户程序

让我们考虑一个特殊的例子，目的是引入在本书中将要遇到的许多概念和说法。图1.5是TCP时间/日期客户程序的一个实现。这个客户建立与服务器的TCP连接并读取服务器送回的当前时间和日期(直观可读格式)。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     sockfd, n;
6     char    recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     if (argc != 2)
9         err_quit("usage: a.out <IPaddress>");
10    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
11        err_sys("socket error");
12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sin_family = AF_INET;
14    servaddr.sin_port = htons(13);  /* daytime server */
15    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16        err_quit("inet_pton error for %s", argv[1]);
17    if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18        err_sys("connect error");
19    while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20        recvline[n] = 0;    /* null terminate */
21        if(fputs(recvline, stdout) == EOF)
22            err_sys("fputs error");
23    }
24    if (n < 0)
25        err_sys("read error");
26    exit(0);
27 }
```

图1.5 TCP时间/日期客户程序[intro/daytimetcpcli.c]

这就是本书所有源代码使用的格式。每一非空行都被编号，代码的正文描述部分的左边标有起始与结束的行号。有的段落开始处含有一醒目的简短标题，概述本段代码的内容。

图标题右侧括号中的intro/daytimetcpcli.c表示源代码所在的目录为intro，文件为daytimetcpcli.c。本书所有源代码都可免费获得，见前言部分。编译、运行和修改本书中的程序是学习网络编程概念的好方法。

全文使用缩进的插入式注解(如此处所示)来说明实现的细节和历史上的观点。

如果我们编译这一程序，生成缺省文件a.out并执行它，将有下述的结果：

```
Solaris% a.out 206.62.226.35        我们的输入
Fri Jan 12 14:27:52 1996            程序的输出
```

任何时候，交互时的输入采用黑体字，计算机的输出用白体字。注解用楷体字加在右边。shell 提示带有系统名字(本例中为 solaris)，说明在哪台主机上运行命令。图 1.16 给出本书大多数例子所运行的系统，它们的主机名本身通常就说明各自的操作系统。

在图 1.16 的 27 行网络程序里有许多地方值得考虑，这里我们只做简单介绍，目的是让初次遇到网络程序的读者有所准备，本书以后将对这些主题作更详细的说明。

头文件

第 1 行　包含一个头文件 unp.h，见 D.1 节。这个头文件含有许多大部分网络程序都需要的系统头文件，并定义了所用到的各种常值(如 MAXLINE)。③

命令行参数

第 2～3 行　这是 main 函数的定义，其形式参数是命令行参数。本书例子中的代码全部用 ANSI(美国国家标准局) C 编译器。

创建 TCP 套接口

第 10～11 行　socket 函数创建网际(AF_INET)字节流(SOCK_STREAM)套接口，它是 TCP 套接口的特色名字。该函数返回一小整数描述字，在以后的其他函数调用中(如随后的 connect 和 read 调用)，我们就用它来标识这个套接口。

if 语句包含对 socket 函数的调用，并把返回值赋给变量 sockfd，然后测试该值是否小于 0。可把该语句分为两个 C 语句：

```
sockfd = socket(AF_INET,SOCK_STREAM,0);
if (socfd < 0)
```

不过把这两行合并成一行是常见的 C 语句用法。按照 C 语言的优先规则(小于运算符的优先级高于赋值运算符)，函数调用和赋值语句必须用括号括起。作者第一次看到这个样式是在 Minix 源代码中[Tenenbaum 1987]，之后一直沿用。后面的 while 语句也使用相同的样式。

③ 译者注：严格地说，C 语言中用 #define 伪命令定义的对象称为常数，用 const 限定词定义并初始化的对象称为常量(相对于变量而言)。常数的值在编译时就确定，常量的值则在运行时初始化后才确定(但此后它只能作为右值使用)。本书绝大多数恒定值是用 #define 定义的常数。不过“常数”的称谓容易让人狭义地的理解成仅仅是数而已，因此本书统一用“常值”指代其值恒定不变的对象。

以后我们将遇到术语套接口(socket)的许多不同用法。④ 首先,我们正在使用的应用程序编程接口称为套接口API(sockets API)。上一段中名为socket的函数就是套接口API的一部分。上一段中我们还提到了"TCP套接口",它是"TCP端点(TCP endpoint)"的同义词。

如果socket函数调用失败,我们就调用自己的err_sys放弃执行。该函数输出我们提供的出错消息和所发生系统错误的描述信息(如"Protocol not supported(协议不受支持)"等)后,终止进程的执行。这个函数和带有err_前缀的少数其他函数的调用将贯穿全书,见D.4节。

指定服务器IP地址和端口

第12～16行 我们把服务器的IP地址和端口号填入网际套接口地址结构(名为servaddr的sockaddr_in结构变量)。调用bzero把整个结构清零,置地址族为AF_INET,端口号为13(这是daytime服务器的众所周知端口,支持这一服务的任何TCP/IP主机都一样,见图2.13),置IP地址为命令行的第一个参数值(argv[1])。IP地址和端口成员必须具有合适格式:我们调用库函数htons("主机到网络短整数")去变换二进制端口号,调用库函数inet_pton("表达到数值")去变换ASCII命令行参数(例如206.62.226.35)到合适的格式。

> bzero不是ANSI C的函数,它起源于Berkeley网络代码。不过,我们在整本书中始终用它代替ANSI C的memset函数,因为bzero(两个参数)比menset(三个参数)更便于记忆。几乎所有支持套接口API的厂商也提供bzero,如果没有这一函数,可用unp.h头文件里提供的这一函数的宏定义代替。
>
> 在TCPv3第一次印刷本中,作者犯了在10多处对换memset函数第二和第三个参数的错误。C编译器找不出这一错误,因为两个参数是同种类型。(其实,第二个参数是int类型,而第三个参数是size_t,为典型的unsigned int类型,但是分别指定其值为0和16对于另一个参数的类型也适用。)对memset的这些调用仍然工作,但不做任何事,因为待初始化的字节数被指定成0。程序之所以仍能工作是因为只有很少的套接口函数要求网际套接口地址结构的最后8个字节置0,不过,它确实是错误的。使用bzero函数可以避免这种错误,因为如果使用函数原型,C编译器总能发现bzero的两个参数是否对换了。
>
> inet_pton函数是一个支持IPv6的新函数,见附录A的详细说明。老代码使用inet_addr函数把ASCII点分十进制数串变换成正确的格式,但它有许多限制,而inet_pton改正了这些缺陷。如果你的系统尚未支持该函数,那你可用3.7节提供的它的实现。

---

④ 译者注:socket一词应译成"套接口",其理由如下。首先,作为网络编程API之一的套接口(sockets,注意这种用法时总是用复数形式,即使用作名词性修饰词也这样,如sockets API、sockets library等)跟XTI一样,是应用层到传输层的接口。其次,具体使用中的套接口是与Unix管道的一端类似的东西,它是具体的,我们既可以往这个"口"写数据,也可以从这个"口"读数据。最后,套接口函数使用套接口描述字访问具体的套接口,如果把套接口描述字的简称sockfd译成"套接字"到比较合适,从这个意义上看,一个套接口可对应多个套接字,因为Unix的描述字既可以复制,也可以继承;反过来,一个套接字对应且只对应一个套接口。

建立与服务器的连接

第17～18行 connect函数用在TCP套接口上的功能是:跟由它的第二个参数所指的套接口地址结构对应的服务器建立连接。connect的第三个参数说明套接口地址结构的长度,对于网际套接口地址结构,我们总是使用C语言的操作符sizeof由编译器来计算。

SA在头文件unp.h中定义为struct sockaddr,即通用套接口地址结构。套接口函数每次需要一个指向套接口地址结构的指针时,这个指针必须转换成指向一个通用套接口地址结构的指针类型。这是因为套接口函数早于ANSI C标准,因此在20世纪80年代早期开发这些函数时,void * 指针类型还不可用。问题是串"struct sockaddr"有15字符长,往往造成源代码行超出屏幕的右边,因此我们把它缩成SA。通用套接口地址结构在介绍图3.3时将详细讨论。

读入并输出服务器的应答

第19～25行 我们使用read函数读服务器的应答,并用标准的I/O函数fputs输出结果。[5]我们必须注意,TCP是一种无记录边界的字节流协议。本例中,服务器的应答是具有下面格式的26字节字符串。

```
Fri Jan 12 14:27:52 1996 \r\n
```

其中,\r\n是ASCII字符集的回车和换行符。在字节流的协议里,这个26字节的数据可以有多种返回方式:既可以是包含所有26个字节的单个TCP分节,也可以是每个分节只含一个字节的总共26个分节,总共26字节的任何其他组合也可以。[6] 通常服务器返回包含所有26个字节的单个分节,但是如果数据量很大,我们不能确保一次read调用就返回服务器的应答。因此从TCP套接口读数据时,总是把read编码在某个循环中,当read返回0(远程端关闭连接)或负值(错误发生)时才终止循环。

本例中,记录的结束由服务器关闭连接表示。这种技术也适用于HTTP(Hypertext

---

⑤ 译者注:为求简洁明确,以后尽可能采用直接把函数名和C语言关键字用作动词的译法。例如本句的这种译法是:"我们read服务器的应答,并fputs结果。"又如"如果connect成功,就break出循环。"的意思是:"如果connect函数调用成功(表示连接成功),就执行C语言的break语句跳出循环。"

⑥ 译者注: 计算机网络各层对等实体间交换的单位信息称为协议数据单元(PDU),分节就是PDU之一,它对应于TCP传输层。由于本书涉及PDU种类较多,为避免混淆,我们在这里汇总简要说明。就TCP/IP协议族而言,应用层实体(如普通的客户和服务器进程)间交换的PDU称为数据(data,TCP应用进程)或记录(record,UDP应用进程),其中数据的大小没有限制,但记录不能超过UDP发送缓冲区大小(这个缓冲区实际上并不存在,但它具备大小这个属性)。传输层实体间交换的PDU是分节(segment,TCP协议)或数据报(datagram,UDP协议),它们的大小都是有限的。TCP应用进程的数据由TCP划分成块(chunk)后封装在分节中传送,UDP应用进程的记录则由UDP整个封装到数据报中传送。网络层实体之间交换的PDU是分组(packet,俗称包),其大小自然有限。传输层的分节或数据报都由IP封装在分组中传送。有时分组可能太大,超过了数据链路层单个PDU的容量,这时分组需要划分成若干个片段(fragment),因此片段也是网络层实体之间交换的信息单元。TCP/IP为提高效率,一般尽可能避免这种分片操作。数据链路层实体之间交换的PDU是帧(frame),网络层的每个小分组(即无需划分片段的分组)或片段由数据链路层封装到一个帧中。由于UDP应用进程、UDP和IP都是无连接的,本书有时统一用数据报表示记录、数据报或分组,读者需加以留意。每层的PDU除用来封装来自紧邻上层的数据单元(称为服务数据单元SDU)外,也用于本层内部的协议通信。例如全书频繁使用的SYN和FIN分节就是TCP专门用来建立与拆除连接的PDU,通常不再携带来自应用进程的SDU。另外书中讨论的MSS是应用层与传输层之间的接口属性,MTU则是网络层与数据链路层之间的接口属性。

Transfer Protocol)协议。其他技术也有效,如FTP(File Transfer Protocol)协议和SMTP(Simple Mail Transfer Protocol)协议都使用由ASCII回车和换行符构成的2字节序列终止记录。Sun RPC(Remote Procedure Call)和DNS(Domain Name System)使用TCP时,在每个记录的前面放一个二进制计数值,由它给出记录的长度。这里有个重要的概念:TCP本身不提供记录结束标记。应用程序需要确定记录边界的话,就必须自己去实现,有一些常用的方法可供选择。

终止程序

第26行 exit语句终止程序。Unix在进程终止时关闭所有描述字,其中包括TCP套接口描述字。

本书将对刚叙述过的所有概念作深入的探讨。

1.3 协议无关性

图1.5程序是与IPv4相关的。分配并初始化sockaddr_in结构,置sin_family成员为AF_INET,指定socket函数的第一个参数为AF_INET。

为使图1.5程序可在IPv6上运行,我们必须修改程序代码。图1.6是一个可以在IPv6上运行的版本,黑体字表示修改的部分。

```
1 #include       "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5         int       sockfd,n;
6         char      recvline[MAXLINE + 1];
7         struct sockaddr_in6 servaddr;
8     if(argc ! = 2)
9         err_quit("usage:a.out <IPaddress>")
10    if( (sockfd = socket(AF_INET6,SOCK_STREAM,0)) < 0)
11        err_sys("socket error");
12    bzero(&servaddr,sizeof(servaddr));
13    servaddr.sin6_family = AF_INET6;
14    servaddr.sin6_port = htons(13);     /* daytime server */
15    if(inet_pton(AF_INET6,argv[1],&servaddr.sin6_addr) <= 0)
16        err_quit("inet_pton error for %s",argv[1]);
17    if(connect(sockfd,(SA * ) &servaddr,sizeof(servaddr)) < 0) {
18        err_sys("connect error");
19    while ( (n = read(sockfd,recvline,MAXLINE)) > 0) {
20        recvline[n] = 0;      /* null terminate */
21        if(fputs(recvline,stdout) == EOF)
22            err_sys("fputs error");
23    }
24    if (n < 0)
25        err_sys("read error");
26    exit(0);
27 }
```

图1.6 适合于IPv6的图1.5程序修改版[intro/daytimetcpcliv6.c]

上述程序仅需修改5行代码，但这次是一个与IPv6相关的程序。图11.7是一个与协议无关的同一客户程序，它使用getaddrinfo函数(由tcp_connect函数调用)。

本程序的另一个不足之处是用户必须以点分十进制数键入服务器的IP地址(如适合于IPv4的206.62.226.35)。人们更习惯于用名字代替数字(如laptop.kohala.com或只用laptop)，我们将在第9章和第11章阐述主机名与IP地址，服务名与端口号的转换函数。在那两章之前，我们不讨论这些函数，继续采用IP地址和端口号，目的是了解我们必须填写和查看的套接口地址结构的细节，同时又可以避免被其他函数集的细节把网络编程的讨论搞复杂了。

1.4 错误处理：包裹函数

任何现实世界的程序都必须检查每个函数调用是否返回错误。在图1.5程序中，我们检查socket、inet_pton、connect和fputs函数是否返回错误，当发生错误时我们再调用err_quit或err_sys输出出错消息并终止程序。我们发现绝大多数情况下这正是所希望的。个别情况下当返回错误时，我们需要做某些特殊处理，而不是终止程序。图5.12就是这样的例子，我们必须检查系统调用是否被中断。

因为多数情况下程序终止于一个错误，我们可以定义包裹函数来简化我们的程序。包裹函数调用实际函数，检查返回值，发生错误时终止进程。确定包裹函数名的约定是大写实际函数名的第一个字符，如：

```
sockfd=Socket(AF_INET,SOCK_STREAM,0);
```

图1.7是socket的包裹函数。

```
172 int
173 Socket(int family,int type,int protocol)
174 {
175     int     n;
176     if( (n = socket(family,type,protocol)) < 0)
177         err_sys("socket error");
178     return (n);
179 }
```

图1.7 socket函数的包裹函数[lib/wrapsock.c]

每当你遇到一个以大写字母打头的函数名时，它就是我们定义的包裹函数。它调用的实际函数的名字与包裹函数名相同，但以对应的小写字母打头。

这些包裹函数也许不见得如何节省代码量，不过当我们在23章讨论线程时，我们会发现线程函数遇到错误时并不设置标准Unix的errno变量，而是把error的值作为函数返回值返回调用者。这意味着每次我们调用pthread函数，我们必须分配一个变量来存放返回值，再用它来设置errno变量后才能调用err_sys。为避免使用花括号，我们使用C的“逗号”操作符，把errno的赋值与err_sys的调用连成单一语句，如下所示：

```
int n;
if ((n = pthread_mutex_lock (&ndone_mutex)) ! =0)
    errno=n , err_sys("pthread mutex_lock error");
```

虽然我们可以定义一个新的错误处理函数，它取系统错误号为它的一个参数，但是通过定义包裹函数，我们可以很容易地实现如下形式的代码：

```
Pthread_mutex_lock(&ndone_mutex);
```

这个包裹函数的定义如图1.8所示。

```
72 void
73 Pthread_mutex_lock(pthread_mutex_t *mptr)
74 {
75     int     n
76     if ( (n = pthread_mutex_lock(mptr)) == 0)
77         return;
78     errno = n;
79     err_sys("pthread_mutex_lock error");
80 }
```

图1.8 我们的pthread_mutex_lock的包裹函数[lib/wrappthread.c]

仔细推敲编码，我们可以用宏替代函数，从而稍微提高运行效率，不过包裹函数很少是程序性能的瓶颈所在。

选择大写函数名的第一个字母是一种较折衷的方法。还有很多其他方法：如用e做为函数名的前缀（如[Kernighan and Pike 1984]第184页所示）或用_e做为函数名的后缀等等。同样提供调用其他函数的可视指示，我们的这种方法看来是最少分散人们的注意力的。这种技术还有助于检查那些其错误返回值通常忽略的函数，如close和listen。

本书后面的例子我们将普遍使用包裹函数，除非我们必须检查某个确定的错误并处理它，而不是终止进程运行。我们并不给出所有包裹函数的源代码，但它们的代码是免费可得的（见前言）。

Unix errno值

每当在一个Unix函数（如socket函数）中发生错误时，全局变量errno将被置成一个指示错误类型的正数，函数本身则通常返回-1。err_sys检查errno变量并输出其相应的出错消息（例如，当errno值等于ETIMEDOUT时，将输出“Connection timed out（连接超时）”）。

errno的值只在函数发生错误时设置。如果函数不返回错误，errno的值就无定义。在Unix中，所有的错误值都是常值，具有以E打头的全大写字母名字，在头文件<sys/errno.h>中定义。值0不表示任何错误。

把errno值存于全局变量不适合共享所有全局变量的多线程。我们将在23章中讲述解决这一问题的方法。

1.5 一个简单的时间/日期服务器程序

我们可以写一个简单的TCP时间/日期服务器程序，它可以和1.2节的客户程序一道工作。图1.9是这个服务器程序，它使用了上述的包裹函数。

```
1 #include    "unp.h"
2 #include    <time.h>
```

```
3 int
4 main(int argc,char * * argv)
5 {
6       int         listenfd,connfd;
7       struct sockaddr_in servaddr;
8       char        buff[MAXLINE];
9       time_t      ticks;
10      Listenfd = Socket(AF_INET,SOCK_STREAM,0);
11      bzero(&servaddr,sizeof(servaddr));
12      servaddr.sin_family = AF_INET;
13      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14      servaddr.sin_port = htons(13);      /* daytime server */
15      Bind(listenfd,(SA * ) &servaddr,sizeof(servaddr));
16      Listen(listenfd,LISTENQ);
17      for( ; ;) {
18          connfd = Accept(listenfd,(SA * )NULL,NULL);
19          ticks = time(NULL);
20          snprintf(buff,sizeof(buff),"%.24s\r\n",ctime(&ticks));
21          Write(connfd,buff,strlen(buff));
22          Close(connfd);
23      }
24 }
```

图 1.9 TCP 时间/日期服务器程序[intro/daytimetcpsrv.c]

创建 TCP 套接口

第 10 行 TCP 套接口的创建与客户程序相同。

捆绑服务器的众所周知端口到套接口

第 11～15 行 通过设置网际套接口地址结构并调用 bind 函数，我们把服务器的众所周知端口捆绑到套接口。我们指定的 IP 地址为 INADDR_ANY，它允许服务器在任意接口上接受客户连接(假定服务器主机有多个接口)。以后我们将了解怎样限定服务器在某个给定接口上接受客户连接。

把套接口变换成监听套接口

第 16 行 通过调用 listen 函数将此套接口变换成一个监听套接口，它使系统内核接受来自客户的连接。socket、bind 和 listen 是任何 TCP 服务器用于准备所谓的监听描述字(listening descriptor，本例为 listenfd)通常的三个步骤。

常值 LISTENQ 在头文件 unp.h 中定义，它指定系统内核允许在这个监听描述字上排队的最大客户连接数。4.5 节将详细说明客户连接的排队。

接受客户连接，发送应答

第 17～21 行 一般情况下，服务器进程在调用 accept 函数后处于睡眠状态，它等待客户的连接和内核对它的接受。TCP 连接使用三路握手(three－way handshake)来建立，当握手完毕时，accept 函数返回，其返回值是一个称为已连接描述字(connected descriptor)的新描述字(connfd)。此描述字用于与新客户的通信。accept 为每个连接到服务器的客户返回一个新的已连接描述字。

本书全文所用的无限循环采用以下风格：

```
for ( ; ; ) {
    . . .
}
```

库函数time返回当前时间和日期，这是自Unix纪元，即1970年1月1日0点0分0秒(国际标准时间)以来的秒数。下一个函数ctime将此整数值变换成直观可读的时间格式，如：

```
Fri Jan 12 14:27:52 1996
```

snprintf函数在这个字符串末端添加了回车和换行两个字符。write函数把结果发送给客户。

这也许是你第一次碰到snprintf函数。许多现有代码调用的是sprintf，但sprintf不检查目标缓冲区是否溢出，相反，snprintf要求其第二个参数是目标缓冲区的大小，因此可确保缓冲区不溢出。snprintf不是标准ANSI C的一部分，但这个标准的修订版C9X正在考虑。不过，许多厂商提供的标准C库含有这个函数。我们在本书里使用snprintf，如果你的系统不提供这个函数，可用我们提供的通过调用sprintf实现的版本。

值得注意的是，许多网络入侵是由黑客通过发送数据，导致服务器对sprintf的调用溢出其缓冲区而发生的。你必须小心使用的函数还有gets、strcat和strcpy，通常应调用fgets、strncat和strncpy函数代替。[Garfinkel and Spafford 1996]的第23章阐述了编写安全的网络程序。

终止连接

第22行 服务器通过调用close关闭与客户的连接。它引发通常的TCP连接终止序列：每个方向上发送一个FIN，每个FIN又由对方确认。2.5节将详细描述三路握手和用于终止TCP连接的四个TCP分节。

如同上节介绍的客户程序，我们仅仅简要地介绍了服务器程序，监听细节则留待本书后面论述。注意以下事项：

- 如同客户程序，本服务器程序与IPv4相关。图11.9的服务器程序版本与协议无关，它调用了getaddrinfo函数。
- 本服务器程序一次只能处理一个客户，如果多个客户连接同时到达，系统内核在最大数目的限制下把它们排入队列，然后每次返回一个给accept函数。这个服务器只需调用两个库函数time和ctime，其速度很快。但是，如果服务器用很多时间处理各个客户(如几秒或一分钟)，我们就必须以某种方式重叠对各个客户的服务。图1.9服务器称为迭代服务器(iterative server)，因为对每个客户它都迭代执行一次。并发服务器(concurrent server)程序有多种编写技术，它同时能处理多个客户。最简单的技术是调用Unix的fork函数(见4.7节)，为每个客户派生一个子进程。其他技术包括使用线程代替fork(见23.4节)，或在服务器启动时预先fork一定数量的子进程(见27.6节)。
- 如果在shell命令行下启动如本例的服务器，我们也许要它运行很长的时间，因为服务器通常在系统启动后就一直运行。这要求我们增加服务器程序的代码，使它作为

Unix 守护进程(daemon,即能在后台运行,不跟任何终端关联的进程)运行,见 12.4 节。

1.6 书中客户-服务器程序例子索引表

两个贯穿全书的客户-服务器程序例子阐述了网络编程所用的各种技术。

- 时间/日期客户-服务器程序例子(开始于图 1.5、1.6 和 1.9)。
- 回射客户-服务器程序例子(开始于第 5 章)。

为了提供本书所涵盖的不同主题的索引,以下 4 个表汇总了我们开发的程序及它们的源代码所在的图号。图 1.10 列出了本书开发的时间/日期客户程序的不同版本,其中的两个我们已叙述过。图 1.11 列出了时间/日期服务器程序的不同版本。图 1.12 列出了回射客户程序的不同版本,图 1.13 列出了回射服务器程序的不同版本。

图	页码	描述
1.5	4	TCP/IPv4,协议相关
1.6	8	TCP/IPv6 协议相关
9.8	218	TCP/IPv4,协议相关,调用 gethostbyname 和 getservbyname
11.7	244	TCP,协议无关,调用 getaddrinfo 和 tcp_connect
11.12	252	UDP,协议无关,调用 getaddrinfo 和 udp_client
15.11	350	TCP,使用非阻塞 connect
28.13	664	TCP/IPv4,XTI,协议相关
29.7	679	TCP,XTI,协议无关,调用 netdir_getbyname 和 tcp_connect
31.3	703	UDP,XTI,协议无关,调用 netdir_getbyname 和 udp_client
31.4	705	UDP,XTI,协议无关,接收异步错
31.7	709	UDP,XTI,协议无关,分片读数据报
33.8	732	TCP,协议相关,用 TPI 代替套接口或 XTI
E.1	797	TCP,协议相关,生成 SIGPIPE
E.5	800	TCP,协议相关,输出套接口的接收缓冲区大小和 MSS
E.13	809	TCP,协议相关,允许主机名(gethostbyname)或者 IP 地址
E.14	810	TCP,协议无关,允许主机名(gethostbyname)

图 1.10 本书开发的时间/日期客户程序的不同版本

图	页码	描述
1.9	11	TCP/IPv4,协议相关
11.9	247	TCP,协议无关,调用 getaddrinfo 和 tcp_listen
11.10	249	TCP,协议无关,调用 getaddrinfo 和 tcp_listen
11.15	255	UDP,协议无关,调用 getaddrinfo 和 udp_server
12.5	290	TCP,协议无关,作为独自的守护进程运行
12.12	295	TCP,协议无关,从 inetd 守护进程派生
30.5	688	TCP,XTI,协议无关,调用 netdir_getbyname 和 tcp_listen
31.6	708	UDP,XTI,协议无关,调用 netdir_getbyname 和 udp_server

图 1.11 本书开发的时间/日期服务器程序的不同版本

图	页码	描述
5.4	96	TCP/IPv4,协议相关
6.9	133	TCP,使用 select
6.13	138	TCP,使用 select 在批处理模式下工作
8.7	184	UDP/IPv4,协议相关
8.9	187	UDP,核实服务器的地址
8.17	194	UDP,调用 connect 获取异步错
13.2	300	UDP,使用 SIGALRM 信号在读服务器的应答时启动超时
13.4	302	UDP,使用 select 函数在读服务器的应答时启动超时
13.5	303	UDP,使用 SO_RCVTIMEO 套接口选项在读服务器的应答时启动超时
14.4	324	Unix 域字节流,协议相关
14.6	325	Unix 域数据报,协议相关
15.3	341	TCP,使用非阻塞 I/O
15.9	347	TCP,使用两个进程(fork)
15.21	361	TCP,建立连接,然后发送 RST
18.5	407	UDP,具有竞争状态的广播
18.6	410	UDP,具有竞争状态的广播
18.7	412	UDP,通过使用 pselect 消除了竞争状态的广播
18.9	413	UDP,通过使用 sigsetjmp 和 siglongjmp 消除了竞争状态的广播
18.10	416	UDP,通过从信号处理程序使用 IPC 消除了竞争状态的广播
20.6	466	UDP,使用超时、重传和序列号实现可靠性
21.14	497	TCP,使用带外数据,对服务器心搏测试
23.2	514	TCP,使用两个线程
24.6	545	TCP/IPv4,指定源路径

图 1.12 本书开发的回射客户程序的不同版本

图	页码	描述
5.2	95	TCP/IPv4,协议相关
5.12	108	TCP/IPv4,协议相关,收拾终止了的子进程
6.21	141	TCP/IPv4,协议相关,使用 select,单个进程处理所有客户
6.25	147	TCP/IPv4,协议相关,使用 poll,单个进程处理所有客户
8.3	182	UDP/IPv4,协议相关
8.24	200	TCP 和 UDP/IPv4,协议相关,使用 select
13.14	313	TCP,使用标准 I/O 库
14.3	323	Unix 域字节流,协议相关
14.5	324	Unix 域数据报,协议相关
14.15	335	Unix 域字节流,从客户端传递凭证
20.4	459	UDP,接收目的地址和接收接口,截取数据报
20.15	474	UDP,捆绑所有接口地址
21.15	499	TCP,使用带外数据对客户心搏测试
22.4	505	UDP,使用信号驱动的 I/O
23.3	515	TCP,每个客户一个线程
23.4	517	TCP 每个客户一个线程,传递可移植的参数

（续）

图	页码	描述
24.6	545	TCP/IPv4，输出接收到的源路径
25.30	586	UDP，使用 icmpd 接收异步错
E.17	821	UDP，捆绑所有接口地址

图 1.13 本书开发的回射服务器程序的不同版本

1.7 OSI 模型

描述网络中各协议层的一般方法是国际标准化组织(ISO)的计算机通信开放系统互连(open systems interconnection，OSI)模型。这是一个七层模型，如图 1.14 所示，图中同时给出了与网际协议族的近似映射。

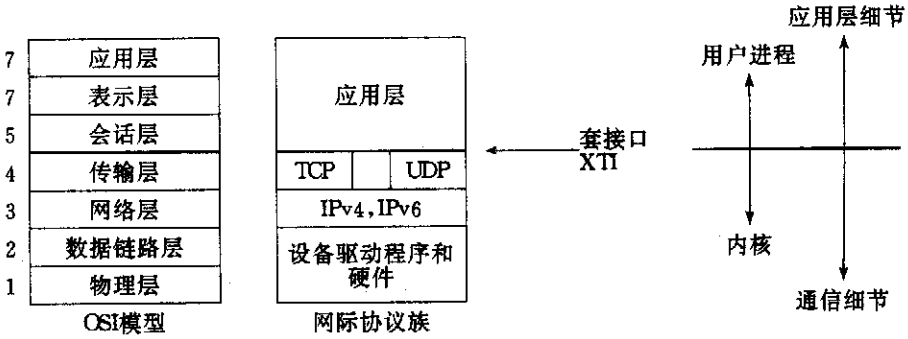


图 1.14 OSI 层和网际协议族

我们认为 OSI 模型的底下两层是随系统提供的设备驱动程序和网络硬件。除需知道数据链路的某些特性如 1500 字节的以太网 MTU(我们将在 2.9 节论述)外，我们不必关心这两层。

网络层由 IPv4 和 IPv6 协议处理，附录 A 描述这两个协议。传输层可以选择 TCP 或 UDP，我们将在第 2 章描述它们。图 1.14 中的网际协议族，在 TCP 与 UDP 之间留有一个间隙，指出应用程序可以绕过传输层而直接使用 IPv4 或 IPv6。这称为原始套接口(raw socket)，我们将在第 25 章阐述。

OSI 模型的上面三层合并成一层，称为应用层。这就是 Web 客户(浏览器)、Telnet 客户、Web 服务器、FTP 服务器或其他应用进程所在的层。对于网际协议，OSI 模型的上三层协议没什么区别。

本书所描述的两个编程接口(套接口和 XTI)是上三层(应用层)到传输层的接口。本书的焦点是：如何使用套接口或 XTI 编写使用 TCP 或 UDP 的网络应用程序。我们已提到原始套接口，在第 26 章我们将看到，甚至彻底绕过 IP 层读-写数据链路层的帧都是可能的。

为什么套接口和 XTI 都提供 OSI 模型上三层与传输层的接口？有两条理由，它们已标在图 1.14 的右侧。第一条理由是上三层处理应用程序(如，FTP，Telnet 或 HTTP)的细节，不大知道通信细节；下四层则不大知道应用程序，但能处理所有的通信细节：发送数据，等待

确认，给无序到达的数据排序，计算与验证校验和等等。第二条理由是上三层通常形成用户进程，而下四层通常作为操作系统内核的一部分提供。Unix与其他现代操作系统都提供分隔用户进程与内核的机制。因此，在第4层和第5层之间的接口很自然成了应用程序编程接口(API)。

1.8 BSD网络支持历史

套接口API起源于1983年发行的4.2BSD操作系统。图1.15描述了各种BSD发行版本的发展，主要注意TCP/IP的发展。当OSI协议于1990年进入BSD内核时，4.3BSD Reno发行版本对套接口API做了少量的改动。

图中从上到下是4.2BSD到4.4BSD的各种发行版本，它们源自Berkeley计算机系统研究组(CSRG)，不过要求获取者拥有Unix的源代码许可权。然而其中的所有网络支持代码，包括内核支持部分(如TCP/IP域协议栈、Unix域协议栈及套接口API)和应用程序(如Telnet和FTP的客户和服务器程序)都是独立于源自AT&T的Unix代码开发的。于是从1989年起，Berkeley开始提供了第一个BSD网络支持版本，它含有所有的网络支持代码及不受Unix源代码许可权约束的其他各种BSD系统软件。这些网络支持版本是可公开获取的，最终因特网上任何人都可通过匿名FTP获取。

源自Berkeley的最终版本是1994年的4.4BSD-Lite和1995年的4.4BSD-Lite2。这两个版本是其他系统的基础，包括BSD/OS、FreeBSD、NetBSD和OpenBSD，它们都还在开发和完善。有关各种BSD版本和各种Unix系统历史的详情见[Mckusick et al. 1996]的第1章。

很多Unix系统起源于某个版本的BSD网络支持代码(包括套接口API)，我们称这些实现为源自Berkeley的实现(Berkeley-derived implementations)。很多Unix商业版本基于系统V版本4(SVR4)，其中一些系统使用源自Berkeley的网络支持代码(如UnixWare 2.x)，而其他SVR4系统的网络支持代码则是独立开发的(如Solaris 2.x)。还有Linux系统：一种著名的免费可得的Unix实现版本，它不属于源自Berkeley的系列，其网络支持代码和套接口API是从头开始开发的。

1.9 测试用网络及主机

图1.16描述了本书例子所用的各种网络和主机。对每台主机我们都说明了它的操作系统和硬件类型(因为有些操作系统可在不同的硬件类型上运行)。各个框内的主机名整本书都用到。

顶部子网地址分别为206.62.226.32/27和206.62.226.64/27的两个以太网上的主机都在kohala.com域内。底部子网地址为140.252.1.0/24的以太网上的主机都在tuc.noao.edu域内，它们由国家光学天文台运行。符号/27和/24指出从地址的最左位开始的用于标识网络和子网的连续位数。图A.4和图A.5详细说明这两种IPv4地址，图A.4还说明当前使用的用于指定子网边界的/n记法。

图1.16中圆矩形框内的节点充当路由器，矩形框内的节点则是纯粹的主机。全书都使

用这个约定,因为有时候区分主机和路由器相当重要。

注意:Sun 操作系统的真实名字是 SunOS 5.x 而不是 Solaris 2.x,但大家习惯称它为 Solaris。

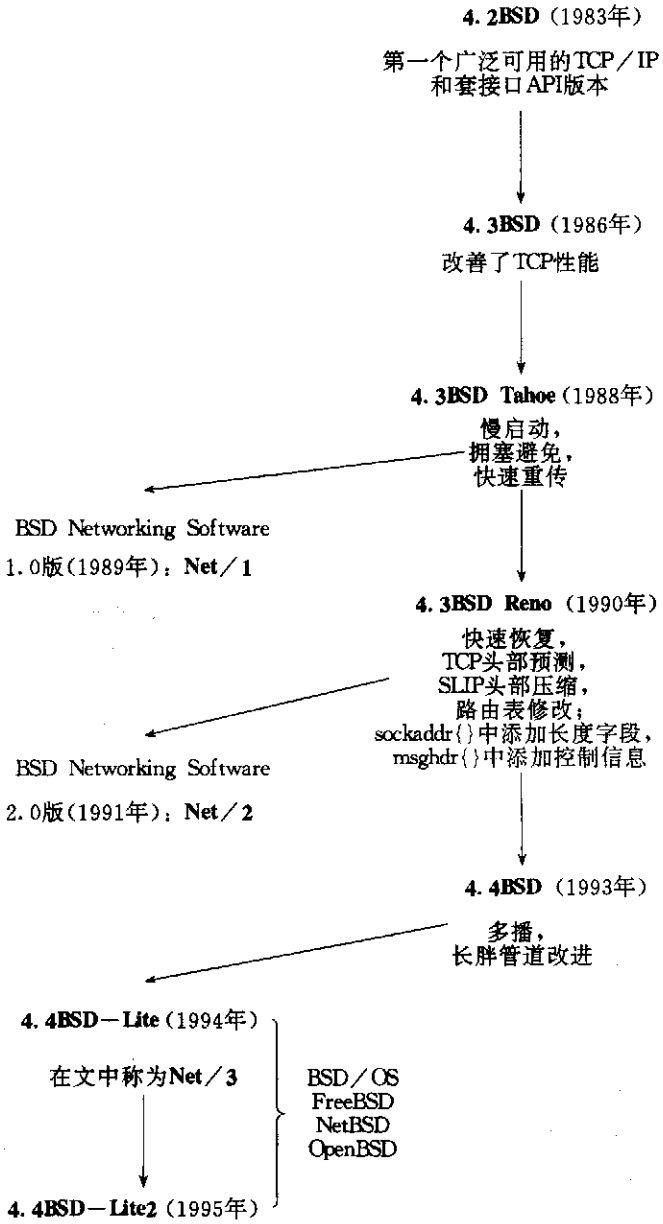


图 1.15 各种 BSD 版本的历史

网络拓扑发现

图 1.16 是全书的例子所用的网络拓扑,但是,你在自己的网络上运行这些例子和做习题时,你得知道自己的网络拓扑。虽然不存在有关网络配置和管理的标准,但是大多数 Unix 系统都有两个命令可用来了解网络细节:netstat 和 ifconfig。下面举一些不同于图 1.16 系统

的例子。通过阅读系统上这些命令的手册页面，你就可以获悉有关它们的输出信息的详情。⑦ 你还必须注意，有些厂商把这些命令存放于/sbin或/usr/sbin等管理目录，而不是普通目录/usr/bin，因此这些目录可能不在你通常的shell搜索路径(PATH)中。

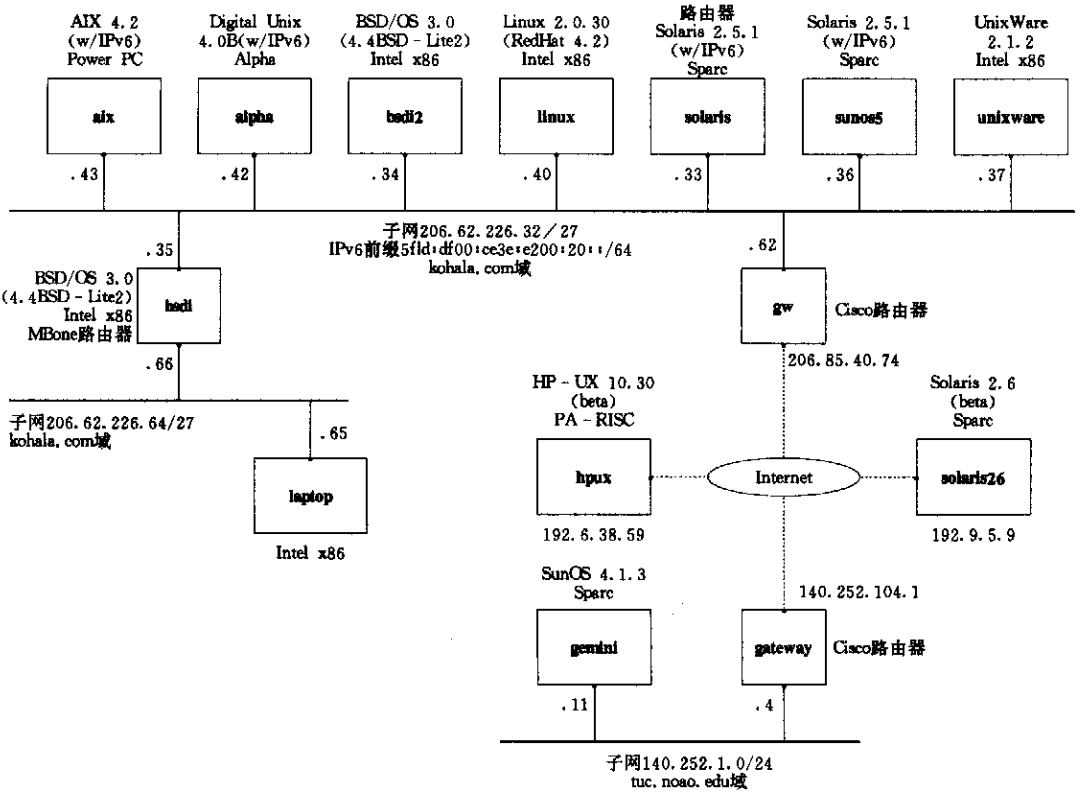


图1.16 本书中例子所用网络和主机

1. netstat -i提供接口信息，也可用标志-n输出数值地址而不是试图找出网络名字。下面例子显示接口和它们的名字。

```
linux % netstat -ni
Kernel Interface table
Iface MTU Met RX-OK RX-ERR RX-DRP RX-OVR TX-OK TX-ERR TX-DRP TX-OVR Flags
lo    3584  0     32      0      0      0     32      0      0      0 BLRU
eth0 1500   0 483929      0      0      0 449881      0      0      0 BRU
```

其中回馈(loopback)接口称为lo，以太网接口称为eth0。下面例子显示支持IPv6的一台主机的信息。

```
alpha % netstat -ni
Name  Mtu   Network     Address              Ipkts Ierrs Opkts Oerrs  Coll
ln0   1500  <Link>      08:00:2b:37:64:26    11220     0  4893     0     4
ln0   1500  DLI         none                 11220     0  4893     0     4
```

⑦ 译者注：手册页面(manual pages或man pages)是所有Unix系统都提供的使用man命令查看到的有关命令、函数和文件等的帮助信息。某个条目的手册页面就是以该条目为命令行参数执行man的输出。

```
ln0    1500  206.62.226.   206.62.226.42          11220   0   4893   0   4
ln0    1500  IPv6 FE80::800:2B37:6426             11220   0   4893   0   4
ln0    1500  IPv6 5F1B:DF00:CE3E:E200:20:800:2B37:6426  11220  0  4893  0  4
lo0    1536  <Link>       Link#3                  12432   0  12432   0   0
lo0    1536  127          127.0.0.1               12432   0  12432   0   0
lo0    1536  IPv6         ::1                     12432   0  12432   0   0
tun0   576   <Link>       Link#4                      0   0      0   0   0
tun0   576   IPv6         ::206.62.226.42             0   0      0   0   0
```

2. netstat -r 输出路由表，这是另一种确定接口的方法。一般使用-n 标志输出数值地址。本命令还输出缺省路由器的 IP 地址。

```
aix % netstat -rn
Routing tables
Destination          Gateway           Flags   Refs   Use    MTU    Netif Expire

Route tree for Protocol Family 2 (Internet):
default              206.62.226.62     UG      0      0      -      en0
127/8                127.0.0.1         U       0      0      -      lo0
206.62.226.32/27     206.62.226.43     U       4    475      -      en0
Route tree for Protocol Family 24 (Internet v6):
::/96                0.0.0.0           UC      0      0   1480      sit0   - =>
default      fe80::2:0:800:2078:e3e3   UG      0      0      -      en0
::1                  ::1               UH      0      0  16896      lo0
5f1b:df00:ce3e:e200:20::/80
                     link#2            UC      0      0   1500      en0    -
fe80::/16            link#2            UC      0      0   1500      en0    -
fe80::2:0:800:2078:e3e3
                     link#2            UHDL    1      0   1500      en0    -
ff01::/16            ::1               U       0      0      -      lo0
ff02::/16    fe80::800:5afc:2b36       U       1      3   1500      en0
ff11::/16            ::1               U       0      0      -      lo0
ff12::/16    fe80::800:5afc:2b36       U       0      0   1500      en0
```

（为对齐输出字段，我们将一些长行折了行。）

3. 给定接口名字，执行 ifconfig 可获得每个接口的详细信息。

```
linux % ifconfig eth0
eth0 link encap: 10Mbps Ethernet HWaddr 00:A0:24:9C:43:34
     inet addr:206.62.226.40 Bcast:206.62.226.63 Mask:255.255.255.224
     UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
     RX Packets:484461 errors:0 dropped:0 overruns:0
     TX Packets:450113 errors:0 dropped:0 overruns:0
     Interrupt:10 Base address:0x300
```

这里输出了 IP 地址、子网掩码和广播地址。MULTICAST 标志通常指示主机支持多播。

```
alpha % ifconfig ln0
ln0: flags=c63<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST,STMPLEX>
     inet 206.62.226.42 netmask fffffffe0 broadcast 206.62.226.63 ipmtu 1500
```

有些 ifconfig 的实现还提供-a 标志，用于输出所有已配置接口的信息。

4. 查找本地网络中多个主机 IP 地址的方法之一是针对广播地址执行 ping 命令。

```
bsdi % ping 206.62.226.63
```

```
PING 206.62.226.63 (206.62.226.63) : 56 data bytes
64 bytes from 206.62.226.35: icmp_seq=0 ttl=255 time=0.316 ms
64 bytes from 206.62.226.40: icmp_seq=0 ttl=64 time=1.369 ms (DUP!)
64 bytes from 206.62.226.34: icmp_seq=0 ttl=255 time=1.822 ms (DUP!)
64 bytes from 206.62.226.42: icmp_seq=0 ttl=64 time=2.27 ms (DUP!)
64 bytes from 206.62.226.37: icmp_seq=0 ttl=64 time=2.717 ms (DUP!)
64 bytes from 206.62.226.33: icmp_seq=0 ttl=255 time=3.281 ms (DUP!)
64 bytes from 206.62.226.62: icmp_seq=0 ttl=255 time=3.731 ms (DUP!)
^?                          敲入中断键(DEL)
--- 206.62.226.63 Ping statistics ---
1 packets transmitted, 1 packets received, +6 duplicates, 0% packet loss round-trip min/
avg/max = 0.316/2.215/3.731 ms
```

1.10 Unix标准

有关Unix标准化的大多数活动是由Posix和Open Group做的。

POSIX

Posix(Portable Operating System Interface)是"可移植操作系统接口"的首字母缩写。它并不是一个单一标准,而是由电气与电子工程师学会即IEEE开发的标准族。Posix也是被ISO和IEC(国际电工委员会)采纳的国际标准(这两个组织合称为ISO/IEC)。

第一个Posix标准是IEEE Std 1003.1—1988(共317页),它说明进入类Unix内核的C语言接口,涉及下述领域:进程原语(fork、exec、信号及定时器)、进程环境(用户ID、进程组)、文件与目录(所有I/O函数)、终端I/O、系统数据库(保密字文件和用户组文件),以及tar和cpio归档格式。

> 第一个Posix标准在1986年是称为"IEEEIX"的试用版,Posix的名字是由Richard Stallman建议使用的。

这个标准在1990年被更新成IEEE Std 1003.1—1990(共356页),它也是国际标准ISO/IEC 9945—1:1990。从1988版本到1990版本只做了少量的修改。新添的副标题为"Part 1: System Application Program Interface (API) [C Language]",指示本标准为C语言API。

下一个Posix标准是IEEE Std 1003.2—1992,它的副标题为"Part 2: Shell and Utilities"。它出版成两卷,共约1300页。这一部分定义了shell(基于系统V的Bourne Shell)和大约100个实用程序(从Shell启动执行的程序,包括awk、basename、vi和yacc等等)。本书称这个标准为Posix.2。

再下一个Posix标准是IEEE Std 1003.1b—1993,先前称IEEE P1003.4。这是对1003.1—1990标准的更新,添加了由P1003.4工作组开发的实时扩展。1003.1b-1993标准与1990标准相比新增的条目有:文件同步、异步I/O、信号量、存储管理(mmap和共享内存)、执行调度、时钟与定时器及消息队列。1003.1b—1993标准共590页。

再下一个Posix标准是IEEE Std 1003.1,1996年版[IEEE 1996],它包含1003.1—1990(基本API)、1003.1b—1993(实时扩展)、1003.1c—1995(pthreads)和1003.1i—1995(对1003.1b的技术性修正)。这个标准也称ISO/IEC 9945—1:1996,它加入了三章线程内容,共

743页。本书称这个标准为 Posix. 1。

> 743页中有四分之一多的篇幅是标题为“Rationale and Notes(原理与注解)”的附录。这些原理含有历史信息和某些特性必须加入或删除的理由,它们通常跟正式标准一样有教益。

这个标准的前言声称 ISO/IEC 9945由下面三个部分构成:

- 第1部分:系统应用程序接口(API)[C 语言]
- 第2部分:Shell 和实用程序
- 第3部分:系统管理(正在开发中)

最影响本书的 Posix 工作是 IEEE Std 1003. 1g:协议独立的接口(PII,Protocol Independent Interfaces),为 P1003. 1g 工作组的产品。这是网络 API 标准,定义有两个 API,称为 DNI(详细的网络接口,Detailed Network Interfaces)。

1. DNI/Socket,基于4. 4BSD 的套接口 API。
2. DNI/XTI,基于 X/Open XPG4规范。

这个标准的工作起始于80年代后期,由 P1003. 12工作组(以后改名为 P1003. 1g)承担,本书编写时它尚未完成(不过已接近)。草案6. 4(1996年5月)是第一个草案,它获得无记名投票75%以上的票数通过。草案6. 6(1997年3月)看来是最终草案[IEEE 1997a]。1998年或1999年的某时候,新版本 IEEE Std 1003. 1将被印刷出来,它将含有 P1003. 1g 标准。

即使 P1003. 1g 标准尚未正式完成,本书也尽可能使用来自这个标准的草案6. 6中的特性。本书称这一草案为 Posix. 1g。例如,connect 函数的第三个参数将被说明成 socklen_t 数据类型尽管它直到 Posix. 1g 才定义。类似地,我们叙述 Posix. 1g 中才有的 sockatmark 函数(21. 3节),并提供它的使用 ioctl 函数实现的版本。对 Unix 域的套接口,我们还使用 Posix. 1g 协议值 AF_LOCAL 代替 AF_UNIX。当前实际与 Posix. 1g 的差异在本书中将被注解出来。虽然今天还没有厂商支持 Posix. 1g(因为它不是最终版本),但是当这个标准完成时,厂商都会支持的。

Posix 标准化工作还将继续,这是任何试图涵盖它的书都在跟踪的目标。从 http://www. pasc. org/standing/sdll. html 可获得各种 Posix 标准的当前状况。

Open Group

Open Group 是由 X/Open 公司(1984年成立)和开放软件基金会(OSF,1988年成立)于1996年合并成的组织。它是厂商、工业界最终用户、政府和学术机构的国际组织。

X/Open 公司于1989年出版了“X/Open Portability Guide(X/Open 移植性指南)”第3期(XPG3)。第4期出版于1992年,其第2版出版于1994年。这个最终版本称为“Spec 1170”,其中魔数1170是系统接口(926个)、头文件(70个)和命令(174个)的数目总和。这个规范集的最终名字是“X/Open Single Unix Specification(X/Open 独立 Unix 规范)”,也称为“Unix 95”。

1997年3月独立 Unix 规范第2版发表,符合这个规范的产品称为“Unix 98”。本书就称这个规范为“Unix 98”。Unix 98的接口数目从1170个扩充到1434个,而用于工作站的数目则跳到3030个,因为它含有 CDE(公共桌面环境,Common Desktop Environment),它反过来又要

求有X Windows系统和Motif的用户接口。其详情见[Josey 1997]和http://www.opengroup.org/public/tech/unix/version2。

我们感兴趣的是Unix 98的网络服务部分，它们在[Open Group 1997]里定义，包括套接口API和XTI API。这个规范几乎与Posix.1g的草案6.6相同。

> 不幸的是，X/Open称它们的网络标准为XNS：X/Open Networking Services。定义Unix 98套接口和XTI的这个文档的版本称为"XNS Issue 5"。在网络界，XNS是Xerox Network Systems体系结构的缩写。我们避免使用XNS，就称这个X/Open文档为Unix 98网络API标准。

因特网工程任务攻坚组

IETF(因特网工程任务攻坚组，Internet Engineering Task Force)是一个由网络设计者、操作员、厂商和研究者联合组成的开放的国际团体，他们关心因特网体系结构的发展和它的顺利运作。它向任何有兴趣的个人开放。

因特网标准处理在RFC 2026[Bradner 1996]中说明。因特网标准一般处理协议事务而不是编程API。不过仍有两个RFC([Gilligan et al. 1997]和[Stevens and Thomas 1997])说明IP版本6的套接口API。它们是信息性RFC而非标准，它们产生的目的是为了加速部署由数家从事早期IPv6工作的厂商所生产的可移植应用程序。相反，标准化主体工作将花费更长时间。不过也许过一些时候，IPv6 API将被更为正式地标准化。

Unix版本和移植性

当今大多数Unix系统符合Posix.1和Posix.2的某个版本。我们这里用限定词"某个"是因为Posix也在更新(如，1993年对实时的扩充和1996年加入pthreads)，它要花费厂商一至两年时间去加上最近的更新。

历史上多数Unix系统或者源自Berkeley，或者源自系统V，不过这些差别在慢慢消失，因为大多数厂商已开始采用Posix标准。系统管理的差别还将存在，这个领域目前还没有Posix标准。

本书的焦点是即将到来的Posix.1g标准，其中又以套接口API为主。任何时候我们都尽可能使用Posix函数。

1.11 64位体系结构

90年代中期到末期倾向于64位体系结构和64位软件。理由之一是进程内部可使用更大的编址长度(即64位指针)，以允许进行大内存寻址(超过2^{32}字节)。已有的32位Unix系统上的一般编程模型称为ILP32模型，表示整数(I)、长整数(L)和指针(P)都占用32位；64位Unix系统上最为流行的编程模型称为LP64模型，它表示长整数(L)和指针(P)都占用64位。图1.17是这两种模型的比较。

数据类型	ILP32模型	LP64模型
char	8	8
short	16	16
int	32	32
long	32	64
pointer	32	64

图1.17 ILP32和LP64模型对不同数据类型所占用的位数的比较

从编程角度看,LP64模型意味着我们不能假设一个指针能存放到一个整数中。我们还必须考虑LP64模型对已有的API的影响。

ANSI C创造了size_t的数据类型,它用于诸如malloc的参数(分配的字节数)、read和write的第三个参数(读或写的字节数)等地方。在32位系统中size_t是32位值,但在64位系统中它必须是64位的值,以发挥大寻址模型的优点。这意味着64位系统中也许含有把size_t定义为unsignecl long的typedef语句。网络API的问题是Posix.1g的某些草案说明,存放套接口地址结构大小的函数参数具有size_t的数据类型(如bind和connect的第三个参数)。同样,某些XTI结构也含有数据类型为long的成员(如,t_info和t_opthdr结构)。当Unix系统从ILP32改变成LP64模型时,两者都将从32位的值变成64位的值。上述两例实际并不需要使用64位的数据类型:套接口地址结构的长度最多也不过是几百个字节,给XTI结构成员使用long数据类型也是错误的。

我们将见到的是创造新的数据类型来处理这些情况。套接口API使用socklen_t数据类型作为套接口地址结构的长度。XTI则使用t_scalar_t和t_uscalar_t数据类型。不把这些值由32位改为64位的理由是,这将使那些已在32位系统编译的应用程序的二进制代码兼容于新的64位系统。

1.12 小 结

图1.5虽然简单,但它说明了一个完整的TCP客户程序,它从一指定的服务器读取当前的时间和日期;而图1.9则给出这个服务器程序的一个完整版本。这两个例子引入了许多本书其他部分将要扩展的概念和术语。

我们的客户程序依赖于IPv4,为此我们把它修改成使用IPv6。但这给了我们另一个与协议相关的程序。第11章中我们将开发一些函数,它们可用来编写协议无关的代码,这在因特网开始使用IPv6时是非常重要。

纵贯本书,我们将使用1.4节的包裹函数以减少代码的长度,同时又保证测试每个函数调用,检查是否返回错误。我们的包裹函数都以一个大写字母开头。

IEEE Posix标准——Posix.1定义了Unix的基本C接口,Posix.2定义了标准命令,Posix.1g定义了网络API——是多数厂商倾向的标准。然而商业标准也正在迅速吸收并扩充Posix,著名的有Open Group的Unix标准,例如Unix 98。

那些对Unix网络支持历史感兴趣的读者可参阅叙述Unix历史的[Salus 1994]和叙述TCP/IP及因特网历史的[Salus 1995]。

1.13 习 题

1.1 按1.9节末尾的步骤去找出你自己的网络拓扑的信息。

1.2 获取本书例子的源代码(见前言),编译并测试图1.5的TCP时间/日期客户程序。运行这个程序若干次,每次以不同IP地址作为命令行参数。

1.3 在图1.5中,把socket的第一参数改为9999。编译并运行这个程序。结果如何?找出对应于所输出出错消息的errno值。你能找到关于这个错误的更详细信息吗?

1.4 修改图1.5的while循环,加入一个计数器,累计read返回大于零值的次数。在终止前输出这个值。编译并运行你的新客户程序。

1.5 按下述步骤修改图1.9程序。首先,把赋于sin_port的端口号从13改为9999。其次,把write的单一调用改为循环调用,每次写出结果字符串的一个字符。编译修改后的服务器程序并在后台启动执行。接下去修改前面习题的客户程序(它在终止前输出计数值),把赋于sin_port的端口号从13改为9999。启动客户,指定运行修改后服务器程序的主机的IP地址作为其命令行参数。客户程序计数器的输出值是多少?如果可能,在不同主机上运行客户与服务器。

第 2 章　传输层:TCP 和 UDP

2.1　概　述

本章提供本书例子所用 TCP/IP 协议的概貌。我们的目的是从网络编程角度提供足够的细节以理解如何使用这些协议,同时提供有关这些协议的实际设计、实现及历史的具体描述的参考点。

本章的焦点是传输层即 TCP 和 UDP 协议,绝大多数的客户-服务器应用程序都使用 TCP 或 UDP。这两个协议转而使用网络层协议 IP:IP 版本 4(IPv4)或 IP 版本 6(IPv6)。尽管可以绕过传输层直接使用 IPv4 或 IPv6,但这种技术(称为原始套接口)较少使用。因此,我们把 IPv4 和 IPv6 以及 ICMPv4 和 ICMPv6 的详细描述安排在附录 A。

UDP 是一种简单的,不可靠的数据报协议,而 TCP 是一种精致的、可靠的字节流协议。我们必须了解由这两个传输层提供给应用进程的服务,这样才能弄清这些协议处理什么,我们的应用进程又需要处理什么。

TCP 的某些特性一旦理解,我们就能更容易地编写健壮的客户和服务器程序,更容易地使用诸如 netstat 等工具来调试我们的客户和服务器程序。本章将阐述下述主题:TCP 的三路握手、TCP 连接终止序列、TCP 的 TIME_WAIT 状态、套接口层的 TCP 和 UDP 缓冲机制等等。

2.2　总　图

虽然称为"TCP/IP"协议族,但是本协议族还有许多其他成员。图 2.1 展示了这些协议的概貌。

在这个图中,我们展示了 IPv4 和 IPv6。从右向左观察这个图,最右边的 4 个应用程序使用 IPv6,这涉及到第 3 章中的 AF_INET6 常值和 sockaddr_in6 结构。另外的 5 个应用程序使用 IPv4。

最左边的应用程序(tcpdump)直接使用 BPF(BSD 分组过滤器)或 DLPI(数据链路提供者接口)同数据链路层进行通信。右边 9 个应用程序的下面用虚线标记出 API,它通常是套接口或 XTI。使用 BPF 或 DLPI 的接口不用套接口或 XTI。

> 这种情况的一种例外将在第 25 章详细描述:Linux 使用一种称为 SOCK_PACKET 的特殊套接口类型提供数据链路层的访问。

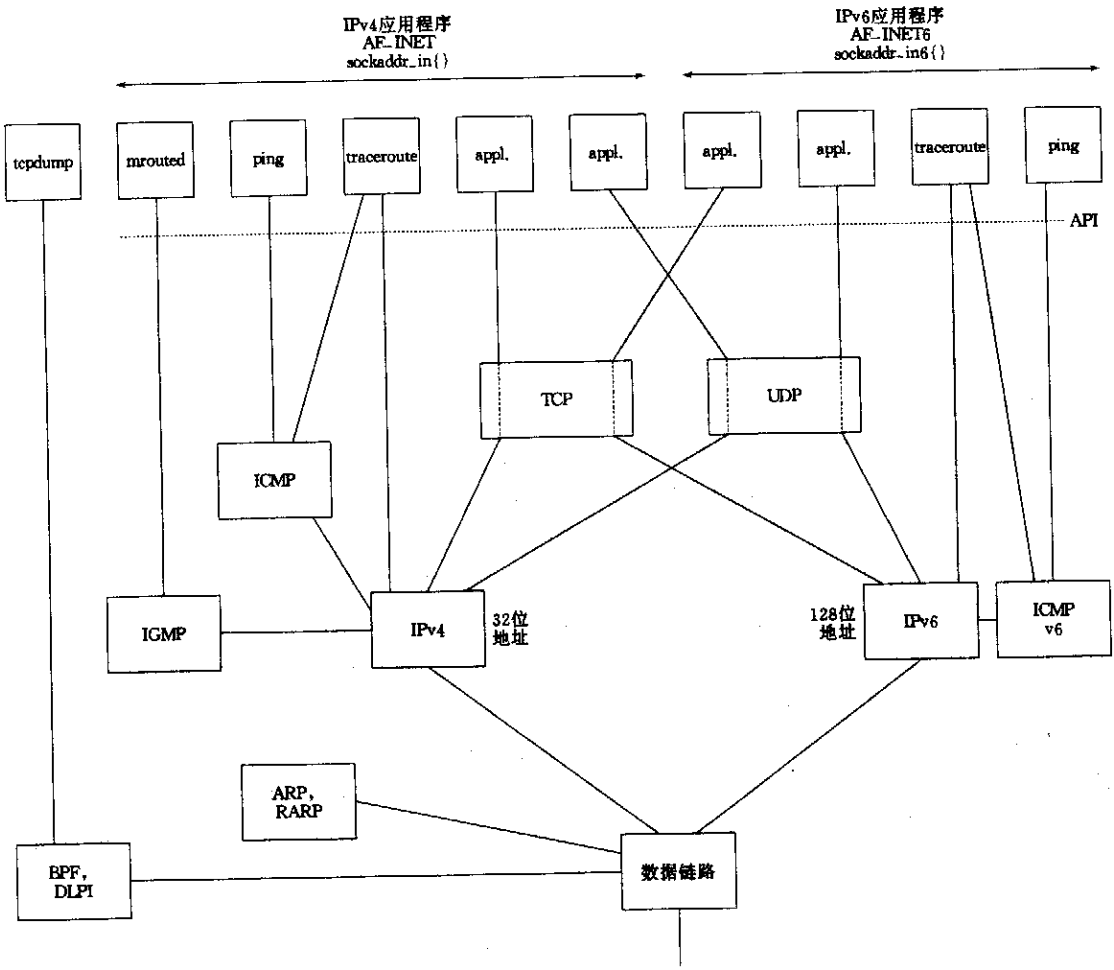


图 2.1 TCP/IP 协议概貌

在图 2.1 中,我们还注意到 traceroute 程序使用两种套接口:IP 套接口和 ICMP 套接口。在第 25 章我们将开发 IPv4 和 IPv6 两种版本的应用程序 ping 和 traceroute。

下面将阐述图 2.1 中的每一种协议。

IPv4 网际协议,版本 4。IPv4(我们通常就称之为 IP)自 80 年代早期以来一直是网际协议族的主力协议。它使用 32 位的地址(A.4 节)。IPv4 给 TCP、UDP、ICMP 和 IGMP 提供递送分组的服务。

IPv6 网际协议,版本 6。IPv6 在 90 年代中期设计出来,用以替代 IPv4。主要变化是使用 128 位的大地址(A.5 节)以处理 90 年代因特网爆发性的增长。IPv6 给 TCP、UDP 和 ICMPv6 提供递送分组的服务。

当无需区别 IPv4 和 IPv6 时,我们经常使用 IP 这个形容词,如 IP 层、IP 地址等等。

TCP	传输控制协议。TCP 是一种面向连接的协议。它给用户进程提供可靠的全双工的字节流。TCP 套接口是字节流套接口(stream socket)的一种。TCP 关心确认、超时和重传等具体细节。大多数因特网应用程序使用 TCP。注意，TCP 可以使用 IPv4 或 IPv6。
UDP	用户数据报协议。UDP 是一种无连接协议。UDP 套接口是数据报套接口(datagram socket)的一种。与 TCP 不同，UDP 不能保证每一 UDP 数据报可以到达目的地。与 TCP 相同的是，UDP 也可以使用 IPv4 或 IPv6。
ICMP	网际控制消息协议。ICMP 处理路由器和主机间的错误和控制消息。这些消息一般由 TCP/IP 网络软件自身(而不是用户进程)产生和处理，不过图中展示的 Ping 程序也使用 ICMP。有时我们称这个协议为 ICMPv4，用于区别 ICMPv6。
IGMP	网际组管理协议。IGMP 用于多播(第 19 章)，它在 IPv4 中是可选的。
ARP	地址解析协议。ARP 把 IPv4 地址映射到硬件地址(如以太网地址)。ARP 一般用于广播网络，如以太网、令牌环网和 FDDI，但不用于点对点网络。
RARP	反向地址解析协议。RARP 把硬件地址映射到 IPv4 地址。它有时用于无盘节点，如引导时的 X 终端。
ICMPv6	网际控制消息协议，版本 6。ICMPv6 综合了 ICMPv4、IGMP 和 ARP 的功能。
BPF	BSD 分组过滤器。它为进程提供访问链路层数据的接口。一般在源自 Berkeley 的内核中可以找到它。
DLPI	数据链路提供者接口。这是提供访问数据链路的接口，一般由 SVR4 提供。

所有网际协议由(多个)RFC(Request for Comments)定义。习题 2.1 的解答说明如何获得各个 RFC。

我们使用术语“IPv4/IPv6 主机”或“双栈主机”表示同时支持 IPv4 和 IPv6 的主机。

有关 TCP/IP 协议的其他细节见 TCPv1。TCP/IP 在 4.4BSD 中的实现见 TCPv2。

2.3　UDP：用户数据报协议

UDP 是一个简单的传输层协议，在 RFC768[Postel 1980]中有详细描述。应用进程写一数据报到 UDP 套接口，由它封装(encapsulating)成 IPv4 或 IPv6 数据报，然后发送到目的地。但是，UDP 并不能确保 UDP 数据报最终可到达目的地。

我们使用 UDP 进行网络编程所碰到的问题是缺乏可靠性。如果我们要确保一个数据报到达目的地，我们必须在应用程序里建立一大堆的特性：来自另一端的确认、超时、重传等等。

每个UDP数据报都有一定长度,我们可以认为一个数据报就是一个记录。如果数据报最终正确地到达目的地(即分组到达目的地且校验和正确),那么数据报的长度将传递给接收方的应用进程。我们已经提到TCP是一字节流协议,无记录边界(1.2节),这与UDP不同。

我们也称UDP提供无连接的(connectionless)服务,因为UDP客户与服务器不必存在长期的关系。例如,一个UDP客户可以创建一个套接口并发送一个数据报给一个服务器,然后立即用同一个套接口发送另一个数据报给另一个服务器。同样,一个UDP服务器可以用同一个UDP套接口从5个不同的客户一连串接收5个数据报。

2.4 TCP:传输控制协议

向应用进程提供的TCP服务与UDP服务不相同。(TCP在RFC793[Postel 1981c]中介绍。)首先,TCP提供客户与服务器的连接(connection)。一个TCP客户建立与一个给定服务器的连接,跨越连接与那个服务器交换数据,然后终止连接。

其次,TCP提供可靠性。当TCP向另一端发送数据时,它要求对方返回一个确认。如果确认没有收到,TCP自动重传数据并等待更长时间。在数次重传失败后,TCP才放弃。重传数据所花的总时间传统上是4~10分钟(与实现有关)。TCP含有用于动态估算客户到服务器往返所花时间RTT(round-trip time)的算法,因此它知道等待一确认需要多少时间。例如,RTT在一局域网上大约几毫秒,而跨过广域网则需数秒钟。另外,某时刻TCP可能测到客户到服务器的RTT为1秒钟,而过30秒后却测到同一连接的RTT为2秒钟,这是因为网络传输的拥挤情况是在不断变化的。

第三,TCP通过给所发送数据的每一个字节关联一个序列号进行排序。例如,假设一个应用进程写2048字节到一个TCP套接口,导致TCP发送2个分节,第1个分节所含数据的序列号为1~1024,第2个分节所含数据的序列号为1025~2048(分节是TCP传递给IP的数据单元)。如果这些分节非顺序到达,接收方的TCP将根据它们的序列号重新排序,再把结果数据传递给应用进程。如果TCP接收到重复的数据(譬如说对方认为一个分节已丢失并因而重传,而它并没有真正丢失,只是刚才网络通信过于拥挤),它也可以判定数据是重复的(根据序列号),从而把它丢弃掉。

> UDP提供不可靠的数据报传送。UDP本身不提供确认、序列号、RTT估算、超时及重传等机制。如果一个UDP数据报在网上被复制,两份拷贝就可能都递送到接收方的主机。同样,如果一个UDP客户发送两个数据报到同一个目的地,它们可能被网络重新排序,颠倒顺序后到达目的地。UDP应用程序必须处理这些情况,我们将在20.5节中详细介绍。

第四,TCP提供流量控制。TCP总是告诉对方它能够接收多少字节的数据,这称为通告窗口(advertised window)。任何时刻,这个窗口指出接收缓冲区中的可用空间,从而确保发送方发送的数据不会溢出接收缓冲区。窗口时刻动态地变化:当接收发送方的数据时,窗口大小减小,而当接收方应用进程从缓冲区中读取数据时,窗口大小增大。窗口的大小减小到0是有可能的:TCP的接收缓冲区满,它必须等待应用进程从这个缓冲区读取数据后才能再

接收从发送方来的数据。

> UDP 不提供流控制。UDP 按发送方的速率发送数据，不管接收方的缓冲区是否装得下，见 8.13 节。

最后，TCP 的连接是全双工的。这意味着在给定的连接上应用进程在任何时刻既可以发送也可以接收数据。因此，TCP 必须跟踪每个方向数据流的状态信息，如序列号和通告窗口的大小。

> UDP 可以是全双工的。

2.5　TCP 连接的建立和终止

为帮助大家理解 connect、accept 和 close 函数并使用 netstat 调试 TCP 应用程序，我们必须了解如何建立和终止 TCP 连接以及 TCP 的状态转换图。这是一个通过加深了解底层网络协议以帮助我们编写网络程序的例子。

三路握手

下述步骤建立一个 TCP 连接：

1. 服务器必须准备好接受外来的连接。这通过调用 socket、bind 和 listen 函数来完成，称为被动打开(passive open)。
2. 客户通过调用 connect 进行主动打开(active open)。这引起客户 TCP 发送一个 SYN 分节(表示同步)，它告诉服务器客户将在(待建立的)连接中发送的数据的初始序列号。一般情况下 SYN 分节不携带数据，它只含有一个 IP 头部、一个 TCP 头部及可能有的 TCP 选项。
3. 服务器必须确认客户的 SYN，同时自己也得发送一个 SYN 分节，它含有服务器将在同一连接中发送的数据的初始序列号。服务器以单个分节向客户发送 SYN 和对客户 SYN 的 ACK。
4. 客户必须确认服务器的 SYN。

连接建立过程至少需要交换三个分组，因此称之为 TCP 的三路握手(three-way handshake)。我们在图 2.2 中展示这三个分节。

图 2.2 给出的客户的初始序列号为 J，而服务器的初始序列号为 K。在 ACK 里的确认号为发送这个 ACK 的一方所期待的对方的下一个序列号。因为 SYN 只占一个字节的序列号空间，所以每一个 SYN 的 ACK 中的确认号都是相应的初始序列号加 1。类似地，每一个 FIN 的 ACK 中的确认号为 FIN 的序列号加 1。

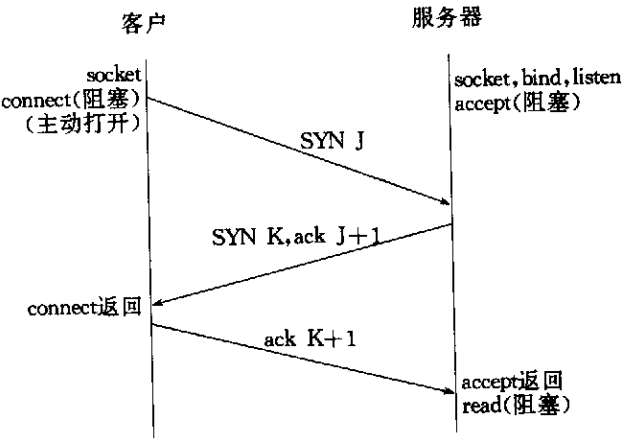


图2.2 TCP的三路握手

建立TCP连接的日常系统类比是电话系统[Nemeth 1997]。socket函数等同于有电话可用。bind用于告诉其他人你的电话号码,让他们可以向你打电话。listen是打开电话振铃,它使你可以听到一个外来的电话。connect要求你知道另一方的电话号码并拨打它。accept是被呼叫回电话。从accept返回客户的标识(即客户的IP地址和端口号)类似于让电话机的呼叫者ID功能部件显示打电话人的电话号码。然而有点不同的地方是:从accept返回客户的标识是在建立连接以后,而呼叫者ID功能部件显示打电话人的电话号码是在我们选择接或不接电话之前。如果使用域名系统(第9章),那么提供了一种类似于电话簿的服务。gethostbyname类似于在电话簿查找个人的电话号码。gethostbyaddr则类似于有一种电话簿按电话号码排序。

TCP选项

每一个SYN可以含有若干个TCP选项。通常使用的选项有:

- MSS选项。TCP发送的SYN中带有这个选项是通知对方它的最大分节大小MSS(maximum segment size),即它能接受的每个TCP分节中的最大数据量。我们会看到如何使用TCP_MAXSEG套接口选项获取与设置这个TCP选项(7.9节)。
- 窗口规模选项。TCP双方能够通知对方的最大窗口大小是65535,因为TCP头部相应的字段只占16位。但是高速连接(每秒45兆位或更快,见RFC1323[Jacobson,Braden,and Borman 1992])或长延迟的路径(卫星链路)要求有更大的窗口以尽可能获得最大的吞吐量。这个新选项指定TCP头部的广告窗口必须扩大(即左移)的位数(0~14),因此所提供的最大窗口几乎是1G字节(65535×2^{14})。两个端系统必须都支持这个选项,否则不具备扩大窗口规模能力。在7.5节我们将看到套接口选项SO_RCVBUF如何影响该选项。

为提供与不支持这个选项的较早实现间的互操作性,应用如下的规则。TCP可以作为主动打开的一部分内容随它的SYN发送该选项,但只有对方也随它的SYN发送该选项时,它才能扩大窗口的规模。类似地,服务器的TCP只有接收到随客户的SYN来的这个选项时,它才能发送该选项。本逻辑假定实现

会忽略它们不理解的选项。这是要求的,而且相当普遍,但不幸的是无法保证所有实现都是这样。

- 时间戳选项。这个选项对高速连接是必要的,它可以防止失而复得的分组可能造成的数据损坏。[①] 因为它是新选项,所以其处理类似于窗口规模选项。作为网络编程人员,对于这个选项我们没什么可担心。

MSS 选项在多数实现中都支持,而窗口规模和时间戳选项则是新的。后两个选项有时也称为"RFC 1323 选项",因为 RFC1323[Jacobson,Braden,and Borman 1992]说明了它们。它们也称为"长胖管道"选项,因为一个高带宽或长延迟的网络被称为"长胖管道(long fat pipe)"。TCPv1 的第 24 章对这些新选项有详细的描述。

TCP 连接终止

TCP 用三个分节建立一个连接,终止一个连接则需四个分节。

1. 某个应用进程首先调用 close,我们称这一端执行主动关闭(active close)。这一端的 TCP 于是发送一个 FIN 分节,表示数据发送完毕。
2. 接收到 FIN 的另一端执行被动关闭(passive close)。这个 FIN 由 TCP 确认。它的接收也作为文件结束符传递给接收方应用进程(放在已排队等候该应用进程接收的任何其他数据之后),因为 FIN 的接收意味着应用进程在相应连接上再也接收不到额外数据。[②]
3. 一段时间后,接收到文件结束符的应用进程将调用 close 关闭它的套接口。这导致它的 TCP 也发送一个 FIN。
4. 接收到这个 FIN 的原发送方 TCP(即执行主动关闭的那一端)对它进行确认。

因为每个方向都需要有一个 FIN 和一个 ACK,所以一般需要四个分节。我们使用限定词"一般"是因为:有时步骤 1 的 FIN 随数据一起发送;另外,执行被动关闭那一端的 TCP 在步骤 2 和 3 发出的 ACK 与 FIN 也可以合并成一个分节。图 2.3 说明了这些分组的交换过程。

FIN 占据 1 个字节的序列号空间,这与 SYN 相同。所以每个 FIN 的 ACK 确认号是这个 FIN 的序列号加 1。

在步骤 2 与步骤 3 之间可以有从执行被动关闭端到执行主动关闭端的数据流。这称为半关闭(half-close),我们将在 6.6 节随 shutdown 函数再详细介绍。套接口关闭时,每一端 TCP 都要发送一个 FIN。这种情况在应用进程调用 close 时会发生,然而在进程终止时,所有打开的描述字将自愿(调用 exit 或从 main 函数返回)或不自愿(进程收到一个终止本进程的

---

① 译者注:失而复得的分组并不是超时重传的分组,而是由暂时的路由原因造成的迷途的分组。当路由稳定后,它们又会正常到达目的地,其前提是它们在此前尚未被路由器主动丢弃。

② 译者注:Unix 文件实际并不使用文件结束符,因为每个文件都对应一个索引节点,其中指出了文件长度,因此不必在文件尾再加个特殊的符号作为文件结束符。本书中文件结束符的读与写分别表示已读到文件尾或结束在文件尾的写。面向字节的数据传送流(如与数据报传送相对的字节流、Unix 的无名管道等)也使用文件结束符表示在某个方向上不再有数据待传送。在 TCP 字节流中,这是通过收发一个特殊的 FIN 分节来实现的。在不引起混淆的前提下,我们分别称发送方或接收方的 TCP 发送或接收了文件结束符。

信号)地关闭,此时仍然打开的TCP连接上也会发出一个FIN。

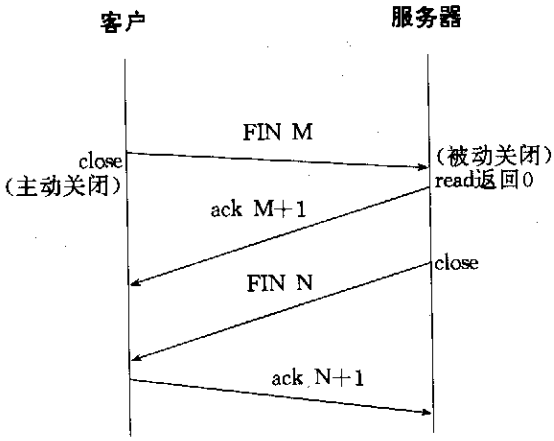


图2.3 TCP关闭连接时的分组交换

图2.3指出客户执行主动关闭,然而不管是客户还是服务器都可以执行主动关闭。通常情况是客户执行主动关闭,但某些协议如HTTP(超文本传送协议)则是服务器执行主动关闭。

TCP状态转换图

TCP连接的建立和终止可以用状态转换图(state transition diagram)来说明,见图2.4。

图中,为一个连接定义了11种状态,并且TCP规则决定如何从一个状态转换到另一个状态,这种转换基于当前状态及在该状态下所接收的分节。例如,当应用进程在CLOSED状态下执行一个主动打开时,TCP将发送一个SYN并从CLOSED状态转换成SYN_SENT状态。如果该TCP接着接收到一个附带ACK的SYN,它将发送一个ACK并转换成ESTABLISHED状态。这个最终状态是数据传送状态。

有两个外向箭头引导自ESTABLISHED状态的连接终止处理。如果应用进程在接收到文件结束符前调用close(主动关闭),则转换成FIN_WAIT_1状态。如果在ESTABLISHED状态下应用进程接收到FIN,则转换成CLOSE_WAIT状态。

我们用实线表示客户的状态转换,虚线表示服务器的状态转换。注意我们没有提到的两个转换:一个为同时打开(当两端几乎同时发送SYN并且这两个SYN在网络中彼此交错时),另一个为同时关闭(当两端同时发送FIN时)。TCPv1的第18章将讨论这两种情况,它们是可能发生的,但非常罕见。

展示状态转换图的理由之一是给出11种TCP状态的名称。netstat命令的输出包括这些状态,它是调试客户-服务器应用程序的有用的工具。在第5章我们将使用netstat去监视这些状态。

观察分组

图2.5说明一个完整的TCP连接所发生的实际分组交换情况:建立连接、传送数据和终止连接。图中展示了每个端点所历经的TCP状态。

例子中客户通告的MSS(最大分节大小)是1460(以太网上IPv4的典型值),而服务器

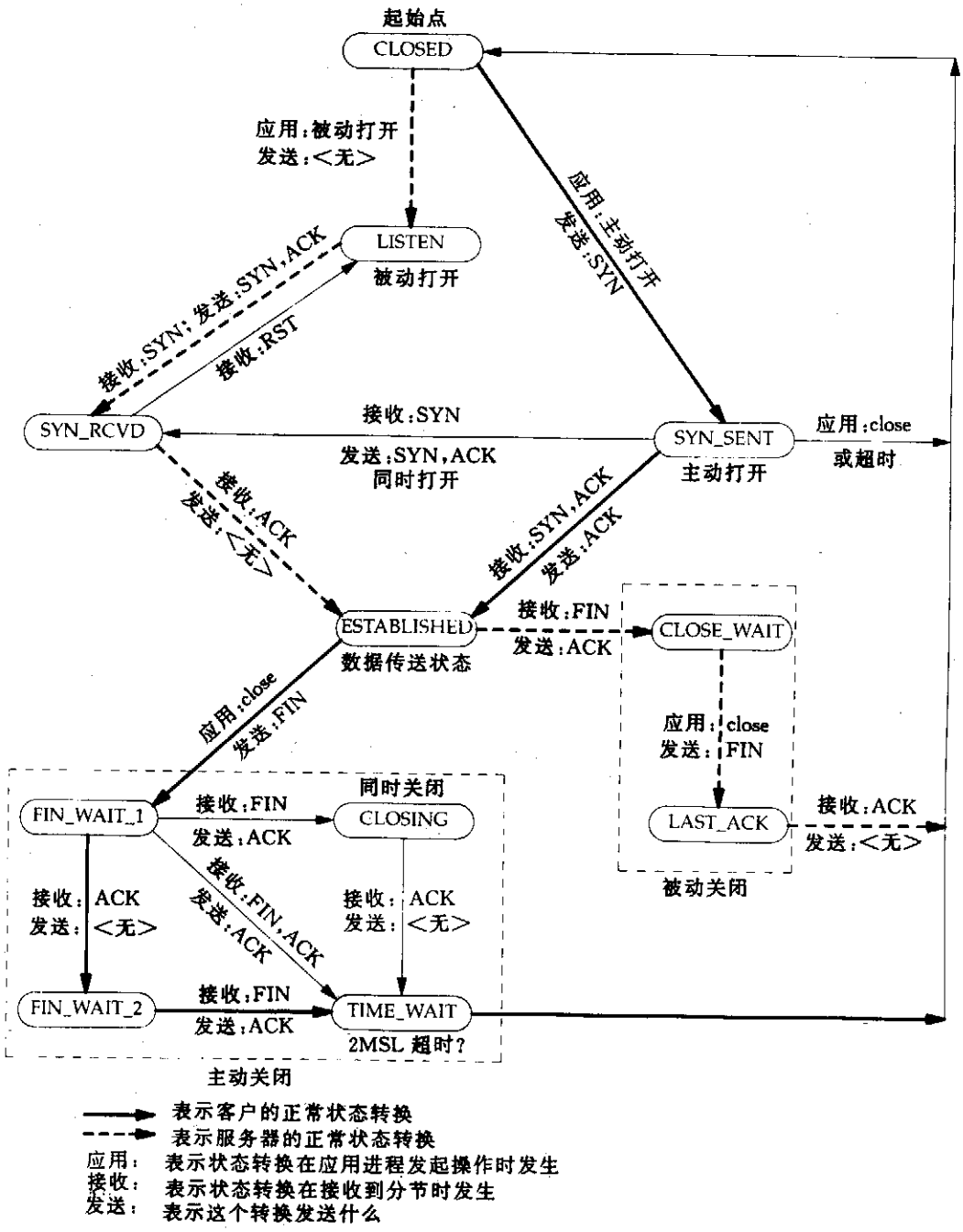


图 2.4　状态转换图

通知的 MSS 为 1024(以太网上较早的源自 Berkeley 实现的典型值)。不同方向的 MSS 可以不相同(习题 2.5)。

一旦连接建立，客户构造一个请求并发送给服务器。这里我们假设请求适合于单个 TCP 分节(即请求大小小于服务器通告的 1024 字节的 MSS)。服务器处理该请求并发送应

答，我们假设应答也适合于单个分节(本例小于1460)。图中，两个数据分节使用粗箭头表示。注意客户请求的确认是伴随服务器的应答发送的。这称为捎带(piggybacking)，它通常在服务器处理请求并产生应答的时间少于200毫秒时发生。如果服务器耗用更长时间，如1秒钟，我们将见到确认过后是应答。(TCP数据流机理在TCPv1的第19和第20章中详细描述。)

以后4个分节终止连接。注意：执行主动关闭的那一端(客户端)进入TIME_WAIT状态，我们将在下一节讨论这种情况。

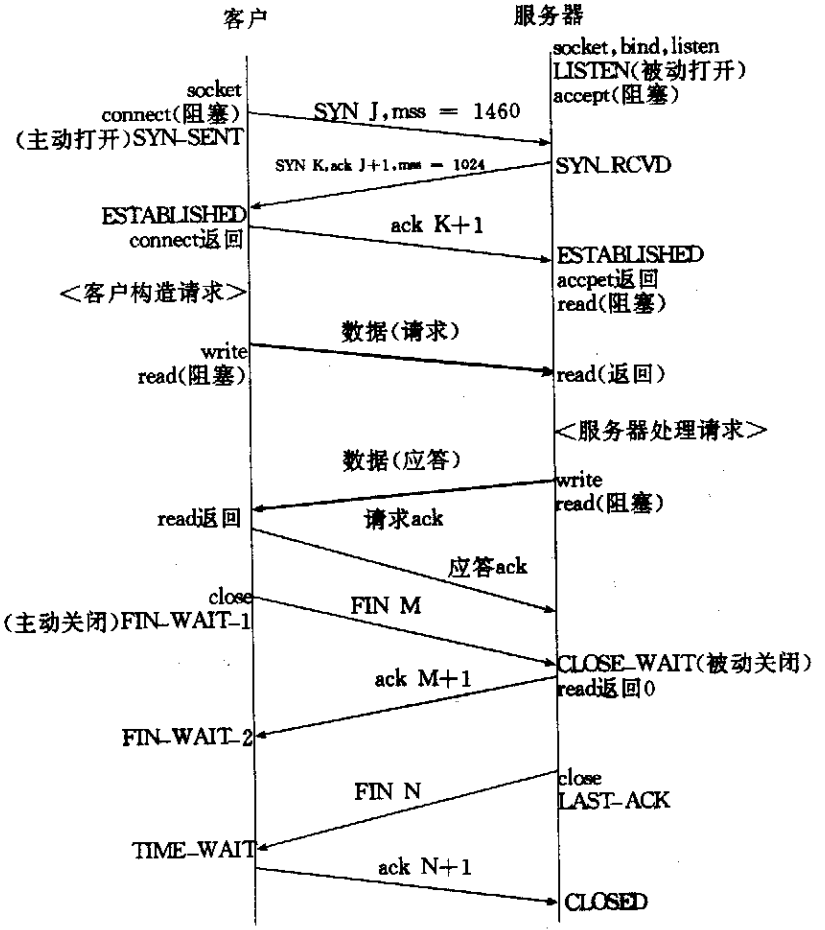


图2.5 TCP连接中的分组交换

图2.5中，如果连接的整个目的是发送一个单一分节的请求和接收一个单一分节的应答，那么使用TCP有8个分节的额外开销。如果使用UDP，那么只有两个分组需要交换：请求和应答。但是如果把TCP换成UDP，TCP提供给应用程序的可靠性将丧失，而把传输层(TCP)的一大堆细节推给UDP应用程序。TCP提供的另一个重要特性是拥塞控制，它也得由UDP应用程序处理。尽管如此，许多应用程序还是使用UDP，因为它们需交换的数据量很小，而UDP又避免了TCP建立连接和终止连接的额外开销。

这种情况下可以用T/TCP即事务TCP代替UDP。我们将在13.9节中描述。

2.6　TIME_WAIT 状态

毫无疑问,有关网络编程的 TCP 中最不容易理解的地方是它的 TIME_WAIT 状态。在图 2.4 中我们看到执行主动关闭的那端进入这种状态。这个端点留在该状态的持续时间是最长分节生命期 MSL(maximum segment lifetime)的两倍,有时称为 2MSL。

每个 TCP 实现都必须选择一个 MSL 值。RFC1122[Braden 1989]的建议值是 2 分钟,而源自 Berkeley 的实现传统上使用的值为 30 秒。这意味着 TIME_WAIT 状态的延迟在 1 分钟至 4 分钟之间。MSL 是 IP 数据报能在互联网中生存的最长时间。我们知道这个时间是有限的,因为每个数据报有一个 8 位的字段称为跳限(见图 A.1 中 IPv4 的 TTL 字段和图 A.2 中 IPv6 的跳限字段),它的最大值为 255。尽管这是一个跳数限制而不是真正的时间限制,我们仍需假设:具有最大跳限(255)的分组不能超过 MSL 秒还继续存在。

分组在路由异常时经常"迷途"。某个路由器崩溃或两个路由器间的链路断开时,路由协议需花数秒或数分钟才能稳定并找出另一条通路。在这段时间内有可能产生路由循环(路由器 A 把分组发送给路由器 B,而 B 再把它们发送回 A),此时分组可能就陷入这样的循环。在此期间,假设迷途的分组是一个 TCP 分节,发送方 TCP 超时并重传该分组,而重传的分组最终选择另一条路径到达目的地。但是,不久(自迷途的分组开始其旅程起最多 MSL 秒内)路由循环修复,迷失在这个循环中的分组最终也到达目的地。这个原来的分组称为迷途的重复分组(lost duplicate)或漫游的重复分组(wandering duplicate),与之相对的重复分组就是超时重传分组。TCP 必须正确处理这种重复的分组。

存在 TIME_WAIT 状态有两个理由:

1. 实现终止 TCP 全双工连接的可靠性
2. 允许老的重复分节在网络中消逝

第一个理由的解释如下:见图 2.5,假设最终的 ACK 丢失,服务器将重发最终的 FIN,因此客户必须维护状态信息以允许它重发最终的 ACK。如果不维护状态信息,它将响应以 RST(另外一个类型的 TCP 分节),而服务器则把该分节解释成一个错误。如果 TCP 打算执行所有必要的工作以彻底终止某个连接上两个方向的数据流(即全双工关闭),那么它必须正确处理连接终止序列四个分节中任何一个分节的丢失情况。本例子也说明执行主动关闭的一端为何进入 TIME_WAIT 状态,因为它可能不得不重发最终的 ACK。

要理解存在 TIME_WAIT 状态的第二个理由,我们假设 206.62.226.33 端口 1500 和 198.69.10.2 端口 21 之间有一个 TCP 连接。我们关闭这个连接后,在以后某个时候又重新建立起相同的 IP 地址和端口之间的 TCP 连接。后一个连接称为前一个连接的化身(incarnation),因为它们的 IP 地址和端口号都相同。TCP 必须防止来自某个连接的老重复分组在连接终止后再现,从而被误解成属于同一连接的化身。要实现这种功能,TCP 不能给处于 TIME_WAIT 状态的连接启动新的化身。既然 TIME_WAIT 状态的持续时间是 2MLS,这就足够让某个方向上的分组最多存活 MSL 秒即被丢弃,另一个方向上的应答最多存活 MSL 秒也被丢弃。通过实施这个规则,我们就能保证当成功建立一个 TCP 连接时,来自该连接先前化身的老重复分组都已在网络中消逝。

这个规则有一个例外：如果到达的SYN的序列号大于前一化身的结束序列号，源自Berkeley的实现将给当前处于TIME_WAIT状态的连接启动新的化身。TCPv2第958～959页对这种情况有详细的描述。它要求服务器执行主动关闭，因为接收到下一个SYN的那一方必须处于TIME_WAIT状态。rsh命令用到这种功能。RFC1185[Jacobson, Braden, and Zhang 1991]讲述了有关这种情形的陷阱。

2.7 端口号

任何时候，多个进程可同时使用TCP或UDP。TCP和UDP都用16位的端口号来区别这些进程。

当一个客户要与服务器接触时，它必须标识要连接的服务器。TCP和UDP定义了一组众所周知端口(well-known ports)，用于标识众所周知的服务。例如，支持FTP的任何TCP/IP实现都把21这个众所周知的端口分配给FTP服务器。TFTP(Trivial File Trqnsfer Protocol)所分配的是UDP端口号69。

另一方面，客户使用临时端口(ephemeral ports)。这些端口号通常由TCP或UDP自动分配给客户。客户一般不关心其临时端口的值，而只需确信它在所在主机中是唯一的。TCP和UDP代码能确保这种唯一性。

RFC1700[Reynolds and Postel 1994]含有一张由因特网已分配数值权威机构(IANA)定义的端口列表。不过，文件ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers通常比RFC 1700还要新。端口号分成三段：

1. 众所周知的端口为0～1023。这些端口由IANA分配和控制。可能时，相同端口号就分配给TCP和UDP的同一给定服务。例如，端口号80对TCP或UDP协议都供Web服务器使用，即使它目前的所有实现都只用TCP也一样。
2. 经注册的端口(registered ports)为1024～49151。这些端口不受IANA控制，但由IANA登记并提供它们的使用情况清单，以方便整个群体。可能时，同一端口号也分配给TCP和UDP的同一给定服务。例如6000～6063分配给这两种协议的X Window服务器。49151的上限是新的，因为RFC1700[Reynolds and Postel 1994]所列的上限为65535。
3. 49152～65535是动态的(dynamic)或私用的(private)端口。IANA不管这些端口，它们就是我们所称的临时端口。

(49152这个数是65535的四分之三。)图2.6说明了这些端口号的分配情况。

图中，我们注意以下各点：

- Unix系统有保留端口(reserved ports)的概念，它是小于1024的任何端口。这些端口只能分配给超级用户进程的套接口。所有众所周知的端口都为保留端口，因此分配这些端口的服务器启动时必须具有超级用户的特权。

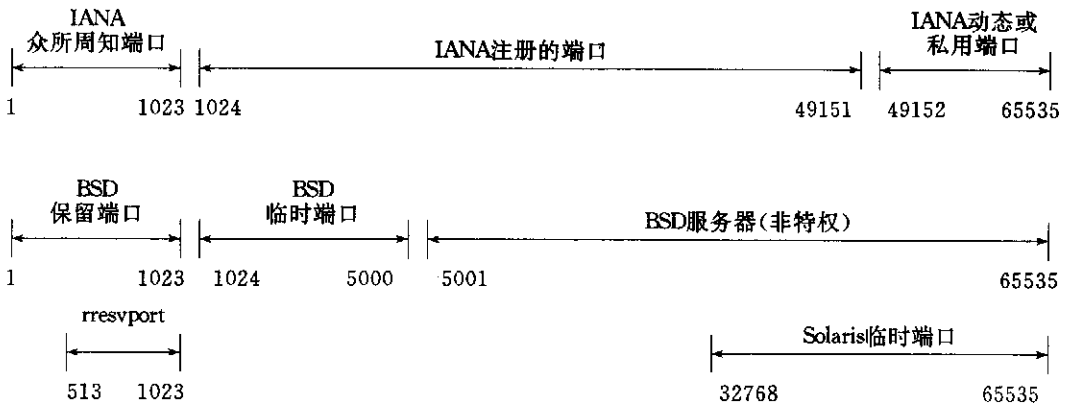


图 2.6　端口号的分配

- 历史上，源自 Berkeley 的实现(起始于 4.3BSD)已在 1024～5000 范围内分配临时端口。这在 80 年代初期情况还不错，因为那时服务器主机还没有能力同时处理那么多客户的请求，但是今天很容易找到一台在任何给定的时间内同时支持多于 3977 个客户的主机。所以有些系统分配另外范围的临时端口(如图 2.6 所示的 Solaris)，用于扩充临时端口的数量。

  由于是这个原因，当前许多系统实现的临时端口范围上限为 5000。5000 这个上限实际上是一个排版错误[Borman 1997a]，本应为 50000。

- 有少数客户(不是服务器)需要一个保留端口用于客户-服务器的认证：rlogin 和 rsh 客户就是常见的例子。这些客户调用库函数 rresvport 创建一个 TCP 套接口并给它分配 513～1023 范围内的一个未使用的端口。这个函数通常先测试 1023，如果失败，再测试 1022、1021……，直到 513。如果到 513 之前能找到一个端口则成功，否则失败。

  注意：BSD 的保留端口和 rresvport 函数都跟 IANA 众所周知端口的后半部分重迭。这是因为早先的 IANA 众所周知端口的上限为 255。1992 年的 RFC1340 开始在 256～1023 之间分配众所周知的端口。先前的“Assigned Numbers”文档即 1990 年的 RFC 1060 称端口 256～1023 为 Unix 标准服务。80 年代有不少源自 Berkeley 的服务器使用 512 以后的端口(留下 256～511 的空档)。rresvport 函数选择从 1023 开始往下寻找，直至 513。

套接口对

一个 TCP 连接的套接口对(socket pair)是一个定义该连接的两个端点的四元组：本地 IP 地址、本地 TCP 端口号、远程 IP 地址、远程 TCP 端口号。套接口对唯一标识一个互联网上的所有 TCP 连接。

标识每个端点的两个值(IP 地址和端口号)通常称为一个套接口。

我们可以把套接口对的概念扩充到 UDP，即使 UDP 是无连接的。当描述套接口函数(bind、connect、getpeername 等)时，我们注明它们在说明套接口对中那个值，如 bind 函数要求应用程序说明本地 IP 地址和本地端口，既可以是 TCP 套接口，也可以是 UDP 套接口。

2.8 TCP端口号与并发服务器

并发服务器中主服务器循环派生子进程来处理每个新的连接。如果子进程仍使用众所周知的端口来服务很长的请求，将发生什么？让我们来检查一个典型的序列。首先服务器在主机bsdi上启动(图1.16)，它是多宿的，有两个IP地址206.62.226.35和206.62.226.66。服务器在它的众所周知的端口(本例为21)上执行被动打开，并等待客户的请求，见图2.7。

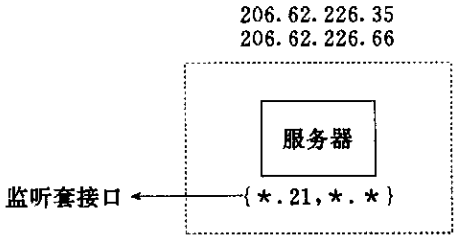


图2.7 TCP服务器在端口21上执行被动打开

我们使用符号{*.21,*.*}指出服务器的套接口对。服务器在任意本地接口(第一个星号)的端口21上等待连接请求。远程IP地址和远程端口没有指定，我们用"*.*"表示。我们称它为监听套接口(listening socket)。

> 我们用点号分开IP地址和端口号，因为这是netstat的用法。这种表示有些让人混淆，因为点号既用于域名(如solaris.kohala.com.21)，也用于IPv4点分十进制数记法(如206.62.226.33.21)。

这里的星号称为通配(wildcard)符。如果运行服务器的主机有多个IP地址(如本例)，服务器可以说明它只接受到达某个特定本地接口的外来连接。这是或者一个接口或者任意接口的选择。服务器不能指定多个地址的列表。通配的本地地址表示"任意"接口。图1.9的通配地址是在调用bind前通过设置套接口地址结构中的IP地址为INADDR_ANY指定的。

此后第一个客户在IP地址为196.69.10.2的主机上启动，对IP地址为206.62.226.35的主机上的服务器执行主动打开。我们假设本例的客户主机TCP选择的临时端口为1500，见图2.8。我们在客户的下方标出了它的套接口对。

当服务器接收并接受这个客户的连接时，它fork一个自身的拷贝，由子进程来处理该客户的请求，见图2.9。我们将在4.7节描述fork函数。

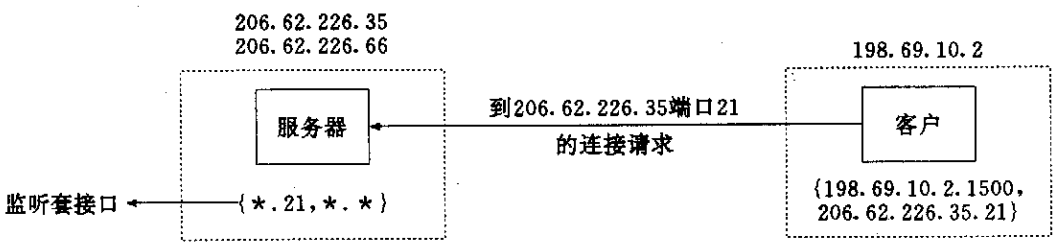


图2.8 客户对服务器的连接请求

此时,我们必须区别服务器的监听套接口和已连接套接口(connected socket)。注意已连接套接口使用跟监听套接口相同的端口号:21。同时注意,一旦连接建立,多宿服务器上已连接套接口的本地地址(206.62.226.35)随即填入。

下一步我们假设在客户主机上有另一个客户进程请求与同一服务器连接。客户主机 TCP 分配给新客户的套接口的未使用临时端口为 1501,见图 2.10。服务器能够区别这两个连接:第一个连接的套接口对和第二个连接的套接口对是不同的,因为客户方的 TCP 给第二个连接选择了一个未使用的端口(1501)。

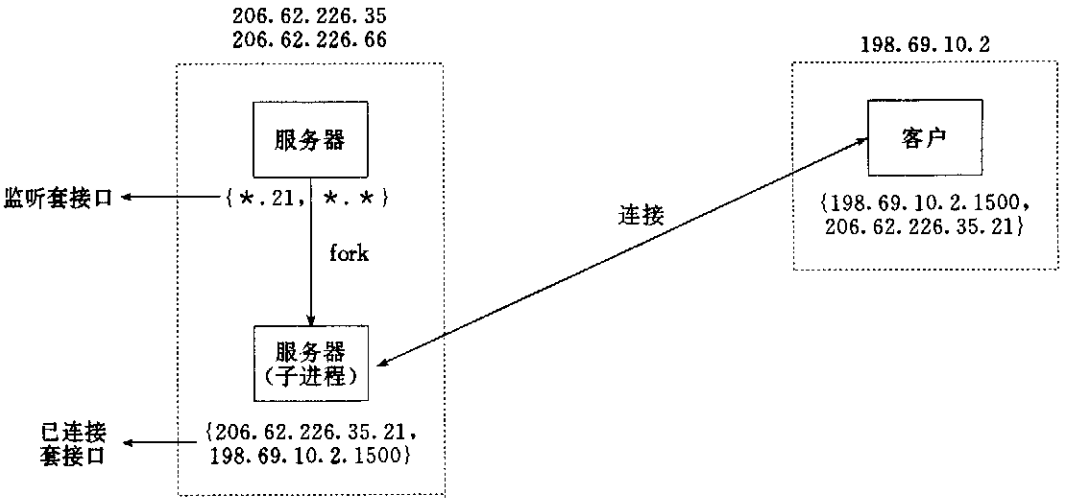


图 2.9　并发服务器让子进程处理客户

通过本例应注意,TCP 无法仅仅通过查看目的端口号来分离外来的分节。它必须查看套接口对的所有四个元素才能确定由哪个端点接收到达的分节。图 2.10 中,对于端口 21 存在三个套接口。如果一个分节来自 198.69.10.2 端口 1501,目的地为 206.62.226.35 端口 21,那它是递送给第一个子进程。如果一个分节来自 198.69.10.2 端口 1501,目的地为 206.62.226.35端口 21,那它是递送给第二个子进程。所有其他的目的端口为 21 的 TCP 分节则是递送给拥有监听套接口的最初服务器(父进程)。

2.9　缓冲区大小及限制

下面我们将介绍一些影响 IP 数据报大小的限制。我们先介绍这些限制,然后就它们如何影响应用进程传送的数据综合分析。

- IPv4 数据报的最大大小是 65535 字节,包括 IPv4 头部。这是因为图 A.1 的总长度字段占 16 位。
- IPv6 数据报的最大大小是 65575 字节,包括 40 字节的 IPv4 头部。这是因为图 A.2 的有效负载长度字段占 16 位。注意 IPv6 的有效负载长度字段不含 IPv6 头部,而 IPv4 的总长度字段含 IPv4 头部。

  IPv6 有一个特大有效负载选项,它把有效负载长度字段扩展到 32 位,不过这个选项要求 MTU(见下一点)超过 65535 的数据链路。这是为主机到主机的内部连接而设

计的，如HIPPI，它们没有内在的MTU。

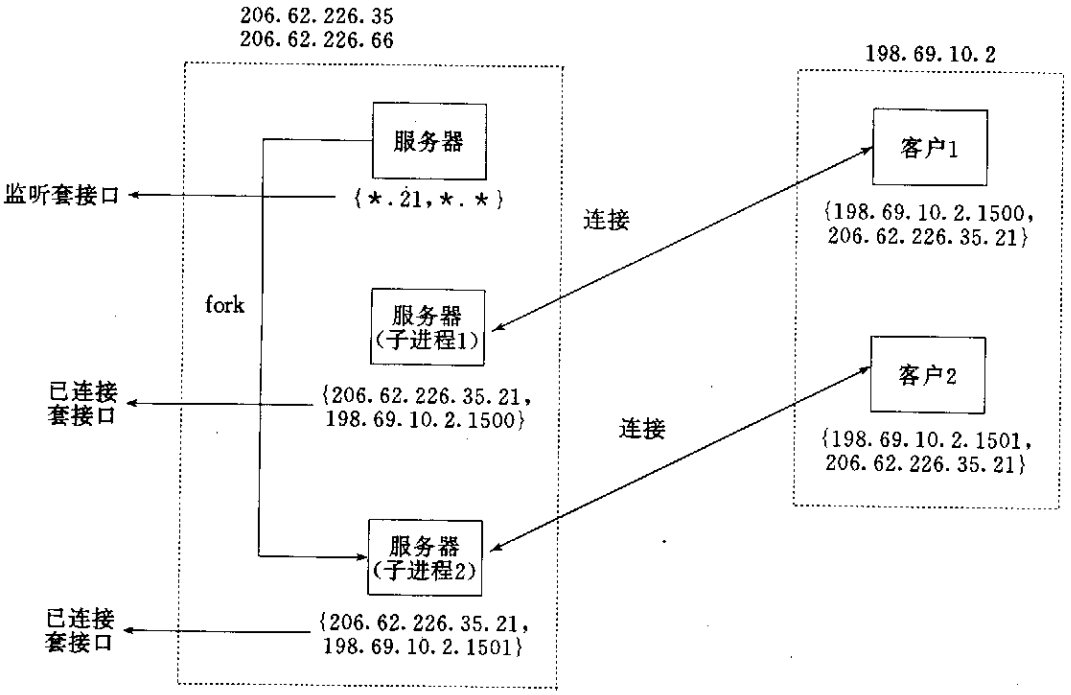


图2.10 第二个客户与同一个服务器的连接

- 很多网络有一个最大传输单元MTU(maximum transmission unit)，它由硬件规定。例如以太网的MTU为1500字节。其他链路(如使用PPP协议的点对点链路)其MTU可以配制。老的SLIP链路通常使用296字节的MTU。
  IPv4要求的最小MTU是68字节，IPv6要求的最小MTU为576字节。
- 在两台主机间的路径上的最小MTU称为路径MTU(path MTU)。1500字节的以太网MTU是当今常见的路径MTU。路径MTU在不同方向可以不相同，因为在因特网中路由是非对称的[Paxson 1196]，即从A到B的路径与从B到A的路径可以不相同。
- 当一个IP数据报将从某个接口发出时，如果它的大小超过相应链路的MTU，IPv4和IPv6都将执行分片(fragmentation)。各片段到达目的地前不会被重组(reassembling)。IPv4主机对其产生的数据报执行分片，IP4路由器对其转发的数据报也执行分片。但IPv6只在数据报产生的主机执行分片；IPv6路由器对其转发的数据报不执行分片。

  我们必须小心这些术语的的使用。IPv6路由器可以执行分片，但只对那些由路由器产生的数据报而不是转发的数据报。这时的路由器实际上作为主机运作。例如，大多数路由器支持Telnet协议，管理员就用它来配置路由器。由路由器的Telnet服务器产生的IP数据报是由路由器产生的，而不是由路由器转发的。

你可能注意到图 A.1 中 IPv4 头部的 DF 字段,它用于处理 IPv4 分片,而在 IPv6 头部不存在相应的字段(图 A.2)。既然分片是例外情况而不是通常情况,因此 IPv6 引入一个可选头部以提供分片信息。

- IPv4 头部(图 A.1)的 DF("不分片")位若被设置,那么不管是发送主机还是转发路由器都不能对本数据报分片。当路由器接收到一个超过其外出链路 MTU 大小且设置了 DF 位的 IPv4 数据报时,它将产生一 ICMPv4 的"destination unreachable, fragmentation needed but DF bit set(目的地不可达,需分片但 DF 位已设置)"出错消息(图 A.15)。

  因为 IPv6 路由器不执行分片,因此 IPv6 数据报隐含设置了 DF 位。如果 IPv6 路由器接收到一个超过其外出链路 MTU 大小的 IPv6 数据报,它将产生一个 ICMPv6 的"packet too big(分组太大)"出错消息(图 A.16)。

  IPv4 的 DF 位和 IPv6 的隐含 DF 位可用于路径 MTU 的发现(IPv4 见 RFC1191 [Mogul and Deering 1990];IPv6 见 RFC 1981[McCann, Deering, and Mogul 1996])。例如,如果 TCP 使用 IPv4 的这个技术,它发送的数据报将都设置 DF 位。如果某个中间路由器返回一个 ICMP"destination unreachable, fragmentation needed but DF bit set"错误,TCP 就减小每个数据报的数据量并重传。路径 MTU 的发现对 IPv4 是可选的,但所有 IPv6 的实现都必须支持它。
- IPv4 和 IPv6 都定义了最小重组缓冲区大小(minimum reassembly buffer size):任何 IPv4 和 IPv6 的实现都必须支持的最小数据报大小。对 IPv4 其值为 576 字节,对 IPv6 为 1500 字节。例如对于 IPv4 来说,我们不能确信给定的目的主机是否能接受 577 字节的数据报。所以有很多使用 UDP 的应用程序(DNS、RIP、TFTP、BOOTP、SNMP)避免产生大于这个大小的数据报。
- TCP 有一个 MSS,用于向对方 TCP 通告对方在每个分节中能发送的最大 TCP 数据量。见图 2.5,SYN 分节带有 MSS 选项。MSS 的目的是告诉对方其重组缓冲区大小的实际值,从而避免分片。MSS 经常设置成 MTU 减去 IP 和 TCP 头部的固定长度。在以太网中使用 IPv4 的 MSS 为 1460,使用 IPv6 的值为 1440(两者的 TCP 头部都为 20 个字节,而 IPv4 头部是 20 字节,IPv6 头部为 40 字节)。

  在 TCP 的 MSS 选项中,MSS 值是一个 16 位的字段,最大值为 65535。这很适合 IPv4,因为 IPv4 数据报中的最大 TCP 数据量为 65495(65535 减去 IPv4 头部 20 个字节和 TCP 头部 20 个字节)。但是 IPv6 有特大有效负载选项,因此需要使用另外的技术(RFC 2147[Borman 1997b])。首先,没有特大有效负载选项的 IPv6 数据报中最大的 TCP 数据量为 65515(65535 减去 TCP 头部 20 字节)。因此 65535 这个 MSS 值被认为是表示"无限"的特殊值,它只在用到特大有效负载选项时才使用,而这种情况又要求 MTU 超过 65535。如果 TCP 使用特大有效负载选项,并且接收到的对方通告的 MSS 为 65535,那么它所发送数据报的大小限制就是接口 MTU。如果这个值太大(也就是说路径上某个链路的 MTU 要比它小),那么路径 MTU 的发现功能将确定这个较小值。

TCP输出

图2.11展示应用进程写数据到TCP套接口的过程。

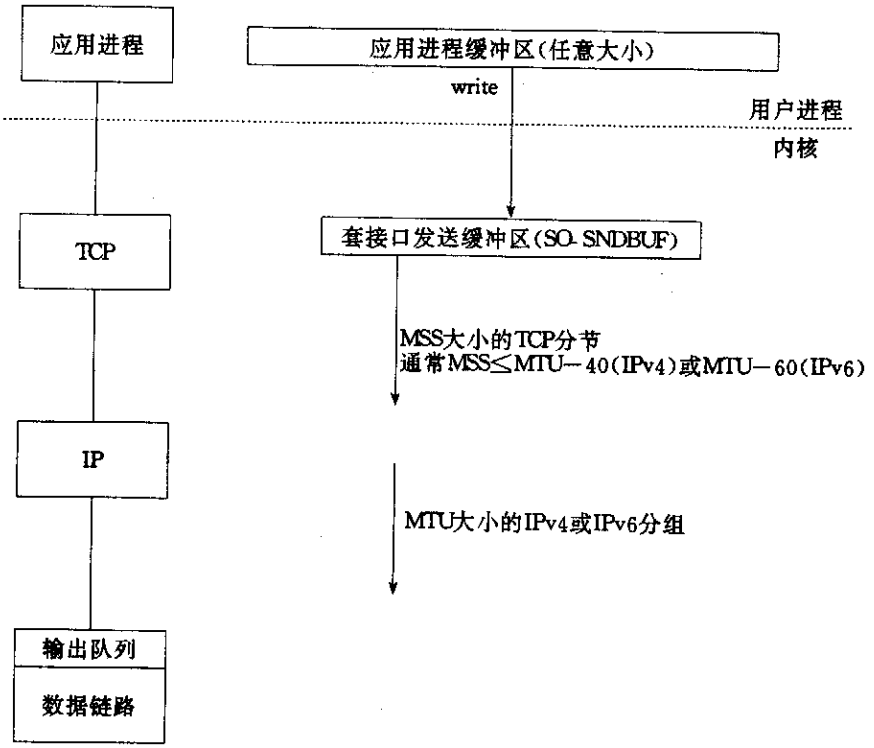


图2.11 应用进程写TCP套接口时涉及的步骤和缓冲区

每一个TCP套接口有一个发送缓冲区,我们可以用SO_SNDBUF套接口选项来改变这一缓冲区的大小(7.5节)。当应用进程调用write时,内核从应用进程的缓冲区中拷贝所有数据到套接口的发送缓冲区。如果套接口的发送缓冲区容不下应用程序的所有数据(或是应用进程的缓冲区大于套接口发送缓冲区,或是套接口发送缓冲区还有其他数据),应用进程将被挂起(睡眠)。这里假设套接口是阻塞的,这是通常的缺省设置(我们将在第15章阐述非阻塞的套接口)。内核将不从write系统调用返回,直到应用进程缓冲区中的所有数据都拷贝到套接口发送缓冲区。所以从写一个TCP套接口的write调用成功返回仅仅表示我们可以重新使用应用进程的缓冲区。它并不告诉我们对方的TCP或对方应用进程已接收到数据(我们将在7.5节详细介绍SO_LINGER套接口选项)。

TCP取套接口发送缓冲区的数据并把它发送给对方TCP,其过程基于TCP数据传送的所有规则(TCPv1的第19和20章)。对方TCP必须确认收到的数据,只有收到对方的ACK,本方TCP才能删除套接口发送缓冲区中已确认的数据。TCP必须保留数据拷贝直到对方确认为止。

TCP以MSS大小或更小的块发送数据给IP(它同时给每个数据块安上TCP头部以构成分节),其中MSS是由对方通告的,当对方未通告时就用536这个值。IP给每个TCP分节安上IP头部以构成数据报,查找其目的IP地址的路由表项以确定外出接口,然后把数据报传递给相应的数据链路。IP可能先将数据报分片,再传送给链路层。但如上所述,MSS选项

的目的是避免分片，而新的实现又使用路径 MTU 发现功能。每一个链路有一个输出队列，如果输出队列满，则分组丢弃，并沿协议栈向上返回一个错误：从链路层到 IP 层，再从 IP 层到 TCP 层。TCP 将注意到这个错误，并在以后某个时刻重传这个分节。应用进程并不知道这种暂时情况。

UDP 输出

图 2.12 展示应用进程写数据到 UDP 套接口的过程。这一次我们展示的套接口发送缓冲区用虚线框，因为它并不存在。UDP 套接口有发送缓冲区大小（我们可用 SO_SNDBUF 套接口选项修改，见 7.5 节），但是它仅仅是写到套接口的 UDP 数据报的大小上限。如果应用进程写一大于套接口发送缓冲区的数据报，则返回 EMSGSIZE 错误。因为 UDP 是不可靠的，它不必保存应用进程的数据拷贝，因此无需一个真正的发送缓冲区。（应用进程的数据在沿协议栈向下传递时，以某种形式拷贝到内核的缓冲区，然而当链路层把数据传出后这个拷贝就丢弃。）

UDP 简单地安上它的 8 个字节的头部以构成数据报并把它传递给 IP。IPv4 或 IPv6 给它安上相应的 IP 头部，执行路由操作确定外出接口，然后或者直接把数据报加入链路层输出队列（如果适合于 MTU），或是分片后再把每个片段加入数据链路层的输出队列。

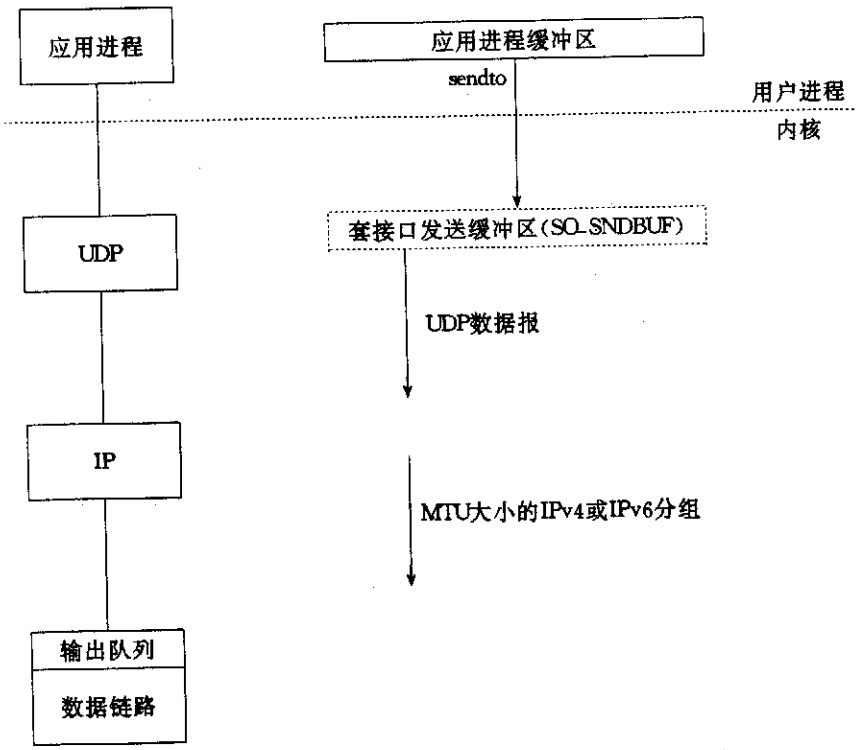


图 2.12　应用进程写 UDP 套接口时涉及的步骤与缓冲区

如果 UDP 应用进程发送一个大数据报（如 2000 字节数据报），它比 TCP 应用进程更有可能分片，因为 TCP 会把应用进程数据划分成 MSS 大小的块，但 UDP 却没有对等的手段。

从写 UDP 套接口的 write 调用成功地返回表示数据报或所有片段已被加入链路层的

输出队列。如果输出队列没有足够的空间存放数据报或它的某个分节,UDP将返回应用进程ENOBUFS错误。

不幸的是,有些UDP的实现不返回这种错误,这样甚至数据报未经发送就丢弃的情况应用进程也不知道。

2.10 标准因特网服务

图2.13列出TCP/IP多数实现都提供的一些标准服务。注意表中所有服务同时使用TCP和UDP提供,并且这两个协议的端口号也相同。

名 字	TCP端口	UDP端口	RFC	说 明
echo(回射)	7	7	862	服务器返回客户发送的数据
discard(丢弃)	9	9	863	服务器废弃客户发送的数据
daytime(时间/日期)	13	13	867	服务器返回直观可读的日期和时间
chargen(字符生成)	19	19	864	TCP服务器发送连续的字符流,直到客户终止连接。每当客户发送一个数据报,UDP服务器就返回一个包含随机数量字符的数据报
time(时间)	37	37	868	服务器返回一个32位二进制数值的时间。这个数值表示从1900年1月1日子时(UTC时间)以来所流逝的秒数

图2.13 大多数实现提供的标准TCP/IP服务③

这些服务通常由Unix主机的inetd守护进程提供。使用标准的Telnet客户程序,很容易测试这些功能。例如,下面测试daytime和echo两种服务器。

```
solaris % telnet bsdi daytime
Trying 206.62.226.35...                    Telnet客户输出
Connected to bsdi.kohala.com.              Telnet客户输出
Escape character is '^]'.                  Telnet客户输出
Tue Mar 19 11:06:49 1996                   daytime服务器输出
Connection closed by foreign host.         Telnet客户输出(服务器关闭连接)
solaris % telnet bsdi echo
Trying 206.62.226.35...                    Telnet客户输出
Connected to bsdi.kohala.com.              Telnet客户输出
Escape character is '^]'.                  Telnet客户输出
hello,world                                我们键入这行
hello,world                                它由服务器回射回来
^]                                         我们键入<Ctrl-]>以与Telnet客户交谈
```

③ 译者注: 本表同时给出了各个标准因特网服务的英文名称和中文名称,其中英文名称是正式名称(/etc/services文件使用这些名称)。之所以这么区分是因为本书是围绕其中两种服务的实现展开的,为区分本书中的实现与各个Unix系统的内部实现,我们用中文名称称呼前者,用英文名称称呼后者(原书也对两者作了区分)。另外,内部实现的服务总是使用标准端口号,本书实现的服务则可根据情况选择。因此我们使用英文名称服务名时,必定与其标准端口号对应。

```
telnet> quit                          告诉客户我们已测试完毕
Connection closed.                    这次客户自己关闭连接
```

在这两个例子中，我们键入主机名和服务名(daytime 和 echo)。这些服务名由/etc/services 文件映射到图 2.13 的端口号，见 9.9 节。

注意：当我们连接到 daytime 服务器时，服务器执行主动关闭；而连接到 echo 服务器时，则客户执行主动关闭。回忆图 2.4，执行主动关闭的 TCP 端是历经 TIME_WAIT 状态的一方。

2.11　常见因特网应用程序的协议使用

图 2.14 总结了各种常见的因特网应用程序对协议的使用情况。

应用程序	IP	ICMP	UDP	TCP
Ping		•		
Traceroute		•	•	
OSPF(路由协议)	•			
RIP(路由协议)			•	
BGP(路由协议)				•
BOOTP(引导协议)			•	
DHCP(引导协议)			•	
NTP(时间协议)			•	
TFTP(低级 FTP)			•	
SNMP(网络管理)			•	
SMTP(电子邮件)				•
Telnet(远程登录)				•
FTP(文件传送)				•
HTTP(Web)				•
NNTP(网络新闻)				•
DNS(域名系统)			•	•
NFS(网络文件系统)			•	•
Sun RPC(远程过程调用)			•	•
DCE RPC(远程过程调用)			•	•

图 2.14　各种常用因特网应用程序的协议使用情况

图中，前两个应用程序 Ping 和 Traceroute 是诊断应用程序，它们使用 ICMP 协议。Traceroute 构造自己的 UDP 分组来发送，并读 ICMP 的应答。紧接着是三个流行的路由协议，它们展示了路由协议使用的各种传输协议。OSPF 采用原始套接口直接使用 IP，而 RIP 使用 UDP，BGP 使用 TCP。

下面 5 个是基于 UDP 的应用程序，接着 5 个应用程序使用 TCP。最后 4 个是同时使用 UDP 和 TCP 的应用程序。

2.12 小 结

UDP是一个简单的、不可靠的无连接的协议，而TCP是一个复杂的、可靠的、面向连接的协议。尽管绝大多数因特网应用程序使用TCP(Web、Telnet、FTP和email)，这两个传输协议都是必要的。20.4节将阐述用UDP替代TCP的理由。

TCP用三路握手建立连接，用四分组交换序列终止连接。当建立TCP连接时，它将从CLOSED状态转换到ESTABLISHED状态，而当终止连接时，它又回到CLOSED状态。在TCP连接中存在11种状态，状态转换图给出从一个状态转换到另一个状态的规则。理解状态转换图是使用netstat命令诊断网络问题的基础，也是理解调用诸如connect、accept和close等函数时所发生过程的关键。

TCP的TIME_WAIT状态是网络编程人员不容易理解的概念。存在这一状态是为了实现完整地终止全双工的连接(即处理最终的ACK丢失的情形)，并允许老的重复分节从网络中消逝。

2.13 习 题

2.1 我们已经提到IP版本4和版本6。IP版本5有什么情况，IP版本0、1、2和3又是什么？(提示：查找最新的"Assigned Numbers"RFC。)

2.2 从哪里你可以找到有关IP版本5的信息。

2.3 图2.11中，如果没收到对方的MSS选项则TCP使用536的MSS值，为什么使用这个值？

2.4 画出类似于图2.5的第1章所述时间/日期客户-服务器的分组交换过程，假设服务器在单个分节中返回26个字节的完整数据。

2.5 在一台以太网上的主机和一台令牌环网上的主机之间建立连接，其中以太网上主机通告的MSS为1460，令牌环网上主机通告的MSS为4096。两端主机都没有实现路径MTU发现功能。观察分组，我们在两个方向上都找不到大于1460字节的数据，为什么？

2.6 图2.14中我们说OSPF直接使用IP，OSPF数据报的IPv4头部(图A.1)的协议字段是什么值？

第 2 部分　基本套接口编程

第 3 章　套接口编程简介

3.1　概　述

本章开始描述套接口 API。我们从套接口地址结构开始，本书的每个例子几乎都有这些结构，它们可以在两个方向上传递：从进程到内核和从内核到进程。其中从内核到进程方向的传递是值-结果参数的一个例子，本书中，我们将会碰到这些参数的许多例子。

地址转换函数在地址的文本表达和套接口地址结构中的二进制值间进行转换。多数现存的 IPv4 代码使用 inet_addr 和 inet_ntoa 函数，而两个新函数 inet_pton 和 inet_ntop 则适用于 IPv4 和 IPv6 两种代码。

这些地址转换函数有一个问题，即它们是与所转换的地址类型协议相关的：要么 IPv4 要么 IPv6。为克服此问题，我们开发了一组函数，这组函数的命名均以“sock_”打头，以协议无关方式使用套接口地址结构。在整个第 2 部分，我们都使用这组函数，以使我们的代码与协议无关。

3.2　套接口地址结构

大多数套接口函数都需要一个指向套接口地址结构的指针作为参数。每个协议族都定义它自己的套接口地址结构。这些结构的名字均以“sockaddr_”开头，并以对应每个协议族的唯一后缀结束。

IPv4 套接口地址结构

IPv4 套接口地址结构通常也称为“网际套接口地址结构”，它以“sockaddr_in”命名，定义在头文件<netinet/in.h>中。图 3.1 所示为 Posix.1g 的定义。

```
struct in_addr {
  in_addr_t  s_addr;          /* 32-bit IPv4 address */
                              /* network byte ordered */
};
struct sockaddr_in {
  uint8_t        sin_len;     /* length of structure (16) */
  sa_family_t    sin_family;  /* AF_INET */
  in_port_t      sin_port;    /* 16-bit TCP or UDP port number */
                              /* network byte ordered */
  struct in_addr sin_addr;    /* 32-bit IPv4 address */
                              /* network byte ordered */
```

```
    char        sin_zero[8];    /* unused */
};
```

图 3.1 网际(IPv4)套接口地址结构:sockaddr_in

利用图 3.1 所示的例子,我们对套接口地址结构做几点一般性的说明。

- 长度成员 sin_len,是为增加 OSI 协议(图 1.15)支持,随 4.3BSD-Reno 一起增加的。在此之前,第一个成员是 sin_family,它是一个无符号短整数 unsigned short。并不是所有的厂家都支持套接口地址结构长度成员,而且 Posix.1g 也不要求有这个成员。我们给出的数据类型 uint8_t 是典型的,且这种数据类型是随 Posix.1g 一起新添加的(图 3.2)。

  正是因为有了长度成员,才简化了变长套接口地址结构的处理。

- 即使有长度成员,我们也无需设置它、无需检查它,除非我们涉及到路由套接口(第 17 章)。它是内核中处理来自不同协议族的套接口地址结构(如路由表代码)的例程使用的。

  从进程到内核传递套接口地址结构的 4 个套接口函数:bind、connect、sendto 和 sendmsg,都要经历源自 Berkeley 的实现中的 sockargs 函数。这个函数从进程拷贝套接口地址结构,并显式地以作为参数传递给这 4 个函数的地址结构的长度设置 sin_len 成员。而从内核到进程传递套接口地址结构的 5 个套接口函数:accept、recvfrom、recvmsg、getpeername 和 getsockname,均在返回到进程之前设置 sin_len 成员。

  遗憾的是,通常没有一个简单的编译时测试来确定一个实现是否为它的套接口地址结构定义了长度成员。在我们的代码中,我们通过测试常值 HAVE_SOCKADDR_SA_LEN 来确定(图 D.2),但是否定义这个常值则需编译一个使用这一可选结构成员的简单测试程序并看是否编译成功来决定。在图 3.4 中我们将看到,如果套接口地址结构有长度成员,则 IPv6 实现需定义 SIN6_LEN。一些 IPv4 实现(如 Digital Unix)基于编译选项(如_SOCKADDR_LEN)确定是否给应用程序提供套接口地址结构中的长度成员,这为较早的程序提供了兼容性。

- Posix.1g 只需要这个结构中的 3 个成员:sin_family、sin_addr 和 sin_port。对于 Posix 兼容的实现来说,定义额外的结构成员是可以接受的,这对于网际套接口地址结构来说也是正常的。几乎所有的实现都增加 sin_zero 成员,所以所有的套接口地址结构大小都至少是 16 字节。
- 我们给出了 s_addr、sin_family 和 sin_port 的 Posix.1g 数据类型。in_addr_t 数据类型必须是一个至少 32 位的无符号整数类型,in_port_t 必须是一个至少 16 位的无符号整数类型,而 sa_family_t 可以是任何无符号整数类型,在支持长度成员的实现中,它一般是一个 8 位的无符号整数,而在不支持长度成员的实现中,它则是一个 16 位的无符号整数。图 3.2 列出了 Posix.1g 定义的这些数据类型以及将会遇到的其他数据类型。

数据类型	说明	头文件
int8_t	带符号的 8 位整数	<sys/types.h>
uint8_t	无符号的 8 位整数	<sys/types.h>
int16_t	带符号的 16 位整数	<sys/types.h>
uint16_t	无符号的 16 位整数	<sys/types.h>
int32_t	带符号的 32 位整数	<sys/types.h>
uint32_t	不带符号的 32 位整数	<sys/types.h>
sa_family_t	套接口地址结构的地址族	<sys/socket.h>
socklen_t	套接口地址结构的长度，一般为 uint 32_t	<sys/socket.h>
in_addr_t	IPv4 地址，一般为 uint32_t	<netinet/in.h>
in_port_t	TCP 或 UDP 端口，一般为 uint16_t	<netinet/in.h>

图 3.2　Posix.1g 要求的数据类型

- 我们也将遇到这样的数据类型：u_char、u_short、u_int 和 u_long，它们都是无符号的。Posix.1g定义这些类型时特地标记它们已过时，是仅为向后兼容才提供的。
- IPv4 地址和 TCP 或 UDP 端口号在套接口地址结构中总是以网络字节序来存储，我们在使用这些成员时，必须牢记这一点。(在 3.4 节我们将详细说明主机字节序与网络字节序的区别。)
- 可以有两种不同的方法来访问 32 位 IPv4 地址。例如如果 serv 定义为网际套接口地址结构，那么 serv.sin_addr 给出的 32 位 IPv4 地址将是一个 in_addr 结构，而 serv.sin_addr.s_addr 给出的 32 位 IPv4 地址则是一个 in_addr_t(通常是无符号的 32 位整数)。因此，我们必须正确地使用 IPv4 地址，尤其是在将它作为函数的参数时，因为编译器对传递结构和传递整数的处理是完全不同的。

  由于历史上的原因，sin_addr 成员是一个结构，而不仅仅是一个无符号长整数。早期的版本(4.2BSD)将 in_addr 结构定义为不同结构的联合，以允许访问 32 位 IPv4 地址 4 个字节中的每个字节，或者 2 个 16 位值中的任一个。这用在 A 类、B 类和 C 类地址中获取地址中的适当字节。但随着子网划分技术的来临和无类地址编排(A.4 节)的出现，各种地址类正在消失，这就不需要联合了。今天，多数系统已经废除了该联合，将 in_addr 定义为仅有一个无符号长整数成员的结构。

- sin_zero 成员暂不使用，但总是将它置为 0。为方便起见，在初始化结构时，我们一般是将整个结构置为 0，而不仅仅是设置 sin_zero 成员为 0。

  虽然多数结构的使用不要求这一成员为 0，但当捆绑一个非通配 IPv4 地址时，此成员必须为 0(TCPv2 第 731～732 页)。

- 套接口地址结构仅在给定主机上使用：虽然结构中的某些成员(如 IP 地址和端口号)用在不同主机间的通信中，但结构本身并不参与通信。

通用套接口地址结构

当作为参数传递给任一个套接口函数时，套接口地址结构总是通过指针来传递，但通过指针来取得此参数的套接口函数必须处理来自所支持的任何协议族的套接口地址结构。

有一个问题是如何声明所传指针的数据类型。ANSI C 中有很简单的解决办法：它有通用的指针类型 void * 。但是，套接口函数是在 ANSI C 之前定义的，1982 年采用了这样的一个办法：在<sys/socket.h>头文件中定义一个通用的套接口地址结构，如图 3.3 所示。

```
struct sockaddr {
  uint8_t       sa_len;
  sa_family_t   sa_family;    /* address family: AF_xxx value */
  char          sa_data[14];  /* protocol-specific address */
};
```

图 3.3 通用套接口地址结构：sockaddr

于是，套接口函数被定义为采用指向通用套接口地址结构的指针，这正如用 ANSI C 函数原型写出来的 bind 函数所示：

```
int bind (int , struct sockaddr * , socklen_t);
```

这要求对这些函数的任何调用都必须将指向特定于协议的套接口地址结构的指针类型转换成指向通用套接口地址结构的指针。例如：

```
struct sockaddr_in  serv;       /* IPv4 socket address structure */
/* fill in serv{} */
bind(sockfd, (struct sockaddr * ) &serv, sizeof(serv));
```

如果我们省略了其中的类型转换“(struct sockaddr *)”，再假设系统的头文件中有一个 bind 函数的 ANSI C 原型，那么 C 编译器就会产生这样的警告信息：“warning：passing arg 2 of 'bind' from incompatible pointer type(警告：把不兼容的指针类型传递给 'bind' 函数的第二个参数)”。

从应用程序开发人员的观点看，这些通用的套接口地址结构的唯一用途是给指向特定于协议的地址结构的指针转换类型。

> 回忆一下 1.2 节，我们在头文件 unp.h 中把 SA 定义为串“struct sockaddr”正是为了缩短转换这些指针的类型必须写的代码。
>
> 从内核的角度看，使用指向通用套接口地址结构指针的原因是：内核必须依据调用者的指针，将其转换为 struct sockaddr * 类型，然后检查 sa_family 的值来确定结构的类型。但从应用程序开发人员的角度看，指针类型为 void * 则更简单，不需进行明确的类型转换。

IPv6 套接口地址结构

IPv6 套接口地址结构在头文件<netinet/in.h>中定义，如图 3.4 所示。

```
struct in6_addr {
  unit8_t s6_addr[16];          /* 128-bit IPv6 address */
                                /* network byte ordered */
```

```
};
#define SIN6_LEN      /* required for compile-time tests */
struct sockaddr_in6 {
  uint8_t          sin6_len;      /* length of this struct (24) */
  sa_family_t      sin6_family;   /* AF_INET6 */
  in_port_t        sin6_port;     /* transport layer port# */
                                  /* network byte ordered */
  uint32_t         sin6_flowinfo; /* priority & flow label */
                                  /* network byte ordered */
  struct in6_addr  sin6_addr;     /* IPv6 address */
                                  /* network byte ordered */
};
```

图 3.4　IPv6 套接口地址结构：sockaddr_in6

关于 IPv6 套接口 API 的扩展定义在 RFC 2133 中[Gilligan et al. 1997]，Posix.1g 对 IPv6 未作任何说明。由于我们在图 3.4 中使用了可能的 Posix.1g 数据类型，而这些类型又是在 RFC 2133 面世后的一个 Posix.1g 草案中规定的，所以图中的某些数据类型与 RFC 2133 不符。

对于图 3.4，我们应注意以下几点：

- 如果系统支持套接口地址结构中的长度成员，则 SIN6_LEN 常值必须定义。
- IPv6 地址族是 AF_INET6，而 IPv4 地址族是 AF_INET。
- 结构中的成员是有序排列的，因此，如果 sockaddr_in6 结构是 64 位对齐的，则 128 位的成员 sin6_addr 也是 64 位对齐的。在一些 64 位处理机上，如果数据存储在 64 位边界的位置，则对 64 位数据的访问将优化处理。
- sin6_flowinfo 成员分成三个字段：
  - 低 24 位是流量标号；
  - 下 4 位是优先级；
  - 再下 4 位保留。

流量标号和优先级字段在图 A.2 中描述。必须注意，优先级字段的使用仍是一个研究课题。

套接口地址结构的比较

在图 3.5 中，我们对本书将遇到的 4 种套接口地址结构进行了比较：IPv4、IPv6、Unix 域(图 14.1)和数据链路(图 17.1)。在这个图中，我们假设所有的套接口地址结构都包含 1 个字节的长度成员，地址族成员也占用一个字节，其他所有成员都占有确切的最短长度。前两种套接口地址结构是定长的，而 Unix 域结构和数据链路结构是可变长度的。为了处理可变长度的结构，我们把指向套接口地址结构的指针作为参数传递给套接口函数，同时把它的长度作为另一个参数传递。我们在每个定长结构下方给出了此结构的字节数(4.4BSD 实现)。

结构 sockaddr_un 本身不是可变长度的(图 14.1),但其信息量即结构中的路径名的长度则是可变的。当传递指向这些结构的指针时,我们必须小心处理长度成员即套接口地址结构本身的长度成员(如果其实现支持此成员),以及传给内核或从内核返回的长度。

这个图展示了我们贯穿全书的一种风格:结构名用黑体字,后跟花括号,如:**sockaddr_in{}**。

前面我们已经注意到,长度成员是随着 4.3BSD Reno 版本一起增加到所有的套接口地址结构中的,如果套接口 API 的原始版本中已经有了长度成员,那么所有的套接口函数就不需要长度参数,如函数 bind 和 connect 的第三个参数。与此相反,结构的大小可以包含在结构的长度成员中。

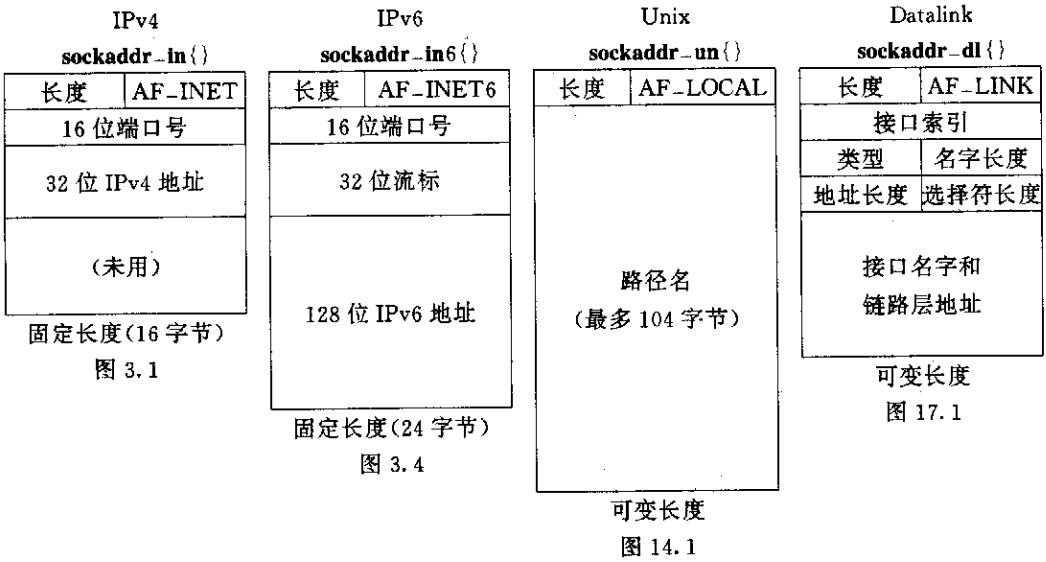


图 3.5 不同套接口地址结构的比较

3.3 值-结果参数

我们已经知道,当把套接口地址结构传递给套接口函数时,总是通过指针来传递的,即传递的是一个指向结构的指针。结构的长度也作为参数来传递,其传递的方式取决于结构的传递方向:从进程到内核,还是从内核到进程。

1. 从进程到内核传递套接口地址结构有 3 个函数:bind、connect 和 sendto,这 3 个函数的一个参数是指向套接口地址结构的指针,另一个参数是结构的整数大小,例如:

```
struct sockaddr_in serv;
/* fill in serv{} */
connect (sockfd,(SA *) &serv, sizeof(serv));
```

由于指针和指针所指结构的大小都传递给内核,所以从进程到内核要确切拷贝多少数据是知道的,如图 3.6 所示。

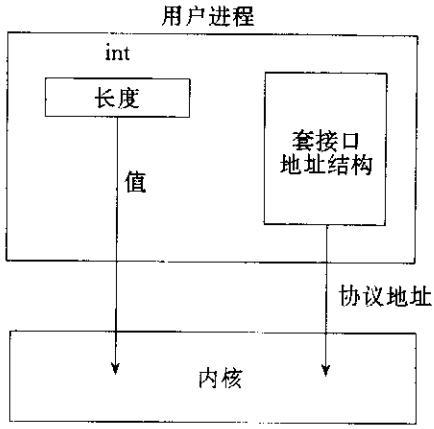


图 3.6　从进程到内核传递套接口地址结构

下一章中我们将看到，套接口地址结构大小的数据类型确切地说应该是 socklen_t，而不是 int，但 Posix.1g 建议将 socklen_t 定义为 uint32_t。

2. 与前面的传递方向相反，从内核到进程传递套接口地址结构有 4 个函数：accept、recvfrom、getsockname 和 getpeername。这 4 个函数的两个参数是：指向套接口地址结构的指针和指向表示结构大小的整数的指针，例如：

```
struct sockaddr_un cli;          /* Unix domain */
socklen_t len;
len = sizeof(cli);               /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

为何将结构大小由整数改为指向整数的指针呢？这是因为：当函数被调用时，结构大小是一个值（此值告诉内核该结构的大小，使内核在写此结构时不至于越界），当函数返回时，结构大小又是一个结果（它告诉进程内核在此结构中确切存储了多少信息），这种参数类型叫值-结果参数，如图 3.7 所示。

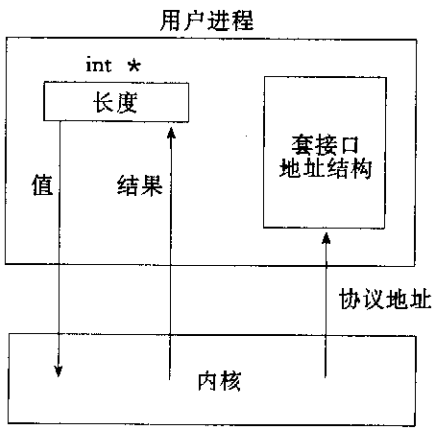


图 3.7　从内核到进程传递套接口地址结构

我们将在图4.11中看到一个值-结果参数的例子。

> 我们已经对在进程和内核间传递的套接口地址结构进行了讨论。对于诸如4.4BSD的实现，所有的套接口函数都是内核中的系统调用，这是正确的。但在某些实现中，如著名的系统V，套接口函数只是作为普通用户进程执行的库函数。这些函数与内核中的协议栈如何接口是一个实现的细节问题，一般来说，对我们不会有什么影响。然而为简单起见，我们继续说这些结构通过诸如bind和connect这样的函数在进程与内核间进行传递。(在C.1节我们将看到系统V的确也在进程和内核间传递用户的套接口地址结构，但这是作为流消息(streams messages)的一部分进行传递的。)
>
> 另有两个传递套接口地址结构的函数：recvmsg和sendmsg(13.5节)，但我们将看到，长度不是一个函数参量，而是一个结构成员。

当使用值-结果参数作为套接口地址结构的长度时，如果套接口地址结构是定长的(图3.5)，则从内核返回的值也是定长的，如对于IPv4，sockaddr_in是16；对于IPv6，sockaddr_in6是24。但对于变长的套接口地址结构(如Unix域的sockaddr_un)，返回值可能比结构的最大长度小(图14.2)。

在网络编程中，值-结果参数的最常见例子就是返回的套接口地址结构长度，本书中我们还能见到其他的值-结果参数：

- select函数中间的3个变量(6.3节)。
- getsockopt函数的长度变量(7.2节)。
- 使用函数recvmsg时，msghdr结构中的两个成员：msg_namelen和msg_controllen(13.5节)。
- ifconf结构中的成员ifc_len(图16.2)。
- sysctl函数的前两个长度参数(17.4节)。

3.4 字节排序函数

考虑一个16位整数，它由2个字节组成。内存中存储这两个字节有两种方法：一种是将低序字节存储在起始地址，这称为小端(little-endian)字节序，另一种方法是将高序字节存储在起始地址，这称为大端(big-endian)字节序。图3.8示出了这两种格式。

在图的顶部存储地址增大的方向为从右到左，在图的底端存储地址增大的方向为从左到右，MSB(最高有效位)为16位数的最左一位，LSB(最低有效位)为16位数的最右一位。

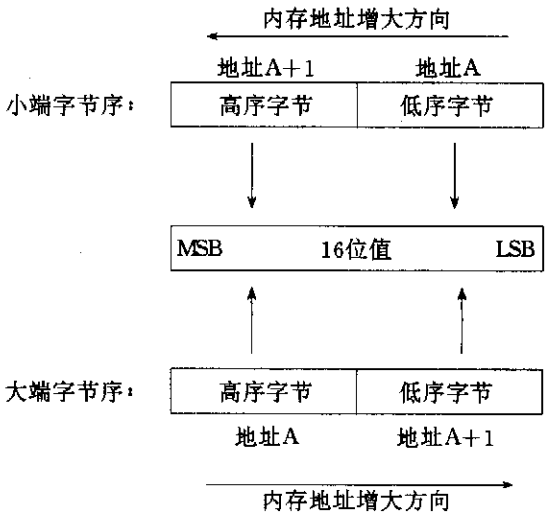


图 3.8 16 位整数的小端字节序和大端字节序

术语“小端”和“大端”表示多字节值的哪一端(小端或大端)存储在该值的起始地址。

遗憾的是,这两种字节序中没有标准,这两种格式都有系统使用。我们把某给定系统所用的字节序称为主机字节序(host byte order),图 3.9 中所示程序能输出主机字节序。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     union {
6         short   s;
7         char    c[sizeof(short)];
8     } un;
9     un.s = 0x0102;
10    printf("%s:",CPU_VENDOR_OS);
11    if(sizeof(short) == 2) {
12        if(un.c[0] == 1 && un.c[1] == 2)
13            printf("big-endian\n")
14        else if (un.c[0] == 2 && un.c[1] == 1)
15            printf("little-endian\n");
16        else
17            printf("unknown\n");
18    } else
19        printf("sizeof(short) = %d\n",sizeof(short));
20    exit(0);
21 }
```

图 3.9 确定主机字节序的程序[intro/byteorder.c]

我们把2字节数0x0102存储为一个短整数，然后查看两个连续的字节c[0]（图3.8中的地址A）和c[1]（图3.8中的地址A+1），以此来确定字节序。

字符串CPU_VENDOR_OS由GNU的autoconf程序在配置本书中的软件时确定，它标识CPU类型、厂家和操作系统版本。我们这儿给出一些例子，它们是这个程序在图1.16所示的各系统上运行时的输出。

```
aix % byteorder
powerpc-ibm-aix4.2.0.0:big-endian
alpha % byteorder
alpha-dec-osf4.0: little-endian
bsdi % byteorder
i386-pc-bsdi3.0: little-endian
gemini % byteorder
sparc-sun-sunos4.1.4: big-endian
hpux % byteorder
hppa1.1-hp-hpux10.30: big-endian
linux % byteorder
i586-pc-linux-gnu: little-endian
solaris % byteorder
sparc-sun-solaris2.5.1: big-endian
unixware % byteorder
i386-univel-sysv4.2MP: little-endian
```

我们已讨论了16位整数的字节序，显然，相同的讨论也适用于32位整数。

> 有些系统，或者在系统复位时（MIPS 2000），或者在程序运行中的任何时刻（Intel i860），可以在大端字节序和小端字节序间互换。

由于网络协议必须指定一个网络字节序（network byte order），作为网络编程人员，必须清楚各种字节序间的区别。例如，在一个TCP分节中，有一个16位的端口号和一个32位的IPv4地址，发送协议栈和接收协议栈必须对此多字节字段的传送字节序进行协调，以达成一致。网际协议在处理这些多字节整数时，使用大端字节序。

从理论上说，具体实现可以把多字节整数按主机字节序存储在套接口地址结构中，当需要写到协议头部或从协议头部中取出时，再转换到网络字节序格式或转换回来，这样可以避免我们牵涉细节。但是，由于历史上的原因和Posix.1g的规定，套接口地址结构中的某些成员按网络字节序进行维护。这样，我们就必须考虑主机字节序和网络字节序间的相互转换，转换时用到下列四个函数：

```
#include <netinet/in.h>
    uint16_t htons(uint16_t host16bitvalue);
    uint32_t htonl(uint32_t host32bitvalue);
                                                    均返回：网络字节序值
    uint16_t ntohs(uint16_t net16bitvalue);
    uint32_t ntohl(uint32_t net32bitvalue);         均返回：主机字节序值
```

在上述函数的名字中，h 代表 host，n 代表 network，s 代表 short，l 代表 long。short 和 long 是4. 2BSD的 Digital VAX 实现的历史产物。我们应该转变思想，把 s 看作一个 16 位的值(如 TCP 或 UDP 端口号)，把 l 看作 32 位的值(如 IPv4 地址)。实际上，在 64 位的 Digital Alpha 中，长整数占用 64 位，而 htonl 和 ntohl 函数却对 32 位值操作。

当使用这些函数时，我们不关心主机字节序和网络字节序的真实值(大端或小端)。我们所做的只是调用适当的函数来对给定值进行主机字节序与网络字节序间的转换。在那些与网际协议有相同字节序(大端)的系统中，这四个函数通常被定义为空宏。

对于网络分组中所含的数据而不是协议头部中字段的字节序问题，我们将在 5. 18 节和习题 5. 8 中进行进一步的讨论。

至此，我们还没有定义术语字节(byte)。既然几乎所有的计算机系统使用 8 位字节，我们就用此术语来表示一个 8 位的量。大多数因特网标准用术语 octet(八位组)而不是 byte 来表示 8 位的量，这在 TCP/IP 时代的早期就已是这种局面了，因为许多早期的工作是在诸如 DEC-10 这样的系统上开展的，而这些系统不使用 8 位的字节。

> 20 世纪 80 年代网络编程有一个通病：在 Sun 工作站(大端 Motorola 68000)上开发代码时没有调用这四个函数中的任一个。这些代码在这些工作站上工作得很好，但当移植到小端机器(如 VAX 系列机)上时，便根本不能工作。

3.5　字节操纵函数

多字节字段的操纵有两组函数，它们无需解释数据，无需假设数据是以空字符结束的 C 字符串。当我们涉及套接口地址结构这类问题时，就需要这样的函数，因为我们要对诸如 IP 地址这样的字段进行操作。这样的字段可能包含多个字节的 0，但又不是 C 字符串。在头文件<string. h>中定义、名字以 str 打头的函数处理的是以空字符结束的 C 字符串。

第一组函数，其名字以字母 b 打头，代表 byte，起源于 4. 2BSD，现仍由几乎任何支持套接口函数的系统提供；第二组函数，其名字以 mem 打头，代表 memory，起源于 ANSI C 标准，由任何支持 ANSI C 库的系统提供。

我们首先给出源自 Berkeley 的函数，本书中我们仅用了一个即 bzero。(我们使用它是因为它只有两个参数，比起 3 个参数的 memset 函数来，要容易记些，这在前边已解释过。)其他两个函数：bcopy 和 bcmp，在别的应用程序中可能会见到。

```
#include <strings.h>
void bzero(void *dest, size_t nbytes);
void bcopy (const void *src,void *dest, size_t nbytes);
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
                                        返回：0——相等，非 0——不相等
```

这是我们首次遇到 ANSI C 的 const 限定词。就它在这儿的三处使用来说，它表示所限定的指针(src、ptr1 和 ptr2)所指的内容是不能由函数改动的。换句话说，函数只读而不修改由 const 指针所指的内存单元。

bzero 将目标中指定数目的字节置为 0，我们经常用此函数来把套接口地址结构初始化为 0；bcopy 将指定数目的字节从源移到目标；bcmp 比较任意两个字节串，若相同则返回值为 0，否则返回值为非 0。

下面的函数是 ANSI C 函数：

```
#include <string.h>
void *memset(void *dest, int c, size_t len);
void *memcpy(void *dest, const void *src, size_t nbytes);
int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
                                        返回：0——相同，>0 或<0——不相同
```

memset 将目标中指定数目的字节置为值 c。memcpy 与 bcopy 类似，但交换了两个指针参数的顺序，源与目标重迭时，bcopy 能正确处理，memcpy 的操作结果则是不可知的。这时，必须使用 ANSI C 的 memmove 函数(习题 30.3)。

有一个方法可记住 memcpy 两个指针的顺序：它们是按跟 C 的赋值语句相同的顺序从左到右书写的：

```
dest = src;
```

有一个方法可记住 memset 最后两个参数的顺序：所有的 ANSI C 的 memXXX 函数都要求有一个长度参数，且它总是最后一个参数。

memcmp 比较任意两个字节串，如果相同则返回 0，如果不同则返回一个非 0 值，具体是大于 0 还是小于 0 则取决于第一个不等的字节：若 ptr1 所指字节大于 ptr2 所指字节，则大于 0，否则小于 0。进行比较操作时，假定两个不等的字节均为无符号字符(unsigned char)。

3.6 inet_aton、inet_addr 和 inet_ntoa 函数

在本节和下一节，我们介绍两组地址转换函数，它们在 ASCII 字符串(人们比较喜欢用的格式)与网络字节序的二进制值(此值存于套接口地址结构中)间转换地址。

1. inet_aton、inet_addr 和 inet_ntoa 在点分十进制数串(例如，“206.62.226.33”)与它的 32 位网络字节序二进制值间转换 IPv4 地址，你在许多代码中可能会遇到这些函数。
2. 两个较新的函数：inet_pton 和 inet_ntop 对 IPv4 和 IPv6 地址都能处理，我们将在下一节对它们进行介绍并纵贯全书使用。

```
#include <arpa/inet.h>
int inet_aton(const char *strptr, struct in_addr *addrptr);
                                        返回:1——串有效,0——串有错
in_addr_t inet_addr(const char *strptr);
  返回:若成功,返回 32 位二进制的网络字节序地址;若有错,则返回 INADDR_NONE
char *inet_ntoa(struct in_addr inaddr);
                                        返回:指向点分十进制数串的指针
```

第一个函数 inet_aton 将 strptr 所指的 C 字符串转换成 32 位的网络字节序二进制值，并通过指针 addrptr 来存储。如果成功返回 1，否则返回 0。

> 函数 inet_aton 有一个没写到正式文档中的特征：如果指针为空，则函数仍然执行输入串的有效性检查，但不存储任何结果。

inet_addr 进行相同的转换，返回值为 32 位的网络字节序二进制值。这个函数存在这样的问题：所有 2^{32} 个可能的二进制值都是有效的 IP 地址(从 0.0.0.0 到 255.255.255.255)，但当出错时返回一个常值 INADDR_NONE(一般为一个 32 位均为 1 的值)。这就意味着点分十进制数串 255.255.255.255(这是 IPv4 的有限广播地址，见 18.2 节)不能由此函数处理，因为它的二进制值被用来指示函数失败。

> 当我们进行 ping 255.255.255.255 这样的操作时，许多较早期的 Ping 都返回一个“unknown host(不可知主机)”这样的错误，原因就是函数 inet_addr 失败，于是就将此点分十进制串作为一个主机名来查找，同样失败。
>
> 函数 inet_addr 还有一个潜在的问题，有些非正式的文档把出错时的返回值定义为 −1 而不是 INADDR_NONE。这样比较函数的返回值(无符号的值)与负常值时可能会出问题，这取决于 C 编译器。

现在，人们都反对使用函数 inet_addr，而用函数 inet_aton 来代替。更好的办法是用下一节介绍的新函数，它们对 IPv4 和 IPv6 都能处理。

函数 inet_ntoa 将一个 32 位的网络字节序二进制 IPv4 地址转换成相应的点分十进制数串。由函数返回值所指的串驻留在静态内存中，这意味着函数是不可重入的，这在 11.14 节我们再进行讨论。最后注意一下，这个函数以结构为参数，而不是指向结构的指针。

> 函数以结构为参数是很少见的，更多的是以指向结构的指针为参数。

3.7　inet_pton 和 inet_ntop 函数

这两个函数较新，对 IPv4 和 IPv6 地址都能处理，本书通篇都使用这两个函数。字母 p 和 n 分别代表 presentation 和 numeric。地址的表达(presentation)格式通常是 ASCII 串，数值(numeric)格式则是存在于套接口地址结构中的二进制值。

```
#include <arpa/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr);
                返回:1——成功,0——输入不是有效的表达格式,-1——出错
const char *inet_ntop(int family ,const void *addrptr ,char *strptr, size_t len);
                返回:指向结果的指针——成功,NULL——出错
```

两个函数的参数 family 既可以是 AF_INET,也可以是 AF_INET6。如果以不被支持的地址族作为 family 参数,两个函数都返回错误,并将 errno 置为 EAFNOSUPPORT。

第一个函数转换由指针 strptr 所指的串,通过指针 addrptr 存储二进制结果。如果成功,则返回值为 1;如果对于指定的 family 输入串不是有效的表达格式,则返回值为 0。

inet_ntop 进行相反的转换,即从数值格式(addrptr)到表达格式(strptr)进行转换。参数 len 是目标的大小,以免函数溢出其调用者的缓冲区。为有助于规定这个大小,在头文件<netinet/in.h>中有如下定义:

```
#define INET_ADDRSTRLEN     16    /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN    46    /* for IPv6 hex string */
```

如果 len 太小,无法容纳表达格式结果(包括终止的空字符),则返回一个空指针,并置 errno 为 ENOSPC。

函数 inet_ntop 的参数 strptr 可不能是个空指针,调用者必须为目标分配内存并指定大小。成功时,此指针即函数的返回值。

图 3.10 总结了这一节和上一节我们所讨论的 5 个函数。

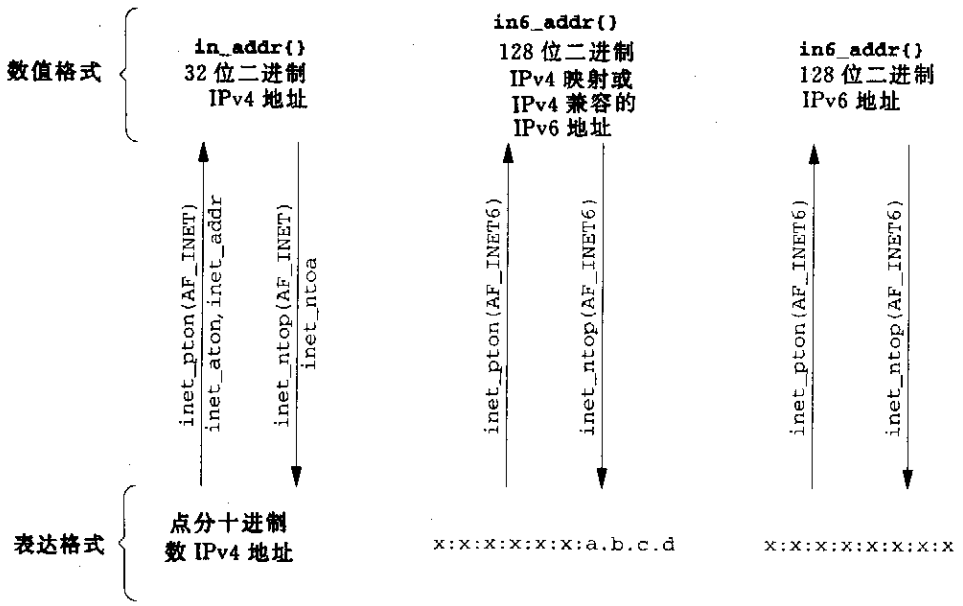


图 3.10 地址转换函数小结

举例

即使你的系统还不支持 IPv6，你可以采取下列措施开始使用这些新函数，即用代码：

```
inet_pton(AF_INET, cp, &foo.sin_addr);
```

代替代码：

```
foo.sin_addr.s_addr = inet_addr(cp);
```

再用代码：

```
char str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET,&foo.sin_addr,str,sizeof(str));
```

代替代码：

```
ptr = inet_ntoa(foo.sin_addr);
```

图 3.11 给出了只支持 IPv4 的 inet_pton 函数的简单定义。类似地，图 3.12 给出了只支持 IPv4 的 inet_ntop 函数的简化版本。

```
10 int
11 inet_pton(int family,const char *strptr,void *addrptr)
12 {
13     if(family == AF_INET) {
14         struct in_addr in_val;
15         if(inet_aton(strptr,&in_val)) {
16             memcpy(addrptr,&in_val,sizeof(struct in_addr));
17             return(1);
18         }
19         return(0);
20     }
21     errno = EAFNOSUPPORT;
22     return (-1);
23 }
```

图 3.11　仅支持 IPv4 的 inet_pton 简化版本[libfree/inet_pton_ipv4.c]

```
 8 const char *
 9 inet_ntop(int family,const void *addrptr,char *strptr,size_t len)
10 {
11     const u_char *p = (const u_char *) addrptr;
12     if(family == AF_INET) {
13         char    temp[INET_ADDRESTRLEN];
14         snprintf(temp,sizeof(temp),"%d.%d.%d.%d",
15                 p[0],p[1],p[2],p[3]);
16         if(strlen(temp) >= len) {
17             errno = ENOSPC;
18             return (NULL);
19         }
20         strcpy(strptr,temp);
21         return(strptr);
22     }
```

```
23      errno = EAFNOSUPPORT;
24      return (NULL);
25  }
```

图 3.12 仅支持 IPv4 的 inet_ntop 简化版本[libfree/inet_ntop_ipv4.c]

3.8 sock_ntop 和相关函数

inet_ntop 的一个基本问题是:它要求调用者传递一个指向二进制地址的指针,而此地址一般是包含在套接口地址结构中的,这就要求调用者必须知道结构的格式和地址族。这就是说,为使用这个函数,我们必须为 IPv4 写如下的代码:

```
struct sockaddr_in addr;
inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));
```

或为 IPv6 写如下的代码:

```
struct sockaddr_in6 addr6;
inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

这就使得我们的代码与协议相关了。

为了解决这个问题,我们自己写一个函数 sock_ntop,它取得作为参数传递的指向套接口地址结构的指针后,查看结构的内部,再调用适当的函数来返回地址的表达格式。

```
#include "unp.h"
char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);
                                    返回:非空指针——成功,NULL——出错
```

本书通篇使用的我们自己定义的函数(不是标准的系统函数)都有这样的表现形式:包围函数原型和返回值的方框是实线,开头包括的头文件一般是 unp.h。

sockaddr 指向一个长度为 addrlen 的套接口地址结构。本函数用它自己的静态缓冲区来保存结果,一个指向此缓冲区的指针即为返回值。

注意:对结果进行静态存储阻碍了函数的可重入与线程安全性。对此,我们在 11.14 节中将作进一步讨论。我们对此函数作这样的设计决策是为了在本书的简单例子中方便地调用它。

表达格式是:IPv4 地址的点分十进制数或 IPv6 地址的十六进制数串,后跟一个终止符(我们使用一个点号,与 netstat 类似),再跟一个十进制的端口号,最后跟一个空字符。因此,缓冲区大小至少是:IPv4 时为 INET_ADDRSTRLEN 加上 6 字节(16+6=22),IPv6 时为 INET6_ADDRSTRLEN 加上 6 字节(46+6=52)。

图 3.13 中我们给出了仅为 AF_INET 情形下的源代码。

```
5 char *
6 sock_ntop(const struct sockaddr *sa, socklen_t salen)
7 {
8   char        portstr[7];
9   static      char str[128];          /* Unix domain is largest */
10  switch (sa->sa_family) {
11  case AF_INET:{
12          struct sockaddr_in *sin = (struct sockaddr_in *) sa;

13          if(inet_ntop(AF_INET, &sin->sin_addr,str,sizeof(str)) == NULL)
14            return (NULL);
15          if(ntohs(sin->sin_port) != 0) {
16            snprintf(portstr,sizeof(portstr),".%d",ntohs(sin->sin_port));
17            strcat(str,portstr);
18          }
19          return (str);
20      }
```

图 3.13 sock_ntop 函数[lib/sock_ntop.c]

我们还为操作套接口地址结构定义了几个其他的函数，它们将简化我们的代码在 IPv4 与 IPv6 间的移植。

```
#include "unp.h"
int sock_bind_wild(int sockfd, int family);
                                                    返回：0——成功，-1——出错
int sock_cmp_addr(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2,socklen_t addrlen);
                                                    返回：0——地址同族且相等，否则非0
int sock_cmp_port(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2, socklen_t addrlen);
                                                    返回：0——地址同族且端口相同，否则非0
int sock_get_port(const struct sockaddr *sockaddr, socklen_t addrlen);
                                    返回：非负端口号 sockaddr 为 IPv4 或 IPv6 地址族，否则-1
char *sock_ntop_host(const struct sockaddr *sockaddr, socklen_t addrlen);
                                                    返回：非空指针——成功，空指针——出错
void sock_set_addr(const struct sockaddr *sockaddr, socklen_t addrlen,void *ptr);
void sock_set_port(const struct sockaddr *sockaddr, socklen_t addrlen,int port);
void sock_set_wild(struct sockaddr *sockaddr, socklen_t addrlen);
```

sock_bind_wild 捆绑通配地址和一个临时端口到一个套接口；sock_cmp_addr 比较两个套接口地址结构的地址部分；sock_cmp_port 比较两个套接口地址结构的端口部分；sock_get_port 返回端口号；sock_ntop_host 将套接口地址结构中的主机部分转换成表达格式(不包括端口号)；sock_set_addr 将套接口地址结构中的地址部分置为指针 ptr 所指的值；sock_set_port 只设置套接口地址结构的端口号；sock_set_wild 将套接口地址结构中的地址

部分置为通配地址。跟本书所有函数一样，我们也为那些返回值不是 void 的上述函数提供了包裹函数，它们的名字以 S 打头，我们的程序通常调用这些包裹函数。这里我们就不给出所有这些函数的源代码了，它们是免费提供的(见前言)。

3.9 readn、writen 和 readline 函数

字节流套接口(如 TCP 套接口)上的 read 和 write 函数所表现的行为不同于通常的文件 I/O。字节流套接口上的读或写输入或输出的字节数可能比要求的数量少，但这不是错误状况，原因是内核中套接口的缓冲区可能已达到了极限。此时所需的是调用者再次调用 read 或 write 函数，以输入或输出剩余的字节。(有些版本的 Unix 在往一个管道中写多于 4096 字节的数据时也会表现这样的行为)，这种情况在读字节流套接口时很常见，但在写字节流套接口时只能在套接口非堵塞的情况下才出现。然而，为预防实现万一返回不足的字节计数值，我们总是调用 writen 函数而不是 write 函数。

当我们对字节流套接口进行读或写操作时，调用下面的三个函数。

```
#include "unp.h"
ssize_t readn(int filedes, void *buff, size_t nbytes);
ssize_t writen(int filedes, const void *buff, size_t nbytes);
ssize_t readline(int filedes, void *buff, size_t maxlen);
                                        均返回：读写字节数，-1——出错
```

图 3.14 给出了 readn 函数，图 3.15 给出了 writen 函数，图 3.16 给出了 readline 函数。

```
1 #include        "unp.h"
2 ssize_t         /* Read "n" bytes from a descriptor. */
3 readn(int fd, void *vptr, size_t n)
4 {
5     ssize_t nleft;
6     ssize_t nread;
7     char    *ptr;
8     ptr = vptr;
9     nleft = n;
10    while (nleft > 0) {
11        if( (nread = read(fd, ptr, nleft)) < 0) {
12            if (errno == EINTR)
13                nread = 0;          /* and call read() again */
14            else
15                return (-1);
16        } else if (nread == 0)
17            break;              /* EOF */
18        nleft -= nread;
19        ptr += nread;
20    }
21    return (n - nleft);         /* return >= 0 */
22 }
```

图 3.14 readn 函数：从一个描述字读 n 字节[lib/readn.c]

```
1 #include       "unp.h"
2 ssize_t                /* Write "n" bytes to a descriptor. */
3 writen(int fd,const void *vptr,size_r n)
4 }
5         size_t nleft;
6         ssize_t nwritten;
7         const char *ptr;
8         ptr = vptr;
9         nleft = n;
10        while (nleft > 0) {
11            if( (nwritten = write(fd,ptr,nleft)) <= 0) {
12                if(errno == EINTR)
13                    nwritten = 0;     /* and call write() again */
14                else
15                    return (-1);      /* error */
16            }
17            nleft -= nwritten;
18            ptr += nwritten;
19    }
20    return (n);
21 }
```

图 3.15　writen 函数：往一个描述字写 n 字节[lib/writen.c]

```
1 #include     "unp.h"
2 ssize_t
3 readline(int fd,void *vptr,size_t maxlen)
4 {
5     ssize_t n,rc;
6     char    c,*ptr;
7     ptr = vptr;
8     for (n = 1; n < maxlen; n++) {
9       again:
10        if ( (rc = read(fd,&c,1)) == 1) {
11            *ptr++ = c;
12            if(c == '\n')
13                break;          /* newline is stored,like fgets() */
14            } else if (rc == 0) {
15                if (n == 1)
16                    return (0); /* EOF,no data read */
17                else
18                    break;      /* EOF,some data was read */
19        } else {
20            if(errno == EINTR)
21                goto again;
22            return(-1);         /* error,errno set by read() */
23        }
24    }
25    *ptr = 0;                   /* null terminate like fgets() */
26    return (n);
27 }
```

图 3.16　readline 函数：从一个描述字读文本行，一次 1 个字节[test/readline1.c]

上述三个函数查找错误EINTR(系统调用被一个捕获的信号中断,我们在5.9节将进行更详细的讨论),如果发生这种错误则继续进行读或写操作。由于这些函数的作用是避免让调用者来处理不足的字节计数值,所以我们就地处理错误,而不是强制调用者再次调用readn或writen函数。

在13.3节我们会提到,标志MSG_WAITALL可随函数recv一起使用来代替独立的readn函数。

注意,函数readline每读一个字节的数据就要调用一次系统的read函数,其效率是非常低的,如APUE的3.9节所示。我们宁愿调用函数read来取得尽可能多的数据,将其缓冲起来,然后在缓冲区中每次检查一个字节。这可以通过标准I/O库来实现,如13.8节中所述。

图3.17给出了函数readline的一个较快速版本,它每次最多读MAXLINE个字节,然后一次返回一个字符。

```
1 #include    "unp.h"
2 static ssize_t
3 my_read(int fd,char *ptr)
4 {
5     static int read_cnt = 0;
6     static char *read_ptr;
7     static char read_buf(MAXLINE);
8     if (read_cnt <= 0) {
9         again:
10          if((read_cnt = read(fd,read_buf,sizeof(read_buf))) < 0) {
11              if(errno == EINTR)
12                  goto again;
13              return (-1);
14          } else if(read_cnt == 0)
15              return (0);
16          read_ptr = read_buf;
17      }
18      read_cnt--;
19      *ptr = *read_ptr++;
20      return (1);
21 }
22 ssize_t
23 readline(int fd,void *vptr,size_t maxlen)
24 {
25      int        n,rc;
26      char       c,*ptr;
27      ptr = vptr;
28      for (n = 1; n < maxlen; n++) {
29          if( (rc = my_read(fd,&c)) ==1 ) {
30              *ptr++ = c;
31              if(c == '\n')
32                  break;          /* newline is stored,like fgets() */
33          } else if (rc == 0) {
34              if (n == 1)
35                  return (0);     /* EOF,no data read */
36              else
37                  break;          /* EOF,some data was read */
38              } else
```

```
39                  return (-1); /* error, errno set by read() */
40      }
41      *ptr = 0;               /* null terminate like fgets() */
42      return (n);
43  }
```

图 3.17　函数 readline 的改进版[lib/readline.c]

第 2～21 行　内部函数 my_read 每次最多读 MAXLINE 个字符，然后每次返回一个字符。

第 29 行　函数 readline 本身的唯一变化是调用函数 my_read 而不是 read。

对函数 readline 做的这个小小的修改使得情况大为改观。如果我们测量一下新老版本读一个 2781 行的文件(135 816 字节)所需时间就会发现，老版本需要 8.8 秒，而新版本只需 0.3 秒。几乎所有的区别都由内核中所耗时间即系统时间引起，老版本执行了 135 816 次系统调用，而新版本仅进行了 34 次系统调用(135 816 除以 MAXLINE 是 4096)。

> 不幸的是，在函数 my_read 中用静态变量来维护会跨越连续调用的状态信息，因而不再是可重入或线程安全的了。在 11.14 节和 23.5 节中，我们将讨论这一点，在图 23.11 中，我们用特定于线程的数据开发了一个此函数的线程安全版本。

3.10　isfdtype 函数

很多时候，我们需要测试一个描述字是不是某给定类型。早些时候，这是通过调用 Posix.1g 的函数 fstat，然后用 S_ISxxx 的某个宏来测试返回的 st_mode 值来实现的。有许多实现，但不是所有的实现，定义了宏 S_ISSOCK 来测试一个描述字是否为套接口描述字。由于有些实现仅靠函数 fstat 的返回信息不能判定一个描述字是否为套接口描述字，因而 Posix.1g 提供了新函数 isfdtype。

```
#include <sys/stat.h>
int isfdtype(int fd, int fdtype);
                    返回：1——是指定类型，0——不是指定类型，-1——出错
```

为了测试是否为套接口描述字，fdtype 应设成 S_IFSOCK。此函数的一个应用是：在一个由另外一个程序调用 exec 执行的程序中(4.7 节)，测试某描述字是否为一个套接口描述字。

图 3.18 给出了这个函数的一个实现例子(假设该实现支持 fstat 返回的 S_IFSOCK 模式)。

```
1 #include    "unp.h"
2 #ifndef S_IFSOCK
3 #error S_IFSOCK not defined
4 #endif
5 int
```

```
 6 lsfdtype(int fd,int fdtype)
 7 {
 8         struct stat buf;
 9         if (fstat(fd,&buf) < 0)
10             return (-1);
11         if ((buf.st_mode & S_IFMT) == fdtype)
12             return (1);
13     else
14             return (0);
15 }
```

图 3.18 用 fstat 实现的 isfdtype[lib/isfdtype.c]

在头文件<sys/stat.h>中定义了大量的 S_IFxxx 常值，我们的实现也认可它们。然而，Posix.1g 只说明当 fdtype 是 S_IFSOCK 时，该函数有效。

3.11 小 结

套接口地址结构是每个网络程序的重要组成部分，我们分配它们，填写它们，把指向它们的指针传递给各种套接口函数，有时我们把指向某结构的指针传递给套接口函数并由它们填写其内容。我们总是通过指针来传递这些结构(也就是说，我们传递的是指向结构的指针，而不是结构本身)，而且将结构的大小作为另外一个参数来传递。当套接口函数填写结构时，长度也作为指针传递，所以它的值可以被函数更新，我们把这样的参数称为值-结果参数。

套接口地址结构总是以一个标识含在结构中的地址族的成员("族")开头，所以套接口地址结构是自定义的。支持变长套接口地址结构的较新实现在开头还包含一个长度成员，它含有整个结构的长度信息。

在表达格式(我们平时所写的格式，如 ASCII 字符)与数值格式(套接口地址结构中的格式)间转换 IP 地址的两个函数是：inet_pton 和 inet_ntop。虽然在后面的章节中我们要使用这两个函数，但必须说明，它们是协议相关的。一个较好的技术是：把套接口地址结构作为不透明对象来操作，仅需知道指向结构的指针和它的大小，为此我们开发了一组 sock_函数来帮助我们的程序做到协议无关。在第 11 章，我们用函数 getaddrinfo 和 getnameinfo 完成协议无关工具的开发。

TCP 套接口为应用进程提供了一个字节流，它们没有记录标记。从函数 read 的返回值可以比我们要求的数量少，但这不表示错误。我们开发了 3 个函数：readn、writen 和 readline 来协助对字节流的读或写，这 3 个函数广泛应用于全书。

3.12　习　题

3.1　为什么诸如套接口地址结构的长度这样的值-结果参数要用指针来传递？

3.2　为什么函数 readn 和 writen 都将 void * 型指针转换为 char * 型指针？

3.3　函数 inet_aton 和 inet_addr 对于接受什么为点分十进制数 IPv4 地址串一直是相当随意的：允许是由小数点分隔的 1～4 个数，允许一个前导的 0x 来表示是一个十六进制数，还允许一个前导的 0 来表示是一个八进制数。（尝试用 telnet 0xe 来检验一下这些特性。）函数 inet_pton 对 IPv4 地址的要求则要严格得多，它明确要求用三个小数点来分隔四个数，每个数都是 0～255 之间的十进制数。当地址族为 AF_INET6 时，函数 inet_pton 不允许指定点分十进制数地址。对这一点，可能有人要争辩：这种做法应该是允许的，返回值就是对应这个点分十进制数串的 IPv4 映射的 IPv6 地址（图 A.10）。试写一个名为 inet_pton_loose 的函数，它能处理这样的情况：如果地址族是 AF_INET 且函数 inet_pton 返回 0，则调用函数 inet_aton 并看它是否成功；类似地，如果地址族是 AF_INET6 且函数 inet_pton 返回 0，则调用函数 inet_aton 且看它是否成功，若成功则返回其 IPv4 映射的 IPv6 地址。

第4章 基本TCP套接口编程

4.1 概 述

本章阐述编写一个完整的TCP客户和服务器程序所需要的基本套接口函数。我们首先对即将使用的所有基本套接口函数进行阐述，在下一章再开发客户和服务器程序。你会发现，全书都围绕着此客户和服务器程序，并多次改进与完善(图1.12和图1.13)。

我们也对并发服务器进行阐述，这是一项常用的Unix技术，在同时有大量的客户连接到同一个服务器上时提供并发性。每个客户连接都迫使服务器为它派生(fork)一个新进程。本章我们只考虑利用fork的每客户单进程模型，在第23章我们讨论线程时，将考虑另外一种模型，即每客户单线程。

图4.1为一个TCP客户与服务器间发生的一些典型事件的时间表。首先启动服务器，稍后的某个时刻启动客户，它要连接到此服务器上。我们假设客户给服务器发了一个请求，服务器处理这个请求，并且给客户发回一个响应。这个过程一直持续下去，直到客户给服务器发一个文件结束符，并关闭客户端连接，接着服务器也关闭服务器端的连接，或结束运行或等待一个新的客户连接。

4.2 socket函数

为了执行网络I/O，一个进程必须做的第一件事情就是调用socket函数，指定期望的通信协议类型(使用IPv4的TCP、使用IPv6的UDP、Unix域字节流协议等)。

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
                                返回：非负描述字——成功，-1——出错
```

代码中的family指明协议族，它是图4.2中所示的某个常值。套接口的类型type是图4.3中所示的某个常值。一般来说，函数socket的参数protocol设置为0，除非用在原始套接口上，这一点我们将在第25章中再讨论。

并非所有套接口family与type的组合都是有效的，图4.4给出了一些有效的组合和对应的真正协议。其中标为“Yes”的项也是有效的，但还没有找到便捷的缩略词；而空白项则是不支持的。

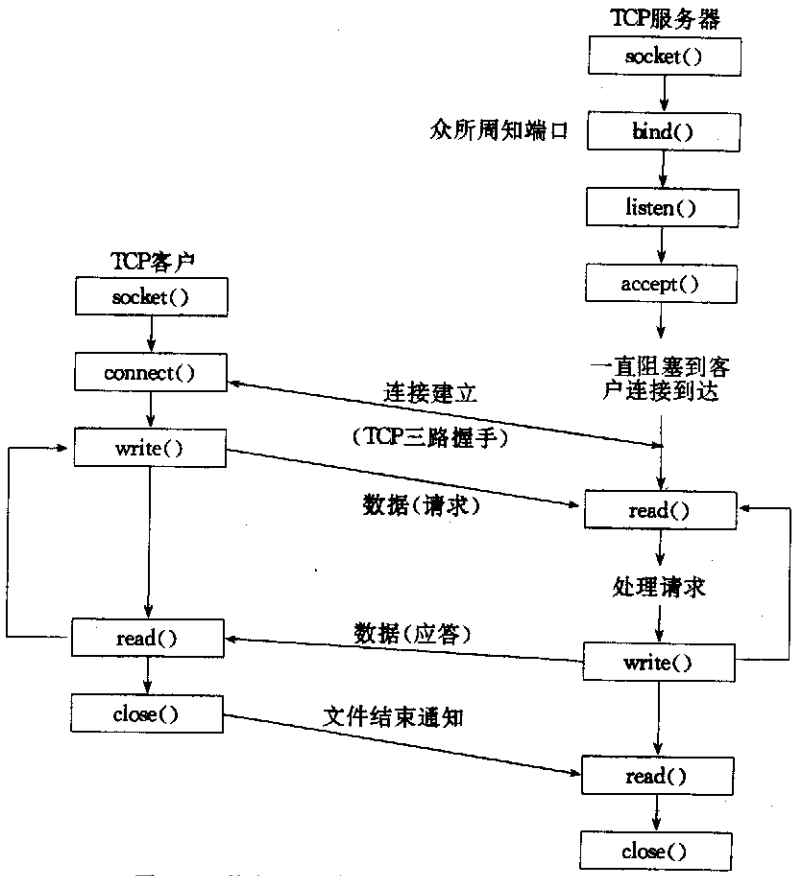


图 4.1　基本 TCP 客户-服务器程序的套接口函数

族	解释
AF_INET	IPv4 协议
AF_INET6	IPv6 协议
AF_LOCAL	Unix 域协议(第 14 章)
AF_ROUTE	路由套接口(第 17 章)
AF_KEY	密钥套接口

图 4.2　socket 函数的协议族(family)常值

类型	解释
SOCK_STREAM	字节流套接口
SOCK_DGRAM	数据报套接口
SOCK_RAW	原始套接口

图 4.3　socket 函数的套接口类型(type)

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

图 4.4　socket 函数的族与类型的组合

你可能也会碰到作为 socket 函数第一个参数的相应的 PF_xxx 常值，我们在本节未讲述。

你也许会碰到 AF_UNIX（原来 Unix 的名字）而不是 AF_LOCAL（Posix. 1g 名字），在第 14 章我们再做详细说明。

参数 family 和 type 还有其他值，例如 4. 4BSD 对 AF_NS（Xerox NS 协议，常称为 XNS）和 AF_ISO（OSI 协议）都支持，但现在很少有人使用这些协议。类似地，Xerox NS 协议和 OSI 协议都实现了 SOCK_SEQPACKET 这种有序分组套接口类型，但是 TCP 是一个字节流，仅支持 SOCK_STREAM 套接口。

Linux 支持一个新的套接口类型：SOCK_PACKET，它与图 2.1 中的 BPF 和 DLPI 类似，支持对数据链路的访问，在第 26 章中我们再做详细说明。

密钥套接口 AF_KEY 是新出现的。IPv6 要求支持基于加密的安全性，许多系统也可能给 IPv4 提供这样的支持。跟路由套接口（AF_ROUTE）是与内核中路由表的接口方式类似，密钥套接口是与内核中密钥表的接口，这个协议族的初级文档见[McDonald，Phan，and Atkinson 1996]和[McDonald，Metz and Phan 1997]。

socket 函数在成功时返回一个小的非负整数值，它与文件描述字类似，我们把它称为套接口描述字（socket descriptor），简称套接字（sockfd）。为了得到这个套接口描述字，我们只是指定了协议族（IPv4、IPv6 或 Unix）和套接口类型（字节流、数据报或原始套接口）。我们并没有指定本地协议地址或远程协议地址。

AF_xxx 与 PF_xxx

AF_前缀代表地址族，PF_前缀代表协议族。历史上曾有这样的想法：单个协议族可以支持多个地址族，PF_值用来创建套接口，而 AF_值用于套接口地址结构。但实际上，支持多个地址族的协议族从来就未实现过，而且头文件<sys/socket. h>中为一给定协议定义的 PF_值总是与此协议的 AF_值相等。尽管这种相等关系并不保证永远正确，若有人试图给已有的协议改变这种约定，则许多现存代码都将崩溃。为与现存代码保持一致，本书中我们仅用 AF_常值（尽管在调用 socket 时，我们可能会碰到 PF_值）。

查看 BSD/OS2. 1 版中调用 socket 的 137 个程序，可以发现，有 143 个调用指定 AF_值，仅有 8 个调用指定 PF_值。

从历史上说，AF_前缀与 PF_前缀具有相似常值集的原因要追溯到4. 1cBSD [Lanciani 1996]和比我们正描述的要早些的 socket 函数版本。socket 的4. 1cBSD 版本采用了四个参数，其中有一个是指向 sockproto 结构的指针。该结构的第一个元素名为 sp_family，它的值是某一 PF_值；第二个元素即 sp_protocol 是一个协议号，与现行 socket 函数的第三个参数相似。指定协议族的唯一方法就是指定该结构，因此，在该早期系统中，PF_值用作在 sockproto 结构中指定协议族的结构标签，而 AF_值用作在套接口地址结构中指定地址族的结构标签。4. 4BSD 中仍有 sockproto 结构（TCPv2 第 626～627 页），但仅由内核内部使用。在最初的定义中，对元素 sp_family 有“protocol family（协议族）”的注释，在4. 4BSD源代码中已改为“address family（地址族）”了。

更让人混淆 AF_常值和 PF_常值区别的是，含有与 socket 函数的第一个参数进行比较的值的 Berkeley 内核数据结构(domain 结构的元素 dom_family，TCPv2 第 187 页)有这样的注释：它含有 AF_值。有些内核中的 domain 结构初始化为相应的 AF_值(TCPv2 第 192 页)，但有些则初始化成 PF_值(TCPv2 第 646 页和 TCPv3 第 229 页)。

另一个必须注意的历史是，1983 年 7 月的 4.2BSD 中 socket 函数的手册页面将其第一个参数称为 af，并列出了作为 AF_常值的可能值。

最后，我们注意到 Posix.1g 将 socket 函数的第一个参数指定为 PF_值，而 AF_值用于套接口地址结构，但它在结构 addrinfo(11.2 节)中又只定义了一个族值，试图让 socket 调用和套接口地址结构都能使用。

4.3　connect 函数

TCP 客户用 connect 函数来建立一个与 TCP 服务器的连接。

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
                                        返回：0——成功，-1——出错
```

sockfd 是由 socket 函数返回的套接口描述字，第二、第三个参数分别是一个指向套接口地址结构的指针和该结构的大小，如 3.3 节所述。套接口地址结构必须含有服务器的 IP 地址和端口号，图 1.5 中我们见过此函数的一个例子。

客户在调用函数 connect 前不必非得调用函数 bind(下一节我们再介绍此函数)，因为如果必要的话，内核会选择源 IP 地址和一个临时的端口。

如果是 TCP 套接口的话，函数 connect 激发 TCP 的三路握手过程(2.5 节)，且仅在连接建立成功或出错时才返回，返回的错误可能有以下几种情况：

1. 如果 TCP 客户没有收到 SYN 分节的响应，则返回 ETIMEDOUT。例如在 4.4BSD 中，当调用函数 connect 时，发出一个 SYN，若无响应，等待 6 秒钟后再发一个，若仍无响应，24 秒钟后再发一个(TCPv2 第 828 页)。若总共等了 75 秒钟之后仍未收到响应，则返回错误。

   有些系统提供对超时值的管理性控制，见 TCPv1 的附录 E。
2. 如果对客户的 SYN 的响应是 RST，则表明该服务器主机在我们指定的端口上没有进程在等待与之连接(例如服务器进程也许没有启动)，这称为硬错(hard error)，客户一接收到 RST，马上就返回错误 ECONNREFUSED。

   RST 的含义为“复位”，它是 TCP 在某些错误情况下所发的一种 TCP 分节。有三个条件可以产生 RST：SYN 到达某端口但此端口上没有正在监听的服务器(如前所述)、TCP 想取消一个已有连接、TCP 接收了一个根本不存在的连接上的分节。(TCPv1 第 246～250 页有更详细的信息。)
3. 如果某客户发出的 SYN 在中间的路由器上引发了一个目的地不可达 ICMP 错误，

这称为软错(soft error)。客户上的内核保存此消息,并按第一种情况中所述的时间间隔连续发出SYN。若在某规定的时间(4.4BSD规定75秒)后仍未收到响应,则把保存的消息(即ICMP错误)作为EHOSTUNREACH或ENETUNREACH错误返回给进程。

许多早期系统,如4.2BSD,在收到目的地不可达ICMP错误时就不正确地放弃建立连接的尝试,这是不正确的,因为此ICMP错误可能只是指示了某个暂时状态。例如,它可能是一个15秒内可以恢复的路由问题。

注意,即使ICMP错误指示目的网络不可达,图A.15中也没有列出ENETUNREACH。网络不可达的错误被认为已过时,即使4.4BSD收到此消息,它也是给应用进程返回EHOSTUNREACH错误。

用图1.5的简单客户程序例子,我们再来仔细看看上述几种不同的错误情况。首先指定本地主机(127.0.0.1),它在运行daytime服务器程序,我们观察正常的输出:

```
solaris % daytimetcpcli 127.0.0.1
Tue Jan 16 16:45:07 1996
```

为了看到返回响应的不同格式,我们指向本地Cisco路由器(见图1.16):

```
solaris % daytimetcpcli 206.62.226.62
Tuesday, May 7, 1996 11:01:33-MST
```

下面,我们将IP地址指向本地子网(206.62.226)但其主机ID(55)并不存在,即本地子网上没有一个主机ID为55的主机,当客户发出ARP请求(要求主机响应其硬件地址)时,将收不到ARP响应:

```
solaris % daytimetcpcli 206.62.226.55
connect error: Connection timed out
```

当函数connect超时后(Solaris 2.5规定此时间为3分钟),我们才得到此错误。函数err_sys输出与ETIMEDOUT相关的人们可读的字符串。

下一个例子中我们指向主机gateway,它是一个Cisco路由器,没有运行daytime服务器程序。

```
solaris % daytimetcpcli 140.252.1.4
connect error:Connection refused
```

服务器主机立刻响应一个RST。

最后一个例子,我们指向Internet中一个不可到达的IP地址。如果我们用tcpdump观察分组的情况,就会发现6跳以远的路由器返回了主机不可达的ICMP错误。

```
solaris % daytimetcpcli 192.3.4.5
connect error:No route to host
```

跟ETIMEDOUT错误一样,本例中的connect也在等待规定的一段时间之后才返回EHOSTUNREACH错误。

根据TCP状态转换图(图2.4),函数connect导致从CLOSED状态(调用函数socket创建套接口后就一直处于此状态)转到SYN_SENT状态,若成功再转到ESTABLISHED状

态。如果函数 connect 失败，则套接口不可再用，必须关闭，不能再对此套接口再调用函数 connect。在图 11.6 中我们将看到，当循环调用函数 connect，以尝试给定主机的每个 IP 地址直到有一个成功时，每当函数 connect 失败，都必须关闭套接口描述字，重新调用 socket。

4.4　bind 函数

函数 bind 给套接口分配一个本地协议地址，对于网际协议，协议地址是 32 位 IPv4 地址或 128 位 IPv6 地址与 16 位的 TCP 或 UDP 端口号的组合。

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
                                                返回：0——成功，-1——出错
```

历史上对函数 bind 的手册页面描述是这样的：函数 bind 为一个无名的套接口命名。“名字”的意义比较含糊，它可能意味诸如 foo.bar.com 之类的域名（第 9 章）。实际上，函数 bind 与名字没有关系。它仅仅给套接口分配一个协议地址，至于协议地址的含义则要取决于协议本身①。

第二个参数是一个指向特定于协议的地址结构的指针，第三个参数是该地址结构的长度。对于 TCP，调用函数 bind 可以指定一个端口号，指定一个 IP 地址，可以两者都指定，也可以一个也不指定。

- 当服务器启动时，要捆绑众所周知端口，这在图 1.9 中我们已看到了。如果 TCP 客户或服务器不这么做，当调用函数 connect 或 listen 时，内核就要为套接口选择一个临时端口。让内核来选择临时端口，这对 TCP 客户来说是正常的，除非应用要求一个预留端口；但对 TCP 服务器来说是极少见的，因为服务器是通过它们的众所周知端口被大家认识的。

  这个规则的例外是 RPC（远程过程调用）服务器，它们一般是让内核来为它们的监听套接口选择一个临时端口，因为这个端口紧接着就通过 RPC 端口映射器进行注册。客户在与服务器连接之前，必须与端口映射器联系以获取服务器

① 译者注：捆绑（binding）操作涉及三个对象：套接口（在 XTI API 中为端点）、地址及端口。其中套接口是捆绑的主体，地址和端口则是捆绑在套接口上的客体。由于涉及对象较多，我们先在这里澄清各种说法：

1. “捆绑地址 A 和/或端口 P 到套接口 S”。同义的说法还有：“把地址 A 和/或端口 P 捆绑到套接口 S”，“给套接口 S 捆绑地址 A 和/或端口 P”，等等。
2. “跟端口 P（地址 A）一块捆绑地址 A（端口 P）”。

绑定（bound）表示捆绑成功后的状态，它的各种说法如下：

1. “绑定地址 A 和/或端口 P 的套接口”。
2. “套接口 S 上绑定的地址或端口”。
3. “已绑定的地址或端口”。也就是说该地址或端口已为某个套接口所用。
4. “跟端口 P（地址 A）一块绑定的地址（端口）”。
5. “套接口 S 已绑定”。相反的说法是“套接口 S 未绑定”。

的临时端口。这种情况也适用于使用 UDP 的 RPC 服务器。

- 进程可以把一个特定的 IP 地址捆绑到它的套接口上，但此 IP 地址必须是主机的一个接口。对于 TCP 客户，这就为在此套接口上发送的 IP 数据报分配了源 IP 地址。对于 TCP 服务器，这就限制了套接口只接收那些目的地为此 IP 地址的客户连接。TCP 客户一般不把 IP 地址捆绑到它的套接口上。当连接套接口时，由内核根据所用的输出接口来选择源 IP 地址，而所用的输出接口则取决于到达服务器的路径(TCPv2 第 737 页)。

  如果 TCP 服务器不把 IP 地址捆绑到套接口上，内核就把客户所发 SYN 所在分组的目的 IP 地址作为服务器的源 IP 地址(TCPv2 第 943 页)。

正如我们所说的，通过调用函数 bind 可以指定 IP 地址或端口，可以都指定，也可以都不指定。根据期望的结果，对 sin_addr 和 sin_port，或 sin6_addr 和 sin6_port 应置为什么值，图 4.5 作了总结。

进程指定		结果
IP 地址	端口	
通配地址	0	内核选择 IP 地址和端口
通配地址	非 0	内核选择 IP 地址，进程指定端口
本地 IP 地址	0	进程指定 IP 地址，内核选择端口
本地 IP 地址	非 0	进程指定 IP 地址和端口

图 4.5 给函数 bind 指定用于捆绑的 IP 地址和/或端口号的结果

若指定端口号为 0，调用函数 bind 时，内核选择一个临时端口；但若指定一个通配 IP 地址，则直到套接口已连接(TCP)或数据报已在套接口上发出(UDP)，内核才选择一个本地 IP 地址。

对于 IPv4 来说，通配地址由常值 INADDR_ANY 来指定，其值一般为 0，它通知内核选择 IP 地址。在图 1.9 中我们已看到了如下的赋值用法：

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);        /* wildcard */
```

这对 IPv4 是可行的，因为其地址是 32 位的值，可以用一个简单的数字常值来表示(这种情况下为 0)，但在 IPv6 中，我们就不能利用这项技术，因为 128 位的 IPv6 地址是保存在结构中的。(在 C 语言中，赋值语句的右边无法表示常值结构。)为解决此问题，我们编写如下代码：

```
struct sockaddr_in6  serv;
serv.sin6_addr = in6addr_any;      /* wildcard */
```

系统分配变量 in6addr_any 并将其初始化为常值 IN6ADDR_ANY_INIT，头文件<netinet/in.h>中含有 in6addr_any 的 extern 声明。

INADDR_ANY 的值(为 0)无论是网络字节序还是主机字节序都相同，所以不必使用 htonl。但既然头文件<netinet/in.h>定义的所有 INADDR_常值都是主机字节序，我们应该

使用 htonl。

如果让内核来为套接口选择一个临时端口号，那么必须注意，函数 bind 并不返回所选择的值。实际上，由于函数 bind 的第二个参数有 const 限定词，它无法返回所选之值。为了得到内核所选择的这个临时端口值，必须调用函数 getsockname 来返回协议地址。

进程捆绑非通配 IP 地址到套接口上的常见例子是在为多个组织提供 Web 服务器的主机上(TCPv3 的 14.2 节)。首先，每个组织都有它自己的域名，就像这样的形式：www.organization.com。其次，每个组织的域名都映射为不同的 IP 地址，但在相同的子网上。例如，子网为 198.69.10，第一个组织的 IP 地址是 198.69.10.128，第二个是 198.69.10.129，等等。然后，所有这些 IP 地址都定义成同一个网络接口的别名(例如，4.4BSD 中使用 ifconfig 命令的 alias 选项来定义)，这样，IP 层将接收所有目的地为某个别名地址的数据报。最后，为每个组织启动一个 HTTP 服务器的副本，每个副本仅捆绑自己组织的 IP 地址。

> 另有一项技术是运行单个服务器但捆绑通配地址。当一个连接请求到达时，服务器调用函数 getsockname 获得来自客户的目的 IP 地址，这在我们上面的讨论中，可以是 198.69.10.128，198.69.10.129 等等。然后，服务器根据这个 IP 地址来处理客户请求。
>
> 捆绑非通配 IP 地址的好处是：由内核将给定的目的 IP 地址解复用后送往服务器进程。
>
> 对于分组到达的接口和分组的目的 IP 地址这两个概念，我们必须仔细区别。在 8.8 节，我们将讨论弱端系统模型和强端系统模型。大多数实现都采用前者，其含义是：一个分组只要其目的 IP 地址能够标识目的主机的某个接口就行，不必一定是它的到达接口。(假设是多宿主机。)捆绑非通配 IP 地址仅根据目的 IP 地址来限制递送给套接口的数据报，对到达接口未做任何限制，除非主机采用强端系统模型。

从函数 bind 返回的一个常见错误是 EADDRINUSE(地址已使用)，对这一点，到 7.5 节我们讨论套接口选项 SO_REUSEADDR 和 SO_REUSEPORT 时再做详细说明。

4.5　listen 函数

函数 listen 仅被 TCP 服务器调用，它做两件事情：

1. 当函数 socket 创建一个套接口时，它被假设为一个主动套接口，也就是说，它是一个将调用 connect 发起连接的客户套接口，函数 listen 将未连接的套接口转换成被动套接口，指示内核应接受指向此套接口的连接请求。根据 TCP 状态转换图(图 2.4)，调用函数 listen 导致套接口从 CLOSED 状态转换到 LISTEN 状态。
2. 函数的第二个参数规定了内核为此套接口排队的最大连接个数。

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
                                        返回：0——成功，-1——出错
```

一般说来，此函数应在调用函数 socket 和 bind 之后，调用函数 accept 之前调用。

为了理解参数 backlog，我们必须明白，对于给定的监听套接口，内核要维护两个队列：

1. 未完成连接队列(incomplete connection queue)，为每个这样的 SYN 分节开设一个条目：已由客户发出并到达服务器，服务器正在等待完成相应的 TCP 三路握手过程。这些套接口都处于 SYN_RCVD 状态(图 2.4)。
2. 已完成连接队列(completed connection queue)，为每个已完成 TCP 三路握手过程的客户开设一个条目。这些套接口都处于 ESTABLISHED 状态(图 2.4)。

图 4.6 描述了监听套接口的这两个队列。

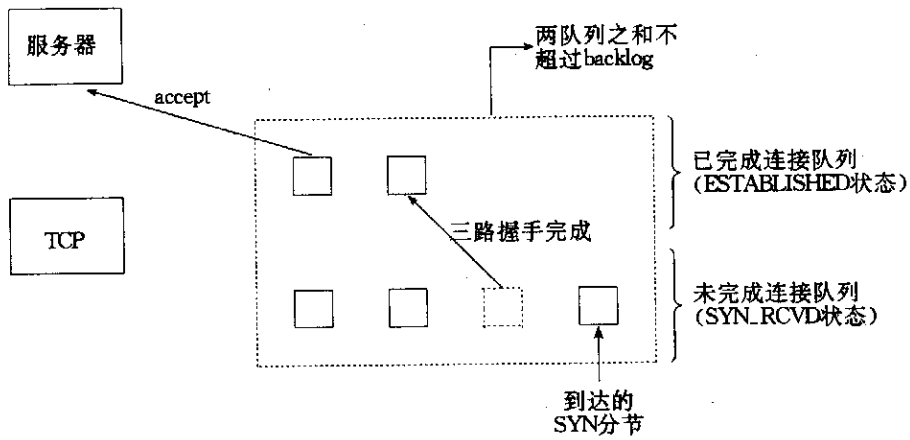


图 4.6 TCP 为监听套接口维护的两个队列

图 4.7 描述了用这两个队列建立连接时所交换的分组。

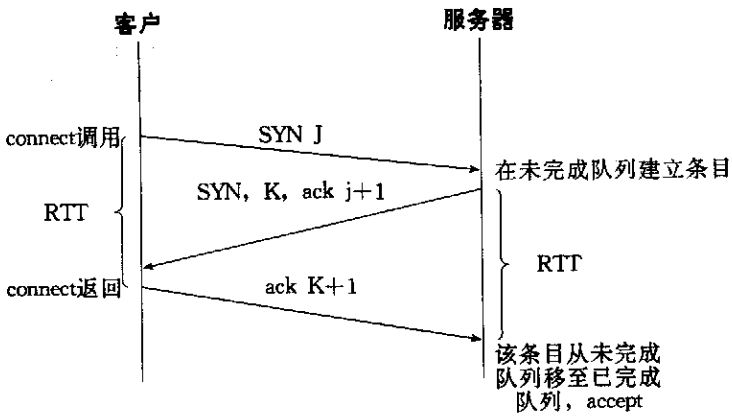


图 4.7 TCP 三路握手和监听套接口的两个队列

当来自客户的 SYN 到达时，TCP 在未完成连接队列中创建一个新条目，然后用三路握手的第二个分节即服务器的 SYN 响应，并附带对客户 SYN 的 ACK(2.5 节)。这个条目一直保留在未完成连接队列中，直到三路握手的第三个分节(客户对服务器 SYN 的 ACK)到达或本条目超时才撤消。(源自 Berkeley 的实现为这些未完成连接条目设置 75 秒的超时时间。)如果三路握手正常完成，该条目将从未完成连接队列搬到已完成连接队列的队尾。当进

程调用函数 accept 时(我们在下一节介绍此函数),已完成连接队列中的队头条目返回给进程,但当队列为空时,进程将睡眠,直到有条目放入已完成连接队列才唤醒它。

处理这两个队列时,必须考虑以下几点:

- 函数 listen 的参数 backlog 曾被规定为两个队列总和的最大值。

  对于 backlog,从未有过正式的定义。4.2BSD 中的手册页面定义它为待处理连接可能增长的最大数目。许多手册页面,甚至 Posix.1g 也将此定义逐字拷贝,但此定义并未解释什么是待处理连接,是处于 SYN_RCVD 状态,处于未接受的 ESTABLISHED 状态,还是处于这两种状态均可。对这个问题的历史性定论是由源自 Berkeley 的实现完成的,可追溯到 4.2BSD,此定义后来被许多其他实现拷贝。

- 源自 Berkeley 的实现给我们说明的 backlog 增加了一个模糊因子:把它乘以 1.5(TCPv1 第 257 页和 TCPv2 第 462 页)。例如,当通常把 backlog 规定为 5 时,在这些系统中实际上却允许最多有 8 个条目在排队,如图 4.10 所示。

  增加此模糊因子的理由已无可考证[Joy 1994],但是如果我们把 backlog 看成是内核能为某套接口排队的最大已完成连接数目([Borman 1997c],稍后讨论),那么增加模糊因子的理由就是把队列中的未完成连接也计算在内。

- 不要把 backlog 定义为 0,因为不同的实现对此有不同的解释(图 4.10),有些实现允许 1 个连接排队,而其他实现不允许有连接排队。如果你不想让任何客户连接到你的监听套接口上,最好关掉该监听套接口。
- 假设三路握手正常完成(例如没有丢失分节,所以不需重传),那么不论客户与服务器间的往返时间(RTT)是多大,未完成队列中的相应条目只保持这段时间。TCPv3 的 14.4 节指出,对于一个 Web 服务器,多对客户与服务器间 RTT 的中间值为 187ms。(由于某些大值对均值影响较大,所以采用中间值作为统计值。)
- 历史上的举例代码常将 backlog 值置为 5,因为这是 4.2BSD 所支持的最大值。20 世纪 80 年代较忙的服务器一天只处理几百个连接,这个值是足够的,但随着万维网(WWW)的发展,较忙的服务器一天要处理几百万个连接,这个小值是根本不够的(TCPv3 第 187～192 页)。较忙的 HTTP 服务器必须指定一个大得多的 backlog 值,且新内核必须支持更大些的值。

  当前的许多系统都允许管理员修改 backlog 的最大值。

- 有个问题:backlog 设置为 5 是常常不够的,那应设为多大呢?这个问题不好回答。现在,HTTP 服务器指定一个较大的值,但如果此值是在源代码中的常值,那就要重新编译服务器程序。另一个方法是假设一个缺省值,但允许设置命令行选项或环境变量来覆盖该缺省值。当指定的值比内核所支持的值要大时,也不受影响,因为内核能把所给的值改为它所支持的最大值且不返回错误(TCPv2 第 456 页)。

通过修改函数 listen 的包裹函数可以解决这个问题,下面给出一个简单的例子,其代码见图 4.8。我们允许环境变量 LISTENQ 来覆盖由调用者所给的值。

```
74 void
75 Listen(int fd,int backlog)
76 {
77          char      *ptr;
78              /* can override 2nd argument with environment variable */
79          if( (ptr = getenv("LISTENQ")) != NULL)
80              backlog = atoi(ptr);
81          if(listen(fd,backlog) < 0)
82              err_sys("listen error");
83 }
```

图 4.8 函数 listen 的包裹函数，允许环境变量指定 backlog 值[lib/wrapsock.c]

- 历史上的手册和书中都指出：仅有固定数目的连接可以排队是为了处理服务器进程在连续的 accept 调用之间处于忙状态的情况。这就意味着两个队列中，已完成连接队列中的条目一般应比未完成连接队列中的条目多。然而，很忙的 Web 服务器已证明这是不对的。指定一个较大 backlog 值的原因是：未完成连接队列中的条目可以随着客户 SYN 分节的到达并等待三路握手的完成而增大。

图 4.9 列出了在某中等忙碌的 Web 服务器上所测出的每个队列条目数的真实值，这些值是这样得到的：在某个工作日，每隔 84ms 便对一个监听 HTTP 套接口采样两个队列的条目数，这样持续 2 个小时。

队列中条目数	未完成队列	已完成队列
0	3 033	90 358
1	7 158	107
2	10 551	59
3	12 960	52
4	11 949	38
5	9 836	27
6	7 754	31
7	6 165	22
8	4 829	30
9	3 687	35
10	2 674	30
11	1 893	25
12	1 431	29
13	1 083	25
14	1 065	49
15	980	7
16	784	
17	696	
18	514	
19	382	
20	294	
21	248	

（续）

队列中条目数	未完成队列	完成队列
22	161	
23	152	
24	121	
25	77	
26	48	
27	33	
28	79	
29	78	
30	90	
31	70	
32	29	
33	16	
34	4	
	90 924	90 924

图 4.9　未完成连接队列和已完成连接队列中的条目数

已完成连接队列 99.4%的时间是空的，但也有一段时间不空。在此服务器上正运行着的系统(BSD/OS 2.0.1)最大的 backlog 值为 64，但列出的值显然未达到此极限。

- 当一个客户 SYN 到达时，若两个队列都是满的，TCP 就忽略此分节(TCPv2 第 930～931 页)，且不发送 RST。这是因为：这种情况是暂时的，客户 TCP 将重发 SYN，期望不久就能在队列中找到空闲条目。要是 TCP 服务器发送了一个 RST，客户的 connect 函数将立即返回一个错误，强制应用进程处理这种情况，而不是让 TCP 正常的重传机制来处理。还有，客户区别不了这两种情况：作为 SYN 的响应，意为“此端口上没有服务器”的 RST 和意为“有服务器在此端口上但其队列满”的 RST。

  Posix.1g 允许以下两种处理方法：忽略新的 SYN，或为此 SYN 响应一个 RST。历史上，所有的源自 Berkeley 的实现都是忽略新的 SYN。

- 三路握手完成之后，服务器调用函数 accept 之前到达的数据应由服务器 TCP 排队，最大数据量为已连接套接口的接收缓冲区大小。

图 4.10 所示为图 1.16 所列的各种操作系统下，参数 backlog 取不同值时已排队连接的实际数目。9 个操作系统分成 6 列，对 backlog 的意义给出各种不同的解释。

backlog	已排列连接实际的最大数目					
	AIX4.2，BSD/OS 3.0	DUnix 4.0，Linux2.0.27，UWare 2.1.2	HP-UX 10.30	SunOS 4.1.4	Solaris 2.5.1	Solaris 2.6
0	1	0	1	1	1	1
1	2	1	1	2	2	3
2	4	2	3	4	3	4

（续）

backlog	已排列连接实际的最大数目					
	AIX4.2，BSD/OS 3.0	DUnix 4.0，Linux2.0.27，UWare 2.1.2	HP-UX 10.30	SunOS 4.14	Solaris 2.5.1	Solaris 2.6
3	5	3	4	5	4	6
4	7	4	6	7	5	7
5	8	5	7	8	6	9
6	10	6	9	8	7	10
7	11	7	10	8	8	12
8	13	8	12	8	9	13
9	14	9	13	8	10	15
10	16	10	15	8	11	16
11	17	11	16	8	12	18
12	19	12	18	8	13	19
13	20	13	18	8	14	21
14	22	14	19	8	15	22

图 4.10 不同 backlog 值时已排列连接的实际数目

AIX、BSD/OS 和 SunOS 4 有传统的 Berkeley 算法，尽管后者（SunOS 4）不允许 backlog 大于 5；HP-UX 和 Solaris 2.6 给 backlog 加了一个模糊因子；Digital Unix、Linux 和 UnixWare 完全照搬 backlog 的原定义，而 Solaris 2.5.1 则对 backlog 加 1。

对于 backlog 为 0 的情况，Linux 允许不受限制的连接数目，这是一个程序缺陷（bug）。

测量这些值的程序见习题 14.5。

我们已提到过，历史上曾把 backlog 值指定为两个队列之和的最大值。在 1996 年间，因特网受到一种称之为 SYN 泛滥（SYN flooding）的新型攻击。黑客写一个程序，它以很高的速率给受害者发送 SYN，装填一个甚至多个 TCP 端口的未完成连接队列。（我们用黑客（hacker）一词来形容攻击者，见[Cheswick and Bellovin 1994]前言部分的描述。）而且，该程序将每个 SYN 的源 IP 地址都置成随机数（称为 IP 欺骗（IP spoofing）），这样服务器的 SYN/ACK 就发往各处，同时也防止了服务器捕获黑客的真实 IP 地址。这样，通过以非法的 SYN 装填未完成连接队列，使合法的 SYN 排不上队，导致合法客户的服务被拒绝（denial of service）。处理这种拒绝服务型攻击通常有两种方法，见[Borman 1997c]中的小结，但这儿我们最感兴趣的是回味一下函数 listen 中 backlog 参数的实际意义，即它应指明为内核将为某套接口排队的最大已完成连接数。对这些已完成的连接规定一个限度，目的是为了防止内核在应用进程无法接收它们时（不管什么原因）还要接受新的连接请求。如果一个系统实现了这一点，如 BSD/OS 3.0，那么应用程序就无需仅仅因为服务器处理太多的客户请求（如一个很忙的 Web 服务器）或保护它不受 SYN 泛滥的便袭而指定一个

巨大的 backlog 值。无论是合法客户还是黑客发出的未完成连接请求，不管数目多大，内核总能处理。但即使是这种情况下，在图 4.9 中我们也可看见，当已完成的连接队列有沉积的条目时(此图中最多增至 15)，SYN 泛滥也会发生，因此传统的值 5 是不够的。

4.6　accept 函数

accept 由 TCP 服务器调用，从已完成连接队列头返回下一个已完成连接(图 4.6)。若已完成连接队列为空，则进程睡眠(假定套接口为缺省的阻塞方式)。

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
                                        返回：非负描述字——OK，-1——出错
```

参数 cliaddr 和 addrlen 用来返回连接对方进程(客户)的协议地址。addrlen 是值-结果参数(3.3 节)：调用前，我们将由 *addrlen 所指的整数值置为由 cliaddr 所指的套接口地址结构的长度，返回时，此整数值即为由内核存在此套接口地址结构内的准确字节数。

如果函数 accept 执行成功，则返回值是由内核自动生成的一个全新描述字，代表与客户的 TCP 连接。当我们讨论函数 accept 时，常把它的第一个参数称为监听套接口(listening socket)描述字(由函数 socket 生成的描述字，用作函数 bind 和 listen 的第一个参数)，把它的返回值称为已连接套接口(connected socket)描述字。将这两个套接口区分开是很重要的。一个给定的服务器常常是只生成一个监听套接口且一直存在，直到该服务器关闭。内核为每个被接受的客户连接创建了一个已连接套接口(也就是说内核已为它完成 TCP 的三路握手过程)。当服务器完成某客户的服务时，关闭已连接套接口。

该函数最多返回三个值：一个既可能是新套接口描述字也可能是错误指示的整数、一个客户进程的协议地址(由指针 cliaddr 所指)以及该地址的大小(由指针 addrlen 所指)。如果我们对返回的客户协议地址不感兴趣，可将指针 cliaddr 和 addrlen 均置为空指针。

图 1.9 给出了这些指针的示例，已连接套接口每次都在循环中关闭，但监听套接口在整个服务器有效期内都保持开放。我们也明白了为什么函数 accept 的第二和第三个参数都是空指针，因为我们对客户的身份不感兴趣。

例：值-结果参数

现在，我们通过修改图 1.9 中所示代码以输出 IP 地址和客户的端口号来看看如何处理函数 accept 的值-结果参数，见图 4.11。

新的声明

第 7～8 行　我们定义两个新变量：len，它是一个值-结果参数；cliaddr，它是客户的协议地址。

接受连接并显示客户地址

第19～23行 我们将len初始化为套接口地址结构的大小，将指向cliaddr结构的指针和指向len的指针分别作为函数accept的第二和第三个参数。调用函数inet_ntop(3.7节)将套接口地址结构中的32位IP地址转换为一个点分十进制数ASCII字符串，调用函数ntohs(3.4节)将16位的端口号从网络字节序转换为主机字节序。

> 调用函数sock_ntop来代替函数inet_ntop将使得我们的服务器更具协议无关性，但该服务器已依赖于IPv4了。在图11.9中，我们将给出该服务器程序协议无关的版本。

如果在同一主机上运行该新服务器程序，然后运行客户程序，连接服务器两次，在客户上我们将看到下面的输出：

```
solaris % daytimetcpcli 127.0.0.1
Wed Jan 17 15:42:35 1996
solaris % daytimetcpcli 206.62.226.33
Wed Jan 17 15:42:53 1996
```

首先，我们将服务器的地址指定为回馈地址(127.0.0.1)，然后指定本身的IP地址(206.62.226.33)，下面是相应的服务器输出：

```
solaris # daytimetcpsrv1
connection from 127.0.0.1 ,port 33188
connection from 206.62.226.33, port 33189
```

```
 1 #include     "unp.h"
 2 #include     <time.h>
 3 int
 4 main(int argc,char **argv)
 5 {
 6     int          listenfd,connfd;
 7     socklen_t len;
 8     struct sockaddr_in servaddr,cliaddr;
 9     char         buff[MAXLINE];
10   time_t         ticks;
11   listenfd = Socket(AF_INET,SOCK_STREAM,0);
12   bzero(&servaddr,sizeof(servaddr));
13   servaddr.sin_family = AF_INET;
14   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15   servaddr.sin_port = htons(13);      /* daytime server */
16   Bind(listenfd,(SA *) &servaddr,sizeof(servaddr));
17   Listen(listenfd,LISTENQ);
18   for(;;) {
19       len = sizeof(cliaddr);
20       connfd = Accept(listenfd,(SA *) &cliaddr,&len);
21       printf("connection from %s,port %d\n",
22           Inet_ntop(AF_INET,&cliaddr.sin_addr,buff,sizeof(buff)),
23           ntohs(cliaddr.sin_port));
```

```
24          ticks = time(NULL);
25          snprintf(buff,sizeof(buff),"%.24s\r\n",ctime(&ticks));
26          Write(connfd,buff,strlen(buff));
27          Close(connfd);
28      }
29  }
```

图 4.11　输出客户 IP 地址和端口号的时间/日期服务器[intro/daytimetcpsrv1.c]

```
solaris # daytimetopsrv1
connection from 127.0.0.1, port 33188
connection from 206.62.226.33 port 33189
```

注意客户 IP 地址变化的结果。由于我们的时间/日期客户(图 1.5)不调用函数 bind,在 4.4 节中我们说此时内核选择的源 IP 地址是基于所用的接口的。第一种情况下,内核将源 IP 地址置为回馈地址,第二种情况下,内核又将地址置为以太网接口的 IP 地址。在该例子中我们还注意到,Solaris 内核所选择的临时端口号首先是 33188,然后是 33189(回忆一下图 2.6)。

最后一点,服务器脚本的 shell 提示符变为(#),此为超级用户的常用提示符。该服务器必须运行在超级用户特权下,绑定保留的 13 号端口。如果没有超级用户权限,对函数 bind 的调用将失败:

```
solaris % daytimetcpsrv1
bind error:Permission denied
```

4.7　fork 和 exec 函数

在阐述如何书写一个并发服务器程序之前(下一节),我们必须首先介绍一下 Unix 的函数 fork,该函数是 Unix 中派生新进程的唯一方法。

```
#include <unistd.h>
pid_t fork(void);
                                返回:在子进程中为 0,在父进程中为子进程 ID,-1——出错
```

如果你以前从未接触过此函数,则弄明白 fork 最困难的部分便是它调用一次却返回两次。在调用进程(称为父进程),它返回一次,返回值是新派生进程(称为子进程)的进程 ID 号,在子进程它还返回一次,返回值为 0。因此,可通过返回值来判断当前进程是子进程还是父进程。

fork 在子进程返回 0 而不是父进程 ID,原因是:子进程只有一个父进程,它总可以调用 getppid 来得到;而父进程有许多子进程,它没有办法来得到各子进程的 ID。如果父进程想跟踪所有子进程的 ID,它必须记住 fork 的返回值。

父进程中调用 fork 之前打开的所有描述字在函数 fork 返回之后都是共享的。我们将看到网络服务器利用此特性:父进程调用 accept,然后调用 fork,这样,已连接套接口就在父进程与子进程间共享,一般来说,接下来便是子进程读、写已连接套接口,而父进程则关闭已连接套接口。

fork 有两个典型应用：

1. 一个进程可为自己创建一个拷贝，这样，当一个拷贝处理一个操作时，其他的拷贝可以执行其他的任务。这是非常典型的网络服务器，本书后面我们将看到许多这样的例子。
2. 一个进程想执行其他的程序，由于创建新进程的唯一方法是调用 fork，进程首先调用 fork 来生成一个拷贝，然后其中一个拷贝（通常为子进程）调用 exec（后面将介绍）来代替自己去执行新程序。这对于像 shell 这样的程序是典型的用法。

以文件形式存储在磁盘上的可执行程序被 Unix 执行的唯一方法是：由一个现有进程调用六个 exec 函数中的一个。（当这六个函数中是哪一个被调用并不重要时，我们常统称它们为 exec 函数。）exec 用新程序代替当前进程映像，且此新程序一般都从 main 函数开始执行，进程 ID 并不改变。我们一般将调用 exec 的进程称为调用进程（calling process），而将新执行的程序称为新程序（new program）。

> 较老一些的手册和书上不确切地将新程序称为新进程（new process），这是不对的，因为并没有派生一个新的进程。

六个 exec 函数间的区别是：（a）被执行的程序是由文件名（filename）还是路径名（pathname）指定；（b）新程序的参数是一一列出还是由一个指针数组来索引；（c）调用进程的环境传递给新程序还是指定新环境。

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ...
        /* (char *) 0, char *const envp[] */);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);
int execvp(const char *filename, char *const argv[]);
                                所有6个函数返回：-1——出错，无返回——成功
```

这些函数只在出错时才返回调用者，否则，控制权传递到新程序的开始，通常是传递到函数 main。

这六个函数间的关系示于图 4.12。一般来说，只有 execve 是内核中的系统调用，其他五个函数都是调用 execve 的库函数。

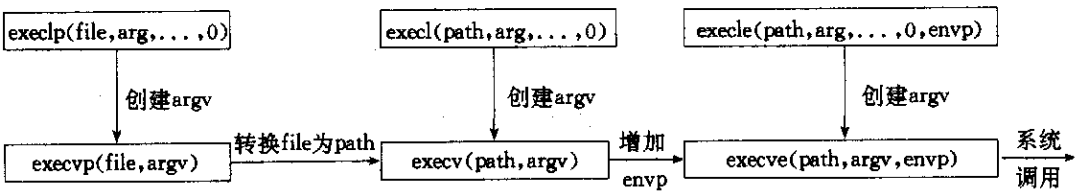


图 4.12 六个 exec 函数的关系

注意这六个函数的下列区别：

1. 上面的三个函数将每个参数串指定为 exec 的独立参数，用一个空指针来表示可变数量参数的终止；下面的三个函数有一个数组 argv，它包含了指向参数串的所有指针。此 argv 数组必须含有一个空指针来表示结束，因为数组的大小没有指定。
2. 左边一列的两个函数指定一个 filename 参数，它根据现行的 PATH 环境变量转换为 pathname。若函数 execlp 或 axecvp 的 filename 参数中含有斜杠(/)(不论在串中的什么位置)，将不再使用 PATH 变量。后两列的四个函数指定一个完全的 pathname 参数。
3. 左边两列的四个函数不显式指定一个环境指针，而是用外部变量 environ 的当前值来创建一个环境表传递给新程序；右边一列的两个函数显式指定一个环境表。envp 指针数组必须以一个空指针结束。

一般来说，调用 exec 之前在进程中打开的描述字在跨 exec 过程中保持打开状态。我们使用限定词“一般来说”是因为描述字也可用函数 fcntl 设置 FD_CLOEXEC 描述字标志来关闭。服务器 inetd 就利用了此特性，这一点我们在 12.5 节中还会讨论。

4.8　并发服务器

图 4.11 中的服务器是一个迭代服务器(iterative server)。对于像时间/日期这样简单的服务器，这是很好的；但是，当客户请求需花很长时间来得到服务时，我们不可能让一个服务器长时间地为某一个客户服务，而是同时为多个客户服务。Unix 下写一个并发服务器程序最简单的办法就是为每个客户均 fork 一个子进程。图 4.13 给出了一个典型的并发服务器程序框架。

当连接建立时，accept 返回，服务器调用 fork，然后子进程为客户提供服务(通过 connfd 即已连接套接口)，父进程等待另一个连接(通过 listenfd 即监听套接口)。子进程开始处理新客户后，父进程便关闭已连接套接口。

图 4.13 中，我们假设函数 doit 包含了客户所要求的所有服务功能。当此函数返回时，我们显式地关闭子进程中的已连接套接口。这一点并不一定要做，因为下一步是调用 exit，而进程结束的部分处理就是关闭所有由内核打开的描述字。是否需显式地调用 close，由个人的编程风格而定。

在 2.5 节中我们知道，对 TCP 套接口调用 close 会引发一个 FIN，后跟 TCP 连接终止序列。为什么图 4.13 中父进程 connfd 的 close 不终止它与客户的连接呢？为了搞明白此执行过程，我们必须了解每个文件或套接口都有一个访问计数，该访问计数在文件表项中维护(APUE 第 58～59 页)，它表示当前指向该文件或套接口的打开的描述字个数。在图 4.13 中，从 socket 返回后，与 listenfd 关联的文件表项访问计数值为 1，从 accept 返回后，与 connfd 关联的文件表项访问计数值也为 1。但是，当 fork 返回后，两个描述字在父进程与子进程间共享(即复制)，所以，与两个套接口相关联的文件表项访问计数值均为 2。当父进程关闭 connfd 时，只是将访问计数值从 2 减为 1。描述字只在访问计数值达到 0 时才真正关闭，这在后面某个时候子进程关闭 connfd 时会碰上。

```
pid_t pid;
int    listenfd,connfd;

listenfd = Socket( ... );
    /* fill in sockaddr_in() with server's well-known port */
Bind(listenfd,...);
Listen(listenfd,LISTENQ);
for( ; ;) {
    connfd = Accept(listenfd,...);      /* probably blocks */
    if( (pid = Fork()) == 0) {
        Close(listenfd);        /* child closes listening socket */
        doit(connfd);           /* process the request */
        Close(connfd);          /* done with this client */
        exit(0);                /* child terminates */
    }
    Close(connfd);              /* parent closes connected socket */
}
```

图 4.13 典型的并发服务器程序框架

我们还可将图 4.13 中出现的套接口和连接进行如下的可视化。首先，图 4.14 给出了在服务器阻塞于 accept 调用、连接请求从客户到达时客户和服务器的状态。

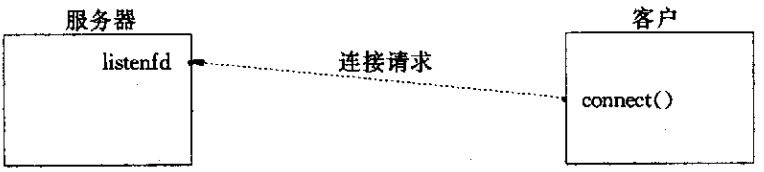


图 4.14 accept 返回前客户-服务器的状态

从 accept 返回后，我们立即就有图 4.15 所示状态。连接被内核接受，新的套接口即 connfd 被创建，这是一个已连接套接口，可由此通过连接读、写数据。

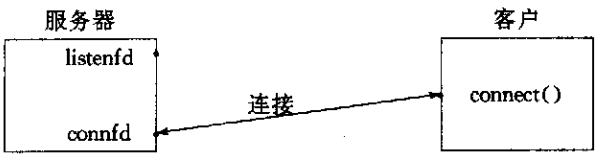


图 4.15 accept 返回后客户-服务器的状态

并发服务器的下一步是调用 fork，图 4.16 给出了从 fork 返回后的状态。

注意，此时描述字 listenfd 和 connfd 都是在父进程和子进程间共享的。

下一步是由父进程关闭已连接套接口，由子进程关闭监听套接口，如图 4.17 所示。

这是所期望的套接口最终状态：子进程处理与客户的连接，父进程可对监听套接口再次调用 accept 来处理下一个客户连接。

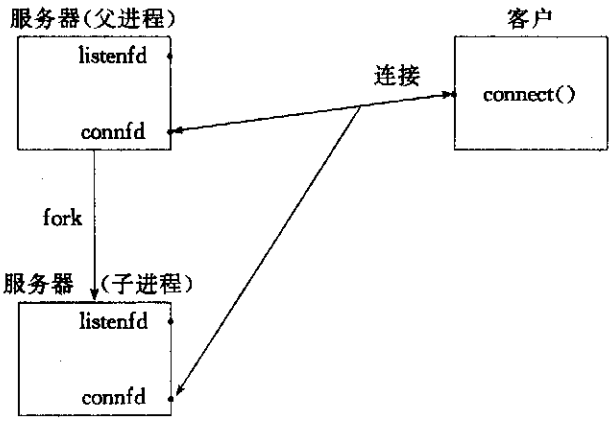


图 4.16　fork 返回后客户-服务器的状态

4.9　close 函数

一般的 Unix 函数 close 也用来关闭套接口，终止 TCP 连接。

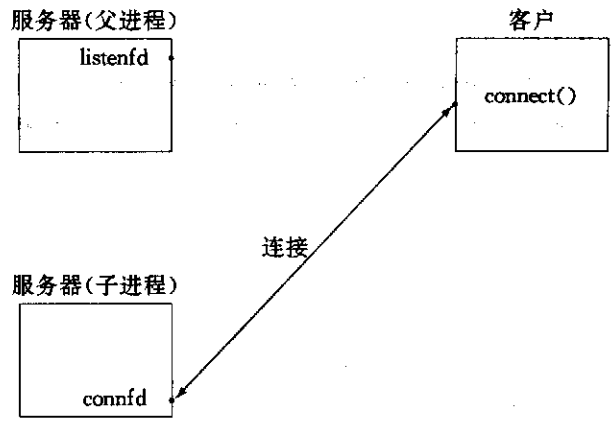


图 4.17　父子进程关闭相应套接口后客户-服务器的状态

```
#include <unistd.h>
int close(int sockfd);
```

返回：0——OK，−1——出错

TCP 套接口的 close 其缺省功能是将套接口做上“已关闭”标记，并立即返回到进程。这个套接口描述字不能再为进程所用：它不能用作为函数 read 或 write 的参数，但 TCP 将试着发送已排队待发的任何数据，然后按正常的 TCP 连接终止序列进行操作(2.5 节)。

在 7.5 节中，我们将介绍 SO_LINGER 套接口选项，它可用来改变 TCP 套接口的缺省功能。在那节，我们还介绍 TCP 应用进程必须做的事情，以保证对方的应用进程接收到已发出的任何数据。

描述字访问计数

在4.8节结尾我们提到，当并发服务器中父进程关闭已连接套接口时，它只是将描述字的访问计数值减1。由于访问计数值仍大于0，这次close调用并不引发TCP的四分组连接终止序列。对于父进程与子进程间共享已连接套接口的并发服务器来说，这正是所期望的。

如果我们确实想对TCP连接发一个FIN，可以改用函数shutdown(6.6节)而不是close，在6.5节我们再对此作介绍。

我们也必须意识到，如果父进程不对每个由accept返回的已连接套接口调用close，并发服务器将会发生什么。首先，父进程最终将耗尽可用描述字，因为任何进程在某时刻打开的描述字数总是有限的。但更重要的是，没有一个客户连接被终止。当子进程关闭已连接套接口时，它的访问计数值由2减为1且保持为1，因为父进程从未关闭已连接套接口，这将妨碍TCP连接终止序列的执行，从而连接永远保持开放。

4.10 getsockname和getpeername函数

这两个函数或返回与套接口关联的本地协议地址(getsockname)，或返回与套接口关联的远程协议地址(getpeername)。

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
                                        两者均返回：0——OK，-1——出错
```

注意，两个函数的最后一个参数是值-结果参数，也就是说，两个函数都装填由指针localaddr或peeraddr所指的套接口地址结构。

> 讨论bind时我们提到，术语"名字"是一个误导。这两个函数返回与某网络连接任一端关联的协议地址，对于IPv4和IPv6，它是IP地址和端口号的组合。这些函数与域名没有任何联系(第9章)。

这两个函数因以下理由而存在：

- 在一个不调用bind的TCP客户上，当connect成功返回后，getsockname返回内核分配给此连接的本地IP地址和本地端口号。
- 在以端口号0调用bind后(通知内核选择本地端口号)，getsockname返回由内核分配的本地端口号。
- getsockname可用来获得某套接口的地址族，如图4.19所示。
- 在捆绑了一个通配IP地址的TCP服务器上(图1.9)，一旦与客户建立了连接(accept成功返回)，就可以调用getsockname来获得分配给此连接的本地IP地址。在这样的调用中套接口描述字参数必须是已连接套接口的描述字，而不是监听套接口的描述字。

  当一个服务器由调用accept的进程调用exec启动执行时，它获得客户身份的唯一

途径便是调用 getpeername。守护进程 inetd(12.5 节)fork 和 exec 一个 TCP 服务器时就是这么做的，如图 4.18 所示。inetd 调用 accept(左上方方框)，返回两个值：已连接套接口描述字 connfd，这是函数的返回值；客户的 IP 地址及端口号，见图中标有"对方地址"的小方框(它是一个网际套接口地址结构)。inetd 随后调用 fork，派生 inetd 的一个子进程。由于子进程从父进程的存储映像拷贝开始执行，因此对方地址的套接口地址结构对子进程也是可访问的。同样，子进程也能访问已连接套接口描述字，因为描述字在父子进程间是共享的。但是当子进程执行真正的服务器程序(譬如说 Telnet 服务器程序)时，子进程的存储映像被 Telnet 服务器的新程序文件所代替(也就是说，含有对方地址的套接口地址结构丢失了)，不过已连接套接口描述字在跨越 exec 后仍保持开放。Telnet 服务器调用的第一个函数便是 getpeername，由此获得客户的 IP 地址和端口号。

显然，在这最后一个例子中，Telnet 服务器在启动时必须知道 connfd 的值。这里有两个常用方法：第一种方法是，调用 exec 的进程可以将描述字号格式化成一个字符串，并将它作为一个命令行参数传递给新程序；第二种方法是，确立调用 exec 前把某个描述字总是设定成已连接套接口的约定。inetd 采用的就是第二种方法，它总是将描述字 0、1 和 2 设定成已连接套接口。

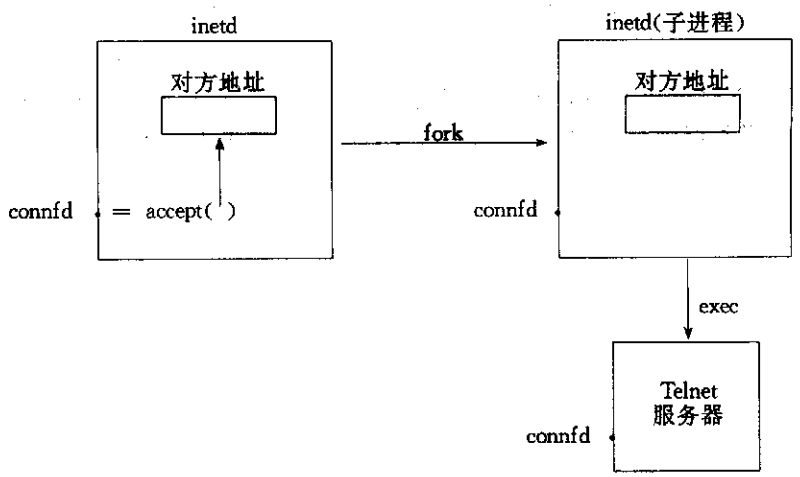


图 4.18　inetd 派生服务器的例子

例子：获取套接口的地址族

图 4.19 中所示的函数 sockfd_to_family 返回某个套接口的地址族。

```
1 #include    "unp.h"
2 int
3 sockfd_to_family(int sockfd)
4 {
5     union {
6         struct sockaddr sa;
7         char    data[MAXSOCKADDR];
8     } un;
9     socklen_t len;
```

```
10      len = MAXSOCKADDR;
11      if(getsockname(sockfd,(SA *) un.data,&len) < 0)
12        return(-1);
13      return (un.sa.sa_family);
14  }
```

图 4.19 返回套接口的地址族[lib/sockfd_to_family.c]

为最大的套接口地址结构分配空间

第5～8行 由于不知道要分配的套接口地址结构的类型，我们在头文件 unp.h 中使用常值 MAXSOCKADDR，它是最大套接口地址结构的字节数。我们在跟一个通用套接口地址结构的联合中定义了该大小的一个字符数组。

调用 getsockname

第10～13行 我们调用 getsockname 返回地址族。由于 Posix.1g 允许对未绑定的套接口进行 getsockname 调用，该函数应对任何打开的套接口描述字都有效。

4.11 小 结

所有的客户和服务器都从调用 socket 开始，返回一个套接口描述字。然后，客户调用 connect，服务器调用 bind、listen 和 accept。套接口一般由标准的 close 函数关闭，当然也可用函数 shutdown 来关闭(6.6节)。我们还要检查套接口选项 SO_LINGER 的效果(7.5节)。

多数 TCP 服务器是与调用 fork 来处理每个客户连接的服务器并发执行的。我们还将看到，多数 UDP 服务器则是迭代的。这两个模型已成功地运用了许多年，在第27章，我们将讨论使用线程与进程的其他服务器程序设计方法。

4.12 习 题

4.1 在4.4节中，我们说头文件<netinet/in.h>中定义的常值 INADDR_是主机字节序的，如何辨别？

4.2 修改图1.5，在 connect 返回成功后，调用 getsockname。使用 sock_ntop 输出分配给 TCP 套接口的本地 IP 地址和本地端口号。你的系统的临时端口在什么范围内(图2.6)？

4.3 在一个并发服务器中，假设在调用 fork 后子进程先运行，而且在 fork 调用返回父进程前，子进程就完成了对客户的服务，图4.13中对 close 的两次调用将会发生什么情况？

4.4 在图4.11中，把服务器的端口号从13改到9999(这样不需有超级用户特权就能启动程序)，并删掉 listen 调用，将会发生什么情况？

4.5 继续上一题，删掉 bind 调用，但保持 listen 调用，又将发生什么情况？

第 5 章　TCP 客户-服务器程序例子

5.1　概　述

现在，我们用前面章节所介绍的基本函数来编写一个完整的 TCP 客户-服务器程序例子，这个简单的例子是完成下述功能的一个回射服务器：

1. 客户从标准输入读一行文本，写到服务器上；
2. 服务器从网络输入读此行，并回射给客户；
3. 客户读此回射行并写到标准输出。

图 5.1 以所用的输入、输出函数描述了这个简单的客户-服务器模型。

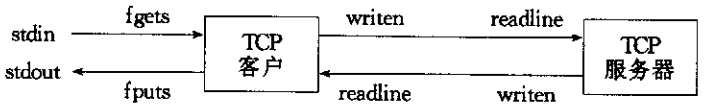


图 5.1　简单的回射客户-服务器

我们在客户与服务器之间画了两个箭头，但这是一个全双工的 TCP 连接。fgets 和 fputs 来自标准 I/O 函数库，writen 和 readline 函数可参见 3.9 节。

尽管我们开发自己的回射服务器实现，多数 TCP/IP 实现已经提供了这样的服务器，TCP 和 UDP 都使用(2.10 节)，我们自己的客户也用此服务器。

回射输入行这样一个客户-服务器程序是一个尽管简单然而有效的网络应用程序例子，客户-服务器实现所需要的所有基本步骤在此例中均出现了。若想将此例子扩充成你自己的应用程序，你只需依据对客户输入内容的处理方式，对服务器的工作作相应改变。

除了在正常方式下运行我们的客户及服务器外(输入一行，观察回射)，我们还检查此例的许多边界条件：客户和服务器都启动时是什么情况；客户正常终止时是什么情况；如果服务器进程在客户之前终止客户会是什么情况；如果服务器主机崩溃则客户又会是什么情况，等等。通过观察这些情况，弄清网络层次发生的一些事情以及它们是如何反映到套接口 API 的，我们将对这些层次的工作原理有更多的认识，并知道如何编写应用代码来处理这些情形。

在所有这些例子中，我们使用了诸如地址和端口这样的特定于协议的“硬编码”常值，这有两个原因：第一，我们必须确切地知道存储在特定于协议的地址结构中的内容；第二，我们还没有讨论到可使这个方便地移植的库函数，这些库函数将在第 9 章和第 11 章中讨论。

我们现在就注意，随着我们学习越来越多的网络编程知识(图 1.12 和图 1.13)，在后续章节中我们将对这儿的客户和服务器程序做多次修改。

5.2 TCP回射服务器程序:main函数

我们的客户和服务器程序依循图4.1中的函数调用流程,其中并发服务器程序见图5.2。

创建套接口,捆绑服务器的众所周知端口

第9～15行 创建一个TCP套接口,用通配地址(INADDR_ANY)和服务器的众所周知端口(在头文件unp.h中定义为9877的SERV_PORT)填写网际套接口地址结构。捆绑一个通配地址意味着通知系统,我们将接受目的地址为任何本地接口的连接(假设系统是多宿的)。我们对TCP端口号的选择基于图2.6,它应比1023大(我们不需要保留端口),大于5000(以免与许多源自Berkeley的实现分配的临时端口冲突),小于49152(以免与临时端口号的"正确"范围冲突),并应不与任何已注册的端口冲突。这个套接口由函数listen转换为监听套接口。

等待完成客户连接

第17～18行 服务器阻塞于accept调用,等待客户连接的完成。

并发服务器

第19～24行 对于每个客户,函数fork派生出一个子进程,由此子进程处理新客户。正如我们在4.8节中所讨论的那样,子进程关闭监听套接口,而父进程关闭已连接套接口,然后,子进程调用函数str_echo(图5.3)来处理客户。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int         listenfd,connfd;
6     pid_t       childpid;
7     socklen_t clilen;
8     struct sockaddr_in cliaddr,servaddr;
9     listenfd = Socket(AF_INET,SOCK_STREAM,0);
10    bzero(&servaddr,sizeof(servaddr));
11    servaddr.sin_family = AF_INET;
12    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
13    servaddr.sin_port = htons(SERV_PORT);
14    Bind(listenfd,(SA * ) &servaddr,sizeof(servaddr));
15    Listen(listenfd,LISTENQ);
16    for(; ;) {
17        clilen = sizeof(cliaddr);
18        connfd = Accept(listenfd,(SA * ) &cliaddr,&clilen);
19        if((childpid= Fork()) == 0) {     /* child process */
20            Close(listenfd);        /* close listening socket */
21            str_echo(connfd);       /* process the request */
22            exit(0);
```

```
23          }
24          Close(connfd);          /* parent closes connected socket */
25      }
26  }
```

图5.2 TCP回射服务器程序(图5.12中改进)[tcpcliserv/tcpserv01.c]

```
1  #include    "unp.h"
2  void
3  str_echo(int sockfd)
4  {
5      ssize_t n;
6      char    line[MAXLINE];
7      for ( ; ; ) {
8          if((n = Readline(sockfd,line,MAXLINE)) == 0)
9              return;             /* connection closed by other end */
10         Writen(sockfd,line,n);
11     }
12 }
```

图5.3 函数str_echo:在套接口上回射行[lib/str_echo.c]

5.3 TCP回射服务器程序:str_echo函数

图5.3中的函数str_echo执行处理每个客户的服务:从客户读入各行并将它们回射给客户。

读一行并回射此行

第7～11行 readline从套接口读下一行,该行随后由writen回射给客户。如果客户关闭连接(正常情况),那么接收到的客户FIN导致子进程的readline返回0,从而使得函数str_echo也返回,终止图5.2中的子进程。

5.4 TCP回射客户程序:main函数

图5.4所示为TCP客户的main函数。

```
1  #include    "unp.h"
2  int
3  main(int argc,char **argv)
4  {
5      int     sockfd;
6      struct sockaddr_in servaddr;
7      if(argc != 2)
8          err_quit("usage:tcpcli <IPaddress>");
9      sockfd = Socket(AF_INET,SOCK_STREAM,0);
10     bzero(&servaddr,sizeof(servaddr));
11     servaddr.sin_family = AF_INET;
12     servaddr.sin_port = htons(SERV_PORT);
```

```
13    Inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
14    Connect(sockfd,(SA *) &servaddr,sizeof(servaddr));
15    str_cli(stdin,sockfd);      /* do it all */
16    exit(0);
17 }
```

图 5.4 TCP 回射客户程序[tcpcliserv/tcpcli01.c]

创建套接口,装填网际套接口地址结构

第 9～13 行 创建一个 TCP 套接口,用服务器的 IP 地址和端口号装填一个网际套接口地址结构。我们可从命令行参数取得服务器的 IP 地址,从头文件 unp.h 取得服务器的众所周知端口号(SERV_PORT)。

与服务器连接

第 14～15 行 connect 建立与服务器的连接,函数 str_cli(图 5.5)完成客户处理的剩余部分工作。

5.5 TCP 回射客户程序:str_cli 函数

图 5.5 中所示该函数完成客户处理循环:从标准输入读一行文本,写到服务器上,读回服务器对此行的回射,并把回射写到标准输出上。

```
1 #include    "unp.h"
2 void
3 str_cli(FILE *fp,int sockfd)
4 {
5     char    sendline[MAXLINE],recvline[MAXLINE];
6     while (Fgets(sendline,MAXLINE,fp) != NULL) {
7         Writen(sockfd,sendline,strlen(sendline));
8         if(Readline(sockfd,recvline,MAXLINE) == 0)
9             err_quit("str_cli:server terminated prematurely");
10        Fputs(recvline,stdout);
11    }
12 }
```

图 5.5 函数 str_cli:客户处理循环[lib/str_cli.c]

读一行,写到服务器

第 6～7 行 fgets 读一行文本,writen 将此行发送到服务器。

从服务器读回射行,写到标准输出

第 8～10 行 readline 从服务器读回射行,fputs 将其写到标准输出。

返回 main 函数

第 11～12 行 当 fgets 遇到文件结束符或错误时,返回一个空指针,此时客户处理循环便终止。我们的包裹函数 Fgets 检查错误,如果发现错误就放弃执行,因此 Fgets 函数在遇到

文件结束符时才返回一个空指针。

5.6　正常启动

虽然我们的 TCP 例子很小(两个 main 函数加上 str_echo、str_cli、readline 和 writen 总共才 150 行代码),但对于我们弄清客户和服务器是如何启动,如何终止的,更为重要的是当某些错误发生时,如:客户主机崩溃,客户进程崩溃,网络连接断开等等,将会发生什么事情,这个例子则至关重要。只有搞清这些边界条件以及它们与 TCP/IP 协议的相互作用,我们才能写出可以处理这些情况的健壮的客户和服务器程序。

首先,我们在主机 bsdi 上后台启动服务器。

```
bsdi % tcpserv01 &
[1]    21130
```

当服务器启动后,调用 socket、bind、listen 和 accept,并阻塞于 accept 调用 (我们还没有启动客户)。在启动客户之前,我们运行程序 netstat 来检查服务器监听套接口的状态。

```
bsdi % netstat -a
Proto   Recv-Q   Send-Q   Local Address      Foreign Address      (state)
tcp          0        0   *.9877             *.*                  LISTEN
```

这里,我们只给出了输出的第一行(标题)以及我们最关心的那一行。此命令列出系统中所有套接口的状态,可能有大量输出。我们必须用标志-a 来查看监听套接口。

这个输出正是我们所期望的:有一个套接口处于 LISTEN 状态,它有通配的本地 IP 地址,本地端口为 9877。netstat 用星号"*"来表示一个为 0 的 IP 地址(INADDR_ANY,通配地址)或为 0 的端口号。

然后,我们在相同的主机上启动客户,指明服务器的 IP 地址为 127.0.0.1,也可指明该地址为 206.62.226.35(图 1.16)。

```
bsdi % tcpcli01 127.0.0.1
```

客户调用 socket 和 connect,后者引起 TCP 的三路握手过程。当三路握手完成后,connect 返回客户,accept 返回服务器,连接建立,接着执行以下步骤:

1. 客户调用 str_cli 函数,该函数将阻塞于 fgets 调用,因为我们还从未输入过一行文本。
2. 当 accept 返回服务器后,服务器调用 fork,由子进程调用 str_echo。此函数调用 readline,它又因调用 read 而阻塞,等待客户送出一行。
3. 另一方面,服务器父进程再次调用 accept 并阻塞,等待下一个客户连接。

至此,我们有三个进程,它们都在睡眠(阻塞):客户、服务器父进程和服务器子进程。

当三路握手完成时,我们特意首先列出了客户的步骤,然后是服务器的步骤。从图 2.5 中可知其原因:客户接收到三路握手的第二个分节时,connect 返回,而服务器要直到接收到三路握手的第三个分节才返回,即在 connect 返回之后过了½往返时间才返回。

我们特意在同一主机上运行客户和服务器，因为这是试验客户-服务器应用程序的最简单方法。由于我们在同一主机上运行客户和服务器，netstat 相应于 TCP 连接又多显示了两行输出。

```
bsdi % netstat -a
Proto Recv-Q Send-Q  Local Address      Foreign Address    (state)
tcp        0      0  localhost. 9877    localhost. 1052    ESTABLISHED
tcp        0      0  localhost. 1052    localhost. 9877    ESTABLISHED
tcp        0      0  *. 9877            *. *               LISTEN
```

第一个 ESTABLISHED 行，其端口号是 9877，对应于服务器子进程的套接口；第二个 ESTABLISHED 行，其端口号是 1052，对应于客户的套接口。如果我们在不同的主机上运行客户和服务器，那么客户主机就只输出客户的套接口，服务器主机也只输出两个服务器套接口。

我们也可用命令 ps 来检查这些进程的状态和关系。

```
bsdi % ps -l
PID   PPID  WCHAN  STAT  TT       TIME COMMAND
19130 19129 wait   Is    p1    0:04.99 -ksh (ksh)
21130 19130 netcon I     p1    0:00.06 tcpserv01
21131 19130 ttyin  I+    p1    0:00.09 tcpcli01 127.0.0.1
21132 21130 netio  I     p1    0:00.01 tcpserv01
21134 21133 wait   Ss    p2    0:03.50 -ksh (ksh)
21149 21134 -      R+    p2    0:00.05 ps -l
```

（我们已删去了不影响讨论的许多输出列。）在此输出中，我们在相同的窗口下运行客户和服务器（p1，表示伪终端号 1），在另一个窗口运行命令 ps（p2）。PID 和 PPID 列给出了父进程和子进程的关系，由于子进程的 PPID 是父进程的 PID，我们可以看出，第一个 tcpserv 行是父进程，第二个 tcpserv 行是子进程。父进程的 PPID 是 shell（ksh）。

我们所有三个网络进程的 STAT 列都是“I”，意味着进程为空闲（即睡眠），两个 STAT 列后面的加号“+”表示该进程为控制终端的前台进程。如果进程处于睡眠态，WCHAN 列指出相应状态。4.4BSD 在进程阻塞于 accept 或 connect 时，输出 netcon；在进程阻塞于套接口输入或输出时，输出 netio；在进程阻塞于终端 I/O 时，输出 ttyin。我们三个网络进程的 WCHAN 值说明是正确的。

5.7 正常终止

此时，连接已建立，无论我们在客户标准输入上键入什么，都将得到回射。

```
bsdi % tcpcli01 127.0.0.1      我们已经给出过本行
hello,world                    现在键入这一行
hello,world                    这一行被回射回来
good bye
good bye
^D                             <Ctrl-D>是我们的终端 EOF 字符
```

我们键入两行，每行都得到回射，我们键入终止的 EOF 字符（Control-D）以终止客户，

此时若立即执行 netstat 命令，我们将看到以下结果。

```
bsdi % netstat -a | grep 9877
tcp        0      0  localhost.1052         localhost.9877         TIME_WAIT
tcp        0      0  *.9877                 *.*                    LISTEN
```

连接的客户端(本地端口号为 1052)进入了 TIME_WAIT 状态(图 2.6)，而监听服务器仍在等待另一个连接。(这一次，我们使命令 netstat 的输出进入管道作为 grep 的输入，只输出与服务器的众所周知端口有关行，但这样做也删掉了标题行。)

我们可以总结出正常终止客户和服务器的步骤：

1. 当我们键入 EOF 字符时，fgets 返回一个空指针，于是 str_cli(图 5.5)返回。
2. 当 str_cli 返回到客户的函数 main(图 5.4)时，main 通过调用 exit 终止。
3. 进程终止处理的一部分是关闭所有打开的描述字，所以客户套接口由内核关闭。这导致客户 TCP 发送一个 FIN 给服务器，服务器 TCP 则以 ACK 响应，这就是 TCP 连接终止序列的前半部分。此时，服务器套接口处于 CLOSE_WAIT 状态，客户套接口处于 FIN_WAIT_2 状态(图2.4和图2.5)。
4. 当服务器 TCP 接收 FIN 时，服务器子进程阻塞于 readline(图 5.3)调用，于是 readline 返回 0，这导致函数 str_echo 返回服务器子进程的 main。
5. 服务器子进程通过调用 exit 来终止(图 5.2)。
6. 服务器子进程中打开的所有描述字随之关闭。由子进程来关闭已连接套接口引发 TCP 连接终止序列的最后两个分节：一个从服务器到客户的 FIN 和一个从客户到服务器的 ACK(图2.5)。至此，连接完全终止，客户套接口进入 TIME_WAIT 状态。
7. 进程终止处理的另一部分是在服务器子进程终止时给父进程发一个信号 SIGCHLD。这在本例中发生了，但我们在代码中未捕获这个信号，而这个信号的缺省动作是忽略的。我们可以用命令 ps 来检查。

```
bsdi % ps
  PID  TT  STAT      TIME  COMMAND
19130  p1  Ss     0:05.08  -ksh (ksh)
21130  p1  I      0:00.06  ./tcpserv01
21132  p1  Z      0:00.00  (tcpserv01)
21167  P1  R+     0:00.10  ps
```

现在，子进程的状态是 Z(僵尸)。

我们必须清除僵尸进程，这要涉及到 Unix 信号的处理。下一节我们将对信号处理作一简述，再下一节再继续我们的例子。

5.8　Posix 信号处理

信号是发生某事件时对进程的通知，有时称为软中断。它一般是异步的，这就是说，进程不可能提前知道信号发生的时间。

信号可以

- 由一进程发往另一进程(或本身)；

· 由内核发往某进程。

前一节结尾我们提到的信号 SIGCHLD 就是内核在某进程终止时发给此终止进程的父进程的信号。

每个信号都有一个处理办法(disposition),也称为与信号关联的行为(action)。我们通过调用函数 sigaction(不久将讨论到)来设置一个信号的处理办法,并有三个选择。

1. 我们可以提供一个函数,在信号发生时随即调用。这个函数称为信号处理程序(signal handler),而此行为则称为捕获(catching)信号。有两个信号不能被捕获,它们是 SIGKILL 和 SIGSTOP。我们的函数由信号值这个单一的整数参数来调用且无返回值,所以其函数原型为:

```
void handler (int signo);
```

对于多数信号来说,调用函数 sigaction 并指定信号发生时调用的函数是捕获信号所需做的所有事情,但我们稍后将看到,有些信号,如 SIGIO、SIGPOLL 和 SIGURG,还要求捕获它们的进程要有其他动作。

2. 我们可以通过设置信号的处理办法为 SIG_IGN 来忽略它,但 SIGKILL 和 SIGSTOP 这两个信号不能忽略。
3. 我们可以设置信号的处理办法为 SIG_DFL 来为它设置缺省处理办法。一般地说,缺省处理办法是在接收到信号时终止进程,特定信号还在当前工作目录产生一个进程的核心映像。个别信号的缺省处理办法是忽略;SIGCHLD 和 SIGURG(带外数据到达时发送,见第 21 章)就是本书中要用的缺省处理办法是忽略的两个信号。

signal 函数

建立信号的处理办法的 Posix 方法就是调用函数 sigaction。不过这有点复杂,因为该函数的一个参数是我们必须分配并填写的结构。简单些的方法就是调用函数 signal,其第一个参数是信号名,第二个参数或为指向函数的指针,或为常值 SIG_IGN 或 SIG_DFL。但是函数 signal 是一个历史较长的函数,它早于 Posix.1。调用时,不同的实现提供不同的信号语义以实现后向兼容性,而 Posix 则明确规定调用函数 sigaction 时的信号语义。解决的方法就是定义我们自己的函数,也称为 signal,它就调用 Posix 函数 sigaction,这样用 Posix 语义提供了一个简单的接口。我们在自己的函数库中包含了这个函数以及 err_XXX 函数和包裹函数等等,构造本书中我们的程序时,我们都指定了这个函数库。① 此函数如图 5.6 所示。

```
1 #include    "unp.h"
2 Sigfunc *
3 signal(int signo,Sigfunc *func)
4 {
5     struct sigaction act, oact;
6     act.sa_handler = func;
7     sigemptyset(&act.sa_mask);
8     act.sa_flags = 0;
```

① 译者注: 构造程序是指使用 make 工具把源程序和/或目标程序编译链接成可执行程序。本书免费可得的源代码(见前言)提供了构造其中各个程序的 makefile 文件。

```
 9    if(signo == SIGALRM) {
10 #ifdef  SA_INTERRUPT
11         act.sa_flags |= SA_INTERRUPT;     /* SunOS 4.x */
12 #endif
13         } else {
14 #ifdef    SA_RESTART
15           act.sa_flags |= SA_RESTART;     /* SVR4, 4.4BSD */
16 #endif
17    }
18    if (sigaction(signo,&act,&oact) < 0)
19        return (SIG_ERR);
20    return(oact.sa_handler);
21 }
```

图 5.6　调用 Posix sigaction 函数的 signal 函数[lib/signal.c]

用 typedef 简化函数原型

第 2～3 行　函数 signal 的正常函数原型因层次太多而变得很复杂：

```
void (*signal (int signo, void (*func)(int)))(int);
```

为了简化它，在头文件 unp.h 中我们定义类型 sigfunc 为

```
typedef void Sigfunc(int);
```

它说明信号处理程序是带有一个整型参数且无返回值的函数。这样，signal 的函数原型就成为：

```
Sigfunc *signal (int signo, Sigfunc *func);
```

此函数的第二个参数和返回值都是指向信号处理函数的指针。

设置处理程序

第 6 行　sigaction 结构的元素 sa_handler 被置为 func 参数。

设置处理程序的信号掩码

第 7 行　Posix 允许我们指定这样一组信号，它们在信号处理程序被调用时阻塞②。任何阻塞的信号都不能递交(delivering)给进程。我们设置成员 sa_mask 为空集，这意味着当信号处理程序运行时没有别的信号阻塞。Posix 保证信号处理程序正在运行时被捕获的信号是阻塞的。

设置标志 SA_RESTART

第 8～17 行　有一个可选标志 SA_RESTART，如果设置，由此信号中断的系统调用将由内核自动重启。(在下一节继续我们的例子时，将详细地讨论中断的系统调用。)如果被捕获的信号不是 SIGALRM，我们将设置标志 SA_RESTART。(将 SIGALRM 作为特殊情况的原因是：生成此信号的目的是为 I/O 操作设置超时，这将在 13.2 节中进行讨论，这时我们

② 译者注：这里的阻塞不同于我们此前一直使用的同名词。这里的阻塞是指阻塞某个信号或某个信号集，防止它们在阻塞期间递交(delivering)。它的反操作称为解阻塞。而此前一直使用的阻塞是指阻塞在某个系统调用上，也就是说这个系统调用因为目前没有必要的资源可用而必须等待，直到这些资源变为可用后才可能返回。等待期间进程进入睡眠状态。与这个阻塞相对的概念是非阻塞，也就是说非阻塞的系统调用即使没有必要的资源可用也立即返回，不过会告诉调用者发生了这种情况，这样调用者可以继续调用同一个系统调用。读者应注意区分这两个同名异义词。

希望阻塞的系统调用被信号中断。)较早期的系统,如 SunOS 4.x,缺省就是自动重启被中断的系统调用,而它的互补标志则定义成 SA_INTERRUPT。如果定义了这个标志,在捕获的信号是 SIGALRM 时我们就设置它。

调用函数 sigaction

第 18～20 行 我们调用函数 sigaction,并将信号的旧行为作为函数 signal 的返回值。本书通篇使用图 5.6 的函数 signal。

Posix 信号语义

我们对 Posix 兼容系统上的信号处理作以下总结:

- 一旦安装了信号处理程序,它便一直安装着(较早期的系统是每执行一次就将其拆除)。
- 当一个信号处理程序正在执行时,所递交的信号是阻塞的,而且,安装处理程序时定义在信号集 sa_mask 中传递给函数 sigaction 的任何额外信号也是阻塞的。在图 5.6 中,我们将 sa_mask 置为空集,这意味着除了被捕获的信号外,没有额外信号阻塞。
- 如果一个信号在阻塞时生成了一次或多次,在信号解阻塞后一般只递送一次。也就是说,缺省时 Unix 信号是不排队的,下一节我们将看到这样的一个例子。Posix 实时标准 1003.1b 定义了一组排队的可靠信号,但本书中我们不使用。
- 利用函数 sigprocmask 来有选择性地阻塞或不阻塞一组信号是可能的,这使得我们可以在某段临界区代码执行时,不许捕获某些信号,以此来保护这段代码。

5.9 处理 SIGCHLD 信号

设置僵尸(Zombie)状态的目的就是维护子进程的信息,以便父进程在稍后的某个时候取回。此信息包括子进程的 ID、终止状态以及子进程的资源利用信息(CPU 时间、内存等等)。如果一个进程终止,且该进程有子进程处于僵尸状态,则所有僵尸子进程的父进程 ID 均置为 1(init 进程),init 进程将作为这些子进程的继父并负责清除它们(也就是说,init 进程将 wait 它们,从而去除僵尸进程)。有些 Unix 系统给僵尸进程输出的 COMMAND 列为<defunct>(ps 命令输出)。

处理僵尸进程

显然,我们不愿看见僵尸进程,它们占用内核空间,最终导致我们无法正常工作。无论何时我们创建子进程都必须等待,以免变成僵尸进程。为此,我们建立了一个信号处理程序来捕获信号 SIGCHLD,在处理程序中我们调用 wait。(我们将在 5.10 节中介绍函数 wait 和 waitpid。)我们在图 5.2 所示的代码中调用 listen 之后,增加函数调用:

```
Signal (SIGCHLD, sig_chld);
```

来建立信号处理程序(这必须在创建第一个子进程之前完成,且只做一次)。然后,我们定义信号处理程序即函数 sig_chld,如图 5.7 所示。

```
1 #include    "unp.h"
2 void
3 sig_chld(int signo)
4 {
5     pid_t    pid;
6     int      stat;
7     pid = wait(&stat);
8     printf("child %d terminated\n",pid);
9     return;
10 }
```

图 5.7　调用 wait 的 SIGCHLD 信号处理程序[tcpcliserv/sigchldwait.c]

警告：在信号处理程序中调用诸如 printf 这样的标准 I/O 函数是不合适的，原因将在 11.14 节讨论。在这里我们调用了函数 printf 是为了把它作为一个诊断工具，看看子进程是什么时候终止的。

在系统 V 和 Unix 98 中，如果进程将 SIGCHLD 的处理方法置为 SIG_IGN，它的子进程就不会成为僵尸进程。遗憾的是，这仅仅适用于系统 V 和 Unix 98，Posix.1 则明确表示它不支持这一点。另一个处理僵尸进程的方法就是捕获 SIGCHLD，并调用 wait 或 waitpid。

如果我们在 Solaris 2.5 下编译这样一个程序：以图 5.2 中代码为基础，加上对 Signal 的调用以及我们的信息处理程序 sig_chld，且所用的函数 signal 是系统库中的(不是图 5.6 中的)，我们有如下结果：

```
solaris % tcpserv02 &                    在后台启动服务器
[2]     16939
solaris % tcpcli01 127.0.0.1             再在前台启动客户
hi there                                 我们键入这一行
hi there                                 这一行被回射回来
^D                                       我们键入 EOF 字符
child 16942 terminated                   这是信号处理程序中 printf 的输出
accept error:interrupted system call     然而 main 函数终止执行
```

具体的各个步骤如下：

1. 我们键入 EOF 字符来终止客户，客户 TCP 发一个 FIN 给服务器，服务器以 ACK 响应。
2. FIN 的接收导致服务器 TCP 递送一个 EOF 给子进程阻塞中的 readline，从而子进程终止。
3. 当信号 SIGCHLD 递交时，父进程阻塞于 accept 调用。函数 sig_chld(信号处理程序)执行，它调用函数 wait 取到子进程的 PID 和终止状态，再调用 printf，然后返回。
4. 由于该信号是在父进程阻塞于慢系统调用(accept)时由父进程捕获的，所以内核将使 accept 返回一个 EINTR 错误(被中断的系统调用)，而父进程不处理此错误(图 5.2)，所以中止。

本例的目的是为了说明，在编书写捕获信号的网络程序时，我们必须认清被中断的系统调用且处理它们。在 Solaris 2.5 环境下运行的这个特定例子中，标准 C 库中提供的函数 sig-

nal 不会让内核自动重启中断的系统调用。也就是说，我们在图 5.6 中设置的标志 SA_RESTART 并没由系统库中的函数 signal 设置的。也有的系统可以自动重启中断的系统调用。如果我们在 4.4BSD 环境下运行上述例子，用函数 signal 的库版本，内核将重启中断的系统调用，于是 accept 不会返回错误。为了处理不同操作系统间这个潜在的问题，我们有理由定义一个自己的函数 signal(图 5.6)，并在全书使用。

在我们的信号处理程序中(图 5.7)，总是有一个显式的 return 语句，即使到了函数结尾也不例外，而信号处理程序并没有返回值(返回值类型为 void)。这么做有助于产生这样的警告：信号处理程序的 return 语句可能中断了某个系统调用。

处理被中断的系统调用

我们已经用术语慢系统调用(slow system call)来描述函数 accept，对于那些可能永远阻塞的系统调用我们也可用此术语。永远阻塞的系统调用是指调用有可能永远无法返回，多数网络支持函数都属于这一类。譬如，如果没有客户连接到服务器上，则服务器对 accept 的调用就没有返回保证。类似地，在图 5.3 中，如果客户从未发送过一行要求服务器回射的文本，则服务器对 read(通过 readline)的调用将永不返回。其他慢系统调用的例子是对管道和终端设备的读和写。有一个例外值得注意，它就是磁盘 I/O，它一般都返回调用者(假设没有灾难性的硬件故障)。

应用于这儿的基本规则是：当一个进程阻塞于慢系统调用时捕获到一个信号，等到信号处理程序返回时，系统调用可能返回一个 EINTR 错误。有些内核自动重启某些被中断的系统调用。为了便于移植，当我们编写一个捕获信号的程序时(多数并发服务器捕获 SIGCHLD)，我们必须对慢系统调用返回 EINTR 有所准备。移植性问题是由早期使用的修饰词“可能”、“有些”和对 Posix 标志 SA_RESTART 的支持可选造成的。即使某实现支持标志 SA_RESTART，也并非所有的被中断系统调用都可自动重启，譬如说，大多数源自 Berkeley 的实现从不自动重启 select，有些实现从不重启 accept 和 recvfrom。

为了处理一个被中断的 accept，我们将图 5.2 中对 accept 的调用从 for 循环开始改起，如下所示：

```
for ( ; ; ) {
    clilen = sizeof(cliaddr);
    if ( (connfd = accept(listenfd, (SA *) &cliaddr,&clilen))< 0) {
        if (errno == EINTR)
            continue;          /* back to for() */
        else
            err_sys ("accept error");
    }
```

注意，我们调用的是函数 accept 而不是包裹函数 Accept，因为我们必须自己处理函数的失败情况。

在这段代码中，我们所做的事情就是自己重启被中断的系统调用，这对于 accept 以及诸如 read、write、select 和 open 这样的函数是合适的，但有一个函数我们自己不能重启：connect。如果这个函数返回 EINTR，我们就不能再调用它，否则将立即返回错误。当 connect 被一个捕获的信号中断而且不自动重启(TCPv2 第 466 页)时，我们必须调用 select 来等待连

接完成，如 15.3 节所述。

5.10　wait 和 waitpid 函数

在图 5.7 中，我们调用了函数 wait 来处理终止的子进程。

```
#include <sys/wait.h>
    pid_t wait (int *statloc);
    pid_t waitpid(pid_t pid, int *statloc, int options);
                                        二者均返回：进程 ID 为 0 成功，或为－1 出错
```

函数 wait 和 waitpid 均返回两个值：函数的返回值是终止子进程的进程 ID 号，子进程的终止状态（一个整数）则是通过指针 statloc 返回的。有三个宏我们可以调用，它们检查终止状态，并告诉我们子进程是否正常终止，是由信号杀死还是仅仅作业控制而停止。还有些宏让我们可以取得子进程的退出状态，杀死进程的信号值或停止进程的作业控制信号值。在图 14.10 中，我们用宏 WIFEXITED 和 WEXITSTATUS。

如果没有终止的子进程让进程来调用 wait，但有一个或多个正在执行的子进程，则 wait 阻塞直到第一个现有子进程终止。

函数 waitpid 对等待哪个进程及是否阻塞给了我们更多的控制。首先，参数 pid 让我们指定想等待的进程 ID，值－1 表示等待第一个终止的子进程。（也有其他选项，它们涉及到进程组 ID，本书中不需考虑。）参数 options 让我们指定附加选项。最常用的选项是 WNOHANG，它通知内核在没有已终止子进程时不要阻塞。

函数 wait 和 waitpid 的区别

现在，我们图示出函数 wait 和 waitpid 在用来清除终止子进程时的区别。为此，我们将 TCP 客户修改为如图 5.8 所示，客户与服务器建立 5 个连接，在调用函数 str_cli 时仅用第一个连接（sockfd[0]）。建立多个连接的目的就是要从并发服务器上创建多个子进程，如图 5.9 所示。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int         i,sockfd[5];
6     struct      sockaddr_in servaddr;
7     if (argc ! = 2)
8         err_quit("usage:tcpcli <IPaddress>");
9     for(i = 0;i < 5; i++) {
10        sockfd[i] = Socket(AF_INET,SOCK_STREAM,0);
11        bzero(&servaddr,sizeof(servaddr));
12        servaddr.sin_family = AF_INET;
13        servaddr.sin_port = htons (SERV_PORT);
14        Inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
```

```
15          Connect(sockfd[i],(SA *) &servaddr,sizeof(servaddr));
16      }
17      str_cli(stdin,sockfd[0]);      /* do it all */
18      exit(0);
19  }
```

图 5.8 与服务器建立了 5 个连接的 TCP 客户程序[tcpcliserv/tcpcli04.c]

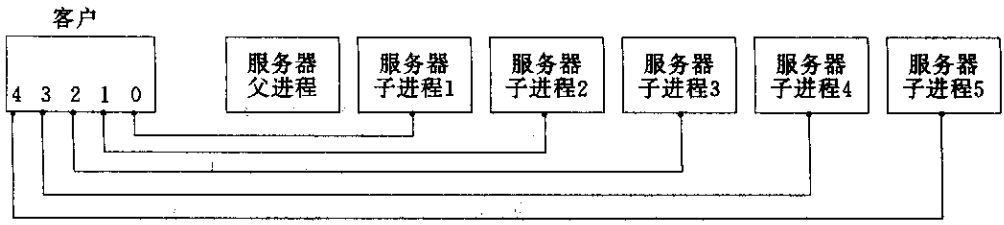


图 5.9 与同一个并发服务器建立了 5 个连接的客户

当客户终止时,所有打开的描述字由内核自动关闭(我们不调用 close,仅调用 exit),且所有 5 个连接基本在同一时刻终止。这就引发了 5 个 FIN,每个连接一个,它们反过来使服务器的 5 个子进程基本在同一时刻终止。这又导致差不多在同一时刻递交 5 个 SIGCHLD 信号给父进程,如图 5.10 所示。

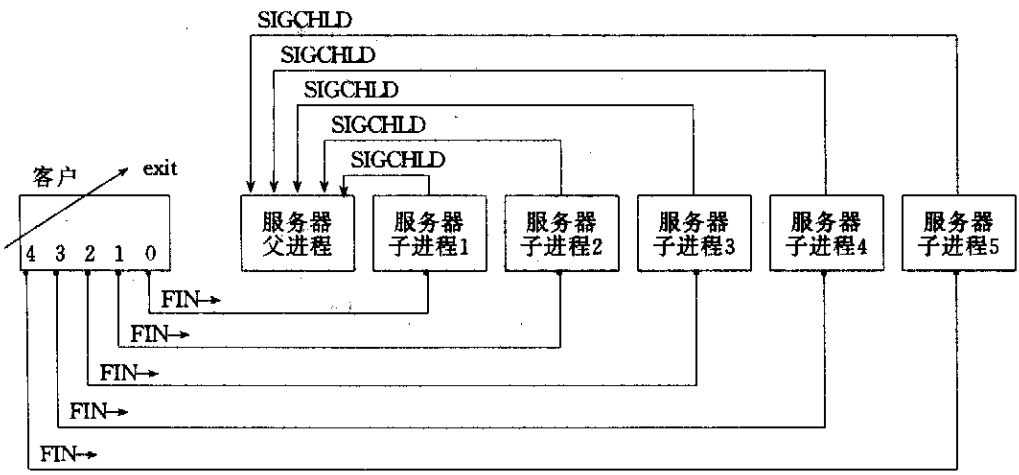


图 5.10 客户终止导致关闭 5 个连接,终止 5 个子进程

正是这种同一信号多个实例的递交造成了问题,这我们不久将看到。

首先,我们在后台运行服务器,接着运行新客户。我们的服务器是由图 5.2 修改而来,它调用函数 signal 来建立如图 5.7 所示的 SIGCHLD 信号处理程序。

```
bsdi % tcpserv03 &
[1]21282
bsdi % tcpcli04 206.62.226.35
hello                          我们键入这一行
hello                          这一行被回射回来
^D                             我们再键入 EOF 字符
child 21288 terminated         这是服务器的输出
```

我们注意到的第一件事情是，当我们期望五个子进程都终止时，只有一个 printf 函数输出了。如果运行 ps，我们将发现其他四个子进程仍作为僵尸进程存在。

```
  PTD     TT   STAT      TIME COMMAND
21282     p1   S      0:00.09 tcpserv03
21284     p1   Z      0:00.00 (tcpserv03)
21285     p1   Z      0:00.00 (tcpserv03)
21286     p1   Z      0:00.00 (tcpserv03)
21287     p1   Z      0:00.00 (tcpserv03)
```

建立一个信号处理程序并在其中调用 wait 并不足以防止出现僵尸进程，问题在于：所有 5 个信号都在信号处理程序执行之前产生，而此信号处理程序又只执行一次，因为 Unix 信号一般是不排队的。更严重的是，此问题是不确定的。在我们刚刚运行的例子中，客户与服务器在同一主机上，信号处理程序执行一次，留下四个僵尸进程。但若我们在不同的主机上运行客户和服务器，信号处理程序一般执行两次：一次作为第一个产生的信号的结果，由于另外 4 个信号在信号处理程序执行时发生，所以处理程序一般情况下会再被调用一次，这就留下了三个僵尸进程。但有时，可能依赖于 FIN 到达服务器主机的时机，信号处理程序执行三次或四次。

正确的解决办法是调用 waitpid 而不是 wait，图 5.11 给出了正确处理 SIGCHLD 的函数 sig_chld 版本。这个版本之所以正确是因为我们在循环内调用 waitpid 取得了所有已终止子进程的状态。我们必须指定选项 WNOHANG，它告诉 waitpid 在有未终止的子进程运行时不要阻塞。图 5.7 中，我们不能在循环中调用 wait，因为没有办法防止 wait 在有未终止的子进程运行时阻塞。

图 5.12 给出了我们的服务器程序的最终版本，它正确处理 accept 返回的 EINTR，且建立一个为所有的已终止子进程调用 waitpid 的信号处理程序(图 5.11)。

```
1 #include    "unp.h"
2 void
3 sig_chld(int signo)
4 {
5     pid_t    pid;
6     int      stat;
7     while( (pid = waitpid(-1, &stat,WNOHANG)) > 0)
8         printf("child %d terminated\n",pid);
9     return;
10 }
```

图 5.11　调用函数 waitpid 的 sig_chld 函数最终(正确)版本[tcpcliserv/sigchldwaitpid.c]

本节的目的是示范我们在网络编程时可能会遇到的三种情况：

1. 当派生子进程时，必须捕获信号 SIGCHLD。
2. 当捕获信号时，必须处理被中断的系统调用。
3. SIGCHLD 的信号处理程序必须正确编写，应使用函数 waitpid 以免留下僵尸进程。

我们的 TCP 服务器程序最终版本(图 5.12)和图 5.11 的 SIGCHLD 信号处理程序能处理上述三种情况。

```
1 #include    "unp.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int     listenfd, connfd;
6     pid_t   childpid;
7     socklen_t clilen;
8     struct sockaddr_in cliaddr, servaddr;
9     void    sig_chld(int);
10    listenfd = Socket(AF_INET, SOCK_STREAM, 0)
11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14    servaddr.sin_port = htons(SERV_PORT);
15    Bind(listenfd, (SA * ) &servaddr, sizeof(servaddr));
16    Listen(listenfd, LISTENQ);
17    Signal(SIGCHLD, sig_chld);      /* must call waitpid() */
18    for( ; ; ) {
19        clilen = sizeof(cliaddr);
20        if ( (connfd = accept(listenfd, (SA * ) &cliaddr, &clilen)) < 0) {
21            if(errno == EINTR)
22                continue;      /* back to for() */
23            else
24                err_sys("accept error");
25            }
26            if( (childpid = Fork() ) == 0) {      /* child process */
27                  Close(listenfd);      /* close listening socket */
28                  str_echo(connfd);      /* process the request */
29                  exit(0);
30              }
31              Close(connfd)      /* parent closes connected socket */
32        }
33 }
```

图 5.12 处理 accept 返回 EINTR 错误的 TCP 服务器程序最终(正确)版本[tcpcliserv/tcpserv04.c]

5.11 accept 返回前连接夭折

有另外一种情况,类似于前一节中被中断的系统调用例子,它可能导致 accept 返回一个非致命的错误,此时我们只需再调用一次 accept。图 5.13 中所示的分组序列在较忙的服务器(比较典型的是较忙的 Web 服务器)上已出现过。

三路握手完成,连接建立,然后,客户 TCP 发一个 RST(复位)。在服务器端,连接由 TCP 排队,等待服务器进程在 RST 到达后调用 accept。稍后,服务器进程调用 accept。

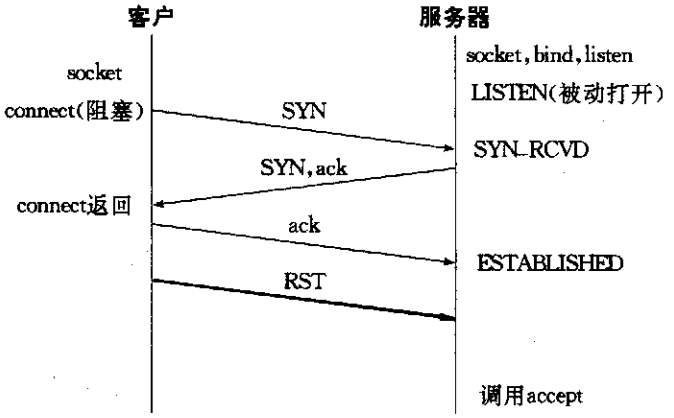


图5.13 ESTABLISHED状态的连接在调用accept之前收到RST

模拟这种情况的一个简单的方法就是:启动服务器,让它调用socket、bind和listen,然后在调用accept之前睡眠一小段时间。在服务器进程睡眠时,启动客户,让它调用socket和connect,一旦connect返回,就设置套接口选项SO_LINGER以生成RST(它将在7.5节中讨论,并在图15.21中给出一个例子),然后终止。

遗憾的是,对于夭折的连接将如何处理是依赖于不同的实现的。源自Berkeley的实现在内核中完成处理夭折的连接,服务器进程根本看不到它。然而,大多数SVR4实现返回一个错误给进程作为accept的返回,此错误与实现方式有关。这些SVR4实现返回一个EPROTO("协议错")errno值,而Posix.1g指出,返回必须是ECONNABORTED("软件引起的连接夭折")。Posix.1g作修改的理由是:当子系统中出现某些致命的协议相关事件时也返回EPROTO,为客户建立的连接非致命夭折而返回相同的错误使得服务器不可能知道是否该再次调用accept。在ECONNABORTED错误情况下,服务器可以忽略错误而简单地再调用accept一次。

Solaris 2.6实现了Posix.1g所做的修改。

源自Berkeley内核中从不传递此错误给进程的做法所涉及的步骤可在TCPv2中找到踪迹。RST在第964页做了处理,导致函数tcp_close被调用,此函数又调用第897页的函数in_pcbdetach,然后它又调用第719页的函数sofree。该函数(第473页)发现待夭折的套接口仍在监听套接口的已完成连接队列,于是从该队列中删去并释放此套接口。当服务器接着调用函数accept时,它根本不知道有一个已完成的连接已从队列中删除掉。

在15.6节我们将再次回到这些夭折的连接,看看在与函数select和正常阻塞模式下的监听套接口混用时如何出现问题的。

5.12 服务器进程终止

现在我们启动客户-服务器，然后杀死服务器子进程。这是为了模拟服务器进程的崩溃，来观察客户将会发生什么。(我们必须小心区别服务器进程崩溃与5.14节中讨论的服务器主机崩溃。)步骤如下：

1. 我们在不同的主机上启动服务器和客户，并在客户上键入一行，若正常，此行应由服务器子进程回射。
2. 找到服务器子进程的进程ID，并杀死该进程。作为进程终止处理的部分工作，子进程中打开的描述字都关闭，这引发了一个FIN发送给客户，客户TCP相应地以ACK响应。这就是TCP连接终止的前一半工作。
3. 信号SIGCHLD被发往服务器父进程并正确处理(图5.12)。
4. 客户上毫无动静。客户TCP从服务器TCP接收FIN并以ACK响应，但问题是客户进程正阻塞于fgets调用，等待从终端上得到一行。
5. 此时，在客户的另外一个窗口上运行命令netstat，以观察客户套接口的状态。

```
solaris % netstat | grep 9877
Local Address      Remote Address  Swind  Send-Q  Rwind Recv-Q  State
solaris.34673      bsdi.9877       8760       0    8760       0  CLOSE_WAIT
```

(这是第一次我们在solaris系统上给出命令netstat的输出，所以加上了标题行，其格式与BSD系统的输出格式略有不同，但内容是相似的。)同时，我们也在服务器的另一窗口上执行命令netstat：

```
bsdi % netstat | grep 9877
tcp        0      0  bsdi.9877          solaris.34673      FIN_WAIT_2
```

参看图2.4，我们可知，至此已完成了TCP连接终止序列的一半。

6. 我们可以在客户上再键入一行，在第一步中启动的客户将发生如下事件：

```
solaris % tcpcli01 206.62.226.35      启动客户
hello                                 我们键入第一行
hello                                 它被正确回射
                                      在这儿我们杀死服务器主机上的服务器子进程
                                      然后给客户键入下一行
another line
str_cli:server terminated prematurely
```

当我们键入"another line"时，函数ser_cli调用writen，客户TCP给服务器发送数据。这在TCP中是允许的，因为客户TCP接收到FIN只表示服务器进程已关闭了连接的服务器一端，它不再发送任何数据。接收到FIN并没有通知客户TCP服务器进程已终止(在这个例子中它是终止了)。在6.6节中当我们讨论到TCP的半关闭问题时再详细讨论这一点。

当服务器TCP接收到来自客户的数据时，由于先前打开那个套接口的进程已终止，所以以RST响应。我们可以用tcpdump来观察分组，以检查是否发出了RST。

7. 但是，由于客户进程在调用writen后立即调用readline，且由于第2步中接收的FIN，

readline立即返回0(文件结束符),所以客户进程将看不到RST。我们的客户不希望在此时接收文件结束符,所以以错误信息"server terminated prematurely(服务器过早终止)"退出。

8. 当客户终止时(在图5.5中调用err_quit),所有打开的描述字都关闭。

我们上述所讨论的一切也都与例子的时序有关。就像我们刚讨论的,当在不同的主机上运行客户和服务器时,从客户到服务器发送数据(即"another line")和客户接收到服务器的RST需几毫秒时间。这就是为什么客户调用readline返回0的原因,因为较早接收到的FIN现已准备好由客户读入。如果我们在相同的主机上运行客户和服务器,或在客户调用readline之前插入一段短暂停顿,那么接收到的RST就会优先于FIN,导致readline返回一个错误,且errno值为ECONNRESET("对方复位连接")。

本例的问题是,当FIN到达套接口时,客户阻塞于fgets调用。客户工作时有两个描述字——套接口和用户输入,它不能只阻塞于两个源中某个特定源输入(正如str_cli所写),而是应阻塞于任一源的输入。事实上,这正是函数select和poll的一个目的,这两个函数将在第6章中讨论。在6.4节中,我们重写函数str_cli,这样一旦杀死服务器子进程,客户就会收到FIN。

5.13 SIGPIPE信号

如果客户不理会函数readline返回的错误而写入更多的数据到服务器上又会怎样?这种情况是可能发生的,譬如,客户在读回任何东西之前对服务器写两次,而第一次写就引发了一RST。

所用规则是:当一个进程向接收了RST的套接口进行写操作时,内核给该进程发一个SIGPIPE信号。此信号的缺省行为就是终止进程,所以,进程必须捕获它以免不情愿地被终止。

进程不论是捕获了该信号并从其信号处理程序返回,还是不理会该信号,写操作都返回EPIPE错误。

一个在Usenet上经常问及的问题(FAQ)是如何在第一次写时捕获此信号而不是在第二次写时才见到。这是不可能的,接着我们上述讨论,第一次写引发了RST分节,第二次写引发了SIGPIPE信号。写一个已接收了FIN的套接口是可行的,但写一个已接收了RST的套接口则是错误的。

为了看清信号SIGPIPE会发生什么,我们对客户进行了修改,如图5.14所示。

```
1 #include    "unp.h"
2 void
3 str_cli(FILE *fp,int sockfd)
4 {
5     char    sendline[MAXLINE],recvline[MAXLINE];
6     while(Fgets(sendline,MAXLINE,fp) != NULL){
```

```
7          Writen(sockfd,sendline,1);
8          sleep(1);
9          Writen(sockfd,sendline + 1,strlen(sendline) -1);
10         if(Readline(sockfd,recvline,MAXLINE) == 0)
11           err_quit("str_cli:server terminated prematurely");
12         Fputs(recvline,stdout);
13     }
14 }
```

图5.14 调用 writen 两次的函数 str_cli[tcpcliserv/str_cli11.c]

第7～9行 我们所做的修改就是调用 writen 两次：第一次将数据的第一个字节写到套接口，暂停一秒钟，第二次再写同一行中剩余的字节。目的就是为了使第一次调用 writen 引发一个 RST，而第二次调用则生成 SIGPIPE。

如果我们在 BSD/OS 主机上运行客户，则有：

```
bsdi % tcpcli11 206.62.226.34
hi there                    我们键入这一行
hi there                    它由服务器回射回来
                            在这儿我们杀死服务器子进程
bye                         然后键入这一行
bsdi % echo $?              最后一个命令的 ksh 返回值是多少?
269                         269 = 256 + 13
bsdi % grep SIGPIPE /usr/include/sys/signal.h
#define SIGPIPE 13     /* write on a pipe with no one to read it */
```

我们启动客户，键入一行，可以看到此行被正确回射，然后在服务器主机上终止服务器子进程。接着，键入另外一行("bye")，但无任何回射，都回到了 shell 提示符下。由于信号 SIGPIPE 的缺省行为是终止进程，不产生 core 文件，所以 Korn Shell 不输出任何东西。这就是由 SIGPIPE 所终止的程序的问题所在：甚至 shell 都不输出任何内容以指示发生了什么。

我们必须执行 echo $?以输出 shell 的返回值为269。然后，输出常值 SIGPIPE 的数字值，可以看到 Korn Shell 的返回值是256加上该信号值。但如果我们在 Digital Unix 4.0、Solaris 2.5或 UnixWare 2.1.2上执行此程序，KornShell 的返回值是141，即128加上13。

> Korn Shell 的11/16/88版返回128加上信号值，而较新的版本返回256加上信号值。Posix.2仅仅规定该返回值应大于128。其他 shell 还有可能返回其他值。

处理 SIGPIPE 的建议方法取决于它发生时应用进程想做什么。如果没有特殊的事情需做，则将信号处理办法很简单地设置为 SIG_IGN，并假设后续的输出操作将捕捉 EPIPE 错误并终止。如果信号出现时需采取特殊措施(可能需在日志文件中登记)，那么就必须捕获该信号，以便在信号处理程序中执行所有期望的动作。但是必须意识到，如果使用了多个套接口，该信号的递交无法告诉我们是哪个套接口出的错。如果我们确实需要知道是哪个 write 出了错，那么必须要么不理会该信号，要么从信号处理程序返回后再处理来自 write 的 EPIPE。

5.14 服务器主机崩溃

我们的下一种情况是看看在服务器主机崩溃时将发生什么。为了模拟这种情况，我们必须在不同的主机上运行客户和服务器。我们先启动服务器，再启动客户，然后在客户上键入一行以确认连接已建立，从网络上断开服务器主机后再在客户上键入另一行。这样一来同时也模拟了当客户发送数据时服务器主机不可达的情况(即建立连接后某些中间路由器不工作)。

步骤如下：

1. 当服务器主机崩溃时，已有的网络连接上发不出任何东西。这里我们假设主机崩溃，而不是由操作员执行命令关机(这在5.16节讨论)。
2. 我们在客户上键入一行，它由 writen(图5.5)所写，由客户 TCP 当作一个数据分节发出，然后，客户就阻塞于 readline 调用，等待回射。
3. 如果我们用 tcpdump 观察网络就会发现，客户 TCP 持续重传数据分节，试图从服务器上接收一个 ACK。TCPv2的25.11节给出了 TCP 重传一个典型模式：源自 Berkeley 的实现重传数据分节12次，放弃前等待约9分钟。当客户 TCP 最后终于放弃时(假设在这段时间内，服务器主机没有重新启动，或者如果服务器主机没有崩溃但网络不可达的情况，则假设网络仍然不可达)，返回客户进程一个错误。因为客户阻塞于 readline 的调用，它返回一个错误。假设服务器主机已崩溃，对客户的数据分节根本就没有响应，则错误为 ETIMEDOUT；但如果某些中间路由器判定服务器主机不可达，且以一个目的地不可达的 ICMP 消息响应，则错误是 EHOSTUNREACH 或 ENETUNREACH。

尽管最终我们的客户还是发现了对方已崩溃或不可达，但有时候我们需要比等待9分钟快许多地检测出这种情况。方法就是对 readline 的调用设置一超时，这在13.2节再作讨论。

我们刚刚讨论的情况只有在我们向服务器主机发送数据时，才能检测出它已经崩溃。如果我们不主动向它发送数据也想检测出服务器主机的崩溃，那需要采用另外一个技术。在7.5节我们将讨论套接口选项 SO_KEEPALIVE，在21.5节将讨论一些客户-服务器心博函数。

5.15 服务器主机崩溃后重启

在这种情况中，我们在客户与服务器之间建立连接，然后假设服务器主机崩溃并重启。在前一节中，当我们发送数据时，服务器主机仍处于崩溃状态，而本节，我们将在发送数据前重新启动服务器主机。模拟这种情况的最简单方法就是：建立连接，从网络上断开服务器，关闭服务器主机，然后重启服务器主机，重新连接服务器主机入网。我们不想让客户知道服务器主机的关机(这点我们将在5.16节讨论)。

正如前一节所提到的，如果客户在服务器主机崩溃时不主动发数据给服务器，那么客户是不会知道服务器已崩溃的(假设不用套接口选项 SO_KEEPALIVE)。采用步骤如下：

1. 我们启动服务器和客户，并键入一行以确认连接已建立。
2. 服务器主机崩溃并重启。
3. 在客户上键入一行，它将作为TCP数据分节发往服务器主机。
4. 当服务器主机崩溃后重启时，它的TCP丢失了崩溃前的所有连接信息，所以服务器TCP对接收的客户数据分节以RST响应。
5. 当RST到达时，客户阻塞于readline调用，导致它返回ECONNRESET错误。

如果客户检测服务器主机的崩溃与否很重要，即使客户不主动发送数据也这样，那就需有其他技术支持(诸如套接口选项SO_KEEPALIVE或某些客户-服务器心博函数)。

5.16 服务器主机关机

前面两节讨论了服务器主机崩溃或无法通过网络到达的情况，现在，我们考虑一下当服务器进程正在运行时，操作员关闭该服务器主机将会发生什么。

当Unix系统关机时，一般是由init进程给所有进程发信号SIGTERM(我们可以捕获此信号)，等待一段固定时间(常常是5秒～20秒)，然后给还在运行的所有进程发信号SIGKILL(此信号我们不能捕获)。这就给了所有运行进程一小段时间来清除和终止。如果我们不捕获信号SIGTERM和终止，服务器将由信号SIGKILL终止。当进程终止时，所有打开的描述字都关闭，然后按5.12节中所述步骤进行。 与在该一节所叙述的一样，我们必须在客户上使用函数select或poll，使得客户在服务器进程开始终止时就检测到。

5.17 TCP程序例子小结

在客户和服务器可以互相通信之前，每一方都必须指定连接的套接口对：本地IP地址、本地端口、远程IP地址、远程端口。在图5.15中我们给出了这四个值，此图是从客户的角度出发的。远程IP地址和远程端口必须在客户调用connect时指定，而两个本地值则一般是由内核作为connect的一部分来选定。客户也可在调用connect之前，通过调用bind来指定其中一个或全部(两个)本地值，但这不常用。

正如第4.10节所述，客户也可在连接建立后通过调用getsockname来获取由内核指定的两个本地值。

图5.16也给出了相同的四个值，但是这是从服务器的角度出发。

本地端口(服务器的众所周知端口)由bind指定。尽管服务器可以通过捆绑一个非通配的IP地址来限定接收目标为某个特定本地接口的连接，它通常是在bind调用中指定通配IP地址。如果服务器在一个多宿主机上绑定一个通配IP地址，它可以在建立连接后调用getsockname来确定本地IP地址，两个远程值则由accept来返回给服务器。我们在4.10节中已提到，如果另外一个程序由调用accept的服务器通过调用exec来执行，则这个新程序可以在必要时调用getpeername来确定客户的IP地址和端口号。

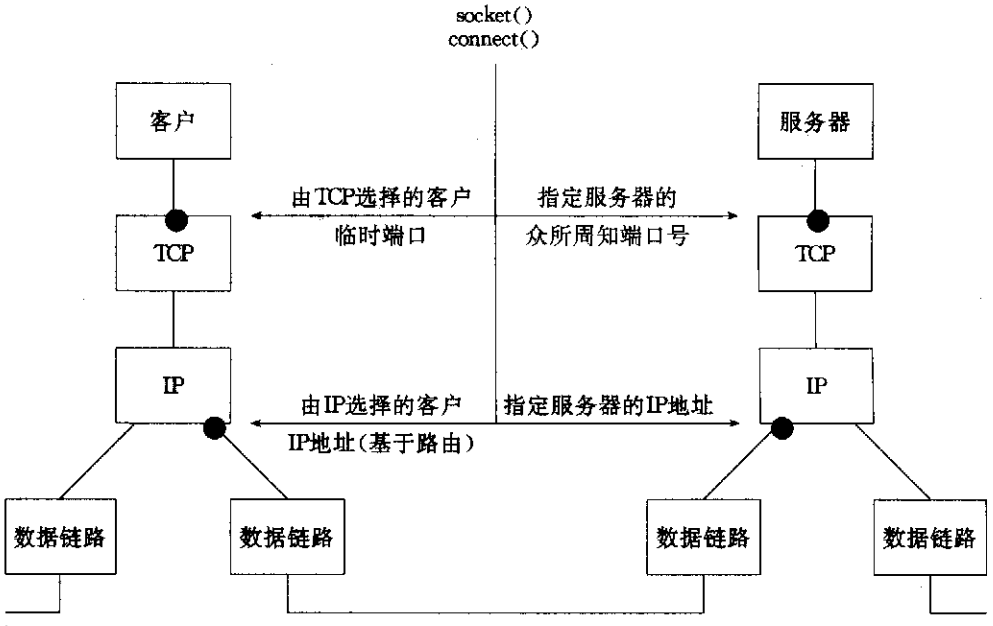


图5.15　从客户的角度总结 TCP 客户-服务器

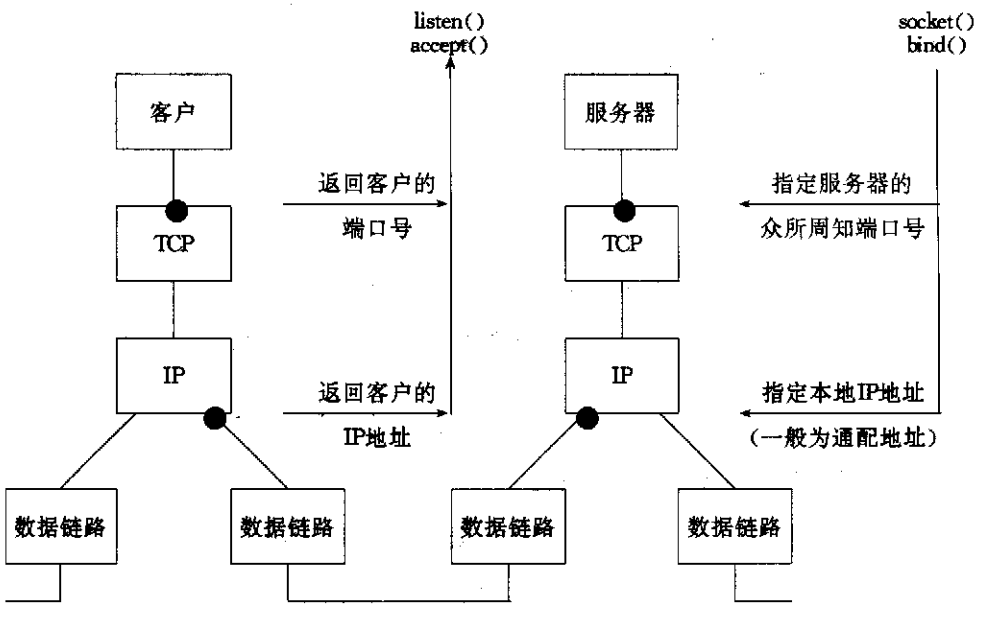


图5.16　从服务器的角度总结 TCP 客户-服务器

5.18 数据格式

在我们的例子中，服务器对从客户上接收的请求从不检查，它只管读入到换行符(包括换行符)的所有数据发回给客户，所搜索的仅仅是换行符。这只是一个例外而不是通常规则，一般来说，我们必须关心在客户和服务器之间进行交换的数据格式。

例子：在客户与服务器之间传递文本串

修改我们的服务器程序，它仍然从客户读入一行文本，但现在，服务器期望这一行包含由空格隔开的两个整数，服务器返回这两个整数的和。我们的客户和服务器 main 函数仍保持不变，str_cli 函数也保持不变，所有修改都在函数 str_echo 中，如图5.17所示。

```
1 #include    "unp.h"
2 void
3 str_echo(int sockfd)
4 {
5     long    arg1,arg2;
6     ssize_t n;
7     char    line[MAXLINE];
8     for( ; ;) {
9         if( (n = Readline(sockfd,line,MAXLINE)) == 0)
10            return;          /* connection closed by other end */
11        if(sscanf(line,"%ld%ld",&arg1,&arg2) == 2)
12            snprintf(line,sizeof(line),"%ld\n",arg1 + arg2);
13        else
14            snprintf(line,sizeof(line),"input error\n");
15        n = strlen(line);
16        Writen(sockfd,line,n);
17    }
18 }
```

图5.17 对两个数求和的函数 str_echo[tcpcliserv/str_echo08.c]

第11～14行 我们调用函数 sscanf 来将文本串中的两个参数转换为长整数，然后由函数 snprintf 来将结果转换为文本串。

不论客户和服务器主机的字节序如何，这个新的客户和服务器都能工作得很好。

例子：在客户与服务器之间传递二进制结构

现在，将我们的客户和服务器程序修改为跨套接口传递二进制值而不是文本串。我们将看到，当客户和服务器在不同字节序的主机上运行或是在不支持相同大小长整型的主机上运行时，客户和服务器便无法工作(图1.17)。

我们的客户和服务器程序 main 函数不作任何修改。我们给两个函数参数定义一个结构，给结果定义另外一个结构，并将这两个定义都放在头文件 sum.h 中，如图5.18所示。图5.19给出了函数 str_cli。

```
1 struct args {
2         long    arg1;
3         long    arg2;
```

```
4 };
5 struct result {
6         long      sum;
7 };
```

图5.18　头文件 sum.h[tcpcliserv/sum.h]

```
1 #include     "unp.h"
2 #include     "sum.h"
3 void
4 str_cli(FILE *fp,int sockfd)
5 {
6     char        sendline[MAXLINE];
7     struct args args;
8     struct result result;
9     while(Fgets(sendline,MAXLINE,fp) != NULL) {
10        if(sscanf(sendline,"%ld%ld",&args.arg1,&args.arg2) != 2) {
11          printf("invalid input: %s",sendline);
12          continue;
13        }
14        Writen(sockfd,&args,sizeof(args));
15        if (Readn(sockfd,&result,sizeof(result))==0)
16          err_quit("str_cli:server terminated prematurely");
17        printf("%ld\n",result.sum);
18    }
19 }
```

图5.19　发送两个二进制整数给服务器的函数 str_cli[tcpcliserv/str_cli09.c]

第10～14行　sscanf 将两个参数从文本串转换为二进制数，我们接着调用 writen 将此结构发送给服务器。

第15～17行　我们调用 readn 来读应答，并用 printf 来输出结果。

图5.20给出了函数 str_echo。

```
1 #include     "unp.h"
2 #include     "sum.h"
3 void
4 str_echo(int sockfd)
5 {
6     ssize_t n;
7     struct args args;
8     struct result result;
9     for( ; ;) {
10        if( (n = Readn(sockfd,&args,sizeof(args))) == 0)
11            return;          /* connection closed by other end */
12        result.sum = args.arg1 + args.arg2;
13        Writen(sockfd,&result,sizeof(result));
14    }
15}
```

图5.20　对两个二进制整数求和的函数 str_echo[tcpcliserv/str_echo09.c]

第9～14行　我们通过调用 readn 来读参数、计算并存储两数之和，然后调用 writen 将结果结构发回。

如果我们在具有相同体系结构的主机上运行我们的客户和服务器(如图1.16中的solaris和sunos5)则什么问题都没有。下面是客户的交互过程:

```
sunos5 % tcpcli09 206.62.226.33
11  22                          我们键入这两个数
33                              这个数是服务器的应答
-11  -44
-55
```

但是,当我们在具有不同体系结构的两台主机上运行客户和服务器时(服务器在大端sparc系统solaris上运行,而客户在小端Intel系统bsdi上运行),它们就无法工作了。

```
bsdi % tcpcli09 206.62.226.33
1  2                           我们键入这两个数
3                              结果正确
-22  -77                       我们再键入另外两个数
-16777314                      结果错误
```

问题就在于跨套接口发送的两个二进制整数在客户上是小端格式,但被服务器解释为大端格式。我们发现,对于正整数,它看起来能工作,但对于负整数则不行了(见习题5.8)。此例有三个潜在的问题:

1. 不同的实现以不同的格式存储二进制数,最常用的方法便是大端格式与小端格式,这在第3.4节已做过介绍。
2. 不同的实现在存储相同的C数据类型时可能不同。例如,大多数32位的Unix系统用33位表示长整数,但64位系统却常常以64位来表示此相同的数据类型(图1.17)。对short、int或long数据类型的大小没有确定的保证。
3. 不同的实现给结构打包的方式也是不同的,取决于所用数据类型的位数及机器的对齐限制。因此,跨套接口来传送二进制结构是很不明智的。

有两个常用方法来解决此数据格式问题:

1. 把所有的数值数据作为文本串来传递,如图5.17所示,当然这也要以两个主机有相同的字符集为基础。
2. 显式定义所支持数据类型的二进制格式(位数,大端或小端),在客户与服务器之间以此格式传递所有数据。远程过程调用(RPC)软件包常用此技术。RFC 1832[Srinivasan 1995]描述了Sun RPC软件包所用的外部数据表示(XDR,External Data Representation)标准。

5.19 小 结

我们的回射客户-服务器程序第一版总共约150行(包括函数readline和writen),然而留下了待检查的许多细节问题。我们遇到的第一个问题是僵尸进程问题,通过捕获信号SIGCHLD来处理。接着,信号处理程序调用waitpid,我们已证明必须调用此函数而不是早期的wait函数,因为Unix信号是不排队的。这引导我们进入了Posix信号处理的一些细节问题,关于这一问题在APUE的第10章中还有许多附加信息。

我们遇到的下一问题是当服务器进程终止时，客户不知道。我们看到，客户的TCP是被通知了，但客户本身由于正阻塞于等待用户输入而未接收到此通知。我们在第6章中将用函数select和poll来处理这种情况，它等待多个描述字中的任一个准备好而不是阻塞于单个描述字。

我们也发现，如果服务器主机崩溃，也要等到客户向该服务器发送了数据才能检测到。一些应用进程必须尽早意识到这个事实，在7.5节中我们将利用套接口选项SO_KEEPALIVE，在21.5节我们将开发一组客户-服务器心博函数来解决这个问题。

我们的简单例子是交换文本行，由于服务器根本不检查它回射的行，所以它工作得很好，但在客户与服务器之间发送数值数据时将引发一组新问题，文中已经讲述。

5.20 习 题

5.1 从图5.2和5.3中构造一个TCP服务器程序，从图5.4和图5.5中构造一个TCP客户程序。先启动服务器，然后启动客户，键入一新行以确认客户和服务器工作正常。键入文件结束符以终止客户，记下时间。在客户主机上使用命令netstat来查看连接的客户端是否进入TIME_WAIT状态，此后每5秒左右执行一次netstat来查看TIME_WAIT状态何时结束。此实现的MSL(最长分节生命期)是什么？

5.2 对于我们的客户-服务器程序，如果我们运行客户并将标准输入重定向到一个二进制文件将会怎样？

5.3 利用Telnet客户来与我们的回射服务器通信，它跟我们的回射客户-服务器有什么区别？

5.4 在5.12节的例子中，我们用命令netstat观察套接口状态来确认连接终止的前两个分节已发送(服务器发出的FIN和客户发出的对此分节的ACK)。后两个分节交换吗(客户发出的FIN和服务器发出的对此分节的ACK)？如果交换，它们何时交换？如果不交换，为什么？

5.5 在5.14节给出的例子中，如果我们在第二步与第三步之间，在服务器主机上重启服务器应用进程将会怎样？

5.6 为了验证在5.13节中我们所说的SIGPIPE发生的一切，我们对图5.4作如下修改。编写一个SIGPIPE信号处理程序，它仅输出一个消息便返回；在调用connect之前建立此信号处理程序。将服务器的端口号改为13，即daytime服务器。当连接建立后，睡眠2秒钟，写几个字节到套接口上，再睡眠2秒钟，写另外几个字节到套接口上。运行程序，观察它将会怎样？

5.7 在图5.15中，如果由客户在connect调用中将服务器主机的IP地址指定为与服务器主机最右边的数据链路相关的IP地址，而不是与服务器主机最左边的数据链路相关的IP地址，它将会怎样？

5.8 在图5.20的例子输出中，当客户和服务器分别在不同的大、小端系统上时，对于小的正整数，例子工作得很好，但对于小的负整数则不能正常工作，为什么？(提示：画一个与图3.8类似的跨套接口数字交换图。)

5.9 在图5.19和图5.20的例子中，我们可以通过让客户利用函数htonl来将两个参数

转换为网络字节序，让服务器在做加法前调用 ntohl 来转换每个参数，并对结果做类似的转换来解决字节序问题可以吗？

5.10 如果客户在某台以32位存储长整数的 Sparc 主机上，而服务器在以64位存储长整数的 Digital Alpha 主机上，图5.19和图5.20中的例子将会怎样？如果将客户和服务器在这两台主机上交换，结果会改变吗？

5.11 在图5.15中，我们说客户 IP 地址是由 IP 基于路由选定的，这是什么含义？

第 6 章　I/O 复用：select 和 poll 函数

6.1　概　述

在 5.12 节中，我们看到 TCP 客户同时处理两个输入：标准输入和 TCP 套接口。我们遇到的问题是客户阻塞于（标准输入上的）fgets 调用，而服务器进程又被杀死。服务器 TCP 虽正确地给客户 TCP 发了一个 FIN，但客户进程正阻塞于从标准输入读入，它直到从套接口读时才能看到此文件结束符（可能已过了很长时间）。我们需要这样的能力：如果一个或多个 I/O 条件满足（例如，输入已准备好被读，或者描述字可以承接更多的输出）时，我们就被通知到。这个能力被称为 I/O 复用，是由函数 select 和 poll 支持的，我们也对较新的 Posix.1g 的变种（称为 pselect）作介绍。

I/O 复用典型地用在下列网络应用场合：

- 当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用 I/O 复用，这在前一段中已做了描述。
- 一个客户同时处理多个套接口是可能的，但很少出现。在 15.5 节一个 Web 客户的上下文中，我们给出使用 select 的例子。
- 如果一个 TCP 服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到 I/O 复用，如 6.8 节所述。
- 如果一个服务器即要处理 TCP，又要处理 UDP，一般也要使用 I/O 复用，8.15 节中我们也要给出这样的一个例子。
- 如果一个服务器要处理多个服务或者多个协议（例如，我们将在 12.5 节描述的 inetd 守护进程），一般要使用 I/O 复用。

I/O 复用并非只限于网络编程，许多正式应用程序也需要使用这项技术。

6.2　I/O 模型

在介绍函数 select 和 poll 之前，我们需回过头来看看整体，检查 Unix 下我们可用的五个 I/O 模型的基本区别：

- 阻塞 I/O
- 非阻塞 I/O
- I/O 复用（select 和 poll）
- 信号驱动 I/O（SIGIO）
- 异步 I/O（Posix.1 的 aio_系列函数）

你在第一次阅读时，可能想略读本节，到后面的章节中详细介绍不同的I/O模型时才回来看。

正如本节我们所给出的所有例子所述，一个输入操作一般有两个不同的阶段：

1. 等待数据准备好。
2. 从内核到进程拷贝数据。

对于一个套接口上的输入操作，第一步一般是等待数据到达网络，当分组到达时，它被拷贝到内核中的某个缓冲区，第二步是将数据从内核缓冲区拷贝到应用缓冲区。

阻塞I/O模型

最流行的I/O模型是阻塞I/O模型，本书中到目前为止的所有例子都使用此模型。缺省时，所有套接口都是阻塞的。以数据报套接口作为例子，我们有示于图6.1中的情形。

此例中，我们用UDP而不是TCP，因为对于UDP来说，数据准备好的概念要简单些：整个数据报是否已接收。而对于TCP则要复杂得多，需考虑诸如套接口的低潮限度(lowwater mark)这样的许多附加变量。

在本节的例子中，我们将函数recvfrom视为系统调用，因为我们正考虑应用进程与内核的区别。不论函数recvfrom如何实现(在源自Berkeley的内核中作为系统调用，在系统V内核中作为调用系统调用getmsg的函数)，一般都有一个从应用进程中运行到内核中运行的切换，一段时间后再跟一个返回到应用进程的切换。

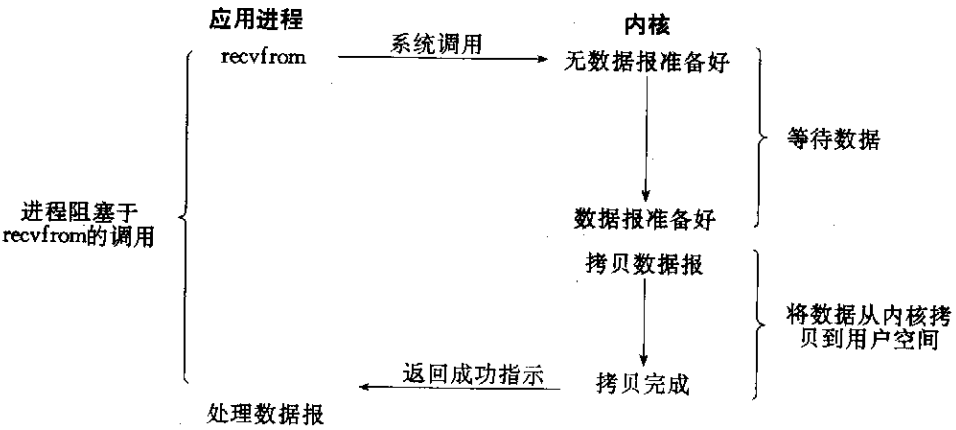


图6.1 阻塞I/O模型

在图6.1中，进程调用recvfrom，此系统调用直到数据报到达且拷贝到应用缓冲区或是出错才返回。最常见的错误是系统调用被信号中断，如5.9节所述。我们所说进程阻塞的整段时间是指从调用recvfrom开始到它返回的这段时间，当进程返回成功指示时，应用进程开始处理数据报。

非阻塞I/O模型

当我们把一个套接口设置成非阻塞方式时，即通知内核：当请求的I/O操作非得让进程睡眠不能完成时，不要让进程睡眠，而应返回一个错误。我们将在第15章详细介绍非阻塞I/

O，但为了说明我们所考虑的例子，在图 6.2 中作一个小结性描述。

前三次调用 recvfrom 时仍无数据返回，因此内核立即返回一个 EWOULDBLOCK 错误。第四次调用 recvfrom 时，数据报已准备好，被拷贝到应用缓冲区，recvfrom 返回成功指示，接着就是我们处理数据。

当一个应用进程像这样对一个非阻塞描述字循环调用 recvfrom 时，我们称此过程为轮询(polling)。应用进程连续不断地查询内核，看看某操作是否准备好，这对 CPU 时间是极大的浪费，但这种模型只是偶尔才遇到，一般是在只专门提供某种功能的系统中才有。

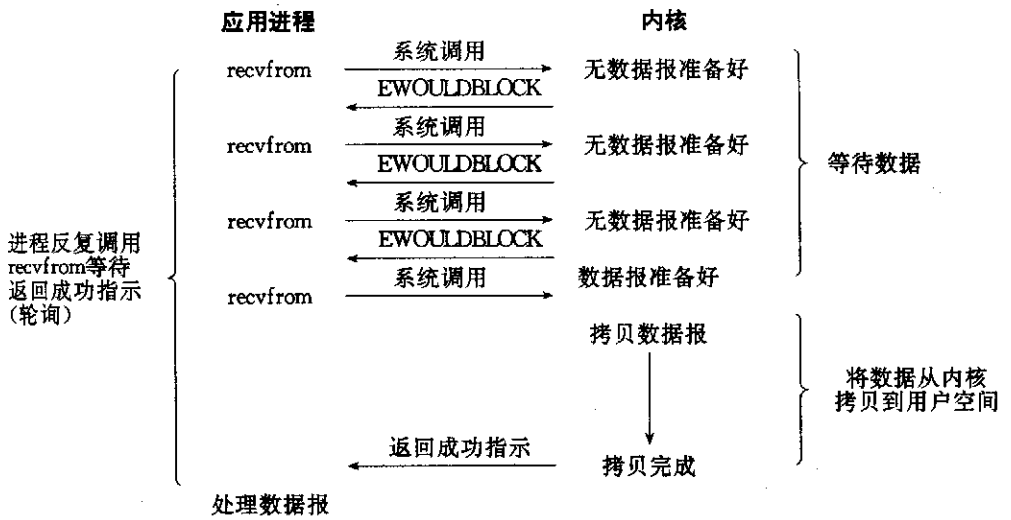


图 6.2　非阻塞 I/O 模型

I/O 复用模型

有了 I/O 复用，我们就可以调用 select 或 poll，在这两个系统调用中的某一个上阻塞，而不是阻塞于真正的 I/O 系统调用。图 6.3 是 I/O 复用模型的一个小结。

我们阻塞于 select 调用，等待数据报套接口可读。当 select 返回套接口可读条件时，我们调用 recvfrom 将数据报拷贝到应用缓冲区中。

将图 6.3 与图 6.1 进行比较，似乎没有显示什么优越性，实际上，因使用了系统调用 select，要求两次系统调用而不是一次，好像变得还有点差。但是，在本章的后面我们将看到，使用 select 的好处在于我们可以等待多个描述字准备好。

信号驱动 I/O 模型

我们也可以用信号，让内核在描述字准备好时用信号 SIGIO 通知我们，我们将此方法称为信号驱动 I/O，图 6.4 对此作了一个小结。

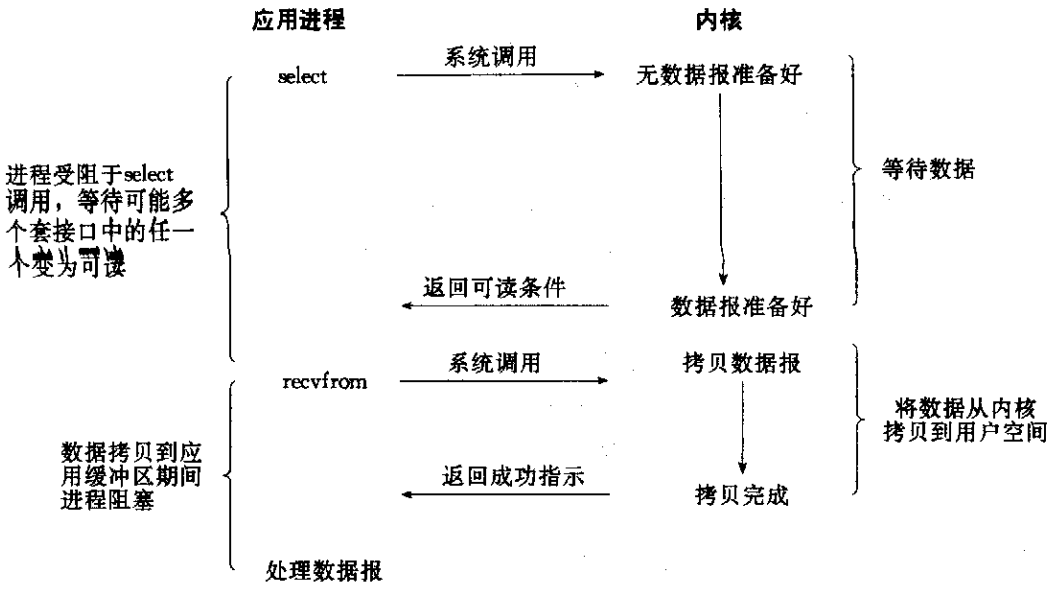


图 6.3 I/O 复用模型

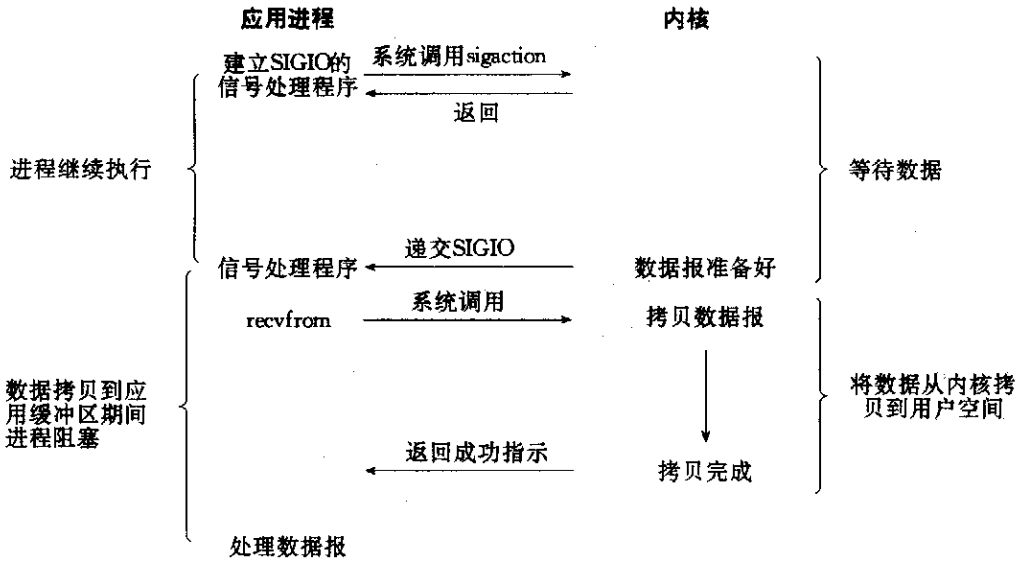


图 6.4 信号驱动 I/O 模型

首先，我们允许套接口进行信号驱动 I/O（我们将在 22.2 节对此进行讨论），并通过系统调用 sigaction 安装一个信号处理程序。此系统调用立即返回，进程继续工作，它是非阻塞的。当数据报准备好被读时，就为该进程生成一个 SIGIO 信号。我们随即可以在信号处理程序中调用 recvfrom 来读数据报，并通知主循环数据已准备好被处理（这正是我们在 22.3 节中所要做的事情）。也可以通知主循环，让它来读数据报。

无论我们如何处理 SIGIO 信号，这种模型的好处是当等待数据报到达时，可以不阻塞。主循环可以继续执行，只是等待信号处理程序的通知：或者数据已准备好被处理，或者数据报已准备好被读。

异步 I/O 模型

异步 I/O 是 Posix.1 的 1993 版本中的新内容（“实时”扩展）。我们让内核启动操作，并在整个操作完成后（包括将数据从内核拷贝到我们自己的缓冲区）通知我们。因为这种模型还没有广泛使用，本书不作讨论。这种模型与前一节介绍的信号驱动模型的主要区别在于：信号驱动 I/O 是由内核通知我们何时可以启动一个 I/O 操作，而异步 I/O 模型是由内核通知我们 I/O 操作何时完成。图 6.5 给出了一个例子。

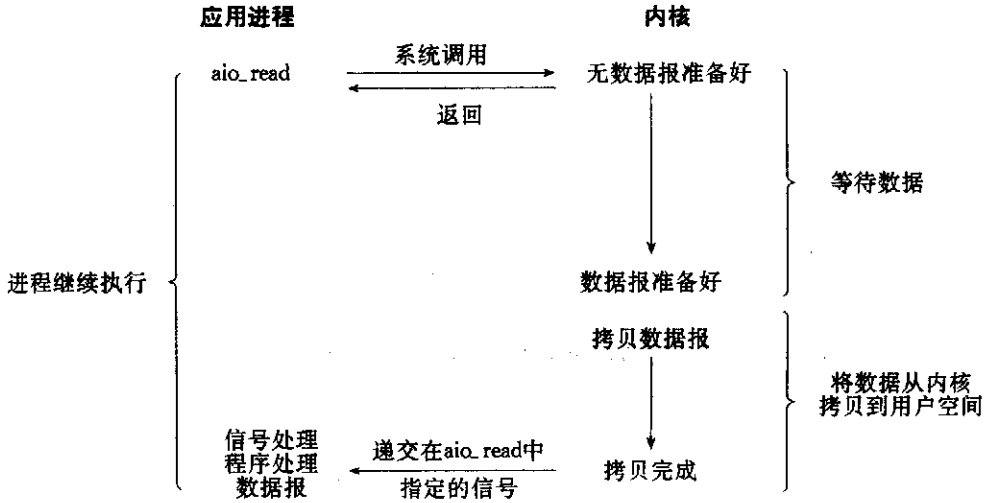


图 6.5　异步 I/O 模型

我们调用函数 aio_read（Posix 异步 I/O 函数以 aio_或 lio_开头），给内核传递描述字、缓冲区指针、缓冲区大小（与 read 相同的三个参数）、文件偏移（与 lseek 类似），并告诉内核当整个操作完成时如何通知我们。此系统调用立即返回，我们的进程不阻塞于等待 I/O 操作的完成。在此例子中，我们假设要求内核在操作完成时生成一个信号，此信号直到数据已拷贝到应用缓冲区才产生，这一点是与信号驱动 I/O 模型不同的。

> 正如本书所述，很少有系统支持 Posix.1 的异步 I/O 模型。例如，我们还不能确定系统是否支持套接口上的这种模型。这儿我们用它，只是作为一个与信号驱动 I/O 模型进行比较的例子。

各种 I/O 模型的比较

图 6.6 示出了上述五种不同 I/O 模型的比较。它表明：前四种模型的主要区别都在第一阶段，因为前四种模型的第二阶段基本相同：在数据从内核拷贝到调用者的缓冲区时，进程阻塞于 recvfrom 调用。然而，异步 I/O 模型处理的两个阶段都不同于前四个模型。

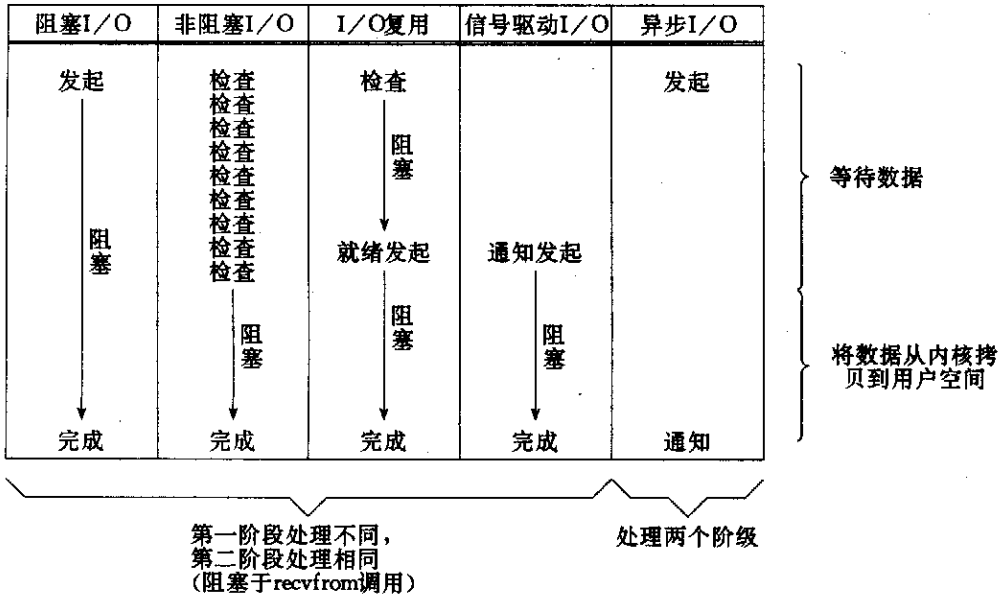


图 6.6 五个 I/O 模型的比较

同步 I/O 与异步 I/O

Posix.1 定义这两个术语如下：

- 同步 I/O 操作引起请求进程阻塞，直到 I/O 操作完成。
- 异步 I/O 操作不引起请求进程阻塞。

根据上述定义，我们的前四个模型——阻塞 I/O 模型、非阻塞 I/O 模型、I/O 复用模型和信号驱动 I/O 模型都是同步 I/O 模型，因为真正的 I/O 操作(recvfrom)阻塞进程，只有异步 I/O 模型与此异步 I/O 的定义相匹配。

6.3 select 函数

这个函数允许进程指示内核等待多个事件中的任一个发生，并仅在一个或多个事件发生或经过某指定的时间后才唤醒进程。

作为一个例子，我们可以调用函数 select 并通知内核仅在下列情况发生时才返回：

- 集合{1,4,5}中的任何描述字准备好读，或
- 集合{2,7}中的任何描述字准备好写，或
- 集合{1,4}中的任何描述字有异常条件待处理，或
- 已经过了 10.2 秒

也就是说，通知内核我们对哪些描述字感兴趣(读、写或异常条件)以及等待多长时间。我们所关心的描述字不受限于套接口：任何描述字都可用 select 来测试。

> 源自 Berkeley 的实现已经允许任何描述字的 I/O 复用。刚开始，SVR3 还限制 I/O 复用只适用于流设备(第 33 章)的描述字，但 SVR4 中就没有了这个限制。

```
#include <sys/select.h>
#include <sys/time.h>

int select (int maxfdp1,fd_set *readset,fd_set *writeset,fd_set * exceptset,const struct
        timeval *timeout);
                                返回：准备好描述字的正数目，0——超时，-1——出错
```

我们从此函数的最后一个参数开始介绍，它告诉内核等待一组指定的描述字中的任一个准备好可花多长时间，结构 timeval 指定了秒数和微秒数成员。

```
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};
```

有三种可能：

1. 永远等待下去：仅在有一个描述字准备好 I/O 时才返回，为此，我们将参数 timeout 设置为空指针。
2. 等待固定时间：在有一个描述字准备好 I/O 时返回，但不超过由 timeout 参数所指 timeval 结构中指定的秒数和微秒数。
3. 根本不等待：检查描述字后立即返回，这称为轮询(polling)。为了实现这一点，参数 timeout 必须指向结构 timeval，且定时器的值(由结构 timeval 指定的秒数和微秒数)必须为 0。

在前两者情况的等待中，如果进程捕获了一个信号并从信号处理程序返回，那么等待一般被中断。

> 源自 Berkeley 的内核从不自动重启函数 select(TCPv2 第 527 页)，但如果在安装信号处理程序时指明了标志 SA_RESTART，SVR4 就自动重启被中断的 select。这意味着，为了实现可移植性，我们在捕获信号时，必须准备好函数 select 返回 EINTR 错误。

虽然结构 timeval 为我们指定了一个微秒级的分辨率，但内核支持的分辨率却要粗糙得多。例如，很多 Unix 内核将超时值向上舍入成 10ms 的倍数。另外还有调度延迟现象，即定时器时间到后内核还需花一点时间调度相应进程的运行。

参数 timeout 前的限定词 const 表示它在返回时不会被 select 修改。例如，如果我们规定了一个 10 秒的时间限制，且在定时器时间到之前 select 返回，或是有一个或多个描述字准备好，或是返回 EINTR 错误，那么 timeout 所指 timeval 结构在函数返回时是不会更新成剩余的秒数的。如果我们想知道这个值，则在调用 select 函数之前必须得到系统时间，返回后再取得系统时间，两者相减即可。

> 现行的 Linux 系统修改了结构 timeval，因此，为了可移植性，需假设结构 timeval 在返回时无定义，并在每次调用 select 之前对它进行初始化。Posix.1g 规定使用限定词 const。

中间的三个参数 readset、writeset 和 exceptset 指定我们要让内核测试读、写和异常条件所需的描述字。现在只支持两个异常条件：

1. 套接口带外数据的到达，对此，我们在第 21 章中再作更为详细的描述。
2. 控制状态信息的存在，可从一个已置为分组方式的伪终端主端读到。本卷我们不讨论伪终端。

有一个设计上的问题，即如何为这三个参数的每一个指定一个或多个描述字值。函数 select 使用描述字集，它一般是一个整数数组，每个数中的每一位对应一个描述字。例如，用 32 位整数，则数组的第一个元素对应于描述字 0～31，数组中的第二个元素对应于描述字 32～63，以此类推。所有的实现细节都与应用程序无关，它们隐藏在数据类型 fd_set 和下面的四个宏中：

```
void FD_ZERO (fd_set * fdset);          /* clear all bits in fdset */
void FD_SET (int fd, fd_set * fdset);   /* turn on the bit for fd in fdset */
void FD_CLR (int fd, fd_set * fdset);   /* turn off the bit for fd in fdset */
int  FD_ISSET(int fd, fd_set * fdset);  /* is the bit for fd on in fdset ? */
```

我们分配一个 fd_set 数据类型的描述字集，并用这些宏设置、测试集合中的每一位，我们还可以用 C 语言中的赋值语句将其赋值成另外一个描述字集。

> 我们所讨论的每个描述字占用整数数组中一位的方法仅仅是函数 select 的可能实现之一。不过，将描述字集中的每个描述字称呼为位(bit)是很常见的，如“打开读集合中表示监听描述字的位”。
>
> 在 6.10 节中，我们将看到函数 poll 用了一个完全不同的表示方法：一个可变长的结构数组，每个结构代表一个描述字。

例如，为了定义一个 fd_set 类型的变量，并打开描述字 1、4 和 5 的相应位，我们写如下代码：

```
fd_set rset;

FD_ZERO (&rset);        /* initialize the set: all bits off */
FD_SET (1, &rset);      /* turn on bit for fd 1 */
FD_SET (4, &rset);      /* turn on bit for fd 4 */
FD_SET (5, &rset);      /* turn on bit for fd 5 */
```

对集合的初始化是很重要的。如果集合作为一个自动变量分配而未初始化，那将导致不可预测的后果。

如果我们对某个条件不感兴趣，函数 select 的三个中间参数 readset、writeset 和 exceptset 中相应参数就可设为空指针。实际上，如果三个指针均为空，我们就有了一个比 Unix 的函数 sleep 更为精确的定时器(sleep 睡眠以秒为最小单位)。函数 poll 提供相似功能。APUE 的图 C.9 和 C.10 给出了一个用 select 和 poll 实现的函数 sleep_us，它的睡眠单位为微秒。

参数 maxfdp1 指定被测试的描述字个数，它的值是要被测试的最大描述字加 1(因此，我们将此参数命名为 maxfdp1)，描述字 0、1、2……一直到 maxfdp1－1 均被测试。

头文件<sys/select.h>中定义的常值 FD_SETSIZE，是数据类型 fd_set 的描述字数量，其值通常是 1024，但很少有程序使用那么多的描述字。参数 maxfdp1 强迫我们计算出所

关心的最大描述字并将此值通知内核。例如，前面给出的打开描述字 1、4 和 5 的代码，其 maxfdp1 值就应是 6。是 6 而不是 5 的原因就在于：我们指定的是描述字的个数而非最大值，而描述字是从 0 开始的。

> 这个参数之所以存在，且因其存在而带来了计算其值的负担，完全是为了从效率考虑。每个 fd_set 都为许多描述字（一般为 1024）开有空间，但这要比普通进程所用的数量大得多。内核正是通过在进程与内核间不拷贝不需要的描述字部分和不测试常为 0 的相应位来获得效率的（TCPv2 的 16.13 节）。

函数 select 修改由指针 readset、writeset 和 exceptset 所指的描述字集。这三个参数均为值-结果参数。当我们调用函数时，指定我们所关心的描述字集，当返回时，结果指示哪些描述字已准备好。返回时，我们用宏 FD_ISSET 来测试结构 fd_set 中的描述字。描述字集中任何与没有准备好的描述字相对应的位返回时清成 0。为此，每次调用 select 时，我们都得将所有描述字集中关心的位都置为 1。

> 当使用 select 时，两个最常见的编程错误是：忘了对最大描述字加 1，忘了描述字集是值-结果参数。第二个错误导致调用 select 时，描述字集中我们认为是 1 的位却置为 0。作者在调试本书中的一个例子时，因忘了对 select 的第一个参数加 1 而浪费了 2 个小时。

此函数的返回值表示跨所有描述字集的已准备好的总位数。如果在任何描述字准备好之前定时器时间到，则返回 0。返回 −1 表示有错（这是可能发生的，譬如函数被一个捕获的信号所中断）。

> SVR4 的早期版本中 select 的实现有一个缺陷：如果多个集合中的同一位均为 1，譬如说，一个描述字已准备好读和写，它只计数一次。现在的版本改正了此 bug。

描述字在什么条件下准备好？

我们已讨论了等待描述字准备好 I/O（读或写）或是有一个异常条件待处理（带外数据）。尽管可读性和可写性对于普通文件这样的描述字很明显，我们必须对引起 select 返回套接口“准备好”的条件说得明确些（TCPv2 的图 16.52）。

1. 下列四个条件中的任何一个满足时，套接口准备好读：
   a. 套接口接收缓冲区中的数据字节数大于等于套接口接收缓冲区低潮限度的当前值。对这样的套接口的读操作将不阻塞并返回一个大于 0 的值（即准备好读入的数据量）。我们可以用套接口选项 SO_RCVLOWAT 来设置此低潮限度，对于 TCP 和 UDP 套接口，其值缺省为 1。
   b. 连接的读这一半关闭（也就是说接收了 FIN 的 TCP 连接）。对这样的套接口的读操作将不阻塞且返回 0（即文件结束符）。
   c. 套接口是一个监听套接口且已完成的连接数为非 0。尽管在 15.6 节中我们将介绍 accept 可能阻塞的时序条件，但正常情况下，这样的监听套接口上的 accept 不会阻塞。
   d. 有一个套接口错误待处理。对这样的套接口的读操作将不阻塞且返回一个错误

(−1),errno 则设置成明确的错误条件。这些待处理的错误(pending errors)也可通过指定套接口选项 SO_ERROR 调用 getsockopt 来取得并清除。

2. 下列三个条件中的任一个满足时,套接口准备好写:

   a. 套接口发送缓冲区中的可用空间字节数大于等于套接口发送缓冲区低潮限度的当前值,且或者(i)套接口已连接,或者(ii)套接口不要求连接(例如 UDP 套接口)。这意味着,如果我们将这样的套接口设置为非阻塞(第 15 章),写操作将不阻塞且返回一个正值(例如由传输层接收的字节数)。我们可以用套接口选项 SO_SNDLOWAT 来设置此低潮限度,对于 TCP 和 UDP 套接口,其缺省值一般为 2048。

   b. 连接的写这一半关闭。对这样的套接口的写操作将产生信号 SIGPIPE(5.12 节)。

   c. 有一个套接口错误待处理。对这样的套接口的写操作将不阻塞且返回一个错误(−1),errno 则设置成明确的错误条件。这些待处理的错误也可通过指定套接口选项 SO_ERROR 调用 getsockopt 来取得并清除。

3. 如果一个套接口存在带外数据或者仍处于带外标记,那它有异常条件待处理。(我们在第 21 章描述带外数据。)

我们对"可读性"和"可写性"的定义直接取自 TCPv2 第 530～531 页中内核的 soreadable 和 sowriteable 宏,与此类似,我们对套接口"异常条件"的定义取自同一页中的函数 soo_select。

注意一个套接口出错时,它由 select 标记为既可读又可写。

接收和发送低潮限度的目的是:在 select 返回可读或可写条件之前,应用进程可以对有多少数据可读或有多大空间可用于写进行控制。例如,如果我们知道除非至少有 64 字节的数据可用,否则我们的应用进程不能完成有效的工作,那么就可以将接收低潮限度设置为 64,以防小于 64 个字节的数据准备好读时,select 就唤醒我们。

只要 UDP 套接口的发送低潮限度小于发送缓冲区大小(缺省关系常常如此),由于不需要连接,这样的 UDP 套接口总是可写的。

图 6.7 对刚描述的就函数 select 来说导致套接口准备好的条件作了小结。

条件	可读吗?	可写吗?	异常吗?
有数据可读	•		
关闭连接的读一半	•		
给监听套接口准备好新连接	•		
有可用于写的空间		•	
关闭连接的写一半		•	
待处理错误	•	•	
TCP 带外数据			•

图 6.7 就 select 来说导致套接口准备好的条件小结

select 的最大描述字数

早些时候我们说过,大多数应用程序不会用到许多描述字,很少能找到一个会使用几百个描述字的应用程序。但使用这么多描述字的应用程序确实存在,而且它们常常利用 select

来复用描述字。当 select 刚开始设计时，操作系统常对每个进程可用的最大描述字数上限作出限制(4.2BSD 的限制为 31)，select 也就用相同的限制值。但是，当今的 Unix 版本对每个进程的描述字数根本不作限制(仅受限于内存总量和管理性限制)，因此，我们的问题是：这对 select 有什么影响？

许多实现有类似于下面的声明，它取自 4.4BSD 的头文件<sys/types.h>：

```
/*
 * select uses bitmasks of file descriptors in longs.  These macros
 * manipulate such bit fields (the filesystem macros use chars).
 * FD_SETSIZE may be defined by the user, but the default here should
 * be enough for most uses.
 */
#ifndef    FD_SETSIZE
#DEFINE    FD_SETSIZE        256
#endif
```

这使我们想到，在包括此头文件之前，可将 FD_SETSIZE 定义为更大些的值以增加 select 所用的描述字集大小，遗憾的是，这常常没有什么作用①。

> 为了弄清到底出了什么差错，请注意 TCPv2 的图 16.53，它在内核中声明了三个描述字集，且将内核的 FD_SETSIZE 定义作为上限使用。因此增大描述字集大小的唯一方法是增大 FD_SETSIZE 的值，然后重新编译内核。不重新编译内核而改变其值是不够的。

有些厂家正在将 select 的实现改变为允许进程将 FD_SETSIZE 定义为比缺省值大些的值。BSD/OS 已改变了内核实现以允许更大的描述字集，还定义了四个新的 FD_xxx 宏来动态分配并操纵这些较大集合。然而，从可移植性角度出发，使用大描述字集应小心。

6.4　str_cli 函数(修订版)

现在，我们重写 5.5 节中的函数 str_cli，这一次我们使用 select，这样服务器进程一终止客户就能马上得到通知。早期版本的问题就在于当套接口上发生了某些事件时，客户可能阻塞于 fgets 调用，而新版本则阻塞于 select 调用，或是等待标准输入，或是等待套接口可读。图 6.8 示出了调用 select 所处理的各种条件。

有三个条件通过套接口处理：

1. 如果对方 TCP 发送数据，套接口就变为可读且 read 返回大于 0 的值(即数据的字节数)。
2. 如果对方 TCP 发送一个 FIN(对方进程终止)，套接口就变为可读且 read 返回 0(文件结束)。
3. 如果对方 TCP 发送一个 RST(对方主机崩溃并重新启动)，套接口就变为可读且

① 译者注：常值 FD_SETSIZE 的声明一直是在头文件<sys/types.h>中(4.4BSD 和 4.4BSD-Lite2)，但更新的源自 BSD 的内核和源自 SVR4 的内核把它放在头文件<sys/select.h>中。值得注意的是，有些应用程序(典型的是需要复用大量描述字的事件驱动服务器程序，所需描述字量超过 1024 个)开始改用 poll 代替 select，这样可避免描述字有限的问题。还要注意的是，select 的典型实现在描述字数增大时可能存在扩展性问题。

read 返回−1,errno 则含有明确的错误码。

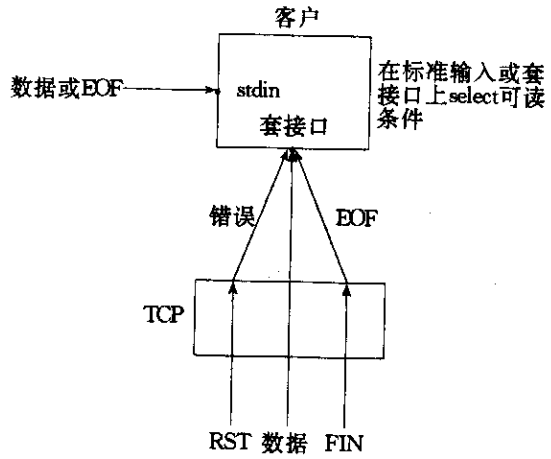


图 6.8 函数 str_cli 中由 select 处理的各种条件

图 6.9 给出了此新版本的源代码。

调用 select

第 8～13 行 我们仅需一个描述字集以检查可读性,此集合由 FD_ZERO 初始化,并用 FD_SET 打开两位:一位对应于标准 I/O 文件指针 fp,一位对应于套接口 sockfd。函数 fileno 把标准 I/O 文件指针转换为其对应的描述字。select(和 poll)仅工作在描述字上。

计算出两个描述字中的较大值后,调用 select。在调用中,写集合指针和异常集合指针都是空指针,且由于我们希望本调用阻塞,一直到某事件准备好,所以最后一个参数(时间限制)也是空指针。

处理可读套接口

第 14～18 行 如果在 select 返回时套接口是可读的,则回射行由 readline 来读,由 fputs 输出。

```
1 #include    "unp.h"
2 void
3 str_cli(FILE *fp,int sockfd)
4 {
5     int          maxfdp1;
6     fd_set       rset;
7     char         sendline[MAXLINE],recvline[MAXLINE];
8     FD_ZERO(&rset);
9     for( ; ; ) {
10        FD_SET(fileno(fp),&rset);
11        FD_SET(sockfd,&rset);
12        maxfdp1 = max(fileno(fp),sockfd) + 1;
13        Select(maxfdp1,&rset,NULL,NULL,NULL);
14        if (FD_ISSET(sockfd,&rset) ) { /* socket is readable */
15            if (Readline(sockfd,recvline,MAXLINE) == 0)
```

```
16                  err_quit("str_cli:server terminated prematurely");
17              Fputs(recvline,stdout);
18          }
19          if (FD_ISSET(fileno(fp),&rset)) {/* input is readable */}
20              if (Fgets(sendline,MAXLINE,fp) == NULL)
21                  return;          /* all done */
22          Writen(sockfd,sendline,strlen(sendline));
23          }
24      }
25 }
```

图 6.9　使用 select 的函数 str_cli 的实现(在图 6.13 中改进)[select/strcliselect01.c]

处理可读输入

第 19～23 行　如果标准输入可读，则由 fgets 读入一行，并用 writen 将其写到套接口。

请注意，此处使用了与 5.5 节相同的四个 I/O 函数：fgets、writen、readline 和 fputs，但程序流中使用函数的顺序发生了变化。它现在由 select 调用来驱动，而不是由 fgets 调用来驱动。与图 5.5 相比，图 6.9 中的代码仅增加了几行，就大大提高了客户程序的健壮性。

6.5　批 量 输 入

很不幸，我们的函数 str_cli 仍不正确。首先让我们回到其最初版本，即图 5.5。它以停-等方式工作，这对交互式使用是非常好的：发一行给服务器，然后便等待应答。这段总时间是 RTT(往返时间)加上服务器的处理时间(对于简单的回射服务器，处理时间几乎为 0)。如果知道了客户与服务器间的 RTT，我们便可以估计出回射固定数目的行数需花多长时间。

程序 Ping 是测量 RTT 的一个简单方法。如果我们在主机 solaris 上往主机 connix.com 执行 ping 命令，则对 30 次测量所取的平均 RTT 值是 175ms。TCPv1 第 89 页说明，这些 Ping 测量是针对长度为 84 字节的 IP 数据报进行的。如果我们取 solaris 2.5 上的文件 termcap 的前 2000 行，所得文件大小为 98349 字节，平均每行 49 字节。如果我们再加上 IP 头的大小(20 字节)和 TCP 头(20 字节)，则每行对应的分组大约有 89 字节，基本与 Ping 分组的大小相同。于是，我们可以估计出所有 2000 行文本的总时间大约为 350 秒(2000×0.175 秒)。如果运行第 5 章中的 TCP 回射客户程序，其真实时间大约为 354 秒，与我们的估计非常接近。

如果我们将客户与服务器间的网络当作全双工管道来考虑，从客户往服务器发请求，相反方向发应答，则图 6.10 所示为停-等方式。

客户在时刻 0 发出请求，我们假设 RTT 为 8 个时间单位。其应答时刻 4 发出并在时刻 7 接收到。我们还假设没有服务器处理时间且请求大小与回答大小相同。我们仅示出客户与服务器间的数据分组，对于同样穿越网络的 TCP 确认则不予理睬。

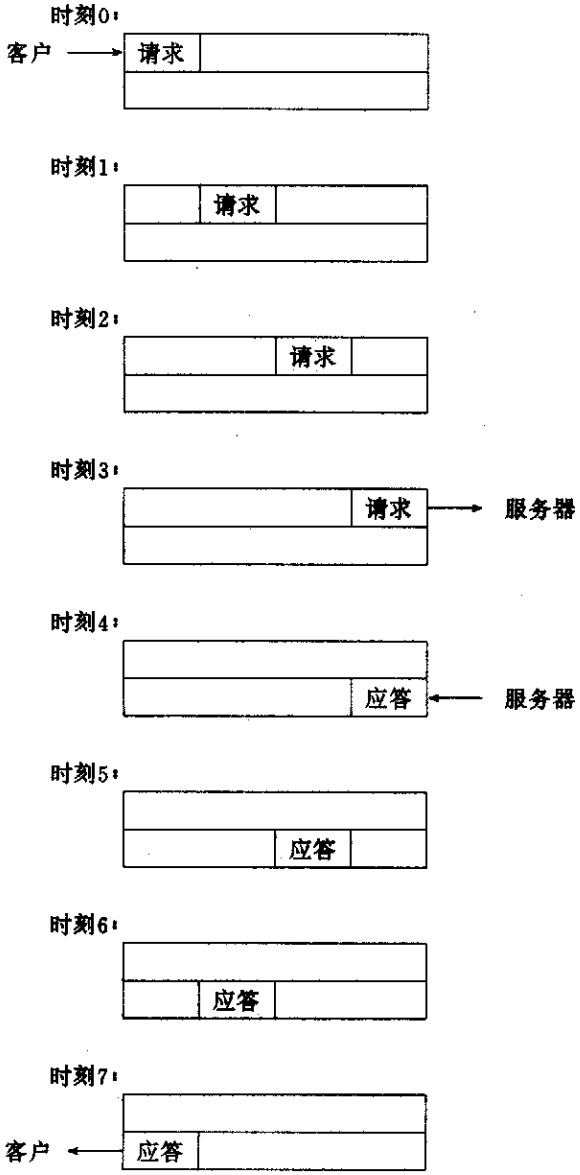


图 6.10 停-等方式的时间关系:交互式输入

但是,由于从发送一个分组到此分组到达管道的另一端有延迟,且由于管道是全双工的,在此例中,我们只用了⅛的管道容量。这种停-等方式对于交互式输入是非常合适的,但由于我们的客户从标准输入读且写到标准输出,在 Unix shell 下对输入和输出进行重定向又不麻烦,所以我们可以很容易地以批量方式运行客户。然而,当我们对输入输出重定向时结果输出文件总是小于输入文件(对于回射服务器,它们理应相等)。

为了弄清到底发生了什么事情,我们认识到在批量方式下,客户可以以网络能接受的最快速度发送请求,服务器也以相同的速度处理它们并发回应答。这在时刻 7 导致管道充满如图 6.11 所示。

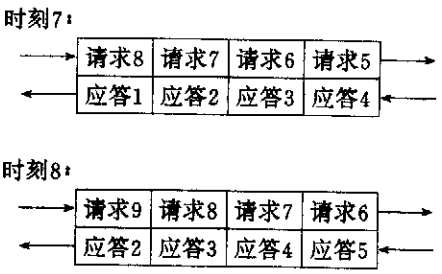


图 6.11　在客户与服务器间装填管道：批量方式

在这里，我们假设发出第一个请求后，立即发出下一个，紧接着再下一个。我们也假设客户可以以网络能接受它们的最快速度发送请求，且对它们的应答也能以网络能提供它们的最快速度处理。

> 这里，我们忽略了涉及到 TCP 批量数据的许多细节问题，如慢启动算法，它限制数据在一个新的或空闲的连接上发送数据和返回 ACK 的速率，这些都在 TCPv1 的第 20 章讨论。

为了弄清图 6.9 中函数 str_cli 存在的问题，假设输入文件只有 9 行，最后一行在时刻 8 发送，如图 6.11 所示。但我们在写完这个请求后，并不能关闭连接，因为管道中还有其他的请求和应答。问题的起因在于输入时对文件结束符的处理：str_cli 函数返回到 main 函数，main 函数随后终止。但在批量方式下，输入的文件结束符并不意味着我们已完成了从套接口的读入：可能仍有请求在去往服务器的路上，或是在去往客户的路上仍有应答。

我们需要一种方法来关闭 TCP 连接的一半。也就是说，我们想给服务器发一个 FIN，告诉它我们已完成了数据发送，但仍为读而开放套接口描述字。这由下一节描述的 shutdown 函数来完成。

6.6　shutdown 函数

终止网络连接的正常方法是调用 close，但 close 有两个限制可由函数 shutdown 来避免。

1. close 将描述字的访问计数减 1，仅在此计数为 0 时才关闭套接口，4.8 节我们已对此进行了讨论。用 shutdown 我们可以激发 TCP 的正常连接终止序列（图 2.5 中由 FIN 开始的四个分节），而不管访问计数。
2. close 终止了数据传送的两个方向：读和写。由于 TCP 连接是全双工的，有很多时候我们要通知另一端我们已完成了数据发送，即使那一端仍有许多数据要发送也是如此，这就是我们在前一节中所遇到的函数 str_cli 有批量输入时的情况。图 6.12 所示为该情况下的典型函数调用。

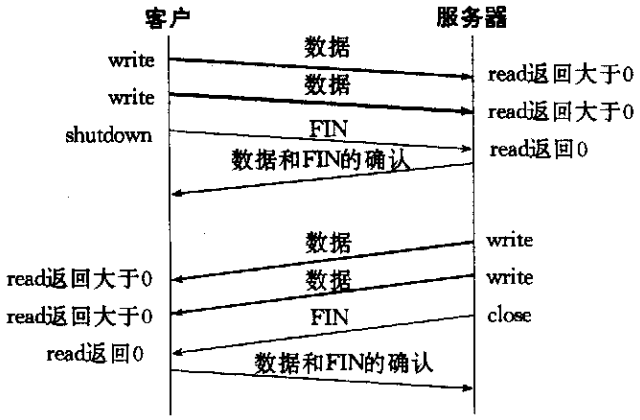


图 6.12 调用 shutdown 关闭一半 TCP 连接

```
#include <sys/socket.h>
int shutdown(int sockfd,int howto);
```

返回:0——成功,−1——出错

该函数的行为依赖于参数 howto 的值。

SHUT_RD 关闭连接的读这一半,不再接收套接口中的数据且现留在套接口接收缓冲区中的数据都作废。进程不能再对套接口执行任何读函数。调用此函数后,由 TCP 套接口接收的任何数据都被确认,但数据本身扔掉。

缺省时,写到路由套接口(第 17 章)上的所有内容都作为同一主机上所有路由套接口的输入而回馈。有些程序将第二个参数设为 SHUT_RD 来调用函数 shutdown 以防止回馈拷贝。另一个防止回馈拷贝的方法是关闭套接口选项 SO_USELOOPBACK。

SHUT_WR 关闭连接的写这一半,在 TCP 场合下,这称为半关闭(half-close)(TCPv1 的 18.5 节)。当前留在套接口发送缓冲区中的数据都被发送,后跟正常的 TCP 连接终止序列。正如我们前面所说,这个写这一半的关闭是不管套接口描述字的访问计数是否大于 0 的。进程不能再执行对套接口的任何写函数。

SHUT_RDWR 连接的读这一半和写这一半都关闭。这等效于调用函数 shutdown 两次:第一次调用时用 SHUT_RD,第二次调用时用 SHUT_WR。

图 7.10 对调用 shutdown 或 close 的进程的可能差别作了小结。close 的操作依赖于套接口选项 SO_LINGER 的值。

SHUT_xxx 这三个名字是随 Posix.1g 新近定义的。你碰到的 howto 参数的典型值可能是 0(关闭读这一半)、1(关闭写这一半)和 2(读这一半和写这一半都关闭)。

6.7　str_cli 函数(再修订版)

图 6.13 给出函数 str_cli 的改进(且正确)版本，它使用了 select 和 shutdown，前者在服务器关闭它这一端的连接时通知我们，后者使我们正确地处理批量输入。

第 5～8 行　stdineof 是一个初始化为 0 的新标志，只要此标志为 0，每次在主循环中我们总是 select 标准输入的可读性。

第 16～24 行　当我们在套接口上读到文件结束符时，如果我们已在标准输入上遇到文件结束符，这就是正常的终止，函数返回；但如果我们在标准输入上没有遇到文件结束符，那么服务器进程已过早终止。

第 25～33 行　当我们在标准输入上碰到文件结束符时，我们将新标志 stdineof 置为 1，并将第二个参数选为 SHUT_WR 来调用 shutdown 以发送 FIN。

如果用同样的 2000 行文件对图 6.13 中所示 str_cli 函数实现的 TCP 客户程序进行测量，现在所需时间约为 12.3 秒，比停-等方式大约快了 30 倍。

```
1  #include    "unp.h"
2  void
3  str_cli(FILE *fp, int sockfd)
4  {
5      int     maxfdp1, stdineof;
6      fd_set  rset;
7      char    sendline[MAXLINE], recvline[MAXLINE];
8      stdineof = 0;
9      FD_ZERO(&rset);
10     for ( ; ; ) {
11         if (stdineof == 0)
12             FD_SET(fileno(fp), &rset);
13         FD_SET(sockfd, &rset);
14         maxfdp1 = max(fileno(fp), sockfd) + 1;
15         Select(maxfdp1, &rset, NULL, NULL, NULL);
16         if (FD_ISSET(sockfd, &rset)) { /* socket is readable */
17             if (Readline(sockfd, recvline, MAXLINE) == 0) {
18                 if (stdineof == 1)
19                     return;         /* normal termination */
20                 else
21                     err_quit("str_cli: server terminated prematurely");
22             }
23             Fputs(recvline, stdout);
24         }
25         if (FD_ISSET(fileno(fp), &rset)) { /* input is readable */
26             if (Fgets(sendline, MAXLINE, fp) == NULL) {
27                 stdineof = 1;
28                 Shutdown(sockfd, SHUT_WR); /* send FIN */
29                 FD_CLR(fileno(fp), &rset);
30                 continue;
31             }
```

```
32              Writen(sockfd,sendline,strlen(sendline));
33          }
34      }
35  }
```

图 6.13 使用 select 正确处理文件结束符的函数 str_cli[select/strcliselect02.c]

我们对函数 str_cli 的讨论还没有结束，15.2 节中我们将开发一个使用非阻塞 I/O 模型的版本，23.3 节中我们将开发一个使用线程的版本。

6.8 TCP 回射服务器程序(修订版)

我们可以对 5.2 节和 5.3 节中介绍的 TCP 回射服务器重新进行讨论，并将服务器程序重写为使用 select 来处理任意数目的客户的单进程程序，而不是为每个客户派生一个子进程。在给出具体代码之前，让我们对用以跟踪客户的数据结构进行仔细的分析。图 6.14 给出了第一个客户建立连接前服务器的状态。

图 6.14 第一个客户建立连接前的服务器状态

服务器有单个监听描述字，我们用一个圆点来表示。

服务器只维护一个读描述字集，如图 6.15 所示。假设服务器是在前台启动的，则描述字 0、1 和 2 分别被设置为标准输入、标准输出和标准错误输出，因而，监听套接口的第一个可用的描述字是 3。我们也给出了一个名为 client 的整型数组，它含有每个客户的已连接套接口描述字，此数组中的所有元素都初始化为 −1。

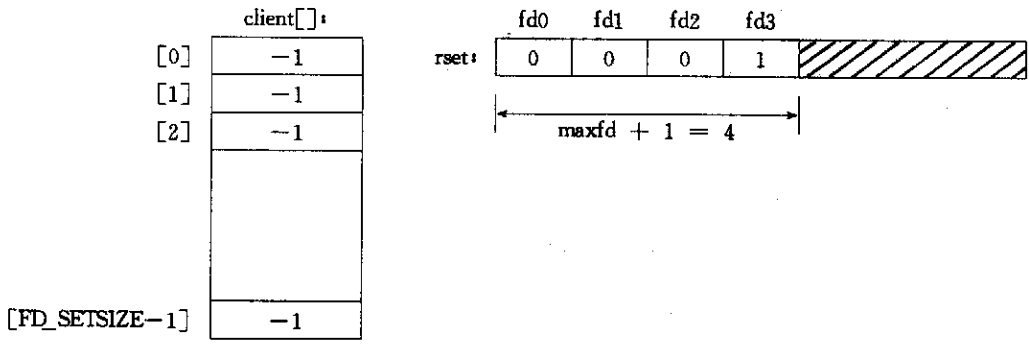


图 6.15 仅有监听套接口的 TCP 服务器数据结构

描述字集中唯一的非 0 条目是表示监听套接口的条目，因此 select 的第一个参数为 4。

当第一个客户与服务器建立连接时，监听描述字变为可读，于是我们的服务器调用 accept。在本例的假设下，由 accept 返回的新的已连接描述字将是 4。图 6.16 所示为从客户到

服务器的连接。

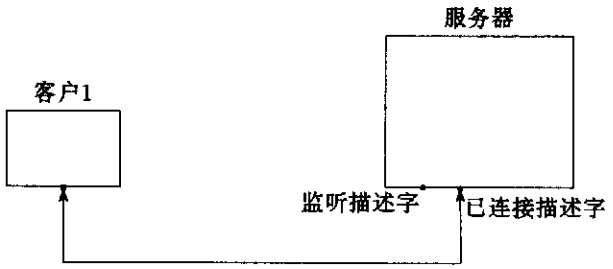


图 6.16　第一个客户建立连接后的 TCP 服务器

从现在起，我们的服务器必须在其数组 client 中记住新的已连接描述字，同时必须加到描述字集合中去，这样更新的数据结构示于图 6.17。

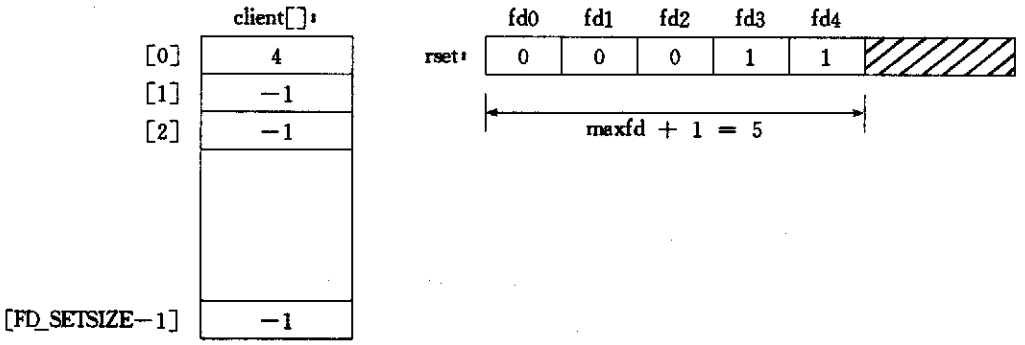


图 6.17　第一个客户连接建立后的数据结构

稍后，第二个客户与服务器建立连接，图 6.18 示出了这种情形。

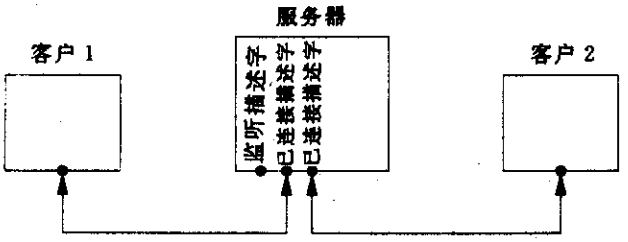


图 6.18　第二个客户建立连接后的 TCP 服务器

新的已连接描述字(我们假设是 5)必须被记住，数据结构如图 6.19 所示。

下面，我们假设第一个客户终止它的连接。客户 TCP 发送一个 FIN，这使得服务器中的描述字 4 变为可读。当服务器读此已连接套接口时，readline 返回 0。接着，我们关闭此套接口并相应地更新数据结构，数组元素 client[0]的值置为−1，描述字集中的描述字 4 被置为 0，如图 6.20 所示。注意，maxfd 的值没有改变。

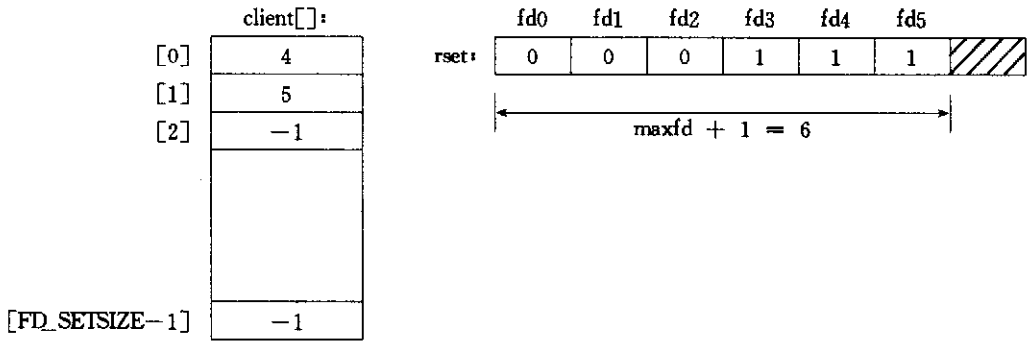


图 6.19 第二个客户建立连接后的数据结构

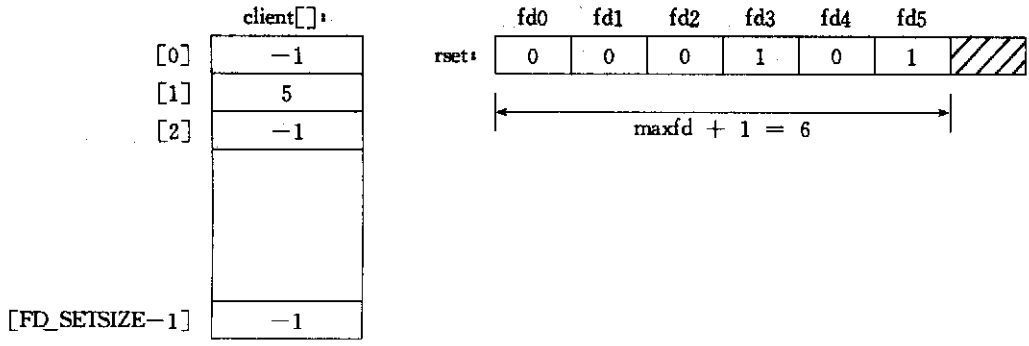


图 6.20 第一个客户终止连接后的数据结构

总之，当有客户到达时，我们在数组 client 中的第一个可用条目(即值为−1 的第一个条目)记录其已连接套接口的描述字。我们还必须把这个已连接描述字加到读描述字集合中。变量 maxi 是当前使用的数组 client 的最大下标，而变量 maxfd(加 1)是函数 select 第一个参数的当前值。对此服务器所能处理的最大客户数目的限制是以下两个值中的较小者：FD_SETSIZE 和内核允许此进程的最大描述字数(这我们在 6.3 节的结尾已讨论过)。

图 6.21 给出了此版本服务器程序的前半部分。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int       i,maxi,maxfd,listenfd,connfd,sockfd;
6     int       nready,client[FD_SETSIZE];
7     ssize_t   n;
8     fd_set    rset,allset;
9     char      line[MAXLINE];
10    socklen_t clilen;
11    struct sockaddr_in cliaddr,servaddr;
12    listenfd= Socket(AF_INET,SOCK_STREAM,0);
13    bzero(&servaddr,sizeof(servaddr));
14    servaddr.sin_family = AF_INET;
15    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
16    servaddr.sin_port = htons(SERV_PORT);
17    Bind(listenfd,(SA *) &servaddr,sizeof(servaddr));
18    Listen(listenfd,LISTENQ);
19    maxfd = listenfd;            /* initialize */
20    maxi = -1;                   /* index into client[] array */
21    for (i = 0; i < FD_SETSIZE; i++)
22        client[i] = -1;          /* -1 indicates available entry */
23    FD_ZERO(&allset);
24    FD_SET(listenfd,&allset);
```

图 6.21　使用单进程和 select 的 TCP 服务器程序：初始化[tcpcliserv/tcpservselect01.c]

创建监听套接口并给 select 初始化

第 12～24 行　创建监听套接口的步骤与前面所讨论的相同：socket、bind 和 listen。我们按照一开始 select 的唯一描述字是监听描述字的前提对数据结构进行初始化。

main 函数的后半部分示于图 6.22 中。

```
25    for( ; ; ) {
26        rset = allset;           /* structure assignment */
27        nready = Select(maxfd + 1,&rset,NULL,NULL,NULL);
28        if (FD_ISSET(listenfd,&rset))     /* new client connection */
29            clilen = sizeof(cliaddr);
30            connfd = Accept(listenfd,(SA *) &cliaddr,&clilen);
31            for(i = 0; i < FD_SETSIZE;i++)
32                if(client[i] < 0) {
33                  client[i] = connfd;          /* save descriptor */
34                  break;
35                }
36            if(i == FD_SETSIZE)
37              err_quit("too many clients");
38            FD_SET(connfd, &allset);     /* add new descriptor to set */
39            if(connfd > maxfd)
40              maxfd = connfd;     /* for select */
41            if(i > maxi)
42              maxi=i;     /* max index in client[] array */
43            if(--nready <= 0)
44              continue;     /* no more readable descriptors */
45        }
46        for (i = 0; i <= maxi; i++)  {   /* check all clients for data */
47            if ( (sockfd = client[i]) < 0)
48              continue;
49            if(FD_ISSET(sockfd,&rset)) {
50              if( (n = Readline(sockfd,line,MAXLINE)) == 0) {
51                        /* connection closed by client */
52                  Close(sockfd);
53                  FD_CLR(sockfd,&allset);
54                  client[i] = -1;
55              } else
56                  Writen(sockfd,line,n);
```

```
57                  if(--nready <= 0)
58                      break;      /* no more readable descriptors */
59              }
60          }
61      }
62  }
```

图 6.22 使用单进程和 select 的 TCP 服务器程序：循环[tcpcliserv/tcpservselect01.c]

阻塞于 select

第 26～27 行 select 等待某个事件发生：或是新客户连接的建立，或是数据、FIN 或 RST 的到达。

接受新连接

第 28～45 行 如果监听套接口变为可读，那么已建立了一个新的连接。我们调用 accept 并相应更新数据结构，使用数组 client 中的第一个未用条目来记录已连接描述字。准备好描述字的数目减 1，若此值为 0，就避免进入下面的 for 循环。这就使得我们可以用 select 的返回值来避免对未准备好的描述字进行检查。

检查现有连接

第 46～60 行 对于每个现有的客户连接，我们测试其描述字是否在 select 返回的描述字集合中。如果是就从客户读入一行并回射给客户。如果客户关闭连接，那么 readline 返回 0，我们就要相应地更新数据结构。

我们从不对 maxi 的值减 1，但每次客户关闭其连接时可以检查这种可能性。

此服务器程序要比图 5.2 和图 5.3 中所示的服务器程序复杂，但它避免了为每个客户创建一个新进程的所有开销，是 select 一个很好的例子。不过，在 15.6 节中我们将讨论此服务器的一个问题，它可以通过将监听套接口定义为非阻塞型，然后检查并忽略 accept 返回的一些错误来很容易地解决。

拒绝服务型攻击

很不幸，我们刚给出的服务器程序有一个问题。考虑一下，如果一个恶意客户连接到服务器上，发送一个字节的数据（而不是一行数据）后就睡眠。服务器将调用 readline，它从客户上读到单个字节的数据，然后就阻塞于下一个 read 调用以等待此客户的其他数据。接着，服务器就阻塞于（“挂起”可能是一个更好的说法）此单个客户，不能为任何其他客户提供服务（不论是新的客户连接还是现有客户的数据）。这种状况一直要持续到此恶意客户发出一个换行符或是终止为止。

这里的一个基本概念就是当一个服务器正在处理多个客户时，服务器决不能阻塞于只与单个客户相关的函数调用。如果这样的话，服务器将悬挂并拒绝为所有其他客户提供服务，这称为拒绝服务（denial of service）型攻击。它对服务器作了某些动作后，服务器就不能为其他合法客户提供服务了。可能的解决办法是：(a)使用非阻塞 I/O 模型（第 15 章）；(b)让每个客户由单独的控制线程提供服务（例如，创建子进程或线程来为每个客户提供服务）；(c)对 I/O 操作设置超时（13.2 节）。

6.9　pselect 函数

函数 pselect 是由 Posix.1g 发明的。

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>

int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
             const struct timespec *timeout, const sygset_t *sigmask);
                                    返回：准备好描述字的个数，0——超时，-1——出错
```

pselect 相对于正常的 select 有两个变化：

1. pselect 使用结构 timespec，这是 Posix.1b 实时标准的一个发明，而不使用结构 timeval。

```
struct timespec {
time_t tv_sec;      /* seconds */
long tv_nsec;       /* nanoseconds */
};
```

这两个结构的区别在第二个成员上：新结构的成员 tv_nsec 规定纳秒数，而老结构的成员 tv_usec 规定微秒数。

2. 函数 pselect 增加了第六个参数：指向信号掩码的指针。这允许程序禁止递交某些信号，测试由这些当前禁止的信号的信号处理程序所设置的全局变量，然后调用 pselect，告诉它临时重置信号掩码。

对于第二点，考虑下面的例子（在 APUE 第 389～309 页讨论）。这个程序的 SIGINT 信号处理程序仅设置全局变量 intr_flag 并返回。如果我们的进程阻塞于 select 调用，则从信号处理程序的返回会引起该函数返回错误并把 errno 设置成 EINTR。但当调用 select 时，代码看起来有以下形式：

```
if (intr_flag)
    handle_intr();      /* handle the signal */
if ( (nready = select( ... )) < 0) {
    if (errno == EINTR ) {
        if (intr_flag)
            handle_intr();
    }
    ...
}
```

问题就在于：在测试 intr_flag 和调用 select 之间如果信号发生，在 select 永远阻塞时它将丢失。有了 pselect 后，现在我们可以可靠地编写这个例子的代码如下：

```
sigset_t newmask, oldmask,zeromask;
```

```
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* block SIGINT */
if (intr_flag)
    handle_intr(); /* handle the signal */
if ( (nready = pselect( ... ,&zeromask)) < 0){
    if (errno == EINTR){
        if (intr_flag)
            handle_intr();
    }
    ...
}
```

在测试 intr_flag 变量之前，我们阻塞 SIGINT。当 pselect 被调用时，它用空集(即 zeromask)来代替进程的信号掩码，接着检查描述字，并可能进入睡眠。但当函数 pselect 返回时，进程的信号掩码又重置为调用 pselect 之前的值(即 SIGINT 阻塞)。

在 18.5 节，我们对 pselect 作更多的讨论，并给出了它的一个例子。在图 18.7 中，我们使用 pselect，然后在图 18.8 中，给出它的一个简单但不太正确的实现。

> 两个 select 函数还有另外一个小区别。结构 timeval 的第一个成员是有符号的长整数，而结构 timespec 的第一个成员是 time_t。前者的有符号长整数也应该是 time_t，但没做这种追溯性修改，以防破坏现有的代码。然而，新的函数 pselect 可以做此修改。

6.10 poll 函数

函数 poll 起源于 SVR3，开始时局限于流设备(第 32 章)，SVR4 取消了此限制，允许 poll 工作在任何描述字上。poll 提供了与 select 相似的功能，但当涉及到流设备时，它还提供附加信息。

```
#include <poll.h>
int poll(struct pollfd *fdarray, unsigned long nfds, int timeout);
                                返回：准备好描述字的个数，0——超时，-1——出错
```

第一个参数是指向一个结构数组第一个元素的指针，每个数组元素都是一个 pollfd 结构，它规定了为测试一给定描述字 fd 的一些条件。

```
struct pollfd {
int fd;          /* descriptor to check */
short events;    /* events of interest on fd */
short revents;   /* events that occurred on fd */
};
```

要测试的条件由成员 events 规定，函数在相应的 revents 成员中返回描述字的状态。(每个描述字有两个变量，一个为调用值，另一个为结果，以此避免使用值-结果参数。回想一下，

函数 select 的中间三个参数都是值-结果参数。)这两个成员中的每一个都由指定某个条件的一位或多位组成。图 6.23 列出了用于指定标志 events 并测试标志 revents 的一些常值。

常　量	能作为 events 的输入吗?	能作为 revents 的结果吗?	解释
POLLIN	•	•	普通或优先级带数据可读
POLLRDNORM	•	•	普通数据可读
POLLRDBAND	•	•	优先级带数据可读
POLLPRI	•	•	高优先级数据可读
POLLOUT	•	•	普通数据可写
POLLWRNORM	•	•	普通数据可写
POLLWRBAND	•	•	优先级带数据可写
POLLERR		•	发生错误
POLLHUP		•	发生挂起
POLLNVAL		•	描述字不是一个打开的文件

图 6.23　函数 poll 的输入 events 和返回值 revents

我们将此图分为三个部分：第一部分为处理输入的四个常值，第二部分为处理输出的三个常值，第三部分为处理错误的三个常值。注意，第三部分的三个常值在 events 中是不能设置的，但是当相应条件存在时就在 revents 中返回。

poll 识别三个类别的数据：普通(normal)、优先级带(priority band)和高优先级(high priority)，这些术语均出自基于流的实现(图 33.5)。

> POLLIN 可被定义为 POLLRDNORM 和 POLLRDBAND 的逻辑或常值。POLLIN 存在于 SVR3 的实现，它要早于 SVR4 中的优先级带，所以此常值保持了向后兼容性。类似地，POLLOUT 等效于 POLLWRNORM，前者早于后者。

考虑到 TCP 和 UDP 套接口，下面的条件引起 poll 返回特定的 revent。不幸的是，Posix.1g在其 poll 的定义中留下了许多漏洞(也就是说，返回相同的条件有多种方法可选择)。

- 所有正规 TCP 数据和 UDP 数据都被认为是普通数据。
- TCP 的带外数据(第 21 章)被认为是优先级带数据。
- 当 TCP 连接的读这一半关闭时(例如，接收了一个 FIN)，这也认为是普通数据，且后续的读操作将返回 0。
- TCP 连接存在错误既可认为是普通数据，也可认为是错误(POLLERR)。无论哪种情况，后续的读操作将返回－1，并将 errno 置为适当的值，这就处理了诸如接收到 RST 或超时等条件。
- 在监听套接口上新连接的可用性既可认为是普通数据，也可认为是优先级数据，大多数实现都将其作为普通数据考虑。

结构数组中元素的个数是由参数 nfds 来规定的。

历史上,这个参数曾被认为是无符号长整数,这似乎太大了,无符号整数可能更合适些。Unix 98 为此参数定义了一个新的数据类型:nfds_t。

参数 timeout 指定函数返回前等待多长时间。它是一个指定应等待的毫秒数的正值。图 6.24 给出了 timeout 的可能值。

timeout 的值	解释
INFTIM	永远等待
0	立即返回,不阻塞
>0	等待指定数目的毫秒数

图 6.24 poll 的 timeout 参数值

常值 INFTIM 定义为一个负值。如果系统不能提供毫秒级精度的定时器,此值向上舍入到最接近的支持值。

Posix.1g 要求在头文件<poll.h>中定义 INFTIM,但许多系统仍在头文件<sys/stropts.h>中定义。

正如 select,为 poll 所规定的任何超时值都受限于实现的时钟分辨率(常常是 10ms)。

当出错时,函数 poll 的返回值为-1;若定时器时间到,还没有描述字准备好,则返回 0;否则返回准备好描述字的个数,即成员 revents 非 0 的描述字个数。

如果我们不关心某个特定描述字,可将其 pollfd 结构的 fd 成员置为一个负值。这样就可忽略成员 events,且返回时将成员 revents 的值置为 0。

回忆一下 6.3 节的结尾,我们对 FD_SETSIZE、每个描述字集合中描述字最大数目及每个进程中描述字最大数目作了讨论。有了 poll 后,我们就没有那个问题了,因为分配一个结构 pollfd 的数组并通知内核数组中元素的数目是调用者的责任。这里没有类似于内核知道的 fd_set 之类固定大小的数据类型。

Posix.1g 对 select 和 poll 都要求,但从可移植性角度出发,当今支持 select 的系统比支持 poll 的系统要多。而且,Posix.1g 还定义了 pselect,它是函数 select 的增强版本,能处理信号阻塞并提供更高的时间分辨率,但未定义与函数 poll 相似的任何东西。

6.11 TCP 回射服务器程序(再修订版)

现在,我们用 poll 而不是 select 来重写 6.8 节中的 TCP 回射服务器程序。在以前使用 select 的版本中,我们必须分配一个 client 数组以及一个名为 rset 的描述字集(图 6.15)。使用 poll 时我们必须分配一个 pollfd 结构的数组来维护客户信息,而不是分配另一个数组。我们以与图 6.15 中处理数组 client 相同的方法处理此数组的 fd 成员:值-1 表示条目未用,否则即为描述字值。回忆一下前一节,传递给 poll 的 pollfd 结构数组中的任何 fd 成员为负值的条目都是被忽略的。

图 6.25 给出了我们的服务器程序的前半部分。

```
1 #include     "unp.h"
2 #include     <limits.h>          /* for OPEN_MAX */
3 int
4 main(int argc,char **argv)
5 {
6     int      i,maxi,listenfd,connfd,sockfd;
7     int      nready;
8     ssize_t  n;
9     char     line[MAXLINE];
10    socklen_t clilen;
11    struct pollfd client[OPEN_MAX];
12    struct sockaddr_in cliaddr,servaddr;
13    listenfd = Socket(AF_INET,SOCK_STREAM,0);
14    bzero(&servaddr,sizeof(servaddr));
15    servaddr.sin_family = AF_INET;
16    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
17    servaddr.sin_port = htons(SERV_PORT);
18    Bind(listenfd,(SA *) &servaddr,sizeof(servaddr));
19    Listen(listenfd,LISTENQ);
20    client[0].fd = listenfd;
21    client[0].events = POLLRDNORM;
22    for (i = 1;i < OPEN_MAX; i++)
23        client[i].fd = -1;          /* -1 indicates available entry */
24    maxi = 0;                       /* max index into client[] array */
```

图 6.25　使用函数 poll 的 TCP 服务器程序的前半部分[tcpcliserv/tcpservpoll01.c]

分配结构 pollfd 的数组

第 11 行　我们声明结构 pollfd 的数组中的 OPEN_MAX 个元素。确定进程任一时刻能打开的最大描述字数目是很困难的，图 12.4 中我们将再次遇到这个问题。一个方法是以参数_SC_OPEN_MAX(如 APUE 的第 42～44 页所述)调用 Posix.1 的函数 sysconf，然后动态分配一个合适大小的数组。但函数 sysconf 的一个可能返回是“indeterminate(不确定)”，这意味着我们仍然不得不猜一个值。这里我们就用 Posix.1 的常值 OPEN_MAX。

初始化

第 20～22 行　我们将数组 client 的第一个条目用于监听套接口，并将其余的描述字条目置为－1。我们还为此描述字设置了事件 POLLRDNORM，当新的连接准备好接受时由 poll 来通知。变量 maxi 含有当前使用的数组 client 的最大下标值。

main 函数的后半部分示于图 6.26 中。

调用 poll，检查新连接

第 26～42 行　我们调用 poll，或等待新连接，或等待现有连接上的数据。当一个新连接被接受时，我们在数组 client 中查找第一个为负的描述字来找到第一个可用条目。注意，由于 client[0]用于监听套接口了，我们从下标 1 开始搜索。当找到一个可用条目时，保存描述字并设置事件 POLLRDNORM。

在现有连接上检查数据

第43行～63行 我们检查的两个返回事件是POLLRDNORM和POLLERR。其中的第二项我们未在成员event中设置，因为在条件成立时它总是返回的。我们检查POLLERR的原因是：有些实现在连接上接收到RST时返回事件POLLERR，而其他实现只返回POLLRDNORM。无论哪种情况下我们调用readline，在出错时它都返回错误。当一个现有的连接由客户终止时，我们就设置其fd成员为－1。

```
25    for( ; ; ) {
26        nready = Poll(client,maxi + 1,INFTIM);
27        if(client[0].revents & POLLRDNORM) {      /* new client connection */
28            clilen = sizeof(cliaddr);
29            connfd = Accept(listenfd,(SA *) &cliaddr,&clilen);
30            for (i = 1; i < OPEN_MAX; i++)
31                if(client[i].fd < 0) {
32                    client[i].fd = connfd;          /* save descriptor */
33                    break;
34                }
35            if (i == OPEN_MAX)
36                err_quit("too many clients");
37            client[i].events = POLLRDNORM;
38            if (i > maxi)
39                maxi = i;          /* max index in client[] array */
40            if (--nready <= 0)
41                continue;          /* no more readable descriptors */
42        }
43        for(i = 1;i <= maxi; i++) {      /* check all clients for data */
44            if ( (sockfd = client[i].fd) < 0)
45                continue;
46            if(client[i].revents & (POLLRDNORM | POLLERR)) {
47                if( (n = readline(sockfd,line,MAXLINE)) < 0) {
48                    if(errno == ECONNRESET) {
49                            /* connection reset by client */
50                        Close(sockfd);
51                        client[i].fd = -1;
52                    } else
53                        err_sys("readline error");
54                } else if (n == 0) {
55                            /* connection closed by client */
56                    Close(sockfd);
57                    client[i].fd = -1;
58                } else
59                    Writen(sockfd,line,n);
60                if(--nready <= 0)
61                  break;          /* no more readable descriptors */
62            }
63        }
64    }
65 }
```

图6.26 使用poll的TCP服务器程序的后半部分[tcpcliserv/tcpservpoll01.c]

6.12　小　结

Unix 提供了五种不同的 I/O 模型：

- 阻塞 I/O 模型
- 非阻塞 I/O 模型
- I/O 复用模型
- 信号驱动 I/O 模型
- 异步 I/O 模型

缺省为阻塞 I/O 模型，这也是最常用的 I/O 模型。在后面章节中，我们将讨论非阻塞 I/O 模型和信号驱动 I/O 模型，而本章已讨论了 I/O 复用模型。真正的异步 I/O 模型是 Posix.1 定义的，但它很少有实现。

I/O 复用模型最常用的函数是 select。我们通知此函数所关心的描述字（读、写和异常条件）、最长等待时间、最大描述字号（加 1）。大多数对 select 的调用指定可读条件，并注意到，当处理套接口时，唯一的异常条件是带外数据的到达（第 21 章）。由于 select 提供了函数阻塞多长时间的限制，我们将此特征用于图 13.3 以设置输入操作的时间限制。

将我们的回射客户程序用到使用 select 的批处理模式中，我们发现，即使已遇到了用户输入的结尾，数据也可能仍在去往或来自服务器的管道中。为了处理这种情形，要求使用函数 shutdown，这使得我们可以利用 TCP 的半关闭特性。

Posix.1g 定义了新的函数 pselect，它将时间精度从微秒级增加到纳秒级，并采用了一个指向信号集的指针作为新参数，这样避免了信号被捕获时的竞争条件，我们在 18.5 节中再作进一步讨论。

系统 V 中的函数 poll 提供类似于 select 的功能，且为流设备提供附加信息。Posix.1g对 select 和 poll 都要求，但前者使用更频繁。

6.13　习　题

6.1　我们说，一个描述字集可以用 C 语言中的赋值语句赋给另一描述字集。如果描述字集是一个整型数组，怎么办？（提示：研究一下你系统中的头文件<sys/select.h>或<sys/types.h>。）

6.2　在讨论 6.3 节中 select 返回“可写”条件时，为什么必须要求套接口为非阻塞型以使写操作返回一个正值？

6.3　在图 6.9 中，在第 19 行中的关键词 if 前加上 else，将会怎样？

6.4　在图 6.21 的例子中，加上一段代码以允许服务器可以使用当前内核所允许的尽量多的描述字。（提示：研究一下函数 setrlimit。）

6.5　当 shutdown 的第二个参数为 SHUT_RD 时，让我们看看将发生什么。以图 5.4 中的 TCP 客户程序为基础并作下列改变：将端口号从 SERV_PORT 改为 19，即 chargen 服务器（图 2.13）；以调用 pause 来代替调用 str_cli。启动此程序并指定运

行服务器 chargen 的本地主机 IP 地址。以诸如 tcpdump(C.5 节)这类的工具来观察分组,看到什么?

6.6 为什么应用程序会以参数 SHUT_RDWR 来调用 shutdown,而不是仅仅调用 close?

6.7 图 6.22 中,当客户发送一个 RST 来终止连接时,将会怎样?

6.8 重写图 6.25 中的代码,调用 sysconf 来确定描述字的最大数目,并相应分配 client 数组。

第 7 章　套接口选项

7.1　概　述

有很多方法来获取和设置影响套接口的选项：

- 函数 getsockopt 和 setsockopt
- 函数 fcntl
- 函数 ioctl

本章从介绍函数 getsockopt 和 setsockopt 开始，接着给出一个输出所有选项缺省值的例子，然后详细介绍所有套接口选项。我们按以下分类进行详细介绍：基本、IPv4、IPv6 和 TCP。在第一次阅读本章时，这些细节可以跳过，当需要时再回来看个别章节。个别选项在后续章节中还有更为详细的讨论，如 IPv4 和 IPv6 多播选项在 19.5 节我们讨论多播时还会讨论到。

我们也介绍了函数 fcntl，因为它是设置套接口为非阻塞 I/O 型或信号驱动 I/O 型以及设置套接口的属主的 Posix 的方法。我们将函数 ioctl 留到第 16 章讨论。

7.2　getsockopt 和 setsockopt 函数

这两个函数仅用于套接口。

```
#include <sys/socket.h>
int getsockopt(int sockfd,int level,int optname, void *optval, socklen_t *optlen);
int setsockopt(int sockfd,int level,int optname, const void *optval,socklen_t *optlen);
                                                    返回：0——OK，-1——出错
```

sockfd 必须指向一个打开的套接口描述字，Level（级别）指定系统中解释选项的代码：普通套接口代码或特定于协议的代码（例如：IPv4、IPv6 或 TCP）。

optval 是一个指向变量的指针，通过它，或由 setsockopt 取得选项的新值，或由 getsockopt 存储选项的当前值。此变量的大小由最后一个参数指定，它对于 setsockopt 是一个值，对于 getsockopt 是一个值-结果参数。

图 7.1 总结了可由 getsockopt 获取或由 setsockopt 设置的一些选项。“数据类型”列给出了指针 optval 必须指向的每个选项的数据类型。我们用后跟一对花括号的记法来表示一个结构，如 linger{}表示结构 linger。

有两种基本类型的套接口选项:打开或关闭某个特性的二进制选项(标志),取得并返回我们可以设置或检查的特定值的选项(值)。标有"标志"的列指明选项是否为标志选项。当给这些标志选项调用函数 getsockopt 时,optval 是一个整数。optval 中返回的值是 0 表示选项关闭,非 0 表示选项打开。类似地,函数 setsockopt 要求一非 0 的 optval 来打开选项,要求用 0 来关闭选项。如果"标志"列不含有"·",则选项用来在用户进程与系统间传递指定数据类型的值。

本章后续各节将给出影响套接口的选项的额外细节。

level(级别)	optname(选项名)	get	set	说 明	标志	数据类型
SOL-SOCKET	SO-BROADCAST	·	·	允许发送广播数据报	·	int
	SO-DEBUG	·	·	使能调试跟踪	·	int
	SO-DONTROUTE	·	·	旁路路由表查询	·	int
	SO-ERROR	·		获取待处理错误并消除		int
	SO-KEEPALIVE	·	·	周期地测试连接是否仍存活	·	int
	SO-LINGER	·	·	若有数据待发送则延迟关闭		linger{}
	SO-OOBINLINE	·	·	让接收到的带外数据继续在线存放	·	int
	SO-RCVBUF	·	·	接收缓冲区大小		int
	SO-SNDBUF	·	·	发送缓冲区大小		int
	SO-RCVLOWAT	·	·	接收缓冲区低潮限度		int
	SO-SNDLOWAT	·	·	发送缓冲区低潮限度		int
	SO-RCVTIMEO	·	·	接收超时		timeval{}
	SO-SNDTIMEO	·	·	发送超时		timeval{}
	SO-REUSEADDR	·	·	允许重用本地地址	·	int
	SO-REUSEPORT	·	·	允许重用本地地址	·	int
	SO-TYPE	·		取得套接口类型		int
	SO-USELOOPBACK	·	·	路由套接口取得所发送数据的拷贝	·	int
IPPROTO-IP	IP-HDRINCL	·	·	IP 头部包括数据	·	int
	IP-OPTIONS	·	·	IP 头部选项		(见正文)
	IP-RECVDSTADDR	·	·	返回目的 IP 地址	·	int
	IP-RECVIF	·	·	返回接收到的接口索引	·	int
	IP-TOS	·	·	服务类型和优先权		int
	IP-TTL	·	·	存活时间		int
	IP-MULTICAST-IF	·	·	指定外出接口		in-addr{}
	IP-MULTICAST-TTL	·	·	指定外出 TTL		u-char
	IP-MULTICAST-LOOP	·	·	指定是否回馈		u-char
	IP-ADD-MEMBERSHIP		·	加入多播组		ip-mreq{}
	IP- DROP- MEMBERSHIP		·	离开多播组		ip-mreq{}
IPPROTO- ICMPV6	ICMP6-FILTER	·	·	指定传递的 ICMPv6 消息类型		icmp6- filter{}
IPPROTO-IPV6	IPV6-ADDRFORM	·	·	改变套接口的地址结构		int
	IPV6-CHECKSUM	·	·	原始套接口的校验和字段偏移		int
	IPV6-DSTOPTS	·	·	接收目标选项	·	int
	IPV6-HOPLIMIT	·	·	接收单播跳限	·	int
	IPV6-HOPOPTS	·	·	接收步跳选项	·	int
	IPV6-NEXTHOP	·	·	指定下一跳地址	·	sockaddr{}
	IPV6-PKTINFO	·	·	接收分组信息	·	int
	IPV6-PKTOPTIONS	·	·	指定分组选项		(见正文)
	IPV6-RTHDR	·	·	接收源路径	·	int
	IPV6-UNICAST-HOPS	·	·	缺省单播跳限		int

（续）

级　别	optname	get	set	解释	标志	数据类型
	IPV6_MULTICAST_IF	•	•	指定外出接口		in6_addr{}
	IPV6_MULTICAST_HOPS	•	•	指定外出跳限		u_int
	IPV6_MULTICAST_LOOP	•	•	指定是否回馈	•	u_int
	IPV6_ADD_MEMBERSHIP		•	加入多播组		ipv6_mreq{}
	IPV6_DROP_MEMBERSHIP		•	离开多播组		ipv6_mreq{}
IPPROTO_TCP	TCP_KEEPALIVE	•	•	控测对方是否存活前连接闲置秒数		int
	TCP_MAXRT	•	•	TCP 最大重传时间		int
	TCP_MAXSEG	•	•	TCP 最大分节大小		int
	TCP_NODELAY	•	•	禁止 Nagle 算法	•	int
	TCP_STDURG	•	•	紧急指针的解释	•	int

图 7.1　getsockopt 和 setsockopt 访问的套接口选项汇总

7.3　检查选项是否受支持并获取缺省值

现在，我们写一个程序来检查是否支持图 7.1 中定义的大多数选项，若支持，则输出其缺省值。图 7.2 包含了我们这个程序的所有声明。

```
1 #include      "unp.h"
2 #include      <netinet/tcp.h>       /* for TCP_xxx defines */
3 union val {
4   int               i_val;
5   long              l_val;
6   char              c_val[10];
7   struct linger     linger_val;
8   struct timeval    timeval_val;
9 } val;
10 static char *sock_str_flag(union val *, int);
11 static char *sock_str_int(union val *, int);
12 static char *sock_str_linger(union val *, int);
13 static char *sock_str_timeval(union val *, int);
14 struct sock_opts {
15     char        *opt_str;
16     int         opt_level;
17     int         opt_name;
18     char     *(*opt_val_str)(union val *, int);
19 } sock_opts[] = {
20     "SO_BROADCAST",    SOL_SOCKET,   SO_BROADCAST,   sock_str_flag,
21     "SO_DEBUG",        SOL_SOCKET,   SO_DEBUG,       sock_str_flag,
22     "SO_DONTROUTE",    SOL_SOCKET,   SO_DONTROUTE,   sock_str_flag,
23     "SO_ERROR",        SOL_SOCKET,   SO_ERROR,       sock_str_int,
24     "SO_KEEPALIVE",    SOL_SOCKET,   SO_KEEPALIVE,   sock_str_flag,
25     "SO_LINGER",       SOL_SOCKET,   SO_LINGER,      sock_str_linger,
```

```
26    "SO_OOBINLINE",    SOL_SOCKET,  SO_OOBINLINE,    sock_str_flag,
27    "SO_RCVBUF",       SOL_SOCKET,  SO_RCVBUF,       sock_str_int,
28    "SO_SNDBUF",       SOL_SOCKET,  SO_SNDBUF,       sock_str_int,
29    "SO_RCVLOWAT",     SOL_SOCKET,  SO_RCVLOWAT,     sock_str_int,
30    "SO_SNDLOWAT",     SOL_SOCKET,  SO_SNDLOWAT,     sock_str_int,
31    "SO_RCVTIMEO",     SOL_SOCKET,  SO_RCVTIMEO,     sock_str_timeval,
32    "SO_SNDTIMEO",     SOL_SOCKET,  SO_SNDTIMEO,     sock_str_timeval,
33    "SO_REUSEADDR",    SOL_SOCKET,  SO_REUSEADDR,    sock_str_flag,
34 #ifdef    SO_REUSEPORT
35    "SO_REUSEPORT",    SOL_SOCKET,  SO_REUSEPORT,    sock_str_flag,
36 #else
37    "SO_REUSEPORT",    0,           0,               NULL
38 #endif
39    "SO_TYPE",         SOL_SOCKET,  SO_TYPE,         sock_str_int,
40    "SO_USELOOPBACK",  SOL_SOCKET,  SO_USELOOPBACK,  sock_str_flag,
41    "IP_TOS",          IPPROTO_IP,  IP_TOS,          sock_str_int,
42    "IP_TTL",          IPPROTO_IP,  IP_TTL,          sock_str_int,
43    "TCP_MAXSEG",      IPPROTO_TCP, TCP_MAXSEG,,     sock_str_int,
44    "TCP_NODELAY",     IPPROTO_TCP, TCP_NODELAY,,    sock_str_flag,
45    NULL,              0,           0,               NULL
46 };
```

图 7.2 检查套接口选项的程序的声明[sockopt/checkopts.c]

声明可能值的联合

第3～9行 对于getsockopt的每个可能的返回值,我们的联合中都有一个成员。

定义函数原型

第10～13行 我们为四个函数定义函数原型,这四个函数被调用来输出给定套接口选项的值。

定义结构并初始化数组

第14～16行 我们的结构sock_opts包含了给每个套接口选项调用getsockopt并输出其当前值所需要的所有信息。它的最后一个成员:opt_val_str,是指向用于输出选项值的四个函数中某一个的指针。我们分配并初始化这个结构的一个数组,它的每个元素代表一个套接口选项。

并非所有实现都支持所有的套接口选项。确定某给定选项是否被支持的方法是用语句#ifdef或if defined,如图中SO_REUSEPORT选项所示。要求完整的话数组中每个元素都应类似于SO_REUSEPORT所示编写,但我们省略了这些,因为一大堆# ifdef语句仅仅加长了代码,对我们的讨论没有什么用处。

图7.3给出了我们的main函数。

```
47 int
48 main(int argc,char * * argv)
49 {
50    int     fd,len;
51    struct sock_opts *ptr;
52    fd = Socket(AF_INET,SOCK_STREAM,0);
```

```
53    for(ptr = sock_opts;ptr->opt_str ! = NULL; ptr++) {
54        printf("%s:",ptr->opt_str);
55        if (ptr->opt_val_str == NULL)
56            printf("(undefined)\n");
57        else {
58            len = sizeof(val);
59            if(getsockopt(fd,ptr->opt_level,ptr->opt_name,
60                    &val,&len) == -1) {
61                err_ret("getsockopt error");
62            } else {
63                printf("default = %s\n",(*ptr->opt_val_str)(&val,len));
64            }
65        }
66    }
67    exit(0);
68 }
```

图 7.3　检查所有套接口选项的 main 函数[sockopt/checkopts.c]

创建 TCP 套接口，浏览所有选项

第 52～56 行　我们创建一个 TCP 套接口，然后浏览数组中所有元素。如果指针 opt_val_str 为空，则该实现没有定义此选项(我们的例子中 SO_REUSEPORT 选项是可能的)。

调用 getsockopt

第 57～61 行　我们调用 getsockopt，但在返回错误时并不终止。许多实现定义了一些套接口选项名字，即使它们不支持这些选项也是如此。不支持的选项应引发一个 ENOPROTOOPT 错误。

输出选项的缺省值

第 62～63 行　如果 getsockopt 返回成功，我们调用相应的函数将选项值转换为一个字符串，然后输出这个字符串。

在图 7.2 中，我们给出了四个函数原型，每个类型的选项值一个。图 7.4 给出了这四个函数中的一个，即 sock_str_flag，它输出标志选项的值，其他三个函数与此类似。

```
69 static char strres[128];
70 static char *
71 sock_str_flag(union val *ptr,int len)
72 {
73    if(len ! = sizeof(int))
74        snprintf(strres,sizeof(strres),"size (%d) not sizeof(int)",len);
75    else
76        snprintf(strres,sizeof(strres),
77                "%s",(ptr->i_val == 0) ? "off" : "on");
78    return(strres);
79 }
```

图 7.4　函数 sock_str_flag：将标志选项转换为字符串[sockopt/checkopts.c]

第73～78行 回忆一下，getsockopt的最后一个参数是值-结果参数。我们所做的第一项检查就是getsockopt返回值的大小是否为期望的大小。本函数返回的字符串或为“on”，或为“off”，这得取决于标志选项的值是0还是非0。

在AIX 4.2下运行此程序有以下输入：

```
aix % checkopts
SO_BROADCAST: default = off
SO_DEBUG: default = off
SO_DONTROUTE: default = off
SO_ERROR: default = 0
SO_KEEPALIVE: default = off
SO_LINGER: default = 1 onoff = 0,1_linger = 0
SO_OOBINLINE: default = off
SO_RCVBUF: default = 16384
SO_SNDBUF: default = 16384
SO_RCVLOWAT: default = 1
SO_SNDLOWAT: default = 4096
SO_RCVTIMEO: default = 0 sec, 0 usec
SO_SNDTIMEO: default = 0 sec, 0 usec
SO_REUSEADDR: default = off
SO_REUSEPORT: (undefined)
SO_TYPE: default = 1
SO_USELOOPBACK:default = off
IP_TOS: default = 0
IP_TTL: default = 60
TCP_MAXSEG: default = 512
TCP_NODELAY: default = off
```

选项SO_TYPE的返回值1对应于此实现的SOCK_STREAM。

7.4 套接口状态

对于某些套接口选项，针对套接口的状态，什么时候进行设置或获取选项有时序上的考虑。我们对受影响的选项论及这一点。

下面的套接口选项是由TCP已连接套接口从监听套接口继承来的(TCPv2第462～463页)：SO_DEBUG、SO_DONTROUTE、SO_KEEPALIVE、SO_LINGER、SO_OOBINLINE、SO_RCVBUF和SO_SNDBUF。这对TCP是很重要的，因为accept一直要到TCP层完成三路握手后才会给服务器返回已连接套接口。如果想在三路握手完成时确保这些套接口选项中的某一个是给已连接套接口设置的，我们必须先给监听套接口设置此选项。

7.5 基本套接口选项

我们从基本套接口选项开始讨论。这些选项是协议无关的(也就是说，它们由内核中的协议无关代码处理，而不是由诸如IPv4这样的一类特殊的协议模块处理)，但有些选项仅能应用到某些确定类型的套接口中。例如，尽管SO_BROADCAST套接口选项称为“基本选

项",它也仅应用于数据报套接口。

SO_BROADCAST 套接口选项

此选项使能或禁止进程发送广播消息的能力。只有数据报套接口支持广播,并且还必须是在支持广播消息的网络上(例如以太网、令牌环网等)。你不可能在一个点对点链路上进行广播。第 18 章我们将更为详细地讨论广播。

由于一个应用进程在发送一个广播数据报之前必须设置此套接口选项,因此它能有效防止该进程在应用程序未设计成能广播时就发送广播消息。例如,一个 UDP 应用程序可能将目的 IP 地址作为命令行参数,但它并不期望用户键入一个广播地址。处理方法并非让应用进程来确定一给定地址是否为广播地址,而是把测试放在内核中进行:如果目的地址是一个广播地址且此套接口选项没有设置,则返回 EACCES(TCPv2 第 233 页)。

SO_DEBUG 套接口选项

此选项仅由 TCP 支持。当给一个 TCP 套接口打开此选项时,内核对 TCP 在此套接口所发送和接收的所有分组跟踪详细信息。这些信息保存在内核的环形缓冲区中,可由程序 trpt 来进行检查。TCPv2 第 916～920 页提供了更为详细的信息和使用了此选项的一个例子。

SO_DONTROUTE 套接口选项

此选项规定发出的分组将旁路底层协议的正常路由机制。例如,对于 IPv4,分组被指向适当的本地接口,也就是目的地址的网络和子网部分所确定的本地接口。如果本地接口不能由目的地址确定(例如,目的主机不在一个点对点链路的另一端上,也不在一个共享网络上),则返回 ENETUNREACH 错误。

给函数 send、sendto 或 sendmsg 使用标志 MSG_DONTROUTE 也能在个别的数据报上取得同样效果。

此选项经常由路由守护进程(routed 和 gated)用来旁路路由表(路由表不正确的情况下),强制一个分组从某个特定接口发出。

SO_ERROR 套接口选项

当套接口上发生错误时,源自 Berkeley 的内核中的协议模块将此套接口的名为 so_error 的变量设为标准的 Unix Exxx 值中的一个,它称为此套接口的待处理错误(pending error)。内核可立即以下面两种方式通知进程:

1. 如果进程阻塞于对此套接口的 select 调用(6.3 节),则无论是检查可读条件还是可写条件,select 均返回并设置其中一个或所有两个条件。
2. 如果进程使用信号驱动 I/O 模型(第 22 章),则给进程或进程组生成信号 SIGIO。

进程然后可以通过获取 SO_ERROR 套接口选项来得到 so_error 的值。由 getsockopt 返回的整数值就是此套接口的待处理错误。so_error 随后由内核复位为 0(TCPv2 第 547 页)。

当进程调用 read 且没有数据返回时,如果 so_error 为非 0 值,则 read 返回 −1 且 errno 设为 so_error 的值(TCPv2 第 516 页),接着 so-error 的值被复位为 0。如果此套接口上有数

据在排队，则 read 返回那些数据而不是返回错误条件。如果在进程调用 write 时 so_error 为非 0 值，则 write 返回 −1 且 errno 设为 so_error 的值(TCPv2 第 495 页)，so_error 也被复位为 0。

> TCPv2 第 495 页所示代码中有一个缺陷，那儿 so_error 没有被复位为 0，这在 BSD/OS 的版本中已经修改了。任何时候某个套接口有待处理错误返回时，它都必须被复位为 0。

这是我们遇到的可以获取但不能设置的第一个套接口选项。

SO_KEEPALIVE 套接口选项

给一个 TCP 套接口设置保持存活(keepalive)选项后，如果 2 小时内在此套接口的任一方向都没有数据交换，TCP 就自动给对方发一个保持存活探测分节(keepalive probe)。这是一个对方必须响应的 TCP 分节，它会导致以下三种情况：

1. 对方以期望的 ACK 响应。应用进程得不到通知(因为一切正常)。又过仍无动静的 2 小时后，TCP 将发出另一个探测分节。
2. 对方以 RST 响应，它告诉本地 TCP，对方已崩溃且已重新启动。套接口的待处理错误被置为 ECONNRESET，套接口本身则被关闭。
3. 对方对保持存活探测分节无任何响应。源自 Berkeley 的 TCP 发送另外 8 个探测分节，相隔 75 秒一个，试图得到一个响应。TCP 在发出第一个探测分节 11 分钟 15 秒后若仍无响应就放弃。如果对 TCP 的所有探测分节根本就没有响应，套接口的待处理错误被置为 ETIMEOUT，套接口本身则被关闭。但如果套接口接收到一个 ICMP 错误作为某个探测分节的响应，则返回相应的错误(图 A.15 和图 A.16)，套接口也被关闭。此情形中一个常见的 ICMP 错误是“host unreachable(主机不可达)”，说明对方主机并没有崩溃，但是不可达，这种情况下待处理错误被置为 EHOSTUNREACH。

TCPv1 第 23 章和 TCPv2 第 828～831 页均有对保持存活选项的详细叙述。

对于此选项的一个最常见的问题就是时间参数是否可改(常常是想将 2 小时的无活动周期改为短些的值)。我们在 7.9 节描述新的 Posix.1g 选项 TCP_KEEPALIVE，但它没有广泛实现。TCPv1 的附录 E 讨论了如何给各种内核修改这些定时参数，但必须注意，大多数内核是以整个内核为基维护这些时间参数的，而不是以每个套接口为基来维护的，因此若将无活动周期从 2 小时改为(譬如说)15 分钟，则将影响到主机上所有打开了此选项的套接口。

这个选项的目的是检测对方主机是否崩溃。如果对方进程崩溃，它的 TCP 将跨连接发送一个 FIN，这可以通过调用 select 很容易地检测到。(这就是为什么我们在 6.4 节中使用 select 的原因。)同时也要认识到，即使对任何保持存活探测分节均无响应(第三种情况)，我们也不能肯定对方主机已崩溃，因而 TCP 可能会终止一个有效连接。某个中间路由器崩溃 15 分钟是有可能的，而这段时间正好与主机的 11 分钟又 15 秒的保持存活探测周期完全重迭。

此选项一般由服务器使用，尽管客户也可用此选项。服务器用此选项是因为它们花大部

分时间阻塞于等待跨 TCP 连接的输入上，也就是说，等待客户请求。但如果客户主机崩溃，服务器进程将永远不会知道，并将继续等待永远不会到达的输入，这称为半开连接(half-open connection)。保持存活选项将检测出这些半开连接并终止它们。

> 大多数 Rlogin 和 Telnet 服务器设置此选项，以在交互式客户未注销就挂断电话线或关掉终端电源的情况下也能终止连接。
>
> 有些服务器，常常是 FTP 服务器，提供一个分钟数量级的超时。这是由应用进程本身完成的，一般在读下一个客户命令的 read 调用附近。这个超时与此套接口选项无关。

图 7.5 对一个 TCP 连接的另一端发生某些事件时我们必须检测的各种方法作了总结。当我们说“使用 select 判断可读条件”时，意味着要调用 select 来检测套接口是否可读。

情　形	对方进程崩溃	对方主机崩溃	对方主机不可达
本地 TCP 正主动发送数据	对方 TCP 发送一个 FIN，我们通过使用 select 判断可读条件立即能检测出来。如果本地 TCP 发送另外一个分节，对方 TCP 就以 RST 响应。如果再发送另外一个分节，本地 TCP 就给我们发一个 SIGPIPE 信号	本地 TCP 将超时，且套接口的待处理错误被设置为 ETIMEDOUT	本地 TCP 将超时，且套接口的待处理错误被设置为 EHOSTUNREACH
本地 TCP 正主动接收数据	对方 TCP 将发送一个 FIN，我们将把它作为一个(可能是过早的)文件结束符读入	我们将停止接收数据	我们将停止接收数据
连接空闲，保持存活选项已设	对方 TCP 发送一个 FIN，我们通过使用 select 判断可读条件立即能检测出来	在毫无动静 2 小时后，发送 9 个保持存活探测分节，然后套接口的待处理错误被设置为 ETIMEDOUT	在毫无动静 2 小时后，发送 9 个保持存活探测分节，然后套接口的待处理错误被设置为 EHOSTUNREACH
连接空闲，保持存活选项未设	对方 TCP 发送一个 FIN，我们通过使用 select 判断可读条件立即能检测出来	(无)	(无)

图 7.5　检测各种 TCP 条件的方法

SO_LINGER 套接口选项

此选项指定函数 close 对面向连接的协议如何操作(例如对 TCP 而不是对 UDP)。缺省操作是 close 立即返回，但如果有数据残留在套接口发送缓冲区中，系统将试着将这些数据发送给对方。

SO_LINGER 套接口选项使我们可以改变这个缺省设置。此选项要求在用户进程与内核间传递如下结构，它在头文件<sys/socket.h>中定义：

```
struct linger {
int l_onoff;    /* 0=off, nonzero=on */
int l_linger;   /* linger time, Posix.1g specifies units as seconds */
};
```

对 setsockopt 的调用将依两个结构成员的值导致下列三种情况的某一种：

1. 如果 l_onoff 为 0，则选项关闭，l_linger 的值被忽略且前面讨论的 TCP 缺省设置生效：close 立即返回。
2. 如果 l_onoff 为非 0 值且 l_linger 为 0，那么当套接口关闭时 TCP 夭折连接(TCPv2 第 1019～1020 页)。TCP 将丢弃保留在套接口发送缓冲区中的任何数据并发送一个 RST 给对方，而不是通常的四分组连接终止序列(2.5 节)。我们在图 15.21 中给出了这样的一个例子。这避免了 TCP 的 TIME_WAIT 状态，但这样做也会有以下可能性：在 2MSL 秒内创建此连接的另一个化身，并使得老的来自刚被终止的连接上的重复分节不正确地递送到新的化身上(2.6 节)。

   有些实现，尤其是 solaris 2.x(此处 x≤5)，不实现 SO_LINGER 选项的这个特性。

   偶尔张贴在 Usenet 上的消息提倡使用此特性，其目的仅仅为了避免 TIME_WAIT 状态，或者在即使跟某个服务器的众所周知端口的连接仍在使用的情况下也能重启其监听服务器。这万万不可，它可能导致数据破坏，细节见 RFC 1337[Braden 1992a]。作为替代，总是在服务器程序中调用 bind 前使用 SO_REUSEADDR 套接口选项，我们马上会论述到。TIME_WAIT 状态是我们的朋友，它是有助于我们的(也就是说，使旧的重复分节在网络中超时消失)。不要试图避免这个状态，我们应该弄清楚它(2.6 节)。

3. 如果 l_onoff 为非 0 值且 l_linger 也为非 0 值，那么当套接口关闭时内核将拖延一段时间。也就是说，如果在套接口发送缓冲区中仍残留有数据，进程将处于睡眠状态，一直到(a)所有数据都已发送完且均被对方确认或(b)延滞时间到。如果套接口被设置为非阻塞型(第 15 章)，它将不等待 close 完成，即使延滞时间为非 0 也是如此。当使用 SO_LINGER 选项的这个特性时，应用进程检查 close 的返回值是非常重要的，因为，如果在数据发送完并被确认前延滞时间到的话，close 将返回 EWOULDBLOCK 错误且套接口发送缓冲区中的任何数据都丢失。

   不幸的是，第三种情况中对成员 l_linger 非 0 的解释是依赖不同的实现的。4.4BSD假设其单位是时钟滴答(clock tick)(一百分之一秒)，但 Posix.1g 规定其单位为秒。现有的源自 Berkeley 的实现的另一个问题是成员 l_linger(一个整数)是拷贝到 16 位的带符号整数内核变量(so_linger)上的，这就限制了延滞时间最大为 32767 时钟滴答即 327.67 秒。

现在我们需要看看，对于已讨论的各种情况，套接口上的 close 确切来说是什么时候返回的。我们假设客户写数据到套接口上，然后调用 close。图 7.6 给出了缺省情况。

我们假设在客户数据到达时，服务器暂时处于忙状态。所以数据由 TCP 加入到其套接口接收缓冲区中。类似地，下一个分节即客户的 FIN 也加入到套接口接收缓冲区中(不论实

现以何种方法记录连接上已收到一个 FIN 的事件)。但是缺省时,客户的 close 立即返回。如图所示,客户的 close 可以在服务器读套接口接收缓区冲的剩余数据之前就返回。对于服务器主机来说,在服务器应用进程读此剩余数据之前就崩溃是完全可能的,且客户应用进程永远不会知道。

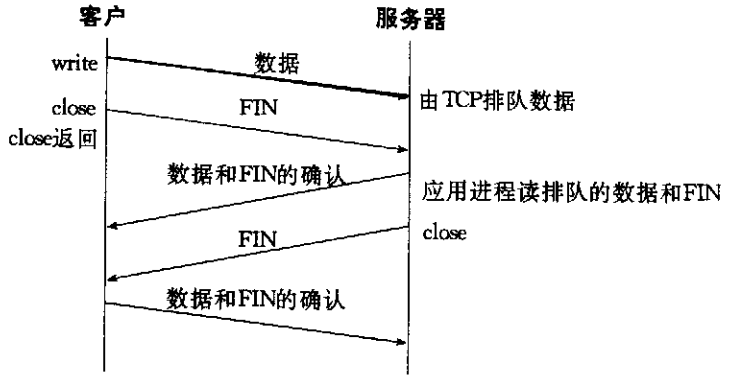


图 7.6　close 的缺省操作:立即返回

客户可以设置 SO_LINGER 套接口选项,指定一个正的延滞时间。这种情况下,客户的 close 要直到它的数据和 FIN 已被 TCP 服务器确认后才返回,如图 7.7 所示。但我们仍有与图7.6类似的问题:在服务器应用进程读剩余数据之前,服务器主机可能崩溃,并且客户应用进程永远不会知道。

这里有一个基本原则:设置 SO_LINGER 套接口选项后,close 的成功返回仅告诉我们发送的数据(和 FIN)已由对方 TCP 确认,它并不能告诉我们对方应用进程是否已读了数据。如果不设置该选项,我们连对方 TCP 是否确认了数据都不知道。

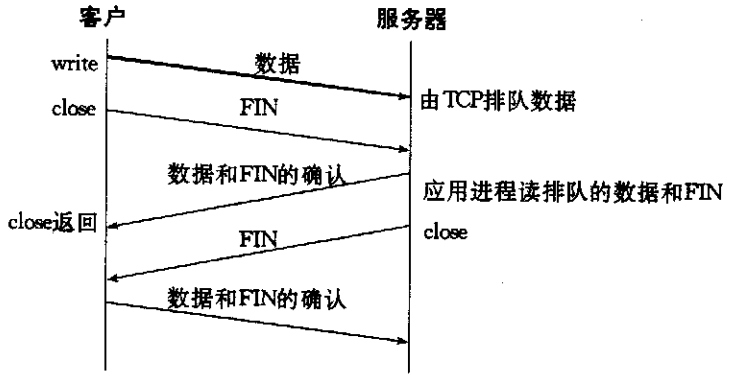


图 7.7　设置 SO_LINGER 套接口选项且 l_linger 为正值时的 close

让客户知道服务器已读其数据的一个方法是:调用 shutdown(第二个参数设为 SHUT_WR)而不是调用 close,并等待对方 close 连接的本地(服务器)端,如图 7.8 所示。

将此图与图 7.6 和图 7.7 进行比较,我们发现,当关闭连接的客户端时,根据所调用的函数(close 或 shutdown)以及是否设置了 SO_LINGER 套接口选项,可在以下三个不同的时机返回:

1. close 立即返回，不等待(缺省状况，图 7.6)。
2. close 一直拖延到接收了 FIN 的 ACK(图 7.7)。

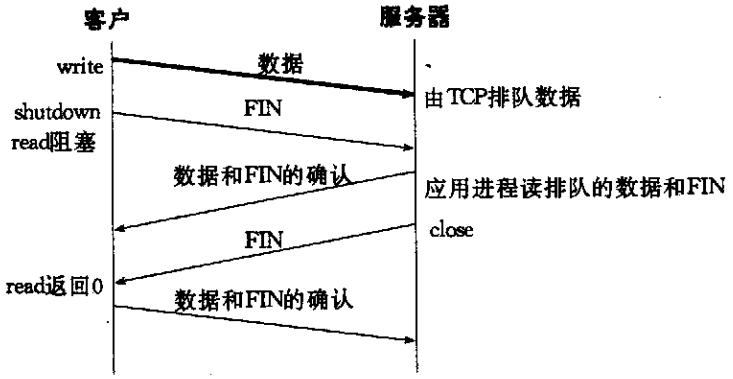


图 7.8 用 shutdown 来获知对方已接收数据

3. 后跟一个 read 调用的 shutdown 一直等到接收了对方的 FIN 才返回(图 7.8)。

获知对方应用进程已读我们的数据的另外一个方法是：使用一个应用级的确认(application-level acknowledge)即应用 ACK(application ACK)。例如，客户给服务器发数据后，调用 read 来读 1 个字节的数据：

```
char ack;
Write(sockfd, data, nbytes);        /* data from client to server */
n = Read(sockfd, &ack, 1);          /* wait for application-level ACK */
```

服务器从客户读数据后，发回 1 字节的应用级 ACK：

```
nbytes = Read(sockfd, buff, sizeof(buff));      /* data from client */
        /* server verifies it received the correct
           amount of data from the client */
Write(sockfd, "", 1);                           /* server's ACK back to client */
```

当客户上的 read 返回时，我们可以保证服务器进程已读完了我们所发送的所有数据。(假设服务器知道客户要发送多少数据，或是由应用程序定义了某种记录结束标志，但这儿没有给出。)这里，应用级 ACK 是 1 字节的 0，但此字节的内容还可以用来指示从服务器到客户的其他条件。图7.9展示了可能进行的分组交换过程：

图 7.10 总结了对 shutdown 的两种可能调用和对 close 的三种可能调用，以及它们对 TCP 套接口的影响。

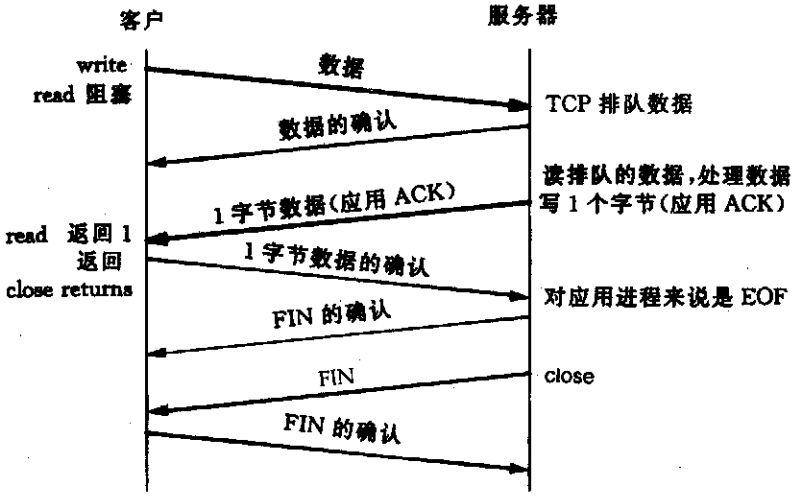


图 7.9　应用 ACK

函数	说　明
shutdown，SHUT_RD	在套接口上不能再发出接收请求；进程仍然可往套接口上发送；套接口接收缓冲区中所有数据被丢弃；再接收的任何数据都由 TCP 丢弃(习题 6.5)；对套接口发送缓冲区无任何影响
shutdown，SHUT_WR	在套接口上不能再发出发送请求；进程仍然可从套接口上接收；套接口发送缓冲区中的内容被发往另一端，后跟正常的 TCP 连接终止序列(FIN)；对套接口接收缓冲区无任何影响
close，l_onoff = 0 (缺省情况)	在套接口上不能再发出发送或接收请求；套接口发送缓冲区中的内容被发往另一端。如果描述字访问计数变为 0：在发完发送缓冲区中的数据后，发出正常的 TCP 连接终止序列(FIN)，套接口接收缓冲区中内容被丢弃
close，l_onoff = 1 l_linger = 0	在套接口上不能再发出发送或接收请求。如果描述字访问计数变为 0：RST 被发往另一端，连接状态被置为 CLOSED(无 TIME_WAIT 状态)，套接口发送缓冲区和套接口接收缓冲区中的数据被丢弃
close，l_onoff = 1 l_linger ! = 0	在套接口上不能再发出发送或接收请求；套接口发送缓冲区中的数据被发往另一端。如果描述字访问计数变为 0：在发完发送缓冲区中的数据后，发出正常的 TCP 连接终止序列(FIN)，套接口接收缓冲区中内容被丢弃，如果在连接变为 CLOSED 状态前延滞时间到，则 close 返回 EWOULDBLOCK 错误

图 7.10　shutdown 和 SO_LINGER 各种情况的总结

SO_OOBINLINE 套接口选项

当此选项打开时，带外数据将被留在正常的输入队列中(即在线存放)。当发生这种情况时，接收函数的 MSG_OOB 标志不能用来读带外数据。第 21 章中我们将详细讨论带外数据。

SO_RCVBUF 和 SO_SNDBUF 套接口选项

每个套接口都有一个发送缓冲区和一个接收缓冲区。在图 2.11 和图 2.12 中，我们描述了 TCP 和 UDP 发送缓冲区的操作。

接收缓冲区被 TCP 和 UDP 用来将接收到的数据一直保存到由应用进程来读。对于 TCP 来说，套接口接收缓冲区中可用空间的大小就是 TCP 通告另一端的窗口大小。TCP 套接口接收缓冲区不可能溢出，因为对方不允许发出超过所通告窗口大小的数据。这就是 TCP 的流量控制，如果对方无视窗口大小而发出了超过窗口大小的数据，则接收方 TCP 将丢弃它。然而，对于 UDP 来说，当接收到的数据报装不进套接口接收缓冲区时，此数据报就被丢弃。回忆一下，UDP 是没有流量控制的：快的发送者可以很容易地就淹没慢的接收者，导致接收方的 UDP 丢弃数据报。这一点我们在 8.13 节再作说明。

这两个套接口选项使我们可以改变缺省大小。对于不同的实现，缺省值的大小可以有很大的差别。较早期的源自 Berkeley 的实现将 TCP 发送和接收缓冲区的大小均缺省为 4 096 字节，但较新的系统使用较大的值，可以是 8 192～61 440 字节间的任一数据。如果主机支持 NFS，则 UDP 发送缓冲区的大小经常缺省为 9 000 字节左右的一个值，而 UDP 接收缓冲区的大小则经常缺省为 40 000 字节左右的一个值。

当设置 TCP 套接口接收缓冲区的大小时，函数调用的顺序是很重要的，这是因为 TCP 的窗口规模选项(2.5 节)是在建立连接时用 SYN 与对方互换得到的。对于一个客户，这意味着 SO_RCVBUF 选项必须在调用 connect 之前设置；对于一个服务器，这意味着在调用 listen 之前必须给监听套接口设置这个选项。给已连接套接口设置这个选项对可能的窗口规模选项无任何影响，因为 accept 要直到 TCP 的三路握手完成才会创建并返回已连接套接口。这就是为什么必须给监听套接口设置此选项的原因(套接口缓冲区的大小总是由新创建的已连接套接口从监听套接口继承来的：TCPv2 第 462～463 页)。

TCP 套接口缓冲区的大小至少必须是连接的 MSS 的三倍。如果我们涉及到单向数据传输，如单方向的文件传送，当我们说“套接口缓冲区大小”时，我们指发送主机端的套接口发送缓冲区大小和接收主机端的套接口接收缓冲区大小。对于双向数据传输，在发送端我们指两个套接口缓冲区的大小，在接收端也指两个套接口缓冲区的大小。典型的缓冲区大小缺省值是 8192 字节或更大，典型的 MSS 为 512 或 1460，因此 TCP 套接口缓冲区大小的要求一般总能满足。在大 MTU 的网络上会出现问题，因为它提供一个比通常 MSS 更大的值(例如，[Comer 和 Lin 1994]中描述的 ATM 网络的 MTU 为 9188)。

TCP 套接口缓冲区大小还必须是连接的 MSS 的偶数倍。有些实现给应用进程处理这个细节问题，在连接建立后向上舍入套接口缓冲区大小(TCPv2 第 902 页)。这是在建立连接之前设置这两个套接口选项的另外一个原因。例如，使用缺省的 4.4BSD 大小 8192，并假设以太网的 MSS 为 1460，在连接建立时两个套接口缓冲区向上舍入成 8760(6×1460)。

在设置套接口缓冲区大小时另一个需考虑的问题涉及到性能。图 7.11 所示为容量为 8 个分节的两端点间的 TCP 连接(我们称其为管道)。

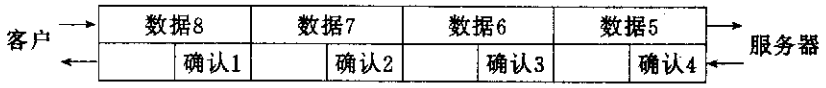


图 7.11　8 个分节容量的 TCP 连接(管道)

我们在上面给出四个数据分节，下面给出四个 ACK。即使管道中只有四个数据分节，客户也必须有至少 8 个分节容量的发送缓冲区，因为客户 TCP 必须为每个分节保留一个拷贝，直到接收到来自服务器的 ACK。

> 这里我们忽略了一些细节。首先，TCP 的慢启动算法限制了在一个空闲连接上最初发送分节的速度；其次，TCP 常常每两个分节确认一次，而不是我们所示的每个分节确认一次。所有这些细节在 TCPv1 的第 20 章和第 24 章均有阐述。

理解的重点是全双工管道的概念、它的容量及如何关系到连接两端的套接口缓冲区大小。管道的容量称为带宽-延迟积(bandwidth-delay product)，我们可以将带宽(位/秒)乘上 RTT(秒)，并将结果由位转换为字节来计算得到。RTT 可很容易地用程序 Ping 来测量到，带宽是两端点间相应于最慢链路的值，某种程度上是已知的。例如，一条 RTT 为 63ms 的 T1 链路(1 536 000 位/秒)的带宽-延迟积为 11 520 字节。如果套接口缓冲区大小小于此值，管道将不会处于满状态，性能将低于期望值。当带宽变大(如 45 兆位/秒的 T3 链路)或 RTT 变大(如 RTT 约为 500ms 的卫星链路)时，就要求有较大的套接口缓冲区。当带宽-延迟积超过 TCP 的最大正常窗口大小(65535 字节)时，两端都需设置 TCP 长胖管道(long fat pipe)选项，这在 2.5 节中已有叙述。

> 大多数实现对套接口发送和接收缓冲区的大小有一个上限，有时这个上限可由管理员来进行修改。较早期的源自 Berkeley 的实现有一个大约 52 000 字节的硬上限，但较新的实现将缺省值上升为 256 000 字节甚至更大，且还常常可由管理员来增加。不幸的是，对于应用程序来说，没有一个简单的方法来确定此极限。Posix.1 定义了函数 fpathconf，这个函数大多数实现都支持；Posix.1g 定义了一个新的常值_PC_SOCK_MAXBUF，它可用作此函数的第二个参数，返回套接口缓冲区的最大值。

SO_RCVLOWAT 和 SO_SNDLOWAT 套接口选项

每个套接口也都有一个接收低潮限度和一个发送低潮限度。它们是函数 select 使用的，如6.3节所述。这两个套接口选项使我们可以修改这两个低潮限度。

接收低潮限度是让 select 返回“可读”而在套接口接收缓冲区中必须有的数据总量。对于一个 TCP 或 UDP 套接口，此值缺省为 1。发送低潮限度是让 select 返回“可写”而在套接口发送缓冲区中必须有的可用空间。对于 TCP 套接口，此值常缺省为 2048。对于 UDP 使用低潮限度的情况，在 6.3 节已作过讲述，但由于对于 UDP 套接口来说，发送缓冲区中可用空间的字节数是从不变化的(因为 UDP 不保留由应用进程发送的数据报的拷贝)，只要 UDP 套接口发送缓冲区大小大于套接口的低潮限度，这样的 UDP 套接口就总是可写的。回忆一下图 2.12，UDP 没有发送缓冲区，只有发送缓冲区的大小。

Posix.1g 不要求支持这两个套接口选项。

SO_RCVTIMEO 和 SO_SNDTIMEO 套接口选项

这两个选项使得我们可以给套接口设置一个接收和发送超时。注意，访问它们的两个 sockopt 函数的参数是指向 timeval 结构的指针，与 select 所用参数相同(6.3 节)，这可让我们用秒数和微秒数来规定超时。我们通过设置其值为 0 秒和 0 微秒来禁止超时。缺省时两个超时都是禁止的。

接收超时影响五个输入函数：read、readv、recv、recvfrom 和 recvmsg；发送超时影响五个输出函数：write、writev、send、sendto 和 sendmsg。13.2 节我们将详细讨论关于套接口的超时。

这两个套接口选项以及套接口接收和发送超时的继承概念是在 4.3BSD Reno 中增加的。Posix.1g 不要求支持这两个套接口选项。

在源自 Berkeley 的实现中，这两个值实际上实现不活动时间计时器而不是读或写系统调用的绝对计时器，TCPv2 第 496～516 页对此作了详细讨论。

SO_REUSEADDR 和 SO_REUSEPORT 套接口选项

SO_REUSEADDR 套接口选项为以下四个不同的目的提供服务：

1. SO_REUSEADDR 允许启动一个监听服务器并捆绑其众所周知端口，即使以前建立的将此端口用作它们的本地端口的连接仍存在。这个条件通常是这样碰到的：

   (a) 启动一个监听服务器；
   (b) 连接请求到达，派生一个子进程来处理这个客户；
   (c) 监听服务器终止，但子进程继续为现有连接上的客户提供服务；
   (d) 重启监听服务器。

   缺省时，当监听服务器在步骤(d)通过调用 socket、bind 和 listen 重新启动时，因为它试图捆绑一个现有连接上的端口(正由以前派生的子进程所处理)，所以调用 bind 失败。但若服务器在 socket 和 bind 的调用间设置 SO_REUSEADDR 套接口选项，bind 将成功。所有 TCP 服务器应指定此套接口选项以允许服务器在这种情况下被重新启动。

   这个情况是 Usenet 中问得最频繁的一个问题。

2. SO_REUSEADDR 允许在同一端口上启动同一服务器的多个实例，只要每个实例捆绑一个不同的本地 IP 地址即可。这对用 IP 别名技术(A.4 节)来提供多个 HTTP 服务器的网点(site)来说是很常见的①。假设本地主机的主 IP 地址为 198.69.10.2，但它有两个别名：198.69.10.128 和 198.69.10.129。启动三个 HTTP 服务器。第一

① 译者注：网点(site)是一个用得越来越广泛的概念，但它没有精确的定义。网点概念最初指互联网上的单个子网或网络。从网络路由拓扑图看，子网或网络用点来表示，而把它们连接起来的路由器则用边来表示。因而有网点的说法。随着子网划分(subnetting)、超网合并(supernetting)、无类域间路由(CIDR)等技术的应用，网点能表示的范围也越来越大，但最大不会超过一个自治系统(AS)。另外，网点概念也用在逻辑范围，例如同一 NIS 域的所有主机，同一 Kerberos 域的所有主机等等都能构成单个网点。本书所指的网点应小于一个组织，例如，一家跨国公司(组织)的各个子公司可能就是单个网点。site 有时指单个主机，这时我们把它译成站点。

个 HTTP 服务器以本地 IP 地址 198.69.10.128 和本地端口号 80(HTTP 的众所周知)调用 bind。第二个 HTTP 服务器捆绑 198.69.10.129 和端口 80,但它的 bind 调用将失败,除非在调用前设置 SO_REUSEADDR 选项。第三个 HTTP 服务器以通配地址作为本地 IP 地址和端口号 80 调用 bind。同样,设置 SO_REUSEADDR 是这个调用成功所必需的。假设设置了 SO_REUSEADDR,从而三个服务器都启动,目的 IP 地址为 198.69.10.128、目的端口号为 80 的外来 TCP 连接请求将递送给第一个服务器,目的 IP 地址为 198.69.10.129、目的端口号为 80 的外来请求将递送给第二个服务器,目的端口号为 80 的所有其他请求将都递送给第三个服务器。最后一个服务器处理目的地址为 198.69.10.2 或该主机已配置的其他任何 IP 别名的请求。通配的意思就是"没有更好的(即更为具体的)匹配的任何地址"。注意,允许某个给定服务存在多个服务器的情形在服务器总是设置 SO_REUSEADDR 套接口选项时是自动处理的(我们建议设置这个选项)。

对于 TCP,我们根本不能启动捆绑相同 IP 地址和相同端口号的多个服务器:这是完全重复的捆绑(completely duplicate binding)。也就是说,即使我们给第二个服务器设置了 SO_REUSEADDR 套接口选项,也不能在启动绑定 198.69.10.2 和端口 80 的服务器后,接着再启动捆绑 198.69.10.2 和端口 80 的另一个服务器。

3. SO_REUSEADDR 允许单个进程捆绑同一端口到多个套接口上,只要每个捆绑指定不同的本地 IP 地址即可。在不支持 IP_RECVDSTADDR 套接口选项的系统上,这对于要求知道客户请求的目的 IP 地址的 UDP 服务器来说是非常普遍的,在 19.11节中我们将用此技术开发一个例子。这项技术一般不用于 TCP 服务器,因为 TCP 服务器在建立连接后总是能通过调用 getsockname 来确定客户请求的目的 IP 地址。
4. SO_REUSEADDR 允许完全重复的捆绑:当一个 IP 地址和端口绑定到某个套接口上时,还允许此 IP 地址和端口捆绑到另一个套接口上。一般来说,这个特性仅在支持多播的系统上才有,这些系统可能还不支持 SO_REUSEPORT 套接口选项(我们马上叙述到),而且仅对 UDP 套接口而言(TCP 不进行多播)。

   此特性用于多播时,允许同一主机上多次运行相同的应用程序。当一个 UDP 数据报需由这些重复捆绑套接口中的一个接收时,所用规则为:如果数据报的目的地址是一个广播地址或多播地址,就给每个匹配的套接口递送数据报的拷贝。但是如果数据报的目的地址是一个单播地址,那它只递送到单个套接口。在单播数据报情况下,如果有多个套接口匹配数据报,则哪个套接口将接收数据报是依赖于实现的。TCPv2 第 777～779 页对此特性作了详细的讨论。我们将在第 18 章和第 19 章对广播和多播作详细的讨论。

习题 7.5 和 7.6 给出了几个此套接口选项的例子。

添加多播支持以后,4.4BSD 引入了 SO_REUSEPORT 套接口选项。与多播中完全重复捆绑的 SO_REUSEADDR 不同,该套接口选项具有以下语义:

1. 此选项允许完全重复捆绑,但仅在每个想捆绑相同 IP 地址和端口的套接口都指定了此套接口选项才行。

2. 如果被捆绑的IP地址是一个多播地址，则SO_REUSEADDR与SO_REUSEPORT等效(TCPv2第731页)。

此套接口选项的一个问题是：并非所有系统都支持它，在那些不支持此选项但支持多播的系统上，使用SO_REUSEADDR而不是SO_REUSEPORT以允许完全重复捆绑(也就是说，同一时刻在同一主机上可运行多次并期待接收广播或多播数据报的UDP服务器)。

我们以下面的建议来总结对这些套接口选项的讨论：

1. 在所有TCP服务器程序中，在调用bind之前设置SO_REUSEADDR套接口选项；
2. 当编写一个同一时刻在同一主机上可运行多次的多播应用程序时，设置SO_REUSEADDR套接口选项，并将本组的多播地址作为本地IP地址捆绑。

TCPv2第22章对这两个套接口选项作了详细的讨论。

SO_REUSEADDR有一个潜在的安全问题。譬如说，如果存在一个套接口，它绑定了通配地址和端口5555，如果我们指定SO_REUSEADDR，我们可以捆绑相同的端口到不同的IP地址上，譬如说主机的主IP地址。目的地为端口5555和我们的套接口上所绑定IP地址的数据报将递送到我们的套接口上，而不是绑定通配地址的另一个套接口上。它们可以是TCP SYN分节或UDP数据报。(习题11.3展示了UDP的这个特性。)对于大多数众所周知的服务如HTTP、FTP和Telnet来说，这不成问题，因为这些服务器绑定一个保留端口。因此，后来的试图更为详细地捆绑此端口的任何实例(也就是说盗用端口)将要求有超级用户特权。然而，NFS可能是一个问题，因为它的通常端口(2049)不被保留。

> 套接口API的一个底层问题是：套接口对的设置是由两个函数调用(bind和connect)来完成的，而不是一个。[Torek 1994]为解决此问题提议了如下单个函数：
>
> ```
> int bind_connect_listen(int sockfd,
>                         const struct sockaddr *laddr, int laddrlen,
>                         const struct sockaddr *faddr, int faddrlen,
>                         int listen);
> ```
>
> laddr指定本地IP地址和本地端口号，faddr指定远程IP地址和远程端口号，listen指定一个客户(0)或服务器(非0，与函数listen的backlog参数相同)。这样的话，bind将是一个用空指针的faddr和为0的faddrlen来调用此函数的库函数，connect将是一个用空指针的laddr和为0的laddrlen来调用此函数的库函数。有些应用程序，尤其是FTP，需要指明本地对和远程对，它们可以直接调用bind_connect_listen。有了这样的一个函数就不需要SO_REUSEADDR了，除非多播服务器显式地要求允许(对同一IP地址和端口)完全重复捆绑。此函数的另一个好处是：TCP服务器可以限制自己为来自指定的IP地址和端口的连接请求提供服务，这是RFC 793[Postel 1981c]规定的，但对现有的套接口API又是不可能的事情。
>
> 一个类似的、名为set_addresses的函数在1993年由Keith Sklower提议到了Posix的1003.12工作组上，不过这个提议被拒绝了。

SO_TYPE 套接口选项

这个选项返回套接口的类型，返回的整数值是一个诸如 SOCK_STREAM 或 SOCK_DGRAM 这样的值。此选项典型地由启动时继承了套接口的进程所用。

SO_USELOOPBACK 套接口选项

此选项仅用于路由域(AF_ROUTE)的套接口，它对这些套接口的缺省设置为打开(这是唯一一个缺省为打开而不是关闭的 SO_xxx 套接口选项)。当此选项打开时，套接口接收在其上发送的任何数据的一个拷贝。

> 禁止这些回馈拷贝的另一个方法是调用 shutdown，第二个参数应设为 SHUT_RD。

7.6 IPv4 套接口选项

这些选项由 IPv4 处理，级别为 IPPROTO_IP。我们将五个多播套接口选项推迟到 19.5 节再做讨论。

> 除 IP_RECVIF 外，本节中我们所描述的所有套接口选项均由 Posix.1g 定义。

IP_HDRINCL 套接口选项

如果此选项给一个原始 IP 套接口(第 25 章)设置，则我们必须为所有发送到此原始套接口上的数据报构造自己的 IP 头部。一般情况下，内核为发送到原始套接口上的数据报构造 IP 头部，但也有某些应用程序(尤其是路由跟踪程序 Traceroute)要构造自己的 IP 头部以取代 IP 可能放到其头部的某些字段的值。

当设置此选项时，我们构造完整的 IP 头部，不过下列情况例外：

- IP 总是计算并存储 IP 头部校验和；
- 如果我们将 IP 标识字段置为 0，内核将设置此字段；
- 如果源 IP 地址是 IN_ADDR_ANY，IP 将它设置为外出接口的主 IP 地址；
- 如何设置 IP 选项是依赖于实现的。有些实现取 IP_OPTIONS 套接口选项中设置的任何 IP 选项，并将它们附加到我们所构造的头部中，而其他实现则要求我们亲自在头部指定任何期望的 IP 选项。

在 26.6 节中，我们将给出此选项的一个例子，TCPv2 第 1056～1057 页提供了此套接口选项额外的详细信息。

IP_OPTIONS 套接口选项

设置此选项允许我们在 IPv4 头部中设置 IP 选项，这要求掌握 IP 头部中 IP 选项的格式信息。在 24.3 节中，我们论述 IPv4 源路径时再讨论这个选项。

IP_RECVDSTADDR 套接口选项

这个套接口选项导致所接收到的 UDP 数据报的目的 IP 地址由函数 recvmsg 作为辅助

数据返回。20.2节中我们给出此选项的一个例子。

IP_RECVIF 套接口选项

这个套接口选项导致所接收到的UDP数据报的接口索引由函数recvmsg作为辅助数据返回,20.2节中我们给出这个选项的一个例子。

> 这是一个新的套接口选项,它由Bill Fenner为FreeBSD和NetBSD的DARTNet测试床而开发[Fenner 1997]。DARTNet是一个用于测试新协议和新应用程序的实验性研究网络。该套接口选项原本以为会加到4.4BSD中,结果却是一直没有加到此版本中。作者使用FreeBSD实现并把它加入到BSD/OS 3.0中。

IP_TOS 套接口选项

此选项使我们可以给TCP或UDP套接口在IP头部中设置服务类型字段(图A.1)。如果我们给此选项调用getsockopt,则放到外出IP数据报头部的TOS字段中的当前值(缺省为0)将返回。还没有方法从接收到的IP数据报中取此值。

我们可以将TOS设置为图7.12所示常值中的一个,它们都在头文件<netinet/ip.h>中定义。

常值	解释
IPTOS_LOWDELAY	最小化延迟
IPTOS_THROUGHPUT	最大化吞吐量
IPTOS_RELIABILITY	最大化可靠性
IPTOS_LOWCOST	最小化成本

图7.12 IPv4服务类型常值

RFC 1349[Almquist 1992]中有TOS字段的详细描述,并介绍此字段应如何给标准的因特网应用程序设置。例如,Telnet和Rlogin应指定IPTOS_LOWDELAY,而FTP传送的数据部分则应指定IPTOS_THROUGHPUT。

IP_TTL 套接口选项

用此选项,我们可以设置和获取系统用于某个给定套接口的缺省TTL值(存活时间字段,见图A.1)。例如,4.4BSD对TCP和UDP套接口都使用缺省值64(这在RFC 1700[Reynolds and Postel 1994]中规定),对原始套接口则使用缺省值255。跟TOS字段一样,调用getsockopt返回系统用于外出数据报的缺省TTL字段值,也就是说没有办法从接收到的数据报中得到此值。图25.18中的Traceroute程序设置了此套接口选项。

7.7 ICMPv6套接口选项

这个唯一的套接口选项由ICMPv6所处理,并具有IPPROTO_ICMPV6级别。

ICMP6_FILTER 套接口选项

此选项使我们可以获取和设置一个 icmp6_filter 结构，它指明 256 个可能的 ICMPv6 消息类型中哪一个传递给在原始套接口上的进程。在 25.4 节我们再讨论此选项。

7.8　IPv6 套接口选项

这些选项由 IPv6 所处理，并具有 IPPROTO_IPV6 级别。我们将五个多播套接口选项推迟到 19.5 节再做讨论。我们注意到，许多这些选项都利用函数 recvmsg 的辅助数据，在 13.6 节我们再对此进行讨论。所有的 IPv6 套接口选项都定义在 RFC 2133[Gilligan et al. 1997]和[Stevens and Thomas 1997]中。

Posix.1g 对 IPv6 未做任何说明。

IPV6_ADDRFORM 套接口选项

这个选项允许套接口从 IPv4 转换到 IPv6，反之亦可。我们在 10.5 节中再讲述这个选项。

IPV6_CHECKSUM 套接口选项

此选项指定用户数据中校验和所处位置的字节偏移。如果此值为非负，则内核将(1)给所有外出分组计算并存储校验和；(2)输入时检查所收到分组的校验和，丢弃带有无效检验和的分组。此选项影响除 ICMPv6 原始套接口外的所有 IPv6 原始套接口。(内核总是给 ICMPv6 原始套接口计算并存储校验和。)如果指定的值为−1(缺省值)，内核在此原始套接口上将不给外出的分组计算并存储较验和，也不检查所收到分组的校验和。

> 所有使用 IPv6 的协议应在它们自己的协议头部有一个校验和。这些校验和包括一个伪头部(pseudoheader)(RFC 1883[Deering 和 Hinden 1995])，它把源 IPv6 地址作为校验和的一部分(这与使用 IPv4 的原始套接口来实现的其他协议有所不同)。这样不必强求使用原始套接口的应用进程进行源地址选择，相反，内核将这么做，并计算、存储含有标准 IPv6 伪头部的检验和。

IPV6_DSTOPTS 套接口选项

设置此选项指明：任何接收到的 IPv6 目标选项都将由 recvmsg 作为辅助数据返回。此选项缺省为关闭。我们在 24.5 节将描述用来创建并处理这些选项的函数。

IPV6_HOPLIMIT 套接口选项

设置此选项指明：接收到的跳限字段将由 recvmsg 作为辅助数据返回。此选项缺省为关闭。我们在 20.8 节中描述此选项。

对 IPv4 而言，是没有办法来取得接收到的存活时间字段的。

IPV6_HOPOPTS 套接口选项

设置此选项指明：任何接收到的 IPv6 步跳选项都将由 recvmsg 作为辅助数据返回。此

选项缺省为关闭。我们在24.5节描述用来创建和处理这些选项的函数。

IPV6_NEXTHOP

这不是一个套接口选项，而是一个可指定给sendmsg的辅助数据对象的类型。此对象以一个套接口地址结构指定某个数据报的下一跳地址。20.8节我们将对此做详细的描述。

IPV6_PKTINFO套接口选项

设置此选项表明，下面关于接收到的IPv6数据报的两条信息将由recvmsg作为辅助数据返回：目的IPv6地址和到达接口索引。在20.8节中我们再描述此选项。

IPV6_PKTOPTIONS套接口选项

大多数IPv6套接口选项假设UDP套接口使用recvmsg和sendmsg所用的辅助数据在内核与应用进程间传递信息。TCP套接口使用IPV6_PKTOPTIONS套接口选项来获取和存储这些值。

由getsockopt和setsockopt所指的缓冲区中的信息与recvmsg或sendmsg利用辅助数据传递的信息是相同的。我们在24.7节中讨论这个选项。

IPV6_RTHDR套接口选项

设置这个选项表明：接收到的IPv6路由头部将由recvmsg作为辅助数据返回。此选项缺省为关闭。我们在24.6节将描述用来创建并处理IPv6路由头部的函数。

IPV6_UNICAST_HOPS套接口选项

此IPv6选项类似于IPv4的IP_TTL套接口选项。它的设置指定发送到套接口上的外出数据报的缺省跳限，而它的获取则返回内核将用于套接口的跳限值。为了从接收到的IPv6数据报中得到真实的跳限字段，要求使用IPV6_HOPLIMIT套接口选项。我们在图25.18的Traceroute程序中就设置此套接口选项。

7.9 TCP套接口选项

TCP有五个套接口选项，但有三个是随Posix.1g新定义的且没有得到广泛支持。我们规定它们的级别为IPPROTO_TCP。

TCP_KEEPALIVE套接口选项

此选项是随Posix.1g新定义的。它指定TCP开始发送保持存活探测分节前以秒为单位的连接空闲时间。缺省值至少必须为7200秒，即2小时。此选项仅在SO_KEEPALIVE套接口选项打开时才有效。

TCP_MAXRT套接口选项

此选项是随Posix.1g新定义的。它指定一旦TCP开始重传数据，在连接断开之前需经历的以秒为单位的时间总量。值0意味着使用系统缺省值，值－1意味着永远重传数据。如果指定一个正值，它可能向上舍入成实现的下一次重传时间。

TCP_MAXSEG 套接口选项

此选项允许我们获取或设置 TCP 连接的最大分节大小(MSS)。返回值是我们的 TCP 发送给另一端的最大数据量,它常常就是由另一端用 SYN 分节通告的 MSS,除非我们的 TCP 选择使用一个比对方通告的 MSS 小些的值。如果此值在套接口连接之前取得,则返回值为未从另一端收到 MSS 选项的情况下所用的缺省值。同时也要注意到,小于此返回值的值可能真正用在连接上,因为譬如说使用时间戳选项的话,它在每个分节上占用 12 字节的 TCP 选项容量。

我们的 TCP 将发送的每个分节的最大数据量也可在连接存活期内改变,但前提是 TCP 要支持路径 MTU 发现功能。如果到对方的路径改变了,此值可上下调整。

在图 7.1 中我们注意到,此套接口选项也可由应用进程设置。在 4.4BSD 出现之前,这是不可能的:它是一个只读选项。4.4BSD 只允许应用进程减少该值:我们不能增加它的值(TCPv2 第 1023 页)。由于这个选项控制 TCP 发送的每个数据分节的数据总量,它禁止应用进程增加值是明智的。一旦连接建立,此值即对方通告的 MSS 选项,我们不能超过此值。然而,我们的 TCP 总是可以发送少于对方通告的 MSS 值的数据。

TCP_NODELAY 套接口选项

如果设置,此选项禁止 TCP 的 Nagle 算法(TCPv1 的 19.4 节和 TCPv2 第 858～859 页)。缺省时,此算法是使能的。

Nagle 算法的目的是减少 WAN 上小分组的数目。算法指出:如果给定连接上有待确认(outstanding)数据(也就是说,我们的 TCP 已发送但还在等待对方确认的数据),则直到现有数据被确认前,将不往连接上发送任何小分组。"小"分组的定义即任何小于 MSS 的分组。如果可能,TCP 总是发送最大大小的分组;Nagle 算法的目的是防止任何时刻连接上有多个待确认的小分组存在。

最容易产生小分组的两个客户是 Rlogin 和 Telnet,因为它们将每次击键都作为一个独立分组发送。在快速 LAN 上,我们一般不会注意到这些客户使用 Nagle 算法,因为小分组要求的确认时间一般也就几毫秒,远远小于我们键入两个连续字符的时间。但在 WAN 上,小分组等待确认的时间可能要达到一秒,因此我们可以观察到字符回显的延迟,且此延迟常由 Nagle 算法进行夸大。

考虑下面的例子。我们键入六字符的串"hello!"到 Rlogin 或 Telnet 客户上,每个字符间准确地相隔 250ms。到服务器的 RTT 是 600ms,且服务器立即发回每个字符的回显。我们假设客户字符的 ACK 是与字符回显一同发回给客户的,且忽略客户发给服务器的回显 ACK。(我们不久将讨论延滞 ACK。)假设 Nagle 算法没有使能,我们有图 7.13 所示的 12 个分组。

每个字符在自己的分组中发送:数据分节从左到右,ACK 从右到左。

但如果使能了 Nagle 算法(缺省),我们有图 7.14 所示的 8 个分组。第一个字符在它自己的分组中发送。但下两个字符不发送了,因为连接有一个小分组没有得到确认。在时刻 600,当收到第一个分组的 ACK 时(与第一个字符回显一起),这两个字符被发出。直到此分组在时刻 1200 被确认,才会发出另外的小分组。

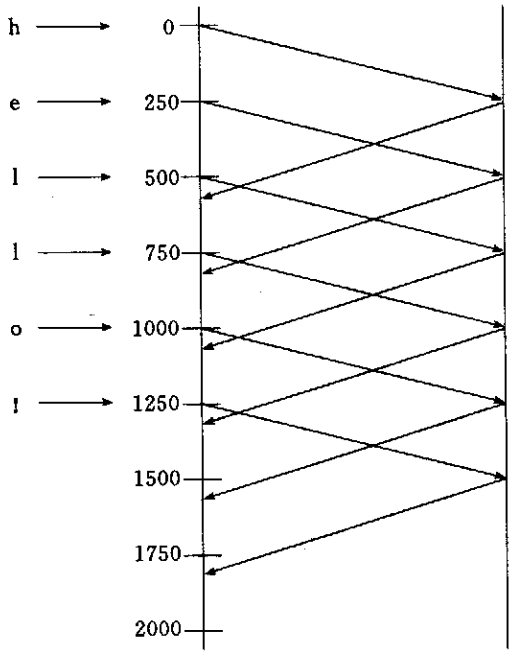


图 7.13 禁止 Nagle 算法时由服务器回显的六个字符

Nagle 算法常常与另一个 TCP 算法联合使用：延滞 ACK(delayed ACK)算法。此算法导致在接收到数据时不立即发送 ACK，而是由 TCP 等待一小段时间(典型地是 50～200ms)才发出 ACK。所希望的是在这一小段时间中接收方有数据发回给对方，这样 ACK 可以捎带在这些数据上发回，于是就节省了一个 TCP 分节。Rlogin 和 Telnet 客户通常使用这种方法，因为服务器一般都回显客户所发的每个字符，这样客户字符的 ACK 完全可以捎带在服务器对此字符的回显上发回。

对于其他客户存在一个问题，即它们的服务器不产生 ACK 可以捎带的反向通信量。这些客户可能觉察到明显的延迟，因为客户 TCP 要直到服务器的延滞 ACK 定时器到才给服务器发数据。这些客户需要一个方法以禁止 Nagle 算法，因此使用选项 TCP_NODELAY。

另一类不适合使用 Nagle 算法和 TCP 的延滞 ACK 算法的客户是：以若干小片数据发送单个逻辑请求给服务器的客户。例如，假设客户给服务器发一个 400 字节的请求，但这是一个 4 字节的请求类型后跟一个 396 字节的请求数据。如果客户执行一个 4 字节的写，后跟一个 396 字节的写，第二个写将一直到服务器 TCP 确认了前 4 字节的写后才由客户 TCP 发出。而且，由于服务器应用要等到其余 396 字节的数据到齐后才能对此 4 字节的数据进行操作，服务器 TCP 将延迟此 4 字节数据的 ACK(也就是说，将不会有从服务器到客户的任何数据可以捎带 ACK)。有三种方法修复这类客户程序：

1. 使用 writev(13.4 节)而不是两次调用 write。writev 对 TCP 输出进行单次调用，而不是两次调用，其结果是我们的例子只用一个 TCP 分节。这是推荐的解决方案。
2. 拷贝 4 字节的数据和 396 字节的数据到一个缓冲区中，然后对此缓冲区调用一次 write。

3. 设置 TCP_NODELAY 套接口选项且继续调用 write 两次。这是最不可取的解决方案。

习题 7.8 和 7.9 继续讨论此例子。

TCP_STDURG 套接口选项

此选项是随 Posix.1g 新定义的。它影响对 TCP 紧急指针的解释(将在第 21 章中讨论带外数据时遇到)。对于 TCP 紧急指针(TCPv1 第 292～293 页)将指向哪,有两种可能的解释。缺省时,紧急指针指向使用 MSG_OOB 标志所发送字节后的数据字节,当今几乎所有实现都是这么解释的。但如果此套接口选项设为非 0,紧急指针就指向使用 MSG_OOB 标志发送的数据字节。

这个选项根本无需设置,为什么 Posix.1g 曾经定义了它也是一个疑问。

7.10　fcntl 函数

fcntl 代表"file control(文件控制)",此函数进行各种描述字控制操作。在描述此函数以及它如何影响套接口之前,我们需要看得远一点。图 7.15 对由 fcntl、ioctl 和路由套接口执行的不同操作进行了总结。

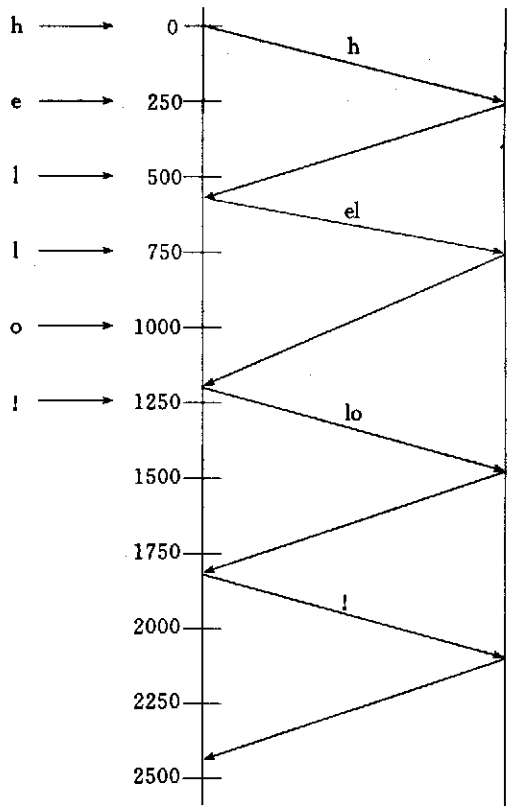


图 7.14　使能 Nagle 算法时由服务器回显的六个字符

操作	fcntl	ioctl	路由套接口	Posix.1g
设置套接口为非阻塞I/O型	F_SETFL，O_NONBLOCK	FIONBIO		fcntl
设置套接口为信号驱动I/O型	F_SETFL,O_ASYNC	FIOASYNC		fcntl
设置套接口属主	F_SETOWN	SIOCSPGRP或FIOSETOWN		fcntl
获取套接口属主	F_GETOWN	SIOCGPGRP或FIOGETOWN		fcntl
获取套接口接收缓冲区中的字节数		FIONREAD		
测试套接口是否在带外数据标志		SIOCATMARK		sockatmark
获取接口表		SIOCGIFCONF	sysctl	
接口操作		SIOC[GS]IFxxx		
ARP高速缓存操作		SIOCxARP	RTM_xxx	
路由表操作		SIOCxxxRT	RTM_xxx	

图7.15 fcntl、ioctl和路由套接口操作小结

前六个操作可由任何进程应用到套接口上，而后四个操作中的许多(接口、ARP和路由表)由诸如ifconfig和route这样的管理程序完成。在第16章中，我们将详细讨论各种ioctl操作。在第17章中，则详细讨论路由套接口的有关操作。

有很多方法来执行前四个操作，但我们注意到，最后一列中Posix.1g规定fcntl是最理想的方法。我们也注意到，Posix.1g提供函数sockatmark(21.3节)作为测试是否在带外数据标志的理想方法。其余的操作，即最后一列为空白者，还没有被Posix标准化。

> 我们还要注意，设置套接口为非阻塞I/O型和信号驱动I/O型的前两个操作，历史上是用fcntl的FNDELAY和FASYNC命令设置的。Posix定义O_xxx常值。

函数fcntl提供了下列关于网络编程的特性：

- 非阻塞I/O。 我们可以通过用F_SETFL命令设置O_NONBLOCK文件状态标志来设置套接口为非阻塞型。我们在第15章讲述非阻塞I/O。
- 信号驱动I/O。 我们可以用F_SETFL命令来设置O_ASYNC文件状态标志，这导致在套接口状态发生变化时内核生成信号SIGIO，这一点我们在第22章再做讨论。

  > 此标志是随Posix.1g新加的。

- F_SETOWN命令让我们设置套接口属主(进程ID或进程组ID)，由它来接收信号SIGIO和SIGURG。前一个信号在设置套接口为信号驱动I/O型时生成(第22章)，后一个信号在新的带外数据到达套接口时生成(第21章)。F_GETOWN命令返回套接口的当前属主。

  > 术语“套接口属主”是随Posix.1g定义的新概念。历史上源自Berkeley的实现

称之为“套接口的进程组 ID”，因为存储此 ID 的变量是结构套接口的成员 so_pgid(TCPv2 第 438 页)。

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* int arg */);
                                    返回：依赖于参数 cmd——成功，-1——出错
```

每个描述字(包括套接口描述字)都有一组由命令 F_GETFL 取得和由命令 F_SETFL 设置的文件标志。影响套接口的两个标志是：

O_NONBLOCK	非阻塞 I/O
O_ASYNC	信号驱动 I/O

后面我们将详细描述这两个特性。现在我们只需注意，用 fcntl 来使能非阻塞 I/O 的典型代码将是：

```
int     flags;
    /* set socket nonblocking */
if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
    err_sys("F_GETFL error");
flags |= O_NONBLOCK;
if (fcntl (fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

下面是你可能会遇到的，简单地设置新期望标志的代码：

```
    /* Wrong way to set socket nonblocking */
if (fcntl(fd,F_SETFL,O_NONBLOCK) <0)
    err_sys("F_SETFL error");
```

这段代码在设置非阻塞标志的同时也清除了所有其他文件状态标志。设置某个文件状态标志的唯一正确的方法是：先取得当前标志，与新标志逻辑或后再设置标志。

下面的代码关闭非阻塞标志，假设标志是由上面所示的 fcntl 调用来设置的

```
flags &= ~O_NONBLOCK;
if (fcntl(fd,F_SETFL,flags) < 0)
    err_sys("F_SETFL error");
```

信号 SIGIO 和 SIGURG 与其他信号的不同之处在于，这两个信号仅在已用命令 F_SETOWN 给套接口指派了属主后才会生成。F_SETOWN 命令的整参数 arg 既可以是一个正整数，指明接收信号的进程 ID，也可以是一个负整数，它的绝对值是接收信号的进程组 ID。F_GETOWN 命令返回套接口属主作为函数 fcntl 的返回值，可以是进程 ID(一个正的返回值)，也可以是进程组 ID(一个除－1 外的负值)。指定进程或进程组来接收信号的区别是：前者只引起一个进程接收信号，而后者则引起进程组中的所有进程(也许多于 1 个进程)接收信号。

SVR4 只允许将套接口的属主设置为进程 ID 而不能是进程组 ID。

当一个新的套接口由函数 socket 创建时，它没有属主，但是当一个新的套接口从一个监

听套接口创建时，套接口属主便由已连接套接口从监听套接口继承而来(许多套接口选项也这样，见 TCPv2 第 462～463 页)。

7.11 小 结

套接口选项从非常普通(SO_ERROR)到非常专门(IP 头部选项)都有。我们可能遇到的、最常用的选项是：SO_KEEPALIVE、SO_RCVBUF、SO_SNDBUF 和 SO_REUSEADDR。最后那个选项应总是在调用 bind 之前给 TCP 服务器设置(图 11.8)。SO_BROADCAST 选项和 10 个多播套接口选项分别用于使用广播或多播的应用程序。

SO_KEEPALIVE 套接口选项由许多 TCP 服务器设置以自动终止一个半开连接。此选项一个较好的特性是它由 TCP 层处理，不要求有一个应用级的无动静计时器，但它的缺点是不能区别客户崩溃与到客户的连通性暂时丢失。

SO_LINGER 套接口选项使我们能更好控制函数 close 返回的时机，让我们可以强制发送一个 RST 而不是 TCP 的四分组连接终止序列。我们必须小心发送 RST，因为这避免了 TCP 的 TIME_WAIT 状态。也有本套接口选项无法提供我们所需信息的情况，此时就要求有应用级的 ACK。

每个 TCP 套接口都有一个发送缓冲区和一个接收缓冲区，每个 UDP 套接口都有一个接收缓冲区。SO_SNDBUF 和 SO_RCVBUF 套接口选项让我们可以改变这些缓冲区的大小。这些选项最常见的用途是用于长胖管道上的批量数据传送。长胖管道是或高带宽或长延时的 TCP 连接，常常使用 RFC 1323 的扩展定义。另一方面，UDP 套接口可能想增加接收缓冲区的大小以允许内核在应用进程较忙时排队更多的数据报。

7.12 习 题

7.1 写一个输出缺省 TCP 和 UDP 发送和接收缓冲区大小的程序，并在你有访问权限的系统上运行该程序。

7.2 将图 1.5 做如下修改：在调用 connect 之前，调用 getsockopt 得到套接口接收缓冲区大小和 MSS，输出这两个值。connect 返回成功后，仍取这两个套接口选项并输出其值，值变化了吗？为什么？运行一个连接到本地网络上的本服务器程序实例，再运行一个连接到远程网络上的本服务器程序实例，MSS 变化吗？为什么？你应在你有访问权的任何不同主机上运行本程序。

7.3 从图 5.2 及图 5.3 的 TCP 服务器程序和图 5.4 及图 5.5 的 TCP 客户程序开始，修改客户程序的 main 函数，以在调用 exit、设置 l_onoff 为 1 和 l_linger 为 0 之前设置 SO_LINGER 套接口选项。先启动服务器，然后启动客户，在客户上键入一行或两行来检验操作，然后键入文件结束符以终止客户，将发生什么情况？在你终止客户后，在客户主机上运行 netstat，看看套接口是否经历了 TIME_WAIT 状态。

7.4 假设两 TCP 客户在同一时间启动，都设置 SO_REUSEADDR 套接口选项，且以相同的本地 IP 地址和相同的端口号(譬如说，1500)调用 bind，但一个客户连接到

198.69.10.2 的端口 7000，另一个客户连接到 198.69.10.2(相同的 IP 地址)的端口 8000。阐述所出现的竞争状态。

7.5　获取本书中例子的源代码(见前言)并编译程序 sock(C.3 节)。首先，将你的主机分类为(1)没有多播支持，(2)有多播支持但不提供 SO_REUSEPORT，(3)有多播支持且提供 SO_REUSEPORT。试着在同一端口上启动程序 sock 的多个实例作为 TCP 服务器(-s 命令行选项)，分别捆绑通配地址、你的主机的某个接口地址以及回馈地址。你需要指定 SO_REUSEADDR 选项(-A 命令行选项)吗？用命令 netstat 来观察监听套接口。

7.6　继续前面的例子，但启动的是 UDP 服务器(-u 命令行选项)并试着启动两个实例，捆绑相同的本地 IP 地址和端口号。如果你的实现支持 SO_REUSEPORT，试着用它(-T 命令行选项)。

7.7　程序 Ping 的许多版本有一个-d 标志来打开 SO_DEBUG 套接口选项，这是干什么用的？

7.8　继续我们在讨论 TCP_NODELAY 套接口选项结尾处的例子，假设客户执行了两个写：第一个 4 字节，第二个 396 字节，还假设服务器的延滞 ACK 时间为 100ms，客户与服务器间的 RTT 是 100ms，服务器处理客户请求的时间是 50ms。画一个时间线图以表示延滞 ACK 与 Nagle 算法的相互作用。

7.9　假设设置了 TCP_NODELAY 套接口选项，重做上个习题。

7.10　假设进程调用 writev 一次性处理完 4 字节缓冲区和 396 字节缓冲区，重做习题 7.8。

7.11　读 RFC 1122[Barden 1989]以确定延滞 ACK 的建议间隔。

7.12　图 5.2 及图 5.3 中的服务器程序什么地方耗时最多？如果服务器设置了 SO_KEEPALIVE 套接口选项，没有数据在连接上交换，客户主机崩溃且没有重启，将发生什么？

7.13　图 5.4 及图 5.5 中的客户程序什么地方耗时最多？如果客户设置了 SO_KEEPALIVE 套接口选项，没有数据在连接上交换，服务器主机崩溃且没有重启，将发生什么？

7.14　图 5.4 及图 6.13 中的客户程序什么地方耗时最多？如果客户设置了 SO_KEEPALIVE 套接口选项。没有数据在连接上交换，服务器主机崩溃且没有重启，将发生什么？

7.15　假设客户和服务器都设置了 SO_KEEPALIVE 套接口选项。通信双方维护连通性但没有数据在连接上交换。当保持存活定时器每 2 小时的定时时间到时，在连接上有多少 TCP 分节被交换？

7.16　几乎所有实现都在头文件<sys/socket.h>中定义了常值 SO_ACCEPTCON，但我们没有描述此选项。读[Lanciani 1996]，弄清此选项为什么存在。

第8章 基本UDP套接口编程

8.1 概 述

在使用TCP与使用UDP的应用程序之间存在本质差异,这是因为两个传输层很不相同:UDP是无连接的、不可靠的数据报协议,而TCP是面向连接的,提供可靠的字节流。然而,有些实例却更适合用UDP而不是TCP,我们到20.4节再讲述这个设计选择。有些流行的应用程序是用UDP实现的:DNS(域名系统)、NFS(网络文件系统)和SNMP(简单网络管理协议)就是这样的例子。

图8.1给出了典型的UDP客户-服务器程序函数调用。客户不与服务器建立连接,它只管用函数sendto给服务器发送数据报(sendto在下一节介绍),此函数要求目的地址(服务器)作为其参数。类似地,服务器不从客户接收连接,它只管调用函数recvfrom,等待来自某客户的数据到达。与数据报一起,recvfrom返回客户的协议地址,所以服务器可以发送响应给正确的客户。

图8.1所示为典型情况下发生的UDP客户-服务器交互的时间线图,我们可将此图与图4.1中典型的TCP交互进行比较。

本章我们介绍用于UDP套接口的新函数:recvfrom和sendto,并用UDP重写我们的回射客户-服务器程序。我们也介绍了函数connect在UDP套接口中的使用和异步错误的概念。

8.2 recvfrom和sendto函数

这两个函数类似于标准的read和write函数,但要求有三个附加参数。

```
#include <sys/socket.h>
ssize_t recvfrom (int sockfd, void *buff, size_t nbytes, int flags,
                  struct sockaddr *from, socklen_t *addrlen);
ssize_t sendto (int sockfd, const void *buff, size_t nbytes, int flags,
                const struct sockaddr *to, socklen_t addrlen);
                                    两者均返回:读写字节数——成功,-1——出错
```

前三个参数:sockfd、buff和nbytes等同于函数read和write的前三个参数:描述字、指向读入或写出缓冲区的指针和读写字节数。

第13章中当我们讨论函数recv、send、recvmsg和sendmsg时,我们再介绍参数flags,因为在我们简单的UDP客户-服务器程序例子中还不需要它们。现在我们总是将flags设置为0。

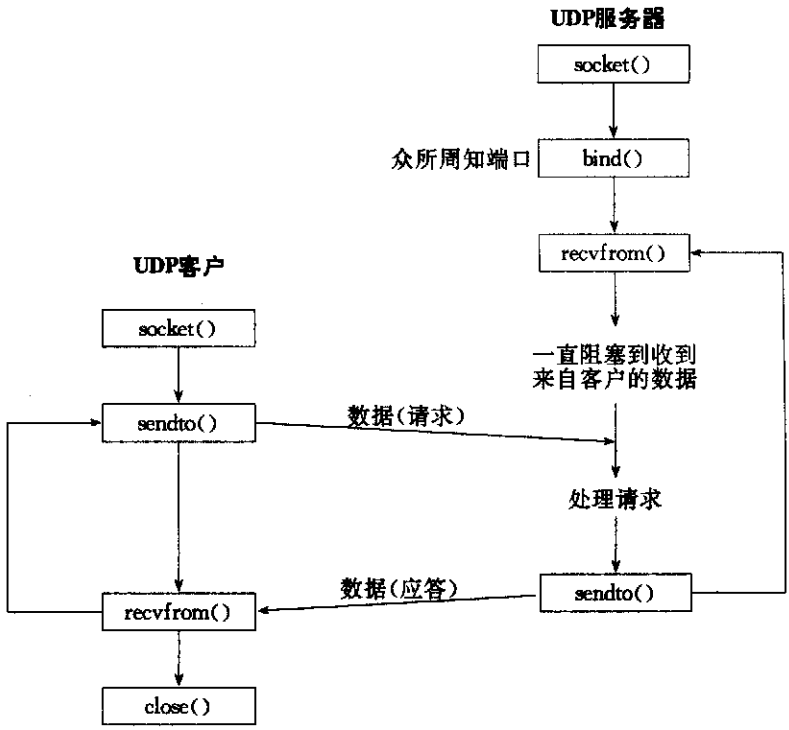


图 8.1　UDP 客户-服务器程序所用套接口函数

函数 sendto 的参数 to 是一个含有数据将发往的协议地址(例如,IP 地址和端口号)的套接口地址结构,它的大小由 addrlen 来指定。函数 recvfrom 用数据报发送者的协议地址装填由 from 所指的套接口地址结构,存储在此套接口地址结构中的字节数也以 addrlen 所指的整数返回给调用者。注意,sendto 的最后一个参数是一整数值,而 recvfrom 的最后一个参数是一指向整数值的指针(值-结果参数)。

recvfrom 的最后两个参数类似于 accept 的最后两个参数:返回时套接口地址结构的内容告诉我们是谁发送了数据报(UDP 情况下)或是谁发起了连接(TCP 情况下)。sendto 的最后两个参数类似于 connect 的最后两个参数:我们用数据报将发往(UDP 情况下)或与之建立连接(TCP 情况下)的协议地址来装填套接口地址结构。

两个函数都返回读写数据的长度作为函数值。在 recvfrom 的典型应用中,对于数据报协议,返回值是所接收的数据报中用户数据的总量。

写一个长度为 0 的数据报是可行的,在 UDP 情况下,这导致一个包含 IP 头部(一般说来,IPv4 的头部为 20 字节,IPv6 的头部为 40 字节)、8 字节 UDP 头部和没有数据的 IP 数据报。这也意味着对于数据报协议,recvfrom 返回 0 值也是可行的:它不表示对方已关闭了连接,这与 TCP 套接口上的 read 返回 0 的情况不同。由于 UDP 是无连接的,这就没有诸如关闭 UDP 连接之类的事情。

如果 recvfrom 的参数 from 是空指针,则相应的长度参数(addrlen)也必须是空指针,这表示我们并不关心发数据方的协议地址。

recvfrom 和 sendto 可用于 TCP,尽管一般来说没有理由这样做。

如13.9节所述，T/TCP即事务TCP(TCP for Transactions)就使用sendto。

8.3 UDP回射服务器程序：main函数

现在，我们用UDP重新编写第5章中简单的回射客户-服务器程序。我们的UDP客户和服务器程序依循图8.1中所示的函数调用流程，图8.2描述了所使用的函数，图8.3给出了服务器程序main函数。

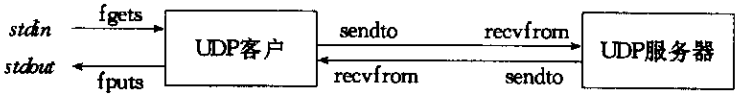


图8.2 使用UDP的简单回射客户-服务器

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int         sockfd;
6     struct sockaddr_in servaddr,cliaddr;
7     sockfd = Socket(AF_INET,SOCK_DGRAM,0);
8     bzero(&servaddr,sizeof(servaddr));
9     servaddr.sin_family = AF_INET;
10    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
11    servaddr.sin_port = htons(SERV_PORT);
12    Bind(sockfd,(SA * ) &servaddr,sizeof(servaddr));
13    dg_echo(sockfd,(SA * ) &cliaddr,sizeof(cliaddr));
14 }
```

图8.3 UDP回射服务器程序[udpcliserv/ucpserv01.c]

创建UDP套接口，捆绑服务器的众所周知端口

第7～12行 通过指定函数socket的第二个参数为SOCK_DGRAM(IPv4协议中的数据报套接口)，我们创建一个UDP套接口。正如TCP服务器程序的例子，bind的IPv4地址被指定为INADDR_ANY，且服务器的众所周知端口是头文件<unp.h>中的常值SERV_PORT。

第13行 接着，调用函数dg_echo来进行服务器的处理。

8.4 UDP回射服务器程序：dg_echo函数

图8.4给出了函数dg_echo。

```
1 #include    "unp.h"
2 void
3 dg_echo(int sockfd,SA * pcliaddr,socklen_t clilen)
4 {
```

```
 5    int        n;
 6    socklen_t len;
 7    char       mesg[MAXLINE];
 8    for ( ; ; ) {
 9        len = clilen;
10        n = Recvfrom(sockfd,mesg,MAXLINE,0,pcliaddr,&len);
11        Sendto(sockfd,mesg,n,0,pcliaddr,len);
12    }
13 }
```

图 8.4　函数 dg_echo：在数据报套接口上回射行[lib/dg_echo.c]

读数据报并回射给发送者

第 8～12 行　此函数是一个简单的循环，它用 recvfrom 读下一个到达服务器端口的数据报，并用 sendto 将它发回。

尽管这个函数很简单，但也有许多细节问题需要考虑。首先，此函数从不终止，因为 UDP 是一个无连接协议，它没有像 TCP 中文件结束符之类的东西。

其次，该函数提供一个迭代服务器(iterative server)，而不是像 TCP 一样提供了一个并发服务器。没有对 fork 的调用，所以单一服务器进程就处理了所有客户。一般来说，大多数 TCP 服务器是并发的，而大多数 UDP 服务器是迭代的。

此套接口在 UDP 层中有隐含的排队动作。实际上，每个 UDP 套接口都有一个接收缓冲区，到达此套接口的每个数据报都进入此套接口接收缓冲区。当进程调用 recvfrom 时，缓冲区中的下一个数据报以 FIFO(先入先出)顺序返回给进程。这样，在进程可读套接口中排好队的数据报之前，如果有多个数据报到达套接口，那么到达的数据报仅仅加到套接口接收缓冲区中。但这个缓冲区的大小有一个限制，我们在 7.5 节中已经随 SO_RCVBUF 套接口选项一起讨论了这个大小及如何增大它。

图 8.5 总结了第 5 章中的 TCP 客户-服务器在两个客户与服务器建立连接时的情形。

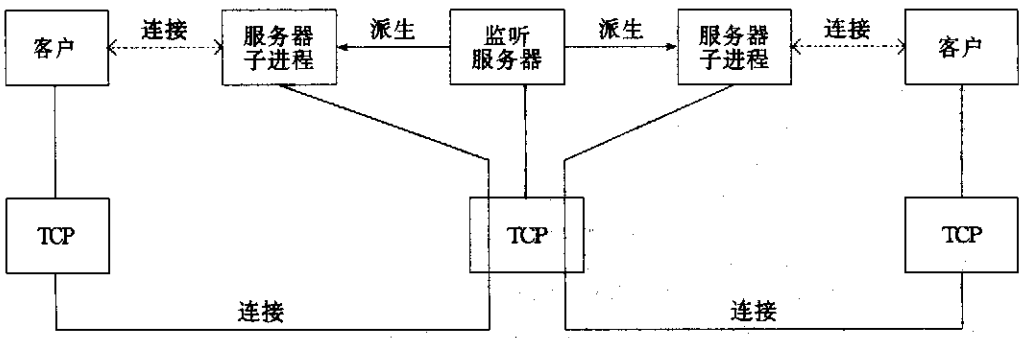


图 8.5　两个客户的 TCP 客户-服务器小结

有两个已连接套接口，服务器主机上的每一个已连接套接口都有自己的套接口接收缓冲区。

图 8.6 为两个客户发送数据报到 UDP 服务器的情形。

只有一个服务器进程，且它有单个套接口以接收所有到达的数据报并发回所有响应，此

套接口有一个接收缓冲区用来存放所到达的数据报。

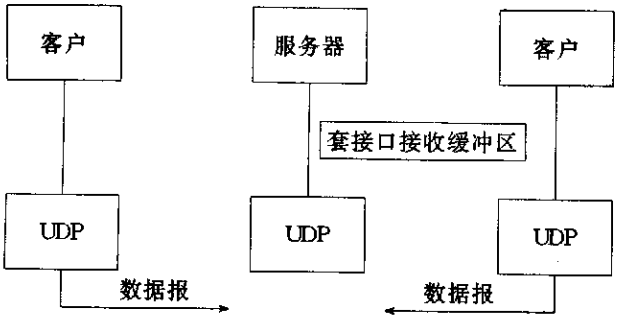


图 8.6 两个客户的 UDP 客户-服务器小结

图 8.3 中的 main 函数是协议相关的(它创建一个 AF_INET 协议的套接口,并分配、初始化一个 IPv4 套接口地址结构),但函数 dg_echo 是协议无关的。dg_echo 协议无关的理由是:调用者(在我们的例子中为 main 函数)必须分配一个正确大小的套接口地址结构,且指向此结构的指针和它的大小都应作为参数传递给 dg_echo。dg_echo 从来不深入到这样的协议相关结构中,它简单地传递一个指向该结构的指针给 recvfrom 和 sendto。recvfrom 用客户的 IP 地址和端口号装填此结构,且由于相同的指针(pcliaddr)又是作为目的地址传递给 sendto 的,这样数据报就回射给发此数据报的客户了。

8.5 UDP 回射客户程序:main 函数

UDP 客户程序 main 函数示于图 8.7。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int         sockfd;
6     struct sockaddr_i servaddr;
7     if (argc ! = 2)
8         err_quit("usage:udpcli <IPaddress>");
9     bzero(&servaddr,sizeof(servaddr));
10    servaddr.sin_family = AF_INET;
11    servaddr.sin_port = htons(SERV_PORT);
12    Inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
13    sockfd = Socket(AF_INET,SOCK_DGRAM,0);
14    dg_cli(stdin,sockfd,(SA * ) &servaddr,sizeof(servaddr));
15    exit(0);
16 }
```

图 8.7 UDP 回射客户程序[udpcliserv/udpcli01.c]

用服务器地址装填套接口地址结构

第 9～12 行　用服务器的 IP 地址和端口号来装填一个 IPv4 的套接口地址结构。此结构将传递给函数 dg_cli，指明数据报将发往何处。

第 13～14 行　创建一个 UDP 套接口，然后调用 dg_cli。

8.6　UDP 回射客户程序：dg_cli 函数

图 8.8 所示为函数 dg_cli，大部分客户的处理都由它执行。

```
1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp,int sockfd,const SA *pservaddr,socklen_t servlen)
4 {
5     int     n;
6     char    sendline[MAXLINE],recvline[MAXLINE + 1];
7     while(Fgets(sendline,MAXLINE,fp) != NULL) {
8         Sendto(sockfd,sendline,strlen(sendline),0,pservaddr,servlen);
9         n = Recvfrom(sockfd,recvline,MAXLINE,0,NULL,NULL);
10        recvline[n] = 0;      /* null terminate */
11        Fputs(recvline,stdout);
12    }
13 }
```

图 8.8　函数 dg_cli：客户处理循环[lib/dg_cli.c]

第 7～12 行　客户处理循环中有四个步骤：用 fgets 从标准输入读一行，用 sendto 将此行发给服务器，用 recvfrom 读回服务器的回射，用 fputs 输出回射行到标准输出。

我们的客户不请求内核给套接口分配一临时端口。（对于 TCP 客户，我们说 connect 调用是分配发生的地方。）对于 UDP 套接口，如果进程首次调用 sendto 时还没有捆绑上一个本地端口，内核就在此时为套接口选择一个临时端口。跟 TCP 一样，客户可以显式地调用 bind，但很少这样做。

注意，对 recvfrom 的调用指定了空指针作为第五和第六个参数。这就通告内核，我们对谁发响应并不关心。这样做有一个风险，即无论是相同主机或不同主机上的进程，都有可能给客户的 IP 地址和端口发送数据报，这些数据报被客户所读并被认为就是服务器的应答。我们在 8.8 节中再讨论此问题。

与服务器函数 dg_echo 一样，客户函数 dg_cli 也是协议无关的，但客户 main 函数是协议相关的。main 函数分配并初始化某个协议类型的套接口地址结构，并将指向此结构的指针及它的大小传递给 dg_cli。

8.7　数据报的丢失

我们的 UDP 客户-服务器例子是不可靠的。如果一个客户数据报丢失（譬如说，被客户与服务器间的某路由器丢弃），客户将永远阻塞于 dg_cli 中对 recvfrom 的调用，等待一个永远不会到达的服务器应答。与此相似，如果客户数据报到达服务器，但服务器的应答丢失了，

客户也将永远阻塞于 recvfrom 的调用。上述问题唯一的解决方法就是给客户 recvfrom 调用设置一个超时,这一点我们在 13.2 节中再作讨论。

仅仅是为调用 recvfrom 而设置超时并不是一个完整的办法。例如,如果我们超时了,我们无法辨别超时原因是数据报没有到达服务器,还是服务器的应答没有回到客户。如果客户的请求有点像"从账户 A 往账户 B 传送固定数目的钱"(而不是我们的简单回射服务器例子),则请求丢失和应答丢失是极不相同的。在 20.5 节,我们讨论如何给 UDP 客户-服务器程序增加可靠性。

8.8 验证接收到的响应

在 8.6 节结尾我们提到,知道客户临时端口号的任何进程都可往客户发送数据报,且这些数据报会与正常的服务器应答混淆。我们解决此问题的办法是:修改图 8.8 中对 recvfrom 的调用以返回发送响应者的 IP 地址和端口号,并忽略不是来自我们的数据报所发往服务器的任何数据报。然而,这也有一些缺陷,下面就将看到。

首先,我们修改客户程序 main 函数(图 8.7)以使用标准回射服务器(图 2.13),我们用赋值语句

```
servaddr.sin_port = htons(7);
```

代替

```
servaddr.sin_port = htons(SERV_PORT);
```

这样,我们的客户就可以使用任何运行标准回射服务器的主机。

然后,我们重写函数 dg_cli 以分配另一个套接口地址结构保存从 recvfrom 返回的结构,如图 8.9 所示。

分配另一个套接口地址结构

第 9 行 我们调用 malloc 来分配另一个套接口地址结构。注意,函数 dg_cli 仍然是协议无关的,由于不关心所处理套接口地址结构的类型,我们仅将其大小用于调用 malloc。

比较返回地址

第 12~18 行 在 recvfrom 的调用中,我们通知内核返回数据报发送者的地址。我们首先比较 recvfrom 从值-结果参数中返回的长度,然后用 memcmp 比较套接口地址结构本身。

如果服务器仅在一台具有单个 IP 地址的主机上运行,则我们的新版客户工作得很好,但若服务器是多宿的,此程序就有可能失败。我们在有两个接口和两个 IP 地址的主机 bsdi 上运行此程序。

```
solaris % host bsdi
bsdi.kohala.com has address 206.62.226.35
bsdi.kohala.com has address 206.62.226.66
solaris % udpcli02 206.62.226.66
hello
reply from 206.62.226.35.7 (ignored)
goodbye
reply from 206.62.226.35.7 (ignored)
```

```
1 #include        "unp.h"
2 void
3 dg_cli(FILE  * fp,int sockfd,const SA  * pservaddr,socklen_t servlen)
4 {
5     int     n;
6     char    sendline[MAXLINE],recvline[MAXLINE + 1];
7     socklen_t len;
8     struct sockaddr  * preply_addr;
9     preply_addr = Malloc(servlen);
10    while(Fgets(sendline,MAXLINE, fp) ! = NULL) {
11        Sendto(sockfd,sendline,strlen(sendline),0,pservaddr,servlen);
12        len = servlen;
13        n = Recvfrom(sockfd,recvline,MAXLINE,0,preply_addr,&len);
14        if(len ! = servlen || memcmp(pservaddr,preply_addr,len) ! = 0) {
15            printf("reply from %s (ignored)\n";
16                Sock_ntop(preply_addr,len);
17            continue;
18        }
19        recvline[n] = 0;      /* null terminate */
20        Fputs(recvline,stdout);
21    }
22 }
```

图 8.9　验证返回的套接口地址的函数 dg_cli 版本[udpcliserv/dgcliaddr.c]

从图 1.16 我们知道，我们指定的 IP 地址不共享与客户主机相同的子网。

> 一般来说，这是允许的。大多数 IP 实现接受目的地址为本主机任一 IP 地址的数据报，而不管数据报到达的接口（TCPv2 第 217～219 页）。RFC 1122 [Braden 1989]称此为弱端系统模型(weak end system model)。如果系统要实现此 RFC 中所称的强端系统模型(strong end system model)，它将只接受到达接口与目的地址一致的数据报。

recvfrom 返回的 IP 地址(UDP 数据报的源 IP 地址)不是我们所发送数据报的目的 IP 地址。当服务器发送应答时，目的 IP 地址是 206.62.226.33。bsdi 主机内核中的路由函数选择 206.62.226.35 作为外出接口。由于服务器没有捆绑实际 IP 地址到其套接口上(服务器已捆绑通配 IP 地址到套接口上了，这可以在 bsdi 主机上运行 netstat 来验证)，因此内核要给 IP 数据报选择源地址，通常选择为外出接口的主 IP 地址(TCPv2 第 232～233 页)。还有，由于它是接口的主 IP 地址，如果我们发送数据报到接口的某个非主 IP 地址(即一个别名)，也将导致图 8.9 中的测试失败。

一个解决办法是，给定由 recvfrom 返回的 IP 地址后，由客户在 DNS(第 9 章)中查找服务器主机的名字来验证相应主机的域名而不是它的 IP 地址。另一个解决办法是，由 UDP 服务器给服务器主机上配置的每个 IP 地址创建一个套接口，捆绑相应的 IP 地址到此套接口，然后在所有这些套接口上使用 select(等待其中任一个成为可读)，再从可读的套接口作应答。由于用于应答的套接口捆绑了客户请求的目的 IP 地址(否则该数据报不会投送到此套接口上)，这就保证应答的源地址与请求的目的地址相同。我们在 19.11 节和 20.6 节将给出这样的例子。

在一个多宿的Solaris系统上，服务器应答的源IP地址是客户请求的目的IP地址。本节所描述的情况是针对源自Berkeley的实现的，它基于外出接口选择源IP地址。不过在Solaris 2.5.1上看来，服务器应答的源IP地址是外出接口的最后一个已配置IP地址。

8.9 服务器进程未运行

我们下一个要检查的情况是在不启动服务器的情况下启动客户，看看这将会怎样。我们在客户上键入一行，什么也看不到，客户永远阻塞于它对recvfrom的调用，等待一个永不到来的服务器应答。但这是一个很好的例子，它要求我们对底层协议有更多的了解以理解网络应用进程发生了什么事情。

首先，我们在主机bsdi上启动tcpdump，然后在相同主机上启动客户，指定主机solaris为服务器主机。接着，我们键入一行，但此行没有被回射。

```
bsdi % udpcli01 206.62.226.33
hello,world          我们键入这一行
                     但没有任何东西回射回来
```

图8.10给出了tcpdump的输出。

```
1  0.0                  arp who-has solaris tell bsdi
2  0.002526 (0.0025)    arp reply solaris is-at 8:0:20:78:e3:e3
3  0.002932 (0.0004)    bsdi.1105 > solaris.9877: udp 13
4  0.006932 (0.0040)    solaris > bsdi: icmp:solaris udp port 9877 unreachable
```

图8.10 当服务器主机上未启动服务器进程时tcpdump的输出

首先我们注意到，在客户主机可以往服务器主机发UDP数据报之前，要求有ARP请求和应答。（我们将此交换放在输出中是为了强调在IP数据报可发往本地网络上另一台主机或路由器之前，ARP请求-应答的潜在可能。）

在第三行，我们看到数据报发出，但服务器主机在第四行响应以“端口不可达”ICMP消息。（长度13是12个字符加回车符。）然而，此ICMP错误不返回给客户进程，原因我们不久将讨论到。客户永远阻塞于图8.8中对recvfrom的调用。我们也注意到ICMPv6有“端口不可达”错误类型，这类似于ICMPv4（见图A.15和图A.16），因此，这里讨论的结果IPv6也类似。

我们将此ICMP错误称为异步错误（asynchronous error）。该错误由sendto引起，但sendto本身却返回成功。回忆一下，在2.9节中说过，从UDP输出操作返回成功仅仅表示在接口输出队列上为IP数据报留出空间。ICMP错误直到很晚才返回（在图8.10是4ms以后）。这就是为什么称其为异步的原因。

一个基本规则是：除非套接口已连接，否则异步错误是不给UDP套接口返回的。在8.11节中，我们讨论如何给UDP套接口调用connect。为什么套接口最初实现时做此设计决策是很难明白的。（实现内涵在TCPv2第748～749页讨论。）考虑UDP客户在单个UDP套接口上连续发送三个数据报给三个不同的服务器（即三个不同的IP地址），然后进入一个调用recvfrom的循环以读应答。有两个数据报被正确递送（也就是说，三个主机中有两个主机

在运行服务器),但第三个主机没有运行服务器,这第三个主机就以一个 ICMP"端口不可达"错误来响应。此 ICMP 出错消息包含有引起错误的数据报的 IP 头部和 UDP 头部。(ICMPv4 和 ICMPv6 出错消息总是包含 IP 头部和所有的 UDP 头部或部分 TCP 头部,以便其接收者确定是哪个套接口引起的错误,如图 25.20 和图 25.21 所示。)发送三个数据报的客户需要知道引起错误的数据报的目的地址以区分是三个数据报中的哪一个引起了错误。但内核如何将此信息返回给进程呢? recvfrom 可以返回的仅有信息是 errno 值,它没有办法返回错误数据报的目的 IP 地址和目的 UDP 端口号。因此做出决定:仅在进程已将 UDP 套接口连接到确切的对方后,这些异步错误才返回给进程。

只要 SO_BSDCOMPAT 套接口选项没有打开,Linux 甚至对未连接的套接口都返回大多数 ICMP 目的地不可达错误。除了代码 0、1、4、5、11 和 12,图 A.15 中的所有目的地不可达错误均被返回。

XTI 接口提供了将这个附加信息返回给进程的方法:t_rcvuderr 可以返回此信息(31.4 节)。遗憾的是,许多 XTI 实现不返回此信息。

在 25.7 节,我们回到 UDP 套接口异步错误的这个问题,并给出了一个使用我们自己的守护进程获取未连接套接口上这些错误的简便方法。

8.10　UDP 程序例子小结

图 8.11 以圆点的形式给出了在客户发送 UDP 数据报时必须指定或选择的四个值。

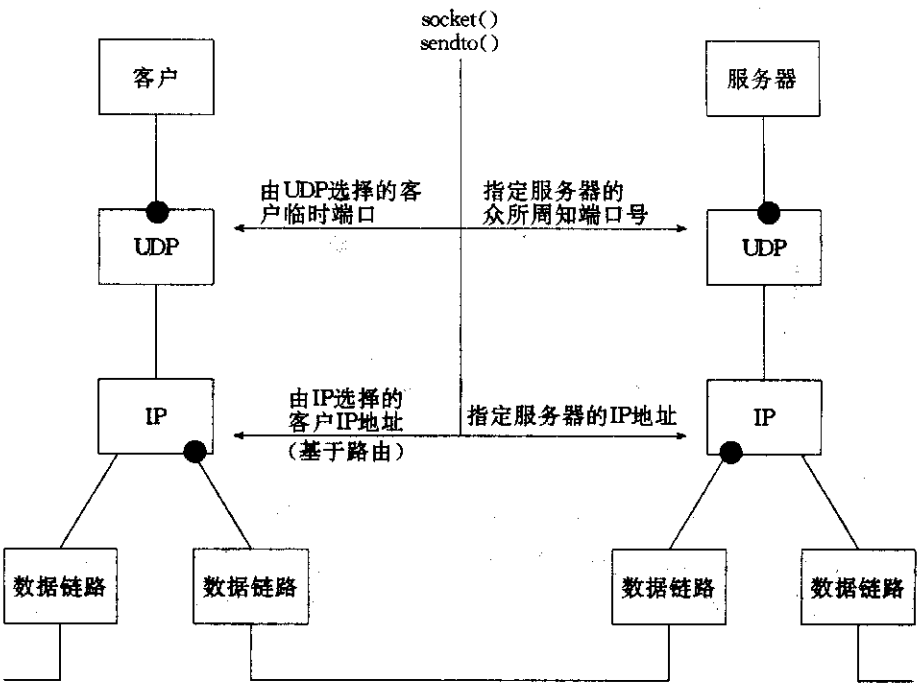


图 8.11　从客户角度小结 UDP 客户-服务器

客户必须给 sendto 调用指明服务器的 IP 地址和端口号。虽然我们提到过，如果愿意，客户可以调用 bind，但一般来说，客户的 IP 地址和端口号都是由内核自动选择的。我们也提到过，如果客户的这两个值由内核选择，客户的临时端口是在第一次调用 sendto 时一次性选定的，不能改变。然而，假设客户不捆绑特定的 IP 地址到套接口上，客户的 IP 地址是可以随客户发送的每个 UDP 数据报修改的。原因如图 8.11：如果客户主机是多宿的，客户将在两个目的地中选一个，一个由左边的数据链路外出，另一个由右边的数据链路外出。在此最坏情况下，客户的 IP 地址由于是由内核基于外出数据链路选择的，它可以随每个数据报而改变。

如果客户捆绑了一个 IP 地址到其套接口上，但内核决定外出数据报必须从另一个数据链路发出，将会怎样？在这种情况下，IP 数据报将包含一个不同于外出链路 IP 地址的源 IP 地址(见习题8.6)。

图 8.12 给出了同样的四个值，但是是从服务器的角度出发的。

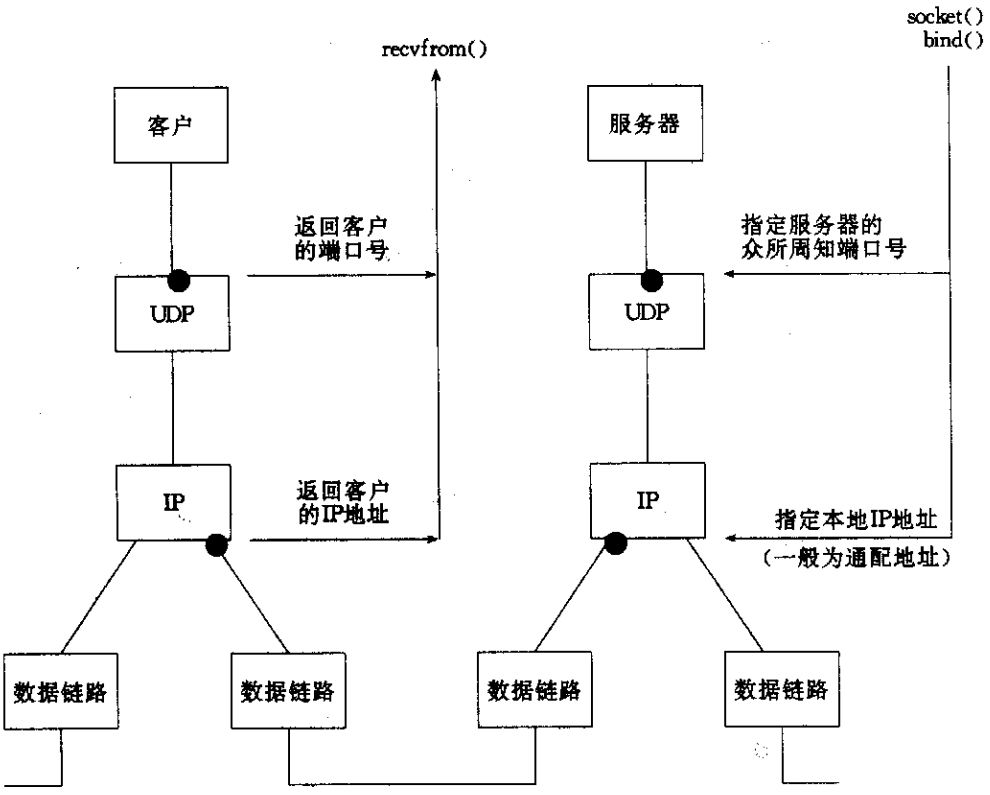


图 8.12 从服务器角度小结 UDP 客户-服务器

服务器可能想从到达的 IP 数据报上取得四条信息：源 IP 地址、目的 IP 地址、源端口号和目的端口号。图 8.13 给出了从 TCP 服务器或 UDP 服务器返回这些信息的函数调用。

来自客户的 IP 数据报	TCP 服务器	UDP 服务器
源 IP 地址	accept	recvfrom
源端口号	accept	recvfrom
目的 IP 地址	getsockname	recvmsg
目的端口号	getsockname	getsockname

图 8.13　服务器可从到达的 IP 数据报中获取的信息

TCP 服务器总是能便捷地访问已连接套接口的所有这四条信息，而且这四个值在连接的整个生命期内保持不变。然而对于 UDP 套接口，目的 IP 地址只能由为 IPv4 设置 IP_RECVDSTADDR 套接口选项或为 IPv6 设置 IPV6_PKTINFO 套接口选项，然后调用 recvmsg 而不是 recvfrom 来得到。由于 UDP 是无连接的，因此目的 IP 地址可随发送到服务器的每个数据报而改变。UDP 服务器也可接收目的地址为服务器主机的某个的广播地址或多播地址的数据报，这我们在第 18 章和第 19 章再做讨论。我们在 20.2 节讨论函数 recvmsg 以后，展示如何确定 UDP 数据报的目的地址。

8.11　UDP 的 connect 函数

在 8.9 节结尾我们提到，除非套接口已连接，否则异步错误是不会返回到 UDP 套接口的。实际上，我们可以给 UDP 套接口调用 connect(4.3 节)，但这样做的结果却与 TCP 连接毫不相同：没有三路握手过程。内核只是记录对方的 IP 地址和端口号，它们包含在传递给 connect 的套接口地址结构中，并立即返回给调用进程。

> 给 connect 函数重载(overload)UDP 套接口的这种能力容易让人混淆。如果使用约定：sockname 是本地协议地址，而 peername 是远程协议地址，则更好的名字可能是 setpeername。类似地，函数 bind 的更好名字也许是 setsockname。

有了这个能力后，我们必须区分：

- 未连接 UDP 套接口(unconnected UDP socket)，当我们创建 UDP 套接口时，缺省为此；
- 已连接 UDP 套接口(connected UDP socket)，对 UDP 套接口调用 connect 的结果。

对于已连接 UDP 套接口，与缺省的未连接 UDP 套接口相比，发生了三个变化：

1. 我们再也不能给输出操作指定目的 IP 地址和端口号。也就是说，我们不使用 sendto，而使用 write 或 send。写到已连接 UDP 套接口上的任何东西都自动发送到由 connect 所指定的协议地址(例如 IP 地址和端口号)。

   > 与 TCP 相似，我们可以给已连接 UDP 套接口调用 sendto，但不能指定目的地址。sendto 的第五个参数(指向套接口地址结构的指针)必须为空指针，第六个参数(套接口地址结构的大小)应为 0。Posix.1g 规定，当第五个参数是空指针时，就不再考虑第六个参数的取值情况。

2. 我们不用 recvfrom 而用 read 或 recv。在已连接 UDP 套接口上由内核为输入操作返

回的唯一数据报是那些来自 connect 所指定协议地址的数据报。目的地址为已连接 UDP 套接口的本地协议地址(例如 IP 地址和端口号),但不是从 connect 所指定套接口协议地址到达的数据报,不传递给已连接套接口。这就限制了已连接 UDP 套接口能且只能与一个对方交换数据报。

严格地说,一个已连接 UDP 套接口仅与一个 IP 地址交换数据报,因为 connect 到多播或广播地址是可能的。

3. 异步错误由已连接 UDP 套接口返回给进程,由此推断,未连接 UDP 套接口不接收任何异步错误,这在前面我们已描述过。

图 8.14 就 4.4BSD 总结了上面列表中第一点。

套接口类型	write 或 send	不指定目的地址的 sendto	指定目的地址的 sendto
TCP 套接口	可以	可以	EISCONN
UDP 套接口,已连接	可以	可以	EISCONN
UDP 套接口,未连接	EDESTADDRREQ	EDESTADDRREQ	可以

图 8.14 TCP 和 UDP 套接口:可指定目的协议地址吗?

Posix.1g 指出,在未连接 UDP 套接口上不指定目的地址的输出操作应返回 ENOTCONN,而不是 EDESTADDRREQ。

Solaris 2.5 允许 sendto 给已连接 UDP 套接口指定目的地址,而 Posix.1g 规定应返回 EISCONN。

图 8.15 就我们给已连接 UDP 套接口所归纳的三点做了一个小结。

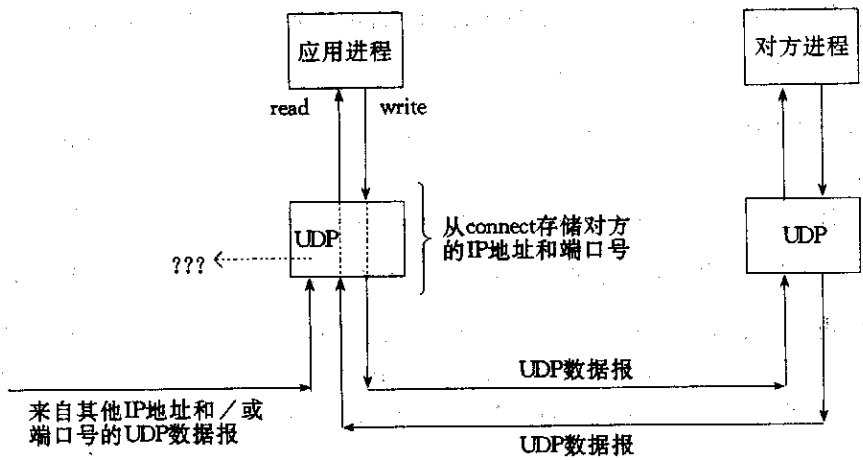


图 8.15 已连接 UDP 套接口

应用进程调用 connect,指定对方的 IP 地址和端口号,然后用 read 和 write 来与对方交换数据。

从任何其他 IP 地址或端口(图 8.15 中我们用"???"表示)到达的数据报不递送给已连接套接口,因为要么源 IP 地址要么源 UDP 端口不与所连接的套接口协议地址相匹配。这些

数据报可能递送给同一主机上的其他 UDP 套接口。如果到达的数据报没有其他匹配的套接口，UDP 将丢弃它并生成一个 ICMP 端口不可达错误。

作为小结，仅在进程用 UDP 套接口与确定的唯一对方进行通信时，UDP 客户或服务器才可以调用 connect。一般来说，都是 UDP 客户调用 connect，但也有 UDP 服务器与单个客户长时间通信的应用程序(如 TFTP)，这种情况下，客户和服务器都可调用 connect。

DNS 提供了另一个例子，如图 8.16 所示。

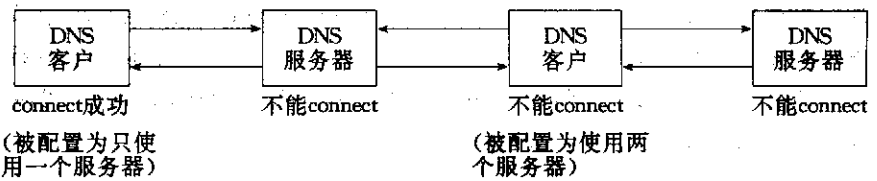


图 8.16　DNS 客户、服务器与函数 connect 的例子

通过在文件/etc/resolv.conf 中列出服务器的 IP 地址，DNS 客户可被配置为使用一个或多个服务器。如果列出单个服务器(图中最左边的方框)，客户可以调用 connect，但如果列出多个服务器(图中从右边数第二个方框)，客户就不能调用 connect。而且，DNS 服务器通常是处理任何客户的请求，所以服务器不能调用 connect。

给一个 UDP 套接口多次调用 connect

对于已连接 UDP 套接口，进程可给那个套接口再次调用 connect 以达到下面两个目的之一：

- 指定新的 IP 地址和端口号；
- 断开套接口。

第一种情况即给已连接 UDP 套接口指定新的对方，与 TCP 套接口中 connect 的使用有所不同：只可给 TCP 套接口调用一次 connect。

为了断开已 UDP 套接口连接，我们调用 connect，但设置套接口地址结构的地址族(对 IPv4 为 sin_family，对 IPv6 为 sin6_family)为 AF_UNSPEC。这可能返回一个 EAFNOSUPPORT 错误(TCPv2 第 736 页)，但没有关系。使得套接口断开连接的是在已连接 UDP 套接口上调用 connect 的进程(TCPv2 第 787～788 页)。

> connect 的 BSD 手册页面传统上说："数据报套接口可通过连接到无效地址，譬如说空地址来断开连接。"但很不幸，手册页面中根本没有定义什么是"空地址"，且没有提到会导致一个错误(但这个错误并没有关系)。Posix.1g 明确指出，地址族必须设置为 AF_UNSPEC，但然后就胡说 connect 的这种调用可以返回，也可以不返回 EAFNOSUPPORT 错误。

性能

当应用进程在未连接 UDP 套接口上调用 sendto 时，源自 Berkeley 的内核暂时连接套接口，发送数据报，然后断开套接口连接(TCPv2 第 762～763 页)。在未连接 UDP 套接口上给两个数据报调用函数 sendto 导致内核执行下列六步：

- 连接套接口；
- 输出第一个数据报；
- 断开套接口连接；
- 连接套接口；
- 输出第二个数据报；
- 断开套接口连接。

另一个考虑是搜索路由表的次数。第一个临时连接为目的IP地址搜索路由表并高速缓存此信息，第二个临时连接注意到目的地址等于已高速缓存的路由表信息的目标(我们假设两个sendto调用有相同的目的地址)，就不用再查找路由表了(TCPv2第737～738页)。

当应用进程知道自己要给同一目的地址发多个数据报时，显式连接套接口效率更高。调用connect，然后调用两次write，导致内核执行如下步骤：

- 连接套接口；
- 输出第一个数据报；
- 输出第二个数据报。

在这种情况下，内核只拷贝一次含有目的IP地址和端口号的套接口地址结构，而当调用两次sendto时，要拷贝两次。[Partridge和Pink 1993]注意到，临时连接未连接的UDP套接口将花去每个UDP传输三分之一的消耗。

8.12 dg_cli函数(修订版)

现在，我们回到图8.8中的函数dg_cli，并重写它以调用connect。图8.17所示为新函数。

```
1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp,int sockfd,const SA *pservaddr,socklen_ servlen)
4 {
5     int     n;
6     char    sendline[MAXLINE],recvline[MAXLINE + 1];
7     Connect(sockfd,(SA *) pservaddr,servlen);
8     while(Fgets(sendline,MAXLINE,fp) != NULL) {
9         Write(sockfd,sendline,strlen(sendline));
10        n = Read(sockfd,recvline,MAXLINE);
11        recvline[n] = 0;        /* null terminate */
12        Fputs(recvline,stdout);
13    }
14 }
```

图8.17 调用connect的dg_cli函数[udpcliserv/dgcliconnect.c]

所做的修改是调用 connect，并以 read 和 write 调用代替 sendto 和 recvfrom 调用。由于此函数不看传递给 connect 的套接口地址结构的内容，它仍然是协议无关的。图 8.7 中的客户程序 main 函数保持不变。

如果我们在主机 bsdi 上运行此程序，指定主机 solaris 的 IP 地址(它不在端口 9877 上运行我们的服务器)，就有下列输出：

```
bsdi % udpcli04 206.62.226.33
hello, world
read error:Connection refused
```

我们注意到的第一点是，当启动客户进程时接收不到错误，错误仅在我们发了第一个数据报给服务器后才会出现。正是发送此数据报引发了来自服务器主机的 ICMP 错误。但当 TCP 客户调用 connect，指定一个不在运行服务器进程的服务器主机时，connect 返回错误。这是因为 connect 的调用导致 TCP 三路握手的第一个分组发出，且正是这个分组引发了 TCP 服务器的 RST(4.3 节)。

图 8.18 给出了 tcpdump 的输出。

```
bsdi % tcpdump
1  0.0                  bsdi.1318 > solaris.9877: udp 13
2  0.000628 (0.0006)    solaris > bsdi: icmp: solaris udp port 9877 unreachable
```

图 8.18　当运行图 8.17 中程序时 tcpdump 的输出

在图 A.15 中，我们还看到，此 ICMP 错误由内核将其映射成 ECONNREFUSED 错误，它相应于函数 err_sys 输出的消息串："Connection refused(连接被拒绝)。"

遗憾的是，并非所有内核都能像本节的示例那样把 ICMP 消息返送给已连接的 UDP 套接口。一般来说，源自 Berkeley 的内核返回错误，而系统 V 内核则不。例如，若我们在 Solaris 2.4 主机上运行同一客户程序并连接到没有运行服务器的主机上，我们可以用 tcpdump 观察到并验证服务器主机返回了 ICMP 端口不可达错，但客户的 read 调用不返回。此缺陷在 Solaris 2.5 中已修复。UnixWare 不返回错误，而 AIX、Digital Unix、HP-UX 和 Linux 都返回错误。

8.13　UDP 缺乏流量控制

现在，我们检查无任何流量控制的 UDP 对数据报传输的影响。首先，将我们的函数 dg_cli 修改为发送固定数目的数据报，它不再从标准输入读。图 8.19 所示为新代码，此函数写 2000 个 1400 字节大小的 UDP 数据报给服务器。

```
1 #include    "unp.h"
2 #define DNG     2000        /* #datagrams to send */
3 #define DGLEN   1400        /* length of each datagram */
4 void
5 dg_cli(FILE *fp,int sockfd,const SA *pservaddr,socklen_t servlen)
6 {
7     int     i;
8     char    sendline[MAXLINE];
```

```
 9      for (i = 0; i < NDG; i++) {
10          Sendto(sockfd,sendline,DGLEN,0,pservaddr,servlen);
11      }
12  }
```

图 8.19 写固定数目的数据报到服务器的 dg_cli 函数[udpcliserv/dgcliloop1.c]

然后，我们修改服务器程序以接收数据报并对接收数目计数。此服务器不再将数据报回射给客户，图 8.20 给出了新的函数 dg_echo。当我们用终端中断键(SIGINT)终止服务器时，它输出所接收到数据报的数目并终止。

```
 1 #include    "unp.h"
 2 static void recvfrom_int(int);
 3 static int count;
 4 void
 5 dg_echo(int sockfd,SA *pcliaddr,socklen_t clilen)
 6 {
 7     socklen_t len;
 8     char      mesg[MAXLINE];
 9     Signal(SIGINT, recvfrom_int);
10     for( ; ; ) {
11         len = clilen;
12         Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
13         count++;
14     }
15 }
16 static void
17 recvfrom_int(int signo)
18 {
19     printf("\nreceived %d datagrams\n",count);
20     exit(0);
21 }
```

图 8.20 对接收到数据报进行计数的 dg_echo 函数[udpcliserv/dgecholoop1.c]

现在，在主机 bsdi 上运行服务器，它是一台慢速 80386 机器，在快得多的 Sparcstation 4 上运行客户。另外，在服务器上运行 netstat -s，在试验前后均运行，并统计输出，以告诉我们丢失了多少数据报。图 8.21 所示为服务器上的输出。

```
bsdi % netstat -s | tail
udp:    80300 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        12 dropped due to no socket
        77725 broadcast/multicast datagrams dropped due to no socket
        1970 dropped due to full socket buffers
        593 delivered
        70592 datagrams output
bsdi % udpserv06                启动我们的服务器
                                再在上面运行客户
```

```
^?                          客户运行完毕后敲入中断键
received 82 datagrams
bsdi % netstat -s | tail
udp:    82294 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        12 dropped due to no socket
        77725 broadcast/multicast datagrams dropped due to no socket
        3882 dropped due to full socket buffers
        675 delivered
        70592 datagrams output
```

图 8.21　服务器主机上的输出

客户发出 2000 个数据报，但服务器仅收到其中的 82 个，丢失率为 96%。对服务器或客户来说，并没有给出任何指示说这些数据报已丢失。这证实了我们前面说的话，UDP 没有流量控制，它是不可靠的。同时也表明，UDP 发送方比 UDP 接收方运行速度快是造成数据报丢失的一个重要原因。

如果我们看一看 netstat 的输出，服务器主机（而不是服务器本身）接收到的数据报总数是 1994（82294－80300）。有 6 个数据报未被接口接收到，可能是因为接口的缓冲区满，也可能是已被发送主机丢弃。“dropped due to full socket buffers（因套接口缓冲区满而丢弃）”计数器的值表示有多少数据报已被 UDP 接收，但因接收套接口的接收队列满而丢弃（TCPv2 第 775 页）。此值为 1912（3882－1970），它加上由应用进程输出的计数值（82）后等于主机接收到的 1994 个数据报。遗憾的是，因套接口缓冲区满而丢弃的 netstat 计数值是全系统范围的值，没有办法确定影响到具体哪些应用进程（如哪些 UDP 端口）。

> 注意，此特定主机上所有接收到的 UDP 数据报中 97%（77725÷80300）是广播或多播数据报，它们因没有任何应用进程的套接口绑定目的端口而丢弃，在第 18 章中讨论广播时，我们再接着讨论此现象。

此例中，由服务器接收的数据报的数目是不确定的，它依赖于许多因素，如网络负载、客户主机的处理负载以及服务器主机的处理负载。如果我们再运行此例子五次，则接收数据报的计数分别是 37、108、30、108 和 114。

如果我们运行相同的客户和服务器，但这一次让客户在慢速 80386 上运行，服务器在快速 Sparcstation 上运行，则没有数据报丢失。

```
solaris % udpserv06
^?                          客户运行完毕后敲入中断键
received 2000 datagrams
```

如果我们在 Solaris 下运行 netstat -s，输出格式与图 8.21 中经典的 Berkeley 输出格式有所不同，Solaris 格式模仿 SNMP 计数器。SNMP 是简单网络管理协议，在 TCPv1 第 25 章讲述。netstat 的计数器 udpInDatagrams：传递给用户进程的 UDP 数据报个数，在试验前是 139，在试验后是 2139，正好为 2000 个数据报。计数器 udpInOverflows（它不是一个正式的 SNMP 计数器）记录着因接收套接口的接收队列没有空间而丢弃的 UDP 数据报数目，它的值在试验前后均为 0，正如我们所期望的那样。

UDP套接口接收缓冲区

由UDP给特定套接口排队的UDP数据报数目受限于套接口接收缓冲区的大小。我们可以用SO_RCVBUF套接口选项改变此值，如7.5节所述。在BSD/OS下UDP套接口接收缓冲区的缺省大小为41 600字节，即29个1400字节数据报的空间。如果我们增大套接口接收缓冲区的大小，则服务器有望接收更多的数据报。图8.22所示为对图8.20中函数dg_echo的修改，它设置套接口接收缓冲区为240K字节。如果我们在80386上运行服务器，在Sparcstation 4上运行客户，做5次试验，则接收的数据报计数分别为115、168、179、145和133，这比前面用缺省套接口接收缓冲区的例子稍有改善，但也并不能从根本上解决问题。

```
1 #include    "unp.h"
2 static void recvfrom_int(int);
3 static int count;

4 void
5 dg_echo(int sockfd,SA  * pcliaddr,socklen_t clilen)
6 {
7     int     n;
8     socklen_t len;
9     char    mesg[MAXLINE];

10    Signal(SIGINT,recvfrom_int);

11    n = 240 * 1024;
12    Setsockopt(sockfd,SOL_SOCKET,SO_RCVBUF, &n, sizeof(n));

13    for ( ; ; ) {
14        len = clilen;
15        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

16        count++;
17    }
18 }

19 static void
20 recvfrom_int(int signo)
21 {
22    printf("\nreceived %datagrams\n", count);
23    exit(0);
24 }
```

图8.22 增大套接口接收队列大小的dg_echo函数[udpcliserv/dgecholoop2.c]

在图8.22中，为什么我们将接收套接口缓冲区大小设为240×1024字节？在BSD/OS 2.1中，套接口接收缓冲区的最大大小缺省为262 144字节(256×1024)，但由于缓冲区分配机制(见TCPv2第2章)的影响，真实的限制是246 723字节。许多基于4.3BSD的早期系统限制此套接口缓冲区的大小为52 000字节左右。

8.14　UDP 中的外出接口的确定

已连接 UDP 套接口还可用来确定用于特定目标的外出接口，这是由于函数 connect 被应用到 UDP 套接口时的副作用：内核选择本地 IP 地址（假设进程并没有调用 bind 以明确地指派它）。这个本地 IP 地址是通过给目的 IP 地址搜索路由表，然后使用结果接口的主 IP 地址而选定的。

图 8.23 给出了一个简单的 UDP 程序，它连接到指定的 IP 地址并调用 getsockname，输出本地 IP 地址和端口号。

```
1 #include    "unp.h"
2 int
3 main(int argc,char * * argv)
4 {
5     int     sockfd;
6     socklen_t len;
7     struct sockaddr_in cliaddr, servaddr;
8     if (argc ! = 2);
9         err_quit("usage: udpcli <IPaddress>");
10    sockfd = Socket(AF_INET,SOCK_DGRAM, 0);
11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_port = htons(SERV_PORT);
14    Inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
15    Connect(sockfd, (SA * ) &servaddr, sizeof(servaddr));
16    len = sizeof(cliaddr);
17    Getsockname(sockfd, (SA * ) &cliaddr, &len);
18    printf("local address %s\n", Sock_ntop((SA * ) &cliaddr, len));
19    exit(0);
20}
```

图 8.23　使用 connect 来确定输出接口的 UDP 程序[udpcliserv/udpcli09.c]

如果我们在多宿主机 bsdi 上运行此程序，则有下面的输出：

```
bsdi % udpcli09 206.62.226.42
local address 206.62.226.35.1331
bsdi % udpcli09 206.62.226.65
local address 206.62.226.66.1332
bsdi % udpcli09 127.0.0.1
local address 127.0.0.1.1335
```

从图 1.16 我们知道，我们运行此程序的前两次，命令行参数是在不同以太网上的 IP 地址，因此内核将本地 IP 地址指派成相应以太网接口的主 IP 地址。也就是说，.42 主机是在顶层以太网上，所以外出接口地址为.35，.65 主机是在稍低层以太网上，所以外出接口地址为.66。在 UDP 套接口上调用 connect 不给对方主机发任何信息，它完全是一个本地操作，只保存对方的 IP 地址和端口。我们还看到，在未绑定 UDP 套接口上调用 connect 也给套接

口指派一个临时端口。

遗憾的是，这项技术并非对所有实现都有效，尤其是SVR4内核。例如，它对HP-UX、Solaris 2.5和UnixWare无效，但对AIX、Digital Unix、Linux和Sloaris 2.6都有效。

8.15 使用select函数的TCP和UDP回射服务器程序

现在，我们将第5章中的并发TCP回射服务器程序与本章中的迭代UDP回射服务器程序组合为一个使用select来复用TCP和UDP套接口的单个服务器程序，图8.24是此程序的前半部分。

```
1 #include    "unp.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int       listenfd, connfd, udpfd, nreadY, maxfdp1;
6     char      mesg[MAXLINE];
7     pid_t     childpid;
8     fd_set    rset;
9     ssize_t   n;
10    socklen_t len;
11    const int on = 1;
12    struct sockaddr_in cliaddr, servaddr;
13    void      sig_chld(int);
14        /* create listening TCP socket */
15    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
16    bzero(&servaddr, sizeof(servaddr));
17    servaddr.sin_family = AF_INET;
18    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19    servaddr.sin_port = htons(SERV_PORT);
20    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21    Bind(listenfd, (SA * ) &servaddr, sizeof(servaddr));
22    Listen(listenfd, LISTENQ);
23        /* create UDP socket */
24    udpfd = Socket(AF_INET, SOCK_DGRAM, 0);
25    bzero(&servaddr, sizeof(servaddr));
26    servaddr.sin_family = AF_INET;
27    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
28    servaddr.sin_port = htons(SERV_PORT);
29    Bind(udpfd, (SA * ) &servaddr, sizeof(servaddr));
```

图8.24 使用select处理TCP和UDP的回射服务器程序前半部分[udpcliserv/udpservselect01.c]

创建监听TCP套接口

第14～22行 创建一个监听TCP套接口并捆绑服务器的众所周知端口，设置SO_

REUSEADDR 套接口选项以防此端口上已有连接存在。

创建 UDP 套接口

第 23～29 行　也创建一个 UDP 套接口并捆绑与 TCP 套接口相同的端口。这里无需在调用 bind 之前设置 SO_REUSEADDR 套接口选项，因为 TCP 端口是独立于 UDP 端口的。

图 8.25 给出了服务器程序的后半部分。

```
30    Signal(SIGCHLD, sig_chld);    /* must call waitpid() */
31    FD_ZERO(&rset);
32    maxfdp1 = max(listenfd, udpfd) + 1;
33    for ( ; ; ) {
34        FD_SET(listenfd, &rset);
35        FD_SET(udpfd, &rset);
36        if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
37            if (errno == EINTR)
38                continue;          /* back to for() */
39            else
40                err_sys("select error");
41        }
42        if      (FD_ISSET(listenfd, &rset) ) {
43                len = sizeof(cliaddr);
44                connfd = Accept(listenfd, (SA *) &cliaddr, &len);
45                if( (childpid = Fork() ) ==0)     {     /* child process */
46                    Close(listenfd);     /* close listening socket */
47                    str_echo(connfd);    /* process the request */
48                    exit(0);
49                }
50                Close(connfd);}        /* parent closes connected socket */
51        }
52        if (FD_ISSET(udpfd, &rset)) {
53            len = sizeof(cliaddr);
54            n = Recvfrom(udpfd, mesg,MAXLINE, 0, (SA *) &cliaddr, &len);
55            Sendto(udpfd, mesg, n, 0,(SA *) &cliaddr,len);
56        }
57    }
58 }
```

图 8.25　使用 select 处理 TCP 和 UDP 的回射服务器程序后半部分[udpcliserv/udpservselect01.c]

给 SIGCHLD 建立信号处理程序

第 30 行　给 SIGCHLD 建立信号处理程序，因为 TCP 连接将由子进程处理。我们已在图 5.11 中给出了这个信号处理程序。

准备调用 select

第 31～32 行　我们给 select 初始化一个描述字集，并计算出我们等待的两个描述字的较大者。

调用 select

第 34～41 行　我们调用 select 仅仅为了等待监听 TCP 套接口的可读条件或 UDP 套

接口的可读条件。由于信号处理程序 sig_chld 可以中断我们对 select 的调用，因此我们要处理 EINTR 错误。

处理新的客户连接

第 42～51 行 当监听 TCP 套接口可读时，我们 accept 一个新的客户连接，fork 一个子进程，并在子进程中调用函数 str_echo。这与第 5 章所采取的步骤相同。

处理数据报的到达

第 52～57 行 如果 UDP 套接口可读，则说明数据报已到达。我们用 recvfrom 读它并用 sendto 将其发回给客户。

8.16 小 结

将我们的 TCP 回射客户-服务器程序转换为 UDP 回射客户-服务器程序是很简单的，但 TCP 提供的许多功能也消失了：检测丢失的分组并重传，验证响应是否来自正确的对方，等等。到 20.5 节我们再回过头来讨论这个话题，并看看为了给 UDP 应用程序加上可靠性将付出什么样的代价。

UDP 套接口可能产生异步错误，也就是说，在分组发送以后的某个时刻才能得到出错的报告。TCP 总是给应用进程报告这些错误，但 UDP 套接口必须连接才能接收这些错误。

UDP 没有流量控制，这是很容易演示证明的。一般来说，这不成什么问题，因为许多 UDP 应用程序是用请求-应答模式构成的，且不用于传送大量数据。

编写 UDP 应用程序时还有许多问题需要考虑，但我们留到第 20 章才讨论，即在讨论了接口函数、广播和多播以后再作讨论。

8.17 习 题

8.1 我们有两个应用程序，一个用 TCP，另一个用 UDP。对于 TCP 套接口，接收缓冲区中有 4096 字节的数据，对于 UDP 套接口，接收缓冲区中有两个 2048 字节的数据报。TCP 应用程序调用 read，其第三个参数为 4096，UDP 应用程序调用 recvfrom，其第三个参数也为 4096。这两个应用程序有什么差别吗？

8.2 在图 8.4 中，如果我们用 clilen 来代替 sendto 的最后一个参数（它原本是 len），将会怎样？

8.3 编译并运行图 8.3 及图 8.4 的 UDP 服务器程序和图 8.7 及图 8.8 的 UDP 客户程序。验证一下客户与服务器能一起工作。

8.4 在一个窗口中运行程序 Ping，并用-i 60 选项（每 60 秒发一个分组；有些系统用-I 而不是-i），-v 选项（输出所有接收到的 ICMP 错误），并指定回馈地址（通常为 127.0.0.1）。我们将用此程序来观察由服务器主机返回的端口不可达 ICMP 错误。然后，在另一个窗口运行上个习题中的客户，指定不在运行服务器的某主机的 IP 地址。将会怎样？

8.5　对于图 8.5，我们说过每个已连接 TCP 套接口都有自己的套接口接收缓冲区。监听套接口怎样？你认为它有自己的套接口接收缓冲区吗？

8.6　用程序 sock(C.3 节)和诸如 tcpdump(C.5 节)之类的工具来测试我们在 8.10 节所做的说明：如果客户捆绑一个 IP 地址到其套接口上，但从其他接口发出一个数据报，则即使此数据报的 IP 地址与外出接口不符，IP 数据报也仍然包含绑定在套接口上的 IP 地址。

8.7　编译 8.13 节中的程序并在不同的主机上运行客户和服务器。在客户程序中每次写一个数据报到套接口处放一个 printf 调用，这改变接收到分组的百分比吗？为什么？在服务器程序中每次从套接口读一个数据报处放一个 printf 调用，这改变接收到分组的百分比吗？为什么？

8.8　对于 UDP/IPv4 套接口，可传递给 sendto 的最大长度是多少；也就是说，可装填在一个 UDP/IPv4 数据报中的最大数据量是多少？若是 UDP/IPv6，有什么不同？修改图 8.8 以发送最大长度的 UDP 数据报，读回它，并输出由 recvfrom 返回的字节数。

第9章 基本名字与地址转换

9.1 概 述

到目前为止，本书中所有例子都用数值地址来表示主机(例如，206.6.226.33)，用数值端口号来标识服务器(例如，端口13代表标准的daytime服务器，端口9877代表我们的回射服务器)。然而，我们应该使用名字而不是数值，这有很多原因：名字比较容易记；数值地址可以改变但名字保持不变；随着往IPv6上转移，数值地址变得更长，手工键入一个地址更易出错。本章讲述在名字和数值地址间进行转换的函数：gethostbyname和gethostbyaddr在主机名字与IP地址间进行转换，getservbyname和getservbyport在服务名字和端口号间进行转换。

主机名函数最近已增强以便除IPv4外还能用于IPv6，我们也将叙述这些变化。这是我们往协议无关性转移的开始，在第11章将继续讨论这个话题。实际上，第11章使用我们在本章介绍的函数，并开发了大量函数以使我们的应用程序具有协议无关性。

9.2 域名系统

域名系统DNS(Domain Name System)主要用于主机名与IP地址间的映射。主机名可以是简单名字，如solaris或bsdi，也可以是全限定域名FQDN(Fully Qualified Domain Name)，如solaris.kohala.com。

> 严格说来，FQDN也称为绝对名字(absolute name)，因此必须以一个点号结尾，但用户经常省略最后的点号。

在本节，我们仅讨论网络编程中需要的DNS基础知识，对更多细节感兴趣的读者可参看TCPv1的第14章和[Albitz and Liu 1997]。IPv6所要求的附加内容在RFC 1886[Thomson and Huitema 1995]中。

资源记录

DNS中的条目称为资源记录RR(resource record)。仅有少数几类RR会影响我们的名字与地址转换。

A　A记录将主机名映射为32位的IPv4地址。例如，这里有kohala.com域中关于主机solaris的四个DNS记录，其中第一个就是一个A记录：

```
solaris IN A      206.62.226.33
        IN AAAA   5f1b:df00:ce3e:e200:0020:0800:2078:e3e3
        IN MX     5 solaris.kohala.com.
        IN MX     10.mailhost.kohala.com.
```

AAAA　AAAA 记录(称为“四 A”记录)将主机名映射为 128 位的 IPv6 地址。选择“四 A”的称法是由于 128 位地址是 32 位地址的四倍。

PTR　PTR 记录(称为“指针记录”)将 IP 地址映射为主机名。对于 IPv4 地址,32 位地址的四个字节顺序反转,每个字节都转换成它的十进制 ASCII 值(0～255),然后附上 in-addr.arpa,结果串用于 PTR 查询。

对于 IPv6 地址,128 位地址中的 32 个 4 位组顺序反转,每组被转换成相应的十六进制 ASCII 值(0～9,a～f),并附上 ip6.int。

例如,主机 solaris 的两个 PTR 记录将为:33.226.62.206.in-addr.arpa 和 3.e.3.e.8.7.0.2.0.0.8.0.0.2.0.0.0.0.2.e.e.3.e.c.0.0.f.d.b.l.f.5.ip6.int。

MX　MX 记录指定一主机作为某主机的“邮件交换器”。在上面主机 solaris 的例子中,提供了两个 MX 记录,第一个记录有优先级值 5,第二个记录有优先级值 10,当有多个 MX 记录存在时,需按优先级值的顺序使用,从最小值开始。

> 本书中我们不用 MX 记录,但因为它们在现实世界中应用很广,所以我们也提到它们。

CNAME　CNAME 代表“canonical name(规范名字)”,其常见的用法是为常用服务如 ftp 和 www 指派一个 CNAME 记录。如果人们使用这些服务名而不是实际上的主机名,则它在服务挪到其他主机上时是透明的。例如,主机 bsdi 的 CNAME。如下:

```
ftp       IN CANME bsdi.kohala.com.
www       IN CANME bsdi.kohala.com.
mailhost  IN CANME bsdi.kohala.com.
```

IPv6 部署得太早,结果管理员应给同时支持 IPv4 和 IPv6 的主机使用怎样的命名约定也不清楚。在本节前面的例子中,我们给主机 solaris 指定了 A 和 AAAA 两个记录。有些管理员将所有 AAAA 记录放到他们自己的子域中,其名字通常为 ipv6,例如,与 AAAA 记录相关联的主机名将是 solaris.ipv6.kohala.com。有时,当这种双协议栈主机的管理员没有整个域的域名管理责任但有独立的 ipv6 子域管理责任时,也会这么干。

但是,作者将 A 记录和 AAAA 记录都放到主机的通常名字(如前面所示)下,并创建另一个名字以-4 结尾、含有 A 记录的 RR,另一个名字以-6 结尾、含有 AAAA 记录的 RR,再另一个名字以-611结尾、含有 AAAA 记录及主机的链路局部地址的 RR(有时便于调试)。我们的主机的所有这些另一个记录如下所示:

```
aix-4     IN   A      206.62.226.43
aix       IN   A      206.62.226.43
          IN   MX     5    aix.kohala.com.
          IN   MX     10   mailhost.kohala.com.
          IN   AAAA   5f1b:df00:ce3e:e200:0020:0800:5afc:2b36
aix-6     IN   AAAA   5f1b:df00:ce3e:e200:0020:0800:5afc:2b36
aix-611   IN   AAAA                    fe80:0800:5afc:2b36
```

这使我们对由某些应用程序选择的协议有更多的控制权,这将在下一章讲述。

解析器和名字服务器

组织运行一个或多个名字服务器(name server),它们通常就是所谓的 BIND(Berkeley Internet Name Domain)程序。各种应用程序,如本书中我们编写的客户和服务器程序,通过调用称为解析器(resolver)的库中的函数来与 DNS 服务器联系。最常见的解析器函数是 gethostbyname 和 gethostbyaddr,两者都在本章中介绍,前者将主机名映射为 IP 地址,后者执行相反的映射。

图 9.1 给出了应用进程、解析器和名字服务器的一个典型关系。我们编写应用程序代码。解析器代码包含在系统库中,在构造应用程序时被链接到应用程序中。应用程序代码使用正常的函数调用来调用解析器代码,最典型的就是调用函数 gethostbyname 和 gethostbyaddr。

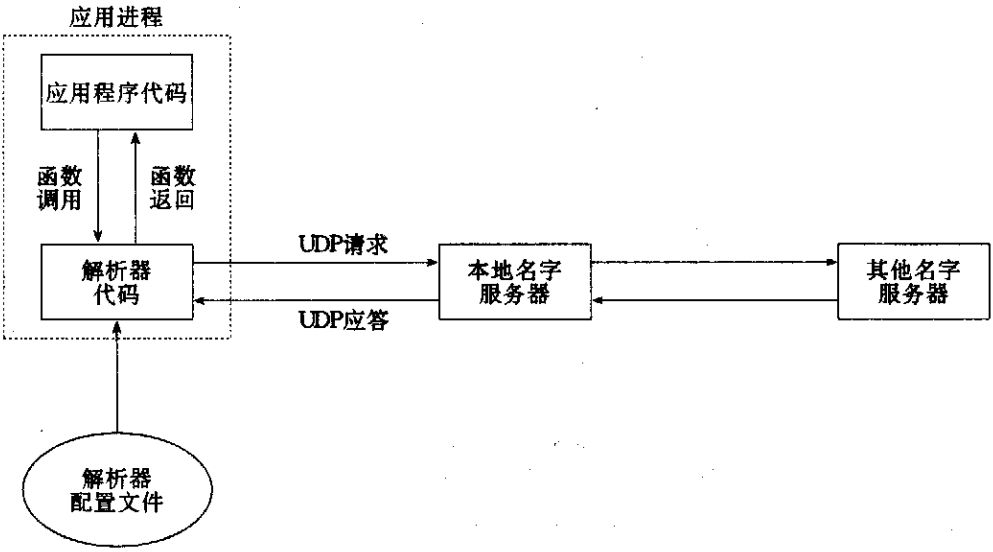


图 9.1 客户、解析器和名字服务器的典型关系

解析器代码读其依赖于系统的配置文件来确定组织的名字服务器们的所在位置(我们用复数"名字服务器们"是因为大多数组织运行多个名字服务器,虽然我们在图中只画了一个本地服务器)。文件/etc/resolv.conf 一般包含本地名字服务器的 IP 地址。

解析器用 UDP 给本地名字服务器发查询,如果本地名字服务器不知道答案,它也用 UDP 在整个因特网上给其他名字服务器发查询。

DNS 替代方法

不使用 DNS 也可得到名字和地址信息,最常用的替代方法为静态主机文件(一般为文件/etc/hosts,如图 9.9 所示)或网络信息系统 NIS(Network Information System)。不幸的是,管理员如何配置一个主机来使用不同的名字服务是依赖于不同的实现的,Solaris 2.x 和 HP-VX 10.30 用文件/etc/nsswitch.conf,Digital Unix 用文件/etc/svc.conf,AIX 用文件/etc/netsvc.conf。BIND 8.1 提供了自己的名为信息检索服务 IRS(Information Retrival Sevice)的版本,它使用文件/etc/irs.conf。如果一个名字服务器将为主机名查找所用,则所有这些系统都使用文件/etc/resolv.conf 来指定此名字服务器的 IP 地址。幸运的是,这些差异一

般都对应用程序开发人员透明，因此，我们只需调用诸如 gethostbyname 和 gethostbyaddr 这样的解析器函数即可。

9.3　gethostbyname 函数

计算机主机通常以人们可读的名字被认知。到现在为止，本书中的所有例子都有意使用 IP 地址而不是名字，所以我们确切地知道，对于诸如 connect 和 sendto 这样的函数，有什么东西进入了套接口地址结构，对于诸如 accept 和 recvfrom 这样的函数，它们返回什么。但是，大多数应用程序应该处理名字而不是地址。当我们往 IPv6 转移时，这显得尤为正确和重要，因为 IPv6 地址（十六进制数串）比 IPv4 点分十进制数串要长得多（前节中的 AAAA 记录和 PTR 记录的例子很明显地说明了这个问题）。

查找主机名最基本的函数是 gethostbyname，如果成功，它返回一个指向结构 hostent 的指针，该结构中包含了该主机的所有 IPv4 地址或 IPv6 地址。

```
#include <netdb.h>

struct hostent * gethostbyname(const char * hostname);

                返回：非空指针——成功，空指针——出错，同时设置 h_errno
```

此函数返回的非空指针指向下面的 hostent 结构：

```
struct hostent {
    char  * h_name;          /* official (canonical) name of host */
    char  * * h_aliases;     /* pointer to array of pointers to alias names */
    int     h_addrtype;      /* host address type: AF_INET or AF_INET6 */
    int     h_length;        /* length of address: 4 or 16 */
    char  * * h_addr_list;   /* ptr to array of ptrs with IPv4 or IPv6 addrs */
    };
#define h_addr h_addr_list[0]        /* first address in list */
```

按照 DNS 的说法，gethostbyname 执行一个对 A 记录的查询或对 AAAA 记录的查询。它返回 IPv4 地址或 IPv6 地址，我们在图 9.5 中总结了它返回这两类地址的条件。

> h_addr 的定义是为了向后兼容，新代码中不应再使用 h_addr。4.2BSD 没有成员 h_addr_list，而是有一个 char * h_addr，它仅指向一个 IP 地址。

图 9.2 所示为结构 hostent 和它所指向的各种信息的关系，它假设被查询的主机名有两个别名和三个 IPv4 地址。在这些字段中，正式（official）主机名和所有的别名都是以空字符（'\0'）结尾的 C 字符串。

返回的 h_name 被称为主机的规范（canonical）名字。例如给定前节所述的 CNAME 记录，主机 ftp.kohala.com 的规范名字将是 bsdi.kohala.com。而且，如果我们以主机 solaris 的某个无限定主机名如 solaris 调用 gethostbyname，则 FQDN（Solaris.kohala.com）将作为规范名字返回。

当返回 IPv6 地址时，结构 hostent 的成员 h_addrtype 被设置为 AF_INET6，成员 h_length 被设置为 16。图 9.3 给出了这些变化，它以阴影表示与图 9.2 不同的字段。

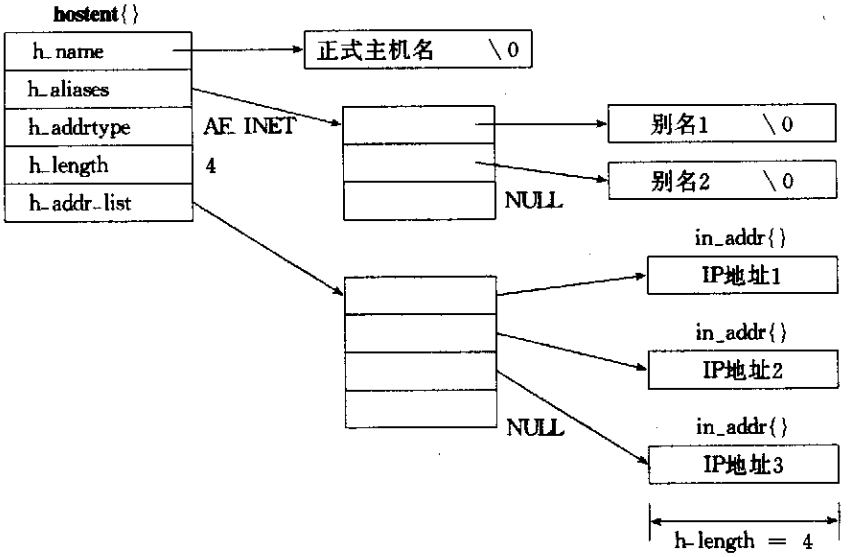


图 9.2 结构 hostent 和它所包含的信息

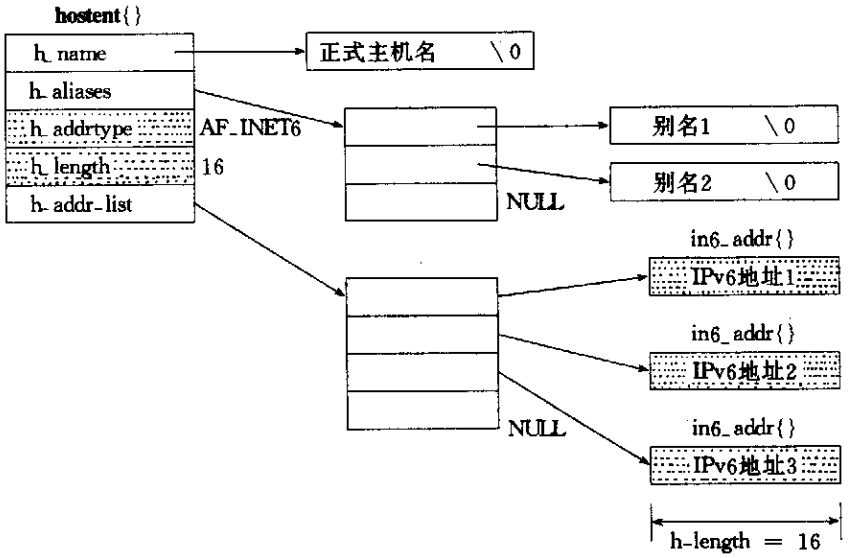


图 9.3 IPv6 地址时,结构 hostent 中所返回的不同信息

从 BIND 4.9.2 开始,新的 gethostbyname 版本允许主机名参数是点分十进制数串,也就是说,下列形式的调用是可行的:

```
hptr = gethostbyname ("206.62.226.33");
```

添加这个代码,是因为 Rlogin 客户仅接受主机名,给它调用 gethostbyname,而不接受点分十进制数串[Vixie 1996]。

gethostbyname 与我们所介绍的其他套接口函数的不同之处在于:当发生错误时,它不

设置 errno，而是将全局整数 h_errno 设置为定义在头文件<netdb.h>中的下列常值中的一个：

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_DATA(等同于 NO_ADDRESS)

错误 NO_DATA 表示指定的名字有效，但它既没有 A 记录，也没有 AAAA 记录。只有 MX 记录的主机名就是这样的例子。

BIND 的当前版本提供函数 hstrerror，它将 h_errno 的值作为唯一的参数，返回一个指向相应错误说明的 const char * 型指针。在下面的例子中，我们给出由此函数返回的一些字符串的例子。

例子

图 9.4 给出一个调用 gethostbyname 的简单例子，它可有任意数目的命令行参数，输出所有返回的信息。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     char            *ptr, **pptr;
6     char            str[INET6_ADDRSTRLEN];
7     struct hostent  *hptr;
8     while (--argc > 0) {
9         ptr = *++argv;
10        if ( (hptr = gethostbyname(ptr)) == NULL) {
11            err_msg("gethostbyname error for host: %s: %s",
12                    ptr, hstrerror(h_errno));
13            continue;
14        }
15        printf("official hostname: %s\n", hptr->h_name);
16        for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
17            printf("\talias: %s\n", *pptr);
18        switch (hptr->h_addrtype) {
19        case AF_INET:
20 #ifdef   AF_INET6
21        case AF_INET6:
22 #endif
23            pptr = hptr->h_addr_list;
24            for ( ; *pptr != NULL; pptr++)
25                printf("\taddress: %s\n",
26                    inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
27            break;
28        default:
29            err_ret("unknown address type");
30            break;
31        }
32    }
```

```
33      exit(0);
34  }
```

图 9.4 调用 gethostbyname 并输出返回信息[names/hostent.c]

第 8～14 行 给每个命令行参数调用 gethostbyname。

第 15～17 行 规范主机名被输出，后跟别名表。

第 20～22 行 为了让程序对 IPv4 地址和 IPv6 地址都支持，我们允许返回的地址类型为 AF_INET 或 AF_INET6，但只有后者被定义(也就是说，主机支持 IPv6)时，才能返回这类地址。

第 23～26 行 pptr 指向一个指针数组，数组中的每个指针都分别指向一个地址。对每个地址，我们调用 inet_ntop 并输出返回的字符串。注意，inet_ntop 基于它的第一个参数，可处理 IPv4 和 IPv6 地址。同时还要注意，我们定义 str 的长度为 INET6_ADDRSTRLEN，在 3.7 节中我们说过这是一个对最长的 IPv6 地址串都足够大的长度。在头文件<unp.h>中，我们定义了这个常值，这样即使主机不支持 IPv6，我们也总能指望它已经定义过(在代码中就省掉了又一个 #ifdef 语句)。

我们首先以主机 solaris 的名字运行此程序，它仅有一个 IPv4 地址。

```
solaris % hostent solaris
official hostname: solaris.kohala.com
        address:206.62.226.33
```

注意，正式主机名(official hostname)就是 FQDN，同时也要注意，即使此主机有 IPv6 地址，也只有 IPv4 地址被返回。下面是有多个 IPv4 地址的主机的情况：

```
solaris % hostent gemini.tuc.noao.edu
official hostname: gemini.tuc.noao.edu
        address: 140.252.1.11
        address: 140.252.3.54
        address: 140.252.4.54
        address: 140.252.8.54
```

下面是 9.2 节中介绍过的，有一个 CNAME 记录的名字：

```
solaris % hostent www
official hostname: bsdi.kohala.com
        alias:www.kohala.com
        address:206.62.226.35
```

正如所预期的那样，正式主机名与我们的命令行参数不同。

为了看到由函数 hstrerror 返回的错误串，我们首先指定一个不存在的主机名，然后指定一个仅有 MX 记录的名字。

```
solaris % hostent nosuchname
gethostbyname error for host:nosuchname:Unknown host
solaris % hostent uunet.uu.net
gethostbyname error for host:uunet.uu.net:No address associated with name
```

9.4　RES_USE_INET6 解析器选项

BIND 的较新版本(4.9.4 及以后版本)提供了一个名为 RES_USE_INET6 的解析器选项,我们可以以三种不同的方法来设置它。我们可以用此选项来通知解析器我们想让 gethostbyname 返回 IPv6 地址而不是 IPv4 地址。

1. 应用程序本身就可以设置此选项,首先调用解析器的 res_init 函数,然后打开该选项:

```
#include <resolv.h>
res_init();
_res.options |= RES_USE_INET6;
```

这必须在第一次调用 gethostbyname 或 gethostbyaddr 之前完成。此选项仅对那些设置了此选项的应用程序才有效。

2. 如果环境变量 RES_OPTIONS 含有串 inet6,则此选项打开。此选项的作用依赖于环境变量的范围。例如,如果我们在.profile 文件(假设使用 Korn Shell)中以 export 属性设置它,如:

```
export RES_OPTIONS=inet6
```

则它对从登录 shell 开始运行的每个程序都有效。但如果我们仅在命令行上设置该环境变量(我们马上会看到),则它仅对那个命令有影响。

3. 解析器配置文件(一般为/etc/resolv.conf)可以包含如下行:

```
options inet6
```

然而必须清楚,在解析器配置文件中设置此选项影响主机上调用解析器函数的所有应用程序,因此,这项技术要直到结构 hostent 中返回的 IPv6 地址可以被主机上的所有应用程序所处理时才能使用。

第一个方法以每个应用程序为基设置此选项,第二个方法以每个用户为基,第三个方法以整个系统为基。

现在,我们设置环境变量 RES_OPTIONS 为值 inet6,运行图 9.4 中的例子程序。

```
solaris % RES_OPTIONS=inet6 hostent solaris      这是具有 AAAA 记录的主机名
official hostname:solaris.kohala.com
        address:5f1b:df00:ce3e:e200:20:800:2078:e3e3
solaris % RES_OPTIONS=inet6 hostent bsdi      这是没有 AAAA 记录的主机名
official hostname:bsdi.kohala.com
        address: ::ffff:206.62.226.35
        address: ::ffff:206.62.226.66
```

第一次执行程序时,它返回主机的 IPv6 地址(回忆一下它在 9.2 节中的 AAAA 记录),第二次运行此程序时,我们指定一个没有 AAAA 记录的主机,它仍然返回 IPv6 地址:IPv4 映射的 IPv6 地址(A.5 节)。

在下两节,我们将详细讨论解析器中的 IPv6 支持。

9.5 gethostbyname2 函数与 IPv6 支持

当 IPv6 支持增加到 BIND 4.9.4 时，函数 gethostbyname2 也增加进去，它有两个参数，允许我们指定地址族。

```
#include <netdb.h>
struct hostent *gethostbyname2(const char *hostname, int family);
                    返回：非空指针——成功，空指针——出错，同时设置 h_errno
```

其返回值与 gethostbyname 的返回值相同，为一个指向结构 hostent 的指针，且此结构也保持不变。该函数的逻辑依赖于参数 family 和解析器选项 RES_USE_INET6（我们在前节结尾已提到）。

在描述具体细节之前，图 9.5 总结了对于新的选项 RES_USE_INET6，函数 gethostbyname 和 gethostbyname2 的操作：

- RES_USE_INET6 选项是打开还是关闭；
- gethostbyname2 的第二个参数是 AF_INET 还是 AF_INET6；
- 解析器是搜索 A 记录还是搜索 AAAA 记录；
- 返回地址长度为 4 还是 16。

函数 gethostbyname2 的操作如下：

- 如果参数 family 是 AF_INET，则查询 A 记录。若不成功，则返回一个空指针，若成功，则返回地址的类型及大小依赖于新的解析器选项 RES_USE_INET6：若选项未设置（缺省），则返回 IPv4 地址，结构 hostent 的成员 h_length 的值将为 4；若选项设置，则返回 IPv4 映射的 IPv6 地址，结构 hostent 的成员 h_length 的值将为 16。
- 如果参数 family 为 AF_INET6，则查询 AAAA 记录。若成功，则返回 IPv6 地址，结构 hostent 的成员 h_length 的值将为 16；否则返回一个空指针。

	RES_USE_INET6 选项	
	关闭	打开
gethostbyname (host)	搜索 A 记录，若找到，返回 IPv4 地址 (h_length = 4)，否则返回错误。这为现存的 IPv4 应用程序提供了向后兼容性	搜索 AAAA 记录，若找到，返回 IPv6 地址 (h_length = 16)，否则搜索 A 记录。若找到，返回 IPv4 映射的 IPv6 地址 (h_length = 16)，否则返回错误
gethostbyname2 (host, AF_INET)	搜索 A 记录，若找到，返回 IPv4 地址 (h_length = 4)，否则返回错误	搜索 A 记录，若找到，返回 IPv4 映射的 IPv6 地址 (h_length = 16)，否则返回错误

（续）

	RES_USE_INET6 选项	
	关闭	打开
gethostbyname2 (host,AF_INET6)	搜索 AAAA 记录，若找到，返回 IPv6 地址(h_length = 16)，否则返回错误	搜索 AAAA 记录，若找到，返回 IPv6 地址(h_length = 16)，否则返回错误

图 9.5　解析器选项 RES_USE_INET6 与函数 gethostbyname 和 gethostbyname2

如果应用程序想强制某个指定地址类型的搜索：IPv4 或 IPv6，则可用此函数。但对应用程序来说，调用 gethostbyname 似乎更常见，更何况此函数的较新版本既可返回 IPv4，也可返回 IPv6 地址。

描述函数 gethostbyname 和选项 RES_USE_INET6 的行为的一个方法是看看它的源代码，如图 9.6 所示。

```
struct hostent *
gethostbyname(const char *name)
{
    struct hostent *hp;
    if ((_res.options & RES_INIT) == 0 && res_init() == -1) {
            h_errno = NETDB_INTERNAL;
            return (NULL);
    }
    if (_res.options & RES_USE_INET6) {
        hp = gethostbyname2(name,AF_INET6);
        if (hp)
            return (hp);
    }
    return ( gethostbyname2(name,AF_INET));
}
```

图 9.6　函数 gethostbyname 和 IPv6 支持

如果解析器还没有被初始化（没有设置标志 RES_INIT），则调用 res_init。此初始化函数检查并处理环境变量 RES_OPTINOS。如果这个变量包含串 inet6 或如果解析器配置文件包含行 options inet6，则标志 RES_USE_INET6 由 res_init 设置。res_init 一般是由函数 gethostbyname 或 gethostbyaddr 在第一次被应用程序调用时自动调用的。此外，我们已展示应用程序也可调用 res_init，然后显式设置标志 RES_USE_INET6。

如果选项 RES_USE_INET6 没有设置，则函数的最后一行被执行，它就是用 AF_INET 地址族参数调用 gethostbyname2。在图 9.5 中我们看到，此调用仅搜索 A 记录，这为所有现存的应用程序提供了向后兼容性。

如果选项 RES_USE_INET6 已经设置，就用 AF_INET6 地址族参数调用 gethostbyname2 以搜索 AAAA 记录（图 9.5）。如果成功，则 gethostbyname 返回，如果失败，则继续用 AF_INET 地址族参数调用 gethostbyname2 以搜索 A 记录。若第二次搜索成功，则图 9.6 中并不明显的事情是：这些 4 字节的地址被自动映射成 16 字节的 IPv4 映射的 IPv6 地址。

总之，当选项 RES_USE_INET6 打开且应用程序调用 gethostbyname 时，应用程序通

知解析器："我只想 IPv6 地址被返回，首先搜索 AAAA 记录，如果未找到则搜索 A 记录，如果 A 记录找到则返回 IPv4 映射的 IPv6 地址。"

9.6 gethostbyaddr 函数

函数 gethostbyaddr 取一个二进制的 IP 地址并试图找到相应于此地址的主机名，这与 gethostbyname 的行为刚好相反。

此函数返回一个指向结构 hostent 的指针。这个讨论 gethostbyname 时描述过的结构中我们最感兴趣的成员是规范主机名 h_name。

```
#include <netdb.h>
struct hostent *gethostbyaddr(const char *addr, size_t len, int family);
                    返回：非空指针——成功，空指针——出错，同时设置 h_errno
```

参数 addr 不是 char * 类型，而是一个真正指向含有 IPv4 或 IPv6 地址的结构 in_addr 或 in6_addr 的指针；len 是此结构的大小：对于 IPv4 地址为 4，对于 IPv6 地址为 16；参数 family 或为 AF_INET 或为 AF_INET6。

按照 DNS 的说法，gethostbyaddr 在域 in_addr.arpa 中给 IPv4 地址在名字服务器上查询 PTR 记录，或在域 ip6.int 中给 IPv6 地址查询 PTR 记录。

函数 gethostbyaddr 和 IPv6 支持

gethostbyaddr 总有一个地址族参数，所以当加上 IPv6 支持到 BIND 时，无需发明另一个函数(类似于函数 gethostbyname2)。但是，当参数是 IPv6 地址时，仍有一些差别。下面的三个测试按步骤进行：

1. 如果 family 是 AF_INET6，len 是 16，且地址是 IPv4 映射的 IPv6 地址，则在域 in-addr.arpa 中查找地址的低 32 位(IP4 地址部分)。
2. 如果 family 是 AF_INET6，len 是 16，且地址是 IPv4 兼容的 IPv6 地址，则在域 in-addr.arpa 中查找地址的低 32 位(IPv4 地址部分)。
3. 如果被查找的是 IPv4 地址(或参数 family 为 AF_INET，或上述两种情况中的一个为真)且解析器选项 RES_USE_INET6 设置，则返回的地址(参数 addr 的一个拷贝)被转换为一个 IPv4 映射的 IPv6 地址：h_addrtype 为 AF_INET6，h_length 为 16。

第三点不太重要，因为很少有应用程序检查由 gethostbyaddr 返回的 IP 地址，它仅是参数的一个拷贝。应用程序一般调用此函数来检查所返回 hostent 结构的 h_name 成员(可能还有别名)。

9.7　uname 函数

函数 uname 返回当前主机的名字。它不是解析器库中的一部分，但我们仍在这里讨论它，因它经常与函数 gethostbyname 一起用来确定本地主机的 IP 地址。

```
#include <sys/utsname.h>
int uname(struct utsname *name);
                                        返回：非负值——成功，-1——出错
```

此函数装填结构 utsname，其地址由调用者传递：

```
#define UTS_NAMESIZE 16
#define UTS_NODESIZE 256
struct utsname {
    char sysname [_UTS_NAMESIZE];     /* name of this operating system */
    char nodename [_UTS_NODESIZE];    /* name of this node */
    char release [_UTS_NAMESIZE];     /* O.S release level */
    char version [_UTS_NAMESIZE];     /* O.S version level */
    char machine [_UTS_NAMESIZE];     /* hardware type */
};
```

不幸的是，Posix.1 所规定的只是我们所示的五个结构成员的名字以及每个数组是一个以空字符('\0')终止的字符数组这两点，对于每个数组的大小及内容则并未作任何说明。我们所给出的大小来源于 4.4BSD，其他操作系统采用不同的大小。

从网络编程的角度看，最严重的忽略是对数组 nodename 大小和内容的定义。有些系统仅在此数组中存储主机名(例如 gemini)，而另外的一些系统存储 FQDN(例如 gemini.tuc.noao.edu)。在有些操作系统如 Solaris 2.x 上，既可存放主机名，也可存放 FQDN，这取决于管理员是如何安装操作系统的。

例子：确定本地主机的 IP 地址

为了确定本地主机的 IP 地址，我们调用 uname 以得到主机名字，然后调用 gethostbyname 以得到它的所有 IP 地址。图 9.7 中所示函数 my_addrs 执行这些步骤。

```
1 #include    "unp.h"
2 #include    <sys/utsname.h>
3 char **
4 my_addrs(int *addrtype)
5 {
6     struct hostent    *hptr;
7     struct utsname    myname;
8     if (uname(&myname) < 0)
9         return(NULL);
10    if ( (hptr = gethostbyname(myname.nodename)) == NULL)
```

```
11          return(NULL);
12      *addrtype = hptr->h_addrtype;
13      return(hptr->h_addr_list);
14  }
```

图 9.7 返回主机上所有 IP 地址的函数[lib/my_addrs.c]

函数的返回值是结构 hostent 的成员 h_addr_list，即指向 IP 地址的指针数组。我们还通过指针参数返回地址族。

确定本地主机 IP 地址的另一个方法是用 ioctl 的命令 SIOCGIFCONF，这在第 16 章再作讨论。

9.8 gethostname 函数

函数 gethostname 也返回当前主机的名字。

```
#include <unistd.h>
int gethostname(char *name, size_t namelen);
                                    返回：0——成功，-1——出错
```

name 是指向主机名存储位置的指针，namelen 是此数组的大小。如果有空间，主机名以空字符结束。主机名的最大大小通常是由头文件<sys/param.h>定义的常值 MAXHOSTNAMELEN。

历史上，uname 由系统 V 定义，而 gethostbyname 由 Berkeley 定义。Posix.1 指定 uname，但 Unix 98 两者都要。

9.9 getservbyname 和 getservbyport 函数

服务器就像主机一样，也常常是由名字来标识的。如果我们在代码中，通过服务器的名字而不是通过服务器端口号来认知它，而且如果从主机到端口号的映射包含在一个文件中（一般是/etc/services），则如果端口号改变，我们所需做的所有修改就是改动文件/etc/services 中的一行，而不是重新编译应用程序。下面的函数 getservtbyname 可以根据给定名字查找服务。

```
#include <netdb.h>
struct servent *getservbyname(const char *servname, const char *protoname);
                                    返回：非空指针——成功，空指针——出错
```

此函数返回一个指向下面所示结构的指针：

```
struct servent {
    char  *s_name;      /* official service name */
    char  **s_aliases;  /* alias list */
```

```
    int     s_port;         /* port number, network-byte order */
    char *  s_proto;        /* protocol to use */
};
```

服务名 servname 必须指定，如果还指定了一个协议(即 protoname 为非空指针)，则结果表项也必须有匹配的协议。有些因特网服务是用 TCP 或 UDP 来提供的(例如 DNS 及图 2.13 中的所有服务)，而其他服务则仅支持单个协议(如 FTP 只需 TCP)。如果 protoname 没有指定且服务支持多个协议，则返回哪个端口是依赖于实现的。一般来说这没有什么关系，因为支持多个协议的服务常常使用相同的 TCP 和 UDP 端口号，但并没有保证。

结构 servent 中我们关心的主要成员是端口号。由于端口号是以网络字节序返回的，在将它存储于套接口地址结构时，绝对不能调用 htons。

对此函数的典型调用是：

```
struct servent *sptr;
sptr = getservbyname("domain", "udp");      /* DNS using UDP */
sptr = getservbyname("ftp","tcp");          /* FTP using TCP */
sptr = getservbyname("ftp",NULL);           /* FTP using TCP */
sptr = getservbyname("ftp","udp");          /* this call will fail */
```

由于 FTP 仅支持 TCP，所以第二个调用和第三个调用是相同的，第四个调用将失败。文件/etc/services 中的典型行是：

```
solaris % grep -e ftp -e domain /etc/services
ftp-data        20/tcp
ftp             21/tcp
domain          53/udp
domain          53/tcp
tftp            69/udp
```

下一个函数 getservbyport 在给定端口号和可选协议后查找相应的服务。

```
#include <netdb.h>
struct servent *getservbyport(int port, const char *protname);
                                    返回：非空指针——成功，空指针——出错
```

port 值必须为网络字节序。对此函数的典型调用是：

```
struct servent *sptr;
sptr = getservbyport(htons(53),   "udp");   /* DNS using UDP */
sptr = getservbyport(htons(21),   "tcp");   /* FTP using TCP */
sptr = getservbyport(htons(21),   NULL);    /* FTP using TCP */
sptr = getservbyport(htons(21),   "udp");   /* this call will fail */
```

对于 UDP，由于没有服务使用端口 21，所以最后一个调用将失败。

必须知道，有些端口号对于 TCP 可能用于一种服务，但对于 UDP，同样的端口号可能用于完全不同的服务。例如：

```
solaris % grep 514 /etc/services
shell           514/tcp
syslog          514/udp
```

表明端口514在TCP协议由rsh命令使用，但在UDP协议是由syslog守护进程使用。端口512～514都有这个特性。

例子：使用gethostbyname和getservbyname

现在，我们可以修改图1.5中的TCP时间/日期客户程序，让它使用gethostbyname和getservbyname，并改用两个新的命令行参数：主机名和服务名。图9.8给出了此程序，它也展示了我们所期望的行为：试着连接到多宿服务器主机的所有的IP地址，直到有一个成功或所有地址都试遍为止。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int         sockfd, n;
6     char        recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     struct in_addr **pptr;
9     struct hostent *hp;
10    struct servent *sp;
11    if (argc != 3)
12        err_quit("usage: daytimetcpcli1 <hostname> <service>");
13    if ( (hp = gethostbyname(argv[1])) == NULL)
14        err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno));
15    if ( (sp = getservbyname(argv[2], "tcp")) == NULL)
16        err_quit("getservbyname error for %s", argv[2]);
17    pptr = (struct in_addr **) hp->h_addr_list;
18    for ( ; *pptr != NULL; pptr++) {
19        sockfd = Socket(AF_INET, SOCK_STREAM, 0);
20        bzero(&servaddr, sizeof(servaddr));
21        servaddr.sin_family = AF_INET;
22        servaddr.sin_port = sp->s_port;
23        memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
24        printf("trying %s\n",
25               Sock_ntop((SA *) &servaddr, sizeof(servaddr)));
26        if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) == 0)
27            break;          /* success */
28        err_ret("connect error");
29        close(sockfd);
30    }
31    if (*pptr == NULL)
32        err_quit("unable to connect");
33    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
34        recvline[n] = 0;   /* null terminate */
35        Fputs(recvline, stdout);
36    }
37    exit(0);
38 }
```

图9.8 使用gethostbyname和getservbyname的时间/日期客户程序[names/daytimetcpcli1.c]

调用 gethostbyname 和 getservbyname

第 13～16 行　第一个命令行参数是主机名，我们将它作为参数传递给 gethostbyname；第二个命令行参数是服务名，我们将它作为参数传递给 getservbyname。假设我们的代码用 TCP，我们将它作为 getservbyname 的第二个参数。

尝试每个服务器

第 18～25 行　现在，我们将对 socket 和 connect 的调用放在同一循环中，给服务器主机的每个地址执行一次，直到连接成功或所有 IP 地址都已试完。调用 socket 以后，我们用服务器的 IP 地址和端口装填网际套接口地址结构。虽然我们可以把对 bzero 的调用和它后面的两条赋值语句放到循环体之外来提高执行效率，但图中所示的代码较易读。与服务器建立连接很少是网络客户的性能瓶颈。

调用 connect

第 26～30 行　调用 connect，如果成功，则 break 出循环；如果连接建立失败，则输出一个错误并关闭套接口。前面讲过，调用 connect 失败的描述字必须关闭，不能再被使用。

检查错误

第 31～32 行　如果循环因没有一个 connect 调用成功而终止，则程序也结束。

读服务器的应答

第 33～37 行　否则，我们读服务器的应答，当服务器关闭连接时，也终止程序。

如果我们指定正在运行时间/日期服务器的主机运行此程序，则我们得到期望的输出：

```
solaris % daytimetcpcli1 aix daytime
trying 206.62.226.35.13
Thu May 22 19:28:11 1997
```

更有意思的是指定一个不在运行时间/日期服务器的多宿路由器运行此程序：

```
solaris % daytimetcpcli1 gateway.tuc.noao.edu daytime
trying 140.252.1.4.13
connect error: Connection refused
trying 140.252.101.4.13
connect error: Connection refused
trying 140.252.102.1.13
connect error: Connection refused
trying 140.252.104.1.13
connect error: Connection refused
trying 140.252.3.6.13
connect error: Connection refused
trying 140.252.4.100.13
connect error: Connection refused
unable to connect
```

9.10　其他网络相关信息

本章我们的重点在于主机名与 IP 地址、服务名与端口号。但扩大一下我们的视野，就会发现，其实应用程序可能需查询四种类型的信息（与网络相关的）：主机、网络、协议和服务。

大多数查询都是针对主机的(gethostbyname 和 gethostbyaddr),有一小部分是针对服务的(getservbyname 和 getservbyport),针对网络和协议的查询就更少了。

所有四类信息都可存储在文件中,而且每类信息都定义有三个函数:

1. 函数 getXXXent 读文件中的下一表项,在必要时可以打开文件;
2. 函数 setXXXent 打开(如果文件还没有打开的话)并回绕文件;
3. 函数 endXXXent 关闭文件。

每类信息都定义了自己的结构,这些定义包含在文件<netdb.h>中:hostent、netent、protoent 和 servent 结构。

除了三个用于文件的顺序处理的 get、set 和 end 函数外,每类信息还提供了一些键值查询(keyed loopup)函数。它们顺序浏览文件(调用函数 getXXXent 来读每一行),但不返回每一行给调用者,而是查找一个与某参数匹配的表项。这些键值查询函数都有形如 getXXXbyYYY 的名字。例如,针对主机信息的两个关键字查询函数是 gethostbyname(查找与主机名匹配的表项)和 gethostbyaddr(查找与 IP 地址匹配的表项)。图 9.9 对此作了总结。

信息	数据文件	结构	键值搜索函数
主机	/etc/hosts	hostent	gethostbyaddr, gethostbyname
网络	/etc/networks	netent	getnetbyaddr, getnetbyname
协议	/etc/protocols	protoent	getprotobyname, getprotobynumber
服务	/etc/services	servent	getservbyname, getservbyport

图 9.9 四类与网络相关的信息

当 DNS 正在使用时,如何得到这些信息?首先,只有主机和网络信息是通过 DNS 提供的,服务和协议信息一般要从相应的文件中读。在本章前面(图 9.1)我们提到过,不同的实现可让管理员使用不同的方法来指定是使用 DNS 还是使用文件来得到主机和网络信息。

第二,如果 DNS 正用于主机和网络信息,则只有键值查询函数才可用。例如,你不能使用 gethostent 期待顺序浏览 DNS 中的所有表项。如果调用 gethostent,它只读主机文件而避开 DNS。

> 虽然网络信息可以通过 DNS 得到,但很少有人这么用。[Albitz and Liu 1997]第 346~348 页介绍了这个特性。通常,管理员创建并维护文件/etc/networks,并使用它而不是使用 DNS。如果有此文件,则指定-i 选项的 netstat 程序使用它,以输出每个网络的名字。

9.11 小 结

应用程序用来将主机名转换为 IP 地址或进行相反过程的一组函数称为解析器。gethostbyname 和 gethostbyaddr 是两个最常用的函数。随着向 IPv6 的转移,由这两个函数装填的 hostent 结构保持不变,但结构中的某些信息发生了变化。为支持 IPv6,还需要一个新的函数 gethostbyname2 和一个新的解析器选项 RES_USE_INET6。

处理服务名和端口号的常用函数是 getservbyname,它取一个服务名并返回一个包含端

口号的结构。这种映射一般包含在文本文件中。还有用来映射协议名到协议号、网络名到网络号的函数，但很少使用。

我们没有提到的另外一种方法是：直接调用解析器函数，而不使用 gethostbyname 和 gethostbyaddr。以这个方法来调用 DNS 的一个程序是 sendmail，它搜索 MX 记录，这是 gethostbyXXX 函数无法做到的。解析器函数都有以 res_开头的名字，在 9.4 节中描述的 res_init 就是一个例子。这些函数的描述和调用它们的一个程序例子在[Albitz and Liu 1997]的第 14 章中，键入 man resolver 应能输出这些函数的手册页面。

当我们在第 11 章查看对 gethostbyname 和 gethostbyaddr 的协议无关接口：函数 getaddrinfo 和 getnameinfo 时，将继续讨论名字和地址转换这个话题。那两个新函数是设计来与 IPv4 和 IPv6 一起工作的，但我们首先需在下一章看看 IPv4 和 IPv6 的互操作性，然后再讨论它们。

9.12　习　题

9.1　修改图 9.4 中的程序，给每个返回的地址调用 gethostbyaddr，然后输出返回的 h_name。首先指定一个只有单个 IP 地址的主机，运行此程序；然后指定一个具有多个 IP 地址的主机，再运行此程序，将会怎样？

9.2　修复上个习题中出现的问题。

9.3　修改图 9.7，调用 gethostname 而不是 uname。写一个 main 函数来调用 my_addrs 并输出 IP 地址。

9.4　在图 9.8 中，如果我们将 memcpy 的第三个参数改为 hp->h_length（在装填套接口地址结构时），将会怎样？（提示：考虑一下，如果我们设置 RES_OPTIONS = inet6 并指定一个具有 IPv6 地址的主机名，运行程序将会有什么结果。）

9.5　指定服务名为 chargen，运行图 9.8 中的程序。

9.6　指定一个点分十进制数 IP 地址作为主机名，运行图 9.8 中的程序。你的解析器允许这么做吗？修改图 9.8 以允许将点分十进制数 IP 地址作为主机名，将十进制端口号串作为服务名。在给点分十进制数串或主机名测试 IP 地址时，这两个测试将以什么顺序执行？

9.7　修改图 9.8，使其对 IPv4 和 IPv6 都能工作。

9.8　修改图 9.8 以查询 DNS，比较返回的 IP 地址和目的主机的所有 IP 地址。也就是说，用由 recvfrom 返回的 IP 地址调用 gethostbyaddr，后跟 gethostbyname 调用以找出主机的所有 IP 地址。

第3部分 高级套接口编程

第10章 IPv4与IPv6的互操作性

10.1 概 述

即将到来的数年中因特网可能会逐渐从IPv4过渡到IPv6，在这个过渡阶段里，使现有的基于IPv4的应用程序能和新的IPv6的应用程序一起运行是十分重要的。举例来说，某厂商不能只提供一个只能与IPv6的Telnet服务器程序一起运行的Telnet客户程序，她必须既提供能与IPv4服务器程序一起运行的Telnet客户程序，又提供能与IPv6服务器程序一起运行的Telnet客户程序。能提供一个同时兼容IPv4和IPv6服务器程序的Telnet客户程序，和一个同时兼容IPv4和IPv6客户程序的Telnet服务器程序就更好。我们将会在本章中了解这是如何实现的。

我们在本章中假定所有的主机上都运行着双重协议栈(dual stacks)，即IPv4和IPv6协议栈。图2.1中的例子是一个双重协议栈的主机。在向IPv6协议过渡的很长时间内，主机和路由器可能都会像这样运行。在某个时候，许多系统将可以关闭它们的IPv4协议栈，但只有时间才能告诉我们何时能这样做。

在本章中我们讨论IPv4和IPv6的应用程序怎样才能互相通信。在使用IPv4和IPv6协议的客户和服务器之间存在四种组合，如图10.1。

	IPv4服务器	IPv6服务器
IPv4客户	几乎全部现有的客户和服务器程序	在10.2节中讨论
IPv6客户	在10.3节中讨论	简单地修改大部分现有的客户和服务器程序(如从图1.5到图1.6的修改)

图10.1 使用IPv4和IPv6的客户与服务器的组合

对于客户和服务器使用相同协议的两种情况我们将不做过多讨论，最有趣的情况是当客户和服务器使用不同协议时。

10.2 IPv4客户与IPv6服务器

拥有双重协议栈的主机的一个基本特性就是：其上运行的IPv6服务器既能应付IPv4客户，又能应付IPv6客户。这是通过使用IPv4映射的IPv6地址实现的(图A.10)。图10.2给出了这样的一个例子。

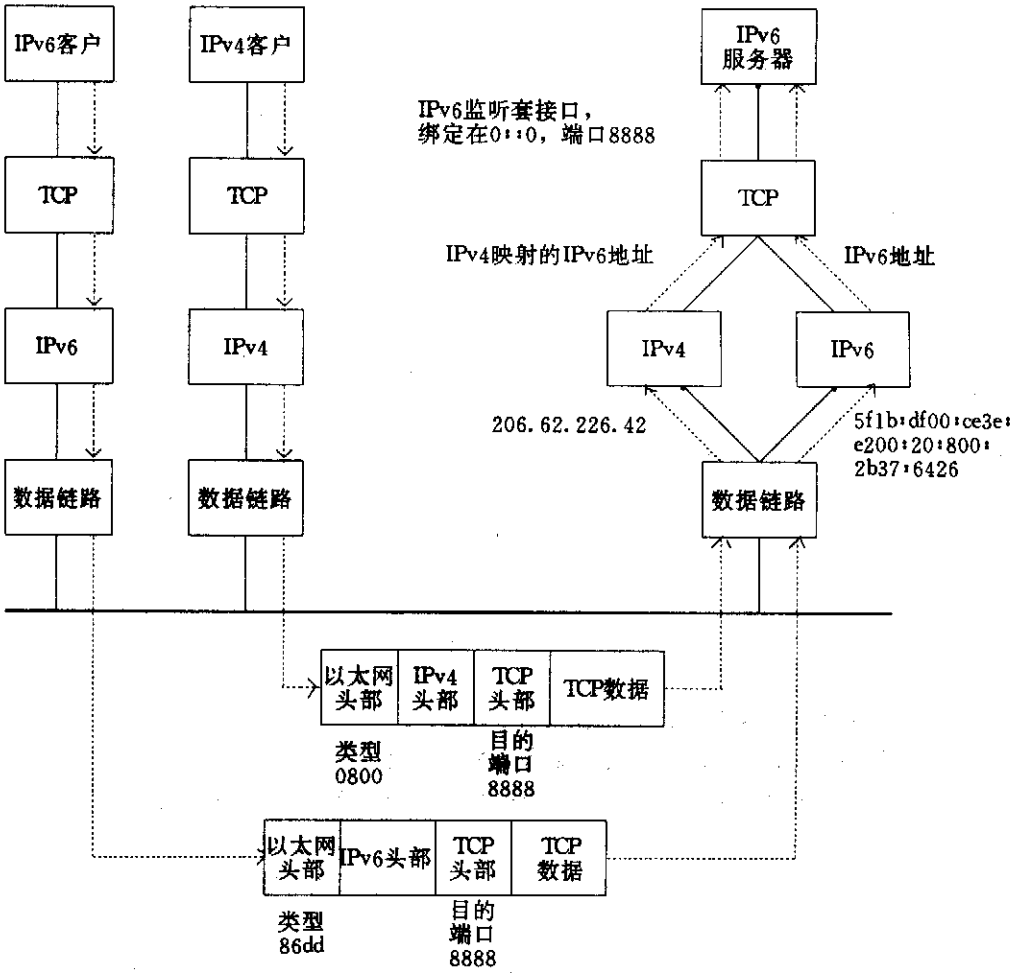


图 10.2　双重协议栈主机上的 IPv6 服务器为 IPv4 和 IPv6 客户服务

在左边是一个 IPv4 客户和一个 IPv6 客户。右边的服务器其程序是用 IPv6 编写的，在一个拥有双重协议栈的主机上运行。该服务器已创建了一个绑定在通配 IPv6 地址和端口 8888 上的 IPv6 监听 TCP 套接口。

我们假定客户和服务器处于同一以太网上。当然，它们也可以是通过路由器来连接的，只要所有的路由器都能支持 IPv4 和 IPv6，但这对我们的讨论并没有什么影响。另外的一种情况是：IPv6 的客户和服务器之间用只支持 IPv4 的路由器连接，这将在 B.3 节中讨论。

我们假定两个客户都发送 SYN 分节与服务器建立连接。IPv4 客户主机将使用 IPv4 数据报发送 SYN，IPv6 客户主机将使用 IPv6 数据报发送 SYN。IPv4 客户发出的 TCP 分节出现在以太网上时是一个以太网头部，紧接着一个 IPv4 头部、一个 TCP 头部和 TCP 数据。以太网头部中包含一个值为 0x0800 的类型字段，标识其为一个 IPv4 帧。在 TCP 头部中包含目的端口 8888。(附录 A 中对这些头部的格式和内容有详细的说明。)在 IPv4 头部中的目的 IP 地址是 206.62.226.42，在图中没有标出。

IPv6客户发出的TCP分节出现在以太网上时是一个以太网头部，紧接着一个IPv6头部、一个TCP头部和TCP数据。以太网头部包含一个值为0x86dd的类型字段，标识其为一个IPv6帧。这个TCP头部和IPv4数据报中的TCP头部格式完全一样，也包含目的端口8888。在IPv6头部中的目的IP地址是5f1b:df00:ce3e:e200:20:800:2b37:6426，在图中没有标出。

正在接收的数据链路检查以太网类型字段，将每个帧送到相应的IP模块。IPv4模块可能还通过其上的TCP模块，检测到IPv4数据报的目的套接口是一个IPv6套接口，于是将其IPv4头部中的源IPv4地址转换成一个等价的IPv4映射的IPv6地址。当accept调用将接受的IPv4客户的连接返回给服务器时，这个映射后的地址作为客户的IPv6地址返回到IPv6套接口中。这个连接上其余的数据报全是IPv4数据报。

当accept调用将接受的IPv6客户的连接返回给服务器时，该客户的IPv6地址就是出现在IPv6头部中的源地址，不做任何改动。这个连接上其余的数据报全是IPv6数据报。我们可以把一个IPv4 TCP客户与一个IPv6服务器之间进行通信的步骤总结如下：

1. 启动IPv6服务器，创建一个IPv6的监听套接口，我们假定该套接口绑定了通配地址。
2. IPv4客户调用gethostbyname找到一个与该服务器对应的A记录(图9.5)。因为这台服务器主机同时支持IPv4和IPv6，所以它应该既有一个A记录，又有一个AAAA记录，但IPv4的客户只需要一个A记录。
3. 客户进程调用connect，客户主机向服务器发送一个IPv4的SYN。
4. 服务器主机收到这个发往IPv6监听套接口的IPv4 SYN，置一个标志，表明这个连接使用IPv4映射的IPv6地址，然后响应一个IPv4的SYN/ACK。当这个连接建立后，accept返回给服务器的地址就是这个IPv4映射的IPv6地址。
5. 在客户和服务器之间的所有通信使用IPv4数据报。
6. 除非服务器明确地去检查这个IPv6地址是不是一个IPv4映射的IPv6地址(使用在10.4节中介绍的IN6_IS_ADDR_V4MAPPED宏)，它将不会知道通信的对方是一个IPv4客户。双重协议栈屏蔽了这个细节。同样，IPv4的客户也不知道与之通信的是一个IPv6的服务器。

这里有一个假设，就是其中拥有双重协议栈的服务器主机既有一个IPv4地址，又有一个IPv6地址。在所有IPv4地址耗尽之前这是可行的。

对于一个IPv6的UDP服务器来说，情形是类似的，但每个数据报都将改变一次地址格式。举例来说，如果一个IPv6服务器收到一个从IPv4客户来的数据报，由recvfrom返回的地址将是该客户的IPv4映射的IPv6地址。服务器使用这个IPv4映射的IPv6地址，调用sendto对客户的请求作出响应。这种地址格式告诉内核向客户发送IPv4数据报。但服务器收到的下一个数据报可能是IPv6数据报，recvfrom将返回其IPv6地址。如果服务器响应，内核将生成一个IPv6数据报。

图10.3总结了在一个双重协议栈的主机上，根据接收套接口的类型(TCP或UDP)，对一个收到的IPv4或IPv6数据报进行处理的流程。

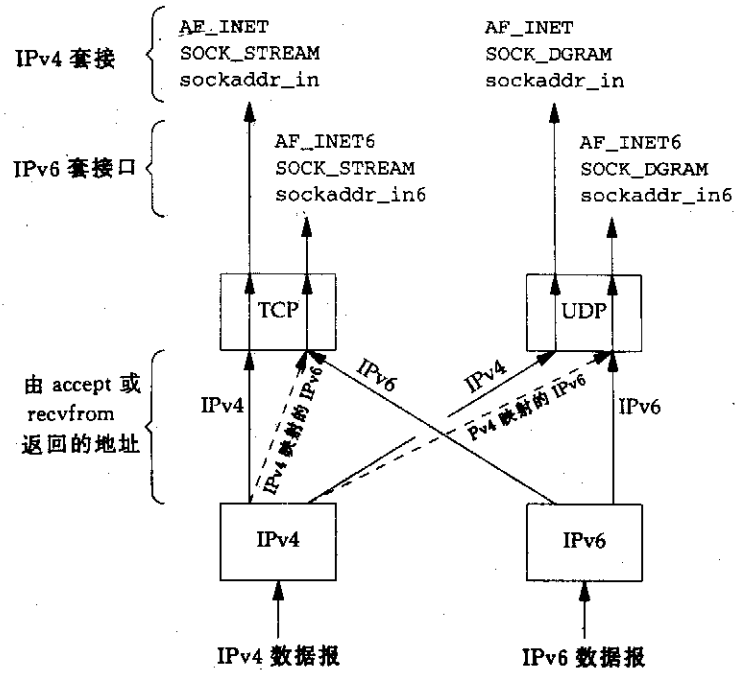


图 10.3　根据接收套接口的类型，对收到的 IPv4 或 IPv6 数据报的处理

- 如果在一个 IPv4 的套接口上收到了 IPv4 数据报，不需作任何特殊处理。在图中有两个标为“IPv4”的箭头，一个到 TCP，一个到 UDP。客户和服务器之间交换 IPv4 数据报。
- 如果在一个 IPv6 的套接口上收到了 IPv6 数据报，也不需作任何特殊处理。在图中有两个标为“IPv6”的箭头，一个到 TCP，一个到 UDP。客户和服务器之间交换 IPv6 数据报。
- 但当在一个 IPv6 套接口上收到 IPv4 数据报时，系统内核会在 accept(TCP)或 recvfrom(UDP)调用返回地址时将对应的 IPv4 映射的 IPv6 地址返回，这就是在图中的两个虚线箭头。因为每个 IPv4 地址都能表示成一个 IPv6 地址，所以这种映射是可行的。在客户和服务器之间交换的是 IPv4 数据报。
- 与上一条相反的过程是不存在的：通常一个 IPv6 地址不能表示成一个 IPv4 地址；因而没有从 IPv6 协议框到两个 IPv4 套接口的箭头。

大多数双重协议栈主机应使用以下规则处理监听套接口：

1. 监听 IPv4 套接口只能接受来自 IPv4 客户的外来连接。
2. 如果服务器有一个绑定在通配地址上的监听 IPv6 套接口，该套接口就能接受来自 IPv4 客户或 IPv6 客户的外来连接。对于来自 IPv4 客户的连接，其在服务器端的本地地址是对应的 IPv4 映射的 IPv6 地址。
3. 如果服务器有一个绑定在非 IPv4 映射的 IPv6 地址上的监听套接口，该套接口就只能接受来自 IPv6 客户的外来连接。

10.3 IPv6客户与IPv4服务器

我们现在把上一节例子中客户和服务器使用的协议掉一下个儿。首先考虑一个在双重协议栈主机上运行的IPv6 TCP客户。

1. 一个IPv4服务器在只有IPv4协议栈的主机上启动，创建了一个IPv4的监听套接口。
2. IPv6客户启动，调用gethostbyname只询问IPv6地址(它打开了RES_USE_INET6选项)。因为IPv4服务器主机只有A记录，从图9.5可以看到客户将得到一个IPv4映射的IPv6地址。
3. IPv6客户调用connect，在相应的IPv6套接口地址结构中存放所得到的IPv4映射的IPv6地址。系统内核检查到这个映射地址，于是自动向服务器发送一个IPv4的SYN。
4. 服务器用IPv4的SYN/ACK响应，连接通过IPv4数据报建立。

我们可用图10.4总结一下这种情况：

- 如果IPv4的TCP客户调用connect时或IPv4的UDP客户调用sendto时指定的是一个IPv4地址，不需要作任何特殊处理。这就是图中标为"IPv4"的两个箭头。
- 如果IPv6的TCP客户调用connect时或IPv6的UDP客户调用sendto时指定的是一个IPv6地址，不需要作任何特殊处理。这就是图中标为"IPv6"的两个箭头。

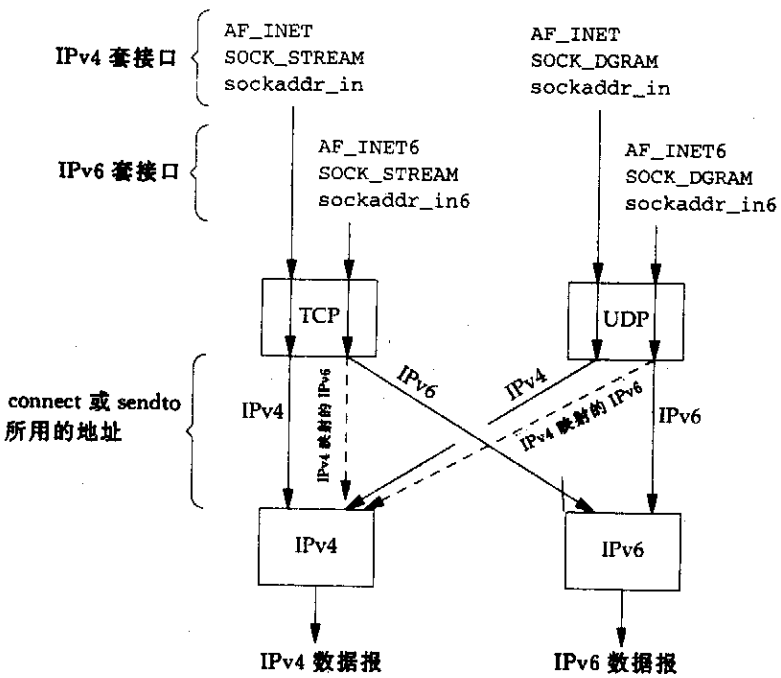


图10.4 根据地址类型和套接口类型，对客户请求的处理

- 如果 IPv6 的 TCP 客户调用 connect 时，或 IPv6 的 UDP 客户调用 sendto 时指定的是一个 IPv4 映射的 IPv6 地址，内核会检测到这个映射地址，并发送一个 IPv4 数据报，而不是 IPv6 数据报。这是图中的两条虚线箭头。
- IPv4 客户不能在调用 connect 或 sendto 时指定一个 IPv6 地址，这是因为在 IPv4 sockaddr_in 结构里的 4 字节的 in_addr 结构中，放不下一个 16 字节的 IPv6 地址。因此图中从 IPv4 套接口到 IPv6 协议框之间没有箭头。

在前一节中（一个 IPv4 的数据报到达 IPv6 服务器套接口），将 IPv4 地址转换成 IPv4 映射的 IPv6 地址是由内核完成的，accept 或 recvfrom 将其透明地返回给应用进程。在这一节中（要在一个 IPv6 套接口上发送 IPv4 数据报），IPv4 地址到 IPv4 映射的 IPv6 地址的转换是由如图 9.5 中的解析器完成的，应用进程将映射后的地址透明地传递给 connect 或 sendto。

对互操作性的总结

图 10.5 对本节和上一节内容，以及客户与服务器的组合情况作了总结。

	IPv4 服务器 IPv4 主机 (A)	IPv6 服务器 IPv6 主机 (AAAA)	IPv4 服务器 双重协议栈主机 (A 和 AAAA)	IPv6 服务器 双重协议栈主机 (A 和 AAAA)
IPv4 客户，IPv4 主机	IPv4	(no)	IPv4	IPv4
IPv6 客户，IPv6 主机	(no)	IPv4	(no)	IPv4
IPv4 客户，双重协议栈主机	IPv4	(no)	IPv4	IPv4
IPv6 客户，双重协议栈主机	IPv4	IPv6	(no*)	IPv6

图 10.5　对 IPv4 和 IPv6 客户与服务器之间互操作性的总结

栏中的“IPv4”或“IPv6”表示这种组合是可互操作的，而且指出了实际通信所使用的协议。栏中的“(no)”则表示对应的组合是不能互操作的。最后一行的第三列标有一个“*”号，是因为其互操作性与客户选择的地址有关。选择 AAAA 记录发送 IPv6 数据报将不能互操作。但选择 A 记录，给客户返回一个 IPv4 映射的 IPv6 地址，因而发送 IPv4 数据报，将能正常工作。

尽管在表中有四分之一的情况不能互操作，实际上在可预见的将来，大多数 IPv6 的实现将是在双重协议栈主机上而不仅仅是在 IPv6 协议栈主机上。如果我们删去表中的第二行和第二列，所有的“(no)”将消失，只剩下一个有问题的“(no)*”。

10.4　IPv6 地址测试宏

有一小部分 IPv6 应用程序必须知道与其通信的对方是否使用 IPv4 协议。这些应用程序需要知道对方的地址是不是一个 IPv4 映射的 IPv6 地址。为测试 IPv6 地址的某些特性共定义了 12 个宏。

```
#include <netinet/in.h>

int IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *aptr);
int IN6_IS_ADDR_LOOPBACK(const struct in6_addr *aptr);
int IN6_IS_ADDR_MULTICAST(const struct in6_addr *aptr);
int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4MAPPED(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4COMPAT(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr *aptr);

                返回值:非零表示IPv6地址是指定类型的,否则返回零
```

前7个宏测试IPv6地址的基本类型。我们在A.5节中介绍了这些不同的地址类型。后5个宏测试一个IPv6多播地址的范围(19.2节)。

IPv6客户可以用IN6_IS_ADDR_V4MAPPED来测试由解析器返回的IPv6地址。IPv6服务器也可以用这个宏来测试由accept或recvfrom返回的IPv6地址。

我们可以把FTP和它的PORT指令作为需要使用这个宏的一个例子。如果启动一个FTP客户,登录到FTP服务器上,并发出FTP dir命令,FTP客户会在控制连接上向FTP服务器发送一个PORT指令。这条指令告诉服务器客户的IP地址和端口号,服务器将使用这些参数向客户发起建立一个数据连接。(TCPv1的第27章中包含FTP应用协议的所有细节。)但IPv6的FTP客户必须知道服务器是一个IPv4的服务器还是IPv6的服务器,因为前者需要一个"PORT a1,a2,a3,a4,P1,P2"格式的指令,其中前四个数字(每个都在0～255之间)组成一个4字节的IPv4地址,后两个数字组成2字节的端口号。而一个IPv6的服务器需要一个LPRT指令(参见RFC 1639 [Piscitello 1994]),其中包含21个数字。习题10.1中给出了这两种指令的例子。

10.5 IPV6_ADDRFORM套接口选项

IPV6_ADDRFORM套接口选项能把一个套接口从一种类型转变成另一种类型,它受到以下限制:

1. 一个IPv4套接口总能转变为IPv6套接口。所有与该套接口关联的IPv4地址被转换成IPv4映射的IPv6地址。
2. 一个IPv6套接口只有在与之关联的地址是IPv4映射的IPv6地址的情况下才能转变成IPv4套接口。

为什么要转变套接口的地址格式呢?其原因是在UNIX上文件描述字可以在进程之间进行传递。最通常的方式是通过fork,我们也将看到文件描述字是怎样在有亲缘关系的进程(14.7节)和无亲缘关系的进程(25.7节)之间传递的。

举例来说，假设一个进程创建了一个 IPv4 的监听套接口，然后接收到一个来自 IPv4 客户的连接。该服务器调用 fork 和 exec，启动一个新程序处理客户请求。这里有个约定，即已连接套接口通过标准输入、标准输出和标准错误输出传递给新程序（与 inetd 所做的相似，见 12.5 节），于是就有图 10.6 中的一段伪代码。它与 4.8 节中的并发服务器的唯一不同是将已连接套接口复制到事先约定的描述字，然后调用 exec。

但被加载的新程序期望一个 IPv6 套接口。我们可用 IPV6_ADDRFORM 套接口选项来转换该套接口的地址格式，如图 10.7 所示。

```
int                 listenfd,connfd;
socklen_t           clilen;
struct sockaddr_in  serv,cli;                    /* IPv4 structs */

listenfd = Socket(AF_INET,SOCK_STREAM,0);     /* IPv4 socket */

/* fill in serv() with well-known prot */
Bind(listenfd,&serv,sizeof(serv));
Listen(listenfd,LISTENQ);

for ( ; ; ) {
    clilen = sizeof(cli);
    connfd = Accept(listenfd,&cli,&clilen);

    if(Fork() == 0) {
        Close(listenfd);              /* child */
        Dup2(connfd,STDIN_FILENO);
        Dup2(connfd,STDOUT_FILENO);
        Dup2(connfd,STDERR_FILENO);
        Close(connfd);
        Exec( ... );                  /* start new program */
    }
    Close(connfd);                    /* parent */
}
```

图 10.6　接受外来连接并启动新程序的服务器程序

```
int                 af;
socklen_t           clilen;
struct sockaddr_in6 cli;          /* IPv6 struct */
struct hostent      *ptr;

af = AF_INET6;
Setsockopt(STDIN_FILENO,IPPROTO_IPV6,IPV6_ADDRFORM,&af,sizeof(af));

clilen = sizeof(cli);
Getpeername(0,&cli,&clilen);

ptr = gethostbyaddr(&cli.sin6_addr, 16, AF_INET6);
```

图 10.7　将一个 IPv4 套接口转换成 IPv6 套接口

假设该套接口是一个 IPv4 套接口，调用 setsockopt 将把套接口的地址格式由 IPv4 变为 IPv6，getpeername 则会返回一个 IPv4 映射的 IPv6 地址。但是如果该套接口是一个 IPv6 套接口，那么调用 setsockopt 将不起作用，因为地址格式已是 IPv6 了。

一种可以使用该套接口选项的情况是，接收外来 IPv4 连接的程序是由他人提供的（即没有源码，我们无法修改使之适用于 IPv6，当然协议无关则更好），但我们的被加载程序是

来处理 IPv6 的。

如果用 IPV6_ADDRFORM 作参数来调用 getsockopt，返回值依赖于套接口地址的格式，将为 AF_INET 或 AF_INET6。getsockopt 和 setsockopt 的第二个参数可以是 IPPROTO_IP 或 IPPROTO_IPV6。

10.6 源代码可移植性

大部分现有的网络应用程序是为 IPv4 编写的。为此将分配和填写 sockaddr_in 结构，并且 socket 调用会指定 AF_INET 作为其第一个参数。从图 1.5 到图 1.6 的转换中可以看出，把这些 IPv4 应用程序转换成能用于 IPv6 并不麻烦。很多修改都可用编辑脚本自动完成。一些对 IPv4 依赖性很强的程序，如使用多播、IP 选项或原始套接口功能的程序，将要花费更多的转换工作。

如果在源代码级对应用程序进行向 IPv6 的移植和发布，就不得不考虑接受者的系统是否支持 IPv6。典型的方法是在代码中使用 #ifdef 伪码，以尽可能使用 IPv6（因为我们已在本章中看到，IPv6 客户仍能与 IPv4 服务器通信，反之亦然）。这种方法存在的问题是代码会随着 #ifdef 的使用很快变得混乱，难于理解和维护。

更好的方法是把这种向 IPv6 的转换看成是一个使程序独立于协议的机会。第一步是将 gethostbyname 和 gethostbyaddr 调用删除，转而使用在下一章中描述的 getaddrinfo 和 getnameinfo 函数。这使我们将套接口地址结构作为不透明的对象来处理，就像 bind、connect、recvfrom 等基本的套接口函数所做的那样，由一个指针和大小来引用。3.8 节中的 sock_XXX 函数能帮助我们在独立于 IPv4 和 IPv6 的情况下处理这些问题。显然这些函数中包含 #ifdef 来处理 IPv4 和 IPv6，但是将所有对协议的依赖隐蔽到几个库函数中，编码会更简单。在 19.6 节中我们将开发一组 mcast_XXX 函数，它们能使多播的应用程序独立于 IPv4 或 IPv6。

另一点需要考虑的是，如果在一个同时支持 IPv4 和 IPv6 的系统上编译源代码，发布其执行码或目标文件（不是源代码），并在不支持 IPv6 的系统上运行这些程序时会发生什么。这有一种可能，即本地的名字服务器支持 AAAA 记录，并对这些程序要连接的主机同时返回 AAAA 记录和 A 记录。如果一个 IPv6 的应用程序在一个不支持 IPv6 的主机上调用 socket 创建 IPv6 套接口，将会失败。在下一章里介绍的函数能处理这种情况，它们将忽略 socket 产生的错误，并尝试使用名字服务器返回的地址表中的下一个地址。假设对方有一个 A 记录，并且名字服务器在返回所有 AAAA 记录后附加一个 A 记录的话，创建 IPv4 套接口将会成功。这种类型的功能应由某个库函数提供，而不是每个应用程序的源代码都提供。

10.7 小 结

在双重协议栈主机上的 IPv6 服务器能为 IPv4 和 IPv6 客户服务。IPv4 客户向服务器发送的仍是 IPv4 数据报，但服务器的协议栈会把客户方的地址转换成一个 IPv4 映射的 IPv6 地址，因为 IPv6 服务器处理的是 IPv6 套接口地址结构。

类似地，在双重协议栈主机上的 IPv6 客户能够和 IPv4 服务器通信。客户的解析器代码将给服务器主机的所有 A 记录返回一个 IPv4 映射的 IPv6 地址，在双重协议栈客户主机上调用 connect 与这些地址建立连接时，由双重协议栈发送一个 IPv4 SYN 分节。只有少数几个特殊的客户和服务器程序需要知道对方使用的协议版本（例如 FTP），可以用 IN6_IS_ADDR_V4MAPPED 宏来判断对方是否使用 IPv4。程序可用 IPV6_ADDRFORM 套接口选项得到自己期望的套接口类型（通常是 IPv6 套接口）。

10.8　习　题

1. 在一台双重协议栈主机上启动 IPv6 FTP 客户。连接一个 IPv4 FTP 服务器，发出 debug 命令，然后是 dir 命令。再对一个 IPv6 FTP 服务器作同样的操作，比较由 dir 命令所发出的 PORT 指令。
2. 编写需要一个 IPv4 点分十进制地址作为命令行参数的程序。它创建一个 IPv4 TCP 套接口，并将这个地址和某个端号口，如 8888，捆绑到该套接口。调用 listen，然后调用 pause。编写一个类似的程序，但其命令行参数是一个 IPv6 的十六进制数串，创建的是监听 IPv6 TCP 套接口。用通配地址作参数运行 IPv4 程序。然后在另一个窗口里用 IPv6 通配地址作参数启动 IPv6 程序。因为 IPv4 程序已经绑定该端口，还能启动 IPv6 程序吗？使用 SO_REUSEADDR 套接口选项会有什么不同吗？如果先启动 IPv6 程序，再启 IPv4 程序又会如何？

第11章 高级名字与地址转换

11.1 概 述

在第9章中介绍的两个函数:gethostbyname 和 gethostbyaddr,是依赖于协议的。使用前一个函数时,我们必须知道放置结果的套接口地址结构的成员是哪一种(举例来说,IPv4 使用 sin_addr 成员,IPv6 使用 sin6_addr 成员),而调用后一个函数时,必须知道存放二进制地址的是哪一种成员。这一章以能为应用程序提供协议独立性的新的 Posix.1g 的 getaddrinfo 函数作为开头,后面将介绍跟它互补的 getnameinfo。

在这之后我们将用该函数开发6个自己的函数,以处理 TCP 或 UDP 客户和服务器程序的典型情况。后面的文章中将会使用这些函数,而不是直接调用 getaddrinfo。

gethostbyname 和 gethostbyaddr 也是不可重入函数的好例子。我们将解释为什么会是这样,并介绍一些替代函数来避免这个问题。重入问题我们将在第23章中再讨论,但在没有详细了解线程的情况下,我们也能指出和解释这个问题。

本章将以展示 getaddrinfo 的完整实现来结束,这将让我们对该函数有更多的理解:它是怎样操作的,返回什么,如何与 IPv4 和 IPv6 交互。

11.2 getaddrinfo 函数

函数 getaddrinfo 在库函数中隐藏了所有协议依赖性。应用程序只需要处理由 getaddrinfo 填写的套接口地址结构。该函数在 Posix.1g 中定义。

Posix.1g 对这个函数的定义来源于 Keith Sklower 早先提出的名为 getconninfo 的函数。这个函数是与 Eric Allman、Walliam Durst、Michael Karels 以及 Steven Wise 讨论的结果,最早的实现是由 Eric Allman 写的。指定主机名和服务名将足以与一个独立于协议细节的服务建立连接,Marshall Rose 在对 X/Open 的一个提议中作出了对此的观测。

```
#include <netdb.h>
int getaddrinfo (const char *hostname,const char *service,
                 const struct addrinfo *hints,struct addrinfo **result);
                                        返回:成功返回0,出错返回非零(见图11.3)
```

这个函数通过 result 指针返回一个指向 addrinfo 结构链表的指针,该结构在<netdb.h>中定义:

```
struct addrinfo {
  int              ai_flags;      /* AI_PASSIVE,AI_CANONNAME */
  int              ai_family;     /* AF_xxx */
  int              ai_socktype;   /* SOCK_xxx */
```

```
    int             ai_protocol;    /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
    size_t          ai_addrlen;     /* length of ai_addr */
    char            *ai_canonname;  /* ptr to canonical name for host */
    struct sockaddr *ai_addr;       /* ptr to socket address structure */
    struct addrinfo *ai_next;       /* ptr to next structure in linked list */
};
```

其中的 hostname 是主机名或地址串(IPv4 的点分十进制数表示或 IPv6 的十六进制数串)。service 是服务名或十进制数的端口号字符串。(回想在习题 9.6 中,我们是如何解决允许用地址串作为主机名,用端口号串作为服务名的。)

hints 是一个空指针或指向一个 addrinfo 结构的指针,由调用者填写关于它所想返回的信息类型的线索。举例来说,如果某种服务既可以用 TCP 又可以用 UDP 提供(譬如,由 DNS 服务器提供的 domain 服务),调用者可以将 hints 结构中的 ai_socktype 成员的值设为 SOCK_DGRAM,返回的信息就只适用于数据报套接口。

调用者可以设置的 hints 结构的成员有:

- ai_flags(AI_PASSIVE, AI_CANONNAME)
- ai_family(某个 AF_xxx 值)
- ai_socktype(某个 SOCK_xxx 值),和
- ai_protocol

AI_PASSIVE 标志表示该套接口是用作被动的打开,AI_CANONNAME 标志则通知 getaddrinfo 函数返回主机的名字。

如果 hints 是一个空指针,该函数将假定 ai_flag、ai_socktype 和 ai_protocol 的值为零,ai_family 的值为 AF_UNSPEC。

如果函数返回成功(0),result 参数指向的变量将被填入一个指针,它指向一个通过 ai_next 指针串起来的 addrinfo 结构链表。返回这个复合结构有两种方式。

1. 如果与该 hostname 对应的有多个地址,将按请求的地址族(如果指定了 ai_family 线索)为每个地址返回一个结构。
2. 如果该服务在多种套接口类型上提供,将根据 ai_socktype 线索为每个套接口类型返回一个结构。

举例来说,如果在没有提供任何线索的条件下,要求 domain 服务查找一个有两个 IP 地址的主机,将返回 4 个 addrinfo 结构:

- 一个是第一个 IP 地址和 SOCK_STREAM 套接口类型
- 一个是第一个 IP 地址和 SOCK_DGRAM 套接口类型
- 一个是第二个 IP 地址和 SOCK_STREAM 套接口类型
- 一个是第二个 IP 地址和 SOCK_DGRAM 套接口类型

图 11.1 中是这个例子的示意图。这里并不能保证返回的多个条目的结构顺序;也就是说,不能假定 TCP 服务在 UDP 服务之前返回。

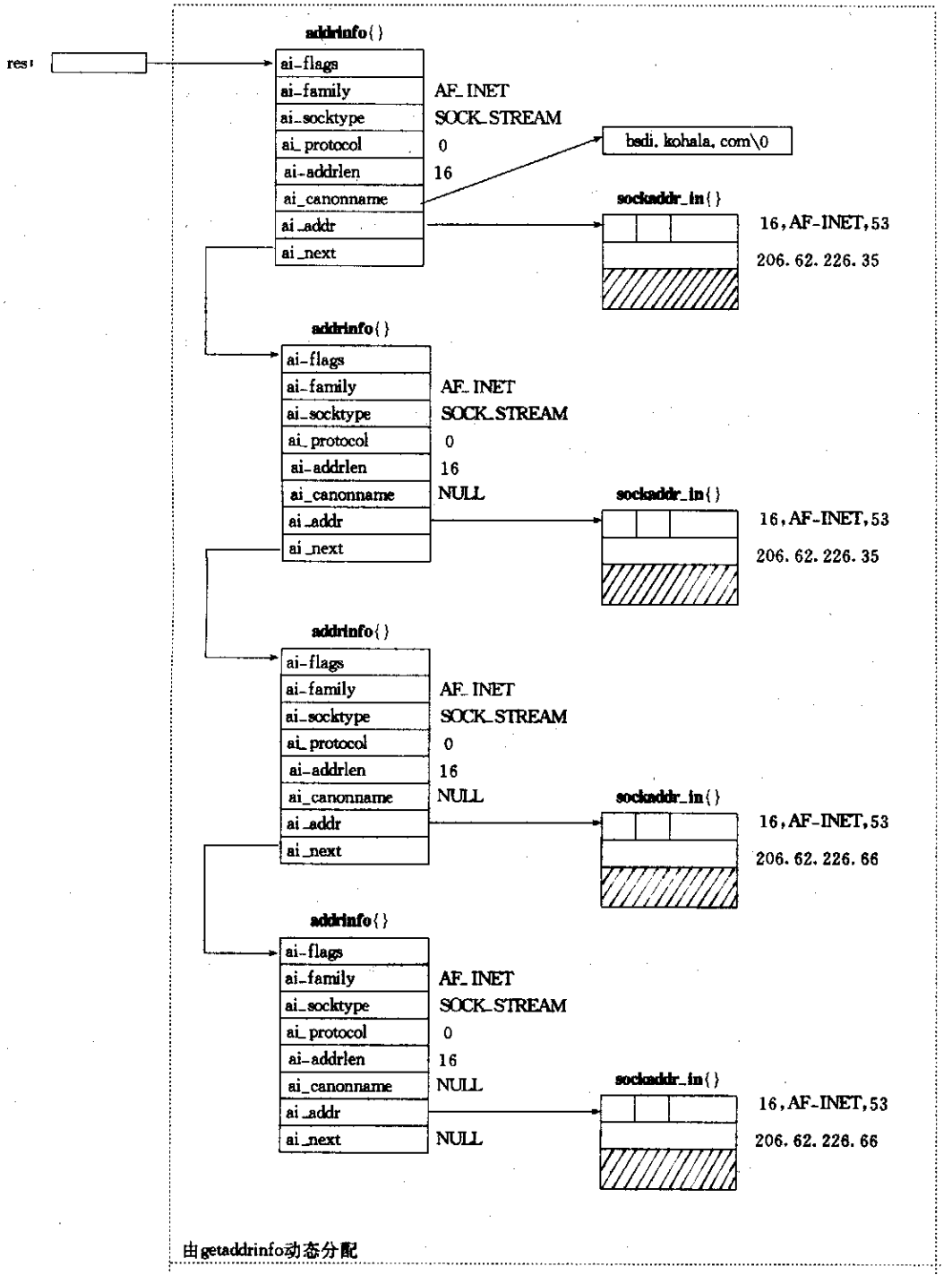


图 11.1 getaddrinfo 返回的信息的实例

尽管没有保证，但在具体实现中应按 DNS 返回的顺序返回 IP 地址。举例来说，许多 DNS 服务器把返回的地址排序，这样如果发出查询请求的主机与名字服务器在同一网络上，在这一共享网络上的地址将首先返回。更新版本的

BIND 也允许解析器在/etc/resolv.conf 文件中指定地址的排序顺序。

在 addrinfo 结构中返回的信息可用于调用 socket，然后调用 connect、sendto（客户）或 bind（服务器）。socket 函数的参数是 addrinfo 结构中的 ai_family、ai_socktype 和 ai_addr。connect 或 bind 函数的第二个和第三个参数是 ai_addr（一个指向适当类型的套接口地址结构的指针，由 getaddrinfo 填写）和 ai_addrlen（套接口地址结构的大小）。

如果设置了在 hints 结构中的 AI_CANONNAME 标志，返回的第一个结构的 ai_canonname 成员指向相应主机的名字。DNS 的术语通常称之为 FQDN。

图 11.1 给出了执行下列程序片段时返回的信息。

```
struct addrinfo     hints, *res;
bzero(&hints,sizeof(hints));
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_INET;
getaddrinfo("bsdi","domain",&hints,&res);
```

在图中除 res 变量外的数据空间都是动态分配的（譬如用 malloc），我们假设主机 bsdi 的别名是 bsdi.kohala.com，而且在 DNS 中它有两个 IPv4 地址（图 1.16）。

端口 53 用于 domain 服务，而且在套接口地址结构中的这个端口号是网络字节序的。返回的 ai_protocol 的值为 0，这是因为 ai_family 和 ai_socktype 的组合完全指定了协议为 TCP 和 UDP。getaddrinfo 在其中的两个 SOCK_STREAM 结构的 ai_protocol 中返回 IPPROTO_TCP，在其中的两个 SOCK_DGRAM 结构的 ai_protocol 中返回 IPPROTO_UDP 也是可以的。

图 11.2 总结了根据指定的服务名（可以是一个十进制数的端口号）和 ai_socktype 线索，为每个地址返回的 addrinfo 结构的数目。

ai_socktype 线索	服务以名字标识，它的提供者为：			服务以端口号标识
	TCP	UDP	TCP 与 UDP	
0	1	1	2	2
SOCK_STREAM	1	错误	1	1
SOCK_DGRAM	错误	1	1	1

图 11.2　为每个 IP 地址返回的 addrinfo 结构的数目

只有在未提供 ai_socktype 线索且或者服务以名字标识，并同时由 TCP 和 UDP 提供支持时（在/etc/services 文件中指明），或者服务以端口号标识时，才会为每个 IP 地址返回多个 addrinfo 结构。

如果枚举 getaddrinfo 的 64 种可能的输入（有六个输入变量），许多都是无效的，有些没什么意义。因而我们将注意力集中到一些常见情况。

- 指定 hostname 和 service。这在 TCP 或 UDP 客户程序中很常见。TCP 客户程序遍历所有返回的 IP 地址，逐一调用 socket 和 connect，直到连接成功或所有地址被试过为止。在图 11.6 中我们开发的 tcp_connect 函数是一个例子。

  在 UDP 客户程序中，由 getaddrinfo 填写的套接口地址结构被用来调用 sendto 或 connect。如果第一个地址不行（在已连接的 UDP 套接口上出错或在未连接的套接口上超时），就会尝试剩下的地址。

  如果客户程序知道它只处理一种类型的套接口（例如，Telnet 和 FTP 客户程序只处

理 TCP，TFTP 客户程序只处理 UDP），就应把 hints 结构中的 ai_socket 设为 SOCK_STREAM 或 SOCK_DGRAM。

- 一个典型的服务器程序只用指定 service 以及 hints 结构中的 AI_PASSIVE 标志，而不需要指明 hostname。返回的套接口地址结构中应包含一个 INADDR_ANY（IPv4）或 IN6ADDR_ANY_INIT（IPv6）的 IP 地址。TCP 服务器程序随后调用 socket、bind 和 listen。如果服务器程序要 malloc 另一个套接口地址结构以从 accept 取得客户的地址，返回的 ai_addrlen 的值将给出所需的内存大小。

  UDP 的服务器程序将调用 socket、bind 和 recvfrom。如果服务器要 malloc 另一个套接口地址结构以从 recvfrom 取得客户的地址，返回的 ai_addrlen 的值将给出所需的内存大小。

  和典型的客户程序代码一样，如果服务器程序知道它只需处理一种类型的套接口，hints 结构中的 ai_socket 应被设为 SOCK_STREAM 或 SOCK_DGRAM。这样可以避免返回多重结构，其中可能出现错误的 ai_socktype 值。
- 到目前为止，我们展示的 TCP 服务器程序都只创建一个监听套接口，UDP 服务器只创建一个数据报套接口。我们在上一个条目中就是这么假设的。另一种服务器程序设计方法是用 select 处理多个套接口。这种情况下服务器将遍历由 getaddrinfo 所返回的整个结构链表，并为每个结构创建一个套接口，然后使用 select。

  这种技术的问题是，getaddrinfo 返回多重结构的一个原因是该服务能同时使用 IPv4 和 IPv6（图 11.4）。但就像在 10.2 节中所看到的，这两种协议并不是完全独立的。也就是说，如果在给定的端口上创建了一个 IPv6 监听套接口，就没有必要在同一端口上再创建 IPv4 套接口了，因为协议栈和 IPv6 监听套接口能自动处理由 IPv4 客户发出的连接。

尽管 getaddrinfo 比 gethostbyname 和 getservbyname 函数更“好”（它使我们能更容易地编写独立于协议的代码，单个函数既处理了主机名又处理了服务，而且所有返回的信息是动态分配的），但它仍不是太好用。问题在于必须分配一个 hints 结构，初始化为 0，填上必要的字段，再调用 getaddrinfo，然后遍历链表逐一尝试。在下一节中会给典型的 TCP 和 UDP 客户或服务器程序提供一些更简单的接口，以在后文中使用。

getaddrinfo 解决了将主机名和服务名转换成套接口地址结构的问题。在 11.13 节中将介绍一个功能与其相反的函数 getnameinfo，它把套接口地址结构转换成主机名和服务名。在 11.16 节中将提供一个 getaddrinfo、getnameinfo 和 freeaddrinfo 的实现。

11.3 gai_strerror 函数

getaddrinfo 返回的非零错误值的名字和含义如图 11.3 所示。gai_strerror 以这些值作为它的一个参数，返回指向对应的出错信息字符串的指针。

```
#include <netdb.h>
char *gai_strerror(int error);
```

返回：一个指向描述出错信息字符串的指针

常　量	描　述
EAI_ADDRFAMILY	不支持 hostname 的地址族
EAI_AGAIN	名字解析中的暂时失败
EAI_BADFLAGS	ai_flags 的值无效
EAI_FAIL	名字解析中不可恢复的失败
EAI_FAMILY	不支持 ai_family
EAI_MEMORY	内存分配失败
EAI_NODATA	没有与 hostname 相关联的地址
EAI_NONAME	hostname 或 service 未提供，或者不可知
EAI_SERVICE	不支持 ai_socktype 类型的 service
EAI_SOCKTYPE	不支持 ai_socktype
EAI_SYSTEM	errno 中有系统错误返回

图 11.3　getaddrinfo 返回的非 0 错误常值

11.4　freeaddrinfo 函数

由 getaddrinfo 返回的存储空间，包括 addrinfo 结构、ai_addr 结构和 ai_canonname 字符串，都是用 malloc 动态获取的。这些空间可调用 freeaddrinfo 释放。

```
#include <netdb.h>
void freeaddrinfo(struct addrinfo *ai);
```

ai 应指向 getaddrinfo 返回的第一个 addrinfo 结构。在该链表中的所有结构，以及这些结构所指向的动态存储空间都将被释放（譬如套接口地址结构和规范主机名）。

假设我们调用 getaddrinfo，顺着 addrinfo 结构链表找到所需的结构。然后只复制该 addrinfo 结构以保存其信息，再调用 freeaddrinfo，就会产生一个潜藏的错误。原因是 addrinfo 结构中的指针指向动态分配的内存（用于存放地址结构和规范主机名），因此由我们保存的结构所指向的内存在调用 freeaddrinfo 后就释放，可能将作它用。

只复制 addrinfo 结构，而不复制 addrinfo 结构所指向的其他结构，叫做浅拷贝或浅复制（shallow copy）。复制 addrinfo 结构，同时复制 addrinfo 结构所指向的其他结构，称为深拷贝或深复制（deep copy）。

11.5 getaddrinfo 函数:IPv6 和 UNIX 域

尽管 Posix.1g 定义了 getaddrinfo 函数,但它没有专门对 IPv6 做任何说明。该函数与名字解析(特别是 RES_USE_INET6 选项,参见图 9.5),以及 IPv6 之间的交互是不平常的。在用图 11.4 对这些交互作出总结前,我们注意以下几点:

- getaddrinfo 处理两种不同的输入:调用者想返回的套接口地址结构类型和应在 DNS 中查找的记录类型。
- 由调用者提供的 hints 结构中的地址协议族,指明了它想返回的套接口地址结构类型。如果其值指定为 AF_INET,就不会返回任何 sockaddr_in6 结构;如果是 AF_INET6,则不会返回任何 sockaddr_in 结构。
- Posix.1g 中提到如指定为 AF_UNSPEC,则应返回可用于该主机名和服务名的所有协议族的地址。这意味着,如果某主机既有 AAAA 记录,又有 A 记录,则将返回一个 sockaddr_in6 结构的 AAAA 记录的和一个 sockaddr_in 结构的 A 记录。将 A 记录转换成一个 IPv4 映射的 IPv6 地址也用 sockaddr_in6 结构返回是没有什么意义的,因为它并没有提供任何额外信息:这些地址已在 sockaddr_in 结构中返回了。
- Posix.1g 中的这个说明也暗示,如果设置了 AI_PASSIVE 标志,而未指定主机名,则将返回一个值为 IPv6 通配地址(IN6ADDR_ANY_INIT 或 0::0)的 sockaddr_in6 结构和一个值为 IPv4 通配地址(INADDR_ANY 或 0.0.0.0)的 sockaddr_in 结构。在 10.2 节中我们看到 IPv6 服务器在双重协议栈主机上对 IPv6 和 IPv4 的客户都能处理,所以首先返回 IPv6 通配地址是有其意义的。
- 解析器选项 RES_USE_INET6 和所调用的函数(gethostbyname 或 gethostbyname2)一块确定在 DNS 中查找的记录类型(A 或 AAAA),以及返回的地址类型(IPv4、IPv6 或 IPv4 映射的 IPv6)。图 9.5 对此作了总结。
- 主机名还可以是 IPv6 十六进制数串或 IPv4 点分十进制数串。该串的有效性取决于调用者指定的地址族。如果指定 AF_INET,就不能接受 IPv6 十六进制数串;如果指定 AF_INET6,则不能接受 IPv4 点分十进制数串。如果指定的是 AF_UNSPEC,那么两种数串都可以,返回的是对应类型的套接口地址结构。

  有人可能会争论说,如果指定了 AF_INET6,则对于点分十进制数串应该用 sockaddr_in6 结构返回对应的 IPv4 映射的 IPv6 地址。但是有另外一种方法得到这种结果,那就是在点分十进制串前加上 0::ffff.

图 11.4 总结了 getaddrinfo 对 IPv4 和 IPv6 地址是怎样处理的。"结果"一栏是在给定前三栏的变量后,返回给调用者的结果。"行为"栏则说明怎样得到这个结果,我们将在 11.6 节 getaddrinfo 的实现中展示执行这些操作的代码。

调用者指定的主机名	调用者指定的地址族	主机名字符串包含	结　果	行　为
非空字符串；主动或被动	AF_UNSPEC	主机名	所有 AAAA 记录以 sockaddr_in6() 返回，所有 A 记录以 sockaddr_in() 返回	两次 DNS 查找(注 1)：RES_USE_INET6 关闭下 gethostbyname2(AF_INET6)；RES_USE_INET6 关闭下 gethostbyname2(AF_INET)
		十六进制数串	一个 sockaddr_in6()	inet_pton(AF_INET6)
		点分十进制数	一个 sockaddr_in()	inet_pton(AF_INET)
	AF_INET6	主机名	所有 AAAA 记录以 sockaddr_in6() 返回，否则所有 A 记录转成 IPv4 映射的 IPv6 地址以 sockaddr_in6() 返回	RES_USE_INET6 打开下 gethostbyname()(注 2)
		十六进制数串	一个 sockaddr_in6()	inet_pton(AF_INET6)
		点分十进制数	出错：EAI_ADDRFAMILY	
	AF_INET	主机名	所有 A 记录以 sockaddr_in() 返回	RES_USE_INET6 关闭下 gethostbyname()
		十六进制数串	出错：EAI_ADDRFAMILY	
		点分十进制数	一个 sockaddr_in()	inet_pton(AF_INET)
空串；被动	AF_UNSPEC	隐含 0::0 隐含 0.0.0.0	一个 sockaddr_in6() 和 一个 sockaddr_in()	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	隐含 0::0	一个 sockaddr_in6()	inet_pton(AF_INET6)
	af_INET	隐含 0.0.0.0	一个 sockaddr_in()	inet_pton(AF_INET)
空串；主动	AF_UNSPEC	隐含 0::1 隐含 127.0.0.1	一个 sockaddr_in6() 和 一个 sockaddr_in()	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	隐含 0::1	一个 sockaddr_in6()	inet_pton(AF_INET6)
	AF_INET	隐含 127.0.0.1	一个 sockaddr_in()	inet_pton(AF_INET)

图 11.4　对 getaddrinfo 函数的行为和结果的总结

表中注 1 是当执行这两次 DNS 查找时，它们都有可能失败(即找不到所给主机名的所需类型的记录)，但起码有一次会成功。如果两次查找都成功(该主机既有 AAAA 记录又有 A 记录)，则两种类型的套接口地址结构都将返回。

注 2 是 DNS 查找必须成功，否则将返回错误。但因为设置了 RES_USE_INET6 选项，gethostbyname 会先查 AAAA 记录，如果没找到，再查 A 记录(图 9.6)。

在这两个注所处的情形中，解析器的 RES_USE_INET6 选项的打开与关闭目的是按照图 9.5 中的规则，强制执行期望的 DNS 查找。

图 11.4 只说明了 getaddrinfo 是怎样处理 IPv4 和 IPv6 的；也就是返回给调用者的地址的数目。返回的 addrinfo 结构的实际数目还依赖于指定的套接口类型和服务名，像图 11.2 中总结的那样。

Posix.1g 没有对 getaddrinfo 和 Unix 域套接口作任何说明(在 14 章中将作详细介绍)。然而，在我们的 getaddrinfo 实现中加入对 Unix 域套接口的支持，并在这些协议上对应用程序进行测试就可以很好地检测协议独立性。

我们的实现作了以下的假设：如果 getaddrinfo 的主机名参数为/local 或/unix，而且服务名参数是绝对路径名(以斜杠开头)，则返回 Unix 域的套接口。有效的 DNS 主机名不能包含斜杠，也不存在以斜杠开头的 IANA 服务名(习题 11.5)。返回的套接口地址结构包含该绝对路径名，它用来调用 bind 或 connect。如果调用者设定了 AI_CANONNAME 标志，该主机的名字(9.7 节)将作为规范主机名返回。

11.6 getaddrinfo 函数：例子

现在用一个测试程序来展示 getaddrinfo 的一些例子，它运行时让我们输入所有的参数：主机名、服务名、地址族、套接口类型、AI_CANONNAME 和 AI_PASSIVE 标志。(这儿没有给出源程序，因为它大约有 350 行。如前言中所介绍的，它以源码形式提供。)测试程序输出返回的可变数目的 addrinfo 结构中的信息，包括调用 socket 所用的参数和每个套接口地址结构中的地址。

我们首先展示与图 11.1 同样的例子。

```
solaris % testga -f inet -c -h bsdi -s domain
socket(AF_INET, SOCK_STREAM,0),ai_canonname = bsdi.kohala.com
        address: 206.62.226.35.53
socket(AF_INET, SOCK_DGRAM, 0)
        address: 206.62.226.35.53
socket(AF_INET, SOCK_STREAM, 0)
        address: 206.62.226.66.53
socket(AF_INET, SOCK_DGRAM, 0)
        address: 206.62.226.66.53
```

-f inet 选项指定地址族，-c 表示返回规范主机名，-h bsdi 指定主机名，-s domain 指定服务名。

通常客户指定地址族、套接口类型(用-t 选项)、主机名和服务名。下面的例子展示了这种情况，其中的主机是一个拥有 6 个 IPv4 地址的多宿主机。

```
solaris % testga -f inet -t stream -h gateway.tuc.noao.edu -s daytime
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.101.4.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.102.1.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.104.1.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.3.6.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.4.100.13
socket(AF_INET,SOCK_STREAM, 0)
        address: 140.252.1.4.13
```

下面指定主机为 alpha，它有一个 AAAA 记录和一个 A 记录，不指定协议族，指定服务名为只能在 TCP 上提供的 ftp 服务。

```
solaris  % testga  -h alpha  -s ftp
socket(AF_INET6,SOCK_STREAM, 0)
        address: 5f1b:df00:ce3e:e200:20:800:2b37:6426.21
socket(AF_INET,SOCK_STREAM, 0)
        address: 206.62.226.42.21
```

因为没有指定地址族，而且该例子是在一个能支持 IPv4 和 IPv6 的主机上运行，所以返回了两个结构：一个 IPv6 的和一个 IPv4 的。

下面设定 AI_PASSIVE 标志(用-p 选项)，不指定地址族，不指定主机名(相当于使用通配地址)，指定端口号为 8888，不指定套接口类型。

```
solaris % testga  -p  -s 8888
socket(AF_INET6,SOCK_STREAM, 0)
        address: ::.8888
socket(AF_INET6,SOCK_DGRAM, 0)
        address: ::.8888
socket(AF_INET,SOCK_STREAM, 0)
        address: 0.0.0.0.8888
socket(AF_INET,SOCK_DGRAM, 0)
        address: 0.0.0.0.8888
```

返回了 4 个结构。因为是在一个既支持 IPv6 又支持 IPv4 的主机上运行，而且没有指定地址族，getaddrinfo 返回 IPv6 通配地址和 IPv4 通配地址。由于指定了端口号而未指定套接口类型，getaddrinfo 为每个地址返回一个 TCP 类型的结构和一个 UDP 类型的结构。在第 10 章中可以看到在双重协议栈主机上的一个 IPv6 的客户或服务器既能与 IPv6 的对方通信，也能与 IPv4 的对方通信，所以两个 IPv6 的结构在 IPv4 结构前返回。

作为 Unix 域套接口的一个例子，下面指定主机名为 /local，服务名为 /tmp/test.1。

```
solaris % testga  -c  -p  -h /local  -s /tmp/test.1
socket(AF_LOCAL, SOCK_STREAM, 0), ai_canonname = solaris.kohala.com
        address: /tmp/test.1
socket(AF_LOCAL, SOCK_DGRAM, 0)
        address: /tmp/test.1
```

因为没有指定套接口类型，所以返回了两个结构：第一个是字节流套接口，第二个是数据报套接口。

11.7　host_serv 函数

我们给 getaddrinfo 设计的第一个接口不需要调用者来分配和填写 hints 结构。这个 host_serv 函数以地址族和套接口类型作为参数。

```
#include "unp.h"
struct addrinfo *host_serv(const char *hostname, const char *service, int
                           family, int socktype);
```

返回：成功返回指向addrinfo结构的指针，出错返回NULL

图11.5为该函数的源代码。

```
1 #include    "unp.h"
2 struct addrinfo *
3 host_serv(const char *host, const char *serv, int family, int socktype)
4 {
5     int     n;
6     struct addrinfo hints, *res;
7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_flags = AI_CANONNAME;      /* always return canonical name */
9     hints.ai_family = family;           /* AF_UNSPEC, AF_INET, AF_INET6, etc. */
10    hints.ai_socktype = socktype;       /* 0, SOCK_STREAM, SOCK_DGRAM, etc. */
11    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
12        return(NULL);
13    return(res);        /* return pointer to first on linked list */
14 }
```

图11.5 host_serv函数[lib/host_serv.c]

第7～13行 这个函数初始化一个hints结构，调用getaddrinfo，如果出错则返回一个空指针。

在图15.17中，当要用getaddrinfo获取主机和服务信息，但想自己建立连接时，调用了该函数。

11.8 tcp_connect函数

现在我们编写两个使用getaddrinfo的函数，以处理我们编写的TCP客户和服务器程序的大部分情况。第一个函数即tcp_connect执行客户程序的一般操作步骤：创建一个TCP套接口并与服务器建立连接。

```
#include "unp.h"
int tcp_connect(const char *hostname, const char *service);
```

返回：如成功则返回已连接套接口的描述字，出错则不返回

图11.6是它的源代码。

```
1 #include    "unp.h"
2 int
3 tcp_connect(const char *host, const char *serv)
4 {
5     int     sockfd, n;
```

```
 6    struct addrinfo   hints, *res, *ressave;
 7    bzero(&hints, sizeof(struct addrinfo));
 8    hints.ai_family = AF_UNSPEC;
 9    hints.ai_socktype = SOCK_STREAM;
10    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11        err_quit("tcp_connect error for %s, %s: %s",
12                 host, serv, gai_strerror(n));
13    ressave = res;
14    do {
15        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16        if (sockfd < 0)
17            continue;           /* ignore this one */
18        if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19            break;              /* success */
20        Close(sockfd);          /* ignore this one */
21    } while ( (res = res->ai_next) != NULL);
22    if (res == NULL)            /* errno set from final connect() */
23        err_sys("tcp_connect error for %s, %s", host, serv);
24    freeaddrinfo(ressave);
25    return(sockfd);
26 }
```

图 11.6　tcp_connect 函数：执行客户程序的一般操作[lib/tcp_connect.c]

调用 getaddrinfo

第 7～13 行　调用了一次 getaddrinfo，这里指定地址族为 AF_UNSPEC，套接口类型为 SOCK_STREAM。

逐一尝试每个 addrinfo 结构，直到成功或到达链表尾

第 14～25 行　每个返回的 IP 地址都会被尝试：调用 socket 和 connect。如果在一台不支持 IPv6 的主机上用 IPv6 地址调用 socket，将返回错误，但这种错误不是致命的。如果 connect 成功，将会 break 出循环。否则，试过所有地址后，循环也会终止。freeaddrinfo 释放所有动态分配的内存。

如果 getaddrinfo 失败，或所有 connect 失败，这个函数（以及在以下各节中描述的给 getaddrinfo 提供一些简单接口的其他函数）将终止。只有在成功时才返回。如果不另加一个参数，是很难返回一个错误码的（EAI_xxx 常值之一）。这意味着我们的包裹函数是很简单的：

```
int
Tcp_connect(const char *host,const char *serv)
{
    return(tcp_connect(host,serv));
}
```

虽然如此，为了保持一致性，在下面的章节中我们使用自己的包裹函数代替 tcp_connect。

返回值的问题在于描述字是非负的值，可是我们并不知道EAI_xxx是正的还是负的。如果这些值是正数，在getaddrinfo失败时我们就可以返回这些值的负值，但我们还得返回某个另外的负值，表明所有结构都已尝试仍不成功的情况。

例子：时间/日期客户程序

图11.7是用tcp_connect重新编码的时间/日期客户程序。

命令行参数

第9～10行　现在需要第二个命令行参数指定服务名或端口号，以使程序连接其他的端口。

连接服务器

第11行　这个客户程序的所有套接口代码现由tcp_connect执行。

输出服务器地址

第12～15行　我们调用getpeername取得服务器的协议地址并输出。这是为了验证本例子中所使用的协议。

注意tcp_connect不返回connect要用的套接口地址结构的大小，可以加上一个指针参数来返回这个值，但是该函数的一个设计目标就是减少所用的参数的数目（与getaddrinfo相比）。因而在unp.h头文件中，定义了一个代表最大的套接口地址结构的大小的MAXSOCKADDR常值来弥补这一点。这个值通常是Unix域套接口地址结构的大小（14.2节），超过100个字节。我们给这么大的结构分配空间，它由getpeername来填写。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int         sockfd, n;
6     char        recvline[MAXLINE + 1];
7     socklen_t   len;
8     struct sockaddr  *sa;
9     if (argc != 3)
10        err_quit("usage: daytimetcpcli <hostname/IPaddress> <service/port#>");
11    sockfd = Tcp_connect(argv[1], argv[2]);
12    sa = Malloc(MAXSOCKADDR);
13    len = MAXSOCKADDR;
14    Getpeername(sockfd, sa, &len);
15    printf("connected to %s\n", Sock_ntop_host(sa, len));
16    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
17        recvline[n] = 0;   /* null terminate */
18        Fputs(recvline, stdout);
19    }
20    exit(0);
21 }
```

图11.7　用tcp_connect重新编码的时间/日期客户程序[names/daytimetcpcli.c]

我们用 malloc 来给该结构分配空间，而不是用：

```
char sockaddr[MAXSOCKADDR];
```

是因为字节对齐的问题。malloc 总是返回一个系统所要求的严格对齐的指针，而 char 数组可能被分配在一个奇数字节的边界上，这将对套接口地址结构中的 IP 地址或端口号造成问题。解决这个潜在问题的另一种方法是使用一个联合（图 4.19）。

这个版本的客户程序能与 IPv4 和 IPv6 一起工作，而图 1.5 中的版本只能用于 IPv4，图 1.6 中的版本只能用于 IPv6。你也应把这个新版本与图 E.14 中的做一下比较，在后者的编码里使用了 gethostbyname 和 getservbyname 来支持 IPv4 和 IPv6。

我们首先指定一个只支持 IPv4 的主机名。

```
solaris % daytimetcpcli bsdi daytime
connected to 206.62.226.35
Fri May 30 12:33:32 1997
```

下面指定一个 IPv4 和 IPv6 都支持的主机名。

```
solaris % daytimetcpcli aix daytime
connected to 5f1b:df00:ce3e:e200:20:800:5afc:2b36
Fri May 30 12:43:43 1997
```

使用 IPv6 地址的原因是该主机有一个 AAAA 记录和一个 A 记录，而且如图 11.4 所注，由于 tcp_connect 将地址族设为 AF_UNSPEC，因此首先搜索的是 AAAA 记录，只有在这失败后才搜索 A 记录。

下一个例子中，我们通过指定带－4 后缀的主机名来强制使用 IPv4 地址，我们在 9.2 节中已指出这是只有 A 记录的主机名的约定命名。

```
solaris % daytimetcpcli aix-4 daytime
connected to 206.62.226.43
Fri May 30 12:43:48 1997
```

11.9　tcp_listen 函数

下一个函数即 tcp_listen 执行 TCP 服务器程序的一般操作步骤：创建一个 TCP 套接口，给它捆绑服务器的众所周知端口，并允许接受外来的连接请求。图 11.8 是它的源代码。

```
#include "unp.h"

int tcp_listen(const char *hostname, const char *service, socklen_t *lenptr);
```

返回：成功返回已连接套接口描述字，出错则不返回

调用 getaddrinfo

第 8～15 行　初始化 addrinfo 结构时用了以下线索：AI_PASSIVE（因为这个函数是服务器程序用的）、AF_UNSPEC（地址族）、以及 SOCK_STREAM。回想图 11.4，如果没有指

定主机名(想绑定通配地址的服务器通常如此),使用 AI_PASSIVE 和 AF_UNSPEC 选项会返回两个套接口地址结构:第一个是 IPv6 的,下一个是 IPv4 的(假定是一个双重协议栈主机)。

创建套接口并给它捆绑地址

第 16～24 行 调用 socket 和 bind 函数。如果任何一个调用失败,则忽略这个 addrinfo 结构而使用下一个。如 7.5 节中所说明的,我们对于 TCP 服务器程序总是设置 SO_REUSEADDR 套接口选项。

失败检查

第 25～26 行 如果对所有的地址结构调用 socket 和 bind 都失败了,就输出错误并终止。就像前一节中的 tcp_connect 函数一样,我们不会试图从这个函数中返回错误。

第 27 行 调用 listen 使该套接口变成一个监听套接口。

返回套接口地址结构的大小

第 28～31 行 如果 addrlenp 参数不为空,就通过这个指针返回协议地址的大小。这可以让调用者给套接口地址结构分配内存,以用 accept 获得客户的协议地址(还可参见习题 11.1)。

```
1 #include    "unp.h"
2 int
3 tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
4 {
5     int       listenfd, n;
6     const int       on = 1;
7     struct addrinfo    hints, *res, *ressave;
8     bzero(&hints, sizeof(struct addrinfo));
9     hints.ai_flags = AI_PASSIVE;
10    hints.ai_family = AF_UNSPEC;
11    hints.ai_socktype = SOCK_STREAM;
12    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
13      err_quit ("tcp_listen error for %s, %s: %s",
14              host, serv, gai_strerror(n));
15    ressave = res;
16    do {
17      listenfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
18      if (listenfd < 0)
19        continue;          /* error, try next one */
20      Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21      if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
22        break;             /* success */
23      Close(listenfd);     /* bind error, close and try next one */
24    } while ( (res = res->ai_next) != NULL);
25    if (res == NULL)       /* errno from final socket() or bind() */
26        err_sys("tcp_listen error for %s, %s", host, serv);
27    Listen(listenfd, LISTENQ);
```

```
28      if (addrlenp)
29          *addrlenp = res->ai_addrlen;    /* return size of protocol address */
30      freeaddrinfo(ressave);
31      return(listenfd);
32  }
```

图 11.8　tcp_listen 函数：执行服务器程序的一般操作步骤[lib/tcp_listen.c]

例子：时间/日期服务器

图 11.9 是用 tcp_listen 重新编码的时间/日期服务器程序

需要服务名或端口号作为命令行参数

第 11～12 行　需要一个命令行参数指定服务名或端口号。这会使对服务器程序的测试更容易，因为捆绑标准时间/日期服务器的端口 13 需要超级用户权限。

```
 1 #include    "unp.h"
 2 #include    <time.h>
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int            listenfd, connfd;
 7     socklen_t      addrlen, len;
 8     char           buff[MAXLINE];
 9     time_t         ticks;
10     struct sockaddr    *cliaddr;
11     if (argc != 2)
12         err_quit("usage: daytimetcpsrv1 <service or port#>");
13     listenfd = Tcp_listen(NULL, argv[1], &addrlen);
14     cliaddr = Malloc(addrlen);
15     for ( ; ; ) {
16         len = addrlen;
17         connfd = Accept(listenfd, cliaddr, &len);
18         printf("connection from %s\n", Sock_ntop(cliaddr, len));
19         ticks = time(NULL);
20         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21         Write(connfd, buff, strlen(buff));
22         Close(connfd);
23     }
24 }
```

图 11.9　用 tcp_listen 重新编码的时间/日期服务器程序(另见图 11.10)[names/daytimetcpsrv1.c]

创建监听套接口

第 13～14 行　tcp_listen 创建监听套接口，调用 malloc 分配一个缓冲区以存放客户的地址。

服务器循环

第 15～23 行　accept 等待客户的连接。调用 sock_ntop 输出客户的地址。无论是 IPv4

或IPv6,这个函数都输出IP地址和端口号。我们可以使用getnameinfo函数(在11.13节中介绍)来试图获得客户的主机名,这将会包含一个对DNS的PTR查询,PTR查询需要花费一定的时间,特别是在查询失败时。TCPv3的14.8节注明在一个繁忙的Web服务器上所有的与之连接的客户中大概有25%在DNS中没有PTR记录。因为我们不想让服务器(特别是一个迭代服务器)为一个PTR查询等待数秒钟,所以只输出IP地址和端口号。

例子:可指定协议的时间/日期服务器

图11.9中有一个小问题:如果tcp_listen的第一个参数是一个空指针,而且地址族为AF_UNSPEC的话,可能会使getaddrinfo返回我们不想要的地址族的套接口地址结构。举例来说,在一个双重协议栈主机上返回的第一个套接口地址结构会是IPv6的(图11.4),但我们可能想要服务器只处理IPv4。

客户没有这个问题,因为客户总是必需指定一个IP地址或主机名。客户程序一般让用户用命令行参数来输入它。这使我们有机会指定一个与特定类型的IP地址相关联的主机名(回想在9.2节中用的-4和-6主机名后缀),或者指定一个IPv4点分十进制数串(强制使用IPv4)或IPv6十六进制数串(强制使用IPv6)。

但是有一种简单的技术可以让我们强制服务器使用一种给定的协议,或IPv4或IPv6:让用户通过命令行参数输入一个IP地址或主机名,并将其传递给getaddrinfo。如果使用IP地址,IPv4点分十进制数串和IPv6十六进制数串是有差别的。下面对inet_pton的调用如所示的那样或成功或失败。

```
inet_pton(AF_INET, "0.0.0.0",  &foo);    /* succeeds */
inet_pton(AF_INET, "0::0",     &foo);    /* fails */
inet_pton(AF_INET6, "0.0.0.0", &foo);    /* fails */
inet_pton(AF_INET6, "0::0",    &foo);    /* succeeds */
```

因此,如果把我们的服务器程序改成能接受一个可选的参数,那么要是键入:

```
% server
```

在一个双重协议栈主机上就缺省使用IPv6,但是键入

```
% server 0.0.0.0
```

则显式指定使用IPv4,键入

```
% server 0::0
```

则指定使用IPv6。

图11.10是我们的时间/日期服务器程序的这个最终版本。

处理命令行参数

第11~16行 与图11.9相比唯一的改变是对命令行参数的处理,允许用户在服务名或端口号外,指定一个服务器绑定的主机名或IP地址,

首先用一个IPv4套接口启动服务器,然后从两个位于本地子网的主机上的客户向服务器发起连接。

```
solaris % daytimetcpsrv2 0.0.0.0 9999
```

```
connection from 206.62.226.36.32789
connection from 206.62.226.35.1389
```

再用 IPv6 套接口启动服务器。

```
solaris % daytimetcpsrv2 0::0 9999
connection from 5f1b:df00:ce3e:e200:20:800:2003:f642.32799
connection from 5f1b:df00:ce3e:e200:20:800:2b37:6426.1026
connection from ::ffff:206.62.226.36.32792
connection from ::ffff:206.62.226.35.1390
```

```
 1 #include    "unp.h"
 2 #include    <time.h>
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int                 listenfd, connfd;
 7     socklen_t           addrlen, len;
 8     struct sockaddr     *cliaddr;
 9     char                buff[MAXLINE];
10     time_t              ticks;
11     if (argc == 2)
12       listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13     else if (argc == 3)
14       listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15     else
16       err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");
17     cliaddr = Malloc(addrlen);
18     for ( ; ; ) {
19         len = addrlen;
20         connfd = Accept(listenfd, cliaddr, &len);
21         printf("connection from %s\n", Sock_ntop(cliaddr, len));
22       ticks = time(NULL);
23       snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
24       Write(connfd, buff, strlen(buff));
25         Close(connfd);
26     }
27 }
```

图 11.10　使用 tcp_listen 独立于协议的时间/日期服务器程序[names/daytimetcpsrv2.c]

```
connection from ::ffff:206.62.226.36.32792
connection from ::ffff:206.62.226.35.1390
```

第一个连接来自主机 sunos5，使用 IPv6，第二个来自主机 alpha，使用 IPv6。下两个连接来自主机 sunos5 和 bsdi，但使用 IPv4，而不是 IPv6。这是因为由 accept 返回的客户地址都是 IPv4 映射的 IPv6 地址。

刚才展示的是，在双重协议栈主机上运行的 IPv6 服务器能处理 IPv4 和 IPv6 客户。如同在 10.2 节中论述的那样，IPv4 客户的地址被作为一个 IPv4 映射的 IPv6 地址传递给

IPv6 服务器。

这个服务器程序以及图 11.7 中的客户程序都可以用于 Unix 域套接口(第 14 章),因为 11.16 节中 getaddrinfo 的实现支持 Unix 域套接口。举例来说,可像下面这样启动服务器。

```
solaris % daytimetcpsrv2 /local /tmp/rendezvous
```

/tmp/rendezvous 是任意选来给服务器捆绑的路径名。然后在同一台主机上启动客户,指定主机名为/local,服务名为/tmp/rendezvous。

```
solaris % daytimetcpcli /local /tmp/rendezvous
connected to /tmp/rendezvous
Fri May 30 16:31:37 1997
```

11.10 udp_client 函数

给 getaddrinfo 提供简单接口的函数在 UDP 这儿有所改变,因为这里我们提供了一个创建未连接 UDP 套接口的客户函数,在下一节中则提供另一个创建已连接 UDP 套接口的函数。

```
#include "unp.h"
int udp_client(const char *hostname, const char *service,
               void **saptr, socklen_t *lenp);
                                返回:成功返回未连接套接口的描述字,出错则不返回
```

这个函数创建一个未连接 UDP 套接口,返回三项数据。第一,返回值是套接口描述字。第二,saptr 是一个指向套接口地址结构(由 udp_client 动态分配)的指针(由调用者声明)的地址,在这个结构中存放目的 IP 地址和端口号,用来调用 sendto。套接口地址结构的大小在 lenp 指向的变量中返回。最后一个参数不能是空指针(tcp_listen 的最后一个参数是允许的),因为套接口地址结构的长度在调用 sendto 和 recvfrom 时都是需要的。

> saptr 应声明为 struct sockaddr **。使用 void **数据类型是因为在 31.3 节中定义了这个函数的另一个版本,它使用这个参数容纳一个指向另一种类型结构的指针的地址。这意味着我们调用该函数时必需加上(void **)强制类型转换。

图 11.11 给出了这个函数的源代码。

getaddrinfo 转换 hostname 和 service 参数。socket 创建数据报套接口。malloc 为一个套接口地址结构分配内存,并由 memcpy 将对应于创建的套接口的地址结构拷贝到这个内存区域中。

例子:独立于协议的时间/日期客户程序

为了使用 UDP 和 udp_client 函数,现在我们将图 11.7 中的时间/日期客户程序重新编

码。图 11.12 给出了这个独立于协议的程序的源代码。

第 11～16 行　调用 udp_client 函数，然后输出我们要向其发送 UDP 数据报的服务器的 IP 地址和端口号。发送一个 1 字节的数据报后读取并输出应答数据报。

```
1 #include    "unp.h"
2 int
3 udp_client(const char *host, const char *serv, void **saptr, socklen_t *lenp)
4 {
5     int          sockfd, n;
6     struct addrinfo    hints, *res, *ressave;
7     bzero(&hints, sizeof(struct addrinfo));
8     hints.ai_family = AF_UNSPEC;
9     hints.ai_socktype = SOCK_DGRAM;
10    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11      err_quit("udp_client error for %s, %s: %s",
12          host, serv, gai_strerror(n));
13    ressave = res;
14    do {
15      sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16      if (sockfd >= 0)
17        break;              /* success */
18    } while ( (res = res->ai_next) != NULL);
19    if (res == NULL)          /* errno set from final socket() */
20      err_sys("udp_client error for %s, %s", host, serv);
21    *saptr = Malloc(res->ai_addrlen);
22    memcpy(*saptr, res->ai_addr, res->ai_addrlen);
23    *lenp = res->ai_addrlen;
24    freeaddrinfo(ressave);
25    return(sockfd);
26 }
```

图 11.11　udp_client 函数：创建一个未连接 UDP 套接口[lib/udp_client.c]

我们只需要发出一个 0 字节的 UDP 数据报，因为触发时间/日期服务器响应的是到来的数据报，而与它的长度和内容无关。但是许多 SVR4 的实现不允许 0 长度的 UDP 数据报。

下面在运行我们的客户程序时指定了一个拥有 AAAA 记录和 A 记录的主机名。因为 getaddrinfo 首先返回的结构是 AAAA 记录的，所以创建了一个 IPv6 套接口。

```
solaris % daytimeudpcli1 aix daytime
sending to 5f1b:df00:ce3e:e200:20:800:5afc:2b36
Sat May 31 08:13:34 1997
```

下面则指定同一主机的点分十进制数地址，结果创建了一个 IPv4 套接口。

```
solaris % daytimeudpcli1 206.62.226.43 daytime
sending to 206.62.226.43
Sat May 31 08:14:02 1997
```

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int             sockfd, n;
6     char            recvline[MAXLINE + 1];
7     socklen_t       salen;
8     struct sockaddr* sa;
9     if (argc != 3)
10        err_quit("usage: daytimeudpcli1 <hostname/IPaddress> <service/port
          >");
11    sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);
12    printf("sending to %s\n", Sock_ntop_host(sa, salen));
13    Sendto(sockfd, "", 1, 0, sa, salen);        /* send 1-byte datagram */
14    n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
15    recvline[n] = 0;  /* null terminate */
16    Fputs(recvline, stdout);
17    exit(0);
18 }
```

图 11.12 使用 udp_client 的 UDP 时间/日期客户程序[names/daytimeudpcli1.c]

11.11 udp_connect 函数

udp_connect 函数创建一个已连接 UDP 套接口。

```
#include "unp.h"
int udp_connect(const char *hostname,const char *service);
                                返回:如果成功则返回套接口描述字,出错则不返回
```

得到已连接 UDP 套接口后,udp_client 所需的后两个参数就不再需要了。调用者可以用 write 代替 sendto,于是就不再需要返回套接口地址结构和它的长度了。

图 11.13 是源代码。

这个函数与 tcp_connect 几乎一样。但有一点不同是,调用在 UDP 套接口上的 connect 不会向对方发送任何数据报。如果有些错误(对方不可达或在指定端口上没有服务器),调用者在向对方发出数据报前不能发现。

```
1 #include    "unp.h"
2 int
3 udp_connect(const char *host, const char *serv)
4 {
5     int                 sockfd, n;
6     struct addrinfo    hints, *res, *ressave;
7     bzero(&hints, sizeof(struct addrinfo));
```

```
 8    hints.ai_family = AF_UNSPEC;
 9    hints.ai_socktype = SOCK_DGRAM;
10    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
11        err_quit("udp_connect error for %s, %s: %s",
12                 host, serv, gai_strerror(n));
13    ressave = res;
14    do {
15        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
16        if (sockfd < 0)
17            continue;    /* ignore this one */
18        if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
19            break;               /* success */
20        Close(sockfd);     /* ignore this one */
21    } while ( (res = res->ai_next) != NULL);
22    if (res == NULL)   /* errno set from final connect() */
23        err_sys("udp_connect error for %s, %s", host, serv);
24    freeaddrinfo(ressave);
25    return(sockfd);
26 }
```

图 11.13　udp_connect 函数：建立一个已连接 UDP 套接口[lib/udp_connect.c]

11.12　udp_server 函数

我们为简化 getaddrinfo 的接口开发的最后一个 UDP 函数是 udp_server。

```
#include "unp.h"
int udp_server(const char *hostname,const char *service,socklen_t *lenptr);
                                        返回：成功返回未连接套接口描述字，失败不返回
```

参数和 tcp_listen 一样：一个可选的 hostname，一个必需的 service(以给它捆绑一个端口)以及一个可选的指针，它指向返回套接口地址结构大小的变量。

图 11.14 是其源代码。

```
 1 #include    "unp.h"
 2 int
 3 udp_server(const char *host, const char *serv, socklen_t *addrlenp)
 4 {
 5     int                 sockfd, n;
 6     struct addrinfo    hints, *res, *ressave;
 7     bzero(&hints, sizeof(struct addrinfo));
 8     hints.ai_flags = AI_PASSIVE;
 9     hints.ai_family = AF_UNSPEC;
10     hints.ai_socktype = SOCK_DGRAM;
```

```
11    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
12        err_quit("udp_server error for %s, %s: %s",
13                 host, serv, gai_strerror(n));
14    ressave = res;
15    do {
16        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
17        if (sockfd < 0)
18          continue;           /* error, try next one */
19        if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0)
20            break;            /* success */
21        Close(sockfd);        /* bind error, close and try next one */
22    } while ( (res = res->ai_next) != NULL);
23    if (res == NULL)          /* errno from final socket() or bind() */
24        err_sys("udp_server error for %s, %s", host, serv);
25    if (addrlenp)
26        *addrlenp = res->ai_addrlen;   /* return size of protocol address */
27    freeaddrinfo(ressave);
28    return(sockfd);
29 }
```

图 11.14 udp_server 函数：给 UDP 服务器创建一个未连接套接口[lib/udp_server.c]

这个函数除了没有调用 listen 外，几乎与 tcp_listen 相同。我们将地址族设为 AF_UNSPEC，但调用者能以与图 11.10 中相同的技术来强制使用某个特定协议(IPv4 或 IPv6)。

对 UDP 套接口没有设置 SO_REUSEADDR 选项，因为就像 7.5 节中介绍的，在支持多播的主机上，这个套接口选项能允许多个套接口绑定同一个 UDP 端口。对 UDP 套接口来说，没有 TCP 那样的 TIME_WAIT 状态，所以启动服务器时不需要设置这个套接口选项。

例子：独立于协议的时间/日期服务器

图 11.15 给出了我们修改图 11.10 以使用 UDP 的时间/日期服务器程序。

```
1 #include    "unp.h"
2 #include    <time.h>
3 int
4 main(int argc, char **argv)
5 {
6     int             sockfd;
7     ssize_t         n;
8     char            buff[MAXLINE];
9     time_t          ticks;
10    socklen_t       addrlen, len;
11    struct sockaddr  *cliaddr;
12    if (argc == 2)
13        sockfd = Udp_server(NULL, argv[1], &addrlen);
14    else if (argc == 3)
15        sockfd = Udp_server(argv[1], argv[2], &addrlen);
16    else
```

```
17          err_quit("usage: daytimeudpsrv [ <host> ] <service or port>");
18      cliaddr = Malloc(addrlen);
19      for ( ; ; ) {
20          len = addrlen;
21          n = Recvfrom(sockfd, buff, MAXLINE, 0, cliaddr, &len);
22          printf("datagram from %s\n", Sock_ntop(cliaddr, len));
23        ticks = time(NULL);
24        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
25        Sendto(sockfd, buff, strlen(buff), 0, cliaddr, len);
26      }
27  }
```

图 11.15　独立于协议的 UDP 时间/日期服务器程序[names/daytimeudpsrv2.c]

11.13　getnameinfo 函数

这个函数与 getaddrinfo 互补：它以一个套接口地址为参数，返回一个描述主机的字符串和一个描述服务的字符串。这个函数以一种独立于协议的方式提供这些信息；也就是说，调用者不必关心在套接口地址结构中的协议地址的类型，这些细节由函数自己处理。

```
#include <netdb.h>
int getnameinfo(const struct sockaddr *sockaddr,socklen_t addrlen,
                char *host,size_t hostlen,
                char *serv,size_t servlen,int flags);
                                        返回：成功返回 0，出错返回－1
```

sockaddr 指向包含协议地址的套接口地址结构，它将会被转换成可读的字符串，addrlen 是结构的长度。这个结构及其长度通常由 accept、recvfrom、getsockname 或 getpeername 返回。

调用者给两个直观可读的字符串分配空间：host 和 hostlen 指定主机字符串，serv 和 servlen 指定服务字符串。如果调用者不想返回主机字符串，将 hostlen 设为 0 即可。同样将 servlen 设为 0 就指定不返回服务的信息。为了给这两个字符串分配空间，图 11.6 给出了在 <netdb.h>中定义的常值。

常值	描述	值
NI_MAXHOST	返回的主机字符串的最大长度	1025
NI_MAXSERV	返回的服务字符串的最大长度	32

图 11.16　getnameinfo 返回的字符串的大小所用的常值

sock_ntop 和 getnameinfo 的差别在于，前者不查 DNS 直接返回可输出的 IP 地址和端口号，而后者通常试图给主机和服务查到一个名字。

图 11.17 中给出了五个可设置的标志，它们能改变 getnameinfo 函数的操作。

常值	描述
NI_DGRAM	数据报服务
NI_NAMEREQD	不能从地址反向解析到名字时返回错误
NI_NOFQDN	只返回FQDN的主机名部分
NI_NUMERICHOST	返回主机的数值格式串
NI_NUMERICSERV	返回服务的数值格式串

图11.17 getnameinfo所用的标志

当调用者知道它处理的是一个数据报套接口时应设置NI_DGRAM标志。原因是若在套接口地址结构中只给出IP地址和端口号的话，getnameinfo不能确定使用的协议(TCP或UDP)。在好几个端口号上使用的TCP服务和UDP服务截然不同。端口514就是一个例子，用TCP的服务是rsh，用UDP的服务是syslog。

如果用DNS不能反向解析到主机名，NI_NAMEREQD会导致返回一个错误。那些要求必须将客户方的IP地址映射到一个主机名的服务器可以使用它。这些服务器用返回的主机名调用gethostbyname，以验证返回的地址与套接口地址结构中的地址相同。

NI_NOFQDN使返回的主机名被截成第一个点号前部分。举例来说，如果在套接口地址结构中的IP地址为206.62.226.42，gethostbyaddr将返回主机名alpha.kohala.com。但要是设置了这个标志，返回的主机名是alpha。

NI_NUMERICHOST指定getnameinfo不调用DNS(这将花费一定的时间)，相反可能会调用inet_ntop以返回IP地址的数值表示。与此类似，NI_NUMERICSERV指定返回以十进制数表示的端口号，代替反向解析的服务名。由于客户的端口号一般没有对应的服务名——它们是临时的端口，因此服务器通常应设置NI_NUMERICSERV标志。

如果各标志的组合有意义的话(如NI_DGRAM和NI_NUMERICHOST)，可以通过逻辑或同时设置多个标志，但有些组合没有意义(如NI_NAMEREQD和NI_NUMERICHOST)。

> 在Posix.1g中没有getnameinfo，但它在RFC 2133[Gilligan et al. 1997]中有说明。

11.14 可重入函数

9.3节中的gethostbyname函数提出了一个我们目前还没有讨论过的问题：它是不可重入的。在23章中处理线程时会很普遍地遇到这个问题，但现在(不用涉及线程的概念)来讨论并解决这个问题也是十分有趣的。

首先让我们来看看这个函数是如何工作的。如果看一下它的源码(这很容易，因为整个BIND版本的源码都是公开的)，我们可以发现一个包含gethostbyname和gethostbyaddr的文件，这个文件的内容大致如下：

```
static struct hostent host;     /* result stored here */

struct hostent *
gethostbyname(const char *hostname)
{
    return(gethostbyname2(hostname,family));     /* Figure 9.6 */
}

struct hostent *
gethostbyname2(const char *hostname,int family)
{
    /* call DNS functions for A or AAAA query */
    /* fill in host structure */
    return(&host);
}

struct hostent *
gethostbyaddr(const char *addr,size_t len,int family)
{
    /* call DNS functions for PTR query in in-addr.arpa domain */
    /* fill in host structure */
    return(&host)
}
```

我们特别指出了结果结构的 static 存储类型，因为这正是问题之所在。这三个函数共用同一个 host 变量的事实还引起我们在习题 9.1 中讨论过的另一个问题。（回想图 9.6，gethostbyname2 是 BIND 4.9.4 为支持 IPv6 而新引入的。我们将忽略当调用 gethostbyname 时实际上也会调用 gethostbyname2 的事实，因为这并不会影响下面的讨论。）

在一个 UNIX 进程的主控制流程中调用 gethostbyname 或 gethostbyaddr，并在信号处理程序中也调用了这些函数，就可能发生重入问题。当调用信号处理程序时（假设每秒产生一个 SIGALRM 信号），进程的主控制流程会暂停而去调用信号处理程序。考虑一下下面的情况。

```
main()
{
    struct hostent *hptr;
    ...
    signal(SIGALRM,sig_alrm);
    ...
    hptr = gethostbyname( ... );
    ...
}

void
sig_alrm(int signo)
{
    struct hostent *hptr;
    ...
    hptr = gethostbyname( ... );
    ...
}
```

如果主控制流程暂停时正执行到 gethostbyname 的中间(在这个函数已经填好了 host 变量正要返回时),信号处理程序调用 gethostbyname,因为在进程中 host 变量只有一份,所以它被重用了。这就重写了 host 变量的值。

如果看看这一章和第 9 章里的名字和地址转换函数,以及第 4 章里的 inet_XXX 函数,我们注意到以下几点:

- 历史上,gethostbyname、gethostbyname2、gethostbyaddr、getservbyname 和 getservbyport 是不可重入的,因为它们都返回指向一个静态结构的指针。
  一些支持线程的实现(Solaris 2.x)提供了这四个函数的可重入版本,函数的名字以_r 结尾,我们会在下一节中进行介绍。
  也有些支持线程的实现(Digital Unix 4.0 和 HP-UX 10.30)提供这些函数的使用线程特定数据的可重入版本。
- inet_pton 和 inet_ntop 总是可重入的。
- 历史上 inet_ntoa 是不可重入的,但一些支持线程的实现提供了使用线程特定数据的可重入版本。
- getaddrinfo 只有在它自己调用的是可重入函数时才是可重入的;这就是说,它要调用可重入版本的 gethostbyname(解析主机名)和 getservbyname(解析服务名)。动态分配存放结果的内存空间的原因之一就是这样可以重入。
- getnameinfo 只有在它自己调用的是可重入函数时才是可重入的;这就是说,它要调用可重入版本的 gethostbyaddr 和 getservbyport 以获取主机名和服务名。注意所有的结果字符串(主机名和服务名)是由调用者分配的,以允许重入。

errno 变量有着同样的问题。历史上这个整型变量每个进程中只有一份拷贝。如果某进程作了一个系统调用且返回错误,一个整型的错误码将存放在这个变量里。举例来说,标准 C 库中的 close 函数被调用时,它执行的操作可能如下:

- 将系统调用的参数(一个整型的描述字)放入一个寄存器
- 将另一个寄存器置为某值表示正在调用 close 系统调用
- 进行系统调用(用特殊指令切换到核心态)
- 测试寄存器的值,以确定是否发生错误
- 如果没有错误,retrun(0)
- 将某个寄存器的值存入 errno
- return(-1)

首先要注意的是如果没有发生错误的话,errno 的值不会改变。因而除非知道发生了错误(通常函数用返回-1 来指出),否则不去看 errno 的值。

假设程序测试 close 函数的返回值,如果发生错误则输出 errno 的值,如下:

```
if (close (fd) < 0) {
    fprintf(stderr,"close error,errno = %d\n", errno)
    exit(1);
}
```

系统调用返回时将错误码存到 errno,该操作和在程序中输出这个值之间有一个小的时

间窗，这期间同一进程内的另一执行流程(即信号处理程序)可能改变了 errno 的值。举例来说，如果当调用信号处理程序时，主控制流程正处于 close 和 fprintf 之间，而且信号处理程序调用了一些其他系统调用(譬如 write)并返回了错误，那么 write 系统调用产生的 errno 值就会覆盖 close 调用的 errno 值。

考虑这两个与信号处理程序相关的问题，对 gethostbyname 产生的问题(返回一个指向静态变量的指针)的一种解决办法是不在信号处理程序中调用不可重入函数。errno 变量的问题(单个全局变量的值可能被信号处理程序改变)可通过在编码信号处理程序时保存和恢复 errno 的值来避免，如下所示：

```
void
sig_alrm(int signo)
{
    int     errno_save;

    errno_save = errno;     /* save its value on entry */
    if (write( ... ) != nbytes)
        fprintf(stderr,"write error,errno = %d\n",errno);
    errno = errno_save;     /* restore its value on return */
}
```

这段代码，在信号处理程序中也调用了 fprintf，它是一个标准的 I/O 函数。因为许多版本的标准 I/O 库是不可重入的，所以这也会导致重入问题：在信号处理程序中不应调用标准 I/O 函数。

我们在第 23 章中还会重提这个重入问题，并阐述线程怎样来处理 errno 变量的问题。下一节中介绍一些可重入版本的主机名转换函数。

11.15　gethostbyname_r 和 gethostbyaddr_r 函数

有两种方法可将像 gethostbyname 这样的不可重入函数变成可重入函数。

1. 变由函数填写并返回一个静态结构为：由调用者分配结构所需的空间，由可重入函数来填写。这是从不可重入的 gethostbyname 到可重入的 gethostbyname_r 所使用的技术。但这种方法也会使问题的解决变得更为复杂。因为 hostent 结构还指向其他的一些信息，包括规范名字、别名指针数组、别名串、地址指针数组和地址(见图 9.2)，所以调用者仅提供 hostent 结构的空间是不行的，它还必需提供一个大的缓冲区存放附加的信息，填写 hostent 结构时其中的一些指针就指向该缓冲区内。结果这个函数至少得加 3 个参数：一个指向要填写的 hostent 结构的指针、一个指向用来存放其他信息的缓冲区的指针以及缓冲区的大小。由于不能再用全局整型变量 h_errno，所以还需要第四个参数，即一个指向整型的指针以存放错误码。(全局变量 h_errno 存在与 errno 同样的重入问题。)

   getnameinfo 和 inet_ntop 也使用了这种技术。
2. 可重入函数调用 malloc 动态分配内存。getaddrinfo 用的是这种技术。这种方法的问题是调用这个函数的应用程序还要调用 freeaddrinfo 来释放所用的内存。如果不这样做就会导致内存漏损，也就是说进程每调用一次这样的函数，它的内存使用量就

增加。如果这个进程长时间运行(如网络服务器),使用的内存就随时间而不断增加。

现在讨论以下 Solaris 2.x 中名字到地址和地址到名字转换的可重入函数。

```
#include <netdb.h>

struct hostent *gethostbyname_r(const char *hostname,
                                struct hostent *result,
                                char *buf,int buflen,int *h_errnop);

struct hostent *gethostbyaddr_r(const char *addr,int len, int type,
                                struct hostent *result,
                                char *buf,int buflen,int *h_errnop);

                                返回:成功都返回非空指针,出错返回 NULL
```

每个函数都需要四个附加的参数。result 是由调用者分配的 hostent 结构,由函数填写。成功时这个指针也是函数的返回值。

buf 是由调用者分配的缓冲区,buflen 是它的大小。缓冲区将存放规范主机名、别名指针、别名串、地址指针和实际地址。result 指向的结构中的所有指针都指到这个缓冲区中。这个缓冲区到底应该有多大呢?不幸的是基本上所有的手册页面都说得很含糊,类似于"缓冲区要大到能放下所有与该主机相关的数据"。gethostbyname 当前的实现内部使用 8192 字节的缓冲区存放别名和地址,最多能返回 35 个别名指针,35 个地址指针。因此 8192 字节的缓冲区应该是足够的。

如果出错,错误码通过 h_errnop 指针返回,而不是通过全局变量 h_errno。

遗憾的是这个重入问题比它表面看来要严重。首先,没有一个关于 gethostbyname 和 gethostbyaddr 的重入问题的标准。Posix.1g 对这两个函数作了说明,但没有提到线程安全性问题。Unix 98 只是说这两个函数不是线程安全的。其次,没有关于_r 函数的标准。这一节中展示的两个_r 函数(作为例子)是由 Solaris 2.x 提供的。而 Digital Unix 4.0 和 HP-UX 10.30 中的这些函数所带的参数与 Solaris 不同,gethostbyname_r 函数的前两个参数与 Solaris 的是一样的,但把 Solaris 版本中的后三个参数合成一个新的 hostent_data 结构(由调用者分配),并以指向这个结构的指针作为第三个,也是最后一个参数。Digital Unix 4.0 和 HP-UX 10.30 中的普通 gethostbyname 和 gethostbyaddr 函数使用了线程特定数据(见 23.5 节),是可重入的。Solaris 2.x 中_r 函数的开发历史可见于[Maslen 1997]。

最后,虽然 gethostbyname 的可重入版本能在不同的线程同时调用它时提供安全性,但一点也没有提及在它底层的名字解析函数的重入性。在本书写作时,BIND 中的名字解析函数还是不可重入的。

11.16　getaddrinfo 和 getnameinfo 函数的实现

现在来看看 getaddrinfo 和 getnameinfo 的一个实现。在前者的实现中我们可以进一步了解它的操作细节。这个实现也支持在 11.5 节中提到的 Unix 域套接口。

> 注意：在本节中，所有依赖于 IPv4、IPv6 或 Unix 域支持的相应部分的代码，都用 #ifdef 和 #endif 包了起来，对应的条件编译常值为：IPV4、IPV6 和 UNIX-DOMAIN。这样可使这个程序能在支持这三种协议的任意组合的系统上进行编译。但是这里删去了这些预处理器语句，因为这些语句对我们的讨论没有什么帮助，而且使源程序难于理解。
>
> 我们也注意到在第 14 章前还没有了解 Unix 域套接口的细节。

图 11.18 展现了 getaddrinfo 调用的函数，它们都由 ga_开头。

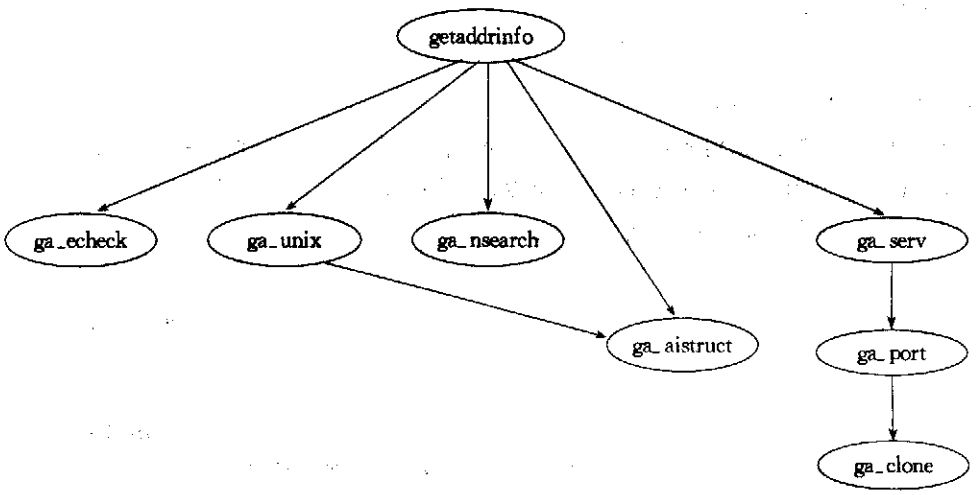


图 11.18　getaddrinfo 实现中调用的函数

第一个文件是在图 11.19 中的 gai_hdr.h 头文件，我们所有的源文件都包含它。

这个头文件包含 udp.h 和另外一个头文件。我们马上会看到 AI_CLONE 标志和 search 结构的使用。它的其余部分定义了本节中各个函数的原型。

```
1 #include    "unp.h"
2 #include    <ctype.h>     /* isxdigit(), etc. */
3           /* following internal flag cannot overlap with other AI_xxx flags */
4 #define AI_CLONE      4     /* clone this entry for other socket types */

5 struct search {
6     const char *host;         /* hostname or address string */
7     int       family;         /* AF_xxx */
8 };

9           /* function prototypes for our own internal functions */
10 int     ga_aistruct(struct addrinfo ***, const struct addrinfo *,
11                     const void *, int);
12 struct addrinfo *ga_clone(struct addrinfo *);
```

```
13 int    ga_echeck(const char *, const char *, int, int, int, int);
14 int    ga_nsearch(const char *, const struct addrinfo *, struct search *);
15 int    ga_port(struct addrinfo *, int, int);
16 int    ga_serv(struct addrinfo *, const struct addrinfo *, const char *);
17 int    ga_unix(const char *, struct addrinfo *, struct addrinfo **);
18 int    gn_ipv46(char *, size_t, char *, size_t, void *, size_t,
19                 int, int, int);
```

图 11.19 gai_hdr.h 头文件[libgai/gai_hdr.c]

图 11.20 给出了 getaddrinfo 函数的第一部分。

定义 error 宏

第 13～17 行 在这个函数中有多处，如果遇到出错，就释放所有已分配的内存并返回相应的返回码。为了简化代码，我们定义了这个宏，把返回码存到变量 error 中，并转到函数结尾处的 bad 标记处(图 11.26)。

初始化自动变量

第 18～20 行 初始化一些自动变量。在图 11.34 中我们描绘了 aihead 和 aipnext 指针。

拷贝调用者的 hints 结构

第 21～25 行 如果调用者提供了一个 hints 结构，我们就把它拷贝到局部变量，这样在后面可以修改它。否则从一个全零的结构开始，不过 ai_family 被初始化为 AF_UNSPEC。后者一般定义为 0，但这不是 Posix.1g 要求的。

检查参数

第 26～29 行 调用图 11.39 中给出的 ga_echeck 函数，对一些参数进行检查。

检查 Unix 域路径名

第 30～34 行 如果主机名为/local 或/unix，而且服务名以斜杠开头，就把这个参数作为 Unix 域路径名来处理。函数 ga_unix(图 11.33)处理这种路径名。

```
 1 #include    "gai_hdr.h"
 2 #include    <arpa/nameser.h>   /* needed for <resolv.h> */
 3 #include    <resolv.h>         /* res_init,  _res */
 4 int
 5 getaddrinfo(const char *hostname, const char *servname,
 6             const struct addrinfo *hintsp, struct addrinfo **result)
 7 {
 8     int       rc, error, nsearch;
 9     char    **ap, *canon;
10     struct hostent *hptr;
11     struct search search[3], *sptr;
12     struct addrinfo hints, *aihead, **aipnext;
13     /*
14      * If we encounter an error we want to free() any dynamic memory
15      * that we've allocated.   This is our hack to simplify the code.
16      */
17 #define     error(e) { error = (e); goto bad; }
18     aihead = NULL;          /* initialize automatic variables */
19     aipnext = &aihead;
```

```
20      canon = NULL;
21      if (hintsp == NULL) {
22          bzero(&hints, sizeof(hints));
23          hints.ai_family = AF_UNSPEC;
24      } else
25          hints = *hintsp;          /* struct copy */
26          /* first some basic error checking */
27      if ( (rc = ga_echeck (hostname, servname, hints.ai_flags, hints.ai_family,
28                            hints.ai_socktype, hints.ai_protocol)) != 0)
29          error(rc);
30          /* special case Unix domain first */
31      if (hostname != NULL &&
32          (strcmp(hostname, "/local") == 0 || strcmp(hostname, "/unix") == 0) &&
33          (servname != NULL && servname[0] == '/'))
34          return(ga_unix(servname, &hints, result));
```

图 11.20　getaddrinfo 函数:第一部分,初始化[libgai/getaddrinfo.c]

getaddrinfo 函数剩下的部分(在图 11.24 中继续)处理 IPv4 和 IPv6 套接口。函数 ga_nsearch 的第一部分在图 11.21 中给出,计算查找一个主机名的次数。如果调用者指定的地址族是 AF_INET 或 AF_INET6,那么只查找一次主机名。但如果地址族是未指定类型即 AF_UNSPEC,就要做两次查找:一次 IPv6,一次 IPv4。我们把这个函数分成三部分:

- 没有主机名,但指定了 AI_PASSIVE
- 没有主机名,也没有指定 AI_PASSIVE(也就是主动)
- 指定了主机名

这三部分对应于图 11.4 中的三个主要部分。

```
6 int
7 ga_nsearch(const char *hostname, const struct addrinfo *hintsp,
8             struct search *search)
9 {
10      int             nsearch = 0;
11      if (hostname == NULL || hostname[0] == '\0') {
12          if (hintsp->ai_flags & AI_PASSIVE) {
13                  /* no hostname and AI_PASSIVE: implies wildcard bind */
14              switch (hintsp->ai_family) {
15              case AF_INET:
16                  search[nsearch].host = "0.0.0.0";
17                  search[nsearch].family = AF_INET;
18                  nsearch++;
19                  break;
20              case AF_INET6:
21                  search[nsearch].host = "0::0";
22                  search[nsearch].family = AF_INET6;
23                  nsearch++;
24                  break;
25              case AF_UNSPEC:
26                  search[nsearch].host = "0::0";          /* IPv6 first, then IPv4 */
27                  search[nsearch].family = AF_INET6;
28                  nsearch++;
```

```
29              search[nsearch].host = "0.0.0.0";
30              search[nsearch].family = AF_INET;
31              nsearch++;
32              break;
33          }
```

图 11.21 ga_nsearch 函数：没有主机名，但指定了 AI_PASSIVE[libgai/ga_nsearch.c]

没有主机名，被动的套接口

第 11～33 行 如果调用者没有指定一个主机名，不过指定了 AI_PASSIVE，那么返回用于创建一个或多个捆绑到通配地址的被动套接口的信息。switch 是根据地址族转移的：IPv4 套接口需要捆绑 0.0.0.0(INADDR_ANY)，而 IPv6 套接口需要捆绑 0::0(IN6ADDR_ANY_INIT)。如果地址族为 AF_UNSPEC，则必须返回用于创建两个套接口的信息：第一个是 IPv6 的，第二个是 IPv6 的。把 IPv6 排在第一，然后才轮到 IPv4，是因为在一个双重协议栈的主机上，IPv6 套接口可以处理 IPv6 和 IPv4 两种客户。在这种情况下，如果调用者只从返回的 addrinfo 结构表中创建一个套接口，它应该是 IPv6 套接口。

这个函数创建一个 search 结构的数组(图 11.19)，其中的每项指明查找的主机名和地址族。指向调用者的 search 结构数组的指针是这个函数的最后一个参数。返回值是创建的这些结构的数目，它总是 1 或 2。

这个函数的下一部分在图 11.22 中给出，这部分处理没有主机名和未设置 AI_PASSIVE 的情况，这意味着调用者想要在本机上创建一个主动的套接口。

```
34      } else {
35              /* no host and not AI_PASSIVE: connect to local host */
36          switch (hintsp->ai_family) {
37          case AF_INET:
38              search[nsearch].host = "localhost"; /* 127.0.0.1 */
39              search[nsearch].family = AF_INET;
40              nsearch++;
41              break;
42          case AF_INET6:
43              search[nsearch].host = "0::1";
44              search[nsearch].family = AF_INET6;
45              nsearch++;
46              break;
47          case AF_UNSPEC:
48              search[nsearch].host = "0::1";      /* IPv6 first, then IPv4 */
49              search[nsearch].family = AF_INET6;
50              nsearch++;
51              search[nsearch].host = "localhost";
52              search[nsearch].family = AF_INET;
53              nsearch++;
54              break;
55          }
56      }
```

图 11.22 ga_nsearch 函数：没有主机名而且没有设置 AI_PASSIVE[libgai/ga_nsearch.c]

第 34～56 行 对 IPv4 我们假定主机名 localhost 将返回回馈地址，一般是 127.0.0.1。在 IPv6 中对于本地机没有一个通用的主机名，因此返回回馈地址 0::1。跟被动情况一样，

如果没有指定地址族就返回两个结构：第一个是 IPv6 的，另一个是 IPv4 的。

图 11.23 给出了这个函数的最后一部分，它是最初的 if 语句的 else 子句。在指定了主机名时执行这段代码。

第 57～82 行　在这种情况下 AI_PASSIVE 标志无关紧要；需要查找主机名。如果调用者创建一个被动套接口，返回的套接口地址结构将用在 bind 调用中，但如果调用者创建一个主动的套接口，套接口地址结构将用于 connect 调用。我们创建一个或两个 search 结构：指定了地址族时是一个，没有指定地址族时是两个。跟前两种情况一样，如果返回两个地址结构，第一个是 IPv6 的，第二个是 IPv4 的。

现在回到 getaddrinfo 函数，在图 11.24 中，以一个 ga_nsearch 调用开始。

```
57    } else {      /* host is specified */
58        switch (hintsp->ai_family) {
59        case AF_INET:
60            search[nsearch].host = hostname;
61            search[nsearch].family = AF_INET;
62            nsearch++;
63            break;
64        case AF_INET6:
65            search[nsearch].host = hostname;
66            search[nsearch].family = AF_INET6;
67            nsearch++;
68            break;
69        case AF_UNSPEC:
70            search[nsearch].host = hostname;
71            search[nsearch].family = AF_INET6;   /* IPv6 first */
72            nsearch++;
73            search[nsearch].host = hostname;
74            search[nsearch].family = AF_INET;    /* then IPv4 */
75            nsearch++;
76            break;
77        }
78    }
79    if (nsearch < 1 || nsearch > 2)
80        err_quit("nsearch = %d", nsearch);
81    return(nsearch);
82 }
```

图 11.23　ga_nsearch 函数：指定了主机名[libgai/ga_nsearch.c]

调用 ga_nsearch

第 36 行　调用 ga_nsearch 函数，填写 search 数组，返回数组中的结构数目：一个或两个。

遍历全部 search 结构

第 37 行　遍历每个由 ga_nsearch 创建的 search 结构。

检查 IPv4 点分十进制数串

第 39～44 行　如果主机名的第一个字符是一个数字，则检查主机名是不是一个点分十进制数串。调用 inet_pton 做这项检查和转换。如果调用成功了但调用者指定的地址族不是

AF_INET，则出错。

第45～46行 我们要检查search结构的family是不是也是AF_INET，但在这里出现不匹配只是使这个search结构被忽略。这种情况有可能发生，譬如，当调用者指定一个主机名192.3.4.5但没有指定地址族时。ga_nsearch为它创建两个search结构：一个是IPv6的，一个是IPv4的。第一次for循环时，inet_pton调用成功，但因为search结构的family是AF_INET6，我们想忽略这个结构，而不发生错误。

```
35          /* remainder of function for IPv4/IPv6 */
36      nsearch = ga_nsearch(hostname, &hints, &search[0]);
37      for (sptr = &search[0]; sptr < &search[nsearch]; sptr++) {
38              /* check for an IPv4 dotted-decimal string */
39          if (isdigit(sptr->host[0])) {
40              struct in_addr      inaddr;
41                  if (inet_pton(AF_INET, sptr->host, &inaddr) == 1) {
42                      if (hints.ai_family != AF_UNSPEC &&
43                          hints.ai_family != AF_INET)
44                          error(EAI_ADDRFAMILY);
45                      if (sptr->family != AF_INET)
46                          continue;           /* ignore */
47                      rc = ga_aistruct(&aipnext, &hints, &inaddr, AF_INET);
48                      if (rc != 0)
49                          error(rc);
50                      continue;
51              }
52          }
53              /* check for an IPv6 hex string */
54          if ((isxdigit(sptr->host[0]) || sptr->host[0] == ':') &&
55              (strchr(sptr->host, ':') != NULL)) {
56              struct in6_addr     in6addr;
57              if (inet_pton(AF_INET6, sptr->host, &in6addr) == 1) {
58                  if (hints.ai_family != AF_UNSPEC &&
59                      hints.ai_family != AF_INET6)
60                      error(EAI_ADDRFAMILY);
61                  if (sptr->family != AF_INET6)
62                      continue;       /* ignore */
63                  rc = ga_aistruct(&aipnext, &hints, &in6addr, AF_INET6);
64                  if (rc != 0)
65                      error(rc);
66                  continue;
67              }
68          }
```

图11.24 getaddrinfo函数：检查IPv4或IPv6地址串[libgai/getaddrinfo.c]

创建addrinfo结构

第47～52行 ga_aistruct函数创建一个addrinfo结构并把它加到正在建立的链表中(aipnext指针)。

检查IPv6地址串

第53～60行 如果主机名的第一个字符是十六进制的数字或冒号，并且字符串中有一

个冒号，则调用 inet_pton 检查主机名是不是一个 IPv6 地址串。如果调用成功但调用者指定了一个不是 AF_INET6 的地址族，则出错。

第 61～62 行　检查 search 结构的地址族是不是也是 AF_INET6，但在这里不匹配只是导致这个 search 结构被忽略。

创建 addrinfo 结构

第 63～68 行　函数 ga_aistruct 创建一个 addrinfo 结构并将其加到已有的链表中。

在循环(图 11.24)中的前两个条件测试处理 IPv4 点分十进制数串或 IPv6 地址串。循环剩余的部分，见图 11.25，调用 gethostbyname 和 gethostbyname2 查找主机名。

初始化名字解析器

第 70～71 行　如果还没有调用过名字解析器的 res_init 函数，就调用它进行初始化。

如果要进行两次查找就调用 gethostbyname2

第 72～74 行　如果 nsearch 为 2，就进行两次 for 循环：一次 IPv6 和一次 IPv4。如果主机名参数只有一个某种协议的地址，我们希望只返回这个地址。举例来说，9.2 节中的主机 solaris 在 DNS 中有一个 AAAA 记录和一个 A 记录。第一次循环希望找到 AAAA 记录，第二次循环希望找到 A 记录。但如果主机只有一个 A 记录，就不需要在第一次循环时处理这个记录，因为第一次循环时 search 结构中的 family 成员为 AF_INET6。这就是说，既然知道将为该主机查找一个 A 记录，我们就不再用 gethostbyname 查找 AAAA 记录，因为这可能会返回一个与该 A 记录对应的 IPv4 映射的 IPv6 地址。看一下图 9.5，当地址族为 AF_INET 时只查找 A 记录、地址族为 AF_INET6 时只查找 AAAA 记录的方法，就是用 gethostbyname2 代替 gethostbyname，同时需将 RES_USE_INET6 选项关闭。

如果只进行一次查找就调用 gethostbyname

第 75～81 行　如果只进行一次查找，就调用 gethostbyname，当地址族为 AF_INET6 时设置 RES_USE_INET6 选项，当地址族为 AF_INET 时清除 RES_USE_INET6 选项。举例来说，如果调用者查找的主机名只有一个 A 记录，但指定的地址族为 AF_INET6，将返回对应的 IPv4 映射的 IPv6 地址。

处理名字解析失败的情况

第 82～97 行　如果名字解析失败，但 nsearch 值为 2，这不算错误，因为某一次循环可能会成功。(在循环结束时要检查是否至少返回一个 addrinfo 结构。)但要是这是唯一的一次调用解析器函数，则需根据解析器的 h_errno 返回相应的错误。

检查地址族是否匹配

第 98～100 行　如果调用者指定了一个地址族，但名字解析返回的地址族与此不符，则返回错误。

保存主机名

第 101～106 行　如果调用者指定了主机名并设置了 AI_CANNONAME 标志，则将由解析器返回的第一个名字保存下来。(回想图 11.22，即使没有指定主机名，也用 localhost 这个名字调用解析器。)复制由解析器返回的字符串，将它的指针存到 canon 中。

```
69              /* remainder of for() to look up hostname */
70          if ((_res.options & RES_INIT) == 0)
71              res_init();             /* need this to set _res.options */
72          if (nsearch == 2) {
73              _res.options &= ~RES_USE_INET6;
74              hptr = gethostbyname2(sptr->host, sptr->family);
75          } else {
76              if (sptr->family == AF_INET6)
77                  _res.options |= RES_USE_INET6;
78              else
79                  _res.options &= ~RES_USE_INET6;
80              hptr = gethostbyname(sptr->host);
81          }
82          if (hptr == NULL) {
83              if (nsearch == 2)
84                  continue;   /* failure OK if multiple searches */
85              switch (h_errno) {
86              case HOST_NOT_FOUND:
87                  error(EAI_NONAME);
88              case TRY_AGAIN:
89                  error(EAI_AGAIN);
90              case NO_RECOVERY:
91                  error(EAI_FAIL);
92              case NO_DATA:
93                  error(EAI_NODATA);
94              default:
95                  error(EAI_NONAME);
96              }
97          }
98              /* check for address family mismatch if one specified */
99           if (hints.ai_family != AF_UNSPEC && hints.ai_family != hptr->h_
               addrtype)
100              error(EAI_ADDRFAMILY);
101             /* save canonical name first time */
102         if (hostname != NULL && hostname[0] != '\0' &&
103             (hints.ai_flags & AI_CANONNAME) && canon == NULL) {
104             if ( (canon = strdup(hptr->h_name)) == NULL)
105                 error(EAI_MEMORY);
106         }
107             /* create one addrinfo() for each returned address */
108         for (ap = hptr->h_addr_list; *ap != NULL; ap++) {
109             rc = ga_aistruct(&aipnext, &hints, *ap, hptr->h_addrtype);
110             if (rc != 0)
111                 error(rc);
112         }
113     }
114     if (aihead == NULL)
115         error(EAI_NONAME);              /* nothing found */
```

图 11.25 getaddrinfo 函数：查找主机名[libgai/getaddrinfo.c]

为每一个地址创建一个 addrinfo 结构

第 107～112 行　为每一个在 h_addr_list 数组中、由解析器返回的地址，调用我们的 ga_aistruct 函数创建一个 addrinfo 结构，并加到已建立的结构链表中。

检查是否没有任何匹配

第 114～115 行　如果 addrinfo 结构链表的头仍为空指针，那么 for 循环的每次迭代都失败了。

图 11.26 是 getaddrinfo 函数的最后一部分。

```
116         /* return canonical name */
117     if (hostname != NULL && hostname[0] != '\0' &&
118         hints.ai_flags & AI_CANONNAME) {
119         if (canon != NULL)
120             aihead->ai_canonname = canon;      /* strdup'ed earlier */
121         else {
122             if ( (aihead->ai_canonname = strdup(search[0].host)) == NULL)
123                 error(EAI_MEMORY);
124         }
125     }
126         /* now process the service name */
127     if (servname != NULL && servname[0] != '\0') {
128         if ( (rc = ga_serv(aihead, &hints, servname)) != 0)
129             error(rc);
130     }
131     *result = aihead;      /* pointer to first structure in linked list */
132     return(0);
133 bad:
134     freeaddrinfo(aihead);   /* free any alloc'ed memory */
135     return(error);
136 }
```

图 11.26　getaddrinfo 函数：处理服务名[libgai/getaddrinfo.c]

返回规范主机名

第 116～125 行　如果调用者指定了主机名并设置了 AI_CANONNAME 标志，而且在 canon 指针中保存了名字的一个拷贝，那么这个指针会在第一个 addrinfo 结构的 ai_canonname 成员中返回。如果解析器没有找到名字(可能主机名是一个地址串)，则返回主机名参数的一个拷贝作为替代品。

处理服务名

第 126～130 行　如果调用者指定了服务名，就调用我们的 ga_serv 函数对其进行处理。

返回指向链表的指针

第 131～132 行　返回的指针指向已建立的 addrinfo 结构链表的头，函数本身返回 0。

出错返回

第 133～135 行　如果遇到错误，则调用 freeaddrinfo 释放所有已分配的内存，并返回一

个 EAI_xxx 的错误值。

图 11.26 中用来处理服务名参数的 ga_serv 函数，在图 11.27 中给出了它的源代码。

检查端口号串

第 12～27 行 如果服务名的第一个字符为数字，我们就认为服务名是一个端口号并调用 atoi 将其转成二进制值。如果调用者指定了套接口类型(SOCK_STREAM 或 SOCK_DGRAM)，就为该套接口类型调用一次 ga_port 函数。但如果没有指定套接口类型，则调用两次 ga_port 函数，一次给 TCP，一次给 UDP(回想图 11.2。)。这里有一个计数器记录 ga_port 返回成功的次数，只有在函数结束时其值仍为 0 时才返回错误。

尝试按 TCP 调用 getservbyname

第 28～36 行 如果没有指定套接口类型，或指定了一个 TCP 类型的套接口，调用 getservbyname 时第二个参数就置为“tcp”。成功的话，就调用 ga_port 函数。失败也没关系，因为这个服务名可能对 UDP 有效。

尝试以 UDP 为参数调用 getservbyname

第 37～44 行 如果没有指定套接口类型，或指定了一个 UDP 类型的套接口，调用 getservbyname 时第二个参数就置为“udp”。成功的话，就调用 ga_port 函数。

错误检查

第 45～51 行 如果 nfound 计数器不为 0，则返回成功。否则返回错误。

在图 11.27 中每找到一个端口号时都要调用 ga_port 函数(详细代码见图 11.28)。

遍历所有的 addrinfo 结构

第 33 行 遍历所有由图 11.24 和 11.25 中的 ga_aistruct 调用创建的 addrinfo 结构。当调用者不指定套接口类型时，ga_aistruct 总是设置 AI_CLONE 标志。这表示这个 addrinfo 结构可能需要为 TCP 和 UDP 各复制一份。

检查 AI_CLONE 标志

第 34～42 行 如果设置了 AI_CLONE 标志而且套接口类型不为 0，将用 ga_clone 函数复制出另一个与之完全相同的 addrinfo 结构。我们很快就会给出这样的例子。

设置套接口地址结构中的端口号

第 44～47 行 设置套接口地址结构中的端口号，并增加计数器 nfound 的值。

```
 5 int
 6 ga_serv(struct addrinfo *aihead, const struct addrinfo *hintsp,
 7         const char *serv)
 8 {
 9     int             port, rc, nfound;
10     struct servent      *sptr;
11     nfound = 0;
12     if (isdigit(serv[0])) {     /* check for port number string first */
13         port = htons(atoi(serv));
14         if (hintsp->ai_socktype) {
15                 /* caller specifies socket type */
16             if ( (rc = ga_port(aihead, port, hintsp->ai_socktype)) < 0)
```

```
17                  return(EAI_MEMORY);
18              nfound += rc;
19          } else {
20                  /* caller does not specify socket type */
21              if ( (rc = ga_port(aihead, port, SOCK_STREAM)) < 0)
22                  return(EAI_MEMORY);
23              nfound += rc;
24              if ( (rc = ga_port(aihead, port, SOCK_DGRAM)) < 0)
25                    return(EAI_MEMORY);
26              nfound += rc;
27          }
28      } else {
29              /* try service name, TCP then UDP */
30          if (hintsp->ai_socktype == 0 || hintsp->ai_socktype == SOCK_STREAM)
            {
31              if ( (sptr = getservbyname(serv, "tcp")) != NULL) {
32                  if ( (rc = ga_port(aihead, sptr->s_port, SOCK_STREAM)) < 0)
33                      return(EAI_MEMORY);
34                  nfound += rc;
35              }
36          }
37          if (hintsp->ai_socktype == 0 || hintsp->ai_socktype == SOCK_DGRAM)
            {
38              if ( (sptr = getservbyname(serv, "udp")) != NULL) {
39                  if ( (rc = ga_port(aihead, sptr->s_port, SOCK_DGRAM)) < 0)
40                      return(EAI_MEMORY);
41                  nfound += rc;
42              }
43          }
44      }
45      if (nfound == 0) {
46          if (hintsp->ai_socktype == 0)
47              return(EAI_NONAME);      /* all calls to getservbyname() failed */
48          else
49              return(EAI_SERVICE);   /* service not supported for socket type */
50      }
51      return(0);
52  }
```

图 11.27　ga_serv 函数[libgai/ga_serv.c]

```
27 int
28 ga_port(struct addrinfo *aihead, int port, int socktype)
29              /* port must be in network byte order */
30 {
31      int                 nfound = 0;
32      struct addrinfo     *ai;
33      for (ai = aihead; ai != NULL; ai = ai->ai_next) {
34          if (ai->ai_flags & AI_CLONE) {
35              if (ai->ai_socktype != 0) {
36                  if ( (ai = ga_clone(ai)) == NULL)
37                      return(-1);          /* memory allocation error */
```

```
38                  /* ai points to newly cloned entry, which is what we want */
39              }
40          } else if (ai->ai_socktype != socktype)
41              continue;           /* ignore if mismatch on socket type */
42          ai->ai_socktype = socktype;
43          switch (ai->ai_family) {
44          case AF_INET:
45              ((struct sockaddr_in *) ai->ai_addr)->sin_port = port;
46              nfound++;
47              break;
48          case AF_INET6:
49              ((struct sockaddr_in6 *) ai->ai_addr)->sin6_port = port;
50              nfound++;
51              break;
52          }
53      }
54      return(nfound);
55  }
```

图 11.28 ga_port 函数[libgai/ga_port.c]

考虑一个例子。在图 11.1 里假定调用 getaddrinfo 时的参数是：一个有两个 IP 地址的主机，一个名为 domain 的服务(TCP 和 UDP 的 53 号端口)，没有指定套接口类型。getaddrinfo 函数(图 11.25)中的循环会创建两个 addrinfo 结构，即为由 gethostbyname 返回的每个 IP 地址创建一个。因为没有指定套接口类型，所以每个地址结构中也设置了 AI_CLONE 标志。在图 11.29 中展示了结果链表的结构。

在图 11.26 中调用了 ga_serv。因为 domain 服务对 TCP 和 UDP 都有效，所以调用了两次 getservbyname 以及两次 ga_port：前一次调用最后一个参数为 SOCK_STREAM，后一次调用为 SOCK_DGRAM。第一次调用 ga_port 时，开始时的链表如图 11.29。在图 11.28 中两个结构都设置了 AI_CLONE 选项，但套接口类型为 0。因此第一次调用 ga_port 后会将每个 addrinfo 结构的 ai_socktype 成员置为 SOCK_STREAM，套接口地址结构中的端口号则置为 53。AI_CLONE 标志保持原设置。这样就得到了如图 11.30 中的链表。

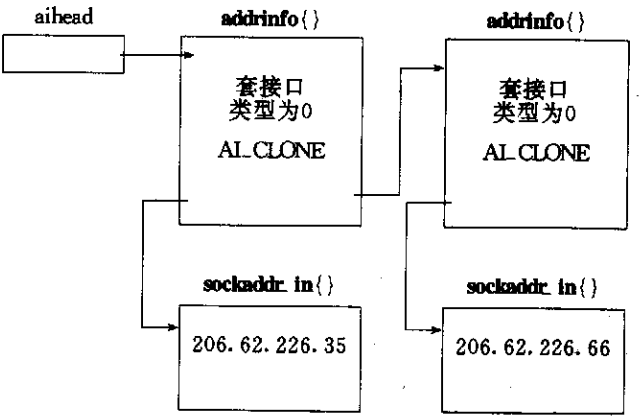


图 11.29 第一次调用 ga_port 时的 addrinfo 结构

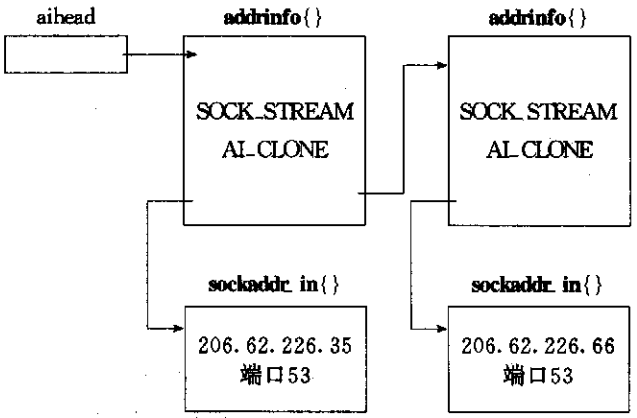


图 11.30　第一次调用 ga_port 后的 addrinfo 结构

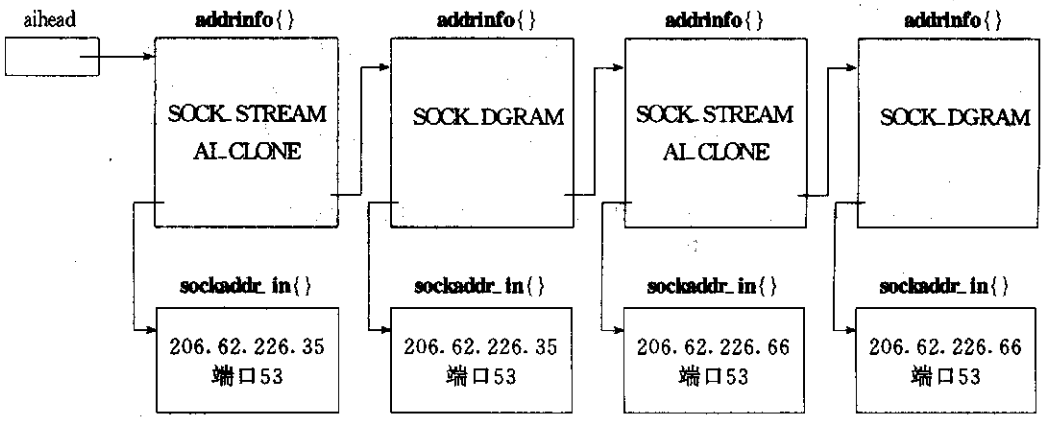


图 11.31　第二次调用 ga_port 后的 addrinfo 结构

但第二次调用 ga_port 时(最后一个参数为 SOCK_DGRAM),因为设置了 AI_CLONE 标志且套接口类型不为 0,所以对每个 addrinfo 结构调用了 ga_clone。在新复制的结构中的 ai_socktype 成员的值设为 SOCK_DGRAM,结果链表如图 11.31 所示。在该图中第二个 addrinfo 结构和它的套接口地址结构是从第一个结构复制而来的,第四个 addrinfo 结构和它的套接口地址结构是从第三个结构复制而来的。

图 11.32 是 ga_clone 函数的源代码,在图 11.28 中调用它从已有的结构集复制新的 addrinfo 结构及其套接口地址结构。

```
 5 struct addrinfo *
 6 ga_clone(struct addrinfo *ai)
 7 {
 8     struct addrinfo    *new;
 9     if ( (new = calloc(1, sizeof(struct addrinfo))) == NULL)
10                 return(NULL);
11     new->ai_next = ai->ai_next;
12     ai->ai_next = new;
13     new->ai_flags = 0;          /* make sure AI_CLONE is off */
14     new->ai_family = ai->ai_family;
```

```
15      new->ai_socktype = ai->ai_socktype;
16      new->ai_protocol = ai->ai_protocol;
17      new->ai_canonname = NULL;
18      new->ai_addrlen = ai->ai_addrlen;
19      if ( (new->ai_addr = malloc(ai->ai_addrlen)) == NULL)
20          return(NULL);
21      memcpy(new->ai_addr, ai->ai_addr, ai->ai_addrlen);
22      return(new);
23  }
```

图 11.32 ga_clone 函数[libgai/ga_clone.c]

分配 addrinfo 结构并将其插入到链表中

第 9～12 行 分配一个新的 addrinfo 结构，把它的 ai_next 指针的值设为被复制的项(也就是将排在链表中新结构前面的那个结构)的 ai_next 指针。被复制项的 ai_next 指针则变成指向新分配的结构。

初始化新复制的项

第 13～22 行 新 addrinfo 结构中的所有字段都是从被复制的项中拷贝而来，不过 ai_flags 被设成 0，ai_canonname 被设成空指针。函数的返回值为一个指向新创建结构的指针。

图 11.33 为 ga_unix 函数的源代码，在图 11.20 中处理 Unix 域路径名时调用过它。

```
3  int
4  ga_unix(const char *path, struct addrinfo *hintsp, struct addrinfo **result)
5  {
6      int         rc;
7      struct addrinfo    *aihead, **aipnext;
8      aihead = NULL;
9      aipnext = &aihead;
10     if (hintsp->ai_family != AF_UNSPEC && hintsp->ai_family != AF_LOCAL)
11         return(EAI_ADDRFAMILY);
12     if (hintsp->ai_socktype == 0) {
13             /* no socket type specified; return stream then dgram */
14         hintsp->ai_socktype = SOCK_STREAM;
15         if ( (rc = ga_aistruct(&aipnext, hintsp, path, AF_LOCAL)) != 0)
16             return(rc);
17         hintsp->ai_socktype = SOCK_DGRAM;
18     }
19     if ( (rc = ga_aistruct(&aipnext, hintsp, path, AF_LOCAL)) != 0)
20         return(rc);
21     if (hintsp->ai_flags & AI_CANONNAME) {
22         struct utsname      myname;
23         if (uname(&myname) < 0)
24             return(EAI_SYSTEM);
25         if ( (aihead->ai_canonname = strdup(myname.nodename)) == NULL)
26             return(EAI_MEMORY);
27     }
28     *result = aihead;       /* pointer to first structure in linked list */
29     return(0);
30 }
```

图 11.33 ga_unix 函数[libgai/ga_unix.c]

ga_aistruct 创建结构

第 10～20 行　如果没有指定套接口类型，则调用两次 ga_aistruct 函数创建两个 addrinfo 结构：一次用 SOCK_STREAM 套接口类型，另一次用 SOCK_DGRAM 套接口类型。但如果调用者指定了一个非零的套接口类型，就只调用一次 ga_aistruct 函数，创建一个所指定套接口类型的 addrinfo 结构。

返回规范主机名

第 21～27 行　如果调用者设置了 AI_CANONNAME 标志，则调用 uname 获取系统名，并将其中的 nodename 成员作为主机名字返回。

我们用下面介绍的 ga_aistruct 函数来解释 aihead 和 aipnext 指针。

ga_aistruct 函数在图 11.24 和 11.25 中被用来创建一个 IPv4 或 IPv6 的 addrinfo 结构，在图 11.33 中用来创建一个 Unix 域套接口的 addrinfo 结构。图 11.34 列出了这个函数源代码的第一部分。

```
5 int
6 ga_aistruct(struct addrinfo  * * * paipnext, const struct addrinfo  * hintsp,
7                 const void  * addr, int family)
8 {
9     struct addrinfo     * ai;
10    if ( (ai = calloc(1, sizeof(struct addrinfo))) == NULL)
11        return(EAI_MEMORY);
12    ai->ai_next = NULL;
13    ai->ai_canonname = NULL;
14     * * paipnext = ai;
15     * paipnext = &ai->ai_next;
16    if ( (ai->ai_socktype = hintsp->ai_socktype) == 0)
17        ai->ai_flags |= AI_CLONE;
18    ai->ai_protocol = hintsp->ai_protocol;
```

图 11.34　ga_aistruct 函数：第一部分[libgai/ga_aistruct.c]

分配 addrinfo 结构并将其加到链表中

第 10～15 行　给一个 addrinfo 结构分配内存并将其加到已有的链表中。有两个指针用来建立这个链表：aihead 和 aipnext。它们都是在图 11.20 中(IPv4 或 IPv6 套接口)或图 11.33中(Unix 域套接口)进行空间分配和初始化的。aihead 被初始化成一个空指针，aipnext 则指向 aihead。如图 11.35 所示。

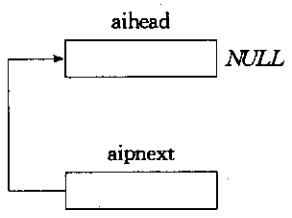


图 11.35　链表指针的初始化

aihead 总是指向链表中的第一个 addrinfo 结构(因此它的数据类型是 struct addrinfo *)。aipnext 通常指向链表中最后一个结构的 ai_next 成员(因此它的数据类型是 struct addrinfo * *)。这里我们对 aipnext 使用“通常”这个定语是因为初始化时它实际指向 aihead，

但在分配了第一个结构并加到链表中后，它就总是指向 ai_next 成员了。

回到 ga_aistruct 函数。在分配了一个新的结构后执行了下面两条语句：

```
* *paipnext = ai ;
*paipnext = &ai->ai_next;
```

第一条语句设置链表中最后一个结构的 ai_next 指针(如果这是链表中的第一个结构则设置 aihead 指针)，使其指向新分配的结构，第二条语句将 aipnext 设成指向新分配的结构的 ai_next 成员。因为它们是作为函数的参数，所以又加上了一层间接引用(见习题 11.4)。在链表中加入第一个结构后，数据结构如图 11.36 所示。

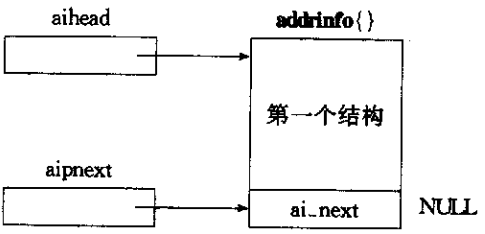


图 11.36 加入第一个结构后的链表

在下一次调用 ga_aistruct 函数分配第二个结构并加入链表后，数据结构如图 11.37 所示。

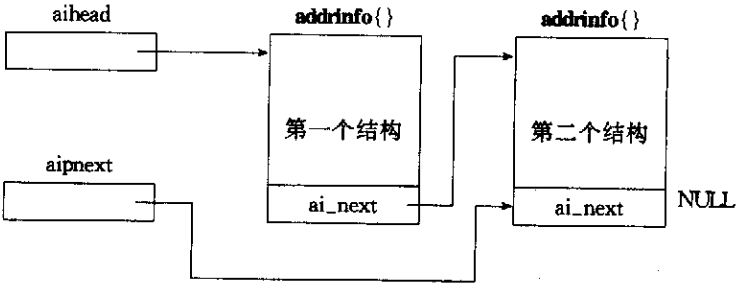


图 11.37 加入第二个结构后的链表

设置套接口类型

第 16～17 行 ai_socktype 成员设置成由调用者提供的套接口类型，如果为 0，则设置 AI_CLONE 标志。

图 11.38 是这个函数的第二部分：使用 switch 语句，每个 case 对应一种地址族，分别分配套接口地址结构并进行初始化。

```
19      switch ((ai->ai_family = family)) {
20      case AF_INET: {
21              struct sockaddr_in      *sinptr;
22                  /* allocate sockaddr_in{} and fill in all but port */
23              if ( (sinptr = calloc(1, sizeof(struct sockaddr_in))) == NULL)
24                      return(EAI_MEMORY);
25 #ifdef     HAVE_SOCKADDR_SA_LEN
26              sinptr->sin_len = sizeof(struct sockaddr_in);
27 #endif
28              sinptr->sin_family = AF_INET;
```

```
29                  memcpy(&sinptr->sin_addr, addr, sizeof(struct in_addr));
30                  ai->ai_addr = (struct sockaddr *) sinptr;
31                  ai->ai_addrlen = sizeof(struct sockaddr_in);
32                  break;
33              }
34      case AF_INET6: {
35                  struct sockaddr_in6     *sin6ptr;
36                      /* allocate sockaddr_in6{} and fill in all but port */
37                  if ( (sin6ptr = calloc(1, sizeof(struct sockaddr_in6))) == NULL)
38                          return(EAI_MEMORY);
39 #ifdef      HAVE_SOCKADDR_SA_LEN
40                  sin6ptr->sin6_len = sizeof(struct sockaddr_in6);
41 #endif
42                  sin6ptr->sin6_family = AF_INET6;
43                  memcpy(&sin6ptr->sin6_addr, addr, sizeof(struct in6_addr));
44                  ai->ai_addr = (struct sockaddr *) sin6ptr;
45                  ai->ai_addrlen = sizeof(struct sockaddr_in6);
46                  break;
47              }
48      case AF_LOCAL: {
49                  struct sockaddr_un      *unp;
50                      /* allocate sockaddr_un{} and fill in */
51                  if (strlen(addr) >= sizeof(unp->sun_path))
52                      return(EAI_SERVICE);
53                  if ( (unp = calloc(1, sizeof(struct sockaddr_un))) == NULL)
54                      return(EAI_MEMORY);
55                  unp->sun_family = AF_LOCAL;
56                  strcpy(unp->sun_path, addr);
57 #ifdef      HAVE_SOCKADDR_SA_LEN
58                  unp->sun_len = SUN_LEN(unp);
59 #endif
60                  ai->ai_addr = (struct sockaddr *) unp;
61                  ai->ai_addrlen = sizeof(struct sockaddr_un);
62                  if (hintsp->ai_flags & AI_PASSIVE)
63                      unlink(unp->sun_path);  /* OK if this fails */
64                  break;
65              }
66      }
67      return(0);
68 }
```

图 11.38　ga_aistruct 函数：第二部分[libgai/ga_aistruct.c]

分配并初始化 IPv4 套接口地址结构

第 20～33 行　分配一个 sockaddr_in 结构，将 addrinfo 结构中的 ai_addr 指针指向它。初始化套接口地址结构中的 IP 地址、地址族和长度等成员。在调用 ga_serv 前不进行端口号的初始化，因为 ga_serv 会调用 ga_port。

分配并初始化 IPv6 套接口地址结构

第 34～47 行　与 IPv4 类似，分配并初始化一个 sockaddr_in6 结构。

分配并初始化 Unix 域套接口地址结构

第 48～65 行　分配并初始化一个 sockaddr_un 结构。该地址是一个路径名，如果调用

者设置了 AI_PASSIVE 标志，程序中会试图 unlink 这个路径名以防调用 bind 时出错。即使 unlink 失败也没有什么问题。

图 11.39 里的 ga_echeck 函数曾在图 11.20 中被调用，以对调用者的参数进行一些初步的错误检查。

```
 5 int
 6 ga_echeck(const char *hostname, const char *servname,
 7            int flags, int family, int socktype, int protocol)
 8 {
 9     if (flags & ~(AI_PASSIVE | AI_CANONNAME))
10         return(EAI_BADFLAGS);   /* unknown flag bits */
11     if (hostname == NULL || hostname[0] == '\0') {
12         if (servname == NULL || servname[0] == '\0')
13             return(EAI_NONAME);     /* host or service must be specified */
14     }
15     switch(family) {
16         case AF_UNSPEC:
17             break;
18         case AF_INET:
19             if (socktype != 0 &&
20                 (socktype != SOCK_STREAM &&
21                  socktype != SOCK_DGRAM &&
22                  socktype != SOCK_RAW))
23                 return(EAI_SOCKTYPE);     /* invalid socket type */
24             break;
25     case AF_INET6:
26         if (socktype != 0 &&
27             (socktype != SOCK_STREAM &&
28              socktype != SOCK_DGRAM &&
29              socktype != SOCK_RAW))
30             return(EAI_SOCKTYPE);         /* invalid socket type */
31         break;
32     case AF_LOCAL:
33         if (socktype != 0 &&
34             (socktype != SOCK_STREAM &&
35              socktype != SOCK_DGRAM))
36             return(EAI_SOCKTYPE);         /* invalid socket type */
37         break;
38         default:
39             return(EAI_FAMILY);           /* unknown protocol family */
40     }
41     return(0);
42 }
```

图 11.39 ga_echeck 函数[libgai/ga_echeck.c]

第 9～14 行 检查各标志以及是否指定了主机名和服务名。

第 15～41 行 各个地址族只支持某些特定的套接口类型，这也是对套接口类型的检查。

这里没有检查调用者的 ai_protocol 线索，因为很少有程序指定它的值(作为函数 socket 的第三个参数)。要是指定了无效的组合，譬如套接口类型为 SOCK_DGRAM，而协议为 IPPROTO_TCP，则在图 11.34 中将协议线索返回给调用者，因而如果用这个值调用 socket，

将返回一个 EPROTONOSUPPORT 错误。

到这里已经完成了 getaddrinfo 和它内部调用的所有函数。图 11.40 是 freeaddrinfo 函数的源代码，它释放链表使用的所有内存空间。在图 11.26 中，一旦出错就调用该函数，用户也可以调用它来释放一个结构链表。

```
1 #include    "gai_hdr.h"
2 void
3 freeaddrinfo(struct addrinfo *aihead)
4 {
5     struct addrinfo    *ai, *ainext;
6     for (ai = aihead; ai != NULL; ai = ainext) {
7         if (ai->ai_addr != NULL)
8             free(ai->ai_addr);          /* socket address structure */
9         if (ai->ai_canonname != NULL)
10            free(ai->ai_canonname);
11        ainext = ai->ai_next;           /* can't fetch ai_next after free() */
12        free(ai);                       /* the addrinfo{} itself */
13    }
14 }
```

图 11.40　freeaddrinfo 函数：第一部分[libgai/freeaddrinfo.c]

第 6～13 行　遍历 addrinfo 结构链表，释放每个已分配的套接口地址结构。释放已分配的规范主机名字符串。最后释放 addrinfo 结构本身。必须在释放结构前保存其中 ai_next 指针的内容，因为在 free 返回后就不能再引用这个结构了。

图 11.41 是 getnameinfo 函数的具体实现。它由一个 switch 语句组成，每种地址族对应一个 case。

```
2 int
3 getnameinfo(const struct sockaddr *sa, socklen_t salen,
4             char *host, size_t hostlen,
5             char *serv, size_t servlen, int flags)
6 {
7     switch (sa->sa_family) {
8     case AF_INET: {
9             struct sockaddr_in *sain = (struct sockaddr_in *) sa;
10            return(gn_ipv46(host, hostlen, serv, servlen,
11                            &sain->sin_addr, sizeof(struct in_addr),
12                            AF_INET, sain->sin_port, flags));
13    }
14    case AF_INET6: {
15            struct sockaddr_in6 *sain = (struct sockaddr_in6 *) sa;
16            return(gn_ipv46(host, hostlen, serv, servlen,
17                            &sain->sin6_addr, sizeof(struct in6_addr),
18                            AF_INET6, sain->sin6_port, flags));
19    }
20    case AF_LOCAL: {
21            struct sockaddr_un *un = (struct sockaddr_un *) sa;
22            if (hostlen > 0)
23                snprintf(host, hostlen, "%s", "/local");
24            if (servlen > 0)
```

```
25                  snprintf(serv, servlen, "%s", un->sun_path);
26              return(0);
27          }
28      default:
29          return(1);
30      }
31  }
```

图 11.41 getnameinfo 函数[libgai/getnameinfo.c]

处理 IPv4 和 IPv6 套接口地址结构

第 8～19 行 调用 gn_ipv46 函数(将在下面介绍)处理 IPv4 和 IPv6 套接口地址结构。

处理 Unix 域套接口地址结构

第 20～27 行 对于 Unix 域套接口地址结构,返回/local 作为主机名,返回绑定在套接口上的路径名作为服务名。如果套接口没有绑定路径名,返回的服务名将是个空串。

> 我们用 snprintf 代替 strncpy 返回主机名和服务名。如果用后者的话可以这样写
>
> ```
> strncpy(host, "/local", hostlen);
> ```
>
> 虽然这样可以保证不使调用者的缓冲区溢出,但如果 hostlen 小于或等于 6,调用者的缓冲区将不是以空字符结尾。因为 getnameinfo 是一个库函数,所以应该按调用者的要求返回一个以空字符结尾的字符串,否则会导致调用者的程序以后出问题。因此要写成:
>
> ```
> strncpy(host, "/local", hostlen-1);
> host[hostlen-1] = '\0';
> ```
>
> 这既保证不让缓冲区溢出,又使结果串以空字符结尾。我们用 snprintf 代替这两条语句以达到同样的目的。另一种办法是定义一个调用 strncpy 并以空字符终止结果串的库函数,但调用现成的 snprintf 似乎更简单些。

图 11.42 是 gn_ipv46 函数的源代码,它可以为 getnameinfo 处理 IPv4 和 IPv6 套接口地址结构。

返回主机名

第 12～23 行 如果设置了 NI_NUMERICHOST 标志,就调用 inet_ntop 返回表达格式的 IP 地址;否则 gethostbyaddr 查找与该 IP 地址对应的主机名。如果 gethostbyaddr 成功,而且设置了 NI_NOFQDN(无需全限定域名)标志的话,主机名在第一个点号前被截断。

gethostbyname 失败处理

第 24～29 行 如果 gethostbyname 失败(在因特网上配置不当的 DNS 服务器不少;见 TCPv3 的 14.8 节),而且设置了 NI_NAMEREQD 标志,则返回错误。否则返回对该 IP 地址用 inet_ntop 处理所形成的地址串。

返回服务字符串

第 32～42 行 如果设置了 NI_NUMERICSERV 标志,就只返回十进制数的端口号;否则调用 getservbyport。除非设置了 NI_DGRAM 标志,最后一个参数将是空指针。如 get-

servbyport 失败，则返回十进制数形式的端口号。

```
 5 int
 6 gn_ipv46(char *host, size_t hostlen, char *serv, size_t servlen,
 7           void *aptr, size_t alen, int family, int port, int flags)
 8 {
 9     char       *ptr;
10     struct hostent    *hptr;
11     struct servent    *sptr;
12     if (hostlen > 0) {
13         if (flags & NI_NUMERICHOST) {
14             if (inet_ntop(family, aptr, host, hostlen) == NULL)
15                 return(1);
16         } else {
17             hptr = gethostbyaddr(aptr, alen, family);
18             if (hptr != NULL && hptr->h_name != NULL) {
19                 if (flags & NI_NOFQDN) {
20                     if ( (ptr = strchr(hptr->h_name, '.')) != NULL)
21                         *ptr = 0;    /* overwrite first dot */
22                 }
23                 snprintf(host, hostlen, "%s", hptr->h_name);
24             } else {
25                 if (flags & NI_NAMEREQD)
26                     return(1);
27                 if (inet_ntop(family, aptr, host, hostlen) == NULL)
28                     return(1);
29             }
30         }
31     }
32     if (servlen > 0) {
33         if (flags & NI_NUMERICSERV) {
34             snprintf(serv, servlen, "%d", ntohs(port));
35         } else {
36             sptr = getservbyport(port, (flags & NI_DGRAM) ? "udp" : NULL);
37             if (sptr != NULL && sptr->s_name != NULL)
38                 snprintf(serv, servlen, "%s", sptr->s_name);
39             else
40                 snprintf(serv, servlen, "%d", ntohs(port));
41         }
42     }
43     return(0);
44 }
```

图 11.42　gn_ipv46 函数：处理 IPv4 和 IPv6 套接口地址结构[libgai/gn_ipv46.c]

11.17　小　结

在编写独立于协议的代码时 getaddrinfo 是个很有用的函数。但直接调用它要花好几步，而且对于不同的情况存在一些都要处理的细节：扫描所有返回的结构，忽略 socket 返回的错误，为 TCP 服务器设置 SO_REUSEADDR 套接口选项，诸如此类。为简化这些细节，我们编写了五个函数：tcp_connect、tcp_listen、udp_client、udp_connect、udp_server。并在独立于协议的时间/日期服务器程序和客户程序的 TCP 和 UDP 版本中展示了这些函数的用法。

gethostbyname 和 gethostbyaddr 也是不可重入函数的例子。这两个函数共享一个静态的结构，返回一个指向该结构的指针。在第 23 章中介绍线程时还会遇到并讨论这个重入问题。我们介绍了一些厂商提供的这两个函数的_r 版本，这是一种解决方法，但是需要对调用这些函数的所有应用程序进行修改。

11.18 习 题

11.1 在图 11.8 中调用者必须传递一个整数指针以得到协议地址的大小。如果调用者没有这么做的话(譬如，传递的最后一个参数为空指针)，调用者还能怎样得到协议地址的大小呢？

11.2 修改图 11.10 中的程序，用 getnameinfo 代替 sock_ntop。应给 getnameinfo 传递那些标志？

11.3 在 7.5 节中我们讨论了使用 SO_REUSEADDR 盗用端口。为了弄清它是怎样工作的，构造一个图 11.15 所示的独立于协议的 UDP 时间/日期服务器程序。在一个窗口中启动该服务器的一个实例，给它捆绑通配地址和某个你选择的端口。在另一个窗口中启动一个客户并验证服务器正在处理客户请求(注意服务器的 printf)。再在第三个窗口中启动服务器的另一个实例，这次给它捆绑该主机的一个单播地址和与第一个服务器相同的端口。马上会遇到什么问题？修复这个问题并重新启动第二个服务器。启动一个客户，发送一个数据报，验证第二个服务器已盗用了第一个服务器的端口。如果可能，用与启动第一个服务器使用的登录账号不同的另一个账号再启动第二个服务器，看能否成功盗用端口，因为一些厂商只允许用户 ID 相同的进程第二次捆绑原来进程已绑定的端口。

11.4 讨论图 11.34 时我们说过 aipnext 的地址是 ga_aistruct 函数的一个参数，从而迫使对其又加上了一层间接的引用。为什么不把 aipnext 变成一个全局变量，以取代作为参数传递它的地址？

11.5 在 11.5 节对 Unix 域的讨论中我们提到没有一个 IANA 的服务名以斜杠开头。这些服务名中有包含斜杠的吗？

11.6 在 2.10 节的末尾举了两个 telnet 的例子：分别连往 daytime 服务器和 echo 服务器。了解客户要经过 gethostbyname 和 connect 这两步后，你能判定它的哪些输出行对应哪个步骤吗？

11.7 当不能给一个 IP 地址找出对应的主机名时，gethostbyaddr 可能要花很长时间(可长达 80 秒)才能返回错误。编写一个新的名为 getnameinfo_timeo 的函数，它有一个附加的整数参数指定等待应答的最大秒数。如果超时，而且没有设置 NI_NAMEREQD 标志，就调用 inet_ntop 返回一个地址串。

第 12 章　守护进程和 inetd 超级服务器

12.1　概　述

守护进程(daemon)是在后台运行不受终端控制的进程。Unix 系统中一般有很多守护进程在后台运行(20 到 50 个),执行不同的管理任务。

想要脱离所有的终端的原因是守护进程可能是从终端上启动(与从初始化脚本中启动相反),在这之后这个终端要能用来执行其他任务。举例来说,如果在某终端上启动了一个守护进程后从终端上注销,其他人又从该终端登录,那么任何守护进程的错误信息不应在后面用户的终端会话过程中出现。同样,由终端上的一些键所产生的信号(如中断信号),不应对以前从该终端上启动的任何守护进程造成影响。虽然使服务器程序在后台运行很容易(只要在 shell 命令行的结尾加一个 & 符号),但是我们还是应该使程序能自动转到后台并且脱离与终端的联系。

有好几种方法启动守护进程:

1. 在系统启动时很多守护程序都是由系统初始化脚本启动。这些脚本一般在/etc 目录或以/etc/rc 开头的目录下,它们的位置和内容依赖于具体的实现。由这些脚本启动的守护进程在开始时都拥有超级用户权限。
   有几个网络服务器一般从这些脚本启动:inetd 超级服务器(见下一步)、Web 服务器或邮件服务器(一般为 sendmail)。12.2 节中介绍的 syslogd 守护进程通常也是由某个这样的脚本启动的。
2. 许多网络服务器是由 inetd 超级服务器启动的,在这一节的后面会对此进行介绍。inetd 自己是由上一步中的某个脚本启动的。inetd 监听网络请求(Telnet、FTP 等),当请求到来时启动实际的服务器(Telnet 服务器、FTP 服务器等)。
3. cron 守护进程按规则定期执行一些程序,由它启动的程序也以守护进程的方式运行。cron 自己是在系统启动过程中由第 1 步启动的。
4. 可用 at 命令指定在将来的某一时刻执行程序。cron 守护进程在到达相应的时间时会启动这些程序,所以它们是以守护程序的方式运行的。
5. 不管是在前台还是在后台,守护进程也可以在用户终端上启动,这在测试守护进程或守护进程因某些原因终止而要重启时经常使用。

由于守护进程没有控制终端,在发生问题时它要用一些其他方式以输出消息。这些消息既有一般的通告消息,也有需管理员处理的紧急事件消息。syslog 函数是输出这些消息的标准方式,它将消息发往 syslogd 守护进程。

12.2 syslogd 守护进程

Unix 系统通常会从一个初始化脚本中启动名为 syslogd 的守护进程，只要系统不停止它就一直运行。源自 Berkeley 的 syslogd 的实现在启动时执行以下操作：

1. 读入配置文件，通常是/etc/syslog.conf，它设定守护进程对接收每次键入的各种登记消息(log message)怎样处理。这些消息可能被写入一个文件(一种特殊情况是文件为/dev/console，这将把消息写到控制台上)，或发给指定的用户(如果该用户已登录到系统)，或转发给另一台主机上的 syslogd 进程。
2. 创建一个 Unix 域套接口，给它捆绑路径名/var/run/log(在某些系统上是/dev/log)。
3. 创建一个 UDP 套接口，给它捆绑端口 514(syslog 服务使用的端口号)。
4. 打开路径名/dev/klog，内核中的所有出错消息作为这个设备的输入出现。

在这之后 syslogd 进程运行一个无限循环，循环中调用 select，等待三个描述字(以上第2、3、4步生成的描述字)之一变为可读，读入登记消息，并按配置文件对消息进行处理。如果该守护进程收到 SIGHUP 信号，它会重新读入配置文件。

通过建立一个 Unix 域数据报套接口，并向 syslogd 守护进程绑定的路径名发送我们的消息，我们就能从自己的守护进程向 syslogd 发送登记消息，但更简单的接口是在下一节中介绍的 syslog 函数。另外也可以创建一个 UDP 套接口，将日志消息发到回馈地址及端口 514。

> 在新的实现中除非管理员指定，将不创建 UDP 套接口，这是因为允许任何人向这个端口发送 UDP 数据报(有可能填满它的套接口接收缓冲区)可能导致收不到真正需要处理的登记消息。
>
> 在 syslogd 的各种实现中存在差异。举例来说，源自 Berkeley 的实现使用 Unix 域套接口，而系统 V 使用一个基于流的登记驱动程序[①]。各种源自 Berkeley 的实现中使用的 Unix 域套接口的路径名也各不相同。但是如使用 syslog 函数就可以忽略这些细节问题。

12.3 syslog 函数

因为守护进程没有控制终端，所以它不能 fprintf 到 stderr 上。守护进程为登记消息通常调用 syslog 函数。

---

① 译者注：注意区别这里的流(streams)和我们一直在使用的字节流(stream)。前者是一种访问驱动程序(driver)的方法，在本书第33章介绍，它也称为 STREAMS。后者是与数据报(datagram)相对的数据传送方式，我们把它译成字节流一方面避免了与流相混淆，另一方面是强调它是无记录边界的数据流(不同于数据报方式的数据流)，或者说它的记录单元是最小的字节(而数据报方式的记录单元就是每个数据报本身)。另外一个应避免混淆的概念是标准 I/O 函数库中的标准 I/O 流(standard I/O streams)，它在本书中出现得极少。

```
#include <syslog.h>
void syslog(int priority,const char *message, ...);
```

尽管这个函数最初是为 BSD 系统开发的，但现在大多数 Unix 厂商都有提供。Posix 中没有提到 syslog，但在 Unix 98 中它是必需的。

在图 12.1 和 12.2 中出现的 priority 参数是级别(level)和设施(facility)的组合。message 与 printf 所用的格式化字符串类似，还增加了%m，它将由对应当前 errno 值的出错消息所取代。在消息的结尾可以加换行符，但这不是必需的。

如图 12.1 所示，登记消息的级别可从 0 到 7，它们是按从高到低的顺序排列的。如果发送者没有指定级别值，就缺省为 LOG_NOTICE。

级别(level)	值	描述
LOG_EMERG	0	系统不可用(优先级最高)
LOG_ALERT	1	必须立即进行处理
LOG_CRIT	2	危险情况
LOG_ERR	3	出错情况
LOG_WARNING	4	警告性情况
LOG_NOTICE	5	常见但值得注意的情况(缺省)
LOG_INFO	6	通告消息
LOG_DEBUG	7	调试消息(优先级最低)

图 12.1　登记消息的级别

登记消息还包含一个标识发送消息进程的类型的设施。我们在图 12.2 中列出了它的各种值。如果没有指定设施，就缺省为 LOG_USER。

设施(facility)	描述
LOG_AUTH	安全/授权消息
LOG_AUTHPRIV	安全/授权消息(私有的)
LOG_CRON	cron 守护进程
LOG_DAEMON	系统守护进程
LOG_FTP	FTP守护进程
LOG_KERN	内核消息
LOG_LOCAL0	本地使用
LOG_LOCAL1	本地使用
LOG_LOCAL2	本地使用
LOG_LOCAL3	本地使用
LOG_LOCAL4	本地使用

（续）

设施(facility)	描述
LOG_LOCAL5	本地使用
LOG_LOCAL6	本地使用
LOG_LOCAL7	本地使用
LOG_LPR	行式打印机系统
LOG_MAIL	邮件系统
LOG_NEWS	网络新闻系统
LOG_SYSLOG	由 syslogd 内部产生的消息
LOG_USER	任意的用户级消息(缺省)
LOG_UUCP	UUCP 系统

图 12.2 登记消息的设施

举例来说，当调用 rename 函数失败时，守护进程可能会做以下调用：

```
syslog(LOG_INFO|LOG_LOCAL2, "rename(%s, %s): %m", file1, file2);
```

设施和级别的目的是，允许在/etc/syslog.conf 文件中进行配置，使得对相同设施的消息得到同样的处理，或使相同级别的消息得到同样的处理。举例来说，配置文件中可能有以下两行：

```
kern.*              /dev/console
local7.debug        /var/log/cisco.log
```

指定内核的所有消息登记到控制台上，所有设施为 local 7 的调试消息添加到/var/log/cisco.log 文件的末尾。

当应用程序第一次调用 syslog 时，它创建一个 Unix 域数据报套接口，然后调用 connect 连往 syslogd 守护进程建立的套接口的众所周知路径名(譬如/var/run/log)。这个套接口在进程终止前一直打开。另外，进程也可以调用 openlog 和 closelog。

```
#include <syslog.h>

void openlog(const char *ident, int options, int facility);

void closelog(void);
```

openlog 可在第一次调用 syslog 前调用，当应用程序不再需要发送登记消息时可调用 closelog。

ident 是一个字符串，它将被 syslog 加到每条登记消息的前面。一般情况下它的值为程序名。[②]。

options 参数由图 12.3 中的一个或多个常值的“逻辑或”组成。

② 作者注： 请留意 openlog 的大多数实现仅仅保存一个指向 ident 字符串的指针；它们不拷贝这个字符串。这就是说这个字符串不应该是在栈上分配的(自动变量就是这样)，因为以后调用 syslog 时如果相应的栈帧已弹走，所保存的指针就不再指向 ident 字符串。

选项(options)	描　述
LOG_CONS	如果不能发往 syslogd 守护进程，则登记到控制台上
LOG_NDELAY	不延迟打开，立即创建套接口
LOG_PERROR	既发往 syslogd 守护进程，又登记到标准错误输出
LOG_PID	登记每条消息的进程 ID

图 12.3　openlog 的选项

通常调用 openlog 时并不创建 Unix 域套接口。它是在第一次调用 syslog 时打开的。LOG_NDELAY 选项能使套接口在调用 openlog 时创建。

openlog 的 facility 参数为后面没有设置设施的 syslog 调用设置一个缺省值。一些守护进程调用 openlog 设置设施(它在一个进程中通常不变)，然后在调用 syslog 时只设置级别(因为级别会随错误而改变)。

logger 命令也能产生日志消息。例如，在 shell 脚本中可以用它向 syslogd 发送消息。

12.4　daemon_init 函数

图 12.4 给出了名为 daemon_init 函数，调用它(通常从服务器程序)可使一个进程变成守护进程。

fork

第 10～11 行　首先调用 fork，然后终止父进程，留下子进程继续运行。如果进程是以 shell 命令方式从前台启动，当父进程终止时，shell 就认为命令已完成。这可以自动使子进程在后台运行。子进程继承了父进程的进程组号，但它拥有自己的进程号。这就保证了这个子进程不是进程组头，这是下一步调用 setsid 所必需的。

setsid

第 12～13 行　setsid 是一个 Posix.1 函数，它创建一个新的登录会话(session)。(APUE 第 9 章讨论了关于进程之间的关系和会话的细节。)这个进程变成新会话的会话头和新进程组的组长，不再有控制终端。

忽略 SIGHUP 信号并再次 fork

第 14～16 行　忽略 SIGHUP 信号 并再次调用 fork。当这个函数返回时，正在运行的是第二次生成的子进程，第一次生成的子进程是它的父进程，而且已经终止。

第二次 fork 的目的是确保守护进程将来即使打开一个终端设备，也不会自动获得控制终端。在 SVR4 中，当没有控制终端的会话头进程打开终端设备时(这个终端现在不是其他会话的控制终端)，该终端自动成为这个会话头的控制终端。但通过第二次调用 fork，可以确保这次生成的子进程不再是一个会话的头，因此它不会获得控制终端。这里必须忽略 SIGHUP 信号是因为当会话头(第一次生成的子进程)终止时，该会话中的所有进程(第二次生成的子进程)都会收到 SIGHUP 信号。

```
1 #include      "unp.h"
2 #include      <syslog.h>
3 #define MAXFD     64
4 extern int daemon_proc;          /* defined in error.c */
5 void
6 daemon_init(const char *pname,int facility)
7 {
8     int         i;
9     pid_t       pid;
10    if( (pid = Fork()) != 0)
11        exit(0);                /* parent terminates */
12        /* 1st child continues */
13    setsid();
14    Signal(SIGHUP,SIG_IGN);
15    if ( (pid = Fork()) != 0)
16        exit(0);                /* 1st child terminates */
17        /* 2nd child continues */
18    daemon_proc = 1;            /* for our err_XXX() functions */
19    chdir("/");                 /* change working directory */
20    umask(0);                   /* clear our file mode creation mask */
21    for (i = 0; i < MAXFD; i++)
22        close(i);
23    openlog(pname,LOG_PID,facility);
24 }
```

图12.4 daemon_init函数:使进程变成一个守护进程[daemon_init.c]

为出错处理函数设置标识

第17～18行 将全局变量daemon_proc设成非零。这个外部变量是由我们的err_XXX函数(D.4节)定义的,当它的值为非零时,出错处理函数将调用syslog函数取代用fprintf输出到标准错误输出。这样我们不用从头到尾改程序,就可以让服务器在以非守护进程方式运行时(即测试服务器时)调用一个自己的出错处理函数,而在以守护进程方式运行时调用syslog。

改变工作目录并清除文件模式创建掩码

第19～20行 将工作目录改到根目录,不过确实有些守护进程有理由将它改到其他的目录。举例来说,打印机守护进程可能会把工作目录改为打印机的假脱机处理(spool)目录,即它工作时要使用的目录。守护进程如果产生core文件,这个core文件将在工作目录下。改变工作目录的另一个原因是守护进程可能在任意文件系统中启动,如果保持工作目录不变的话,该文件系统就不能拆卸(unmount)。文件模式创建掩码被重置成0,这样守护进程创建自己的文件时,新文件的权限位不受原先的文件模式创建掩码的权限位的影响。

关闭所有打开的文件描述字

第21～22行 关闭本守护进程从运行它的进程(通常为shell)继承而来的所有打开的文件描述字。问题是怎样得到正在使用的最大的文件描述字;没有一个Unix函数提供这个值。有方法能得到一个进程能打开的最大的描述字数目,但由于这个限制可能是无限的,所

以确定最大描述字本身也变得复杂起来(参见 APUE 第 43 页)。我们的解决办法是关闭前 64 个描述字,即使它们之中可能有很多并没有打开。

有些守护进程以读写方式打开/dev/null,并将它复制到标准输入、标准输出和标准错误输出。这样可以保证这些常用的描述字是打开的,从这些描述字读时会返回 0(文件结束符),向它们写时内核会丢弃所有写入的信息。之所以要打开这些描述字是因为,这样的话守护进程调用的那些假定能使用标准输入、标准输出和标准错误输出的库函数就不会发生问题。另外,有些守护进程打开一个登记文件,并将其描述字复制到标准输出和标准错误输出。

用 syslogd 处理错误

第 23 行　调用 openlog。调用者一般将第一个参数设成程序名(譬如 argv[0])。这里的设置将进程号加到每条登记消息中。这里还设定了 facility 参数,其值为图 12.2 中出现的常值,如果缺省值 LOG_USER 能满足要求的话可设为 0。

守护进程运行时没有控制终端,所以它不会收到来自内核的 SIGHUP 信号。因此很多守护进程将该信号作为管理员通知其配置文件已修改之用,守护进程收到 SIGHUP 信号后应重新读入配置文件。另外两个守护进程不应收到的信号是 SIGINT 和 SIGWINCH,这些信号也可用作通知守护进程一些其他事项。

例子:守护进程方式的时间/日期服务器程序

图 12.5 是从图 11.10 进行修改后的不依赖于协议的时间/日期服务器程序。它调用 daemon_init 函数变成一个守护进程。

```
 1 #include   "unp.h"
 2 #include   <time.h>
 3 int
 4 main(int argc, char * * argv)
 5 {
 6     int    listenfd, connfd;
 7     socklen_t    addrlen, len;
 8     struct sockaddr    * cliaddr;
 9     char        buff[MAXLINE];
10     time_t      ticks;
11     daemon_init(argv[0], 0);
12     if (argc == 2)
13         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
14     else if (argc == 3)
15         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
16     else
17         err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");
18     cliaddr = Malloc(addrlen);
19     for ( ; ; ) {
20         len = addrlen;
21         connfd = Accept(listenfd, cliaddr, &len);
22         err_msg("connection from %s", Sock_ntop(cliaddr, len));
23       ticks = time(NULL);
24       snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
```

```
25        Write(connfd, buff, strlen(buff));
26        Close(connfd);
27    }
28 }
```

图 12.5 以守护进程方式运行的不依赖于协议的时间/日期服务器程序[inetd/daytimetcpsrv2.c]

只改了两个地方：当程序开始时尽快调用 daemon_init，并用自己的 err_msg 代替 printf 输出 IP 地址和端口号。实际上，如果想要程序以守护进程方式运行，就必须避免调用 printf 和 fprintf 函数，而以自己的 err_msg 函数代替。

如果在主机 solaris 上运行该程序，再从主机 bdsi 进行连接，然后检查/var/adm/messages 文件(所有 LOG_USER 消息发往的地方)，就可以在其中找到：

```
Jun  4 15:15:33 solaris.kohala.com daytimetcpsrv2[14882]
connection from ::ffff:206.62.226.35.3356
```

(这里因为行太长而折行了)。syslogd 守护进程在前面自动加上了日期、时间和 FQDN。

12.5 inetd 守护进程

在典型的 Unix 系统中都有很多服务器在运行，等待客户的请求。例如 FTP、Telnet、TFTP 等等。在 4.3BSD 版本前的 Unix 系统中，这些服务都有一个与之对应的进程。这些进程都是在系统启动时从/etc/rc 文件里启动，它们启动时所做的工作差不多一样：创建套接口，给它捆绑服务器的众所周知端口，等待连接(如果是 TCP)或数据报(如果是 UDP)，然后 fork。子进程为客户服务，父进程继续等待下一个客户请求。该模型有两个问题。

1. 这些守护进程都有几乎相同的启动代码，首先是创建套接口，还要考虑变成守护进程(与 daemon_init 函数类似)。
2. 每个守护进程在进程表中要占用一项，但它们在大部分时间里处于睡眠状态。

4.3BSD 版本通过提供一个因特网超级服务器：inetd 守护进程使这些问题得到简化。基于 TCP 或 UDP 的服务器都可以使用这个守护进程。它不处理其他的协议，如 Unix 域套接口。该守护进程解决了刚才提到的两个问题。

1. 因为大部分启动时要做的工作由 inetd 处理了，所以守护进程的编写得到简化。这避免了每个服务器程序都要调用 daemon_init 函数。
2. 单个进程(inetd)能为多个服务等待客户的请求，取代了每个服务一个进程的方式，这样减少了系统中的进程总数。

inetd 进程使用前面介绍的 daemon_init 函数中的技术将自己变成一个守护进程，然后读入并处理它的配置文件，通常为/etc/inetd.conf。这个文件配置超级服务器处理的服务，以及当一个服务请求到来时怎么做。文件中每行所包含的栏目如图 12.6 所示。下面是一些实例：

```
ftp     stream    tcp   nowait    root        /usr/bin/ftpd        ftpd -l
telnet  stream    tcp   nowait    root        /usr/bin/telnetd     telnetd
```

```
login   stream   tcp   nowait   root      /usr/bin/rlogind   rlogind -s
tftp    dgram    udp   wait     nobody    /usr/bin/tftpd     tftpd -s /tftpboot
```

服务器程序的实际名字总是在由 inetd 调用 exec 执行时作为第一个参数传递给它。

栏　目	说　明
service-name	必须是在/etc/services 文件中已定义的服务名
socket-type	stream(TCP)或 dgram(UDP)
protocol	必须在/etc/protocols 文件中已有定义:tcp 或 udp
wait-flag	一般 TCP 是 nowait,UDP 是 wait
login-name	/etc/passwd 中的用户名,一般为 root
server-program	exec 使用的全路径名
server-program-arguments	exec 使用的参数

图 12.6　inetd.conf 文件中的栏目

上图和示例行只是作为例子。许多厂商都在 inetd 中加入了一些自己的功能。如在 TCP 和 UDP 服务器之外,加入处理远程过程调用(RPC)服务器的能力,以及处理除 TCP 和 UDP 外其他协议的能力。当然,exec 服务器程序使用的路径名和命令行参数依赖于具体的实现。

IPv6 与/etc/inetd.conf 的交互依赖于厂商的实现。一些厂商用名为 tcp6 和 udp6 的 protocol 表示应为该服务创建一个 IPv6 套接口。

图 12.7 给出了 inetd 守护进程的工作流程:

1. 启动时读/etc/inetd.conf 文件并给文件中指定的所有服务创建一个相应类型的套接口(字节流或数据报)。inetd 能处理的服务器的数目依赖于它最多能创建的描述字的数目。每个新创建的套接口都被加入到 select 调用所用到的描述字集中。
2. 为每个套接口调用 bind,给它们捆绑服务器的众所周知端口和通配地址。它们的 TCP 或 UDP 端口号是通过调用 getservbyname 获得的,其中使用了配置文件中的 service-name 和 protocol 栏目作为参数。
3. 对 TCP 套接口调用 listen,以接受外来的连接请求。对数据报套接口则不做这一步。
4. 所有套接口建立后,调用 select 等待这些套接口变为可读。回想 6.3 节里提到过,当在 TCP 套接口上到来一个新的连接请求或在 UDP 套接口上到来一个数据报时它们会变成可读。inetd 在大部分时间里阻塞在 select 调用上,等待有一个套接口变成可读。
5. select 返回一个可读的套接口后,如果是一个 TCP 套接口,就调用 accept 接受这个新的连接。
6. inetd 守护进程 fork,由子进程处理服务请求。这和标准的并发服务器(4.8 节)类似。

子进程关闭除要处理的套接口描述字之外的所有描述字:对 TCP 服务器来说是由 accept 返回的新的已连接套接口,对 UDP 服务器则是最初的那个 UDP 套接口。

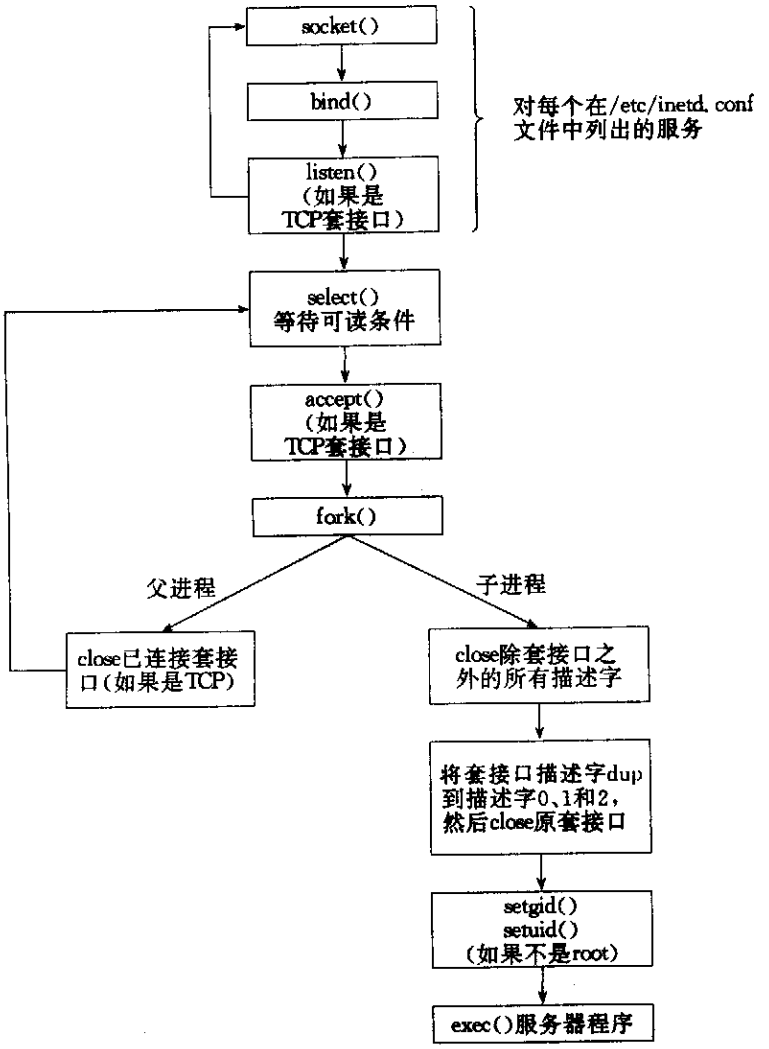


图 12.7 inetd 的工作流程

子进程三次调用 dup2,把套接口描述字复制到描述字 0、1 和 2(标准输入、标准输出和标准错误输出)。然后关闭原套接口描述字。这样,子进程打开的描述字就只有 0、1 和 2。如果子进程读标准输入,它实际是从套接口读,写标准输出和标准错误输出也是写到套接口。

子进程调用 getpwnam 得到在配置文件中指定的 login-name 对应的保密字文件项。如果 login-name 不是 root,子进程会调用 setgid 和 setuid 变为指定的用户。(因为 inetd 是以用户 ID 为 0 运行,子进程跨 fork 继承了这个用户 ID,所以它能变成任何用户。)

子进程用 exec 执行相应的 server-program 处理请求,并将配置文件中指定的参数传递给它。

7. 如果是一个字节流套接口,父进程必须关闭已连接套接口(就像标准并发服务器那

样)。父进程再调用 select 以等待下一个变成可读的套接口。

如果我们要更深入地了解对描述字的处理,图 12.8 展示了有一个新的从 FTP 客户来的连接请求时,inetd 中的描述字。

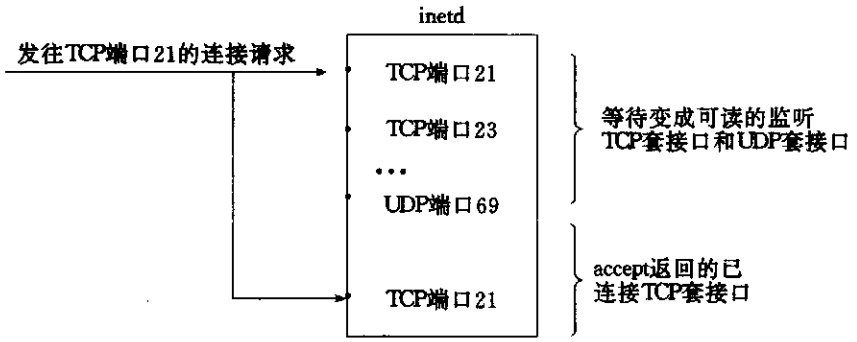


图 12.8　当 TCP 端口 21 的连接请求到来时,inetd 中的描述字

连接请求指向 TCP 的 21 号端口,但 accept 创建了一个新的已连接套接口。

图 12.9 展示了子进程在调用 fork 及关闭除已连接套接口描述字外的所有描述字之后剩下的描述字。

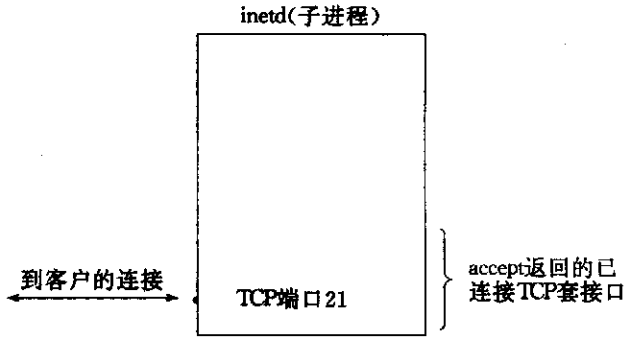


图 12.9　inetd 子进程中的描述字

下一步是子进程把已连接套接口描述字复制到描述字 0、1 和 2,然后关闭该套接口。如图 12.10 所示。

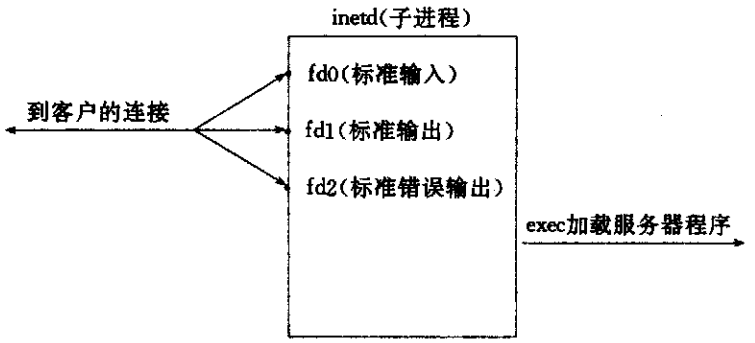


图 12.10　dup2 后 inetd 中的描述字

子进程接着调用 exec。回想 4.7 节，exec 后所有的描述字通常还是打开的，因此加载的实际服务器程序用描述字 0、1 或 2 与客户进行通信。服务器只打开这些描述字。

以上介绍的是对配置文件中设置为 nowait 方式的服务器的处理。对 TCP 服务来说这种情况十分典型，它意味着 inetd 在接受请求同一服务的其他连接之前不需要等待该服务的子进程终止。如果有同一服务的其他连接请求到来，父进程再调用 select 时就会将其返回。再次执行前面列出的第 4、第 5 和第 6 步操作，由另一个子进程处理这个新的请求。

给数据报服务设置 wait 标志需要对父进程所做的操作步骤作一定的修改。这个标志是说 inetd 在该 UDP 套接口上再次选择之前，必须等待在该套接口上服务的子进程终止。于是有下面的这些改变：

1. 父进程中的 fork 返回时，把子进程的进程号记录下来。这样在子进程终止时，父进程可以用 waitpid 的返回值查知是哪一个子进程。
2. 父进程用 FD_CLR 宏关闭 select 使用的描述字集中与这个套接口对应的位，以不对该套接口作 select。这意味着子进程接管这个套接口直至其终止。
3. 当子进程终止时，父进程收到一个 SIGCHLD 信号，父进程的信号处理程序得到终止子进程的进程号。父进程通过打开描述字集中对应的位恢复对该套接口的 select。

数据报服务器必须接管套接口直至它终止，防止 inetd 在这个套接口继续做可读性的 select（等待另一个客户的数据报），这是因为一个数据报服务器只有一个套接口，不像 TCP 服务器那样有一个监听套接口，每个客户有一个已连接套接口。如果 inetd 不关闭对这个数据报套接口的可读条件检查，而且父进程（inetd）在子进程前面执行，那么从客户来的数据报会还在套接口的接收缓冲区里，导致 select 再次返回可读，inetd 于是 fork 另一个（不需要的）子进程。inetd 必须在它确信子进程已从套接口接收队列中读走数据报前忽略这个数据报套接口。inetd 通过收到 SIGCHLD 信号的方式了解子进程何时终止。在 20.7 节中有这样的例子。

图 2.13 中介绍的五种标准因特网服务是由 inetd 内部处理的。（见习题 12.2。）

因为替 TCP 服务器调用 accept 的是 inetd，所以实际的服务器一般用 getpeername 获得客户的 IP 地址和端口号。回想图 4.18 中展示了在 fork 和 exec 后（inetd 就这么做）实际的服务器识别客户的唯一方法是调用 getpeername。

需要提供大量服务的服务器通常不用 inetd，其中值得注意的是邮件和 Web 服务器。举例来说，在 4.8 节中介绍的 sendmail 通常是以一个标准的并发服务器来运行。这种方式下每个客户连接在进程控制上的开销只是一个 fork，而一个通过 inetd 的 TCP 服务器的开销是一个 fork 和一个 exec。Web 服务器使用了多种技术把每个客户连接的进程控制开销减到最小，我们会在第 27 章中对此进行讨论。

12.6 daemon_inetd 函数

图 12.11 给出了一个名为 daemon_inetd 的函数，我们可以在一个由 inetd 启动的服务器程序中调用它。

```
1 #include    "unp.h"
```

```
2 #include    <syslog.h>
3 extern int daemon_proc;          /* defined in error.c */
4 void
5 daemon_inetd(const char *pname, int facility)
6 {
7     daemon_proc = 1;        /* for our err_XXX() functions */
8     openlog(pname,LOG_PID,facility);
9 }
```

图 12.11　daemon_inetd 函数：使通过 inetd 运行的进程变为守护进程[daemon_inetd.c]

这个函数与 daemon_init 比起来要简单，因为变成守护进程的工作都由 inetd 在启动时做过了。唯一要做的就是给错误处理函数(附录中的图 D.4)设置 daemon_proc 标志，并使用与图 12.4 中的调用相同的参数调用 openlog。

例子：由 inetd 启动的时间/日期服务器程序

图 12.12 是图 12.5 中的时间/日期服务器程序的一个修改版本，它可以由 inetd 启动。

```
1 #include    "unp.h"
2 #include    <time.h>
3 int
4 main(int argc, char **argv)
5 {
6       socklen_t          len;
7       struct sockaddr    *cliaddr;
8       char               buff[MAXLINE];
9       time_t                    ticks;
10      daemon_inetd(argv[0], 0);
11      cliaddr = Malloc(MAXSOCKADDR);
12      len = MAXSOCKADDR;
13      Getpeername(0, cliaddr, &len);
14      err_msg("connection from %s", Sock_ntop(cliaddr, len));
15    ticks = time(NULL);
16    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
17    Write(0, buff, strlen(buff));
18      Close(0);    /* close TCP connection */
19      exit(0);
20 }
```

图 12.12　可由 inetd 启动的独立于协议的时间/日期服务器程序[inetd/daytimetcpsrv3.c]

这个程序中有两个大的改动。首先，所有创建套接口的代码即对 tcp_listen 和 accept 的调用都没有了。这些工作都由 inetd 做了，我们可以用描述字 0(标准输入)来引用 TCP 连接。其次，去掉了 for 的无限循环，因为对每个客户连接都会启动一次程序。服务完这个客户后程序即终止。

调用 getpeername

第 11～14 行　因为程序中没有调用 tcp_listen，所以不知道它返回的套接口地址结构的大小，而且由于没有调用 accept，也不知客户的协议地址。因此用 MAXSOCKADDR 常值

给套接口地址结构分配缓冲区，并用描述字0作第一个参数调用getpeername。

为了在BSD/OS系统上运行这个例子程序，首先给它指派一个名字和端口，将下面一行加到/etc/services文件中：

```
mydaytime      9999/tcp
```

然后在/etc/inetd.conf加一行：

```
mydaytime  stream  tcp  nowait  rstevens
    /usr/home/rstevens/daytimetcpsrv3  daytimetcpsrc3
```

(这行因过长而被折行。)把执行代码放在指定的文件中并向inetd发SIGHUP信号，通知它重新读入配置文件。下一步是执行netstat检验是否已创建了一个在TCP的端口9999上的监听套接口：

```
dsdi % netstat -na | grep 9999
tcp        0  0      *.9999          *.*          LISTEN
```

然后从另外一台主机访问这个服务器：

```
alpha % telnet bsdi 9999
Trying 206.62.226.35 ...
Connected to bsdi.
Escape character is '^]'.
Thu Jun 5 11:13:50 1997
Connection closed by foreign host.
```

在/var/log/messages文件(已在/etc/syslog.conf文件中指定LOG_USER设施的消息登记到该文件)中会有以下一项：

```
Jun 5 11:13:50 bsdi daytimetcpsrv3[28724]: connection from 206.62.226.42.1042
```

12.7 小 结

守护进程是在后台运行的，独立于所有终端控制的进程。多数网络服务器是以守护进程的方式运行。守护进程的所有输出通常是调用syslog函数发送给syslogd。管理员可以根据发送消息的守护进程以及消息的紧急程度，完全控制如何处理这些消息。

启动一个程序并使其以守护进程的方式运行需要以下步骤：调用fork以转到后台运行，调用setsid建立一个新的Posix.1会话并成为会话头，再次fork以避免获得新的控制终端，改变工作目录和文件模式创建掩码，并关闭所有不需要的文件。daemon_init函数处理了以上的这些问题。

多数Unix服务器是由inetd守护进程启动的。它处理了所有转变成守护进程所需的步骤，在启动真正的服务器时套接口已在标准输入、标准输出、标准错误输出上打开。这样我们就不必再调用socket、bind、listen和accpet了，因为这些操作inetd已经做了。

12.8　习　题

12.1　图 12.5 中如果直到处理完命令行参数后再调用 daemon_init，使调用 err_quit 在程序变成守护进程之前出现会发生什么？

12.2　对于由 inetd 内部处理的 5 种服务（图 2.13），考虑每种服务有 TCP 和 UDP 两个版本，你认为这 10 个服务器在实现时哪些需要调用 fork，哪些不需要调用 fork？

12.3　如果创建一个 UDP 套接口，将其绑定端口 7（图 2.13 中的标准 echo 服务器），然后向 chargen 服务器发送一个 UDP 数据报，将会发生什么？

12.4　Solaris 2.x 中 inetd 的手册页面中介绍了一个-t 标志，它会使 inetd 调用 syslog（用设施 LOG_DAEMON 和级别 LOG_NOTICE）登记所有 inetd 处理的 TCP 服务的客户的 IP 地址和端口号。inetd 是如何获得这个信息的？

该手册页面中还说 inetd 不能对 UDP 服务做上述的动作。为什么？有办法绕过对 UDP 服务的这个限制吗？

第13章 高级I/O函数

13.1 概 述

本章覆盖了各种被归为"高级I/O"的函数和技术。首先是为一个I/O函数设置超时,这有三种方法。

然后是三个read和write函数的变体:recv和send,它们可以把含有标志的第四个参数从进程传给内核;readv和writev,这两个函数可以指定一个缓冲区的向量以输入或输出数据;以及recvmsg和sendmsg,在其他I/O函数的所有功能基础上结合了新的接收和发送辅助数据的能力。

本章中还介绍了如何确定在套接口接收缓冲区中有多少数据,如何在套接口上使用C的标准I/O库。本章最后概要介绍一下T/TCP即事务TCP,它可以避免三路握手。

13.2 套接口超时

有三种方法给套接口上的I/O操作设置超时。

1. 调用alarm,在到达指定时间时产生SIGALRM信号。这涉及到信号处理,这一点随不同实现而变化,而且可能与进程中其他已有的alarm调用冲突。
2. 使用select阻塞在等待I/O上,select内部有一个时间限制,以此代替在read或write调用上阻塞。
3. 使用新的SO_RCVTIMEO和SO_SNDTIMEO套接口选项。这种方法存在一个问题,即并不是所有的实现都能支持这两个套接口选项。

这三种技术都可以用于输入和输出操作(例如,read、write和其他的一些变种如recvfrom和sendto),但是我们仍想要一种可用于connect的技术,因为TCP的connect的超时时间很长(典型情况为75秒)。只有在套接口为非阻塞方式时(在15.3节中将会介绍),才可以用select为connect设置超时时间,而且那两个新的套接口选项对connect不起作用。值得注意的是前两种技术可以用于任何描述字,而第三种技术只能用于套接口描述字。

我们下面举例说明所有这三种技术。

用SIGALRM给connect设置超时

图13.1给出了函数connect_timeo的代码,它以调用者设定的时间上限来调用connect。前三个参数是connect所需的,第四个参数是等待的秒数。

```
1 #include    "unp.h"
2 static void connect_alarm(int);
3 int
```

```
4 connect_timeo(int sockfd, const SA *saptr, socklen_t salen, int nsec)
5 {
6     Sigfunc       *sigfunc;
7     int           n;
8     sigfunc = Signal(SIGALRM, connect_alarm);
9     if (alarm(nsec) != 0)
10        err_msg("connect_timeo: alarm was already set");
11    if ( (n = connect(sockfd, (struct sockaddr *) saptr, salen)) < 0) {
12        close(sockfd);
13        if (errno == EINTR)
14            errno = ETIMEDOUT;
15    }
16    alarm(0);                    /* turn off the alarm */
17    Signal(SIGALRM, sigfunc);    /* restore previous signal handler */
18    return(n);
19 }
20 static void
21 connect_alarm(int signo)
22 {
23    return;                      /* just interrupt the connect() */
24 }
```

图 13.1　带超时的 connect[lib/connect_timeo.c]

建立信号处理程序

第 8 行　为 SIGALRM 建立一个信号处理程序。现有的信号处理程序(如果有的话)被保存,因此它可以在这个函数结束时恢复。

设置报警(时钟)

第 9～10 行　进程的报警时钟设成调用者指定的秒数。alarm 的返回值是进程的报警时钟(如果进程设置过一个报警时钟)现在剩下的秒数或 0(如果进程当前没有设置报警)。对于前一种情况显示一个警告信息,因为在设置新的报警时钟时去掉了前面设置的报警时钟。

调用 connect

第 11～15 行　调用 connect,如果被中断(EINTR),则将 errno 的值设为 ETIMEOUT。关闭该套接口以防止继续进行三路握手。

关闭 alarm 并恢复原来的信号处理程序

第 16～18 行　将 alarm 的参数设成 0 以关闭 alarm,并恢复原来的信号处理程序(如果有的话)。

处理 SIGALRM

第 20～24 行　信号处理程序只是简单地返回,但它将中断挂起的 connect,使 connect 返回 EINTR 错误。回想我们的 signal 函数(图 5.6)在捕获信号为 SIGALRM 时没有设置 SA_RESTART 标志。

这个例子中值得注意的一点是我们可以用这种技术减少 connect 的超时时间,但不能延长内核中已有的超时时间。也就是说,在源自 Berkeley 的内核中 connect 的超时时间一般为 75 秒。我们可以在上面的函数中设定一个比它小的超时值,比方说 10,但不能设定一个

比它大的值，譬如80，这样的话connect仍会在75秒后超时。

另外一点是我们使用系统调用(connect)的可中断性在内核超时之前返回。这在我们执行该系统调用并能处理返回的EINTR错误时是没有问题的。但在26.6节中我们会遇到一个执行系统调用的库函数，而且这个库函数在系统调用返回EINTR时重新发出系统调用。在这种情况下仍能使用SIGALRM，但在图26.10中可以看到，我们还必须使用sigsetjmp和siglongjmp以绕过函数库对EINTR的忽略。

用SIGALRM为recvfrom设置超时

图13.2是图8.8中的dg_cli函数的一个改写版本，如果在5秒内没有收到应答，recvfrom将被alarm中断。

处理recvfrom的超时

第8～22行 为SIGALRM建立一个信号处理程序，然后在每次recvfrom前调用alarm设置5秒的超时。如果recvfrom被信号处理程序中断，就输出一条信息并继续运行。如果从服务器读到一行，就关闭alarm，并输出返回的应答。

```
1 #include    "unp.h"
2 static void sig_alrm(int);
3 void
4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
5 {
6     int    n;
7     char   sendline[MAXLINE], recvline[MAXLINE + 1];
8     Signal(SIGALRM, sig_alrm);
9     while (Fgets(sendline, MAXLINE, fp) != NULL) {
10        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
11        alarm(5);
12        if ( (n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL)) < 0) {
13            if (errno == EINTR)
14                fprintf(stderr, "socket timeout\n");
15            else
16                err_sys("recvfrom error");
17        } else {
18            alarm(0);
19            recvline[n] = 0;   /* null terminate */
20            Fputs(recvline, stdout);
21        }
22    }
23 }
24 static void
25 sig_alrm(int signo)
26 {
27    return;                    /* just interrupt the recvfrom() */
28 }
```

图13.2 用alarm使recvfrom超时的dg_cli函数[advio/dgclitimeo3.c]

SIGALRM信号处理程序

第24～28行 信号处理程序只是简单地返回，以中断被阻塞的recvfrom。

因为每次建立alarm时只读一个应答，所以这个例子能正确工作。在18.4节中使用了

相同的技术，但由于用一个 alarm 读多个应答，因此必须处理存在的竞争。

用 select 为 recvfrom 设置超时

在图 13.3 中给出了第二种设置超时的技术(使用 select)的例子。其中有一个名为 readable_timeo 的函数，以指定的秒数等待一个描述字变成可读。

```
1 #include    "unp.h"
2 int
3 readable_timeo(int fd, int sec)
4 {
5     fd_set     rset;
6     struct timeval   tv;
7     FD_ZERO(&rset);
8     FD_SET(fd, &rset);
9     tv.tv_sec = sec;
10    tv.tv_usec = 0;
11    return(select(fd+1, &rset, NULL, NULL, &tv));
12          /* > 0 if descriptor is readable */
13 }
```

图 13.3　readable_timeo 函数：等待一个描述字变成可读[lib/readable_timeo.c]

给 select 准备参数

第 7～10 行　在读描述字集中打开要读的描述字所对应的位。timeval 结构设成调用者要等待的秒数。

阻塞在 select 上

第 11～12 行　select 等待描述字变成可读，或者超时。这个函数的返回值就是 select 的返回值：出错为－1，超时为 0，大于 0 则是已就绪的描述字的数目。

这个函数不执行读操作；它只是等待描述字变成可读。因此这个函数可以用在任何类型的套接口上：TCP 或 UDP。

建立类似的名为 writeable_timeo 的函数，等待一个描述字变成可写是很简单的。

在图 13.4 中将使用这个函数重写图 8.8 中的 dg_cli 函数。这个新版本只在 readable_timeo 返回一个大于零的值时才调用 recvfrom。

直到 readable_timeo 告诉我们描述字可读后我们才调用 recvfrom。这保证了 recvfrom 不会阻塞。

```
1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int    n;
6     char   sendline[MAXLINE], recvline[MAXLINE + 1];
7     while (Fgets(sendline, MAXLINE, fp) != NULL) {
8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
9         if (Readable_timeo(sockfd, 5) == 0) {
10            fprintf(stderr, "socket timeout\n");
11        } else {
12            n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
```

```
13              recvline[n] = 0;   /* null terminate */
14              Fputs(recvline, stdout);
15          }
16      }
17}
```

图 13.4 调用 readable_timeo 以设置超时的 dg_cli 函数[advio/dgclitimeo1.c]

用 SO_RCVTIMEO 套接口选项为 recvfrom 设置超时

最后一个例子是说明 SO_RCVTIMEO 套接口选项的用法。一旦为每个描述字设置了这个选项，并指定了超时值，那么这个超时对该描述字上的所有读操作都起作用。这种方法的好处是只需要设置一次选项，而前两种方法需要在每次操作前都设一遍超时。但这个套接口选项只对读操作起作用，与此类似 SO_SNDTIMEO 只对写操作起作用，它们都不能为 connect 设置超时。

图 13.5 是使用了 SO_RCVTIMEO 套接口选项的 dg_cli 函数的另一个版本。

设置套接口选项

第 8～10 行 setsockopt 的第四个参数是一个指向 timeval 结构的一个指针，其中填入超时时间。

测试超时

第 15～17 行 如果 I/O 操作超时，函数(这里是 recvfrom)会返回 EWOULDBLOCK。

13.3 recv 和 send 函数

这两个函数和标准的 read 和 write 函数都很类似，只是多了一个附加的参数。

```
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);

ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);

                                返回：成功返回读入或写出的字节数，出错返回-1
```

```
1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int          n;
6     char         sendline[MAXLINE], recvline[MAXLINE + 1];
7     struct timeval   tv;
8     tv.tv_sec = 5;
9     tv.tv_usec = 0;
10    Setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
11    while (Fgets(sendline, MAXLINE, fp) != NULL) {
12        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
13        n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
```

```
14          if (n < 0) {
15              if (errno == EWOULDBLOCK) {
16                  fprintf(stderr, "socket timeout\n");
17                  continue;
18              } else
19                  err_sys("recvfrom error");
20          }
21          recvline[n] = 0;    /* null terminate */
22          Fputs(recvline, stdout);
23      }
24  }
```

图 13.5　用 SO_RCVTIMEO 套接口选项设置超时的 dg_cli 函数[advio/dgclitimeo2.c]

recv 和 send 的前三个参数与 read 和 write 相同。参数 flags 的值或为 0，或由在图 13.6 中列出的一个或多个常值的逻辑或构成。

flags	描述	recv	send
MSG_DONTROUTE	不查路由表		•
MSG_DONTWAIT	本操作不阻塞	•	•
MSG_OOB	发送或接收带外数据	•	•
MSG_PEEK	查看外来的消息	•	
MSG_WAITALL	等待所有数据	•	

图 13.6　I/O 函数的 flags

MSG_DONTROUTE　这个标志告诉内核目的主机在直接连接的本地网络上，不要查路由表。这是对提供这种特性的 SO_DONTROUTE 套接口选项(7.5节)的补充。用 MSG_DONTROUTE 标志可以对单个输出操作提供这种特性，而套接口选项则针对某个套接口上的所有输出操作。

MSG_DONTWAIT　这个标志将单个 I/O 操作设为非阻塞方式，而不需要在套接口上打开非阻塞标志，执行 I/O 操作，然后关闭非阻塞标志。在第 15 章我们将介绍非阻塞 I/O 以及在一个套接口上对所有 I/O 操作开关非阻塞标志。

这个标志是 Net/3 中新增加的，不一定在所有系统中都支持。

MSG_OOB　用 send 时，这个标志指明发送的是带外数据。如在第 21 章中介绍的那样，TCP 连接上只能发送 1 个字节的带外数据。用 recv 时，这个标志指明要读的是带外数据而不是一般数据。

MSG_PEEK　这个标志可以让我们查看可读的数据，在 recv 或 recvfrom 后系统不会将这些数据丢弃。在 13.7 节中会更详细地谈这个问题。

MSG_WAITALL 这个标志由4.3BSD Reno引入。它告诉内核在没有读到请求的字节数之前不使读操作返回。如果系统支持这个标志，我们就可以去掉readn函数(图3.14)而用下面的宏来代替：

```
#define readn(fd, ptr, n)  recv(fd, ptr, n, MSG_WAITALL)
```

即使设定了MSG_WAITALL，如果发生下列情况：(a)捕获一个信号，(b)连接被终止，或(c)在套接口上发生错误，这个函数返回的字节数仍会比请求的少。

除了TCP/IP，还有一些用于其他协议的附加标志。例如，OSI的传输层是基于记录的(不像TCP那样基于字节流)，在输出操作时支持MSG_EOR标志指明逻辑记录的结束。T/TCP(事务TCP，在13.9节中介绍)支持一个新的MSG_EOF标志，把一个输出操作和FIN的发送结合起来。

flags参数在设计上存在一个基本问题：它是按值传递的；而不是值-结果参数。因此它只能从进程向内核传递标志。内核不能向进程传回标志。这在用TCP/IP时不构成问题，因为TCP/IP基本不需要从内核向进程传回标志。但当在4.3BSD Reno中加入了OSI协议后，引起了在输入操作时向进程返回MSG_EOR的需要。4.3BSD Reno做出的决定是保持常用的输入函数(recv和recvfrom)的参数不变，改变recvmsg和sendmsg的msghdr结构。在13.5节中将看到在此结构中加入了一个整型的msg_flags成员，因为这个结构是用引用来传递的，所以内核可以在返回时修改这些标志。这也意味着，如果进程需要让内核来更新标志，就必须调用recvmsg代替recv或recvfrom。

13.4 readv和writev函数

这两个函数与read和write相似，但readv和writev可以让我们在一个函数调用中读或写多个缓冲区，这些操作被称为分散读(因为输入数据分散到多个应用缓冲区中)和集中写(因为多个缓冲区被集中到一次写操作中)。

```
#include <sys/uio.h>
ssize_t readv(int filedes,const struct iovec *iov,int iovcnt);
ssize_t writev(int filedes,const struct iovec *iov,int iovcnt);
                                        返回：读到或写出的字节数，出错时为-1
```

两个函数的第二个参数都是一个指向iovec结构的数组的指针，在<sys/uio.h>头文件中定义：

```
struct iovec {
    void *iov_base;     /* starting address of buffer */
    size_t iov_len;     /* size of buffer */
};
```

readv和writev函数在Posix中还没有标准化。但recvmsg和sendmsg函数

(13.5 节)中也使用了 iovec 结构,而且这个结构在 Posix.1g 中已被标准化。上面给出的 iovec 结构中的各个成员的数据类型就是在 Posix.1g 中说明的。你可能在一些实现中遇到将 iovec_base 定义为 char *,而把 iov_len 定义为 int 的情况。

在具体的实现中对 iovec 结构数组的元素个数会有一些限制。举例来说,4.3BSD 最多允许 1024 个,而 Solaris 2.5 的上限是 16。Posix.1g 要求在<sys/uio.h>头文件中定义一个常值 IOV_MAX,而且它的值不小于 16。

readv 和 writev 函数可用于任何描述字,不仅限于套接口描述字。而且 writev 是一个原子操作。对一个基于记录的协议如 UDP,调用一次 writev 只产生单个 UDP 数据报。

在 7.9 节中我们提到将 writev 和 TCP_NODELAY 套接口选项一起使用。其中谈到 write 4 字节,接着 write 396 字节会引发 Nagle 算法,对这两个缓冲区调用 writev 是一种更好的解决办法。

13.5　recvmsg 和 sendmsg 函数

这两个函数是最通用的 I/O 函数。实际上,可以用 recvmsg 代替 read、readv、recv 和 recvfrom。同样,各种输出函数都可以用 sendmsg 取代。

```
#include <sys/socket.h>

ssize_t recvmsg(int sockfd,struct msghdr *msg,int flags);

ssize_t sendmsg(int sockfd,struct msghdr *msg,int flags);

                                    返回:成功时为读入或写出的字节数,出错时为-1
```

这两个函数把大部分参数包装到一个 msghdr 结构中:

```
struct msghdr {
  void         *msg_name;        /* protocol address */
  socklen_t     msg_namelen;     /* size of protocol address */
  struct iovec *msg_iov;         /* scatter/gather array */
  size_t        msg_iovlen;      /* # elements in msg_iov */
  void         *msg_control      /* ancillary data; must be aligned for a cmsghdr
                                    structure */
  socklen_t     msg_controllen;  /* length of ancillary data */
  int           msg_flags;       /* flags returned by recvmsg() */
};
```

上面给出的 msghdr 结构源自 4.3BSD Reno,也是 Posix.1g 中所说明的。一些系统(Solaris 2.5)仍使用一种源自 4.2BSD 的老的 msghdr 结构。这种老的结构中没有 msg_flags 成员,而且 msg_control 和 msg_controllen 成员分别被叫做 msg_accrights 和 msg_accrightslen。老系统中支持的唯一一种辅助数据形式是文件描述字(称为访问权限)的传递。4.3BSD Reno 在加入 OSI 协议时增加了更多形式的辅助数据,因此使这个结构中的成员名更为通用化了。

msg_name 和 msg_namelen 成员用于未经连接的套接口(譬如未连接 UDP 套接口)。它们与 recvfrom 和 sendto 的第五和第六个参数类似:msg_name 指向一个套接口地址结构,调用者在其中存放对 sendmsg 来说是目的方的协议地址,对 recvmsg 来说是发送方的协议地址。如果不需要指明协议地址(譬如 TCP 套接口或已连接 UDP 套接口),msg_name 应被设成空指针。msg_namelen 对 sendmsg 是一个值,而对 recvmsg 是一个值-结果参数。

msg_iov 和 msg_iovlen 成员指明输入或输出的缓冲区数组(iovec 结构的数组),这与 readv 或 writev 的第二个和第三个参数相似。

msg_control 和 msg_controllen 成员指明可选的辅助数据的位置和大小,msg_controllen 对 recvmsg 是一个值-结果参数,我们会在 13.6 节中介绍辅助数据。

使用 recvmsg 和 sendmsg 时我们必须区别两个标志变量:传值的 flags 参数和 msghdr 结构中的 msg_flags 成员,它是以引用方式传递的(因为给函数传递的是 msghdr 结构的地址)。

- msg_flags 只用于 recvmsg。调用 recvmsg 时,flags 参数被拷贝到 msg_flags 成员(TCPv2 第 502 页),而且内核用这个值进行接收处理,接着它的值会根据 recvmsg 的结果而更新。
- sendmsg 会忽略 msg_flags 成员,因为它在进行输出处理时使用 flags 参数。这意味着如果我们在调用 sendmsg 时要设置 MSG_DONTWAIT 标志,应将 flags 设成该值;设置 msg_flags 为该值没用。

图 13.7 总结了使用输入和输出函数时内核检查的标志,以及 recvmsg 可能返回的 msg_flags。因为前面已经提到 msg_flags 对 sendmsg 没用,所以图中 sendmsg msg_flags 没有对应的栏。

标志	在 send flags sendto flags sendmsg flags 中检查	在 recv flags recvfrom flags recvmsg flags 中检查	在 recvmsg msg_flags 中 返回
MSG_DONTROUTE	•		
MSG_DONTWAIT	•	•	
MSG_PEEK		•	
MSG_WAITALL		•	
MSG_EOR	•		•
MSG_OOB	•	•	•
MSG_BCAST			•
MSG_MCAST			•
MSG_TRUNC			•
MSG_CTRUNC			•

图 13.7 各种 I/O 函数输入和输出标志的总结

前四个标志只检查不返回,下两个既检查又返回,最后四个只返回。下面是对 recvmsg 返回的六个标志的解释:

MSG_BCAST	这个标志是BSD/OS中新出现的，当收到的数据报是一个链路层的广播或其目的IP地址为广播地址时，将返回此标志。与IP_RECVDSTADDR套接口选项相比，该标志是判断UDP数据报是否发往广播地址的一种更好的方法。
MSG_MCAST	这个标志是BSD/OS中新出现的，当收到的数据报是链路层的多播时，将返回该标志。
MSG_TRUNC	这个标志在数据报被截断时返回：内核有比进程所分配的空间(所有iov_len成员的总和)所能容纳的还要多的数据待返回。我们会在20.3节中进一步讨论这个问题。
MSG_CTRUNC	这个标志在辅助数据被截断时返回：内核有比进程所分配的空间(msg_controllen)所能容纳的还要多的辅助数据待返回。我们在20.3节中会详细讨论。
MSG_EOR	如果返回的数据不是一个逻辑记录的结尾，该标志将清位，反之则置位。TCP不使用这个标志，因为它是一种字节流协议。
MSG_OOB	这个标志不是为TCP的带外数据返回的。它用于其他协议族(譬如OSI协议等)。

实现可能会在msg_flags成员中返回一些输入flags的标志，因此我们应该只检查那些感兴趣的标志的值(譬如图13.7中的后六个)。

图13.8给出了一个msghdr结构和它指向的各种信息。图中假定进程是对一个UDP套接口调用recvmsg。

给协议地址分配了16个字节，辅助数据分配了20个字节。初始化一个有三个iovec结构的数组：第一个指定一个100字节的缓冲区，第二个是60字节的缓冲区，第三个是80字节的缓冲区。我们还假定该套接口已设置了IP_RECVDSTADDR套接口选项，以接收UDP数据报中的目的IP地址。

假设有一个从198.69.10.2的端口2000到来的170字节的UDP数据报，目的地是我们的UDP套接口，目的IP地址为206.62.226.35。图13.9展示了当recvmsg返回时msghdr结构中的所有信息。

图中阴影的字段是被recvmsg修改的。从图13.8到13.9有以下改变：

- msg_name指向的缓冲区已经填入了一个含有收到的数据报的源IP地址和源UDP端口的网际套接口地址结构。
- 值-结果参数msg_namelen的值更新为msg_name中存放的数据的总量。因为调用前它的值是16，而recvmsg返回的还是16，所以它的值没有改变。
- 数据的前100个字节存放在第一个缓冲区，后续的60字节存放在第二个缓冲区，最后10字节存放在第三个缓冲区。最后的那个缓冲区中的后70个字节没有改变。recvmsg函数的返回值是该数据报的大小即170。
- msg_control指向的缓冲区被填入了一个cmsghdr结构。(在13.6节中对辅助数据有更多的说明，IP_RECVDSTADDR这个套接口选项在20.2节中有详细介绍。)cmsg_len为16，cmsg_level为IPPROTO_IP，cmsg_type为IP_RECVDSTADDR，

下四个字节是收到的UDP数据报中的目的IP地址。这个20字节缓冲区中用来存放辅助数据的最后4个字节没有被修改。

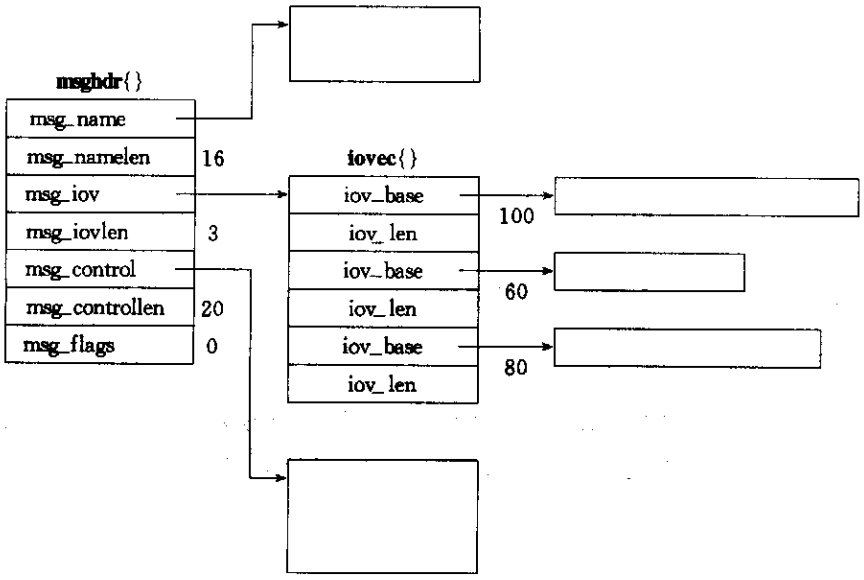


图13.8 对一个UDP套接口调用recvmsg时的数据结构

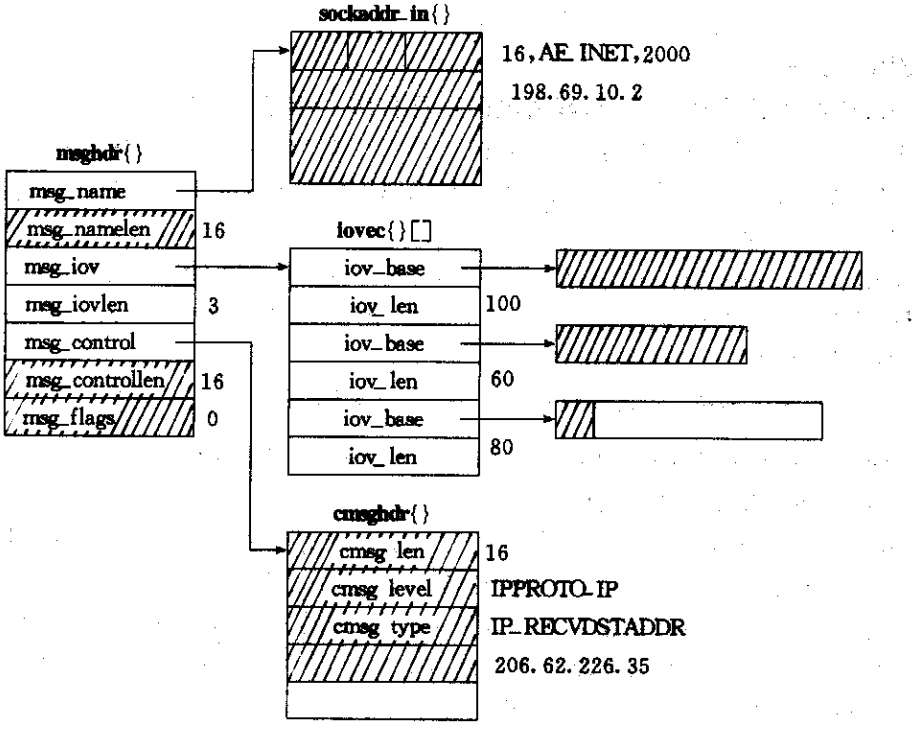


图13.9 recvmsg返回时对图13.8的更新

- msg_controllen成员更新为存放的辅助数据的实际数量，它也是值-结果参数，返回时它的值为16。

• msg_flags 成员被 recvmsg 更新，但是没有给进程返回任何标志。

图 13.10 小结了我们已述的五组 I/O 函数的差异。

函数	任何描述字	只对套接口描述字	单个 read/write 缓冲区	分散/集中 read/write	可选标志	可选对方地址选项	可选控制信息
read，write	•		•				
readv，writev	•			•			
recv，send		•	•		•		
recvfrom，sendto		•	•		•	•	
recvmsg，sendmsg		•		•	•	•	•

图 13.10　五组 I/O 函数的比较

13.6　辅助数据

可以在 sendmsg 和 recvmsg 时使用 msghdr 结构中的 msg_control 和 msg_controllen 成员发送和接收辅助数据(ancillary data)。辅助数据的另一种叫法是控制信息(control information)。本节中介绍这个概念并给出用来建立和处理辅助数据的结构和宏，介绍辅助数据实际使用的源代码在以后章节中再给出。

图 13.11 是对本文中辅助数据的各种用法的一个总结。

协议	cmsg_level	cmsg_type	说明
IPv4	IPPROTO_IP	IP_ RECVDSTADDR	接收 UDP 数据报的目的地址
		IP_RECVIF	接收 UDP 数据报的接口索引
IPv6	IPPROTO_IPV6	IPV6_DSTOPTS	指定/接收目标选项
		IPV6_HOPLIMIT	指定/接收跳限
		IPV6_HOPOPTS	指定/接收步跳选项
		IPV6_NEXTHOP	指定下一跳地址
		IPV6_PKTINFO	指定/接收分组信息
		IPV6_RTHDR	指定/接收路由头部
Unix 域	SOL_SOCKET	SCM_RIGHTS	发送/接收描述字
		SCM_CREDS	发送/接收用户凭证

图 13.11　辅助数据用法的总结

OSI 协议族也有辅助数据用于各种目的，这里我们就不做讨论了。

辅助数据由一个或多个辅助数据对象组成，每个对象由一个 cmsghdr 结构开头，该结构在<sys/socket.h>文件中定义如下：

```
struct cmsghdr {
  socklen_t     cmsg_len;     /* length in bytes, including this structure */
  int           cmsg_level;   /* originating protocol */
  int           cmsg_type;    /* protocol-specific type */
```

```
    /* followed by unsigned char cmsg_data[] */
};
```

我们已经在图13.9中见到过这个结构，当时是在设置IP_RECVDSTADDR套接口选项下用于返回接收到的UDP数据报的目的IP地址。msg_control指向的辅助数据必须按csmghdr结构进行对齐。在图14.11中展示了一种对齐方法。

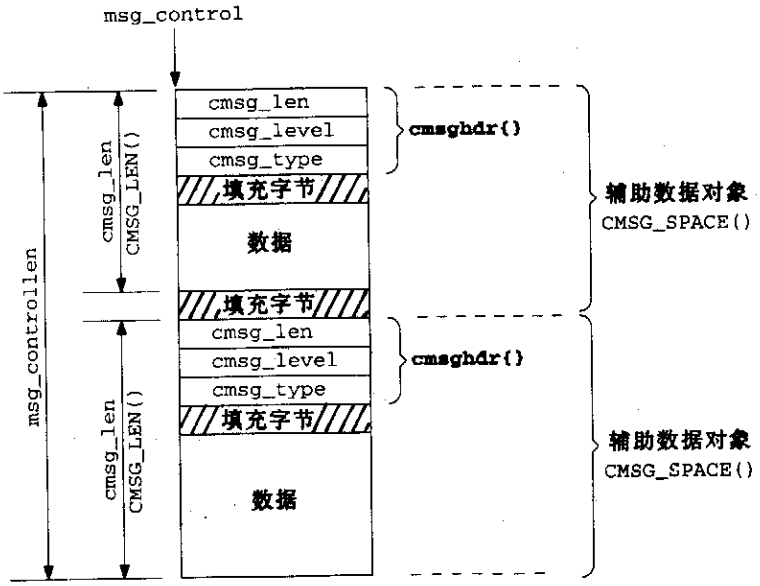


图13.12 包含两个辅助数据对象的辅助数据

图13.12展示了在一个控制缓冲区中的两个辅助数据对象的例子。msg_control指向第一个辅助数据对象，辅助数据的总长度由msg_controllen指定。每个对象开头是一个描述该对象的cmsghdr结构。在cmsg_type成员和实际的数据之间可能有填充字节，在数据之后、下一个对象之前也可能有填充字节。下面简要介绍的五个宏解决这种可能的填充问题。

不是所有的实现都支持在一个控制缓冲区中有多个辅助数据对象。

图13.13展示了当使用Unix域套接口传递描述字(14.7节)或传递凭证(14.8节)时，cmsghdr结构的格式。

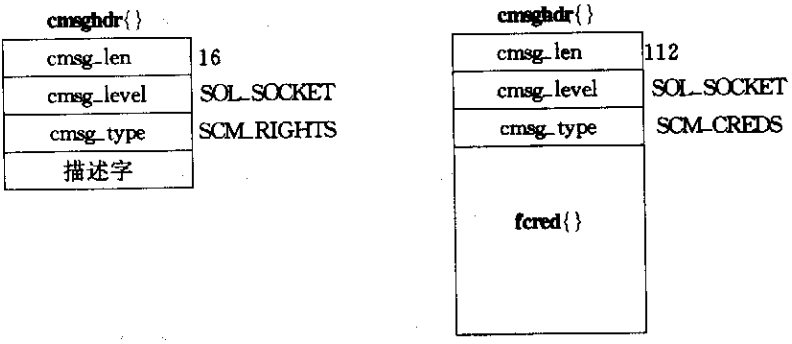


图13.13 使用Unix域套接口时的cmsghdr结构

在这幅图中，我们假定 cmsghdr 结构中的三个成员每个都占 4 字节，在 cmsghdr 结构和实际数据之间没有填充字节。传递描述字时 cmsg_data 数组的内容是实际的描述字的值。在上图中只展示了一个文件描述字的传递，但一般能传递多个描述字（这种情况下 cmsg_len 的值为 12 加上 4 乘描述字的数目，这里假定每个描述字占 4 个字节）。

因为由 recvmsg 返回的辅助数据可以包含任意数目的辅助数据对象，为了对应用程序屏蔽可能出现的填充字节，在<sys/socket.h>中定义了以下五个宏，以简化对辅助数据的处理。

```
#include <sys/socket.h>
#include <sys/param.h>      /* for ALIGN macro on many implementations */

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mhdrptr);
                        返回：指向第一个 cmsghdr 结构的指针，无辅助数据时为 NULL

struct cmsghdr *CMSG_NXTHDR(struct msghdr *mhdrptr,struct cmsghdr *cmsgptr);
                返回：指向下一个 cmsghdr 结构的指针，不再有辅助数据对象时为 NULL

unsigned char *CMSG_DATA(struct cmsghdr *cmsgptr);
                        返回：指向与 cmsghdr 结构关联的数据的第一个字节的指针

unsigned int CMSG_LEN(unsigned int length);
                                    返回：给定数据量下存储在 cmsg_len 中的值

unsigned int CMSG_SPACE(unsigned int length);
                                    返回：给定数据量下一个辅助数据对象的总大小
```

Posix.1g 定义了前三个宏，[Stevens and Thomas 1997]定义了后两个。

这些宏可以用在下面的伪代码中：

```
struct msghdr    msg;
struct cmsghdr   *cmsgptr;

/* fill in msg structure */
/* call recvmsg() */

for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
     cmsgptr = CMSG_NXTHDR(&msg,cmsgptr)) {
    if (cmsgptr->cmsg_level == ... &&
        cmsgptr->cmsg_type == ...) {
        u_char  *ptr;

        ptr = CMSG_DATA(cmsgptr);
        /* process data pointed to by ptr */
    }
}
```

CSMG_FIRSTHDR 返回指向第一个辅助数据对象的指针，如果在 msghdr 结构中没有辅助数据（msg_control 为空指针，或 csmg_len 小于一个 cmsghdr 结构的大小），则返回空指

针。当在控制缓冲区中没有下一个辅助数据对象时，CSMG_NXTHDR 返回空指针。

> 许多现有的 CMSG_FIRSTHDR 的实现并不检查 msg_controllen，直接返回 cmsg_control 的值。在图 20.2 中调用这个宏之前检测了 msg_controllen 的值。

CMSG_LEN 和 CMSG_SPACE 的区别在于，前者不将填充字节计算在内，因而等于 cmsg_len 的值，但后者将这些填充字节都计算在内，因此在动态申请辅助数据对象的数据空间时使用这个值。

13.7 排队的数据量

在不读出数据的情况下，如何知道一个套接口的接收队列中有多少数据可读呢？有三种方法。

1. 如果在没有数据可读时还有其他事情要做，为了不阻塞在内核中，可以使用非阻塞 I/O。在第 15 章中将对此进行介绍。
2. 如果想检查一下数据而使数据仍留在接收队列中，可以使用 MSG_PEEK 标志（图 13.6）。如果想这样做，但又不能肯定是否有数据可读，可以把这个标志和非阻塞套接口相结合，或与 MSG_DONTWAIT 标志结合使用。

   要注意的是，对一个字节流套接口来说，接收队列中的数据量在两次连续的 recv 调用之间可能会改变。举例来说，假设在一个 TCP 套接口上通过指定一个 1024 字节的缓冲区和 MSG_PEEK 标志调用一次 recv，返回值是 100。如果再次调用 recv，返回值可能会超过 100 字节（假定指定的缓冲区长度大于 100），因为在两次调用之间可能接收到更多的数据。

   如果是一个 UDP 套接口，并且在接收队列中有一个数据报的情况下，指定 MSG_PEEK 标志调用 recvfrom，接着是一个不使用 MSG_PEEK 的 recvfrom 调用，这两个调用的返回值是完全相同的（数据报的大小、内容和发送方的地址），即使在两次调用之间有另外的数据报加入套接口接收缓冲区也不会改变。（这里有一个前提是其他进程没有共享该描述字并在此时从该套接口上读取数据。）
3. 一些实现支持 ioctl 的 FIONREAD 命令。ioctl 的第三个参数是一个指向整数的指针，在该整数中返回的值是套接口接收队列中数据的字节数（TCPv2 第 553 页）。这个值是队列中字节数的总和，对 UDP 套接口来说包括在队列中的所有数据报。还要注意的是在源自 Berkeley 的实现中，对 UDP 套接口返回的计数包括套接口地址结构要占的空间，其中含有每个数据报的发送方 IP 地址和端口号（IPv4 为 16 字节，IPv6 为 24 字节）。

13.8 套接口和标准 I/O

目前为止的所有例子中使用的 read 及 write 函数和它们的一些变种（recv、send 等），称为 Unix I/O。这些函数使用描述字工作，通常是作为 Unix 内核中的系统调用来实现的。

进行输入输出的另一种方法是标准 I/O 函数库（standard I/O Library）。在 ANSI C 的标

准中对此进行了说明，其目的是方便移植到支持 ANSI C 的非 Unix 系统上。标准 I/O 函数库处理一些我们在使用 Unix I/O 函数时必须担心的细节问题，譬如自动缓冲输入和输出流。不幸的是，它对流的缓冲处理会产生新的问题。APUE 的第 5 章涵盖了标准 I/O 库的细节，[Plauger 1992]介绍和讨论了标准 I/O 库的一个完整的实现。

> 在标准 I/O 库中使用了流这个术语，譬如“打开一个输入流”或“刷新输出流”。不要将此与第 33 章中讨论的系统 V 的流子系统相混淆。

标准 I/O 库可以用于套接口，但有几点要考虑。

- 对任何描述字调用 fdopen 函数都可以生成一个标准 I/O 流。同样，对于一个标准 I/O流，用 fileno 可以得到对应的描述字。我们第一次遇到 fileno 是在图 6.9 中，当时要在一个标准 I/O 流上调用 select。select 只能用于描述字，所以必须获得这个标准 I/O 流的描述字。
- TCP 和 UDP 套接口是全双工的。标准 I/O 流也可以是全双工的：只要以 r+方式打开流即可，r+意味着读写。但是对这样的流不能在一个输出函数之后紧接一个输入函数，必须插入一个 fflush、fseek、fsetpos 或 rewind 调用。同样，不能在一个输入函数之后紧接一个输出函数，必须插入一个 fseek、fsetpos 或 rewind 调用，除非输入函数遇到一个文件结束符。后三个函数的问题是它们都调用 lseek，可 lseek 在套接口上会失败。
- 解决这个读写问题的最简单的方法是，对一个套接口打开两个标准 I/O 流：一个读，一个写。

例子：使用标准 I/O 的 str_echo 函数

下面我们用标准 I/O 代替 readline 和 writen 重新编写 TCP 回射服务器(图 5.3)。图 13.14是使用标准 I/O 的 str_echo 函数的新版本。(这个版本存在一个问题，我们马上会谈到。)

```
1 #include   "unp.h"
2 void
3 str_echo(int sockfd)
4 {
5     char     line[MAXLINE];
6     FILE     *fpin, *fpout;
7     fpin = Fdopen(sockfd, "r");
8     fpout = Fdopen(sockfd, "w");
9     for ( ; ; ) {
10        if (Fgets(line, MAXLINE, fpin) == NULL)
11            return;          /* connection closed by other end */
12        Fputs(line, fpout);
13    }
14 }
```

图 13.14　使用标准 I/O 重新编码的 str_echo 函数[advio/str_echo_stdio02.c]

将描述字转换成输入流和输出流

第7～13行 用fpopen创建两个标准I/O流：一个用于输入，一个用于输出。用fgets和fputs代替readline和writen。

如果服务器使用这个版本的str_echo，那么在运行客户程序时，我们会看到以下情况：

```
solaris % tcpcli02 206.62.226.33
hello, world          键入这一行，但没有回射
and hi                再键入一行，仍无回射
hello??               再键入一行，还是没有回射
^D                    键入文件结束符
hello, world          然后才把回射的三行输出
and hi
hello??
```

这里有一个缓冲问题，因为在键入文件结束符之前服务器没有任何回射。实际发生了以下事情：

- 我们键入第一行，将其发往服务器。
- 服务器用fgets读入这一行，再用fputs回射这一行。
- 但服务器的标准I/O流被标准I/O库完全缓冲。这意味着该库将回射的行拷贝到输出流的标准I/O缓冲区，而没有将缓冲区中的内容写到描述字，因为缓冲区还没有满。
- 我们键入第二行，将其发往服务器。
- 服务器用fgets读入这一行，再用fputs回射这一行。
- 服务器的标准I/O库再次将回射的行拷贝到输出流的标准I/O缓冲区，而没有将缓冲区中的内容写到描述字，因为缓冲区还没有满。
- 我们输入第三行时发生了同样的情况。
- 我们键入文件结束符，str_cli函数(图6.13)调用shutdown，向服务器发送一个FIN。
- 服务器收到这个TCP FIN，这使fgets返回一个空指针。
- str_echo函数返回到服务器的main函数(图5.12)，子进程调用exit终止。
- C的库函数exit调用标准I/O清除函数(APUE第162～164页)，由fputs装入部分内容的输出缓冲区中的数据被输出。
- 服务器的子进程终止，使它的已连接套接口被关闭，发一个FIN给客户，完成TCP的四分组终止序列。
- str_cli函数收到回射的三行并输出。
- str_cli接着在该套接口收到一个文件结束符，客户程序终止。

问题是服务器上的标准I/O库自动进行缓冲。标准I/O库执行三种缓冲。

1. 完全缓冲意味着只有在以下情况时才进行I/O：缓冲区满，进程明确地调用fflush或进程调用exit终止。标准I/O缓冲区大小通常为8192字节。
2. 行缓冲意味着在以下情况时进行I/O：遇到一个换行符，进程调用fflush或进程调用exit终止。
3. 不缓冲意味着每次调用标准I/O输出函数时都进行I/O。

大多数 Unix 中标准 I/O 库的实现遵循了以下规则：

- 标准错误输出总是不缓冲。
- 标准输入和标准输出是全缓冲的。除非它们是一个终端设备，那样的话它们是行缓冲的。
- 其他的流都是全缓冲的，除非它们是一个终端设备，那样的话它们是行缓冲的。

既然套接口不是终端设备，图 13.14 中的 str_echo 函数的问题就在于输出流(fpout)是全缓冲的。有两种解决方法：调用 setvbuf 将输出流强制成行缓冲的，或在每次 fputs 之后调用 fflush 强制输出回射行。以上的任意一种方法都可以使 str_echo 函数正常运行。

另一种解决办法是完全避免使用标准 I/O 库，而用 sfio 库。这在[Korn and Vo 1991]中进行了介绍，而且其源代码是公开的。

要注意的是一些标准 I/O 库的实现对大于 255 的描述字还有问题。这对处理多个描述字的网络服务器是一个问题。检查你的<stdio.h>头文件中 FILE 结构的定义，看看存放描述字的变量的类型。

13.9　T/TCP：事务 TCP

T/TCP 是在 TCP 的基础上作了少量的修改，以避免在最近通信过的主机之间进行三路握手。T/TCP 的细节在 TCPv3、RFC 1379[Braden 1992b]和 RFC 1644[Braden 1994]中做了说明。

T/TCP 最广为流传的实现是在 FreeBSD 中。

T/TCP 能将 SYN、FIN 和数据组合到单个分节中，前提是数据的长度小于 MSS。图 13.15 中对这种情况进行了说明。第一个分节是由一个 sendto 调用生成的客户的 SYN、FIN 和数据。它把 connect、write 和 shutdown 的功能组合起来了。服务器按照通常的步骤是调用 socket、bind、listen 和 accept，后者在客户的分节到来时返回。服务器用 send 发回应答并关闭套接口。这使服务器向客户发出 SYN、FIN 和应答。如果和图 2.5 相比较，我们可以看到不仅在网络上传输的分节要少(T/TCP 是 3 个，TCP 是 10 个，UDP 是 2 个)，而且客户初始化连接、发送请求和接收应答所花费的时间也减少了一个 RTT。

T/TCP 的好处是保持了 TCP 的所有可靠性(序列号、超时、重传，等等)，而不像 UDP 那样把可靠性推给应用程序去实现。T/TCP 还维持了 TCP 的慢启动和拥塞避免措施，这些特性在 UDP 应用程序中是没有的。

我们在这儿忽略了一些细节，它们都在 TCPv3 中介绍。例如，客户首次与服务器交互时，三路握手是需要的。不过以后只要双方高速缓存的一些信息不过时，并且没有一方的主机崩溃并重启过，那么就可避免三路握手。图中给出的三个分节来自于最少的请求-应答交换。如果或者请求或者应答在一个分节中放不下，那么就需要额外的分节。术语“事务”的含义是客户的请求与服务器的应答。常见例子有 DNS 请求与服务器的应答，以及 HTTP 请求与服务器的应答。这个术语不用来指称两阶段提交协议(two-phase commit protocol)。

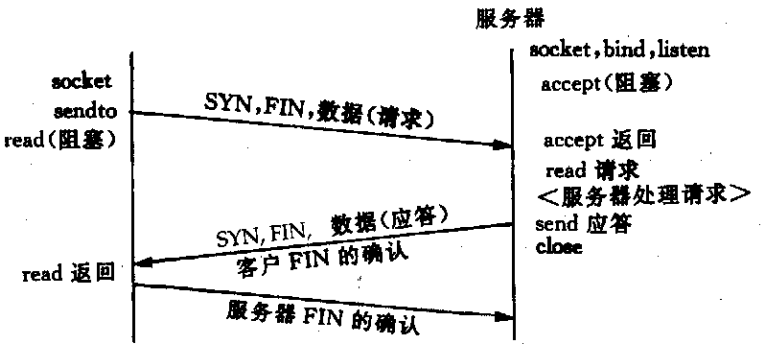


图 13.15 最小 T/TCP 事务的时间线

为处理 T/TCP,套接口 API 作了些变动。我们注意到在提供 T/TCP 的系统上,TCP 应用程序不必做任何改动,除非需要 T/TCP 提供的特性。所有已有的 TCP 应用程序继续使用已经叙述过的套接口 API 工作。

- 客户调用 sendto 以把连接的建立与数据的发送结合起来。它替代了分开的 connect 和 write 调用。服务器的协议地址现在是传递给 sendto,而不是 connect。
- 所提供的新的输出标志 MSG_EOF(图 13.6)用于指示相应套接口上不再发送数据。它允许我们把输出操作(send 或 sendto)与 shutdown 给合起来。发送含有 SYN、FIN 和数据的单个分节的方法就是给 sendto 指定这个标志以及服务器的地址。还要注意图 13.15 中服务器是使用 send 而不是 write 发送应答的,其原因是为了指定 MSG_EOF 标志以随应答一起发送 FIN(不要把这个新标志与已有的 MSG_EOR 标志混为一谈,后者指示的是面向记录协议的记录结束条件)。
- 所定义的级别为 IPPROTO_TCP 的新套接口选项 TCP_NOPUSH 可防止 TCP 就为腾空套接口发送缓冲区的目的而发送分节。当客户需要用单个 sendto 发送一个请求,并且请求可能超过 MSS 大小时,它们应该设置这个选项,因为它可以减少所发送的分节数。TCPv3 第 47～49 页详细讨论了这个新套接口选项。
- 希望跟服务器建立连接并且使用 T/TCP 发送请求的客户应该调用 socket、setsockopt(以设置 TCP_NOPUSH 选项)和 sendto(如果只发送一个请求就指定 MSG_EOF 标志)。如果 setsockopt 以错误 ENOPROTOOPT 失败返回,或者 sendto 以错误 ENOTCONN 失败返回,那么相应主机是不支持 T/TCP 的。这种情况下客户可以调用 connect 和 write,可能再跟一个 shutdown(如果只有一个请求需发送的话)。
- 服务器所需的唯一变动是,如果它想随应答一起发送 FIN 的话,它应调用 send 并指定 MSG_EOF 标志来发送应答,而不是调用 write。
- T/TCP 的编译时测试可以使用伪代码 #ifdef MSG_EOF。

TCPv3 的附录 B 有 T/TCP 客户和服务器程序的例子。

13.10　小　结

在套接口操作上设置时间限制的方法有三种：

- 使用 alarm 函数和 SIGALRM 信号
- 使用由 select 提供的时间限制
- 使用更新的 SO_RCVTIMEO 和 SO_SNDTIMEO 套接口选项

第一种方法容易使用，但涉及到信号处理，而它真如我们在 18.5 节将看到的那样会导致竞争状态。使用 select 意味着我们阻塞在这个函数及其所提供的时间限制上，而不是阻塞在对 read、write 或 connect 的调用上。第三种方法也容易使用，但不是所有实现都提供。

recvmsg 和 sendmsg 是所提供的五组 I/O 函数中最通用的。它们结合了这些能力：指定 MSG_xxx 标志（来自 recv 和 send），返回或指定对方的协议地址（来自 recvfrom 和 sendto），使用多个缓冲区（来自 readv 和 writev）；又增加了两个新的特性：给应用进程返回标志，接收或发送辅助数据。

我们在文中叙述了 10 种不同格式的辅助数据，其中 6 种是随 IPv6 新定义的。辅助数据由一个或多个辅助数据对象构成，每个对象都以一个 cmsghdr 结构打头，它指定数据的长度、协议级别及类型。5 个以 CMSG_打头的函数是用来构建和分析辅助数据的。

套接口可以跟 C 标准 I/O 函数库一块使用，但这样一来给已经由 TCP 提供的缓冲能力又增加了另外一级缓冲。实际上，缺少对由标准 I/O 库执行的缓冲机制的认识是使用这个库时最常见的问题。既然套接口不是终端设备，对这种潜在问题的常用解决办法就是把标准 I/O 流设置成不缓冲。

T/TCP 是对 TCP 的一个简单的增强，能在客户和服务器最近曾通信过的情况下避免三路握手，使服务器对客户的请求更快地做出响应。从编程的角度来看，其优点在于，客户程序以 sendto 调用代替了通常的 connect、write 和 shutdown 调用序列。

13.11　习　题

13.1　在图 13.1 中当重置信号处理程序时，如果进程还没有为 SIGALRM 建立信号处理程序，将会发生什么结果？

13.2　在图 13.1 中如果进程已设置了一个 alarm 定时器，它就会输出一个警告。修改该函数使它在 connect 后、函数返回前重置 alarm。

13.3　对图 11.7 做以下修改：在调用 read 前，以 MSG_PEEK 标志调用 recv。recv 返回时，以 FIONREAD 调用 ioctl，并输出在套接口接收缓冲区中的字节数。然后再调用 read 实际读取数据。

13.4　如果进程在 main 函数的结尾异常终止而不是以 exit 退出，那些在标准 I/O 缓冲区中还没输出的数据会发生什么结果？

13.5　按图 13.14 下面介绍的两种方法修改程序，以确认它们能解决缓冲问题。

第14章 Unix域协议

14.1 概 述

Unix域协议并不是一个实际的协议族，它只是在同一台主机上进行客户-服务器通信时，使用与在不同主机上的客户和服务器间通信时相同的API(套接口或XTI)的一种方法。当客户和服务器在同一台主机上时，Unix域协议是这套系列书的第二卷将介绍的IPC通信方式的一种替代品。TCPv3的第三部分提供了Unix域套接口在源自Berkeley的内核中的具体实现。

Unix域提供了两种类型的套接口：字节流套接口(与TCP类似)和数据报套接口(与UDP类似)。尽管还提供了原始套接口，但它的语义不曾见于任何文档，笔者也没有见到任何程序使用过它，在Posix.1g中没有对它的定义。

使用Unix域套接口有三个原因：

1. 在源自Berkeley的实现中，当通信双方位于同一台主机上时，Unix域套接口的速度通常是TCP套接口的两倍(TCPv3第223～224页)。有一个应用系统利用了这个优点：X Window系统。当X11客户启动并打开到X11服务器的连接时，客户检查DISPLAY环境变量的值，该变量指明服务器的主机名、窗口和屏幕。如果服务器与客户在同一台主机上，客户就打开一个到服务器的Unix域字节流套接口连接，否则打开一个到服务器的TCP连接。
2. Unix域套接口可以用来在同一台主机上的各进程之间传递描述字。在14.7节中对此提供了一个完整的例子。
3. Unix域套接口的较新实现中可以向服务器提供客户的凭证(用户ID和组ID)，这能提供附加的安全检查。在14.8节中对此进行了介绍。

Unix域用来标识客户和服务器的协议地址是在通常的文件系统里的路径名。我们还记得IPv4的协议地址由一个32位地址和16位的端口号组成，IPv6的协议地址由128位的地址和16位的端口号组成。这些路径名不是普通的Unix文件：除非将它们和Unix域套接口关联起来，否则程序是不能读写这些文件的。

14.2 Unix域套接口地址结构

图14.1列出了在<sys/un.h>头文件中定义的Unix域套接口地址结构。

```
struct sockaddr_un {
  uint8_t       sun_len;
  sa_family_t   sun_family;        /* AF_LOCAL */
  char          sun_path[104];     /* null-terminated pathname */
};
```

图 14.1　Unix 域套接口地址结构 sockaddr_un

BSD 的早期版本定义的 sun_path 数组的大小为 108 字节，而不是在图中的 104 字节。Posix.1g 只要求它的大小至少有 100 字节。之所以有这个限制，是因为自 4.2BSD 以来的实现要求这个结构能装入一个 128 字节的 mbuf(一种内核内存缓冲块)。

sun_path 数组中存放的路径名必须是以空字符结尾的。系统提供了 SUN_LEN 宏，它以一个指向 sockaddr_un 结构的指针为参数，返回该结构的长度，长度里包括路径名中的非空字节数。未指定地址以空字符串的路径名表示，也就是一个 sun_path[0]为 0 的地址结构。这是 Unix 域中与 IPv4 的 INADDR_ANY 常值或 IPv6 的 IN6ADDR_ANY_INIT 常值等价的一个地址。

Posix.1g 将 Unix 域协议改名为“本地 IPC”，以消除对 Unix 操作系统的依赖。原来的常值 AF_UNIX 变成 AF_LOCAL。虽然如此，我们仍使用“Unix 域”这个名词，因为它已成为一个既成事实的名字，与底层的操作系统无关。而且，即使 Posix.1g 试图使它独立于操作系统，该套接口地址结构仍保留_un 后缀。

例子：Unix 域套接口的捆绑

图 14.2 中的程序建立一个 Unix 域套接口，给它捆绑一个路径名，然后调用 getsockname 输出已绑定的路径名。

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int      sockfd;
 6     socklen_t   len;
 7     struct sockaddr_un     addr1, addr2;
 8     if (argc != 2)
 9         err_quit("usage: unixbind <pathname>");
10     sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
11     unlink(argv[1]);          /* OK if this fails */
12     bzero(&addr1, sizeof(addr1));
13     addr1.sun_family = AF_LOCAL;
14     strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) - 1);
15     Bind(sockfd, (SA *) &addr1, SUN_LEN(&addr1));
16     len = sizeof(addr2);
17     Getsockname(sockfd, (SA *) &addr2, &len);
18     printf("bound name = %s, returned len = %d\n", addr2.sun_path, len);
19     exit(0);
20 }
```

图 14.2　给 Unix 域套接口捆绑路径名[unixdomain/unixbind.c]

删除路径名

第11行 给套接口bind的路径名在命令行参数中给出。但如果该路径名在文件系统中已存在,bind将失败。因此为预防路径名已存在的情况,我们先调用unlink将其删除。如果路径名不存在,unlink会返回错误,程序将其忽略。

bind然后getsockname

第12~18行 我们用strncpy拷贝命令行参数,以避免路径名过长导致结构溢出。因为已将这个结构初始化为0,并从sun_path数组的大小中减去1,所以路径名肯定是以空字符结尾的。调用bind并用SUN_LEN宏计算该函数的长度参数。接着调用getsockname取得刚才绑定的路径名并输出结果。

如果在BSD/OS下运行这个程序,会得到以下结果:

```
bsdi % umask                                    首先输出umask的值
0002                                            shell以八进制格式输出该值
bsdi % unixbind /tmp/foo.bar
bound name = /tmp/foo.bar,returned len = 14
bsdi % unixbind /tmp/foo.bar                    再运行一次
bound name = /tmp/foo.bar,returned len = 14
bsdi % ls -l /tmp/foo.bar
srwxrwxrwx  1 rstevens wheel 0 May 20 11:02 /tmp/foo.bar
bsdi % ls -lF /tmp/foo.bar
srwxrwxrwx  1 rstevens wheel 0 May 20 11:02 /tmp/foo.bar=
```

首先输出umask的值,因为Posix.1g中说生成的路径名的权限跟该值相关。2这个值关闭了其他用户的写权限位(有时叫做全球写(world-write))。接着运行该程序,可以看到getsockname返回的长度是14:sun_len占1个字节,sun_family占1个字节,路径名占12个字节(不含结尾的空字符)。这是一个值-结果参数的例子,返回的结果与调用时的值不同。我们可以用printf的%s格式输出该路径名,因为sun_path中的路径名是以空字符结尾的。然后再运行该程序,以检验调用unlink删除了路径名。

用ls -l查看文件的权限和类型。在4.4BSD上其文件类型为套接口,输出表示为s。还可以注意到所有的9个权限位都是置位的,因为4.4BSD没有用umask来修改缺省值。最后再次用-F选项运行ls,这会使4.4BSD在路径名后附加一个等号。

> Posix.2中没有关于套接口的说明,它只规定-F选项对目录输出一个斜杠,对可执行文件输出一个星号,对FIFO输出一个竖杠。

现在在Solaris 2.5上运行同样的程序。

```
solaris % umask
02
solaris % unixbind /tmp/foo.bar
bound name = /tmp/foo.bar,returned len = 110
solaris % unixbind /tmp/foo.bar
bound name = /tmp/foo.bar,returned len = 110
solaris % ls -lF /tmp/foo.bar
p---------  1 rstevens other1        0 May 20 11:36 /tmp/foo.bar|
```

第一个差别是getsockname返回的长度是110,这是Solaris中sockaddr_un结构总的大

小。因为 sun_path 成员中的路径名是以空字符结尾的，所以这是可行的。还可以注意到缺省的文件权限是 0：所有读、写和执行权限都是关闭的。因为所有权限位都关闭，所以无法分辨是否使用了 umask 的值。最后我们注意到 ls -l 指出该路径名是一个 FIFO，这是在 SVR4 中 Unix 域套接口的表现形式，-F 选项对 FIFO 会以一个竖杠输出表示。

14.3 socketpair 函数

socketpair 函数建立一对相互连接的套接口。这个函数只对 Unix 域套接口适用。

```
#include <sys/socket.h>
int socketpair(int family,int type,int protocol,int sockfd[2]);
                                        返回：成功返回 0，出错返回 -1
```

family 必须为 AF_LOCAL，protocol 必须为 0，type 可以是 SOCK_STREAM 或 SOCK_DGRAM。新创建的两个套接口描述字作为 sockfd[0]和 sockfd[1]返回。

> 这个函数和 Unix 的 pipe 函数类似：都返回两个描述字，而且两个描述字互相连接。实际上，源自 Berkeley 的实现中，pipe 使用了与 sokcetpair[TCPv3 第 253～254 页]同样的内部操作。

创建的两个套接口是没有名字的；即没有涉及隐式 bind。

以 SOCKET_STRAEM 作为 type 调用 socketpair 所得到的结果称作流管道(stream pipe)。这和一般的 Unix 管道(由 pipe 函数生成)类似，但流管道是全双工的，即两个描述字都是可读写的。在图 14.7 中展示了用 socketpair 建立的流管道。

> Posix.1 不要求全双工的管道。在 SVR4 中 pipe 返回两个全双工的描述字，但源自 Berkeley 的内核一般返回两个半双工的描述字(TCPv3 的图17.31)。

14.4 套接口函数

当用于 Unix 域套接口时，套接口函数有一些差别和限制。下面列出 Posix.1g 对此的一些要求，需注意的是并不是所有的实现中都做到了这一级。

1. bind 建立的路径名的缺省访问权限应为 0777(用户、用户组和其他用户都能读、写和执行)，并被当前的 umask 值修改。
2. 与 Unix 域套接口相关联的路径名应为一个绝对路径名，而不是相对路径名。避免使用后者的原因是它依赖于调用者的当前工作目录。这就是说，如果服务器捆绑一个相对路径名，客户必须在与服务器相同的目录下(或必须知道这个目录)调用 connect 或 sendto 才能成功。

   > Posix.1g 中说给 Unix 域套接口捆绑相对路径名将产生不可预计的结果。

3. connect 使用的路径名必须是一个绑定在某个已打开的 Unix 域套接口上的路径名，

而且套接口的类型(字节流或数据报)也必须一致。下列情况将会出错:(a)该路径名存在但不是一个套接口,(b)路径名存在且是一个套接口,但没有与该路径名相关联的打开的描述字,(c)路径名存在且是一个打开的套接口,但类型不符(也就是说,Unix 域字节流套接口不能连到与 Unix 域数据报套接口相关联的路径名,反之亦然)。

4. 用 connect 连接 Unix 域套接口时的权限检查和用 open 以只写方式访问路径名时完全相同。
5. Unix 域字节流套接口和 TCP 套接口类似:它们都为进程提供一个没有记录边界的字节流接口。
6. 如果 Unix 域字节流套接口的 connect 调用发现监听套接口的队列已满(4.5 节),会立刻返回一个 ECONNREFUSED 错误。这和 TCP 有所不同:如果监听套接口的队列已满,它将忽略到来的 SYN,TCP 连接的发起方会接着发送几次 SYN 重试。
7. Unix 域数据报套接口和 UDP 套接口类似:它们都提供一个保留记录边界的不可靠的数据报服务。
8. 与 UDP 套接口不同的是,在未绑定的 Unix 域套接口上发送数据报不会给它捆绑一个路径名。(回想在未绑定的 UDP 套接口上发送 UDP 数据报会为该套接口捆绑一个临时端口。)这意味着,数据报的发送者除非绑定一个路径名,否则接收者无法发回应答数据报。同样,与 TCP 和 UDP 不同的是,给 Unix 域数据报套接口调用 connect 不会捆绑一个路径名。

14.5 Unix 域字节流客户-服务器程序

下面重写第 5 章中的 TCP 回射客户-服务器程序,以使用 Unix 域套接口。图 14.3 给出从图 5.12 修改而来的、用 Unix 域字节流协议代替 TCP 的服务器程序。

第 8 行 两个套接口地址结构的数据类型现在是 sockaddr_un。

第 10 行 socket 的第一个参数是 AF_LOCAL,以建立 Unix 域字节流套接口。

第 11～15 行 unp.h 中定义的 UNIXSTR_PATH 常值为/tmp/unix.str。为预防路径名因前次运行这个程序而已存在的情况,我们先 unlink 这个路径名,然后在 bind 前初始化套接口地址结构。unlink 出错没有关系。

注意 bind 调用与图 14.2 中的调用不同。这里将 sockaddr_un 结构的总的大小作为套接口地址结构的大小(第三个参数),而不只是路径名成员占用的字节数。因为路径名是以空字符结尾的,所以两个长度都是正确的。

函数的其余部分和图 5.12 中的相同。使用了同样的 str_echo 函数(图 5.3)。

图 14.4 是使用 Unix 域字节流协议的回射客户程序,它由图 5.4 修改而来。

第 6 行 含有服务器地址的套接口地址结构现在是一个 sockaddr_un 结构。

第 7 行 socket 的第一个参数是 AF_LOCAL。

第 8～10 行 填写套接口地址结构的代码与服务器的相同:把结构初始化成 0,family 置为 AF_LOCAL,拷贝路径名到 sun_path 成员中。

第 12 行 str_cli 函数和前面的一样(图 6.13 是我们开发的最近一个版本)。

```
1 #include    "unp.h"
```

```
2 int
3 main(int argc, char * * argv)
4 {
5     int          listenfd, connfd;
6     pid_t        childpid;
7     socklen_t    clilen;
8     struct sockaddr_un   cliaddr, servaddr;
9     void         sig_chld(int);
10    listenfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
11    unlink(UNIXSTR_PATH);
12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sun_family = AF_LOCAL;
14    strcpy(servaddr.sun_path, UNIXSTR_PATH);
15    Bind(listenfd, (SA * ) &servaddr, sizeof(servaddr));
16    Listen(listenfd, LISTENQ);
17    Signal(SIGCHLD, sig_chld);
18    for ( ; ; ) {
19        clilen = sizeof(cliaddr);
20        if ( (connfd = accept(listenfd, (SA * ) &cliaddr, &clilen)) < 0) {
21            if (errno == EINTR)
22                continue;          /* back to for() */
23            else
24                err_sys("accept error");
25        }
26        if ( (childpid = Fork()) == 0) {     /* child process */
27            Close(listenfd);       /* close listening socket */
28            str_echo(connfd);      /* process the request */
29            exit(0);
30        }
31        Close(connfd);             /* parent closes connected socket */
32    }
33 }
```

图 14.3　使用 Unix 域字节流协议的回射服务器程序[unixdomain/unixstrserv01.c]

14.6　Unix 域数据报客户-服务器程序

下面重写 8.3 和 8.4 节中的 UDP 客户-服务器程序，以使用 Unix 域数据报套接口。图 14.5 给出了从图 8.3 修改而来的服务器程序。

```
1 #include    "unp.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int        sockfd;
6     struct sockaddr_un     servaddr;
7     sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
8     bzero(&servaddr, sizeof(servaddr));
```

```
 9      servaddr.sun_family = AF_LOCAL;
10      strcpy(servaddr.sun_path, UNIXSTR_PATH);

11      Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
12      str_cli(stdin, sockfd);         /* do it all */
13      exit(0);
14  }
```

图 14.4 使用 Unix 域字节流协议的回射客户程序[unixdomain/unixstrcli01.c]

```
 1  #include    "unp.h"

 2  int
 3  main(int argc, char **argv)
 4  {
 5      int     sockfd;
 6      struct sockaddr_un  servaddr, cliaddr;

 7      sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

 8      unlink(UNIXDG_PATH);
 9      bzero(&servaddr, sizeof(servaddr));
10      servaddr.sun_family = AF_LOCAL;
11      strcpy(servaddr.sun_path, UNIXDG_PATH);

12      Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

13      dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14  }
```

图 14.5 使用 Unix 域数据报协议的回射服务器程序[unixdomain/unixdgserv01.c]

第 6 行 两个套接口地址结构的数据类型现在是 sockaddr_un。

第 7 行 socket 的第一个参数是 AF_LOCAL，以建立 Unix 域数据报套接口。

第 8～12 行 unp.h 中定义的 UNIXDG_PATH 常值为/tmp/unix.dg。为预防路径名因前次运行服务器而已存在的情况，我们先 unlink 这个路径名，然后在调用 bind 前初始化套接口地址结构。unlink 出错没有关系。

第 13 行 使用同样的 dg_echo 函数(图 8.4)。

图 14.6 是使用 Unix 域数据报协议的回射客户程序。它是对图 8.7 的一个修改版本。

```
 1  #include    "unp.h"

 2  int
 3  main(int argc, char **argv)
 4  {
 5      int     sockfd;
 6      struct sockaddr_un  cliaddr, servaddr;

 7      sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

 8      bzero(&cliaddr, sizeof(cliaddr));       /* bind an address for us */
 9      cliaddr.sun_family = AF_LOCAL;
10      strcpy(cliaddr.sun_path, tmpnam(NULL));

11      Bind(sockfd, (SA *) &cliaddr, sizeof(cliaddr));

12      bzero(&servaddr, sizeof(servaddr));  /* fill in server's address */
13      servaddr.sun_family = AF_LOCAL;
14      strcpy(servaddr.sun_path, UNIXDG_PATH);
```

```
15      dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));
16      exit(0);
17  }
```

图 14.6　使用 Unix 域数据报协议的回射客户程序[unixdomain/unixdgcli01.c]

第 6 行　含有服务器地址的套接口地址结构现在是一个 sockaddr_un 结构。我们还分配了一个这样的结构以存放客户的地址。

第 7 行　socket 的第一个参数是 AF_LOCAL。

第 8～11 行　与 UDP 客户不同的是，当使用 Unix 域数据报协议时，我们必须显式地给套接口 bind 一个路径名，这样服务器才能有路径名发回结果。我们调用 tmpnam 分配一个唯一的路径名，然后用 bind 来将它捆绑到套接口上。回想 14.4 节，在一个未绑定的 Unix 域数据报套接口上发出数据报不会自动给套接口捆绑路径名。因此如果省掉这一步，服务器在 dg_echo 中的 recvfrom 调用会返回一个空路径名，这将使服务器在调用 sendto 时发生错误。

第 12～14 行　向套接口地址结构中填入服务器的众所周知路径名的代码和以前的服务器代码相同。

第 15 行　dg_cli 函数和以前的一样(图 8.8)。

14.7　描述字传递

当考虑从一个进程向另一个进程传递打开的描述字时，我们通常会想到：

1. 在 fork 调用后，子进程共享父进程的所有打开的描述字。
2. 在调用 exec 时所有描述字仍保持打开。

第一个例子中进程打开一个描述字，调用 fork，然后父进程关闭描述字，让子进程处理这个描述字。这样将一个打开的描述字从父进程传递到子进程。但我们也想让子进程打开一个描述字并将其传给父进程。

当前的 Unix 系统提供了一种方法，可以从一个进程向其他任何进程传递打开的描述字。也就是说，进程之间不需要有什么关系，譬如父子进程。这种技术要求先在两个进程之间建立一个 Unix 域套接口，然后用 sendmsg 从这个套接口上发一个特殊的消息。这个消息由内核做特殊处理，以将打开的描述字从发送方传递到接收方。

TCPv3 的第 18 章中介绍了 4.4BSD 在 Unix 域套接口上传递打开的描述字的过程细节。

SVR4 在内核中使用了一种不同的技术来传递打开的描述字，即 I_SENDFD 和 I_RECVFD 这两个 ioctl 命令，APUE 的 15.5.1 节中对此作了介绍。但进程仍然可以使用 Unix 域套接口来访问这个内核功能。在本书中我们介绍怎样用 Unix 域套接口来传递描述字，因为这是可移植性最好的编程技术：它在源自 Berkeley 的内核或 SVR4 上都能工作，而 I_SENDFD 和 I_RECVFD 这两个 ioctl 命令只能在 SVR4 上使用。

4.4BSD 的技术允许一次 sendmsg 传递多个描述字，而 SVR4 的技术一次只能

传递一个描述字。我们所有的例子都是一次传递一个描述字。

在两个进程之间传递描述字的步骤如下：

1. 创建一个字节流的或数据报的 Unix 域套接口。
   如果目标是 fork 一个子进程，让子进程打开描述字并将它传回给父进程，那么父进程可以用 socketpair 创建一个流管道，用它来传递描述字。
   如果进程之间没有亲缘关系，那么服务器必须创建一个 Unix 域字节流套接口，bind 一个路径名，让客户 connect 到这个套接口。然后客户可以向服务器发送一个请求以打开某个描述字，服务器将描述字通过 Unix 域套接口传回。在客户和服务器之间也可以使用 Unix 域数据报套接口，但这样做没什么好处，而且数据报存在丢失的可能性。在本节的例子中客户和服务器之间将使用字节流套接口。
2. 进程可以用任何返回描述字的 Unix 函数打开一个描述字：譬如 open、pipe、mkfifo、socket 或 accept。可以在进程之间传递任何类型的描述字，这是为什么我们将这种技术称为“传递描述字”而不是“传递文件描述字”的原因。
3. 发送进程建立一个 msghdr 结构(13.5 节)，其中包含要传递的描述字。Posix.1g 中说明该描述字作为辅助数据(msghdr 结构的 msg_control 成员，见 13.6 节)发送，但老的实现使用 msg_accright 成员。发送进程调用 sendmsg 通过第一步得到的 Unix 域套接口发出描述字。这时我们说这个描述字是“在飞行中(in flight)”的。即使在发送进程调用 sendmsg 之后，但在接收进程调用 recvmsg(下一步要做的)之前将描述字关闭，它仍会为接收进程保持打开状态。描述字的发送导致它的访问计数加 1。
4. 接收进程调用 recvmsg 在 Unix 域套接口上接收描述字。通常接收进程收到的描述字的编号和发送进程中的描述字的编号不同，但这没有问题。传递描述字不是传递描述字的编号，而是在接收进程中创建一个新的描述字，指向内核的文件表中与发送进程发送的描述字相同的项。

客户和服务器之间必须有某种应用协议，使接收方知道何时接收描述字。如果接收方调用 recvmsg 但没有分配接收描述字的空间，而且有一个描述字已被传递并正待读出，这个已传递的描述字就会关闭(TCPv2 第 518 页)。在用 recvmsg 接收描述字时还要避免使用 MSG_PEEK 标志，否则后果不可预料。

描述字传递的例子

下面提供一个描述字传递的例子。我们将编写一个叫 mycat 的程序，它从命令行得到一个路径名，打开这个文件，并将文件的内容拷贝到标准输出。但是该程序不用通常的 Unix open 函数，而是调用自己的名为 my_open 的函数。这个函数创建一个流管道，调用 fork 和 exec 执行另一个程序，这个程序才打开想要的文件。接着这个程序将打开的描述字通过流管道传回给父进程。

图 14.7 展示了第一步：调用 socketpair 创建流管道后的 mycat 进程。我们用[0]和[1]来指明 socketpair 返回的两个描述字。

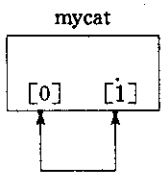


图 14.7　用 socketpair 创建流管道后的 mycat 进程

然后 mycat 进程调用 fork，子进程调用 exec 执行 openfile 程序。父进程关闭[1]描述字，子进程关闭[0]描述字。(流管道的两端没有什么不同。也可以让子进程关闭[1]，父进程关闭[0]。)这样会得到图 14.8 所示的结果。

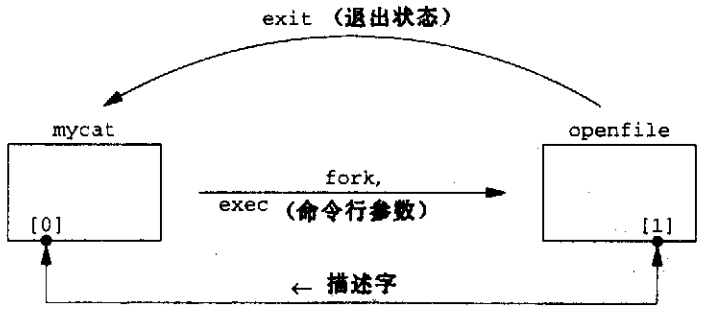


图 14.8　调用 openfile 程序后的 mycat 进程

父进程必须给 openfile 程序传递三个信息：(1)要打开的文件的路径名，(2)打开方式(只读、读写或只写)，以及(3)流管道的本进程端(就是图中的[1])对应的描述字的编号。我们选择以命令行方式在 exec 调用时传递这三个信息。另一种方法是通过流管道将这些信息作为数据发送过去。openfile 程序通过流管道发回打开的描述字并终止。程序的退出状态告诉父进程文件能否打开，如果不能打开则发生了什么类型的错误。

通过执行另一个程序来打开文件的好处在于，另一个程序可以是一个 setuid 到 root 的程序，能打开原来没有权限打开的文件。这个程序能够扩展通常的 Unix 权限(用户、用户组和其他用户)到它想要的任何形式的访问检查。

下面以 mycat 程序开始讨论，见图 14.9。

```
1 #include    "unp.h"
2 int     my_open(const char *, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     fd, n;
7     char    buff[BUFFSIZE];
8     if (argc != 2)
9         err_quit("usage: mycat <pathname>");
10    if ( (fd = my_open(argv[1], O_RDONLY)) < 0)
11        err_sys("cannot open %s", argv[1]);
```

```
12      while ( (n = Read(fd, buff, BUFFSIZE)) > 0)
13          Write(STDOUT_FILENO, buff, n);
14      exit(0);
15  }
```

图 14.9 mycat 程序:把一个文件拷贝到标准输出[unixdomain/mycat.c]

如果把 my_open 换成 open,这个简单的程序只是将一个文件拷贝到标准输出。

图 14.10 中的 my_open 函数和通常的 Unix open 函数看起来一样。它取两个参数:一个路径名和一个打开方式(譬如 O_RDONLY 表示只读),打开该文件并返回描述字。

```
1  #include    "unp.h"
2  int
3  my_open(const char *pathname, int mode)
4  {
5      int         fd, sockfd[2], status;
6      pid_t       childpid;
7      char        c, argsockfd[10], argmode[10];
8      Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);
9      if ( (childpid = Fork()) == 0) {            /* child process */
10         Close(sockfd[0]);
11         snprintf(argsockfd, sizeof(argsockfd), "%d", sockfd[1]);
12         snprintf(argmode, sizeof(argmode), "%d", mode);
13         execl("./openfile", "openfile", argsockfd, pathname, argmode,
14                 (char *) NULL);
15         err_sys("execl error");
16     }
17     /* parent process - wait for the child to terminate */
18     Close(sockfd[1]);          /* close the end we don't use */
19     Waitpid(childpid, &status, 0);
20     if (WIFEXITED(status) == 0)
21         err_quit("child did not terminate");
22     if ( (status = WEXITSTATUS(status)) == 0)
23         Read_fd(sockfd[0], &c, 1, &fd);
24     else {
25         errno = status;        /* set errno value from child's status */
26         fd = -1;
27     }
28     Close(sockfd[0]);
29     return(fd);
30 }
```

图 14.10 my_open 函数:打开一个文件并返回其描述字[unixdomain/myopen.c]

创建流管道

第 8 行 socketpair 创建一个流管道,返回两个描述字:sockfd[0]和 sockfd[1]。这是图 14.7 中展示的情形。

fork 并 exec

第 9~16 行 调用 fork,然后子进程关闭流管道的一端。流管道另一端的描述字编号填

入 argsockfd 数组，打开方式填入 argmode 数组。这里调用 snprintf 是因为 exec 的参数必须是字符串。openfile 程序随后执行。execl 函数不应该返回，除非出错。如果成功，openfile 程序的 main 函数开始执行。

父进程等待子进程

第 17～22 行 父进程关闭流管道的另一端并调用 waitpid 等待子进程的结束。子进程的终止状态在 status 变量中返回。先要检查一下它是否正常终止(即它不是被信号终止)。接着 WEXITSTATUS 宏把终止状态转换成退出状态，退出状态的取值在 0～255 之间。我们马上就会看到，如果 openfile 程序在打开文件时出错，它将以对应的 errno 值作为退出状态来终止。

接收描述字

第 23 行 下面给出的 read_fd 函数在流管道上接收描述字。除了描述字之外，它还读 1 字节数据，但不对数据做任何处理。

> 在流管道上发送和接收描述字时，我们总是发送至少 1 字节的数据，即使接收方不对数据进行任何处理。否则接收方就分不清 read_fd 的返回值为 0 时，意味着“没有数据(但可能有一个描述字)”还是“文件结束”符。

图 14.11 给出了 read_fd 函数，它调用 recvmsg 在 Unix 域套接口上接收数据和描述字。这个函数的前三个参数和 read 函数一样，第四个参数是一个指向整数的指针，以返回接收到的描述字。

第 9～26 行 这个函数必须处理两种版本的 recvmsg：使用 msg_control 成员的和使用 msg_accrights 成员的。如果支持 msg_control 版本，我们的 config.h 头文件(图 D.2)中就会定义常值 HAVE_MSGHDR_MSG_CONTROL。

使 msg_control 适当地对齐

第 10～13 行 msg_control 缓冲区必须与 cmsghdr 结构适当地对齐。简单地分配一个字符数组是不合适的。这里声明了一个 cmsghdr 结构和一个字符数组的联合，以确保数组能适当地对齐。另一种方法是调用 malloc，但需要在函数返回前释放内存。

第 27～45 行 调用 recvmsg。如果返回辅助数据，则其格式与图 13.13 中所示的一样。检查长度、级别和类型是否正确，然后取出新建的描述字，并通过调用者的 recvfd 指针返回。CMSG_DATA 返回一个 unsigned char 指针，指向辅助数据对象的 cmsg_data 成员。我们将它强制成一个 int 指针并取出它指向的整型描述字。

```
1 #include    "unp.h"
2 ssize_t
3 read_fd(int fd, void *ptr, size_t nbytes, int *recvfd)
4 {
5     struct msghdr msg;
6     struct iovec iov[1];
7     ssize_t n;
8     int         newfd;
9 #ifdef  HAVE_MSGHDR_MSG_CONTROL
10    union {
```

```
11          struct cmsghdr   cm;
12          char     control[CMSG_SPACE(sizeof(int))];
13      } control_un;
14      struct cmsghdr   *cmptr;

15      msg.msg_control = control_un.control;
16      msg.msg_controllen = sizeof(control_un.control);
17  #else
18      msg.msg_accrights = (caddr_t) &newfd;
19      msg.msg_accrightslen = sizeof(int);
20  #endif

21      msg.msg_name = NULL;
22      msg.msg_namelen = 0;

23      iov[0].iov_base = ptr;
24      iov[0].iov_len = nbytes;
25      msg.msg_iov = iov;
26      msg.msg_iovlen = 1;

27      if ( (n = recvmsg(fd, &msg, 0)) <= 0)
28          return(n);

29  #ifdef      HAVE_MSGHDR_MSG_CONTROL
30      if ( (cmptr = CMSG_FIRSTHDR(&msg)) != NULL &&
31          cmptr->cmsg_len == CMSG_LEN(sizeof(int))) {
32          if (cmptr->cmsg_level != SOL_SOCKET)
33              err_quit("control level != SOL_SOCKET");
34          if (cmptr->cmsg_type != SCM_RIGHTS)
35              err_quit("control type != SCM_RIGHTS");
36          *recvfd = *((int *) CMSG_DATA(cmptr));
37      } else
38          *recvfd = -1;       /* descriptor was not passed */
39  #else

40      if (msg.msg_accrightslen == sizeof(int))
41          *recvfd = newfd;
42      else
43          *recvfd = -1;       /* descriptor was not passed */
44  #endif

45      return(n);
46  }
```

图 14.11 read_fd 函数:接收数据和描述字[lib/read_fd.c]

如果支持老的 msg_accrights 成员,长度应为一个整数的大小,新建的描述字通过调用者的 recvfd 指针返回。

图 14.12 给出了 openfile 程序。它从命令行得到调用 open 函数必需的三个参数。

```
1 #include    "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     int         fd;
6     ssize_t     n;
```

```
 7    if (argc != 4)
 8        err_quit("openfile <sockfd#> <filename> <mode>");
 9    if ((fd = open(argv[2], atoi(argv[3]))) < 0)
10        exit((errno > 0) ? errno : 255);
11    if ((n = write_fd(atoi(argv[1]), "", 1, fd)) < 0)
12        exit((errno > 0) ? errno : 255);
13    exit(0);
14 }
```

图 14.12　openfile 函数：打开一个文件并传回描述字[unixdomain/openfile.c]

命令行参数

第 7～12 行　因为三个参数中的两个被 my_open 转换成字符串，所以用 atoi 将它们转回整数。

打开文件

第 9～10 行　调用 open 打开文件。如果出错，open 对应的 errno 的值作为进程的退出状态返回。

传回描述字

第 11～12 行　下面给出的 write_fd 将描述字传回。这个进程随后终止，但本章的前面曾说过发送进程在传递完描述字后将其关闭（调用 exit 时将发生这种情况）不会有任何问题，因为内核知道描述字在飞行中，会为接收进程保持打开状态。

> 退出状态码必须在 0 和 255 之间。最大的 errno 值大概是 150。另一种不要求 errno 的值小于 256 的方法是，在调用 sendmsg 时将错误代码作为普通数据传回。

图 14.13 给出了最后一个函数即 write_fd，它调用 sendmsg 通过 Unix 域套接口发送描述字（和可选的数据，这里没有用上）。

```
 1 #include    "unp.h"
 2 ssize_t
 3 write_fd(int fd, void *ptr, size_t nbytes, int sendfd)
 4 {
 5     struct msghdr    msg;
 6     struct iovec     iov[1];
 7 #ifdef    HAVE_MSGHDR_MSG_CONTROL
 8     union {
 9       struct cmsghdr   cm;
10       char             control[CMSG_SPACE(sizeof(int))];
11     } control_un;
12     struct cmsghdr    *cmptr;
13     msg.msg_control = control_un.control;
14     msg.msg_controllen = sizeof(control_un.control);
15     cmptr = CMSG_FIRSTHDR(&msg);
16     cmptr->cmsg_len = CMSG_LEN(sizeof(int));
17     cmptr->cmsg_level = SOL_SOCKET;
```

```
18      cmptr->cmsg_type = SCM_RIGHTS;
19      *((int *) CMSG_DATA(cmptr)) = sendfd;
20 #else
21      msg.msg_accrights = (caddr_t) &sendfd;
22      msg.msg_accrightslen = sizeof(int);
23 #endif

24      msg.msg_name = NULL;
25      msg.msg_namelen = 0;

26      iov[0].iov_base = ptr;
27      iov[0].iov_len = nbytes;
28      msg.msg_iov = iov;
29      msg.msg_iovlen = 1;

30      return(sendmsg(fd, &msg, 0));
31 }
```

图14.13 write_fd函数:调用sendmsg传递描述字[lib/write_fd.c]

read_fd函数必须既能处理辅助数据又能处理老的访问权限。任何一种情况都要初始化msghdr结构,然后调用sendmsg。

在25.7节中举了一个无亲缘关系进程之间描述字传递的例子,27.9节中举了一个有亲缘关系进程之间传递描述字的例子。这些例子中将使用以上介绍的read_fd和write_fd函数。

14.8 接收发送者的凭证

图13.13列出了另一种能在Unix域套接口上作为辅助数据传送的数据:fcred结构的用户凭证,该结构在<sys/ucred.h>头文件中定义。

```
struct fcred {
  uid_t     fc_ruid;                  /* real user ID */
  gid_t     fc_rgid;                  /* real group ID */
  char      fc_login[MAXLOGNAME];     /* setlogin() name */
  uid_t     fc_uid;                   /* effective user ID */
  short     fc_ngroups;               /* number of groups */
  gid_t     fc_groups[NGROUPS];       /* supplementary group IDs */
};
#define     fc_gid fc_groups[0]       /* effective group ID */
```

通常MAXLOGNAME为16,NGROUPS也是16。fc_ngroups总是至少为1,fc_groups数组的第一个元素为有效的组ID。

这个结构实际的定义是一个在fcred结构内的ucred结构,只是用#define使它看上去像一个单独的结构。上面列出的是对该结构的"逻辑"定义。

这个功能在BSD/OS2.1中是新增的。虽然这种功能还没普及,但是因为它是对Unix域协议的一个十分重要和简单的补充,所以在这里对它进行介绍。当客户和服务器进行通信时,服务器通常需要一种方法确认客户,以验证客户有权限请求该服务。

只要遵循以下条件，该信息在 Unix 域套接口上总是可用的。

- 凭证是作为辅助数据在 Unix 域套接口上发送的，但接收方必须打开 LOCAL_CREDS 套接口选项。这个选项的级别(7.2 节)为 0。
- 在数据报套接口上，每个数据报都带有凭证。在字节流套接口上凭证只发送一次，是在第一次发送数据时发出的。
- 凭证不能和描述字一起发送。也就是说，在单个消息中只能发送这两种辅助数据中的一种。

  这是 BSD/OS 实现中的一个限制。按常理，既然每个辅助数据对象都有自己的类型和长度，在一个 sendmsg 调用中应能传送多种类型的辅助数据(图 13.12)。

- 用户是不能伪造凭证的。也就是说，在 Unix 域套接口上发送辅助数据时，内核会加以检验，确保不是级别为 SOL_SOCKET、类型为 SCM_CREDS 的辅助数据。如果发送方试图伪造凭证，辅助数据会被内核丢弃。

前一节中有一点没有提到，在 SVR4 上收到一个描述字时(用 ioctl 的 I_RECVFD 指令)，内核也将发送方的凭证传送给接收进程：一个含有新建的描述字、有效用户 ID 和有效用户组 ID 的 strrecvfd 结构。这是每次传递描述字时都会传送的凭证的形式。另外，当用 connld 流模块建立客户-服务器连接时(与在 Unix 域套接口上用 accept 建立一个新的套接口类似)，新的描述字和一个包含客户凭证的 strrecvfd 结构一起传回。在 SVR4 上不能用 Unix 域套接口访问这些凭证。(APUE 的 15.5.1 节详细介绍了 SVR4 的 connld 流模块的使用。)如果一个源自 Berkeley 的实现不支持本节介绍的新的凭证传送手段，就不能保证 Unix 域服务器能得到客户的凭证。APUE 的 15.5.2 节中详细介绍了获得这些信息的一种迂回方法，但用户凭证应总是由内核来提供。

例子

作为传送凭证的一个例子，我们修改 Unix 域字节流服务器程序，使它向客户请求凭证。图 14.14 给出了一个名为 read_cred 的新函数，它和 read 类似，但还返回一个含有发送方凭证的 fcred 结构。

第 4～5 行　前三个参数和 read 相同，第四个参数是一个指向 fcred 结构的指针。返回的辅助数据的格式如图 13.13 所示。

第 26～36 行　如果返回了凭证，就检验辅助数据的长度、级别和类型，并将结果拷回给调用者。如果没有返回凭证，就将结构置为 0。因为用户组的数目(fc_ngroups)总是大于等于 1 的，它的值为 0 对调用者表示内核没有返回凭证。

图 14.3 中回射服务器的 main 函数没有变化。图 14.15 给出了从图 5.3 修改而来的 str_echo 函数的新版本。这个函数由子进程在父进程接收了一个新的客户连接并 fork 之后调用。

第 12 行　对已连接套接口设置 LOCAL_CREDS 套接口选项。

第 13～14 行　第一次调用 read_cred 函数。将长度置为 0，表示不接收一般数据，只想接收辅助数据。

第17～27行 如果返回了凭证，将其输出。

第28～32行 循环剩下的部分没变。这段代码从客户读入数据，再将数据返回给客户。从图14.4而来的客户程序没有改变。

```
1 #include    "unp.h"
2 #include    <sys/param.h>
3 #include    <sys/ucred.h>
4 ssize_t
5 read_cred(int fd, void *ptr, size_t nbytes, struct fcred *fcredptr)
6 {
7     struct msghdr msg;
8     struct iovec iov[1];
9     ssize_t n;
10    union {
11      struct cmsghdr  cm;
12      char    control[CMSG_SPACE(sizeof(struct fcred))];
13    } control_un;
14    struct cmsghdr    *cmptr;
15    msg.msg_control = control_un.control;
16    msg.msg_controllen = sizeof(control_un.control);
17    msg.msg_name = NULL;
18    msg.msg_namelen = 0;
19    iov[0].iov_base = ptr;
20    iov[0].iov_len = nbytes;
21    msg.msg_iov = iov;
22    msg.msg_iovlen = 1;
23    if ( (n = recvmsg(fd, &msg, 0)) < 0)
24        return(n);
25    if (fcredptr) {
26        if (msg.msg_controllen > sizeof(struct cmsghdr)) {
27            cmptr = CMSG_FIRSTHDR(&msg);
28            if (cmptr->cmsg_len != CMSG_LEN(sizeof(struct fcred)))
29                err_quit("control length = %d", cmptr->cmsg_len);
30            if (cmptr->cmsg_level != SOL_SOCKET)
31                err_quit("control level != SOL_SOCKET");
32            if (cmptr->cmsg_type != SCM_CREDS)
33                err_quit("control type != SCM_CREDS");
34            memcpy(fcredptr, CMSG_DATA(cmptr), sizeof(struct fcred));
35        } else
36            bzero(fcredptr, sizeof(struct fcred)); /* none returned */
37    }
38    return(n);
39 }
```

图14.14 read_cred函数：读取并返回发送方的凭证[unixdomain/readcred.c]

```
1 #include    "unp.h"
2 #include    <sys/param.h>
3 #include    <sys/ucred.h>
```

```
4 ssize_t     read_cred(int, void *, size_t, struct fcred *);
5 void
6 str_echo(int sockfd)
7 {
8     ssize_t n;
9     const int on = 1;
10    char     line[MAXLINE];
11    struct fcred cred;
12    Setsockopt(sockfd, 0, LOCAL_CREDS, &on, sizeof(on));
13    if ( (n = read_cred(sockfd, NULL, 0, &cred)) < 0)
14        err_sys("read_cred error");
15    if (cred.fc_ngroups == 0)
16        printf("(no credentials returned)\n");
17    else {
18        printf("real user ID = %d\n", cred.fc_ruid);
19        printf("real group ID = %d\n", cred.fc_rgid);
20        printf("login name = %-*s\n", MAXLOGNAME, cred.fc_login);
21        printf("effective user ID = %d\n", cred.fc_uid);
22        printf("effective group ID = %d\n", cred.fc_gid);
23        printf("%d supplementary groups:", cred.fc_ngroups - 1);
24        for (n = 1; n < cred.fc_ngroups; n++)      /* [0] is the egid */
25            printf(" %d", cred.fc_groups[n]);
26        printf("\n");
27    }
28    for ( ; ; ) {
29        if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
30            return;              /* connection closed by other end */
31        Writen(sockfd, line, n);
32    }
33 }
```

图 14.15　请求客户凭证的 str_echo 函数[unixdomain/strecho.c]

如果一个窗口运行服务器，另一个窗口运行客户，下面就是运行一次客户程序后服务器的输出。

```
bsdi % unixstrserv02
real user ID = 482
real group ID = 52
login name = rstevens
effective user ID = 482
effective group ID = 52
7 supplementary groups: 20 0 1 2 3 5 7
```

因为前面曾经提到过，在字节流套接口上凭证是在第一次发送数据时(不是在连接建立时)由内核一块发送的，所以客户发出第一行数据后服务器才输出信息。这和前面提到的 SVR4 中的技术有所不同，SVR4 是在返回新建的描述字时一块发送凭证的(对于 Unix 域套接口来说，这相当于 accept 的返回)。

14.9 小 结

客户和服务器在同一台主机上时，Unix 域套接口是 IPC 的一种替代方法。和 IPC 相比使用 Unix 域套接口的好处在于，对网络客户-服务器的 API 几乎是相同的。当客户和服务器在同一台主机上时，Unix 域套接口和 TCP 相比，其性能在多数实现中要更高。

为了使用 Unix 域协议，我们对自己的 TCP 和 UDP 回射客户和服务器程序做了修改，唯一主要的差别是 UDP 客户的套接口必须 bind 一个路径名，UDP 服务器才有地方发送应答。14.5 和 14.6 节中的代码是直接操作 Unix 域套接口地址结构的，更好的方法是使用 11 章里的 tcp_XXX 和 udp_XXX 函数，因为 getaddrinfo 的实现支持 Unix 域套接口。

同一台主机上的客户和服务器之间传递描述字是一种强大的技术，它是通过 Unix 域套接口进行的。在 14.7 节中列举了一个从子进程向父进程传回描述字的例子。25.7 节中将列举没有亲缘关系的客户和服务器进程传递描述字的例子，27.9 节中将列举另一个从父进程向子进程传递描述字的例子。

14.10 习 题

14.1 如果一个 Unix 域服务器在调用 bind 之后调用 unlink，会发生什么结果？

14.2 如果 Unix 域服务器在终止时不 unlink 它的众所周知路径名，并且有一个客户在服务器终止后试图 connect 该服务器，将会发生什么结果？

14.3 编译图 11.12 和 11.15 中独立于协议的 UDP 时间/日期客户和服务器程序，并指定一个 Unix 域套接口(11.6 节)来运行这些程序。会发生什么结果？你怎样修改能使其正常运行？

14.4 对图 11.7 进行修改，在输出对方的协议地址后调用 sleep(5)，每次 read 返回一个大于 0 的值时输出读到的字节数。对图 11.10 进行修改，对向客户发送的结果中的每个字节都调用 write。(在习题1.5的解答中讨论了类似的修改。)在同一台主机上以 TCP 运行客户和服务器。客户读到了多少字节？
在同一台主机上以 Unix 域套接口运行客户和服务器。结果有没有变化？
再在服务器程序中用 send 代替 write，并设置 MSG_EOR 标志。(完成这道题需要一个源自 Berkeley 的实现。)在同一台主机上以 Unix 域套接口运行客户和服务器。结果有没有变化？

14.5 编写一个程序以确定图 4.10 中展示的值。一种方法是建立一个流管道然后 fork。父进程进入一个 for 循环，backlog 从 0 增加到 14。每次循环父进程首先把 backlog 的值写到流管道中。子进程读取这个值，建立一个捆绑到回馈地址的监听套接口，将 backlog 设为该值。然后子进程写流管道，告诉父进程它已准备好。接着父进程尝试建立尽可能多的连接，但设置一个 2 秒的 alarm，因为达到 backlog 上限的 connect 调用将会阻塞，重发 SYN。子进程不调用 accept 以让内核把父进程来的连接送入缓冲队列。当父进程的 alarm 超时时，它从循环的计数器可以知道哪个 connect 达到了 backlog 的上限。在这之后父进程关闭套接口，

并将下一个新的 backlog 的值写到流管道。当子进程读到这下一个值时，它关闭原来的监听套接口，建立一个新的监听套接口，又重新开始这个过程。

14.6　验证在图 14.6 中删掉 bind 调用会导致服务器出错。

第15章 非阻塞I/O

15.1 概 述

缺省状态下,套接口是阻塞方式的。这意味着当一个套接口调用不能立即完成时,进程进入睡眠状态,等待操作完成。我们将可能阻塞的套接口调用分成四种。

1. 输入操作:read、readv、recv、recvfrom 和 recvmsg 函数。如果在一个阻塞的 TCP 套接口(缺省情况)上调用这些函数,而且在套接口接收缓冲区中没有数据,进程将在数据到来前一直睡眠。因为 TCP 是一个字节流,当有数据到来时进程将被唤醒:可以是一个字节的数据,也可以是一个完整的 TCP 数据分节。如果希望等到某个固定数目的数据再返回,可以调用一个自己写的函数 readn(图3.14),或指定 MSG_WAITALL 标志(图 13.6)。

   因为 UDP 是一个数据报协议,如果一个阻塞的 UDP 套接口的接收缓冲区为空,进程将在一个 UDP 数据报到来之前一直处于睡眠状态。

   在一个非阻塞套接口上,如果输入操作不能被满足(在 TCP 套接口上至少有一个字节的数据,或在 UDP 套接口上有一个完整的数据报),它们将会立即返回一个 EWOULDBLOCK 错误。

2. 输出操作:write、writev、send、sendto 和 sendmsg 函数。我们在 2.9 节中说过,对一个 TCP scoket,内核从应用进程缓冲区向套接口发送缓冲区中拷贝数据。如果在套接口发送缓冲区中没有空间,进程会一直睡眠到腾出空间。

   对于一个非阻塞 TCP 套接口,如果在套接口的发送缓冲区中没有空间,输出操作会立即返回一个 EWOULDBLOCK 错误。如果发送缓冲区中有一些空间,返回值为内核能向缓冲区中拷贝的字节数。(这叫做不足计数(short count)。)

   在 2.9 节中还说过,实际上 UDP 套接口没有发送缓冲区。内核只是拷贝应用进程数据并将其向协议栈的下层传递,加上 UDP 和 IP 头部。因此在一个阻塞 UDP 套接口上的输出操作不会阻塞。

3. 接收外来连接:accept 函数。如果在一个阻塞套接口上调用 accept 函数,而且没有新的连接,进程就会进入睡眠状态。

   如果在一个非阻塞套接口上调用 accept 函数,而且没有新的连接,将返回 EWOULDBLOCK 错误。

4. 初始化外出的连接:用于 TCP 的 connect 函数。(回想 connect 可用于 UDP,但它不会建立一个"真实"的连接;它只是让内核保存对方的 IP 地址和端口号。)在 2.5 节中展示了 TCP 连接的建立包含一个三路握手过程,而且 connect 函数在客户接收到它的 SYN 的 ACK 前不会返回。这意味着 TCP connect 总是会使调用它的进程阻塞起码到服务器的一次往返时间(RTT)。

如果在一个非阻塞的 TCP 套接口上调用 connect，而且连接不能马上建立，连接的建立过程将开始（譬如 TCP 三路握手的第一个分组将发出），但返回一个 EINPROGRESS 错误。注意这个错误和前面三种情况下返回的不同。还要注意的是一些连接可以立即建立，这一般在服务器和客户位于同一台主机上时发生，所以即使对于一个非阻塞的 connect，也必须准备处理 connect 返回成功的情况。在 15.3 节中将举一个非阻塞 connect 的例子。

通常，对于不能马上完成的非阻塞 I/O 操作，系统 V 返回 EAGAIN 错误，而源自 Berkeley 的实现返回 EWOULDBLOCK 错误。更混乱的是，Posix.1 指定使用 EAGAIN 而 Posix.1g 指定使用 EWOULDBLOCK。幸运的是，大多数当前使用的系统（包括 SVR4 和 4.4BSD）将这两个错误码定义为相同的值（检查一下你使用的系统中的<sys/errno.h>头文件），因此不管用哪个都没有关系。在本文中我们使用 Posix.1g 中指定的 EWOULDBLOCK。

6.2 节总结了 I/O 可以使用的不同模型，并比较了非阻塞 I/O 和其他模型。本章将提供全部四种类型操作的例子，并且开发一个与 Web 客户程序类似的新的客户程序，它使用非阻塞 connect 同时开始多个 TCP 连接。

15.2　非阻塞读和写：str_cli 函数（修订版）

我们再次回到在 5.5 和 6.4 节中讨论过的 str_cli 函数。后者使用 select 的那个版本，使用的仍是阻塞 I/O。举例来说，如果在标准输入有一行可读，我们用 fgets 读入它，然后用 writen 把它发往服务器。但是如果套接口的缓冲区已满，writen 调用会阻塞。当 writen 调用阻塞时，套接口的接收缓冲区中可能会有可读的数据。类似的，当标准输出比网络慢时，如果套接口上有一行输入可读，我们可能阻塞在后面的 fputs 调用上。这一节的目标是开发这个函数的一个使用非阻塞 I/O 的版本。这样可以避免阻塞，同时做一些有价值的事情。

不幸的是加入非阻塞 I/O 使函数的缓冲区管理显著地复杂化了，因此下面分成若干部分介绍这个函数。我们还要修改对标准输入和标准输出的处理，改为直接使用 read 和 write，以代替使用标准 I/O。这样避免了在非阻塞描述字上使用标准 I/O，这是防止发生问题的良方。

我们维护两个缓冲区：to 容纳从标准输入到服务器去的数据，fr 容纳自服务器到标准输出来的数据。图 15.1 展示了 to 缓冲区的组织和指向缓冲区中的指针。

toiptr 指针指向从标准输入读入的数据可以存放的下一个字节。tooptr 指向下一个必须写入套接口的字节。有（toiptr-tooptr）个字节需写到套接口。可从标准输入读入的字节数是（&to[MAXLINE]－toiptr）。tooptr 一旦到达 oiptr，两个指针都复位到缓冲区的开始。

图 15.2 展示了相应的 fr 缓冲区的组织。

图 15.3 给出了函数的第一部分。

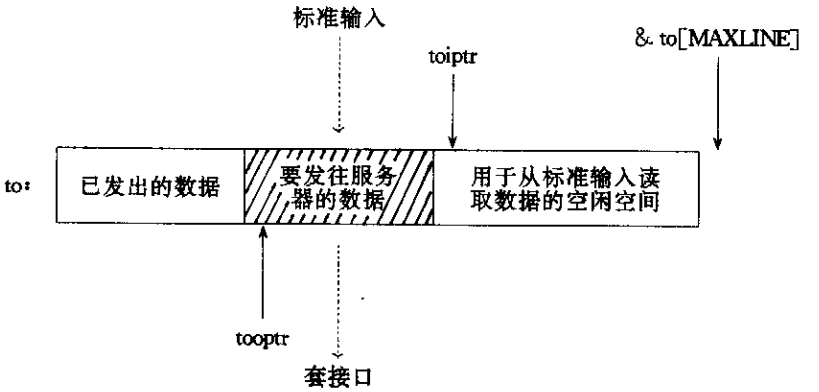


图 15.1 容纳从标准输入到套接口的数据的缓冲区

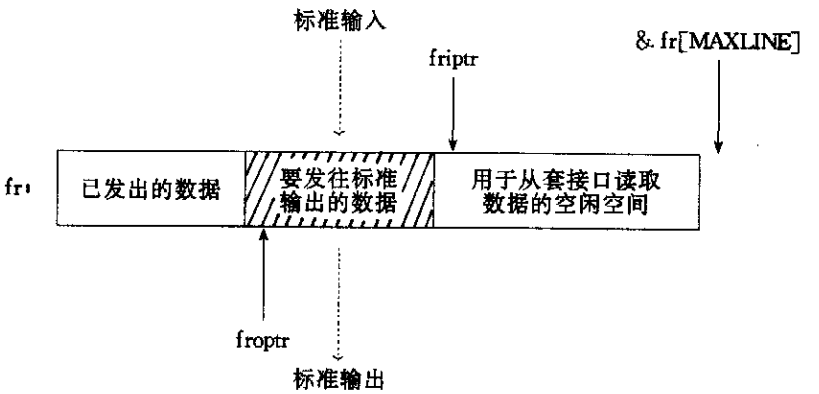


图 15.2 容纳从套接口到标准输出的数据的缓冲区

```
1  #include    "unp.h"
2  void
3  str_cli(FILE *fp, int sockfd)
4  {
5      int          maxfdp1, val, stdineof;
6      ssize_t      n, nwritten;
7      fd_set       rset, wset;
8      char         to[MAXLINE], fr[MAXLINE];
9      char         *toiptr, *tooptr, *friptr, *froptr;
10     val = Fcntl(sockfd, F_GETFL, 0);
11     Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);
12     val = Fcntl(STDIN_FILENO, F_GETFL, 0);
13     Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);
14     val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
15     Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);
16     toiptr = tooptr = to;    /* initialize buffer pointers */
17     friptr = froptr = fr;
18     stdineof = 0;
19     maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
20     for ( ; ; ) {
```

```
21          FD_ZERO(&rset);
22          FD_ZERO(&wset);
23          if (stdineof == 0 && toiptr < &to[MAXLINE])
24              FD_SET(STDIN_FILENO, &rset);  /* read from stdin */
25          if (friptr < &fr[MAXLINE])
26              FD_SET(sockfd, &rset);        /* read from socket */
27          if (tooptr != toiptr)
28              FD_SET(sockfd, &wset);        /* data to write to socket */
29          if (froptr != friptr)
30              FD_SET(STDOUT_FILENO, &wset); /* data to write to stdout */
31          Select(maxfdp1, &rset, &wset, NULL, NULL);
```

图 15.3　str_cli 函数：第一部分，初始化并调用 select[nonblock/strclinonb.c]

将描述字设置为非阻塞

第 10～15 行　用 fcntl 把全部三个描述字设置为非阻塞：连到服务器的套接口、标准输入和标准输出。

初始化缓冲区指针

第 16～19 行　初始化指向两个缓冲区的指针，并将最大的描述字加 1，以用作 select 的第一个参数。

主循环：准备调用 select

第 20 行　和这个函数在前面图 6.13 中的版本相似，函数的主循环是一个 select 调用，紧接着是各种我们感兴趣的条件测试。

指定感兴趣的描述字

第 21～30 行　将两个描述字集都清零，然后在每个集合中最多打开 2 位。如果在标准输入上还没有读到一个文件结束符，而且在缓冲区中至少还有一个字节的空间可存放数据，读集合中与标准输入对应的位将被打开。如果在 fr 缓冲区中有至少存放一个字节数据的空间，读集合中与套接口对应的位将被打开。如果在 to 缓冲区中有要写入套接口的数据，写集合中与套接口对应的位将被打开，最后，如果在 fr 缓冲区中有发往标准输出的数据，写集合中与标准输出对应的位将被打开。

调用 select

第 31 行　调用 select，等待四个条件之一得到满足。我们没有为这个函数设置超时。

这个函数的下一部分在图 15.4 中给出。其中包含 select 返回后进行的前两个测试（总共 4 个）。

从标准输入 read

第 32～33 行　如果标准输入上有数据可读，调用 read。第三个参数是 to 缓冲区中可用的空间数。

处理非阻塞错误

第 34～35 行　如果发生错误，而且是 EWOULDBLOCK，就不作任何操作。一般这种条件“不应该发生”，这意味着，select 告诉我们该描述字可读但 read 却返回 EWOULDBLOCK，不过我们还是对这种情况进行了处理。

read返回文件结束符

第36～40行 如果read返回0，标准输入处理就结束。我们设置stdineof标志。如果在to缓冲区中没有数据待发送(tooptr等于toiptr)，就用shutdown向服务器发送FIN。如果在to缓冲区中仍有数据待发送，则要等到缓冲区写入套接口后再发送FIN。

我们输出一行到标准错误输出以表示文件结束条件，同时给出当前的时间，在介绍这个函数后我们会展示怎样使用这个输出信息。类似的fprintf贯穿这个函数的始终。

```
32      if (FD_ISSET(STDIN_FILENO, &rset)) {
33          if ( (n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr)) < 0) {
34              if (errno != EWOULDBLOCK)
35                  err_sys("read error on stdin");
36          } else if (n == 0) {
37              fprintf(stderr, "%s: EOF on stdin\n", gf_time());
38              stdineof = 1;                   /* all done with stdin */
39              if (tooptr == toiptr)
40                  Shutdown(sockfd, SHUT_WR);/* send FIN */
41          } else {
42              fprintf(stderr, "%s: read %d bytes from stdin\n", gf_time(), n);
43              toiptr += n;                    /* # just read */
44              FD_SET(sockfd, &wset);  /* try and write to socket below */
45          }
46      }
47      if (FD_ISSET(sockfd, &rset)) {
48          if ( (n = read(sockfd, friptr, &fr[MAXLINE] - friptr)) < 0) {
49              if (errno != EWOULDBLOCK)
50                  err_sys("read error on socket");
51          } else if (n == 0) {
52              fprintf(stderr, "%s: EOF on socket\n", gf_time());
53              if (stdineof)
54                  return;     /* normal termination */
55              else
56                  err_quit("str_cli: server terminated prematurely");
57          } else {
58              fprintf(stderr, "%s: read %d bytes from socket\n",
59                      gf_time(), n);
60              friptr += n;    /* # just read */
61              FD_SET(STDOUT_FILENO, &wset); /* try and write below */
62          }
63      }
```

图15.4 str_cli函数：第二部分，从标准输入或套接口读入数据[nonblock/strclinonb.c]

read返回数据

第41～45行 当read返回数据时，我们相应地增加toiptr。另外还打开在写集合中与套接口对应的位，使后半部分的循环中对该位的测试能返回真，以试图在该套接口上write。

这是在编写代码时艰难的设计决断之一。这里有几种选择。我们可以用什么也不做来代替在写集合中设置对应的位。但是这样在已经知道有数据可向套接口写的情况下，还要进行另一次循环和select调用。另一个选择是把向套接

口写的代码复制到这儿，但这看上去有些浪费，而且是一个潜在的错误源（万一在复制的代码中存在缺陷，在一个位置修复了但另一个位置却没有）。最后，我们可以创建一个写套接口的函数，用调用这个函数来代替复制代码，但这个函数需要共享三个 str_cli 使用的局部变量，这足以要求这些变量成为全局变量。最终作出的选择是作者自认为最好的。

从套接口读

第 47～63 行　这些代码行和刚才介绍的当标准输入可读时的 if 语句类似。如果 read 返回 EWOULDBLOCK，则不做任何处理。如果遇到从服务器来的一个文件结束符，那么在我们已经遇到从标准输入来的文件结束符的情况下是没有问题的，但不期望是别的情况。如果 read 返回一些数据，我们就增加 friptr 并打开写集合中标准输出对应的位，以试图在该函数的下一部分中写数据。

图 15.5 给出了该函数的最后一部分。

```
64          if (FD_ISSET(STDOUT_FILENO, &wset) && ( (n = friptr - froptr) > 0)) {
65              if ( (nwritten = write(STDOUT_FILENO, froptr, n)) < 0) {
66                  if (errno != EWOULDBLOCK)
67                      err_sys("write error to stdout");
68              } else {
69                  fprintf(stderr, "%s: wrote %d bytes to stdout\n",
70                          gf_time(), nwritten);
71                  froptr += nwritten;             /* # just written */
72                  if (froptr == friptr)
73                      froptr = friptr = fr;   /* back to beginning of buffer */
74              }
75          }
76          if (FD_ISSET(sockfd, &wset) && ( (n = toiptr - tooptr) > 0)) {
77              if ( (nwritten = write(sockfd, tooptr, n)) < 0) {
78                  if (errno != EWOULDBLOCK)
79                      err_sys("write error to socket");
80              } else {
81                  fprintf(stderr, "%s: wrote %d bytes to socket\n",
82                          gf_time(), nwritten);
83                  tooptr += nwritten;     /* # just written */
84                  if (tooptr == toiptr) {
85                      toiptr = tooptr = to;   /* back to beginning of buffer */
86                      if (stdineof)
87                          Shutdown(sockfd, SHUT_WR);      /* send FIN */
88                  }
89              }
90          }
91      }
92  }
```

图 15.5　str_cli 函数：第三部分，写标准输出或套接口[nonblock/strclinonb.c]

写标准输出

第 64～67 行　如果标准输出可写，而且要写的字节数大于 0，就调用 write。如果返回 EWOULDBLOCK，则不做任何处理。注意这种情况完全可能发生，因为这个函数前一部分末尾的代码在不知道 write 是否能成功的条件下，打开了写集合中标准输出对应的位。

write 成功

第68～74行 如果write成功，froptr就加上写入的字节数。如果输出指针追上了输入指针，两个指针就都复位，指向缓冲区的开头。

写套接口

第76～90行 这一段代码和刚才介绍的写标准输出的代码类似。有一个不同是当输出指针追上输入指针时，不仅两个指针都复位到缓冲区的开头，而且如果在标准输入上遇到文件结束符，还向服务器发出FIN。

现在检查一下这个函数的操作和非阻塞I/O的重迭。图15.6给出了str_cli调用的gf_time函数。

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 char *
 4 gf_time(void)
 5 {
 6     struct timeval tv;
 7     static char str[30];
 8     char    *ptr;

 9     if (gettimeofday(&tv, NULL) < 0)
10         err_sys("gettimeofday error");

11     ptr = ctime(&tv.tv_sec);
12     strcpy(str, &ptr[11]);
13         /* Fri Sep 13 00:00:00 1986\n\0 */
14         /* 012345678901234567890123 4   */
15     snprintf(str+8, sizeof(str)-8, ".%06ld", tv.tv_usec);

16     return(str);
17 }
```

图15.6 gf_time函数：返回指向时间字符串的指针[lib/gf_time.c]

这个函数返回一个包含当前时间的字符串，包括微秒，格式如下

```
12:34:56.123456
```

这里特意和tcpdump输出的时间戳的格式一样。还要注意的是，str_cli函数中的所有fprintf调用都写到标准错误输出，允许我们把标准输出(服务器回射的行)和诊断输出分开。然后可以运行客户程序和tcpdump，并将这些诊断输出和tcpdump的输出放在一起以时间排序。这让我们能查看程序中到底发生了什么，并和对应的TCP动作联系起来。

举例来说，我们首先在主机solaris上运行tcpdump，只捕获去往或来自端口7(echo服务器)的TCP分节，输出信息存到名为tcpd的文件中。

```
solaris % tcpdump -w tcpd tcp and port 7
```

然后在这台主机上运行我们的TCP客户程序，指明服务器在主机bsdi上。

```
solaris % tcpcli02 206.62.226.35 < 2000.lines > out 2> diag
```

标准输入是文件2000.lines，跟图6.13中用的文件一样。标准输出定向到文件out，标准

错误输出定向到文件 diag。完成后我们运行

```
solaris % diff 2000.lines out
```

以验证回射的行与输入行相同。最后用中断键终止 tcpdump 并输出 tcpdump 记录，将这些记录和客户程序输出的诊断信息一起排序。图 15.7 给出了这个结果的第一部分。

```
solaris % tcpdump -r tcpd -N | sort diag -
10:18:34.486392 solaris.33621 > bsdi.echo: S 1802738644:1802738644(0)
                                            win 8760 <mss 1460>
10:18:34.488278 bsdi.echo > solaris.33621: S 3212986316:3212986316(0)
                                            ack 1802738645 win 8760<mss 1460>
10:18:34.488490 solaris.33621 > bsdi.echo: . ack 1 win 8760
10:18:34.491482: read 4096 bytes from stdin
10:18:34.518663 solaris.33621 > bsdi.echo: p 1:1461(1460) ack 1 win 8760
10:18:34.519016: wrote 4096 bytes to socket
10:18:34.528529 bsdi.echo > solaris.33621: p 1:1461(1460) ack 1461 win 8760
10:18:34.528785 solaris.33621 > bsdi.echo: . 1461:2921(1460) ack 1461 win 8760
10:18:34.528900 solaris.33621 > bsdi.echo: p 2921:4097(1176) ack 1461 win 8760
10:18:34.528958 solaris.33621 > bsdi.echo: . ack 1461 win 8760
10:18:34.536193 bsdi.echo > solaris.33621: . 1461:2921(1460) ack 4097 win 8760
10:18:34.536697 bsdi.echo > solaris.33621: P 2921:3509(588) ack 4097 win 8760
10:18:34.544636: read 4096 bytes from stdin
10:18:34.568505: read 3508 bytes from socket
10:18:34.580373 solaris.33621 > bsdi.echo: . ack 3509 win 8760
10:18:34.582244 bsdi.echo > solaris.33621: P 3509:4097(588) ack 4097 win 8760
10:18:34.593354: wrote 3508 bytes to stdout
10:18:34.617272 solaris.33621 > bsdi.echo: P 4097:5557(1460) ack 4097 win 8760
10:18:34.617610 solaris.33621 > bsdi.echo: P 5557:7017(1460) ack 4097 win 8760
10:18:34.617908 solaris.33621 > bsdi.echo: P 7017:8193(1176) ack 4097 win 8760
10:18:34.618062: wrote 4096 bytes to socket
10:18:34.623310 bsdi.echo > solaris.33621: . ack 8193 win 8760
10:18:34.626129 bsdi.echo > solaris.33621: . 4097:5557(1460) ack 8193 win 8760
10:18:34.626339 solaris.33621 > bsdi.echo: . ack 5557 win 8760
10:18:34.626611 bsdi.echo > solaris.33621: P 5557:6145(588) ack 8193 win 8760
10:18:34.628396 bsdi.echo > solaris.33621: . 6145:7605(1460) ack 8193 win 8760
10:18:34.643524: read 4096 bytes from stdin
10:18:34.667305: read 2636 bytes from socket
10:18:34.670324 solaris.33621 > bsdi.echo: . ack 7605 win 8760
10:18:34.672221 bsdi.echo > solaris.33621: P 7605:8193(588) ack 8193 win 8760
10:18:34.691039: wrote 2636 bytes to stdout
```

图 15.7　排序后的 tcpdump 输出和诊断输出

我们把那些包含 SYN 的过长的行折行了，还删掉了 Solaris 分节的(DF)记号，它表示设置了不分片位(路径 MTU 发现)。tcpdump 的-N 选项只输出全限定域名(solaris.kohala.com)中的主机部分(solaris)。

通过这个输出，我们可以把发生的事情以时间线图描绘出来。在图 15.8 中展示了这个结果，图中时间是向下递增的。

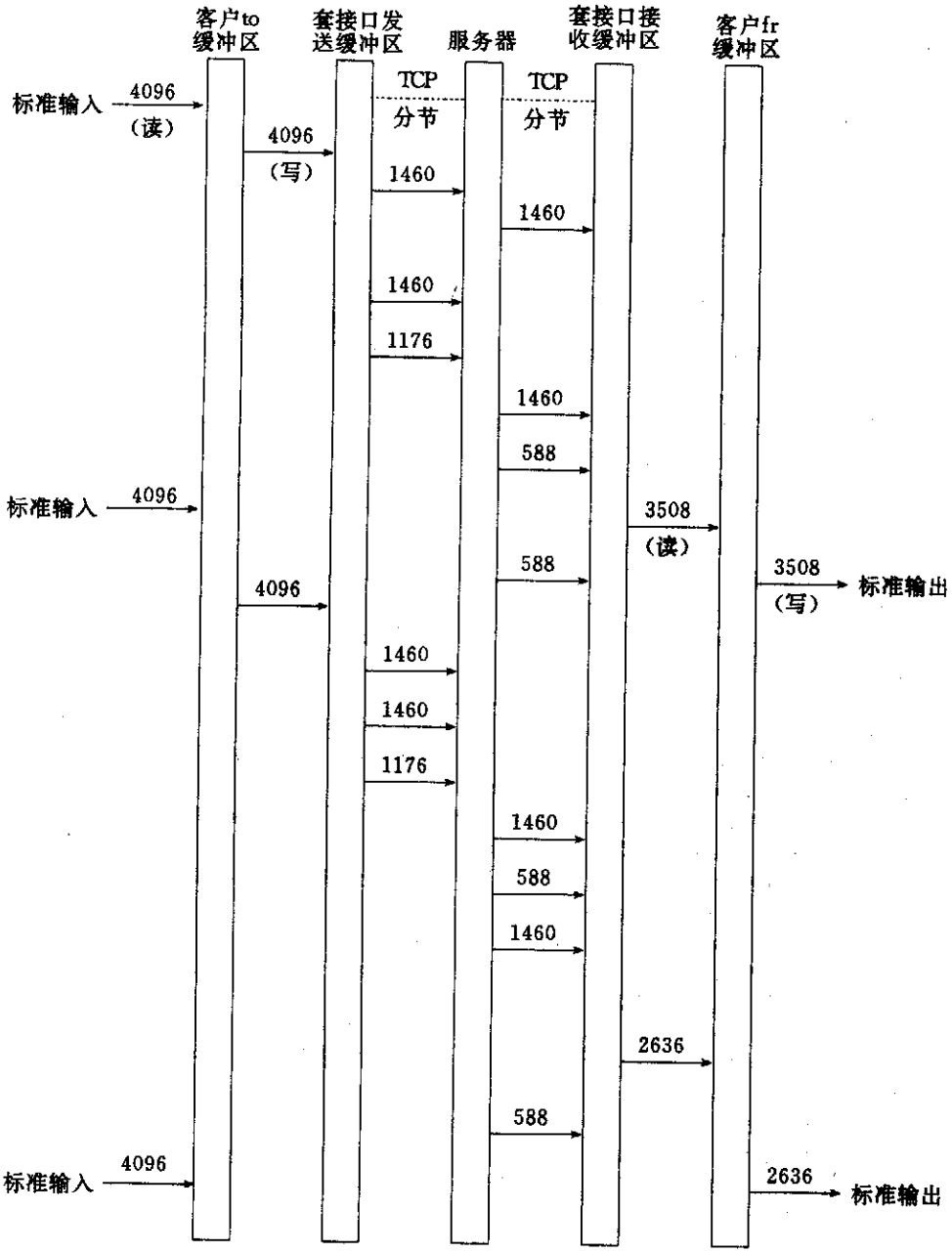


图 15.8 非阻塞实例的时间线

在上图中没有给出 ACK 分节。还要认识到当程序输出"wrote N bytes to stdout(已写N字节到标准输出)"时,write 已经返回了,可能导致 TCP 发送了一个或多个分节的数据。

从时间线我们可以看到客户和服务器之间的数据交换的动态性。用非阻塞 I/O 使程序利用了这些动态性的优势,在能做读或写操作时就做这些操作。用 select 函数让内核告诉我们什么时候能做 I/O 操作。

可以用 6.7 节中所用的同一个 2000 行的文件和同样的服务器主机(与客户主机的

RTT 为 175ms)来测算非阻塞版本运行所用的时间。和 6.7 节中的版本所花的 12.3 秒相比,非阻塞版本现在花了6.9秒。因此就本例子而言,非阻塞 I/O 减少了向服务器发送一个文件的全部时间。

str_cli 的更简单版本

刚才给出的 str_cli 的非阻塞版本是比较复杂的:大约有 135 行代码,与此相比,图 6.13 中使用 select 和阻塞 I/O 的版本是 40 行,最初的停-等版本(图 5.5)是 20 行。代码的长度从 20 行翻番到 40 行是值得的,因为在批操作模式下速度几乎提高了 30 倍,而且在阻塞的描述字上使用 select 并不太复杂。但在增加代码复杂度的情况下,是否值得用非阻塞 I/O 编写应用程序呢?回答是否定的。每当我们发现需要使用非阻塞 I/O 时,把应用程序分成多个进程(使用 fork)或线程(第 23 章)通常会更简单。

图 15.9 是 str_cli 函数的另一个版本,这个函数用 fork 把自己分成了两个进程。

```
1 #include    "unp.h"
2 void
3 str_cli(FILE *fp, int sockfd)
4 {
5     pid_t pid;
6     char   sendline[MAXLINE], recvline[MAXLINE];
7     if ( (pid = Fork()) == 0) {          /* child: server -> stdout */
8         while (Readline(sockfd, recvline, MAXLINE) > 0)
9             Fputs(recvline, stdout);
10        kill(getppid(), SIGTERM);       /* in case parent still running */
11        exit(0);
12    }
13    /* parent: stdin -> server */
14    while (Fgets(sendline, MAXLINE, fp) != NULL)
15        Writen(sockfd, sendline, strlen(sendline));
16    Shutdown(sockfd, SHUT_WR);      /* EOF on stdin, send FIN */
17    pause();
18    return;
19 }
```

图 15.9　str_cli 函数使用 fork 的版本[nonblock/strclifork.c]

这个函数一开始就调用 fork 分为父进程和子进程。子进程把从服务器来的行拷贝到标准输出,父进程把标准输入来的行拷贝到服务器,如图 15.10 所示。

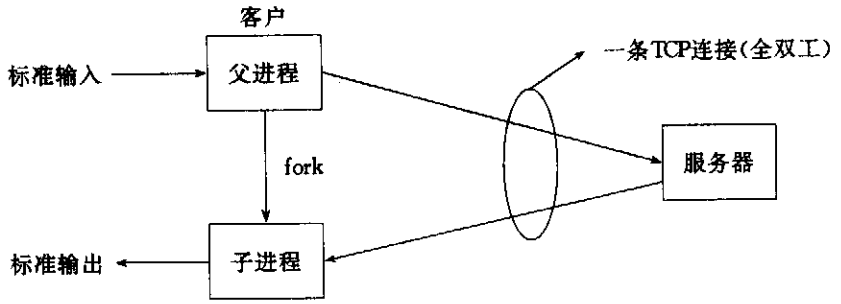


图 15.10　使用两个进程的 str_cli 函数

我们清楚地注意到这个TCP连接是全双工的，父进程和子进程共享同一个套接口：父进程向该套接口写，子进程从该套接口读。这里只有一个套接口、一个套接口接收缓冲区和一个套接口发送缓冲区，但这个套接口被两个描述字引用：一个在父进程，一个在子进程。

我们还要担心终止顺序的问题。通常终止是在标准输入上遇到一个文件结束符时发生。父进程读到这个文件结束符后调用shutdown发送一个FIN。(父进程不能调用close，参见习题15.1。)但这时，子进程需要继续把从服务器来的数据拷贝到标准输出，直到在套接口上读到一个文件结束符。

也有可能服务器进程过早终止(5.12节)，如果发生了这种情况，子进程将在套接口上读到一个文件结束符。如果这样子进程必须通知父进程停止从标准输入向套接口拷贝数据(见习题15.2)。在图15.9中，在父进程仍在运行的情况下，子进程向父进程发送SIGTERM信号(见习题15.3)。另一种方法是让子进程终止，从而使父进程获SIGCHLD。

父进程在完成拷贝后调用pause，进入睡眠状态直到收到信号。即使父进程不捕获任何信号，pause也使父进程进入睡眠状态，直到收到从子进程来的SIGTERM信号为止。这个信号的缺省动作是终止该进程，这在这个例子中是十分合适的。通常这个子进程在父进程之后结束，但由于我们用shell的time命令来计时，因此父进程终止时计时也结束。所以在这里让父进程等待子进程，以更准确地衡量str_cli的这个版本所使用的时间。

注意这个版本和这一节前面的非阻塞版本相比要简单得多。非阻塞的版本同时管理四个不同的I/O流，而且由于四个流都是非阻塞的，就必须考虑这四个流上的部分读和写的问题。但在这个fork版本中，每个进程只处理两个I/O流，从一个拷贝到另一个。因为如果输入流没有数据可读，对应的输出流就没有数据可写，所以不需要非阻塞I/O。

str_cli所用的时间

现在已经有str_cli函数的四种不同的版本。我们总结一下这些版本以及一个使用线程的版本(图23.2)在从一个Solaris 2.5客户主机向一个RTT为175毫秒的服务器主机拷贝2000行所花的时间。

- 354.0秒，停等方式(图5.5)，
- 12.3秒，使用select和阻塞I/O(图6.13)，
- 6.9秒，非阻塞I/O(图15.3)，
- 8.7秒，fork(图15.9)，
- 8.5秒，线程版本(图23.2)。

非阻塞版本的速度几乎是使用阻塞I/O和select版本的两倍那么快。使用fork的简单版本比非阻塞版本慢，然而，考虑到非阻塞版本代码的复杂性，我们推荐简单的版本。

15.3 非阻塞connect

在一个TCP套接口被设置为非阻塞后调用connect，connect会立即返回一个EINPROGRESS错误，但TCP的三路握手继续进行。在这之后我们可以用select检查这个连接是否建立成功。非阻塞的connect有三种用途。

1. 我们可以在三路握手的同时做一些其他的处理。connect 要花一个往返时间完成(2.5 节),而且可以是在任何地方,从几个毫秒的局域网到几百毫秒或几秒的广域网。在这段时间内我们可能有一些其他的处理想要执行。
2. 可以用这种技术同时建立多个连接。这在 Web 浏览器中很普遍,在 15.5 节中有这样的一个例子。
3. 由于我们用 select 等待连接的完成,因此可以给 select 设置一个时间限制,从而缩短 connect 的超时时间。在多数实现中,connect 的超时时间在 75 秒到几分钟之间。有时应用程序想要一个更短的超时时间,使用非阻塞 connect 就是一种方法。13.2 节中谈到了在套接口操作中设置超时的其他方法。

非阻塞 connnct 听起来虽然简单,有一些细节问题我们还得处理。

- 即使套接口是非阻塞的,如果连接的服务器在同一台主机上,在调用 connect 时连接通常会立刻建立。我们必须处理这种情况。
- 源自 Berkeley 的实现(和 Posix.1g)有两条与 select 和非阻塞 I/O 相关的规则:(1)当连接成功建立时,描述字变成可写(TCPv2 第 531 页),(2)当连接建立出错时,描述字变成既可读又可写(TCPv2 第 530 页)。

> 这两条和 select 相关的规则是从 6.3 节中使描述字就绪的相关规则而来。TCP 套接口在它的发送缓冲区有空闲空间(对于连接中的套接口这个条件总是成立的,因为还没有向它写任何数据),而且套接口已连接(当三路握手完成时才成立)时是可写的。待处理错误则导致套接口变成既可读又可写。

在下面的例子中提到了许多有关 connect 的移植性问题。

15.4　非阻塞 connect:时间/日期客户程序

图 15.11 给出了 connect_nonb 函数,它执行一个非阻塞 connect。我们把图 1.5 的 connect 替换成:

```
if (connect_nonb(sockfd, (SA *) &servaddr, sizeof(servaddr), 0) < 0)
    err_sys("connect error");
```

前三个参数和 connect 的一样,第四个参数是等待连接完成的秒数。它的值为 0 表示 select 没有超时时间;因此内核将使用通常的 TCP 连接建立超时时间。

设置非阻塞套接口

第 9～10 行　调用 fcntl 设置非阻塞套接口。

第 11～14 行　开始非阻塞 connect。我们期望的错误是 EINPROGRESS,表示开始建立连接但还没有完成(TCPv2 第 466 页),其他的错误将被返回给调用者。

在建立连接时进行其他的处理

第 15 行　在等待连接建立完成时可以做任何我们想做的事情。

检查连接是否立即建立

第 16～17 行　如果非阻塞 connect 返回 0,连接已建立。前面已经提到,这种情况在服

务器和客户在同一台主机上时可能发生。

```
1 #include    "unp.h"
2 int
3 connect_nonb(int sockfd, const SA *saptr, socklen_t salen, int nsec)
4 {
5     int     flags, n, error;
6     socklen_t len;
7     fd_set  rset, wset;
8     struct timeval tval;
9     flags = Fcntl(sockfd, F_GETFL, 0);
10    Fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
11    error = 0;
12    if ( (n = connect(sockfd, saptr, salen)) < 0)
13        if (errno != EINPROGRESS)
14            return(-1);
15    /* Do whatever we want while the connect is taking place. */
16    if (n == 0)
17        goto done;  /* connect completed immediately */
18    FD_ZERO(&rset);
19    FD_SET(sockfd, &rset);
20    wset = rset;
21    tval.tv_sec = nsec;
22    tval.tv_usec = 0;
23    if ( (n = Select(sockfd+1, &rset, &wset, NULL,
24                     nsec ? &tval : NULL)) == 0) {
25        close(sockfd);          /* timeout */
26        errno = ETIMEDOUT;
27        return(-1);
28    }
29    if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
30        len = sizeof(error);
31        if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0)
32            return(-1);             /* Solaris pending error */
33    } else
34        err_quit("select error: sockfd not set");
35 done:
36    Fcntl(sockfd, F_SETFL, flags);     /* restore file status flags */
37    if (error) {
38        close(sockfd);          /* just in case */
39        errno = error;
40        return(-1);
41    }
42    return(0);
43 }
```

图 15.11 启动一个非阻塞 connect[lib/connect_nonb.c]

调用 select

第 18～24 行　调用 select 等待套接口变成可读或可写。把 rset 清零，打开该描述字集中对应 sockfd 的位，然后把 rset 拷贝到 wset。这个赋值可能是一个结构的赋值，因为描述字集通常表示为结构。我们还初始化 timeval 结构并调用 select。如果调用者将第四个参数指定为 0(使用缺省的超时时间)，我们就必须把 select 的最后一个参数指定为一个空指针而不是一个值为 0 的 timeval 结构(后者意味着根本不等待)。

处理超时

第 25～28 行　如果 select 返回 0，那么时钟超时，于是返回 ETIMEOUT 错误给调用者。我们还关闭套接口，以防止三路握手继续下去。

检查可读或可写条件

第 29～34 行　如果描述字可读或可写，我们调用 getsockopt 得到套接口上待处理的错误(SO_ERROR)。如果连接建立成功，这个值将为 0。如果建立连接时遇到错误，这个值是连接错误对应的 errno 值(譬如 ECONNREFUSED、ETIMEDOUT 等)。这里还遇到第一个可移植性问题。如果发生错误，getsockopt 源自 Berkeley 的实现返回 0，待处理的错误在变量 error 中返回。但 Solaris 让 getsockopt 返回－1，errno 置为待处理的错误。我们的程序对两种情况都处理。

关闭非阻塞方式并返回

第 36～42 行　恢复文件状态标志并返回。如果 getsockopt 返回的 error 变量不为 0，就将这个值存到 errno，函数返回－1。

就像前面所说，非阻塞 connect 在不同的套接口实现中存在移植性问题。首先，有可能在调用 select 之前连接已建立而且对方的数据已到来。在这种情况下，连接成功时套接口将既可读又可写，和连接失败时一样。图 15.11 中的代码通过调用 getsockopt 并检查套接口上待处理的错误来处理这种情况。

如果我们不能假定返回可写是成功的唯一一种情况，下一个移植性问题就是怎样判断连接是否成功建立。在 Usenet 上曾发表过各种解决方法。这些方法可以代替图 15.11 中的 getsockopt 调用。

1. 调用 getpeername 代替 getsockopt。如果这个调用失败，返回 ENOTCONN，表示连接失败，我们必须以 SO_ERROR 调用 getsockopt 得到套接口上待处理的错误。
2. 调用 read，长度参数为 0。如果 read 失败，表示 connect 失败，而且 read 返回的 errno 指明了连接失败的原因。如果连接成功，read 应返回 0。
3. 再调用 connect 一次。它应该失败，如果错误是 EISCONN，那么套接口已经连接而且第一次连接是成功的。

不幸的是非阻塞 connect 是网络编程中最不好移植的部分。要准备应付移植性问题，特别是对那些老的实现，一种简单的技术是创建一个线程(第 23 章)来处理这个连接。

被中断的 connect

如果在一个阻塞式套接口上调用的 connect，在 TCP 的三路握手完成前被中断，譬如说被捕获的信号中断，将会发生什么呢？假定 connect 不被自动重启，它会返回 EINTR。但是

我们不能再调用 connect 等待连接建立完成，这样做将返回 EADDRINUSE。

在这种情况下应该做的是调用 select，就和这节中对非阻塞 connect 所做的一样。然后 select 在连接建立成功(使套接口可写)或连接失败(使套接口既可读又可写)时返回。

> Posix.1g 明确地说明了调用 XTI 函数 t_connect 被信号中断时做些什么，但没有说明对 connect 被中断如何处理。上面介绍的是源自 Berkeley 的内核对于这种情况的处理。

15.5 非阻塞 connect：Web 客户程序

非阻塞 connect 的现实例子是从 Netscape 的 Web 客户程序(TCPv3 的 13.4 节)开始的。这个客户通过和 Web 服务器建立 HTTP 连接以获取主页。在页面中通常有多个引用指向其他 Web 页面。客户可以使用非阻塞 connect 同时取多个页面，以此代替每次只取一个页面。图 15.12 展示了一个并行建立多个连接的例子。最左边显示了顺序操作全部三个连接的的情况。假定第一个连接用 10 个时间单位，第二个用 15 个，第三个用 4 个，总计 29 个时间单位。

中间是并行操作两个连接的情况。在时间为 0 时开始前两个连接，当其中之一结束时，开始第三个连接。总共的时间差不多减半，从 29 到 15，但这只是理想的情况。如果并行进行的连接共享同一条链路(譬如客户通过一个拨号调制解调器连到因特网)，那每一个连接之间相互竞争有限的资源，各个连接所花的时间可能会变得更长一些。举例来说，花 10 个时间单位可能会变成 15，15 可能变成 20，4 可能变成 6。虽然如此，总共的时间为 21，仍然比顺序的情况要短。

第三种情况并行操作三个连接，而且再次假设这三个连接之间没有冲突(理想情况)。但在这个例子中总共所花的时间和第二种情况一样(15 个时间单位)。

在处理 Web 客户时，第一个连接是它自己完成的，接着是多个连接，连往从第一个连接的数据中找到的引用。在图 15.13 中展示了这一点。

在进一步优化时，客户可以在第一个连接完成前开始分析返回的数据，并按需要随时进行附加的连接。

由于要同时操作多个非阻塞连接，我们不能使用图 15.11 中的 connect_nonb 函数，因为它在连接建立前不返回。我们必须自己管理这些连接。

我们的程序最多会从 Web 服务器读 20 个文件。最大并行连接数、服务器的主机名和每个从服务器取的文件名作为命令行参数。程序运行的一个典型例子如下：

```
solaris % web 3 www.foobar.com / image1.gif image2.gif \
image3.gif image4.gif image5.gif \
image6.gif image7.gif
```

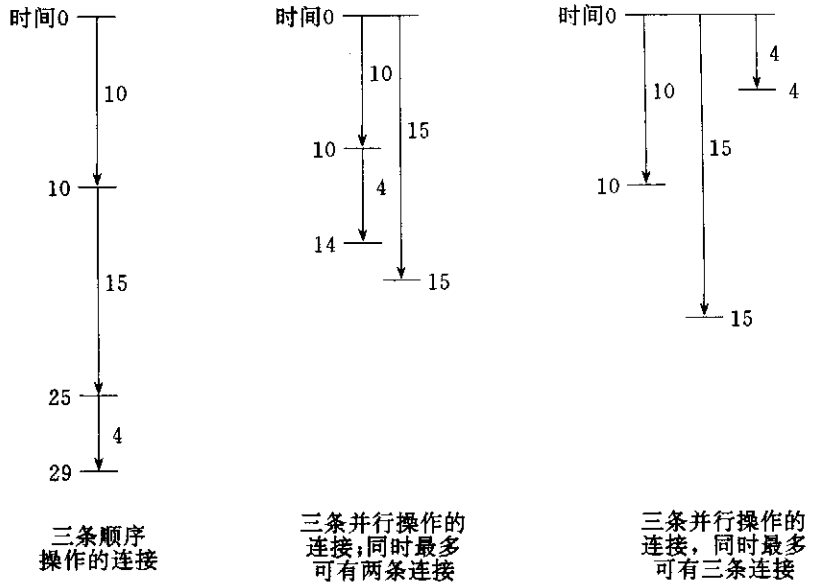


图 15.12　并行操作多个连接

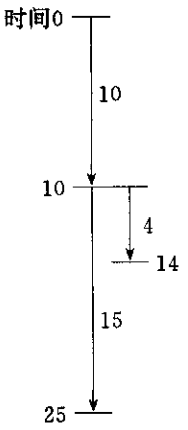


图 15.13　完成第一个连接，然后并行操作多个连接

命令行参数指定最多三个连接、服务器的主机名、主页的文件名(/是服务器的根页面)和随后要读的七个文件(在这个例子中都是 GIF 图像)。这七个文件通常是被主页引用的，Web 客户会读入主页并分析 HTML 以获得这些文件名。我们不想在这个例子中加入 HTML 分析使其复杂化，因此在命令行上指定文件名。

这个例子较大，因此我们把它分成几个部分给出。图 15.14 是每个文件都包括的 web.h 头文件。

```
1 #include    "unp.h"
2 #define MAXFILES    20
3 #define SERV        "80"        /* port number or service name */
4 struct file {
```

```
 5   char        * f_name;              /* filename */
 6   char        * f_host;              /* hostname or IPv4/IPv6 address */
 7   int     f_fd;                      /* descriptor */
 8   int   f_flags;                     /* F_xxx below */
 9 } file[MAXFILES];

10 #define F_CONNECTING    1           /* connect() in progress */
11 #define F_READING       2           /* connect() complete; now reading */
12 #define F_DONE          4           /* all done */

13 #define GET_CMD             "GET %s HTTP/1.0\r\n\r\n"

14                     /* globals */
15 int          nconn, nfiles, nlefttoconn, nlefttoread, maxfd;
16 fd_set       rset, wset;

17                     /* function prototypes */
18 void   home_page(const char *, const char *);
19 void   start_connect(struct file *);
20 void   write_get_cmd(struct file *);
```

图15.14 web.h头文件[nonblock/web.h]

定义file结构

第2～13行 本程序从Web服务器读最多MAXFILES个文件。我们维护一个file结构,其中包含关于每个文件的信息:它的名字(从命令行参数复制而来)、文件所在的服务器的主机名或IP地址、取这个文件用的套接口描述字以及一个表明正在对这个文件做什么(连接、读或完成)的标志集。

定义全局变量和函数原型。

第14～20行 定义全局变量和马上要叙述的函数原型。

图15.15给出了main函数的第一部分。

```
 1 #include    "web.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int           i, fd, n, maxnconn, flags, error;
 6     char    buf[MAXLINE];
 7     fd_set        rs, ws;

 8     if (argc < 5)
 9         err_quit("usage: web <#conns> <hostname> <homepage> <file1>
           ...");
10     maxnconn = atoi(argv[1]);

11     nfiles = min(argc - 4, MAXFILES);
12     for (i = 0; i < nfiles; i++) {
13         file[i].f_name = argv[i + 4];
14         file[i].f_host = argv[2];
15         file[i].f_flags = 0;
16     }
17     printf("nfiles = %d\n", nfiles);

18     home_page(argv[2], argv[3]);
```

```
19      FD_ZERO(&rset);
20      FD_ZERO(&wset);
21      maxfd = -1;
22      nlefttoread = nlefttoconn = nfiles;
23      nconn = 0;
```

图 15.15　同时 connect 的第一部分：全局变量和 main 的开头[nonblock/web.c]

处理命令行参数

第 11～17 行　用从命令行参数来的相关信息填写 file 结构。

读主页

第 18 行　下面给出的 home_page 函数创建一个 TCP 连接，向服务器发出一个命令，然后读主页。这是第一个连接，在开始并行建立多条连接之前自己完成。

初始化全局变量

第 19～23 行　初始化两个描述字集合，一个读一个写。maxfd 是 select 使用的最大描述字(我们把它初始化成－1，因为描述字是非负的)，nlefttoread 是剩余要读的文件数(当它到 0 时程序就结束)，nlefttoconn 是还需要一个 TCP 连接的文件数，nconn 是当前打开的连接数(它不能超过第一个命令行参数)。

图 15.16 给出了在 main 函数开始时调用了一次的 home_page 函数。

```
1  #include    "web.h"
2  void
3  home_page(const char *host, const char *fname)
4  {
5      int          fd, n;
6      char    line[MAXLINE];
7      fd = Tcp_connect(host, SERV); /* blocking connect() */
8      n = snprintf(line, sizeof(line), GET_CMD, fname);
9      Writen(fd, line, n);
10     for ( ; ; ) {
11         if ( (n = Read(fd, line, MAXLINE)) == 0)
12             break;              /* server closed connection */
13         printf("read %d bytes of home page\n", n);
14         /* do whatever with data */
15     }
16     printf("end-of-file on home page\n");
17     Close(fd);
18 }
```

图 15.16　home_page 函数[nonblock/home_page.c]

和服务器建立连接

第 7 行　tcp_connect 和服务器建立一个连接。

向服务器发 HTTP 命令；读应答

第 8～17 行　发出一个 HTTP GET 命令以获取主页(一般为/)。读应答数据(我们不对这些数据做任何处理)，然后关闭连接。

图15.17中给出的下一个函数start_connect启动非阻塞connect。

创建套接口，设置非阻塞方式

第7～13行 调用host_serv函数(图11.5)查找和转换主机名和服务名，返回一个指向addrinfo结构的数组的指针。我们只用第一个结构。创建一个TCP套接口并将其设置为非阻塞方式。

```
1 #include    "web.h"
2 void
3 start_connect(struct file * fptr)
4 {
5     int     fd, flags, n;
6     struct addrinfo * ai;
7     ai = Host_serv(fptr->f_host, SERV, 0, SOCK_STREAM);
8     fd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
9     fptr->f_fd = fd;
10    printf("start_connect for %s, fd %d\n", fptr->f_name, fd);
11        /* Set socket nonblocking */
12    flags = Fcntl(fd, F_GETFL, 0);
13    Fcntl(fd, F_SETFL, flags | O_NONBLOCK);
14        /* Initiate nonblocking connect to the server. */
15    if ( (n = connect(fd, ai->ai_addr, ai->ai_addrlen)) < 0) {
16        if (errno != EINPROGRESS)
17            err_sys("nonblocking connect error");
18        fptr->f_flags = F_CONNECTING;
19        FD_SET(fd, &rset);          /* select for reading and writing */
20        FD_SET(fd, &wset);
21        if (fd > maxfd)
22            maxfd = fd;
23    } else if (n >= 0)              /* connect is already done */
24        write_get_cmd(fptr);        /* write() the GET command */
25 }
```

图15.17 启动非阻塞connect[nonblock/start_connect.c]

启动非阻塞connect

第14～22行 启动非阻塞connect，并将相应文件的标志置为F_CONNECTING。在读集合和写集合中都打开这个套接口描述字对应的位，因此select将等待任何一个表示连接已完成的条件成立。我们还根据需要更新maxfd的值。

处理连接的完成

第23～24行 如果connect返回成功，那么连接已经完成，write_get_cmd函数(下面给出)向服务器发出一个命令。

我们把connect所用的套接口设置为非阻塞，但永远不恢复它缺省的阻塞模式。这样做没有问题是因为只向该套接口写了少量的数据(下一个函数中的GET命令)，从而可以认为这个命令大大小于套接口的发送缓冲区。即使write因为非阻塞标志造成返回的数目小于要写的数目，writen函数也会对此进行处理。让这个套接口非阻塞对后面的read没有什

么影响，因为我们总是调用 select 等待它变成可读。

图 15.18 给出了函数 write_get_cmd，它向服务器发出一个 HTTP 的 GET 命令。

```
1 #include    "web.h"
2 void
3 write_get_cmd(struct file * fptr)
4 {
5     int     n;
6     char    line[MAXLINE];
7     n = snprintf(line, sizeof(line), GET_CMD, fptr->f_name);
8     Writen(fptr->f_fd, line, n);
9     printf("wrote %d bytes for %s\n", n, fptr->f_name);
10    fptr->f_flags = F_READING;          /* clears F_CONNECTING */
11    FD_SET(fptr->f_fd, &rset);          /* will read server's reply */
12    if (fptr->f_fd > maxfd)
13        maxfd = fptr->f_fd;
14 }
```

图 15.18　向服务器发出一个 HTTP 的 GET 命令[nonblock/write_get_cmd.c]

构造命令并发出

第 7～9 行　构造命令并写入套接口。

设置标志

第 10～23 行　设置相应文件的 F_READING 标志，它同时清除 F_CONNECTING 标志(如果设置了的话)。这在主循环中表示该描述字已经就绪可以输入了。这个描述字也在读集合中打开，并根据需要更新 maxfd。

现在回到图 15.19 中的 main 函数，继续在图 15.15 中剩下的内容。这是程序的主循环：只要还有文件要处理(nlefttoread 大于 0)，可能的话就启动另一个连接，然后在所有活动的描述字上 select，处理非阻塞连接的完成和到来的数据。

能开始另一个连接吗?

第 24～35 行　如果没有达到同时连接数的上限，而且有另外的连接要建立，就寻找一个还没有处理的文件(用 f_flags 为 0 来表示)，调用 start_connect 启动相应的连接。活动的连接数(nconn)增 1，剩下要建立的连接数(nlefttoconn)减 1。

select：等待事件发生

第 36～37 行　select 等待可读或可写条件。正在进行非阻塞 connect 的描述字在这两个集合中都打开，而已完成连接并正在等待来自服务器的数据的描述字则只在读集合中打开。

```
24    while (nlefttoread > 0) {
25        while (nconn < maxnconn && nlefttoconn > 0) {
26                /* find a file to read */
27            for (i = 0 ; i < nfiles; i++)
28                if (file[i].f_flags == 0)
29                    break;
30            if (i == nfiles)
31                err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
32            start_connect(&file[i]);
```

```
33              nconn++;
34              nlefttoconn--;
35          }
36          rs = rset;
37          ws = wset;
38          n = Select(maxfd+1, &rs, &ws, NULL, NULL);
39          for (i = 0; i < nfiles; i++) {
40              flags = file[i].f_flags;
41              if (flags == 0 || flags & F_DONE)
42                  continue;
43              fd = file[i].f_fd;
44              if (flags & F_CONNECTING &&
45                  (FD_ISSET(fd, &rs) || FD_ISSET(fd, &ws))) {
46                  n = sizeof(error);
47                  if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &error, &n) < 0 ||
48                      error != 0) {
49                      err_ret("nonblocking connect failed for %s",
50                              file[i].f_name);
51                  }
52                      /* connection established */
53                  printf("connection established for %s\n", file[i].f_name);
54                  FD_CLR(fd, &wset);          /* no more writeability test */
55                  write_get_cmd(&file[i]);/* write() the GET command */
56              } else if (flags & F_READING && FD_ISSET(fd, &rs)) {
57                  if ( (n = Read(fd, buf, sizeof(buf))) == 0) {
58                      printf("end-of-file on %s\n", file[i].f_name);
59                      Close(fd);
60                      file[i].f_flags = F_DONE;     /* clears F_READING */
61                      FD_CLR(fd, &rset);
62                      nconn--;
63                      nlefttoread--;
64                  } else {
65                      printf("read %d bytes from %s\n", n, file[i].f_name);
66                  }
67              }
68          }
69      }
70      exit(0);
71  }
```

图 15.19 main 函数的主循环[nonblock/web.c]

处理所有就绪的描述字

第 39～55 行 现在对 file 结构的数组中的每个元素进行处理，以判断哪个描述字需要处理。如果 F_CONNECT 标志被设置，而且该描述字在读或写集合中被置位，非阻塞 connect 已完成。如我们在图 15.11 中介绍的，调用 getsockopt 获取该套接口上待处理的错误。如果这个值为 0，连接已是成功完成。在这种情况下在写集合中关闭该描述字，调用 write_get_cmd 向服务器发送 HTTP 请求。

检查描述字是否有数据

第 56～67 行　如果 F_READING 标志被设置而且描述字已读就绪，则调用 read。如果连接被另一端关闭，我们就关闭连接，设置 F_DONE 标志，在读集合中关闭该描述字，活动连接的数目和要处理的连接总数都减 1。

在这个例子中有两点没有优化(以避免使它变得更复杂)。首先，在图 15.19 中处理完 select 返回的已就绪的描述字数后可以终止 for 循环。其次，可以尽可能地减少 maxfd 的值，省下 select 检查那些不再设置的描述字的时间。因为这段代码处理的描述字的数目在任何时候都小于 10，而不是成千上万，所以这些优化与附加的复杂性相比是否值得，令人怀疑。

同时连接的性能

同时建立多个连接的性能如何呢？图 15.20 给出了取一个 Web 服务器上的主页和该服务器上的 9 个图像文件所花的时间。这个服务器的 RTT 大约为 150ms。主页的大小为 4017 字节，9 个图像文件的平均大小为 1621 字节。TCP 的分节大小是 512 字节。为了便于比较，本图还包含了在 23.9 节中开发的这个程序的使用线程的版本的数据。

同时 连接数	时钟时间 (秒数)，非阻塞	时钟时间 (秒数)，线程
1	6.0	6.3
2	4.1	4.2
3	3.0	3.1
4	2.8	3.0
5	2.5	2.7
6	2.4	2.5
7	2.3	2.3
8	2.2	2.3
9	2.0	2.2

图 15.20　不同数目的同时连接所花的时间

大部分的性能提高是在同时三个连接时获得的(时间减半)，在四个或更多的连接时性能的增加很少。

> 我们提供这个使用同时连接的例子是因为它是一个使用非阻塞 I/O 的好例子，而且可以衡量对性能的影响。这也是一个流行的 Web 应用程序即 Netscape 浏览器使用的一个特性。网络上有拥塞时这种技术也有缺陷。TCPv1 的第 21 章介绍了 TCP 的慢启动和拥塞避免算法的细节。当客户向服务器建立多个连接时，各个连接之间在 TCP 层没有通信。也就是说，如果在一个连接上遇到一个分组丢失，连往同一个服务器的其他连接不会被通知，而且这些连接很可能马上遇到分组丢失，除非它们慢下来。这些连接会向一个已经拥塞的网络继续发送分组。这种技术还会增加服务器主机的负载。

15.6 非阻塞 accept

在第6章提到过当一个已完成的连接准备好被 accept 时，select 会把监听返回成可读。因此，如果用 select 等待外来的连接，应该不需要把监听套接口设置为非阻塞，因为如果 select 告诉我们连接已就绪，accept 就不应该阻塞。

不幸的是，这里存在一个时间问题[Gierth 1996]。为了观察这个问题，我们修改了 TCP 回射客户程序(图5.4)，在跟服务器建立连接后发送一个 RST。图15.21给出了这个新版本。

设置 SO_LINGER 套接口选项

第16～19行 一旦连接建立，我们设置 SO_LINGER 选项，把 l_onoff 标志置为1，l_linger 时间置为0。如7.5节中说明的，这会导致在关闭连接时在 TCP 套接口上发送一个 RST。我们随后关闭该套接口。

下一步，修改图6.21和6.22中的 TCP 服务器程序，在 select 返回监听套接口可读之后、调用 accept 之前暂停。在下面这段来自图6.22开头的代码中，前面有加号的两行是新加的。

```
        if(FD_ISSET(listenfd,&rset)) {      /* new client connection */
+           printf("listening socket readable\n");
+           sleep(5);
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd,(SA *) &cliaddr,&clien);
```

这里模拟的是一个繁忙的服务器，它不能做到在 select 一返回监听套接口可读时就调用 accpet。服务器的这种延迟一般情况下不会有问题(实际上这是为什么维护一个已完成连接队列的原因)，但是如果在连接建立后又有从客户来的 RST，就会出现问题。

在5.11节中我们注意到，在服务器调用 accept 之前客户如果放弃这个连接，源自 Berkeley 的实现不对服务器返回这个夭折的连接，而其他实现应返回 ECONNABORTED 错误，但一般返回 EPROTO 错误来代替。考虑一个源自 Berkeley 的实现。

```
1  #include    "unp.h"
2  int
3  main(int argc, char **argv)
4  {
5      int     sockfd;
6      struct linger ling;
7      struct sockaddr_in servaddr;
8      if (argc != 2)
9          err_quit("usage: tcpcli <IPaddress>");
10     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(SERV_PORT);
14     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
```

```
15      Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
16      ling.l_onoff = 1;          /* cause RST to be sent on close() */
17      ling.l_linger = 0;
18      Setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling));
19      Close(sockfd);
20      exit(0);
21  }
```

图 15.21　创建连接并发送 RST 的 TCP 回射客户程序[nonblock/tcpcli03.c]

- 客户如图 15.21 中那样建立连接然后放弃它。
- select 向服务器进程返回可读，但服务器要在一段时间后才能调用 accept。
- 在服务器从 select 返回和调用 accept 之间，收到从客户来的 RST。
- 这个已完成的连接从队列中删除，我们假设没有其他已完成的连接存在。
- 服务器调用 accept，但是由于没有已完成的连接，因而阻塞了。

服务器会一直阻塞在 accept 调用上，直到其他某个客户建立一个连接为止。但是在这期间，假定这个服务器程序与图 6.22 类似，服务器仍会阻塞在 accept 调用上，不处理任何其他已就绪的描述字。

> 这个问题和 6.8 节中介绍的拒绝服务攻击有几分相似，但存在这个缺陷的服务器在另一个客户建立连接时会打破这个阻塞的 accept。

解决这个问题的办法是：

1. 如果用 select 来获知何时有连接已就绪可以 accept 时，总是把监听套接口置为非阻塞，并且
2. 在后面的 accept 调用中忽略以下错误：EWOULDBLOCK（源自 Berkeley 的实现在客户放弃连接时出现的错误）、ECONNABORTED（Posix.1g 的实现在客户放弃连接时出现的错误）、EPROTO（SVR4 的实现在客户放弃连接时出现的错误）和 EINTR（如果信号被捕获）。

15.7　小　结

15.2 节中非阻塞读和写的例子是把 str_cli 回射客户程序改成在到服务器的 TCP 连接上使用非阻塞 I/O。select 一般和非阻塞 I/O 一起使用，以判断描述字何时可以读或写。虽然代码的修改比较复杂，但是这个版本的客户程序是我们列出的版本中最快的。在这之后我们展示了把这个客户程序用 fork 分成两部分会更简单，在图 23.2 的线程版本中使用了同样的技术。

非阻塞 connect 使我们在 TCP 的三路握手时可以做其他处理，而不是在 connect 上阻塞。不幸的是这些也是不可移植的，在不同的实现中表示连接成功完成或出错的方式也不一样。我们用非阻塞 connect 开发了一个新的客户程序，它和 Web 客户程序类似，同时打开多个连接以减少从服务器上取多个文件的时间。像这样发起多个连接可以减少时间，但考虑到避免 TCP 的拥塞，它是“网络不友好的”。

15.8 习 题

15.1 在对图15.9的讨论中我们提到父进程必须调用shutdown，而不是close。这是为什么？

15.2 图15.9中如果服务器进程过早终止，客户子进程收到文件结束符并终止，但子进程不通知父进程，会发生什么？

15.3 图15.9中如果父进程在子进程前意外死亡，然后子进程在套接口上读到文件结束符，会发生什么？

15.4 在图15.11中如果删掉以下两行会发生什么？

```
if (n == 0)
    goto done      /* connect complete immediately */
```

15.5 在15.3节中提到有可能在connect返回前在套接口上有数据到来。这是怎样发生的？

第 16 章　ioctl 操作

16.1　概　述

在传统上 ioctl 函数是用于那些普遍使用、但不适合归入其他类别的任何特性的系统接口。Posix 去掉了 ioctl，它通过创建特殊的其功能已被 Posix 标准化的包裹函数来代替 ioctl。举例来说，Unix 的终端接口一般用 iotcl 来访问，但 Posix.1 为终端创建了 12 个新的函数：tcgetattr 取终端属性，tcflush 刷新输入和输出，等等。与此类似，Posix.1g 替换了一个 ioctl 请求（以后简称 XXX ioctl）：新的 sockatmark 函数（21.3 节）代替了 SIOCATMARK ioctl。虽然如此，在网络编程方面一些依赖于具体实现的功能仍保留了不少 ioctl 请求：譬如获得接口信息、访问路由表和 ARP 高速缓存。

这一章对和网络编程有关的 ioctl 请求做一个概述，但其中有很多都依赖于具体的实现。另外，新的源自 Berkeley 的实现用 AF_ROUTE 域的套接口（路由套接口）来完成这些操作中的大部分。在第 17 章中会详细介绍路由套接口。

网络程序（一般是服务器程序）中 ioctl 常用于在程序启动时获得主机上所有接口的信息：接口的地址、接口是否支持广播、是否支持多播，等等。我们开发自己的函数返回这些信息，在本章中提供一个使用 ioctl 的实现，在 17 章中再提供一个使用路由套接口的实现。

16.2　ioctl 函数

这个函数影响由 fd 参数指向的打开的文件。

```
#include <unistd.h>
int ioctl(int fd,int request,... /* void *arg */);
                                    返回：成功返回 0，出错返回 -1
```

第三个参数总是一个指针，但指针的类型依赖于 request（请求）。

4.4BSD 把第二个参数定义为 unsigned long 而不是 int，但这不会有问题，因为头文件定义了用作这个参数的常值。

一些实现中把第三个参数指定为 void * 指针，取代了 ANSI C 中的省略号表示法。

包含 ioctl 的函数原型的头文件没有标准，因为 Posix 没有对此进行标准化。多数系统如上所示地在 <unistd.h> 中定义，但老的 BSD 系统在 <sys/ioctl.h> 中定义。

我们可以把和网络有关的请求分为 6 类。

- 套接口操作
- 文件操作
- 接口操作
- ARP 高速缓存操作
- 路由表操作
- 流系统(第 33 章)

回想图 7.15,不但有一些 ioctl 操作和 fcntl 操作功能重迭(譬如把套接口设置为非阻塞),而且也有一些操作可以用不止一种 ioctl 来完成(譬如设置套接口的进程组属主关系)。

图 16.1 列出了各种请求以及 arg 地址必须指向的数据类型。下面几节对这些请求作详细说明。

16.3 套接口操作

有三种 ioctl 请求是明确针对套接口的(TCPv2 第 551～553 页)。它们都要求 ioctl 的第三个参数是一个指向整数的指针。

SIOCATMARK 如果套接口的读指针当前在带外标志上,则通过第三个参数指向的整数返回一个非零值,否则返回零。在第 21 章中会对带外数据作详细介绍。Posix.1g 用 sockatmark 代替了这种请求,在 21.3 节中给出了这个新函数的一个实现。

类别	request(请求)	描述	数据类型
套接口	SIOCATMARK	在带外标志上吗?	int
	SIOCSPGRP	设置套接口的进程 ID 或进程组 ID	int
	SIOCGPGRP	获取套接口的进程 ID 或进程组 ID	int
文件	FIONBIO	设置/清除非阻塞标志	int
	FIOASYNC	设置/清除异步 I/O 标志	int
	FIONREAD	获取接收缓冲区中的字节数	int
	FIOSETOWN	设置文件的进程 ID 或进程组 ID	int
	FIOGETOWN	获取文件的进程 ID 或进程组 ID	int
接口	SIOCGIFCONF	获取所有接口的列表	struct ifconf
	SIOCSIFADDR	设置接口地址	struct ifreq
	SIOCGIFADDR	获取接口地址	struct ifreq
	SIOCSIFFLAGS	设置接口标志	struct ifreq
	SIOCGIFFLAGS	获取接口标志	struct ifreq
	SIOCSIFDSTADDR	设置点到点地址	struct ifreq
	SIOCGIFDSTADDR	获取点到点地址	struct ifreq
	SIOCGIFBRDADDR	获取广播地址	struct ifreq
	SIOCSIFBRDADDR	设置广播地址	struct ifreq
	SIOCGIFNETMASK	获取子网掩码	struct ifreq
	SIOCSIFNETMASK	设置子网掩码	struct ifreq

（续）

类别	请求	描述	数据类型
	SIOCGIFMETRIC SIOCSIFMETRIC SIOCxxx	获取接口的测度(metric) 设置接口的测度(metric) (有很多,依赖于实现)	struct ifreq struct ifreq
ARP	SIOCSARP SIOCGARP SIOCDARP	创建/修改 ARP 项 获取 ARP 项 删除 ARP 项	struct arpreq struct arpreq struct arpreq
路由	SIOCADDRT SIOCDELRT	增加路径 删除路径	struct rtentry struct rtentry
流	I_xxx	(参见 33.5 节)	

图 16.1　网络支持 ioctl 请求的总结

SIOCGPGRP　通过第三个参数指向的整数返回为接收来自这个套接口的 SIGIO 或 SIGURG 信号而设置的进程 ID 或进程组 ID。这和 fcntl 的 F_GETOWN 相同,并注意在图 7.15 中 Posix.1g 标准倾向使用 fcntl 方式。

SIOCSPGRP　用第三个参数指向的整数设置进程 ID 或进程组 ID 以接收这个套接口的 SIGIO 或 SIGURG 信号。这和 fcntl 的 F_SETOWN 相同,并注意在图 7.15 中 Posix.1g 标准倾向使用 fcntl 方式。

16.4　文件操作

下一组请求以 FIO 开始,除套接口外,它们对其他某些类型的文件可能也适用。在这里只讨论适用于套接口的请求(TCPv2 第 553 页)。下面的五种请求都要求 ioctl 的第三个参数指向一个整数。

FIONBIO　套接口的非阻塞标志会根据 ioctl 的第三个参数指向的值是否为零而清除或设置。这个请求和用 fcntl 的 F_SETFL 命令设置和清除 O_NONBLOCK 文件状态标志效果相同。

FIOASYNC　这个标志根据 ioctl 的第三个参数指向的值是否为零决定清除或接收套接口上的异步 I/O 信号(SIGIO)。这个标志和用 fcntl 的 F_SETFL 命令设置和清除 O_AYNC 文件状态标志效果相同。

FIONREAD　在 ioctl 的第三个参数指向的整数里返回套接口接收缓冲区中当前的字节数。这种功能在文件、管道和终端上都能用。13.7 节中对此曾有详细介绍。

FIOSETOWN　在套接口上等价于 SIOCSPGRP。

FIOGETOWN　在套接口上等价于 SIOCGPGRP。

16.5 接口配置

很多处理网络接口的程序的第一步是从内核获取系统中配置的所有接口。这是通过SIOCGIFCONF请求来实现的，它使用了ifconf结构，ifconf又用了ifreq结构，这两种结构在图16.2中给出。

在调用ioctl之前分配一个缓冲区和一个ifconf结构，然后初始化后者。在图16.3中展示了这样的一幅图像，其中假定缓冲区的大小为1024字节。ioctl的第三个参数指向ifconf结构。

如果假定内核返回两个ifreq结构，我们在ioctl返回时会得到图16.4所示的结果。阴影区域已被ioctl修改。缓冲区填入了两个结构，ifconf结构的ifc_len成员被更新以反映缓冲区中存放的信息数量。在这幅图中假设每个ifreq结构占用32个字节。

指向ifreq结构的指针也被用作图16.1中其余的接口类ioctl请求的参数，在16.7节中会对此进行介绍。注意每个ifreq结构包含一个联合，有好几个#define隐藏了这些域实际上是这个联合的成员这一事实。对每个成员的引用都使用这样定义的名字。要注意的是有些系统在ifr_ifru联合中增加了很多依赖于实现的成员。

```
struct ifconf {
    int   ifc_len;                  /* size of buffer, value-result */
    union {
        caddr_t ifcu_buf;           /* input from user -> kernel */
        struct  ifreq *ifcu_req;    /* return from kernel -> user */
    } ifc_ifcu;
};
#define ifc_buf ifc_ifcu.ifcu_buf     /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req     /* array of structures returned */
#define IFNAMSIZ    16
struct ifreq {
    char    ifr_name[IFNAMSIZ];         /* interface name, e.g., "le0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int   ifru_metric;
        caddr_t ifru_data;
    } ifr_ifru;
};
#define ifr_addr      ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr   ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags     ifr_ifru.ifru_flags     /* flags */
#define ifr_metric    ifr_ifru.ifru_metric    /* metric */
#define ifr_data      ifr_ifru.ifru_data      /* for use by interface */
```

图16.2 用于各种ioctl请求的ifconf和ifreq结构[<net/if.h>]

16.6 get_ifi_info 函数

因为有很多程序需要知道在系统中的所有接口，所以我们将开发一个名为 get_ifi_info 的函数，它返回一个结构的链表，每个结构对应一个当前状态为“up”的接口。在这一节中将用 SIOCGIFCONF ioctl 实现这个函数，在第 17 章中将开发一个使用路由套接口的版本。

BSD/OS 提供了一个具有类似功能的名为 getifaddrs 函数。

对 BSD/OS 2.1 的全部源码树的检索结果显示有 12 个程序执行 SIOCGIFCONF ioctl 以确定存在的接口。

我们首先在一个新的名为 unpifi.h 的头文件中定义 ifi_info 结构，如图 16.5 所示。

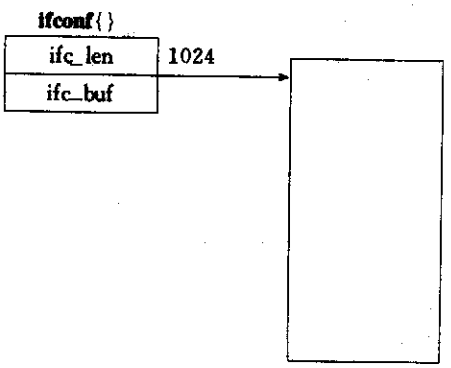


图 16.3 在 SIOCGIFCONF 前对 ifconf 结构的初始化

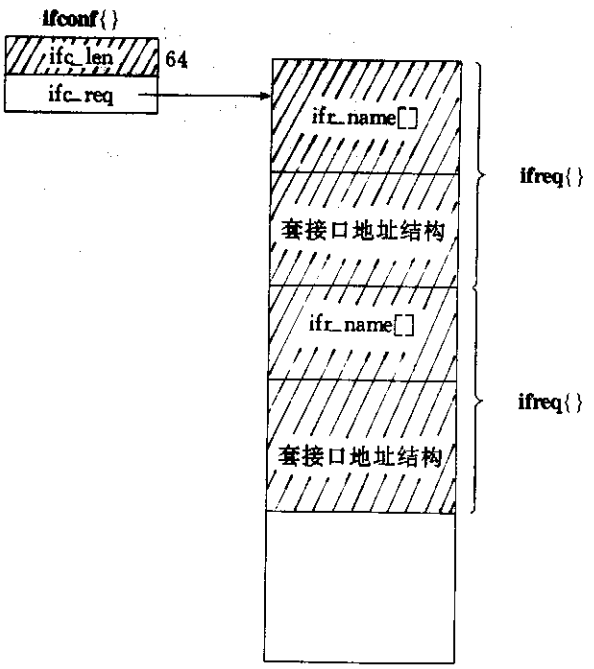


图 16.4 SIOCGIFCONF 返回的值

```
1 /* Our own header for the programs that need interface configuration info.
2  Include this file, instead of "unp.h". */

3 #ifndef  __unp_ifi_h
4 #define  __unp_ifi_h

5 #include       "unp.h"
6 #include       <net/if.h>

7 #define IFI_NAME       16      /* same as IFNAMSIZ in <net/if.h> */
8 #define IFI_HADDR       8      /* allow for 64-bit EUI-64 in future */

9 struct ifi_info {
10    char       ifi_name[IFI_NAME];    /* interface name, null terminated */
11    u_char     ifi_haddr[IFI_HADDR];  /* hardware address */
12    u_short    ifi_hlen;              /* #bytes in hardware address: 0, 6, 8 */
13    short      ifi_flags;             /* IFF_xxx constants from <net/if.h> */
14    short      ifi_myflags;           /* our own IFI_xxx flags */
15    struct sockaddr  *ifi_addr;       /* primary address */
16    struct sockaddr  *ifi_brdaddr;    /* broadcast address */
17    struct sockaddr  *ifi_dstaddr;    /* destination address */
18    struct ifi_info  *ifi_next;       /* next of these structures */
19 };
20 #define IFI_ALIAS      1            /* ifi_addr is an alias */

21                        /* function prototypes */
22 struct ifi_info  *get_ifi_info(int, int);
23 struct ifi_info  *Get_ifi_info(int, int);
24 void      free_ifi_info(struct ifi_info *);

25 #endif   /* __unp_ifi_h */
```

图 16.5 unpifi.h 头文件[ioctl/unpifi.h]

第 9～19 行 我们的函数返回一个这些结构的链表，其中每个结构的 ifi_next 成员指向下一个结构。在这个结构中只返回典型应用程序可能感兴趣的东西：接口的名字、硬件地址(譬如以太网地址)、接口的标志(让应用程序判断接口是否支持广播或多播或是一个点到点接口)、接口的地址、广播地址及点到点链路的目的地址。存放 ifi_info 结构和套接口地址结构的内存是动态申请的。因此我们还提供了一个 free_ifi_info 函数释放这部分内存。

> 今天的大部分硬件地址是 48 位的 MAC 地址(譬如以太网、令牌环网，等等)。但有向叫做 EUI-64[IEEE 1997b]的 64 位标识发展的趋势。IPv6 地址在低 64 位中包含一个 EUI-64 的值(A.5 节)，而且有一个简单的方法可以把 48 位的 MAC 地址封装到 64 位的 EUI 中。因此在 ifi_info 结构中分配了足够存放一个 64 位标识的空间，同时保存硬件地址的长度。

在给出 get_ifi_info 函数的实现之前，我们先来看一个简单的程序，该程序调用这个函数，然后输出所有信息。这个程序是 ifconfig 程序的一个简化版本，如图 16.6 所示。

```
1 #include    "unpifi.h"

2 int
3 main(int argc, char **argv)
4 {
5     struct ifi_info   *ifi, *ifihead;
```

```
 6    struct sockaddr  *sa;
 7    u_char   *ptr;
 8    int      i, family, doaliases;
 9    if (argc != 3)
10        err_quit("usage: prifinfo <inet4|inet6> <doaliases>");
11    if (strcmp(argv[1], "inet4") == 0)
12        family = AF_INET;
13 #ifdef      IPV6
14    else if (strcmp(argv[1], "inet6") == 0)
15        family = AF_INET6;
16 #endif
17    else
18        err_quit("invalid <address-family>");
19    doaliases = atoi(argv[2]);
20    for (ifihead = ifi = Get_ifi_info(family, doaliases);
21         ifi != NULL; ifi = ifi->ifi_next) {
22        printf("%s: <", ifi->ifi_name);
23        if (ifi->ifi_flags & IFF_UP)                printf("UP ");
24        if (ifi->ifi_flags & IFF_BROADCAST)         printf("BCAST ");
25        if (ifi->ifi_flags & IFF_MULTICAST)         printf("MCAST ");
26        if (ifi->ifi_flags & IFF_LOOPBACK)          printf("LOOP ");
27        if (ifi->ifi_flags & IFF_POINTOPOINT)       printf("P2P ");
28        printf(">\n");
29        if ( (i = ifi->ifi_hlen) > 0) {
30            ptr = ifi->ifi_haddr;
31            do {
32                printf("%s%x", (i == ifi->ifi_hlen) ? "  " : ":", *ptr++);
33            } while (--i > 0);
34            printf("\n");
35        }
36        if ( (sa = ifi->ifi_addr) != NULL)
37            printf("  IP addr: %s\n",
38                   Sock_ntop_host(sa, sizeof(*sa)));
39        if ( (sa = ifi->ifi_brdaddr) != NULL)
40            printf("  broadcast addr: %s\n",
41                   Sock_ntop_host(sa, sizeof(*sa)));
42        if ( (sa = ifi->ifi_dstaddr) != NULL)
43            printf("  destination addr: %s\n",
44                   Sock_ntop_host(sa, sizeof(*sa)));
45    }
46    free_ifi_info(ifihead);
47    exit(0);
48 }
```

图 16.6　调用 get_ifi_info 函数的 prifinfo 程序[ioctl/prifinfo.c]

第 20～45 行　这个程序是一个 for 循环，调用一次 get_ifi_info 后遍历返回的所有 ifi_info 结构。

第 22～35 行　输出所有的接口名和标志。如果硬件地址长度大于 0，就把它以十六进制数形式输出。（如果无法得到硬件地址，get_ifi_info 函数返回的 ifi_hlen 将为 0。）

第36～44行 如果返回了所需的三个IP地址，就输出它们。

如果在主机solaris(图1.16)上运行这个程序，会得到以下的输出：

```
solaris % prifinfo inet4 0
lo0: <UP MCAST LOOP>
  IP addr: 127.0.0.1
le0: <UP BCAST MCAST>
  IP addr: 206.62.226.33
  broadcast addr: 206.62.226.63
```

第一个命令行参数inet4指明要获取IPv4地址，第二个参数0指定不返回地址别名(在A.4节中介绍了IP地址别名)。注意在Solaris上以太网接口的硬件地址是得不到的。

如果给以太网接口(le0)加上三个别名地址，其主机ID分别为44、45和46，而且把第二个命令行参数改为1，会有以下结果：

```
solaris % prifinfo inet4 1
lo0:<UP MCAST LOOP>
  IP addr: 127.0.0.1
le0:<UP BCAST MCAST>
  IP addr: 206.62.226.33          主IP地址
  broadcast addr: 206.62.226.63
le0:1:<UP BCAST MCAST>
  IP addr: 206.62.226.44          第一个别名
  broadcast addr: 206.62.226.63
le0:2:<UP BCAST MCAST>
  IP addr: 206.62.226.45          第二个别名
  broadcast addr: 206.62.226.63
le0:3: <UP BCAST MCAST>
  IP addr: 206.62.226.46          第三个别名
  broadcast addr: 206.62.226.63
```

如果在BSD/OS下运行同样的程序，使用图17.16中get_ifi_info的实现(用它可以很容易地获得硬件地址)，会有以下结果：

```
bsdi % prifinfo inet4 1
we0: <UP BCAST MCAST>
  0:0:c0:6f:2d:40
  IP addr: 206.62.226.66
  broadcast addr: 206.62.226.95
ef0: <UP BCAST MCAST>
  0:20:af:9c:ee:95
  IP addr:206.62.226.35           主IP地址
  broadcast addr:206.62.226.63
ef0: <UP BCAST MCAST>
  0:20:af:9c:ee:95
  IP addr: 206.62.226.50          别名
  broadcast addr: 206.62.226.63
lo0: <UP MCAST LOOP>
  IP addr: 127.0.0.1
```

在这个例子中指示程序输出别名地址，可以看到第二个以太网接口定义了一个主机ID为50的别名。

这个输出依赖于接口的别名地址是如何建立的。在这个例子中别名地址是赋给以太网接口 ef0 的，在 BSD/OS 2.1 中这是通常的用法。但在 BSD/OS 3.0 中建议把别名地址赋给回馈接口 lo0。

下面给出用 SIOCGIFCONF ioctl 实现的 get_ifi_info。图 16.7 给出了函数的第一部分，它从内核获取接口配置。

```
1  #include    "unpifi.h"
2  struct ifi_info *
3  get_ifi_info(int family, int doaliases)
4  {
5      struct ifi_info *ifi, *ifihead, **ifipnext;
6      int          sockfd, len, lastlen, flags, myflags;
7      char         *ptr, *buf, lastname[IFNAMSIZ], *cptr;
8      struct ifconf ifc;
9      struct ifreq *ifr, ifrcopy;
10     struct sockaddr_in   *sinptr;
11     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
12     lastlen = 0;
13     len = 100 * sizeof(struct ifreq);   /* initial buffer size guess */
14     for ( ; ; ) {
15         buf = Malloc(len);
16         ifc.ifc_len = len;
17         ifc.ifc_buf = buf;
18         if (ioctl(sockfd, SIOCGIFCONF, &ifc) < 0) {
19             if (errno != EINVAL || lastlen != 0)
20                 err_sys("ioctl error");
21         } else {
22             if (ifc.ifc_len == lastlen)
23                 break;              /* success, len has not changed */
24             lastlen = ifc.ifc_len;
25         }
26         len += 10 * sizeof(struct ifreq);   /* increment */
27         free(buf);
28     }
29     ifihead = NULL;
30     ifipnext = &ifihead;
31     lastname[0] = 0;
```

图 16.7　发出 SIOCGIFCONF 请求以获取接口配置[lib/get_ifi_info.c]

创建一个网际套接口

第 11 行　创建一个用于 ioctl 的 UDP 套接口。TCP 或 UDP 套接口都可以(TCPv2 第 163 页)。

在循环中发出 SIOCGIFCONF 请求

第 12～28 行　SIOCGIFCONF 请求有一个严重的问题是，一些实现在缓冲区的大小不足以存放结果时不返回错误，而是截断结果并返回成功(ioctl 返回 0)。这意味着要知道缓冲区是否够大的唯一办法是，发出请求，记下返回的长度，用更大的缓冲区发出请求，再比较返

回的长度和前面记下的长度。只有在这两个长度相同的情况下，缓冲区才是足够大的。

> 源自 Berkeley 的实现在缓冲区太小的情况下不返回错误(TCPv2 第 118～119 页)；结果会被截断以放入缓冲区。与此相反，Solaris 2.5 在返回的长度将大于或等于缓冲区长度时返回 EINVAL。但是在返回的长度小于缓冲区大小的情况下我们也不能肯定成功，因为源自 Berkeley 的实现在剩下的空间装不下另一个结构时返回的长度会小于缓冲区长度。
>
> 一些实现中提供了一个 SIOCGIFNUM 请求以返回接口的数目。这可以让应用进程在发出 SIOCGIFCONF 请求前分配足够的缓冲区空间，但这个新的请求没有被广泛实现。
>
> 随着 Web 应用的增长，为 SIOCGIFCONF 请求返回的结果分配一个固定长度的缓冲区已经成为一个问题，因为大的 Web 服务器给单独一个接口就分配很多别名地址。举例来说，Solaris 2.5有每个接口最多 256 个别名地址的限制，但这个限制在 2.6 中增加到 8192。有大量别名地址的站点发现对接口信息使用固定大小缓冲区的程序开始不工作。尽管 Solaris 在缓冲区太小的情况下返回错误，这些程序还是分配它们固定大小的缓冲区，发出 ioctl，结果要是早先返回过错误的话就死掉。

第 12～15 行　分配一个缓冲区，开始时为 100 个 ifreq 结构的空间。把 lastlen 初始化成 0，并在 lastlen 中记录最近一个 SIOCGIFCONF 请求返回的长度。

第 19～20 行　如果 ioctl 返回一个 EINVAL 错误，而且还没有得到过一个成功的返回(也就是 lastlen 还是 0)，那么我们还没有分配一个足够大的缓冲区，继续进行循环。

第 22～23 行　如果 ioctl 返回成功，而且返回的长度等于 lastlen，那么这个长度没有改变(即缓冲区足够大)，于是 break 出这个循环，因为已经得到了所有信息。

第 26～27 行　每循环一次，把缓冲区的大小增加能多存放 10 个 ifreq 结构的空间。

初始化链表指针

第 29～31 行　因为将要返回一个指向 ifi_info 结构的链表头的指针，所以用两个变量 ifihead 和 ifipnext 在建立链表时保存指针。这和在图 11.34 中介绍的技术相同。

get_ifi_info 函数的下一部分，即主循环的开头，如图 16.8 所示。

```
32      for (ptr = buf; ptr < buf + ifc.ifc_len; ) {
33          ifr = (struct ifreq *) ptr;
34  #ifdef  HAVE_SOCKADDR_SA_LEN
35          len = max(sizeof(struct sockaddr), ifr->ifr_addr.sa_len);
36  #else
37          switch (ifr->ifr_addr.sa_family) {
38  #ifdef  IPV6
39          case AF_INET6:
40            len = sizeof(struct sockaddr_in6);
41            break;
42  #endif
43          case AF_INET:
44          default:
45              len = sizeof(struct sockaddr);
```

```
46                  break;
47          }
48 #endif    /* HAVE_SOCKADDR_SA_LEN */
49          ptr += sizeof(ifr->lfr_name) + len; /* for next one in buffer */
50          if (ifr->ifr_addr.sa_family != family)
51              continue;   /* ignore if not desired address family */
52          myflags = 0;
53          if ( (cptr = strchr(ifr->lfr_name, ':')) != NULL)
54              *cptr = 0;          /* replace colon will null */
55          if (strncmp(lastname, ifr->ifr_name, IFNAMSIZ) == 0) {
56              if (doaliases == 0)
57                  continue;   /* already processed this interface */
58              myflags = IFI_ALIAS;
59          }
60          memcpy(lastname, ifr->ifr_name, IFNAMSIZ);
61          ifrcopy = *ifr;
62          ioctl(sockfd, SIOCGIFFLAGS, &ifrcopy);
63          flags = ifrcopy.ifr_flags;
64          if((flags & IFF_UP)==0)
65              continue;           /* ignore if interface not up */
66          ifi = Calloc(1, sizeof(struct ifi_info));
67          *ifipnext = ifi;        /* prev points to this new one */
68          ifipnext = &ifi->ifi_next;      /* pointer to next one goes here */
69          ifi->ifi_flags = flags;         /* IFF_xxx values */
70          ifi->ifi_myflags = myflags;     /* IFF_xxx values */
71          memcpy(ifi->ifi_name, ifr->ifr_name, IFI_NAME);
72          ifi->ifi_name[IFI_NAME-1] = '\0';
```

图 16.8 处理接口配置[lib/get_ifi_info.c]

移到下一个套接口地址结构

第 32～49 行 在遍历所有 ifreq 结构时，ifr 指向每个结构，然后我们增加 ptr 指向下一个结构。但我们必须处理两种情况，即给套接口地址结构提供长度成员的新系统和不提供这个长度的老系统。虽然图 16.2 中声明的包含于 ifreq 结构中的套接口地址结构是一个通用套接口地址结构，但是在较新的系统中它可以是任何类型的套接口地址结构。实际上，在 4.4BSD 中每个接口还返回一个数据链路套接口地址结构(TCPv2 第 118 页)。因此如果支持长度成员，我们必须用这个值来将指针移到下一个套接口地址结构。否则使用基于地址族的长度，缺省为通用套接口地址结构的大小(16 字节)。

> 在支持 IPv6 的系统中，没有关于对 SIOCGIFCONF 请求是否返回 IPv6 地址的标准。我们给支持 IPv6 的新系统加了一个 case 语句，这是为了预防万一。问题在于 ifreq 结构中的联合把返回的地址定义成一个通用的 16 字节套接口地址结构，适合 16 字节的 IPv4 sockaddr_in 结构，但对 24 字节的 IPv6 sockaddr_in6 结构太小了。如果返回 IPv6 地址，将可能破坏现有的在每个 ifreq 结构中采用固定大小的套接口地址结构的代码。

第 50～51 行 忽略所有不是调用者期望的地址族的地址。

处理别名地址

第52～60行 我们必须检测在该接口上可能存在的任何别名地址，也就是赋给这个接口的其他地址。注意跟在图16.6之后的例子，在Solaris上别名地址的接口名中含有一个冒号，但在4.4BSD中接口名并不因别名地址而改变。为了处理这两种情况，我们把最近处理的接口名存到lastname，如果存在冒号的话，只比较到冒号。如果没有冒号，而且名字和最近处理的接口相同，我们也忽略这个接口。

取接口标志

第61～65行 发出一个SIOCGIFFLAGS ioctl请求(16.5节)取接口标志。ioctl的第三个参数是一个指向ifreq结构的指针，结构中包含要获取标志的接口名。在发出ioctl之前对ifreq结构做一个拷贝，因为如果不这样做的话，这个请求会覆盖接口的IP地址，因为它们都是图16.2中所示的同一个联合的成员。如果接口不在工作，就忽略掉。

分配和初始化ifi_info结构

第66～72行 到这时我们知道将向调用者返回这个接口。给ifi_info结构分配内存，并把它加到正在建立的链表的末尾。把接口的标志和名字复制到这个结构中。我们保证接口名是以空字符结尾的，而且因为calloc把分配的区域都初始化成0，我们从而知道ifi_hlen被初始化成0，ifi_next也被初始化成空指针。

图16.9是这个函数的最后一部分。

```
73          switch (ifr->ifr_addr.sa_family) {
74          case AF_INET:
75            sinptr = (struct sockaddr_in *) &ifr->ifr_addr;
76            if (ifi->ifi_addr == NULL) {
77                ifi->ifi_addr = Calloc(1, sizeof(struct sockaddr_in));
78                memcpy(ifi->ifi_addr, sinptr, sizeof(struct sockaddr_in));
79 #ifdef     SIOCGIFBRDADDR
80                    if (flags & IFF_BROADCAST) {
81                        ioctl(sockfd, SIOCGIFBRDADDR, &ifrcopy);
82                        sinptr = (struct sockaddr_in *) &ifrcopy.ifr_broadaddr;
83                        ifi->ifi_brdaddr = Calloc(1, sizeof(struct sockaddr_in));
84                        memcpy(ifi->ifi_brdaddr, sinptr, sizeof(struct sockaddr_in));
85                    }
86 #endif
87 #ifdef     SIOCGIFDSTADDR
88                    if (flags & IFF_POINTOPOINT) {
89                        ioctl(sockfd, SIOCGIFDSTADDR, &ifrcopy);
90                        sinptr = (struct sockaddr_in *) &ifrcopy.ifr_dstaddr;
91                        ifi->ifi_dstaddr = Calloc(1, sizeof(struct sockaddr_in));
92                          memcpy(ifi->ifi_dstaddr, sinptr, sizeof(struct sockaddr_
                            in));
93                    }
94 #endif
95                }
96                break;
97          default:
98              break;
```

```
 99          }
100      }
101      free(buf);
102      return(ifihead);    /* pointer to first structure in linked list */
103  }
```

图 16.9　取得并返回接口地址[ioctl/get_ifi_info.c]

第 73～78 行　把从最初的 SIOCGIFCONF 请求返回的 IP 地址复制到我们正在创建的结构中。

第 79～96 行　如果该接口支持广播，就用 SIOCGIFBRDADDR ioctl 取得广播地址。给包含这个地址的套接口地址结构分配内存，并把它加到正在创建的 ifi_info 结构中。与此类似，如果接口是一个点到点接口，SIOCGIFDSTADDR 返回链路另一端的 IP 地址。

> 因为前面已经提到过，现在还不知道 IPv6 的实现在 SIOCGIFCONF 请求时是否返回 IPv6 地址，所以这里没有讨论 AF_INET6 的情况。

图 16.10 给出了 free_ifi_info 函数，它以一个由 get_ifi_info 返回的指针做参数，翻放所有动态分配的内存。

```
104 void
105 free_ifi_info(struc ifi_info *ifihead)
106 {
107     struct ifi_info     *ifi, *ifinext;
108     for(ifi = ifihead; ifi != NULL; ifi = ifinext) {
109         if(ifi->ifi_addr != NULL)
110             free(ifi->ifi_addr);
111         if (ifi->ifi_brdaddr != NULL)
112             free(ifi->ifi_brdaddr);
113         if (ifi->ifi_dstaddr != NULL)
114             free(ifi->ifi_dstaddr);
115         ifinext = ifi->ifi_next;      /* can't fetch ifi_next after free() */
116         free(ifi);          /* the ifi_info{} itself */
117     }
118 }
```

图 16.10　free_ifi_info 函数：释放由 get_ifi_info 分配的内存[ioctl/get_ifi_info.c]

16.7　接口操作

如在前一节中给出的，SIOCGIFCONF 请求返回每个已配置接口的名字和套接口地址结构。另外有很多请求可以设置或获取接口的其他特性。这些请求的 get 版本(SIOCGxxx)一般由 netstat 程序使用，而 set 版本(SIOCSxxx)一般由 ifconfig 程序使用。任何用户都可以获取接口信息，但要想设置这些信息必须有超级用户权限。

这些请求以 ifreq 结构为参数，或返回一个 ifreq 结构，其地址由 ioctl 的第三个参数指定。接口总是用名字来标识：le0、lo0、ppp0 或其他在 ifr_name 成员中的任何内容。

这些请求中有许多使用套接口地址结构在应用进程中指定或返回 IP 地址或地址掩码。

对于IPv4,这个地址或掩码包含在网际套接口地址结构的sin_addr成员中。

SIOCGIFADDR	在ifr_addr成员中返回单播地址。
SIOCSIFADDR	用ifr_addr成员设置接口地址。这个接口的初始化函数也被调用
SIOCGIFFLAGS	在ifr_flags成员中返回接口标志。各种标志的名字为IFF_xxx,在<net/if.h>头文件中定义。举例来说,这些标志表示接口是否在工作(IFF_UP),接口是不是一个点到点接口(IFF_POINTOPOINT),接口是否支持广播(IFF_BROADCAST),等等。
SIOCSIFFLAGS	用ifr_flags成员设置接口标志。
SIOCGIFDSTADDR	在ifr_dstaddr成员中返回点到点地址。
SIOCSIFDSTADDR	用ifr_dstaddr成员设置点到点地址。
SIOCGIFBRDADDR	在ifr_broadaddr成员中返回广播地址。应用进程必须首先取到接口的标志,然后发出正确的请求:对广播接口用SIOCGIFBRDADDR,对点到点接口用SIOCGIFDSTADDR。
SIOCSIFBRDADDR	用ifr_broadaddr成员设置广播地址。
SIOCGIFNETMASK	在ifr_addr成员中返回子网掩码。
SIOCSIFNETMASK	用ifr_addr成员设置子网掩码。
SIOCGIFMETRIC	用ifr_metric成员返回接口的测度。每个接口的测度由内核维护,但使用它的是路由守护进程routed。接口的测度被加到跳数上(使这个接口更难被选中)。
SIOCSIFMETRIC	用ifr_metric成员设置接口的路由测度。

在本节中介绍了一般的接口请求。很多实现中都加入了其他的请求。

16.8 ARP高速缓存操作

ARP高速缓存也是由ioctl函数操作的。这些请求使用一个arpreq结构,在图16.11中给出了这个结构,它是在<net/if_arp.h>头文件中定义的。

```
struct arpreq {
    struct sockaddr arp_pa;          /* protocol address */
    struct sockaddr arp_ha;          /* hardware address */
    int             arp_flags;       /* flags */
};
#define ATF_INUSE       0x01    /* entry in use */
#define ATF_COM         0x02    /* completed entry (hardware addr valid) */
#define ATF_PERM        0x04    /* permanent entry */
#define ATF_PUBL        0x08    /* published entry (respond for other host) */
```

图16.11 用于ARP高速缓存的ioctl请求的arpreq结构[<net/if_arp.h>]

ioctl的第三个参数必须指向这些结构之一。它支持下面的三种请求:

SIOCSARP	把新项加到 ARP 高速缓存中或修改一个已有项。arp_pa 是一个网际套接口地址结构，其中包含 IP 地址，arp_ha 是一个通用套接口地址结构，其中 sa_family 为 AF_UNSPEC，sa_data 中是硬件地址（例如 6 字节的以太网地址）。ATF_PERM 和 ATF_PUBL 这两个标志可以由应用程序指定。另外两个标志，ATF_INUSE 和 ATF_COM 由内核设置。
SIOCDARP	从 ARP 高速缓存中删除一项。调用者指定要删除项的网际套接口地址。
SIOCGARP	从 ARP 高速缓存中取一项。调用者指定所取项的网际套接口地址，对应的以太网地址会和标志一起返回。

只有超级用户才能增加或删除数据项。这三种请求一般是由 arp 程序发出的。

在一些新系统中不支持这些和 ARP 有关的 ioctl 请求，它们使用路由套接口进行这些 ARP 操作。

注意 ioctl 没有办法列出 ARP 高速缓冲中的所有项。arp 命令的大部分版本，在有-a 标志时（列出 ARP 高速缓存中的所有项）读内核的内存（/dev/kmem）以获取当前 ARP 高速缓存中的内容。在 17.4 节中我们会看到用 sysctl 做到这一点的一种更简单（且更好）的方法。

例子：输出主机的硬件地址

现在用图 9.7 中的的 my_addrs 函数返回一台主机的全部 IP 地址，然后对每个 IP 地址发出 SIOCGARP ioctl 请求，得到并输出硬件地址。程序如图 16.12 所示。

获取地址列表并就每个地址进行循环

第 12～13 行　调用 my_addrs 获取主机的所有 IP 地址，然后就每个地址进行循环。

输出 IP 地址

第 14～17 行　用 inet_ntop 输出 IP 地址，接着根据 my_addrs 返回的地址族作不同处理。我们只处理 IPv4 地址，因为厂商可能将不支持 IPv6 地址的 SIOCGARP 请求。

发出 ioctl 请求，输出硬件地址

第 18～26 行　向 arp_pa 结构中填入一个包含 IPv4 地址的 IPv4 套接口地址结构。调用 ioctl 然后输出返回的硬件地址。

在我们的 solaris 主机上运行这个程序会有以下结果：

```
solaris % prmac
206.62.226.33: 8:0:20:78:e3:e3
```

```
 1 #include    "unp.h"
 2 #include    <net/if_arp.h>
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     family, sockfd;
 7     char    str[INET6_ADDRSTRLEN];
```

```
 8    char            * * pptr;
 9    unsigned char  * ptr;
10    struct arpreq arpreq;
11    struct sockaddr_in  * sin;
12    pptr = my_addrs(&family);
13    for ( ; * pptr ! = NULL; pptr++) {
14        printf("%s: ", inet_ntop(family, * pptr, str, sizeof(str)));
15        switch (family) {
16        case AF_INET:
17            sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
18            sin = (struct sockaddr_in  * ) &arpreq.arp_pa;
19            bzero(sin, sizeof(struct sockaddr_in));
20            sin->sin_family = AF_INET;
21            memcpy(&sin->sin_addr, * pptr, sizeof(struct in_addr));
22            ioctl(sockfd, SIOCGARP, &arpreq);
23            ptr = &arpreq.arp_ha.sa_data[0];
24            printf("%x:%x:%x:%x:%x:%x\n", * ptr, * (ptr+1),
25                   * (ptr+2), * (ptr+3), * (ptr+4), * (ptr+5));
26            break;
27        default:
28            err_quit("unsupported address family: %d", family);
29        }
30    }
31    exit(0);
32 }
```

图 16.12 输出一台主机的硬件地址[ioctl/prmac.c]

16.9 路由表操作

有两种 ioctl 请求用来操作路由表。这两个请求要求 ioctl 的第三个参数必须是一个指向 rtentry 结构的指针,这个结构在<net/route.h>头文件中定义。这些请求一般由 route 程序发出。只有超级用户才能发出这些请求。

SIOCADDRT　　向路由表中加一项。
SIOCDELRT　　从路由表中删去一项。

ioctl 没有办法列出路由表中的所有项。这种操作通常是带有-r 标志的 netstat 程序执行的。这个程序通过读内核的内存(/dev/kmem)获得路由表。和列出 ARP 高速缓存一样,在 17.4 节中我们会看到用 sysctl 做到这一点的一种更简单(且更好)的方法。

16.10 小 结

在网络编程中使用的 ioctl 命令可分为六类:

- 套接口操作(是否位于带外标志上?),
- 文件操作(设置或清除非阻塞标志),

- 接口操作(返回接口表,获取广播地址),
- ARP 表操作(创建,修改,获取,删除),
- 路由表操作(增加或删除),和
- 流系统(第 33 章)。

我们将使用这些套接口和文件操作,而获取接口表是一个常用操作,因此我们专为此开发了一个函数。在本书的后面会数次使用这个函数。只有几个特殊的专门的程序使用 ioctl 操作 ARP 高速缓冲和路由表。

16.11　习　题

16.1　在 16.7 节中我们说 SIOCGIFBRDADDR 请求是在 ifr_broadaddr 成员中返回广播地址。但在 TCPv2 第 173 页却说在 ifr_dstaddr 成员中返回。这有问题吗?

16.2　修改 get_ifi_info 程序,对一个 ifreq 结构发出第一次 SIOCGIFCONF 请求,然后每次循环,把缓冲区长度增加这些结构之一的大小。在循环中加一些语句,在每次发出请求时,不管 ioctl 是否返回错误都输出缓冲区的大小,在成功时再输出返回的缓冲区长度。运行 prifinfo 程序,看看你的系统在缓冲区太小时是怎样处理这个请求的。对所有返回的地址族不是想要的值的结构,也输出出它们的地址族,看看你的系统返回了哪些其他的结构。

16.3　修改 get_ifi_info 函数,返回接口的各个别名地址中与前一个不处于同一子网上的那些地址的信息。也就是说,16.6 节中的版本忽略从 206.62.226.44 到 206.62.226.46 的别名地址,因为它们和这个接口的主地址 206.62.226.33 在同一个子网。但是如果别名地址在不同的子网,譬如说 192.3.4.5,修改后的版本应该返回一个带有这个附加地址信息的 ifi_info 结构。

16.4　如果你的系统支持 SIOCGIFNUM ioctl,那么修改图 16.7 中的程序,发出这个请求,用返回的值作为最初猜测的缓冲区大小。

第17章 路由套接口

17.1 概 述

过去在内核中的Unix路由表是用ioctl命令来访问的。在16.9节中介绍了提供的两个命令:SIOCADDRT和SIOCDELRT,它们增加或删除一条路径。我们还提到没有命令能取到整个路由表。像netstat这样的程序通过读取内核得到路由表的内容。还有一点是路由守护进程如gated需要监视内核收到的ICMP重定向消息,它们通常创建一个原始ICMP套接口(第25章),在这个套接口上监听所有收到的ICMP消息。

4.3BSD Reno通过创建AF_ROUTE域去掉了这个内核路由子系统的接口。在路由域中支持的唯一一种套接口类型是原始套接口。在路由套接口中支持三种类型的操作。

1. 进程能通过写路由套接口向内核发消息。举例来说,路径就是这样增加和删除的。
2. 进程能在路由套接口上从内核读消息。这是核心通知进程已收到一个ICMP重定向消息并进行了处理的方式。
   一些操作包含这两步:举例来说,进程在路由套接口上向内核发出一个消息,请求在一个给定路径上的所有信息,这个进程又从路由套接口上读回内核的应答。
3. 进程可以用sysctl函数(17.4节)得到路由表或列出所有已配置的接口。

前两种操作需要超级用户权限,最后一种操作任何进程都可以执行。

> 从技术上说,第三种操作不是用路由套接口执行的,而是用一般的sysctl函数。但是sysctl所用的输入参数之一是地址族,它是本章将介绍的操作所用的AF_ROUTE,而且sysctl返回的信息也与内核在路由套接口上返回的信息格式相同。实际上,sysctl对AF_ROUTE族的处理是4.4BSD内核的路由套接口代码的一部分(TCPv2第632~643页)。
>
> sysctl是在4.4BSD中出现的。不幸的是并不是所有支持路由套接口的实现都提供sysctl。举例来说,AIX4.2、Digital Unix 4.0和Solaris 2.6都支持路由套接口,但没有一个支持sysctl。

17.2 数据链路套接口地址结构

在路由套接口上返回的一些消息中包含数据链路套接口地址结构。图17.1给出了这个结构的定义,它是在<net/if_dl.h>中定义的。

```
struct sockaddr_dl {
    uint8_t         sdl_len;
    sa_family_t     sdl_family;     /* AF_LINK */
    uint16_t        sdl_index;      /* system assigned index, if > 0 */
```

```
    uint8_t     sdl_type;       /* IFT_ETHER, ect. from <net/if_types.h> */
    uint8_t     sdl_nlen;       /* name length, starting in sdl_data[0] */
    uint8_t     sdl_alen;       /* link-layer address length */
    uint8_t     sdl_slen;       /* link-layer selector length */
    char        sdl_data[12];   /* minimum work area, can be larger; contains i/f
                                   name and link-layer address */
};
```

图 17.1　数据链路套接口地址结构

每个接口都有一个唯一的大于 0 的索引号，本章后面介绍的 if_nametoindex 和 if_nameindex 函数会返回这个索引号，第 19 章中的 IPv6 多播套接口选项也用到它。

sdl_data 成员包含名字和链路层地址(例如以太网接口的 48 位 MAC 地址)。名字从 sdl_data[0]开始，而且不以空字符终止。链路层地址从名字后面的 sdl_nlen 字节开始。这个头文件定义了下面这个宏以返回指向链路层地址的指针：

```
#define LLADDR(s)  ((caddr_t)((s)->sdl_data + (s)->sdl_nlen))
```

这些套接口地址结构是可变长度的(TCPv2 第 89 页)。如果链路层地址和名字超出 12 字节，这个结构将大于 20 字节。在 32 位系统上，这个结构的大小一般向上舍入到下一个 4 字节的倍数。在图 20.3 中还会看到，当用 IP_RECVIF 套接口选项返回这些结构之一时，全部三个长度成员都是 0，而且根本没有 sdl_data 成员。

17.3　读　和　写

进程在创建路由套接口后，可以通过写或读套接口向内核发命令或从核心读信息。有 12 个不同的路由命令，其中 5 个可以由进程发出。这些命令在<net/route.h>头文件中定义，如图 17.2 所示。

消息类型	去往内核?	来自内核?	描　述	结构类型
RTM_ADD	•	•	增加路径	rt_msghdr
RTM_CHANGE	•	•	修改网关、测试或标志	rt_msghdr
RTM_DELADDR		•	地址被从接口中删除	ifa_msghdr
RTM_DELETE	•	•	删除路径	rt_msghdr
RTM_GET	•	•	报告测试及其他路径信息	rt_msghdr
RTM_IFINFO		•	接口工作，不工作等等	if_msghdr
RTM_LOCK	•	•	锁住指定的测度	rt_msghdr
RTM_LOSING		•	内核怀疑路径将断开	rt_msghdr
RTM_MISS		•	地址查找失败	rt_msghdr
RTM_NEWADDR		•	地址被加到接口上	ifa_msghdr
RTM_REDIRECT		•	内核被告知使用另外的路径	rt_msghdr
RTM_RESOLVE		•	请求把目的地址解析成链路层地址	rt_msghdr

图 17.2　通过路由套接口交换的消息类型

通过路由套接口交换的结构有三种，如上图的最后一列所示：rt_msghdr、if_msghdr 和 ifa_msghdr，具体的定义如图 17.3 所示。

```
struct rt_msghdr {      /* from <net/route.h> */
```

```
    u_short     rtm_msglen;     /* to skip over non-understood messages */
    u_char      rtm_version;    /* future binary compatibility */
    u_char      rtm_type;       /* message type */
    u_short     rtm_index;      /* index for associated ifp */
    int         rtm_flags;      /* flags, incl. kern & message, e.g., DONE */
    int         rtm_addrs;      /* bitmask identifying sockaddrs in msg */
    pid_t       rtm_pid;        /* identify sender */

    int         rtm_seq;        /* for sender to identify action */
    int         rtm_errno;      /* why failed */
    int         rtm_inits;      /* from rtentry */
    u_long      rtm_inits;      /* which metrics we are initializing */
    struct rt_metrics rtm_rmx;  /* metrics themselves */
};

struct if_msghdr {          /* from <net/if.h> */
    u_short     ifm_msglen;     /* to skip over non-understood messages */
    u_char      ifm_version;    /* future binary compatibility */
    u_char      ifm_type;       /* message type */

    int         ifm_addrs;      /* like rtm_addrs */
    int         ifm_flags;      /* value of if_flags */
    u_short     ifm_index;      /* index for associated ifp */
    struct if_data ifm_data;    /* statistics and other data about if */
};

struct ifa_msghdr {     /* from <net/if.h> */
    u_short     ifam_msglen;    /* to skip over non-understood messages */
    u_char      ifam_version;   /* future binary compatibility */
    u_char      ifam_type;      /* message type */

    int         ifam_addrs;     /* like rtm_addrs */
    int         ifam_flags;     /* value of ifa_flags */
    u_short     ifam_index;     /* index for associated ifp */
    int         ifam_metric;    /* value of ifa_metric */
};
```

图 17.3 路由消息返回的三种结构

每个结构的前三个成员是相同的：长度、版本和消息类型。类型是图 17.2 的第一列中的常值之一。长度成员可以让应用程序跳过它不理解的消息类型。

rtm_addrs、ifm_addrs 和 ifam_addrs 成员是位掩码，用来指明消息后的套接口地址结构是八种可能的类型中的哪一种。图 17.4 给出了在<net/route.h>头文件中为该位掩码定义的各常值的值。

位掩码		数组下标		套接口地址结构包含
常值名	值	常值名	值	
RTA_DST	0x01	RTAX_DST	0	目的地址
RTA_GATEWAY	0x02	RTAX_GATEWAY	1	网关地址
RTA_NETMASK	0x04	RTAX_NETMASK	2	网络掩码
RTA_GENMASK	0x08	RTAX_GENMASK	3	复制掩码

（续）

位掩码		数组索引		套接口地址结构包含
常值名	值	常值名	值	
RTA_IFP	0x10	RTAX_IFP	4	接口名
RTA_IFA	0x20	RTAX_IFA	5	接口地址
RTA_AUTHOR	0x40	RTAX_AUTHOR	6	重定向的发起者
RTA_BRD	0x08	RTAX_BRD	7	广播或点到点目的地址
		RTAX_MAX	8	最大元素数目

图 17.4　用于在路由消息中指称套接口地址结构的常值

当有多个套接口地址结构时，它们总是按表中所示的顺序排列。

例子：获取并输出路由表项

现在举一个使用路由表的例子。这个程序的命令行参数是一个 IPv4 点分十进制数地址，它就这个地址向内核发送一个 RTM_GET 消息。内核发送在 IPv4 路由表中查找这个地址，返回一个带有其路由表项信息的 RTM_GET 消息。举例来说，如果在主机 bsdi 上执行下列代码：

```
bsdi # getrt 4.5.6.7
dest: 0.0.0.0
gateway: 206.62.226.62
netmask: 0.0.0.0
```

我们看到这个目的地址使用缺省路由（在路由表中的目的 IP 为 0.0.0.0，掩码为 0.0.0.0）。下一跳的路由器是 gw（图 1.16）。如果执行下列代码，指定主以太网为目的地址：

```
bsdi # getrt 206.62.226.32
dest: 206.62.226.32
gateway: AF_LINK, index=2
netmask: 255.255.255.224
```

目的地址就是网络本身。网关现在是发出数据的接口，返回的 sockaddr_dl 结构的接口索引为 2。

在给出源码之前，先在图 17.5 中展示向路由套接口中写入和由内核返回的数据。

我们建立一个缓冲区，其中包含 rt_msghdr 结构，接着是一个套接口地址结构，含有要内核查找的目的地址。rtm_type 为 RTM_GET，rtm_addrs 为 RTA_DST（图 17.4），表示在 rt_msghdr 结构后的唯一套接口地址结构中含有目的地址。这个命令可以用于任何协议族（提供路由表的协议族），因为要查找的地址的协议族包含在套接口地址结构中。

向内核发出消息后，读回应答，其格式如图 17.5 的右边所示：一个 rt_msghdr 结构后最多可以有 4 个套接口地址结构。返回这四个套接口地址结构中的哪此依赖于路由表项，我们是根据返回的 rt_msghdr 结构中的 rtm_addrs 成员的值知道的。每个套接口地址结构的地址族包含在 sa_family 成员中，如在前面的例子中看到的，前一次返回的网关是一个 IPv4 套接口地址结构，下面一次是一个数据链路套接口地址结构。

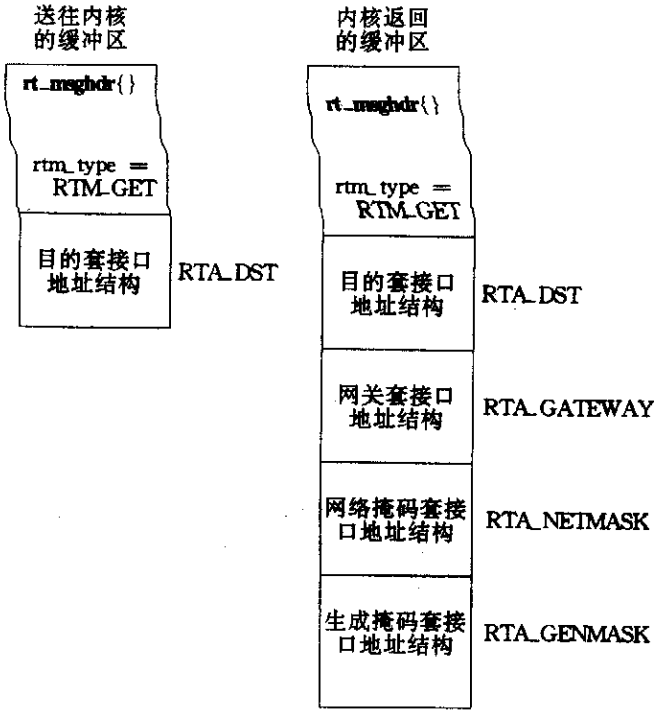


图 17.5 RTM GET 命令通过路由套接口与内核交换的数据

图 17.6 给出了这个程序的第一部分。

```
1 #include    "unproute.h"
2 #define BUFLEN     (sizeof(struct rt_msghdr) + 512)
3                          /* 8 * sizeof(struct sockaddr_in6) = 192 */
4 #define SEQ     9999
5 int
6 main(int argc, char **argv)
7 {
8     int           sockfd;
9     char          *buf;
10    pid_t         pid;
11    ssize_t       n;
12    struct rt_msghdr *rtm;
13    struct sockaddr *sa, *rti_info[RTAX_MAX];
14    struct sockaddr_in *sin;
15    if (argc != 2)
16        err_quit("usage: getrt <IPaddress>");
17    sockfd = Socket(AF_ROUTE,SOCK_RAW,0); /* need superuser privileges */
18    buf = Calloc(1, BUFLEN);      /* and initialized to 0 */
19    rtm = (struct rt_msghdr *) buf;
20    rtm->rtm_msglen = sizeof(struct rt_msghdr) + sizeof(struct sockaddr_in);
21    rtm->rtm_version = RTM_VERSION;
22    rtm->rtm_type = RTM_GET;
```

```
23      rtm->rtm_addrs = RTA_DST;
24      rtm->rtm_pid = pid = getpid();
25      rtm->rtm_seq = SEQ;

26      sin = (struct sockaddr_in *) (rtm + 1);
27      sin->sin_family = AF_INET;
28      inet_pton(AF_INET, argv[1], &sin->sin_addr);

29      Write(sockfd, rtm, rtm->rtm_msglen);

30      do {
31          n = Read(sockfd, rtm, BUFLEN);
32      } while (rtm->rtm_type != RTM_GET || rtm->rtm_seq != SEQ ||
33               rtm->rtm_pid != pid);
```

图 17.6　在路由套接口上发出 RTM_GET 命令的程序的第一部分[route/getrt.c]

第 1～3 行　unproute.h 头文件中包含一些需要的文件和 unp.h 文件。常值 BUFLEN 是我们分配用于存放发送给内核的消息和内核的应答的缓冲区的大小。这需要能存放一个 rt_msghdr 结构和可能会有的八个套接口地址结构(在一个路由套接口上可以返回的最大数目)。既然一个 IPv6 套接口地址结构的大小是 24 个字节,512 字节是足够了。

创建路由套接口

第 17 行　创建一个 AF_ROUTE 域的原始套接口,如前面提到的,这个请求需要超级用户权限。分配一个缓冲区并初始化成 0。

填写 rt_msghdr 结构

第 18～25 行　把我们的请求填写到这个结构。把进程号和我们选择的序列号存到这个结构中。我们将比较读到的应答中的这些值,找寻正确的应答。

用目的地址填写网际套接口地址结构

第 26～28 行　紧接着 rt_msghdr 结构,建立一个 sockaddr_in 结构,其中含有要内核在路由表中查找的目的 IPv4 地址。我们所设置的只是地址长度、地址族和地址本身。

向内核写消息并读取应答

第 29～32 行　向内核写消息并读取应答。因为其他进程可能打开了路由套接口,而内核会向所有路由套接口传送一个全部路由消息的拷贝,所以我们必须检查消息的类型、序列号和进程号以确保收到的消息是我们所等待的。

程序的后半部分在图 17.7 中给出。这一半处理应答。

```
34      rtm = (struct rt_msghdr *) buf;
35      sa = (struct sockaddr *) (rtm + 1);
36      get_rtaddrs(rtm->rtm_addrs, sa, rti_info);
37      if ( (sa = rti_info[RTAX_DST]) != NULL)
38          printf("dest: %s\n", Sock_ntop_host(sa, sa->sa_len));

39      if ( (sa = rti_info[RTAX_GATEWAY]) != NULL)
40          printf("gateway: %s\n", Sock_ntop_host(sa, sa->sa_len));

41      if ( (sa = rti_info[RTAX_NETMASK]) != NULL)
42          printf("netmask: %s\n", Sock_masktop(sa, sa->sa_len));

43      if ( (sa = rti_info[RTAX_GENMASK]) != NULL)
```

```
44          printf("genmask: %s\n", Sock_masktop(sa, sa->sa_len));
45      exit(0);
46  }
```

图 17.7 在路由套接口上发出 RTM_GET 命令的程序的后半部分[route/getrt.c]

第 34～35 行 rtm 指向 rt_msghdr 结构，sa 指向接在后面的第一个套接口地址结构。

第 36 行 rtm_addrs 是一个位掩码，其值对应于接在 rt_msghdr 结构后面的八种可能的套接口地址结构。get_rtaddrs 函数(下面会给出)以这个掩码和指向第一个套接口地址结构的指针(sa)为参数，在 rti_info 数组中填入指向对应的套接口地址结构的指针。假定图 17.5 中展示的全部四个套接口地址结构被内核返回，得到的 rti_info 数组如图 17.8 所示。

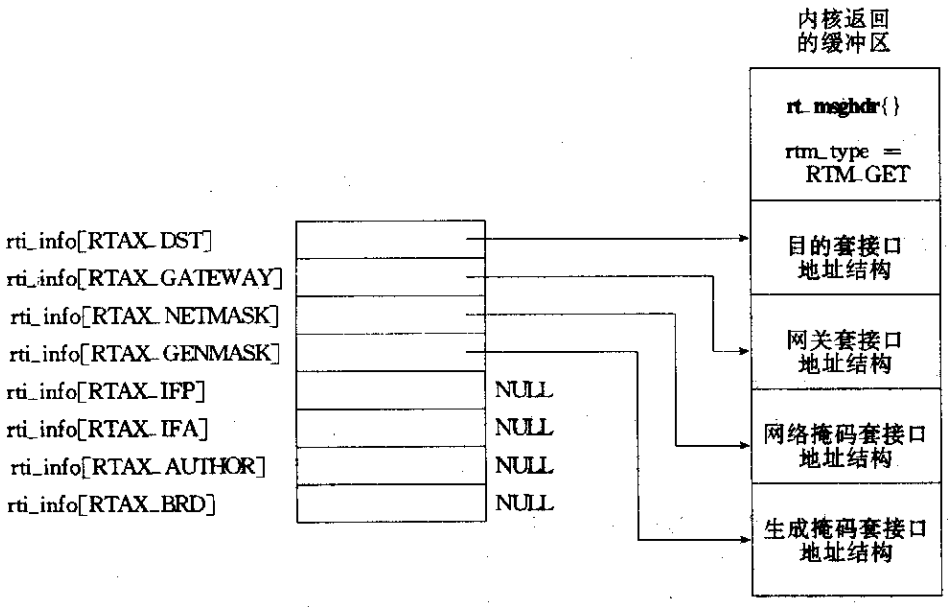


图 17.8 get_rtaddrs 函数填写的 rti_info 结构

然后检查 rti_info 数组，对数组中所有的非空指针做想做的处理。

第 37～44 行 四个可能的地址如果出现的话都被输出。调用 sock_ntop_host 函数输出目的地址和网关地址，调用 sock_masktop 输出两个掩码。我们马上给出这个新函数。

图 17.9 给出了在图 17.7 中调用的 get_rtaddrs 函数。

遍历八个可能的指针

第 17～23 行 图 17.4 中的 RTAX_MAX 为 8，它是内核在一个路由套接口上能返回的套接口地址结构的最大数目。函数里的这个循环查看图 17.4 中八个 RTA_xxx 位掩码常值中的每一个，而这些常值可能由图 17.3 中三种结构的 rtm_addrs、ifm_addrs 或 ifam_addrs 成员来设置。如果被置位，rti_info 数组中对应的元素被设置为指向相应套接口地址结构的指针，否则数组中对应的元素被设为空指针。

```
1 #include    "unproute.h"
2 /*
3  * Round up 'a' to next multiple of 'size'
4  */
5 #define ROUNDUP(a, size) (((a) & ((size)-1)) ? (1 + ((a) | ((size)-1))) : (a))
6 /*
7  * Step to next socket address structure;
8  * if sa_len is 0, assume it is sizeof(u_long).
9  */
10 #define NEXT_SA(ap)      ap = (SA *) \
11     ((caddr_t) ap + (ap->sa_len ? ROUNDUP(ap->sa_len, sizeof (u_long)) : \
12                                             sizeof(u_long)))
13 void
14 get_rtaddrs(int addrs, SA *sa, struct sockaddr **rti_info)
15 {
16     int         i;
17     for (i = 0; i < RTAX_MAX; i++) {
18         if (addrs & (1 << i)) {
19             rti_info[i] = sa;
20             NEXT_SA(sa);
21         } else
22             rti_info[i] = NULL;
23     }
24 }
```

图 17.9　在路由消息中构造指向套接口地址结构的指针数组[libroute/get_rtaddrs.c]

移到下一个套接口地址结构

第 2～12 行　套接口地址结构是变长的，但这段代码假定每个结构由一个 sa_len 成员指明它的长度。有两种复杂情况必须处理。首先，在套接口地址结构中返回的网络掩码和复制掩码的 sa_len 的值可以为 0，但实际上还是占用一个 unsigned long 的大小。(TCPv2 的第 19 章讨论了 4.4BSD 路由表的复制特性)。这个值表示一个全 0 的掩码，前面例子中展示的缺省路由的网络掩码值就是 0.0.0.0。其次，每个套接口地址结构在结尾处可能存在填充字节，使下一个结构在特定的边界上开始，在这个例子中是一个 unsigned long 的大小(譬如对 32 位体系结构就是 4 字节边界)。虽然 sockaddr_in 结构占 16 个字节，不需要填充，但是掩码的末尾经常有填充字节。

例子程序中的最后一个函数即 sock_masktop 在图 17.10 中给出，它返回能被返回的两个掩码值之一的表达字符串。掩码存在套接口地址结构中。sa_family 成员没有定义，但对 32 位的 IPv4 掩码有一个值为 0、5、6、7 或 8 的 sa_len。当这个长度大于 0 时，实际的掩码离开头的偏移和 IPv4 地址在 sockaddr_in 结构中离开头的偏移一样；离结构开头 4 个字节(如 TCPv2 第 577 页图 18.21 所示)，即通用套接口地址结构的 s_data[2]成员。

```
1 #include    "unproute.h"
2 char *
3 sock_masktop(SA *sa, socklen_t salen)
4 {
5     static char       str[INET6_ADDRSTRLEN];
6     unsigned char    *ptr = &sa->sa_data[2];
```

```
 7    if (sa->sa_len == 0)
 8        return("0.0.0.0");
 9    else if (sa->sa_len == 5)
10        snprintf(str, sizeof(str), "%d.0.0.0", *ptr);
11    else if (sa->sa_len == 6)
12        snprintf(str, sizeof(str), "%d.%d.0.0", *ptr, *(ptr+1));
13    else if (sa->sa_len == 7)
14        snprintf(str, sizeof(str), "%d.%d.%d.0", *ptr, *(ptr+1), *(ptr+2));
15    else if (sa->sa_len == 8)
16        snprintf(str, sizeof(str), "%d.%d.%d.%d",
17                 *ptr, *(ptr+1), *(ptr+2), *(ptr+3));
18    else
19        snprintf(str, sizeof(str), "(unknown mask, len = %d, family = %d)",
20                 sa->sa_len, sa->sa_family);
21    return(str);
22 }
```

图17.10 把一个掩码的值转换成它的表达格式[libroute/sock_masktop.c]

第7～21行 如果长度为0,隐含的掩码就是0.0.0.0。如果长度是5,就只存32位掩码的第一个字节,剩下的三个字节的值隐含为0。当长度为8时,掩码的四个字节都被存入。

在这个例子中我们想读内核的应答,因为应答中包含我们寻找的信息。但一般写路由套接口的write的返回值已告诉我们命令是否成功。如果这就是我们所需要的全部信息,就可以调用shutdown,其第二个参数为SHUT_RD,以防止内核发送应答。举例来说,如果删除一条路径,write返回0表示成功,返回ESRCH错误表示找不到这条路径(TCPv2第608页)。与此类似,write在增加一条路径时返回EEXIST错误表示这条路径已存在。在图17.6的例子中,如果没有所需路由表项(譬如主机上没有缺省路径),write会返回一个ESRCH错误。

17.4 sysctl操作

我们对路由套接口的主要兴趣是用sysctl函数检查路由表和接口表。创建路由套接口(一个AF_ROUTE域的原始套接口)需要超级用户权限,然而任何权限的进程都可以用sysctl检查路由表和接口列表。

```
#include <sys/param.h>
#include <sys/sysctl.h>

int sysctl(int *name, u_int namelen, void *oldp, size_t *oldlenp,
           void *newp, size_t newlen);
```

返回:成功为0,出错为-1

这个函数使用类似SNMP(简单网络管理协议)MIB(管理信息库)的名字。TCPv1的第25章详细讲述了SNMP和它的MIB。这些名字是分层结构的。

name参数是指定名字的一个整数数组,namelen是数组中的元素数目。数组的第一个元素指明请求被发往内核的哪个子系统。第二个参数指明这个子系统的某个部分,依此类

推。图 17.11 给出了前三级使用的一些常值的分层排列。

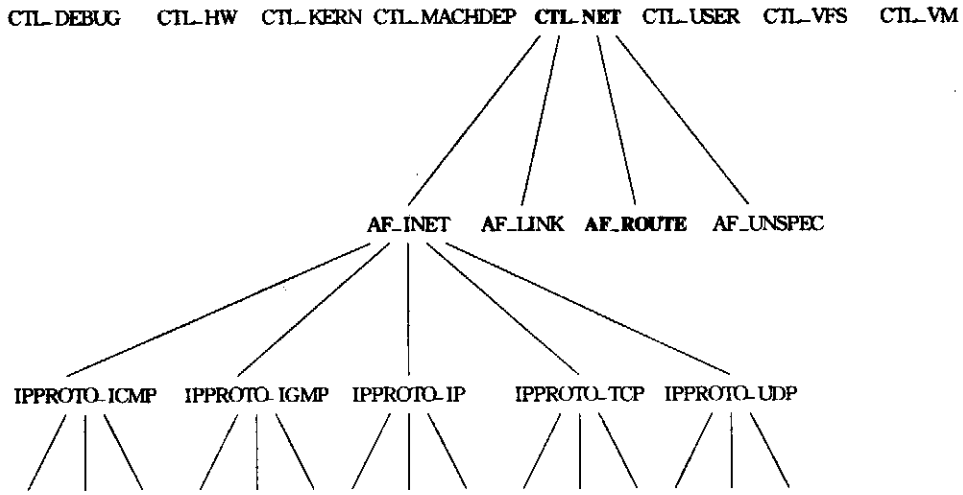


图 17.11　sysctl 名字的分层排列

要取一个值，oldp 需指向一个缓冲区，以让内核存放该值。oldlenp 是一个值-结果参数：调用函数时 oldlenp 指向的值是缓冲区的大小，返回的值是内核在缓冲区中返回的数据量。如果缓冲区不够大，就返回 ENOMEM 错误。作为一个特例，oldp 可以是一个空指针而 oldlenp 是一个非空指针，内核确定这个调用本应返回的数据量，并通过 oldlenp 返回这个值。

要设置一个新值，newp 需指向一个大小为 newlen 的缓冲区。如果没有指定新值，newp 应为一个空指针，newlen 应为 0。

sysctl 的手册页面中详细介绍了用这个函数能获取的各种系统信息：有关文件系统、虚存、内核的限制、硬件等等的信息。我们感兴趣的是网络子系统，通过把名字数组的第一个元素设为 CTL_NET 来指定。(CTL_xxx 常值在<sys/sysctl.h>头文件中定义。)第二个元素可以是：

- AF_INET：获取或设置影响网际协议的变量。下一级用一个 IPPROTO_xxx 常值指定协议。BSD/OS 3.0 在这一级提供了大约 30 个变量，用于对一些特性加以控制，如内核是否产生 ICMP 重定向消息，TCP 是否应使用 RFC 1323 选项，是否发送 UDP 校验和，等等。在本节的末尾将给出一个 sysctl 的这种用法的例子。
- AF_LINK：获取或设置链路层信息，譬如 PPP 接口的数目。
- AF_ROUTE：返回路由表或接口列表的信息。我们马上介绍这个信息。
- AF_UNSPEC：获取或设置一些套接口层的变量，譬如套接口的发送或接收缓冲区的最大容量。

当 name 数组的第二个元素为 AF_ROUTE 时，第三个元素(协议编号)总是为 0(因为 AF_ROUTE 族不像 AF_INET 族，AF_ROUTE 族里没有协议)，第四个元素是一个地址族，第五和第六级指明做什么。图 17.12 对此进行了总结。

name[]	返回 IPv4 路由表	返回 IPv4 ARP 高速缓存	返回接口列表
0	CTL_NET	CTL_NET	CTL_NET
1	AF_ROUTE	AF_ROUTE	AF_ROUTE
2	0	0	0
3	AF_INET	AF_INET	AF_INET
4	NET_RT_DUMP	NET_RT_FLAGS	NET_RT_IFLIST
5	0	RTF_LLINFO	0

图 17.12 sysctl 在 AF_ROUTE 域返回的信息

可见支持由 name[4]指明的三种操作。(NET_RT_xxx 常值在<sys/socket.h>头文件中定义。)这三种操作是通过 sysctl 调用中的 oldp 指针返回信息的。这个缓冲区中包含可变数目的 RTM_xxx 消息(图 17.2)。

1. NET_RT_DUMP 返回由 name[3]指定的地址族的路由表。如果这个地址族是 0,则返回所有地址族的路由表。

   路由表作为可变数目的 RTM_GET 消息返回,每个消息后最多可接四个套接口地址结构:本路由表项的目的地址、网关、网络掩码和复制掩码。在图 17.5 的右边展示了这些消息中的一个,图 17.7 中的代码分析这些消息。sysctl 操作的所有改动就是内核可以返回一个或多个这些信息。
2. NET_RT_FLAGS 返回由 name[3]指定的地址族的路由表,但只返回那些具有由 name[5]指定的 RTF_xxx 标志的路由表项。路由表中所有的 ARP 高速缓存项其 RTF_LLINFO 标志位都是置位的。

   信息返回的格式和前一条相同。
3. NET_RT_IFLIST 返回所有已配置接口的信息。如果 name[5]不为 0,那它就是一个接口的索引号,于是只返回这个接口的信息。(在 17.6 节中详细介绍了接口索引。)每个接口被赋予的所有地址也都返回,不过如果 name[3]不为 0 则只返回指定地址族的地址。

   给每个接口返回一个 RTM_IFINFO 消息,后面接着的是赋给该接口的每个地址对应的各个 RTM_NEWADDR 消息。RTM_IFINFO 消息后接的是一个数据链路套接口地址结构,每个 RTM_NEWADDR 消息后接最多三个套接口地址结构:接口地址、网络掩码和广播地址。在图 17.13 中展示了这两个消息。

   由于源自 4.4BSD 的内核加入了对 IPv6 的支持,因此应加入对 name[1]成员为 AF_INET6 的支持(以设置和获取 IPv6 特定变量),以及对图 17.12 中 name[3]为 AF_INET6 的支持(以列出 IPv6 路由表或 IPv6 邻居高速缓存,或者返回 IPv6 接口地址)。

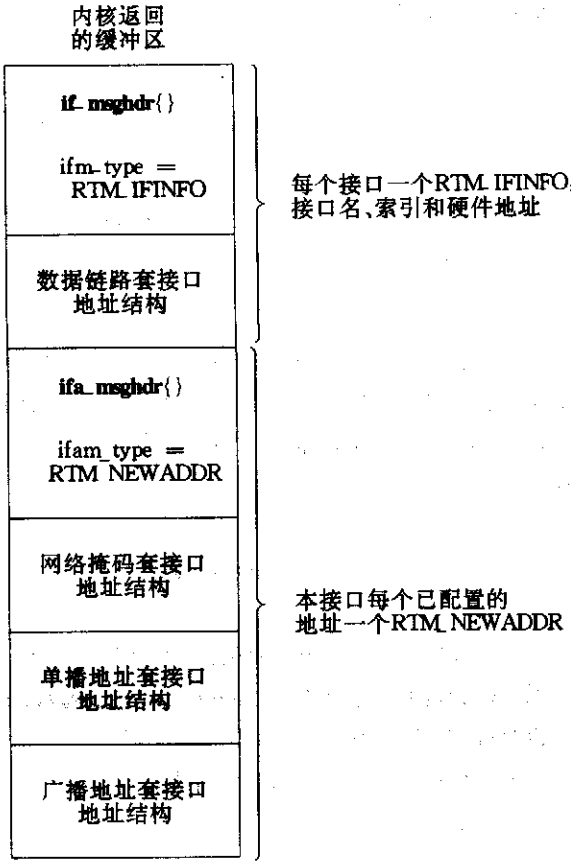


图 17.13　sysctl 的 CTL_NET,NET_RT_IFLIST 命令返回的信息

例子:判断 UDP 校验和是否打开

现在提供一个 sysctl 的简单例子,这个例子使用网际协议检查 UDP 校验和是否打开。一些 UDP 应用程序(譬如 BIND)在启动时检查 UDP 校验和是否打开,如果没有,则试图打开它。当然这需要超级用户权限来打开这样的特性,但我们现在只是检查这个特性是否打开。图 17.14 是这个程序。

```
1 #include    "unproute.h"
2 #include    <netinet/udp.h>
3 #include    <netinet/ip_var.h>
4 #include    <netinet/udp_var.h>        /* for UDPCTL_xxx constants */
5 int
6 main(int argc, char **argv)
7 {
8     int           mib[4], val;
9     size_t        len;
10    mib[0] = CTL_NET;
11    mib[1] = AF_INET;
12    mib[2] = IPPROTO_UDP;
13    mib[3] = UDPCTL_CHECKSUM;
14    len = sizeof(val);
```

```
15      Sysctl(mib, 4, &val, &len, NULL, 0);
16      printf("udp checksum flag: %d\n", val);
17      exit(0);
18  }
```

图 17.14 检查 UDP 校验和是否打开[route/checkudpsum.c]

包含头文件

第 2～4 行 我们必须包含<netinet/udp_var.h>头文件,以得到对 UDP sysctl 常值的定义。另外两个头文件是本头文件所需的。

调用 sysctl

第 10～16 行 分配一个四个元素的整数数组,存放图 17.11 中展示的各层对应的常值。因为只取变量而不是存入变量,所以把 sysctl 的 newp 参数设为空指针,newlen 参数设为 0。oldp 指向一个整数变量以存放结果,oldenp 指向的值-结果参数是这个整数的大小。输出的这个标志将为 0(关闭)或 1(打开)。

17.5 get_ifi_info 函数

现在回到 16.6 节中的例子:以一个 ifi_info 结构的链表返回所有工作的接口(图 16.5),prifinfo 程序保持原样(图 16.6),但现在给出一个 get_ifi_info 函数的新版本,用 sysctl 代替在图 16.7 中使用的 SIOCGIFCONF ioctl。

首先在图 17.15 中给出函数 net_rt_iflist。这个函数用 NET_RT_IFLIST 命令调用 sysctl,返回指定地址族的接口列表。

```
1  #include    "unproute.h"
2  char *
3  net_rt_iflist(int family, int flags, size_t *lenp)
4  {
5      int     mib[6];
6      char    *buf;
7      mib[0] = CTL_NET;
8      mib[1] = AF_ROUTE;
9      mib[2] = 0;
10     mib[3] = family;        /* only addresses of this family */
11     mib[4] = NET_RT_IFLIST;
12     mib[5] = flags;         /* interface index, or 0 */
13     if (sysctl(mib, 6, NULL, lenp, NULL, 0) < 0)
14         return(NULL);
15     if ( (buf = malloc(*lenp)) == NULL)
16         return(NULL);
17     if (sysctl(mib, 6, buf, lenp, NULL, 0) < 0) {
18         free(buf);
19     return(NULL);
20 }
21 return(buf);
22 }
```

图 17.15 调用 sysctl 返回接口列表[libroute/net_rt_iflist.c]

第 7～14 行　如图 17.12 所示，初始化数组 mib 以返回接口列表和指定族的所有已配置的地址。然后调用两次 sysctl。第一次调用时第三个参数为空，在 lenp 指向的变量中返回存放所有结构信息要用的缓冲区的大小。

第 15～21 行　给缓冲区分配空间并再次调用 sysctl，这次的第三个参数为非空。这次 lenp 指向的变量将返回存放在缓冲区中的信息量，这个变量是调用者分配的。指向这个缓冲区的指针也返回给调用者。

因为路由表的大小和接口的数目在两次 sysctl 调用之间可能改变，第一次调用返回的值在实际值上加了 10%的余量(TCPv2 第 639～640 页)。

图 17.16 给出了 get_ifi_info 函数的前半部分。

```
3 struct ifi_info *
4 get_ifi_info(int family, int doaliases)
5 {
6     int             flags;
7     char            *buf, *next, *lim;
8     size_t          len;
9     struct if_msghdr    *ifm;
10    struct ifa_msghdr   *ifam;
11    struct sockaddr *sa, *rti_info[RTAX_MAX];
12    struct sockaddr_dl  *sdl;
13    struct ifi_info *ifi, *ifisave, *ifihead, **ifipnext;
14    buf = Net_rt_iflist(family, 0, &len);
15    ifihead = NULL;
16    ifipnext = &ifihead;
17    lim = buf + len;
18    for (next = buf; next < lim; next += ifm->ifm_msglen) {
19        ifm = (struct if_msghdr *) next;
20        if (ifm->ifm_type == RTM_IFINFO) {
21            if ( ((flags = ifm->ifm_flags) & IFF_UP) == 0)
22                continue;   /* ignore if interface not up */
23            sa = (struct sockaddr *) (ifm + 1);
24            get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25            if ( (sa = rti_info[RTAX_IFP]) != NULL) {
26                ifi = Calloc(1, sizeof(struct ifi_info));
27                *ifipnext = ifi;                /* prev points to this new one */
28                ifipnext = &ifi->ifi_next;      /* ptr to next one goes here */
29                ifi->ifi_flags = flags;
30                if (sa->sa_family == AF_LINK) {
31                    sdl = (struct sockaddr_dl *) sa;
32                    if (sdl->sdl_nlen > 0)
33                        snprintf(ifi->ifi_name, IFI_NAME, "%*s",
34                                 sdl->sdl_nlen, &sdl->sdl_data[0]);
35                    else
36                        snprintf(ifi->ifi_name, IFI_NAME, "index %d",
37                                 sdl->sdl_index);
38                    if ( (ifi->ifi_hlen = sdl->sdl_alen) > 0)
```

```
39                          memcpy(ifi->ifi_haddr, LLADDR(sdl),
40                                 min(IFI_HADDR, sdl->sdl_alen));
41                  }
42              }
```

图 17.16 get_ifi_info 函数,前半部分[route/get_ifi_info.c]

第 6～14 行 声明局部变量,然后调用 net_rt_iflist 函数。

第 17～19 行 for 循环对 sysctl 返回的缓冲区中的每个路由消息进行处理。我们假定每个消息是一个 if_msghdr 结构,并查看其 ifm_type 成员。(全部三种结构的前三个成员是相同的,因此用这三种结构中的哪一个来查看类型都可以。)

检查接口是否工作

第 20～22 行 给每个接口返回一个 RTM_IFINFO 结构。如果接口不在工作则将其忽略。

判断出现的是那种套接口地址结构

第 23～24 行 sa 指向 if_msghdr 结构后的第一个套接口地址结构。get_rtaddrs 函数根据出现的是哪一种套接口地址结构来初始化 rti_info 数组。

处理接口名

第 25～42 行 如果该接口名有对应的套接口地址结构,我们就分配一个 ifi_info 结构存放接口标志。这个套接口地址结构所期望的地址族为 AF_LINK,表示它是一个数据链路 socket 地址结构。如果 sdl_nlen 成员不为 0,则把接口名拷贝到 ifi_info 结构中。否则把含有这个接口索引的字符串存为接口名。如果 sdl_alen 成员不为 0,则把硬件地址(譬如以太网地址)拷贝到 ifi_info 结构中,其长度在 ifi_hlen 中返回。

图 17.17 给出了 get_ifi_info 函数的第二部分,返回该接口的 IP 地址。

返回 IP 地址

第 43～63 行 sysctl 对该接口的每个地址返回一个 RTM_NEWADDR 消息:包括主地址和所有别名地址。如果已经给该接口填写了 IP 地址,那么我们是在进行对别名地址的处理。在这种情况下,如果调用者想要别名地址,就必须给另外一个 ifi_info 结构分配内存,拷贝已填写的字段,然后填入已返回的各个别名地址。

返回广播地址和目的地址

第 64～73 行 如果接口支持广播,则返回广播地址;如果接口是一个点对点接口,则返回目的地址。

```
43          } else if (ifm->ifm_type == RTM_NEWADDR) {
44              if (ifi->ifi_addr) {      /* already have an IP addr for i/f */
45                  if (doaliases == 0)
46                      continue;
47                      /* we have a new IP addr for existing interface */
48                  ifisave = ifi;
49                  ifi = Calloc(1, sizeof(struct ifi_info));
50                  *ifipnext = ifi;                /* prev points to this new one */
51                  ifipnext = &ifi->ifi_next;      /* ptr to next one goes here */
52                  ifi->ifi_flags = ifisave->ifi_flags;
53                  ifi->ifi_hlen = ifisave->ifi_hlen;
```

```
54                  memcpy(ifi->ifi_name, ifisave->ifi_name, IFI_NAME);
55                  memcpy(ifi->ifi_haddr, ifisave->ifi_haddr, IFI_HADDR);
56              }
57              ifam = (struct ifa_msghdr *) next;
58              sa = (struct sockaddr *) (ifam + 1);
59              get_rtaddrs(ifam->ifam_addrs, sa, rti_info);
60              if ( (sa = rti_info[RTAX_IFA]) != NULL) {
61                  ifi->ifi_addr = Calloc(1, sa->sa_len);
62                  memcpy(ifi->ifi_addr, sa, sa->sa_len);
63              }
64              if ((flags & IFF_BROADCAST) &&
65                  (sa = rti_info[RTAX_BRD]) != NULL) {
66                  ifi->ifi_brdaddr = Calloc(1, sa->sa_len);
67                  memcpy(ifi->ifi_brdaddr, sa, sa->sa_len);
68              }
69              if ((flags & IFF_POINTOPOINT) &&
70                  (sa = rti_info[RTAX_BRD]) != NULL) {
71                  ifi->ifi_dstaddr = Calloc(1, sa->sa_len);
72                  memcpy(ifi->ifi_dstaddr, sa, sa->sa_len);
73              }
74          } else
75              err_quit("unexpected message type %d", ifm->ifm_type);
76      }
77      /* "ifihead" points to the first structure in the linked list */
78      return(ifihead);    /* ptr to first structure in linked list */
79  }
```

图 17.17　get_ifi_info 函数，第二部分[route/get_ifi_info.c]

17.6　接口名和索引函数

RFC 2133[Gilligan et al. 1997]定义了四个处理接口名和索引的函数。这四个函数用于在第 19 章介绍的 IPv6 多播。基本概念是每个接口有一个唯一的名字和一个唯一的大于 0 的索引(0 从来不用做索引)。

```
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);
                                        返回:成功时为正的接口索引,出错时为 0

char *if_indextoname(unsigned int ifindex,char *ifname);
                                    返回:成功时为指向接口名的指针,出错时为 NULL

struct if_nameindex *if_nameindex(void);
                                        返回:成功时为非空指针,出错时为 NULL

void if_freenameindex(struct if_nameindex *ptr);
```

if_nametoindex 返回名为 ifname 的接口的索引。if_indextoname 对给定的 ifindex 返回一个指向其接口名的指针。ifname 参数指向一个大小为 IFNAMSIZ(在<net/if.h>头文件中定义;如图 16.2 所示)的缓冲区,调用者必须分配这个缓冲区以保存结果,成功时这个指

针也是函数的返回值。

if_nameindex 返回一个指向 if_nameindex 结构的数组的指针：

```
struct if_nameindex {
  unsigned int   if_index;  /* 1, 2... */
  char           *if_name;  /* null terminated name: "le0", ... */
};
```

数组的最后一项是一个 if_index 为 0，if_name 为空指针的结构。这个数组和数组中各元素指向的名字所用的内存是动态分配的，调用 if_freenameindex 可释放这些内存。

下面使用路由套接口提供这四个函数的一个实现。

if_nametoindex 函数

图 17.18 给出了 if_nametoindex 函数。

```
 1 #include    "unpifi.h"
 2 #include    "unproute.h"
 3 unsigned int
 4 if_nametoindex(const char *name)
 5 {
 6     unsigned int index;
 7     char        *buf, *next, *lim;
 8     size_t       len;
 9     struct if_msghdr   *ifm;
10     struct sockaddr *sa, *rti_info[RTAX_MAX];
11     struct sockaddr_dl *sdl;
12     if ( (buf = net_rt_iflist(0, 0, &len)) == NULL)
13         return(0);
14     lim = buf + len;
15     for (next = buf; next < lim; next += ifm->ifm_msglen) {
16         ifm = (struct if_msghdr *) next;
17         if (ifm->ifm_type == RTM_IFINFO) {
18             sa = (struct sockaddr *) (ifm + 1);
19             get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
20             if ( (sa = rti_info[RTAX_IFP]) != NULL) {
21                 if (sa->sa_family == AF_LINK) {
22                   sdl = (struct sockaddr_dl *) sa;
23                   if (strncmp(&sdl->sdl_data[0], name, sdl->sdl_nlen) == 0) {
24                         index = sdl->sdl_index; /* save before free() */
25                         free(buf);
26                         return(index);
27                     }
28                 }
29             }
30         }
31     }
32     free(buf);
33     return(0);          /* no match for name */
34 }
```

图 17.18 给定接口的名字返回其接口索引[libroute/if_nametoindex.c]

获取接口列表

第 12～13 行　net_rt_iflist 函数返回接口列表。

只处理 RTM_INFO 消息

第 17～30 行　处理缓冲区中的消息(图 17.13),只查找 RTM_IFINFO 消息。当找到一个时,调用 get_rtaddrs 函数设置指向各套接口地址结构的指针;如果存在一个接口名结构(rti_info 数组的 RTAX_IFP 元素),就把该接口名和调用者使用的参数做一下比较。

if_indextoname 函数

下一个函数:ifindextoname,如图 17.19 所示。

```
1 #include    "unpifi.h"
2 #include    "unproute.h"
3 char *
4 if_indextoname(unsigned int index, char *name)
5 {
6     char *buf, *next, *lim;
7     size_t len;
8     struct if_msghdr   *ifm;
9     struct sockaddr    *sa, *rti_info[RTAX_MAX];
10    struct sockaddr_dl *sdl;
11    if ( (buf = net_rt_iflist(0, index, &len)) == NULL)
12        return(NULL);
13    lim = buf + len;
14    for (next = buf; next < lim; next += ifm->ifm_msglen) {
15        ifm = (struct if_msghdr *) next;
16        if (ifm->ifm_type == RTM_IFINFO) {
17            sa = (struct sockaddr *) (ifm + 1);
18            get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
19            if ( (sa = rti_info[RTAX_IFP]) != NULL) {
20                if (sa->sa_family == AF_LINK) {
21                    sdl = (struct sockaddr_dl *) sa;
22                    if (sdl->sdl_index == index) {
23                        strncpy(name, sdl->sdl_data,sdl->sdl_nlen);
24                        name [sdl->sdl_nlen]=0 ; /* null terminate */
25                        free(buf);
26                        return(name);
27                    }
28                }
29            }
30        }
31    }
32    free(buf);
33    return(0);          /* no match for name */
34 }
```

图 17.19　给定接口的索引返回其接口名字[libroute/if_indextoname.c]

这个函数和前一个函数差不多一样,不过这里不是查找接口名,而是比较接口索引和调用者的参数。还有,其中调用的 net_rt_iflist 函数的第二个参数是期望的索引,因此结果应只含有期望接口的信息。如果找到匹配的接口,就返回以空字符终止的接口名。

if_nameindex 函数

下一个函数:if_nameindex,返回一个 if_nameindex 结构数组,它包含所有的接口名和索引。如图 17.20 所示。

```
1 #include    "unpifi.h"
2 #include    "unproute.h"
3 struct if_nameindex *
4 if_nameindex(void)
5 {
6     char        *buf, *next, *lim;
7     size_t      len;
8     struct if_msghdr *ifm;
9     struct sockaddr *sa, *rti_info[RTAX_MAX];
10    struct sockaddr_dl *sdl;
11    struct if_nameindex *result, *ifptr;
12    char    *namptr;
13    if ( (buf = net_rt_iflist(0, 0, &len)) == NULL)
14        return(NULL);
15    if ( (result = malloc(len)) == NULL)     /* overestimate */
16        return(NULL);
17    ifptr = result;
18    namptr = (char *) result + len;     /* names start at end of buffer */
19    lim = buf + len;
20    for (next = buf; next < lim; next += ifm->ifm_msglen) {
21        ifm = (struct if_msghdr *) next;
22        if (ifm->ifm_type == RTM_IFINFO) {
23            sa = (struct sockaddr *) (ifm + 1);
24            get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
25            if ( (sa = rti_info[RTAX_IFP]) != NULL) {
26                if (sa->sa_family == AF_LINK) {
27                    sdl = (struct sockaddr_dl *) sa;
28                    namptr += sdl->sdl_nlen + 1;
29                    strncpy(namptr, &sdl->sdl_data[0], sdl->sdl_nlen);
30                    namptr[sdl->sdl_nlen] = 0;     /* null terminate */
31                    ifptr->if_name = namptr;
32                    ifptr->if_index = sdl->sdl_index;
33                    ifptr++;
34                }
35            }
36        }
37    }
38    ifptr->if_name = NULL;     /* mark end of array of structs */
39    ifptr->if_index = 0;
40    free(buf);
41    return(result);            /* caller can free() this when done */
42 }
```

图 17.20 返回所有的接口名和索引[libroute/if_nameindex.c]

获取接口列表，给结果分配空间

第 13～18 行　调用 net_rt_iflist 函数返回接口列表。用返回的大小作为将分配的缓冲区的大小，以存放返回的 if_nameindex 结构数组。这是一个过高的估计，但要比遍历两遍接口表简单：一次是计算接口的数目和名字总共使用的空间大小，另一次是填写信息。从这个缓冲区的开头往前（正向）构建 if_nameindex 数组，从缓冲区末尾往后（反向）存放接口名。

只处理 RTM_IFINFO 消息

第 22～36 行　在所有消息中查找 RTM_INFO 消息以及紧接着的数据链路套接口地址结构。把接口名和索引存到正在构建的数组中。

终止数组

第 38～39 行　把数组的最后一项的 if_name 置为空，索引置为 0。

if_freenameindex 函数

最后一个函数如图 17.21 所示，它释放给 if_nameindex 结构数组和其中的名字分配的内存。

```
43 void
44 if_freenameindex(struct if_nameindex * ptr)
45 {
46     free(ptr);
47 }
```

图 17.21　释放由 if_nameindex 分配的内存[libroute/if_nameindex.c]

这个函数很简单，因为我们把结构数组和名字存放在同一个缓冲区中。如果对每个名字都调用 malloc，释放这些内存时就将不得不遍历整个数组，给每个名字释放内存，然后释放数组本身。

17.7　小　结

我们在本书中最后遇到的套接口地址结构是 sockaddr_dl 结构，它是一种可变长度的数据链路套接口地址结构。源自 Berkeley 的内核把它们和接口联系起来，在这些结构中返回接口索引、名字和硬件地址。

进程可以向路由套接口写五种类型的消息，内核可通过路由套接口异步返回 12 种不同消息。我们给出了一个例子：进程向内核请求一个路由表项的信息，内核返回了所有的细节信息。内核的这些应答最多含有八个套接口地址结构，我们必须对它们进行分析以得到信息的各个部分。

sysctl 函数是获取和设置操作系统参数的常规方法。我们对 sysctl 感兴趣的是：

- 输出接口列表
- 输出路由表
- 输出 ARP 高速缓存

IPv6需要对套接口API做的修改包括四个接口名和索引之间的映射函数。每个接口被赋予一个唯一的大于0的索引。源自Berkeley的实现已经把索引和每个接口联系起来了,因此可以很容易地用sysctl实现这些函数。

17.8 习 题

17.1 对于一个名为eth10、链路层地址是一个64位的IEEE EUI-64地址的接口,数据链路套接口地址结构的sdl_len成员的值是多少?

17.2 图17.6中在调用write前关闭SO_USELOOPBACK套接口选项。会发生什么?

第18章 广 播

18.1 概 述

在本章和下一章中，我们分别介绍广播和多播。迄今为止，正文中所有的示例都是处理单播的：一个进程只与另一个进程对话。虽然UDP支持各种形式的地址，但TCP只支持单播地址。图18.1对各种编址方式进行比较。

类型	IPv4?	IPv6?	TCP?	UDP?	标识的接口数	递送到的接口数
单播(unicast)	•	•	•	•	一个	一个
任播(anycast)		•	尚未实现	•	一组	一组中的一个
多播(multicast)	可选	•		•	一组	一组中的全部
广播(broadcast)	•			•	全部	全部

图18.1 不同的编址方式

图中，我们同时给出了任播(anycasting)方式，这是因为IPv6将支持它。但在今天，它还是一个尚未实现的概念。RFC1546[Partridge, Mendez, and Milliken 1993]对它作了详细的描述。

图18.1的要点是：

- IPv4对多播的支持是可选的，而IPv6则是必须的。
- IPv6没有提供对广播的支持：当使用广播的IPv4应用程序移植到IPv6时，必须使用IPv6的多播方式进行重新编码。
- 广播和多播要使用UDP；二者都不能使用TCP。

广播的用途之一是假定服务器主机在本地子网上，但不知道它的单播IP地址时，对它进行定位，这就是资源发现(resource discovery)。另一用途是当有多个客户和单个服务器通信时，减少局域网上数据流量。下面是几个以此为目的使用广播的因特网应用实例。

- ARP(地址解析协议，Address Resolution Protocol)。ARP是IPv4的一个基本组成部分，而不是一个用户应用程序。ARP在本地子网上广播一个请求："具有IP地址a.b.c.d的系统请表明自己，并告诉我，你的硬件地址。"
- BOOTP(引导协议，Bootstrap Protocol)。客户假定有一台服务器主机在本地子网上。它以广播地址(通常是255.255.255.255，因为这时客户还不知道自己的IP地址、子网掩码或子网的受限广播地址)为目的地址发出自己的引导请求。
- NTP(网络时间协议，Network Time Protocol)。一种常见的情形是：一个NTP客户主机可能配置成使用一个或多个服务器主机的IP地址，其上面的NTP客户于是以某个频率(每64秒一次或更长)轮询这些服务器。客户采用基于服务器返送的时刻和到达服务器的往返时间的精确算法更新时钟。但在支持广播的局域网上，就不需要采用客户轮询服务器的方法，而代之以服务器以每64秒一次的频率向本地子网

上的所有客户广播当前时刻。这样便可以减少网络上的数据流量。

- 路由后台进程。routed 是最常用的后台进程。它输出自己的路由表的方法便是局域网广播。所有其他连接到这些局域网上的路由器便可以同时接收这些路由通告,而不用每个路由器都必须配置其邻居路由器的 IP 地址。这个特性也被局域网上的主机用于侦听路由通告并相应更新它们的路由表(许多人认为这是一种"误用")。

我们必须注意到多播可以代替广播的这两种用途(资源发现和减少网络流量)。我们将在本章的后面和下一章介绍广播存在的问题。

> 虽然广播可以减少局域网上的数据流量,但却与无盘系统有我们不希望的交互问题。假设一个 NTP 服务器以 64 秒一次的频率广播当前时刻。如果所有无盘客户主机上的 NTP 后台进程在这段时间内被换出主存,那么当无盘客户主机每 64 秒一次收到 NTP 数据报时,不得不立刻从也在该局域网上的磁盘服务器将 NTP 后台进程读回主存。这样,随着无盘客户主机上 NTP 后台进程的换进换出,局域网上每 64 秒便会出现一次数据报涌流。幸运的是随着磁盘驱动器价格的下降,无盘系统将会逐步消失。

18.2 广播地址

如果用{netid,subnetid,hostid}({网络 ID,子网 ID,主机 ID})表示 IPv4 地址,那么有四种类型的广播地址。我们用-1 表示所有比特位均为 1 的字段。

1. 子网广播地址:{netid,subnetid,-1}。这类地址编排指定子网上的所有接口。例如,如果我们对 B 类地址 128.7 采用 8 位子网 ID,那么 128.7.6.255 将是 128.7.6 子网上所有接口的子网广播地址。

   路由器通常不转发这类广播(TCPv2 第 226～227 页)。图 18.2 给出了一个连到 128.7.1 和 128.7.6 两个子网的路由器。

   路由器在 128.7.1 子网上接收到一个目的地址为 127.7.6.255(另一个接口的子网广播地址)的单播 IP 数据报。路由器通常不向 128.7.6 子网转发这个数据报。有些系统具有允许转发子网广播数据报的配置选项(TCPv1 的附录 E)。
2. 全部子网广播地址:{netid,-1,-1}。这类广播地址编排指定网络上的所有子网。如果说这类地址曾被用过的话,那么现在已很少见了。
3. 网络广播地址:{netid,-1}。这类地址用于不进行子网划分的网络。但不进行子网划分的网络现在几乎不存在了。
4. 受限广播地址:{-1,-1,-1}或 255.255.255.255。路由器从不转发目的地址为 255.255.255.255 的 IP 数据报。

在这四类广播地址中,子网广播地址是今天最常见的。但有些老系统仍然发送目的地址为 255.255.255.255 的数据报。还有些老系统不理解子网广播地址,它们仅将发往 255.255.255.255 的数据报解释为广播。

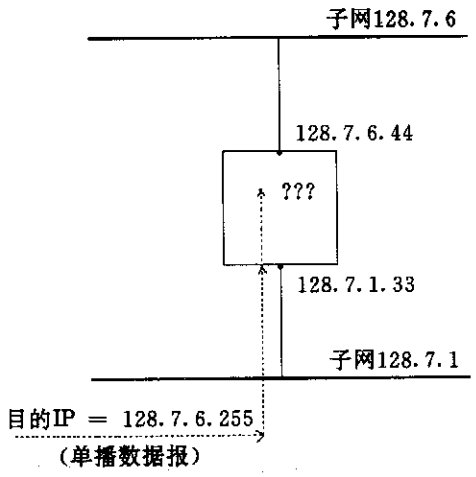


图 18.2 路由器转发子网广播吗

TFTP 和 BOOTP 等应用程序在启动过程中使用 255.255.255.255 作目的地址，因为这时它们还不知道自己的 IP 地址。

问题是：当应用进程向 255.255.255.255 发送 IP 数据报时，主机如何做？大多数主机允许发送(假定进程已经设置了 SO_BROADCAST 套接口选项)。它们在发送时，将目的地址转化为外出接口的子网广播地址。BSD/OS3.0 有一个新的套接口选项 IP_ONESBCAST。当该选项设置时，不管作为 sendto 的目标指定的广播地址是哪一类(子网广播或受限广播)，目的 IP 地址均被内核设置为 255.255.255.255。

另一个问题是：当应用进程向 255.255.255.255 发送 IP 数据报时，连接多网络的主机如何做？有些系统只在主接口(配置的第一个接口)发送目的地址为该接口的子网广播地址的单个广播数据报(TCPv2 第 736 页)。其他系统却在每个能进行广播的接口发送一个数据报拷贝。RFC1122[Braden 1989]的 3.3.6 节没有对此问题作出规定。然而，为了方便移植，如果应用进程需要在每个能进行广播的接口发送广播数据报，那它应首先获取接口的配置(16.6 节)，然后在每个能进行广播的接口上进行一次以接口的广播地址为目的地址的 sendto。

18.3 单播和广播的比较

在查看广播之前，我们应该已清楚向单播地址发送 UDP 数据报的步骤。图 18.3 给出了某个以太网上的三台主机。

图中以太网的地址为 128.7.6，其中 8 位用作子网 ID，8 位用作主机 ID。左边主机的应用程序在一个 UDP 套接口上调用 sendto 函数，将数据报发往 IP 地址 128.7.6.5、端口 7433。UDP 层附加一个 UDP 头部，并将 UDP 数据报传递到 IP 层。IP 层给它附加一个 IPv4 头部，并确定其外出接口。在以太网的情况下，将调用 ARP 来确定与目的 IP 地址相应的以太网地址：08:00:20:03:f6:42。然后，将分组作为以太网帧发送出去。以太网帧的目的地址

是上述48位地址。帧类型字段的值为0800，指示这是一个IPv4分组。IPv6帧类型字段的值为86dd。

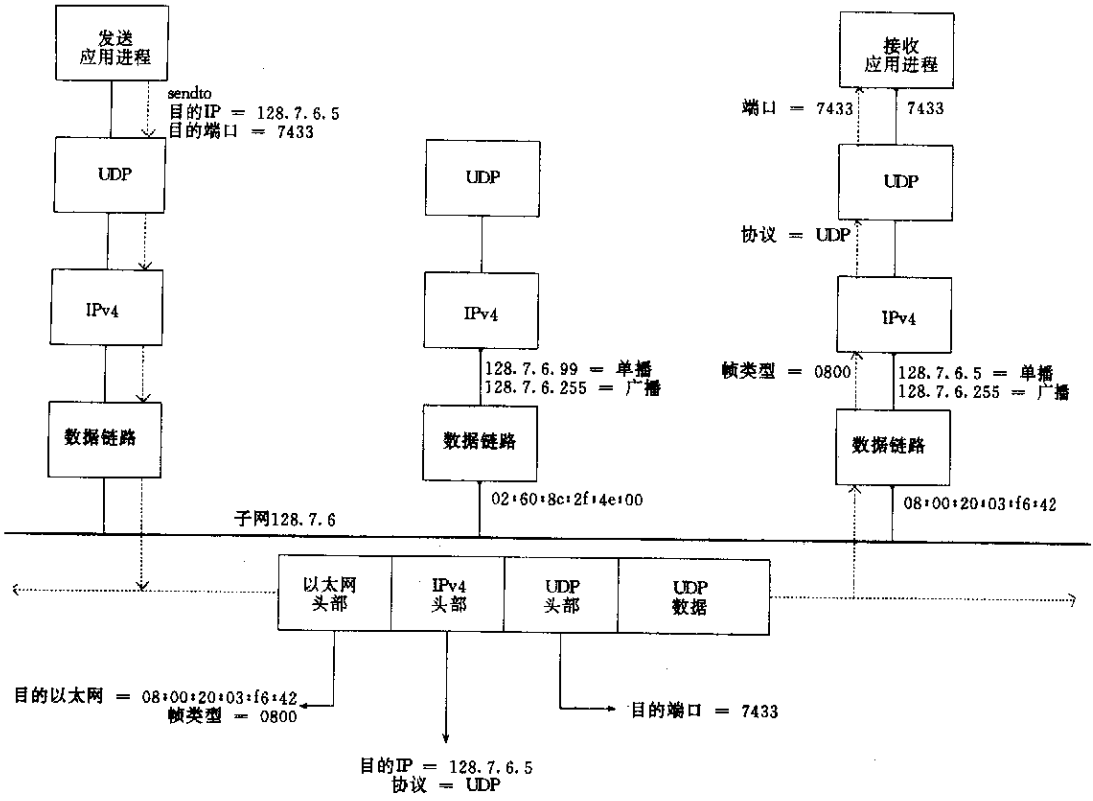


图18.3 UDP数据报单播示例

中间主机的以太网接口看到该帧，并将它的目的以太网地址与自己的以太网地址(02:60:8c:2f:4e:00)进行比较。由于二者不相等，接口便忽略该帧。因此，单播帧不会对这台主机造成任何额外开销。

右边主机的以太网接口也看到该帧。当它将该帧的目的以太网地址与自己的以太网地址进行比较时，发现二者相等，接口便读入整个帧。当帧全部读入后，可能产生一个硬件中断，设备驱动程序于是从接口内存中读取该帧。由于帧类型字段值为0800，于是将分组放入IP输入队列。

当IP层处理该分组时，它首先将目的IP地址与自己所有的IP地址进行比较。(回顾一下连接多网络的主机。也回顾一下我们在8.8节讨论的强端系统模型和弱端系统模型。)由于目的地址是其中之一，于是就接受这个分组。

然后，IP层检查IPv4头部协议字段，其值对于UDP为17。于是将IP数据报传送到UDP层。

UDP层检查其目的端口(如果其UDP套接口已连接，也可能检查源端口)，将数据报放到相应套接口的接收队列。如果需要，就唤醒进程，由进程读取这个新接收的数据报。

这个例子的关键点是单播IP数据报只能由目的IP地址指定的主机接收。子网上的其他主机不受任何影响。

我们现在考虑一个类似的例子：同样的子网，但发送进程发送的是子网广播数据报，其地址为：128.7.6.255。图 18.4 给出了这种情况。

当左侧的主机发送数据报时，它注意到目的 IP 地址是子网广播地址，于是便将它映射成 48 位的以太网地址：ff.ff.ff.ff.ff.ff。这使得子网上的每一个以太网接口都会接收该帧。在图中右侧两台运行 IPv4 的主机都接收该帧。由于以太网帧类型是 0800，两个主机都将数据报传递到 IP 层。由于目的 IP 地址匹配二者的广播地址，并且协议字段为 17(UDP)，两个主机于是都将分组上传至 UDP。

最右边的主机将 UDP 数据报传递给绑定端口 520 的应用进程。接收广播 UDP 数据报的应用进程不需要任何特殊的处理，它仅仅创建一个 UDP 套接口，并将应用的端口号捆绑到其上。(我们假设捆绑的 IP 地址为 INADDR_ANY，这是典型的情况。)

但是，中间的主机没有任何应用进程绑定 UDP 端口 520。于是主机的 UDP 代码丢弃这个已收到的数据报。这台主机禁止发送端口不可达的 ICMP 消息，因为这样做会产生广播风暴(broadcast storm)：子网上许多主机几乎同时产生响应，这将导致网络在几秒钟内不可用。

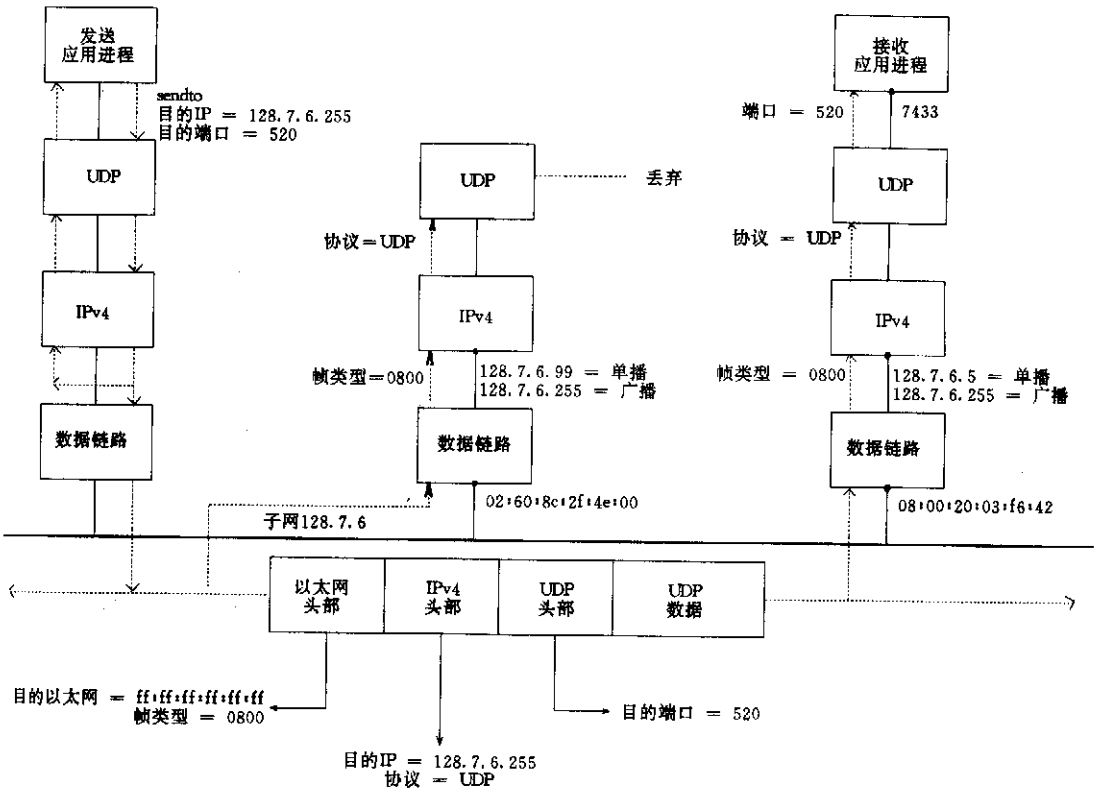


图 18.4　UDP 数据报广播示例

在图中，我们也表示出了左边主机将输出数据报又递送给自己的情况。这是一种广播属性：根据定义，广播要到达子网上的所有主机，包括发送者自身(TCPv2 第 109～110 页)。我们还假定发送应用进程已经绑定要发送到的端口(520)，因此它将收到它发送的每个数据报的拷贝。(但是一般情况下，没有将进程捆绑数据报所发送到的 UDP 端口的要求。)

在图中，我们给出了一个由IP层或数据链路层执行的逻辑回馈。这个IP层或数据链路层拷贝一份数据报(TCPv2第109～110页)，并将它沿协议栈向上发送。也可以使用物理回馈，但这在网络故障的情况下(例如未终结以太网)会引起问题。

本例也表明了广播存在的根本问题：子网上所有未参与广播应用系统的主机也必须完成对数据报的协议处理，直至在UDP层将它丢弃。(回顾一下我们对图8.21的讨论。)所有非IP主机(例如运行Novell IPX的主机)也必须在链路层接收完整的帧，并在该层将它丢弃(假定这些主机不支该帧的帧类型，IPv4分组的帧类型域值为0800)。因此以高速率产生IP数据报的应用系统(例如音频、视频应用系统)会严重影响子网上其他主机的运行。我们将在下一章看到多播是怎样解决这一问题的。

在图18.4中，我们选择UDP端口520是有意的。该端口由routed守护进程在交换RIP(路由信息协议)分组时使用。子网上所有使用RIP的路由器每30秒钟广播一次UDP数据报。如果子网上有200台主机，其中有两台使用RIP的路由器，那么其余198台主机也不得不每30秒就处理一次广播数据报(假定这198台主机都没有运行routed。)

18.4 使用广播的dg_cli函数

我们再次修改我们的dg_cli函数。这次，我们让它向标准UDP时间/日期服务器(图2.13)广播请求数据报，并且输出所有的应答。我们对main函数(图8.7)进行的仅有的修改是将目的端口改为13：

```
servaddr.sin_port = htons(13);
```

我们首先将修改后的main函数和未修改的dg_cli函数(图8.8)一起编译，然后在主机bsdi上运行。

```
bsdi % udpcli01 206.62.226.63
hi
sendto error: Permission denied
```

命令行参数是直接相连以太网的子网广播地址。我们键入一行信息后，程序调用sendto，结果返回EACCES错误。我们收到错误的原因是：除非我们显式地告诉内核我们将发送广播数据报，否则系统不允许我们进行广播。我们可以通过设置SO_BROADCAST套接口选项来做到这一点(7.5节)。

源自Berkeley的实现都要进行上述所谓的健全检查。相反，Solaris 2.5可以在没有指定该套接口选项的情况下接受目的地址为广播地址的数据报。Posix.1g的规定是内核"可以"返回错误。

对4.2BSD来说，广播是一种特权操作，它没有提供SO_BROADCAST选项。4.3 BSD增加了该选项，并且任何进程都允许设置它。

我们现在如图18.5所示修改dg_cli函数。该版本设置SO_BROADCAST选项，并输出在5秒钟内收到的所有应答。

给服务器地址分配空间，设置套接口选项

第 11～13 行　malloc 为由 recvfrom 返回的服务器地址分配空间。设置 SO_BROADCAST 套接口选项，并为 SIGALRM 信号安装信号处理程序。

从标准输入读取一行，发送至套接口，读取所有应答

第 14～24 行　以下两步即 fgets 和 sendto 与该函数的以前版本类似。但是由于我们发送的是一个广播数据报，因此我们将收到多个应答。我们用一个循环调用 recvfrom，并输出 5 秒钟内收到的所有应答。5 秒后，SIGALARM 信号产生，于是信号处理程序运行，导致 recvfrom 返回 EINTR 错误。

```
 1 #include    "unp.h"
 2 static void recvfrom_alarm(int);
 3 void
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int     n;
 7     const int on = 1;
 8     char    sendline[MAXLINE], recvline[MAXLINE + 1];
 9     socklen_t  len;
10     struct sockaddr *preply_addr;
11     preply_addr = Malloc(servlen);
12     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
13     Signal(SIGALRM, recvfrom_alarm);
14     while (Fgets(sendline, MAXLINE, fp) != NULL) {
15         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
16         alarm(5);
17         for ( ; ; ) {
18             len = servlen;
19             n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
20             if (n < 0) {
21                 if (errno == EINTR)
22                     break;      /* waited long enough for replies */
23                 else
24                     err_sys("recvfrom error");
25             } else {
26                 recvline[n] = 0;   /* null terminate */
27                 printf("from %s: %s",
28                         Sock_ntop_host(preply_addr,len), recvline);
29             }
30         }
31     }
32 }
33 static void
34 recvfrom_alarm(int signo)
35 {
36     return;         /* just interrupt the recvfrom() */
37 }
```

图 18.5　使用广播的 dg_cli 函数[bcast/dgclibcast1.c]

输出收到的每个应答

第25～29行 对于收到的每个应答，我们都调用sock_ntop_host函数。该函数在IPv4下将返回包括服务器的点分十进制数IP地址在内的字符串。这些信息将和服务器的应答一道输出。

如果我们指定用子网广播地址206.62.226.63运行该程序，我们将看到以下信息：

```
bsdi % udpcli01 206.62.226.63
hi
from 206.62.226.35: Sat Jun 14 12:19:36 1997
from 206.62.226.40: Sat Jun 14 12:19:36 1997
from 206.62.226.34: Sat Jun 14 12:19:36 1997
from 206.62.226.43: Sat Jun 14 12:19:36 1997
from 206.62.226.37: Sat Jun 14 12:19:36 1997
from 206.62.226.42: Sat Jun 14 12:19:36 1997
hello
from 206.62.226.35: Sat Jun 14 12:19:43 1997
from 206.62.226.40: Sat Jun 14 12:19:43 1997
from 206.62.226.34: Sat Jun 14 12:19:43 1997
from 206.62.226.43: Sat Jun 14 12:19:43 1997
from 206.62.226.42: Sat Jun 14 12:19:43 1997
from 206.62.226.37: Sat Jun 14 12:19:43 1997
```

为产生UDP数据报输出，我们每次都必须键入一行。我们每次都接收到6个应答，其中有一个来自发送主机本身。如我们以前所说的，广播的目的主机是包括发送主机本身在内的连接到同一网络上的所有主机。所有应答数据报都是单播的，因为作为其目的地址的请求数据报源地址是一个单播地址。

由于所有系统都运行NTP，所以它们都报告同样的时间。我们看到所在子网上(图1.16)9个节点中仅有6个作了响应。其余两台主机和一台路由器都没有进行响应，因为请求是送往广播地址的。

IP分片和广播

源自Berkeley的内核不允许广播数据报分片。如果发送到广播地址的IP数据报超过外出接口的MTU，就返回EMSGSIZE错误(TCPv2第233～234页)。这是一个自BSD4.2以来就存在的策略。事实上，并没有什么原因不让广播数据报分片，只是认为广播已经给网络带来很大负担，没有必要再由于分片而使这种负担扩大数倍罢了。

我们可以从图18.5的程序中看到这种情形。我们将标准输入重定向到一个包含单个2000字节长行的文件，这会导致在以太网上传输时的分片。

```
bsdi % udpcli01 206.62.226.63 < 2000line
sendto error: Message too long
```

AIX、BSD/OS、Digital Unix及UnixWare都实施了这种限制。UnixWare虽然不发送过大的数据报，但也不返回错误信息。Linux和Solaris都允许对发送到广播地址的数据报分片。然而，为了保证程序的可移植性，需要广播的应用程序应将广播数据报限制在1472字节以内，因为对局域网来说，以太网MTU通常是最小的。

18.5 竞争状态

多个进程访问共享数据，但正确结果依赖于进程的执行顺序，这种情况我们称之为竞争状态(race condition)。由于在典型的 Unix 系统中，进程的执行顺序是不确定的，因此，有时结果正确，有时不正确。最难调试的竞争状态类型是在通常情况下结果正确，偶尔才出现不正确结果的那些。当我们在第 23 章讨论互斥变量和条件变量时，还会进一步探讨竞争状态类型。竞争状态通常是线程编程中始终要注意的一个重要问题，因为在线程中有非常之多的数据需要共享(例如所有的全程变量)。

在进行信号处理时，通常会出现各种类型的竞争状态。这是因为在我们的程序执行过程中，内核随时都会递交信号。Posix.1 虽然允许我们阻塞信号递交，但在我们进行 I/O 操作时，它几乎没有任何作用。

考察这个问题的最简单方法是看看例子。图 18.5 存在着一个竞争状态；花几分钟时间分析一下，你能否将它找出来。(提示：信号递交时，我们正在哪里运行?)你可按下述方法迫使竞争状态显现出来：将 alarm 的参数从 5 改为 1，并且在 printf 前面增加 sleep(1)。

当我们作了这些修改并键入一行输入后，这一行被作为广播数据报发送出去，同时我们给 alarm 设置了 1 秒钟报警时间。此后我们便被阻塞在 recvfrom 调用中。第一个应答可能在几个毫秒内到达套接口。它由 recvfrom 返回后，我们睡眠 1 秒钟。其他应答也将陆续到达，并被依次放入套接口的接收缓冲区内。在我们睡眠的过程中，alarm 定时器到时，产生 SIGALRM 信号：信号处理程序于是运行。它只是返回，中断阻塞我们 sleep 调用。接着，我们循环读取缓冲区内排成队的应答。每一次我们都在暂停一秒钟后，再将应答输出。但在我们读完所有的应答后，我们再次阻塞在 recvfrom 调用中。这时，定时器已不再运行，因此我们将永远阻塞在 recvfrom 中。根本问题是：我们的意图是用信号处理程序中断已阻塞的 recvfrom，但由于信号可在任何时候递交，因此当信号递交时，我们可能在无限 for 循环内的任一点运行。

我们下面讨论四个解决这个问题的方法：一个错误方法，三个正确方法。

阻塞和解阻塞信号

第一个(不正确)方法是通过在执行 for 循环其他部分时阻塞信号的递交来减小出错窗口。图 18.6 给出了这个方法。

```
1 #include    "unp.h"
2 static void recvfrom_alarm(int);
3 void
4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
5 {
6     int         n;
7     const int on = 1;
8     char        sendline[MAXLINE], recvline[MAXLINE + 1];
9     sigset_t sigset_alrm;
10    socklen_t len;
11    struct sockaddr *preply_addr;
12    preply_addr = Malloc(servlen);
```

```
13      Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
14      Sigemptyset(&sigset_alrm);
15      Sigaddset(&sigset_alrm, SIGALRM);
16      Signal(SIGALRM, recvfrom_alarm);
17      while (Fgets(sendline, MAXLINE, fp) != NULL) {
18          Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
19          alarm(5);
20          for ( ; ; ) {
21              len = servlen;
22              Sigprocmask(SIG_UNBLOCK, &sigset_alrm, NULL);
23              n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
24              Sigprocmask(SIG_BLOCK, &sigset_alrm, NULL);
25              if (n < 0) {
26                  if (errno == EINTR)
27                      break;      /* waited long enough for replies */
28                  else
29                      err_sys("recvfrom error");
30              } else {
31                  recvline[n] = 0;  /* null terminate */
32                  printf("from %s: %s",
33                          Sock_ntop_host(preply_addr, len), recvline);
34              }
35          }
36      }
37  }
38  static void
39  recvfrom_alarm(int signo)
40  {
41      return;         /* just interrupt the recvfrom() */
42  }
```

图 18.6 在 for 循环内执行时阻塞信号(不正确方法)[bcast/dgclibcast3.c]

声明信号集并初始化

第 14～15 行 声明一个信号集,并将它初始化为空集(sigemptyset),接着将与 SIGALRM 相应的位置位(sigaddset)。

解阻塞信号和阻塞信号

第 21～24 行 在调用 recvfrom 前,我们解阻塞 SIGALRM 信号(以便我们被阻塞在该调用时内核仍可以递交该信号)。接着当 recvfrom 返回时,马上阻塞该信号。如果在信号阻塞期间该信号产生(即定时器到时),内核将记下这个事件,但在信号解阻塞前不能递交信号(即调用信号处理程序)。这便是信号产生和信号递交两者的根本区别。APUE 第 10 章提供了 Posix.1 信号处理方面的详细资料。

如果编译运行该程序,它看起来工作正常,但大部分具有竞争状态的程序在大部分时间内工作正常!该程序仍存在一个问题:解阻塞信号、调用 recvfrom 和阻塞信号都是互相独立的系统调用。假定 recvfrom 返回了最后一个应答数据报并且 SIGALRM 信号在 recvfrom 和阻塞信号之间递交,下一次对 recvfrom 的调用将永远阻塞。我们虽然已缩小了出错窗口,但问题仍然存在。

这种方案的一个变体是在信号递交时设置一个全局标志。

```
static void
recvfrom_alarm(int signo)
{
    had_alarm = 1;
    return;
}
```

每次调用 alarm 时，将标志初始化为 0。dg_cli 函数在调用 recvfrom 前检查这个标志，如果标志非 0，就不进行调用。

```
for (; ;) {
    len = servlen;
    Sigprocmask(SIG_UNBLOCK,&sigset_alrm,NULL);
    if(had_alarm = 1)
        break;
    n = recvfrom(sockfd,recvline,MAXLINE,0,preply_addr,&(len);
```

如果 SIGALRM 信号是在它被阻塞期间(在前一次 recvfrom 返回后)或者它在这段代码内被解阻塞时产生的，它将在 sigprocmask 返回前递交，并设置标志。但在标志测试和 recvfrom 调用之间仍然存在一个 SIGALRM 信号可能产生并递交的时间窗，如果真地如此发生了，recvfrom 调用将永远阻塞(当然假定没有收到另外的应答)。

用 pselect 阻塞和解阻塞信号

一个正确的方法是使用 pselect(6.9 节)，如图 18.7 所示。

```
 1 #include    "unp.h"
 2 static void recvfrom_alarm(int);
 3 void
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int         n;
 7     const int on = 1;
 8     char        sendline[MAXLINE], recvline[MAXLINE + 1];
 9     fd_set      rset;
10     sigset_t sigset_alrm, sigset_empty;
11     socklen_t len;
12     struct sockaddr *preply_addr;
13     preply_addr = Malloc(servlen);
14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
15     FD_ZERO(&rset);
16     Sigemptyset(&sigset_empty);
17     Sigemptyset(&sigset_alrm);
18     Sigaddset(&sigset_alrm, SIGALRM);
19     Signal(SIGALRM, recvfrom_alarm);
20     while (Fgets(sendline, MAXLINE, fp) ! = NULL) {
21         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
22         Sigprocmask(SIG_BLOCK, &sigset_alrm, NULL);
23         alarm(5);
24         for ( ; ; ) {
25             FD_SET(sockfd, &rset);
26             n = pselect(sockfd+1, &rset, NULL, NULL, NULL, &sigset_empty);
27             if (n < 0) {
```

```
28                  if (errno == EINTR)
29                      break;
30                  else
31                      err_sys("pselect error");
32              } else if (n != 1)
33                      err_sys("pselect error: returned %d", n);
34              len = servlen;
35              n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
36              recvline[n] = 0;  /* null terminate */
37              printf("from %s: %s",
38                      Sock_ntop_host(preply_addr, len), recvline);
39          }
40      }
41 }
42 static void
43 recvfrom_alarm(int signo)
44 {
45      return;     /* just interrupt the recvfrom() */
46 }
```

图 18.7 使用 pselect 阻塞和解阻塞信号[bcast/dgclibcast4.c]

第 22～33 行 阻塞 SIGALRM 并调用 pselect。pselect 最后一个参数是一个指向 sigset_empty 变量的指针。sigset_empty 是一个没有任何信号阻塞的信号集，也就是说，所有的信号都是解阻塞的。pselect 操作包括：保存当前信号掩码(该信号掩码用于阻塞 SIGALRM 信号)，测试指定的描述字，如果必要将信号掩码设置为空集后阻塞调用，但在返回前将掩码恢复到 pselect 调用时的值。pselect 的关键点是设置信号掩码、测试描述字及恢复信号掩码等操作都是原子操作。

第 34～38 行 如果套接口是可读的，则调用 recvfrom。这时我们知道它不会阻塞。

正如 6.9 节提到的，pselect 是 Posix.1g 的一个新增加的调用，图 1.16 中的系统都不支持它。不过我们在图 18.8 中给出了它的一个尽管不正确然而简单的实现。给出这个不正确的实现的原因是为了说明与之相关的操作步骤：保存当前信号掩码并将掩码设置为调用者指定的值，测试描述字，然后恢复信号掩码。

```
 9 #include    "unp.h"
10 int
11 pselect(int nfds, fd_set *rset, fd_set *wset, fd_set *xset,
12         const struct timespec *ts, const sigset_t *sigmask)
13 {
14     int n;
15     struct timeval tv;
16     sigset_t savemask;
17     if (ts != NULL) {
18         tv.tv_sec = ts->tv_sec;
19         tv.tv_usec = ts->tv_nsec / 1000;     /* nanosec -> microsec */
20     }
21     sigprocmask(SIG_SETMASK, sigmask, &savemask);    /* caller's mask */
22     n = select(nfds, rset, wset, xset, (ts == NULL) ? NULL : &tv);
```

```
23      sigprocmask(SIG_SETMASK, &savemask, NULL);      /* restore mask */
24      return(n);
25  }
```

图 18.8　pselect 的一个简单但不正确的实现[lib/pselect.c]

使用 sigsetjmp 和 siglongjmp

另一个解决同一问题的正确方法是不使用信号处理程序中断被阻塞的系统调用，而从信号处理程序调用 siglongjmp。我们称之为非本地跳转(nonlocal goto)，因为使用它可以从一个函数跳转到另一个函数。图 18.9 给出了这个方案。

```
 1 #include    "unp.h"
 2 #include    <setjmp.h>
 3 static void recvfrom_alarm(int);
 4 static sigjmp_buf jmpbuf;
 5 void
 6 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 7 {
 8     int         n;
 9     const int on = 1;
10     char    sendline[MAXLINE], recvline[MAXLINE + 1];
11     socklen_t   len;
12     struct sockaddr *preply_addr;
13     preply_addr = Malloc(servlen);
14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
15     Signal(SIGALRM, recvfrom_alarm);
16     while (Fgets(sendline, MAXLINE, fp) != NULL) {
17         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
18         alarm(5);
19         for ( ; ; ) {
20             if (sigsetjmp(jmpbuf, 1) != 0)
21                 break;
22             len = servlen;
23             n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
24             recvline[n] = 0;    /* null terminate */
25             printf("from %s: %s",
26                     Sock_ntop_host(preply_addr, len), recvline);
27         }
28     }
29 }
30 static void
31 recvfrom_alarm(int signo)
32 {
33     siglongjmp(jmpbuf, 1);
34 }
```

图 18.9　在信号处理程序中使用 sigsetjmp 和 siglongjmp[bcast/dgclibcast5.c]

分配跳转缓冲区

第 4 行　分配由函数和信号处理程序使用的跳转缓冲区。

调用 sigsetjmp

第 20～23 行 从 dg_cli 函数直接调用 sigsetjmp，它在建立了跳转缓冲区后返回 0。接着调用 recvfrom。

处理 SIGALRM 并调用 siglongjmp

第 30～34 行 当 SIGALRM 信号递交时，我们调用 siglongjmp。这将引起 dg_cli 函数中的 sigsetjmp 返回，返回值为第二个参数(1)，它必须为非 0 值。sigsetjmp 返回导致 dg_cli 中 for 循环的结束。

以这种方式使用 sigsetjmp 和 siglongjmp 保证了不会出现由于信号递交时间不当而引起的 recvfrom 永远阻塞现象。发生问题的唯一潜在可能是如果信号在 printf 处理输出的过程中递交。为了防止这种情况出现，应将图 18.6 中的信号阻塞和解阻塞方法同非本地跳转方法结合起来使用①。

使用从信号处理程序到函数的 IPC

还有另一个解决同一问题的正确方法。不是让信号处理程序简单地返回，期望这样能够中断被阻塞的 recvfrom，相反，我们让信号处理程序使用 IPC(进程间通信)通知 dg_cli 函数定时器到时。这与前面提到的当定时器到时时利用信号处理程序设置全局变量 had_alarm 有些相似，因为全局变量也是 IPC 的一种形式(函数和信号处理程序共享内存)。使用全局变量方法的问题在于函数必须测试该变量，如果信号几乎在测试的同时递交，就会导致时序配合问题。

我们在图 18.10 中使用的是进程内部的管道。当定时器到时时，信号处理程序就向管道中写 1 个字节。函数 dg_cli 读取该字节并以此决定什么时候终止 for 循环。使该方法成为一个好方法的原因是我们使用了 select 来测试管道是否可读。我们测试要么套接口可读，要么管道可读。

创建管道

第 15 行 我们创建一个普通的 Unix 管道，并返回两个描述字。pipefd[0]是管道的读端，pipefd[1]是管道的写端。

> 我们也可以使用套接口对来实现一个双工管道。在某些系统上，特别是 SVR4，Unix 管道总是双工的，可从任一端进行读写操作。

对套接口和管道读端进行 select 操作

第 23～30 行 对套接口和管道读端进行 select 操作。

第 45～50 行 当 SIGALRM 信号递交时，信号处理程序向管道中写入 1 个字节，使读

---

① 作者注：图 18.9 有两个潜在的时序问题。首先考虑如果信号是在 recvfrom 返回和将其返回值存入 n 之间递交，那么会发生什么现象。该数据报被认为已丢失(尽管它已由 recvfrom 收到)，不过 UDP 应用程序应该能够处理丢失的数据报。然而如果同样的技术用于 TCP 应用程序，数据就永远丢失了(因为 TCP 已确认了这个数据并把它递送给了应用进程)。
图 18.10 使用 IPC 的 dg_cli 函数也有类似的问题：信号可能在 recvfrom 成功返回和把返回值存入 n 之间递交。这可以通过在 select 返回之后关掉 alarm 来解决。另外一种办法是不用 alarm，而改用 select 的定时功能。
第二个问题是 alarm 调用和首次 sigsetjmp 调用之间的时间小于 alarm 时间(5 秒)也无法保证。解决办法之一是在调用 sigsetjmp 之后设置另外一个标志，并在信号处理程序中测试这个标志：如果该标志还没有设置，那么不调用 siglongjmp，仅仅重置 alarm 就行。
结论是：为了在这些可能的情形下保证健壮性，应避免使用 siglongjmp，而改用 pselect 或 IPC 方法。

端可读。由于信号处理程序返回时可能中断 select，因此，如果 select 返回 EINTR 错误，我们就忽略它，并且由此可知管道的读端是可读的，这样一来将终结 for 循环。

读管道

第 38～41 行 当管道的读端可读时，我们从管道中 read 信号处理程序写入的空字节，并将它忽略。但这告诉我们定时器到时，于是我们 break 出这个无限的 for 循环。

```
 1 #include    "unp.h"
 2 static void recvfrom_alarm(int);
 3 static int  pipefd[2];
 4 void
 5 dg_cli(FILE * fp, int sockfd, const SA * pservaddr, socklen_t servlen)
 6 {
 7     int          n, maxfdp1;
 8     const int on = 1;
 9     char     sendline[MAXLINE], recvline[MAXLINE + 1];
10     fd_set   rset;
11     socklen_t len;
12     struct sockaddr * preply_addr;
13     preply_addr = Malloc(servlen);
14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
15     Pipe(pipefd);
16     maxfdp1 = max(sockfd, pipefd[0]) + 1;
17     FD_ZERO(&rset);
18     Signal(SIGALRM, recvfrom_alarm);
19     while (Fgets(sendline, MAXLINE, fp) ! = NULL) {
20         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
21         alarm(5);
22         for ( ; ; ) {
23             FD_SET(sockfd, &rset);
24             FD_SET(pipefd[0], &rset);
25             if ( (n = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
26                 if (errno == EINTR)
27                     continue;
28                 else
29                     err_sys("select error");
30             }
31             if (FD_ISSET(sockfd, &rset)) {
32                 len = servlen;
33                 n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
34                 recvline[n] = 0;  /* null terminate */
35                 printf("from %s: %s",
36                         Sock_ntop_host(preply_addr, len), recvline);
37             }
38             if (FD_ISSET(pipefd[0], &rset)) {
39                 Read(pipefd[0], &n, 1);      /* timer expired */
40                 break;
41             }
42         }
43     }
44 }
```

```
45 static void
46 recvfrom_alarm(int signo)
47 {
48     Write(pipefd[1], "", 1);     /* write 1 null byte to pipe */
49     return;
50 }
```

图 18.10 使用从信号处理程序到函数的管道作为 IPC[bcast/dgclibcast6.c]

18.6 小 结

连接到子网上的所有主机都要接收广播数据报。广播的缺点是子网上的所有主机都要处理数据报，如果是 UDP 数据报的话，需一直向上处理到 UDP 层，即使不参与广播应用系统的主机也这样。像音频、视频等高数据速率应用系统将给这些主机带来过度的处理负担。我们将在下一章看到使用多播可以解决这个问题。在多播时，只有对这个应用系统感兴趣的主机才会接收到多播数据报。

我们还举例展示了由 SIGALRM 信号带来的竞争状态。由于使用 alarm 函数和 SIGALRM 信号是对读操作设置超时的常用方法，因此这种微妙的错误在网络应用程序中是比较常见的。我们给出了解决这个问题的一个不正确的方法和三个正确方法：

- 使用 pselect
- 使用 sigsetjmp 和 siglongjmp
- 使用从信号处理程序到主循环的 IPC(典型手段是管道)

18.7 习 题

18.1 运行采用广播 dg_cli 函数(图 18.5)的 UDP 客户程序。你接收到了多少个应答？应答总是以同样的顺序到达吗？你网上的主机是否有同步时钟？

18.2 在 18.4 节末尾，我们讨论了广播数据报分片问题。我们说为了方便移植，应用程序应将广播数据报限制在 1472 字节以内。这个数据是如何得来的？

18.3 在图 18.10 中 select 的后面加上若干 printf 语句，以便观察 select 的返回值情况(返回一个错误或者两个描述字之一的可读条件)。当 alarm 到时时，你的系统返回的是 EINTR 还是管道的可读条件？

18.4 运行 tcpdump 等工具以捕获局域网上的广播数据报，所用 tcpdump 命令为 tcpdump ether broadcast。分析一下这些广播数据报属于哪些协议族。

第19章 多 播

19.1 概 述

如图18.1所示，单播地址标识单个接口，广播地址标识子网上的所有接口，多播地址标识一组接口。单播和广播是编址方案的两个极端(要么一个要么全部)，多播的目的就在于提供一种折衷方案。多播数据报仅由对该数据报感兴趣的接口接收，也就是说，由运行希望参加多播会话应用系统的主机上的接口接收。广播一般局限于局域网，而多播既可用于局域网，也可跨越广域网。事实上，基于MBone(B.2节)的应用系统每天跨越整个因特网多播。

套接口API中增加的多播支持内容比较简单：5个套接口选项，其中3个选项影响到多播UDP数据报的发送，另外2个选项影响到多播数据报的接收。

19.2 多播地址

我们必须区分IPv4多播地址和IPv6多播地址。

IPv4中的D类地址

IPv4 D类地址(从224.0.0.0到239.255.255.255)是多播地址(图A.3和A.4)。D类地址的低28位构成了多播组ID(group ID)，而整个32位地址则称为组地址(group address)。

图19.1给出了从多播地址映射到以太网地址的方法，RFC1112[Deering 1989]对此作了描述。RFC1390[Katz 1993]描述了从多播地址到FDDI网地址的映射，RFC1469[Pusateri 1993]描述了到令牌环网地址的映射。图中我们同时给出了IPv6多播地址的映射，以便比较二者的结果以太网地址。

考察一下IPv4的映射。以太网地址的高序24位总是01:00:5e。下一位总是0，低序23位是多播地址低序23位的拷贝。多播地址的高序5位在映射过程中被忽略。这意谓着多个32位多播地址会映射为单个以太网地址：映射不是一对一。

以太网地址第一个字节的低序2位说明该地址是一个统一管理的组地址。统一管理意味着高序24位由IEEE分配，组地址需由接收接口进行特别识别和处理。

下面是几个特殊的IPv4多播地址：

- 224.0.0.1是一个所有主机(all-hosts)组。子网上所有具有多播能力的主机必须在所有具有多播能力的接口上加入该组。(我们不久将谈到加入一个多播组意味着什么。)
- 224.0.0.2是一个所有路由器(all-routers)组。所有多播路由器必须在所有具有多播能力的接口加入该组。

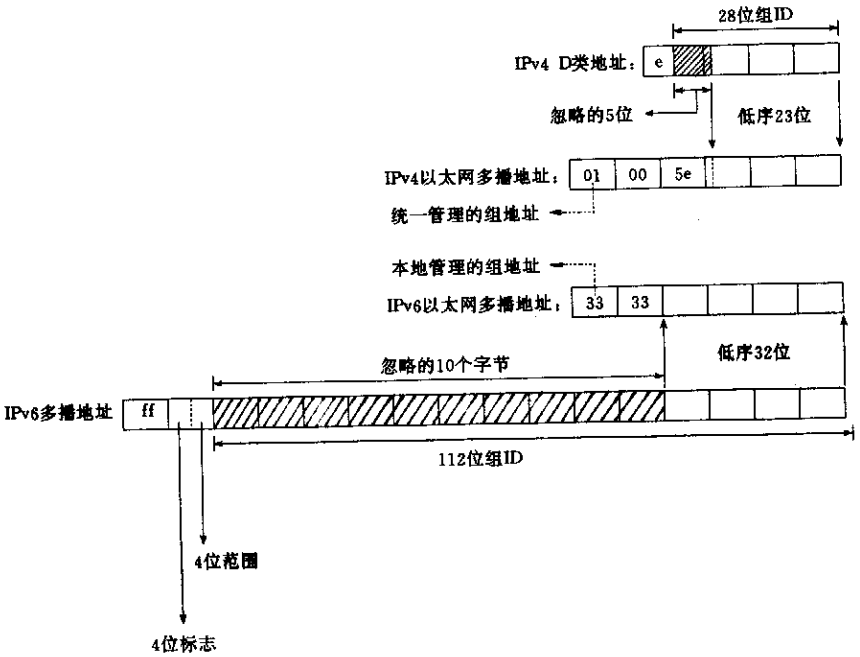


图 19.1 IPv4 和 IPv6 多播地址到以太网地址的映射

介于 224.0.0.0 到 224.0.0.255 间的地址(我们也将它写成 224.0.0.0/24)称为链路局部(link local)地址。这些地址被保留用于低级拓扑发现和维护协议。以这些地址为目的地址的数据报不能被多播路由器转发。我们将在考察了 IPv6 多播地址后再进一步讨论不同 IPv4 多播地址的范围。

IPv6 多播地址

IPv6 多播地址的高序字节值为 ff。图 19.1 给出了将 16 字节的 IPv6 多播地址映射到 6 字节以太网地址的方法。组地址的低序 32 位拷贝到以太网地址的低序 32 位。以太网的高序 2 字节值为 33:33。RFC1972[Crawford 1996a]描述了这种映射关系,RFC2019[Crawford 1996b]描述了 IPv6 组地址到 FDDI 网地址的映射,[Thomas 1997]描述了到令牌环网地址的映射。

以太网地址第一字节的低序 2 位表明该地址是一个本地管理组地址。本地管理意味着不能保证地址的唯一性。可能有除 IPv6 外的其他协议族共享同一网络并使用同样的以太网地址高序 2 字节值。正如我们前面提到的,组地址由接收接口进行特殊识别和处理。

4 位的多播标志用于区分众所周知(well-known)多播组(值为 0)和临时(transient)多播组(值为 1)。该字段的高 3 位保留。IPv6 多播地址还包含一个 4 位的范围(scope)字段,我们不久将对它加以讨论。

下面是一些特殊的 IPv6 多播地址。

- ff02::1 是一个所有节点(all-nodes)组。子网上的具有多播能力的所有主机必须在具有多播能力的所有接口上加入该组。这与 IPv4 的 224.0.0.1 地址相类似。
- ff02::2 是一个所有路由器(all-routers)组。所有子网上的多播路由器必须在具有多

播能力的所有接口上加入该组。这与IPv4的224.0.0.2多播地址相类似。

由于在映射到硬件地址时只用到IPv6多播地址的低序32位，[Hinden and Deering 1997]推荐至少在需要使用位于这32位左边的10个字节(图19.1)前，将这10个字节值置为0。

多播地址的范围

IPv6多播地址有一个4位的显式范围字段，该字段决定了多播数据报能够游走的范围。IPv6分组也有一个跳限字段，它限制分组被路由器转发的次数。下面是几个已经分配了的范围字段值：

1：节点局部即局部于节点(node-local)
2：链路局部即局部于链路(link-local)
3：网点局部即局部于网点(site-local)
8：组织局部即局部于组织(orgainization-local)
14：全球(global)

其他值或者还没有分配，或者已经保留。节点局部数据报禁止从接口输出，而链路局部数据报不能被路由器转发。网点和组织的定义由该网点或组织的多播路由器管理员决定。只是范围字段值不同的IPv6多播地址代表不同的组。

IPv4的多播数据报没有单独的范围字段。从历史上看，IPv4的TTL字段具有多播范围的含义：0意味着局部于节点，1意味着局部于链路，2～32意味着局部于网点，33～64意味着局部于区域(region-local)，65～128意味着局部于大洲(contionent-local)，129～255意味着无范围限制(全球)。这种TTL字段的双重应用导致了很多问题，[Meyer 1997]对此有详细的描述。

虽然将IPv4 TTL字段用作多播范围控制已被接受并被推荐实际使用，但可管理的范围划分是更可取的。于是239.0.0.0到239.255.255.255间的地址被规定为管理上划分范围的IPv4多播地址空间(administratively scoped IPv4 multicast space)([Meyer 1997])，它占据多播地址空间的高端。该范围内的地址由组织内部分配，但在跨越组织边界时不能保证地址的唯一性。一个组织应将其边界路由器(组织边界的多播路由器)配置为禁止转发以这些地址为目的地址的多播数据报。

我们将IPv4管理上划分范围的多播地址划分为本地范围和组织局部范围两部分。前者与IPv6的网点局部范围相似(但不是语义等价)。图19.2对不同的范围划分规则作了总结。

范围	IPv6范围	IPv4	
		TTL范围	管理范围
节点局部	1	0	
链路局部	2	1	224.0.0.0到224.0.0.255
网点局部	5	<32	239.255.0.0到239.255.255.255
组织局部	8		239.192.0.0到239.195.255.255
全球	14	<255	224.0.1.0到238.255.255.255

图19.2 IPv4和IPv6多播地址范围

19.3 局域网上多播和广播的比较

我们现在返回到图 18.3 和图 18.4 中的示例来看一下在多播的情况下发生了什么。我们以图 19.3 所示的 IPv4 为例，但 IPv6 与此相似。

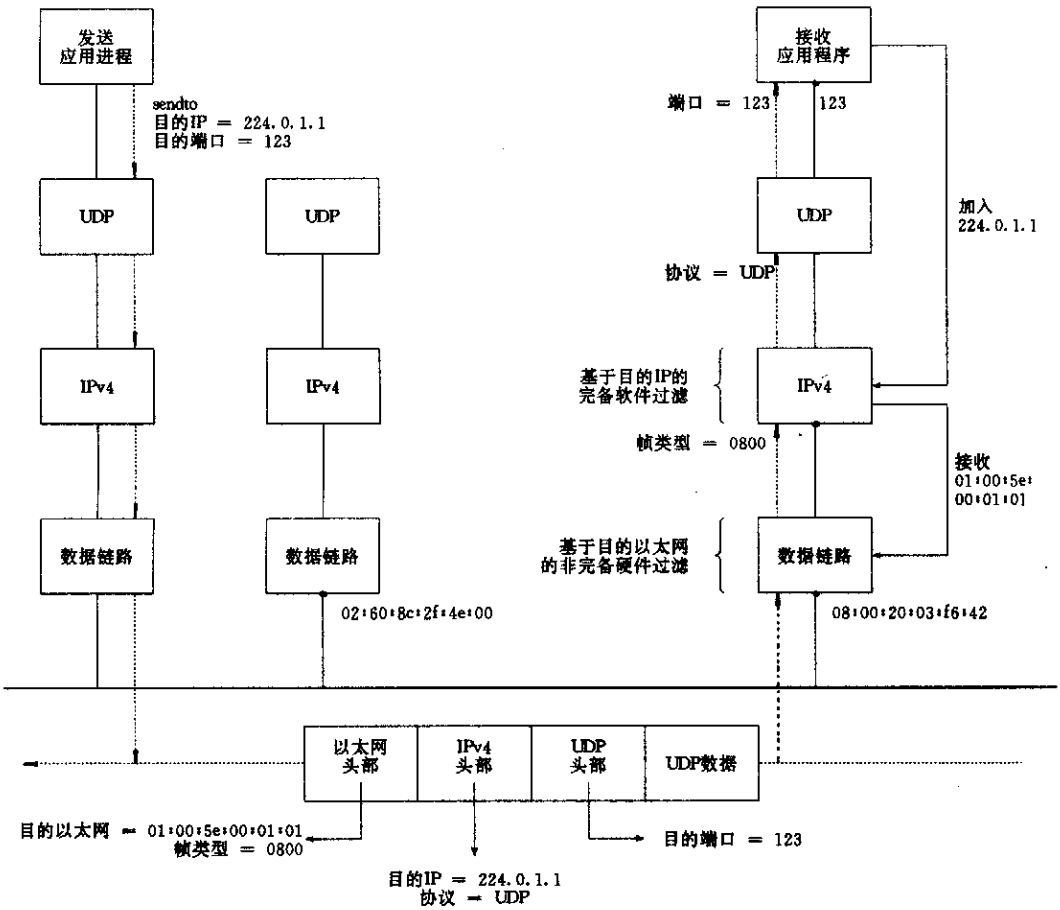


图 19.3 UDP 数据报多播示例

最右边主机的用于接收的应用进程启动，创建 UDP 套接口，捆绑端口 123，然后加入多播组 224.0.1.1。我们不久将看到这种“加入”操作是通过调用 setsockopt 完成的。在上述操作完成后，IPv4 层将这些信息保存起来，告诉适当的数据链路接收以 01:00:5e:00:01:01 为目的地址的帧(TCPv2 的 12.11 节)。该地址是上述多播地址利用图 19.1 中的方法映射来的。

下一步是最左边主机的用于发送的应用进程创建一个 UDP 套接口，并向 IP 地址 224.0.1.1 及端口 123 发送数据报。发送多播数据报不需要任何特殊处理；应用程序不需要加入多播组。发送主机将 IP 地址映射到相应的以太网目的地址，并将帧发送出去。注意帧中同时包含着目的以太网地址(由接口检查)和目的 IP 地址(由 IP 层检查)。

我们假定中间的主机不支持 IPv4 多播(在 IPv4 中支持多播是可选的)。它将完全忽略

该帧,因为(1)目的以太网地址与接口地址不匹配,(2)目的以太网地址不是以太网广播地址,(3)没有通知接口接收任何组地址(组地址的高序字节的低序位为1,如图19.1所示)。

该帧基于一种我们称之为非完备过滤(imperfect filtering)的机制被右边的数据链路接收。这种机制是接口利用以太网目的地址实现的。我们之所以说它不完备是因为当我们告诉接口接收以某个具体以太网组地址为目的地址的帧时,它也可能接收以其他以太网组地址为目的地址的帧。

> 当我们告诉以太网接口接收目的地址为某个具体以太网组地址的帧时,当前许多网卡对该地址实施一种散列(hash)函数运算,计算出一个介于0和63之间的值,并将一个64位数组的相应位置1。当一个去往某个组地址的帧在电缆上经过时,接口对其目的地址实施同样的散列函数运算,计算出一个介于0到63之间的一个值。如果数组中的相应位置1,就接收这个帧;否则就忽略这个帧。为了减少接口接收与它无关帧的可能性,较新的网络接口卡便将位数组的大小从64增加到512。随着时间的推移和越来越多的应用系统使用多播,位数组的大小可能进一步增加。现在,某些接口已经实现了完备过滤(perfect filtering)。但有些接口卡根本不进行多播过滤,当告诉接口卡接收某个具体组地址时,它必须接收所有的多播帧(有时我们称之为多播混杂(multicast promiscuous)方式)。流行的接口卡能跟拥有512位散列表一样准确地完成16个组地址的完备过滤。即使接口实现了完备过滤,IP层的完备软件过滤仍然是需要的,因为从IP多播地址到硬件地址的映射不是一对一的。

假设右边的数据链路接收该帧。由于以太网帧类型为IPv4,相应分组被传送到IP层。由于所接收分组的目的地址为多播IP地址,IP层于是将该分组的目的地址和自己的应用进程已加入的所有多播地址进行比较,如果是自己需要的分组就接受,否则就丢弃。我们称这个过程为完备过滤(perfect filtering),因为它是基于完整的32位D类地址完成的。在本例中分组被IP层接收,并被传送到UDP层,UDP层随后将相应数据报传送到绑定端口123的套接口。

下面是图19.3中没有说明的三种情况。

1. 一台运行加入到多播地址225.0.1.1的应用进程的主机。由于多播地址的高5位在映射到以太网地址的过程中被忽略,该主机的以太网地址也将接收以太网目的地址为01:00:5e:00:01:01的帧。在这种情况下,分组将在IP层由完备过滤丢弃。
2. 一台运行加入某个多播组的应用进程的主机,其多播组的相应以太网地址实施散列后恰好与01:00:5e:00:01:01的散列结果一致(也就是说接口卡执行非完备过滤)。这种帧将在数据链路层或IP层丢弃。
3. 去往同组(224.0.1.1)不同端口(譬如4000)的分组。图19.3最右边主机仍接收这类分组,但它们由IP层接受后由UDP层丢弃(假定无套接口绑定端口4000)。

> 这说明了如果一个进程要接收多播数据报,该进程必须加入相应组并绑定相应端口。

19.4 广域网上的多播

就像我们在前面讨论的，在单个局域网上的多播是简单的。一台主机发送一个多播分组，任何有兴趣的主机都会接收此数据报。多播相对于广播的优势在于会减轻其他对此多播分组不感兴趣的主机的负担。

在广域网上，多播也是有用的。考虑一下在图19.4中展示的广域网。

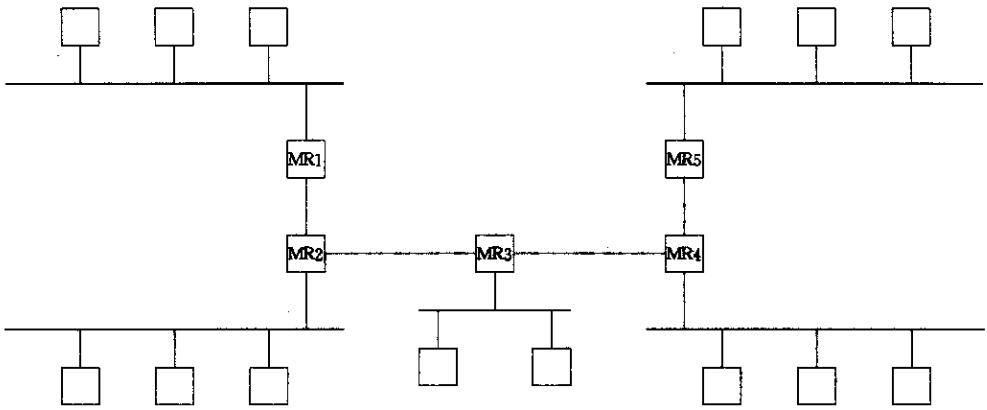


图19.4 用5台多播路由器连接的5个局域网

我们展示了用5台多播路由器连接着的5个局域网。

下面假设在其中的5台主机上启动了一些程序(例如一个监听多播音频会话的程序)且这5个程序(进程)加入了一个给定的多播组。于是这5台主机也就加入了那个多播组。我们还假设多播路由器都与相邻的多播路由器通过"多播路由协议(multicast routing protocol)"相互通信，我们用MRP指称这个协议。整个情况如图19.5所示。

当一台主机上的进程加入一个多播组时，这台主机向所有连着的多播路由器发送一个IGMP消息，通知它们本主机加入了那个组。这些多播路由器于是利用多播路由协议交换这个信息。这样每台路由器在它收到一个目的地为多播地址的分组时就知道如何处理。

> 多播路由仍是一个研究课题，其本身就能写一本书。[Maufer and Semeria 1997]提供了一个简介和概貌。

我们现在假设一个进程在左上的主机上开始发送一个到多播地址的数据报，譬如说这个进程发送一些多播接收者正等待接收的音频数据报。我们在图19.6中展示了这些数据报。

我们可以随着多播数据报从发送者到所有接收者发生的几步看一下。

- 在左上的局域网上，数据报由发送者多播。接收者H1与MR1接收到了这些数据报。(因为H1加入了这个组，而多播路由器必须接收所有多播数据报。)
- MR1将多播数据报转发给MR2，因为多播路由协议告诉MR1，MR2要接收目的地为这个组的数据报。

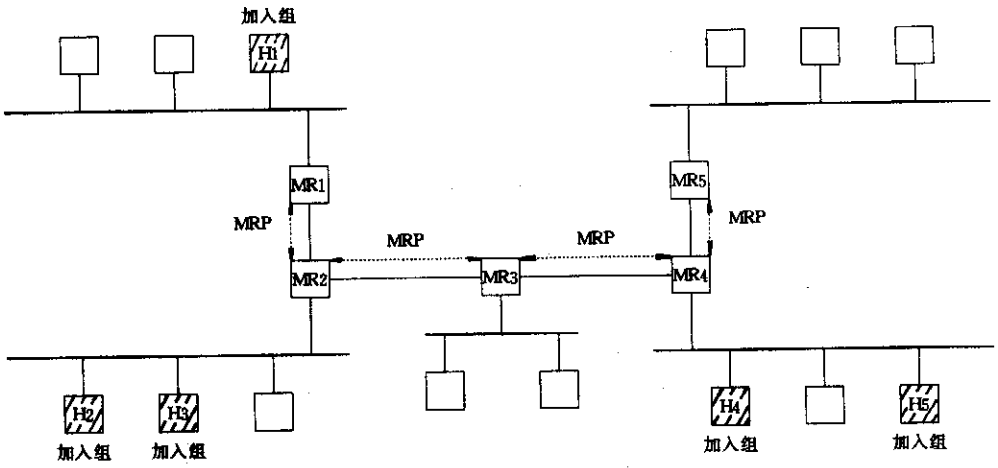


图 19.5 在广域网上 5 台主机加入一个多播组

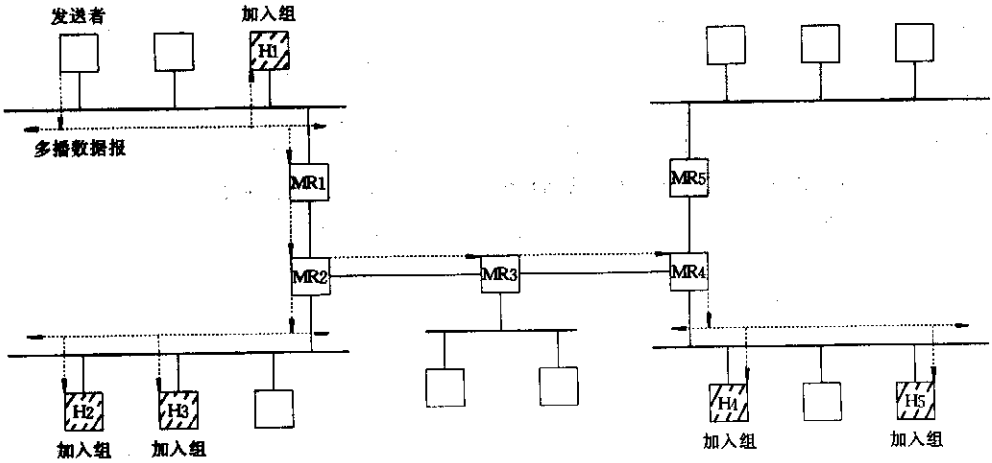


图 19.6 在广域网上发送多播数据报

- MR2 向跟它连着的局域网多播此数据报，因为 H2 和 H3 属于这个组。MR2 也向 MR3 发送一个数据报的复本。
  像 MR2 那样复制数据报(分组)是多播转发时特有的。单播分组在被路由器转发时是从不会被复制的。
- MR3 向 MR4 发送多播数据报，但它不在跟自己连着的局域网上多播，因为我们假设在这个局域网上没有主机加入这个组。
- MR4 将数据报多播到它所连接的局域网上，因为 H4 和 H5 属于这个组。它并不向 MR5 发送数据报的复本，因为在 MR5 所连的局域网上没有主机属于这个组，而且 MR4 根据与 MR5 交换的多播路由信息已经知道这一点。

在广域网上的两个稍差的多播替代方法是广播泛滥(broadcast flooding)和给每一个接收者都发送一个单独的数据报复本。在第一种方法中，数据报由发送者广播，每个路由器也要向除该数据报的到达接口以外的每一个接口广播。很显然，这将会增加对该数据报不感兴趣但又必须处理它的主机和路由器的数目。

在第二种方法中,发送者必须知道所有接收者的IP地址并且发送给每个接收者一份复本。在图19.6的5个接收者的例子中,这要求在发送者的局域网上发送5个数据报,其中从MR1到MR2走4个数据报,从MR2到MR3到MR4走2个数据报。

19.5 多播套接口选项

多播的API支持仅需要5个新的套接口选项。图19.7展示了这5个套接口选项和在IPv4和IPv6中调用getsockopt与setsockopt时所期望参数的数据类型。指向所给出数据类型变量的指针是getsockopt和setsockopt的第四个参数。所有10个(5对)选项对setsockopt都是合法的,但加入和离开多播组的4个(2对)选项在getsockopt中是不允许的。

选项名	数据类型	说明
IP_ADD_MEMBERSHIP	struct ip_mreq	加入一个多播组
IP_DROP_MEMBERSHIP	struct ip_mreq	离开一个多播组
IP_MULTICAST_IF	struct in_addr	指定外出多播数据报的外出接口
IP_MULTICAST_TTL	u_char	指定外出多播数据报的TTL
IP_MULTICAST_LOOP	u_char	使能或禁止外出多播数据报的回馈
IPV6_ADD_MEMBERSHIP	struct ipv6_mreq	加入一个多播组
IPV6_DROP_MEMBRSHIP	struct ipv6_mreq	离开一个多播组
IPV6_MULTICAST_IF	u_int	指定外出多播数据报的外出接口
IPV6_MULTICAST_HOPS	int	指定外出多播数据报的跳限
IPV6_MULTICAST_LOOP	u_int	使能或禁止外出多播数据报的回馈

图19.7 多播套接口选项

IPv4的TTL和回馈选项用u_char的参数,而IPv6的跳限和回馈选项使用int和u_int的参数。既然图7.1中大多数其他套接口选项都使用整数参数,使用IPv4多播选项时的一个常见错误就是也使用int参数指定TTL和回馈选项(这是不允许的,见TCPv2第354~355页)。IPv6对这两个选项类型的改动减少了出错的倾向。

我们现在详细地描述这5对套接口选项。注意IPv4和IPv6的各自5个选项在概念上是相同的;只是名字和参数类型不同。

IP_ADD_MEMBERSHIP和IPV6_ADD_MEMBERSHIP

在一个指定的本地接口上加入一个多播组。我们用IPv4中的单播地址或IPv6中的接口索引去指定本地接口。当加入或离开一个组时,要用到下面两个结构。

```
struct ip_mreq {
  struct in_addr imr_multiaddr;     /* IPv4 class D multicast addr */
  struct in_addr imr_interface;     /* IPv4 addr of local interface */
};
struct ipv6_mreq {
  struct in6_addr ipv6mr_multiaddr;     /* IPv6 multicast addr */
```

```
    unsigned int    ipv6mr_interface;    /* interface index, or 0 */
};
```

如果本地接口指定为通配地址(在IPv4中为INADDR_ANY)或者IPv6中的索引0,那么本地接口就由内核来选择。

如果某台主机上当前有一个或多个进程属于某个接口上的一个给定多播组,我们就称该主机在那个接口上属于所给定组。

在一个给定的套接口上可以多次加入多播组,但是每次加入的必须是一个不同的多播地址,或者虽为同一个多播地址但不在此套接口以前加入此地址的接口上。这可以用于多宿主机,例如,创建一个套接口,接着以给定的多播地址加入每一个接口。

回忆图19.2中,IPv6的多播地址有一个明确的范围字段。正像我们指出的,IPv6的多播地址仅仅范围不同就代表不同组。因此,如果一个网络时间协议的实现想接收所有的NTP数据报,不管范围,它必须加入ff01::101(节点局部)、ff02::101(链路局部)、ff05::101(网点局部)、ff08::101(组织局部)和ffoe::101(全球)。所有的加入可以在单个套接口上完成,然后设置IPV6_PKTINFO套接口选项(20.8节)去让recvmsg返回每个数据报的目的地址。

> 大多数的实现对允许每个套接口加入的(多播组-本地接口对)数目有一个限制。对源自Berkeley的实现这个限制是20。
>
> 当加入的接口没有指定时,源自Berkeley的内核在通常的IP路由表中查找多播地址,并使用查出的接口(TCPv2第357页)。有些系统在初始化时给所有多播地址(对于IPv4就是目的地址为224.0.0.0/8的路径)安装一个路径去处理这种情况①。
>
> 在IPv6中有了改变,它使用接口索引去指定接口,而不像IPv4那样使用本地单播地址,这样可允许加入未指定网络地址的(unnumbered)接口和隧道端点(tunnel endpoint)。这就是为什么我们在17.6节中介绍的接口名和索引函数在RFC 2133[Gilligan et al. 1997]中引入的原因。

IP_DROP_MEMBERSHIP 和 IPV6_DROP_MEMBERSHIP

在一个指定的本地接口上离开一个多播组。我们刚才给出的加入一个组的结构也用于这对套接口选项。如果本地接口没有指定(也就是说其值为IPv4的INADDR_ANY或者IPv6的接口索引0),那么第一个匹配的多播组的组成员关系将被去掉。

如果一个进程加入了一个组但从未明确地离开那个组,当套接口关闭时(或者显式地关闭或者因进程终止),成员关系将被自动地去掉。同一个主机上的多个进程都加入同一个组也是可以的,这种情况下,主机一直是那个组的成员,直到最后一个进程离开那个组。

IP_MULTICAST_IF 和 IPV6_MULTICAST_IF

给从本套接口上发送的外出多播数据报指定接口。这个接口在IPv4中被指定为in_addr结构,在IPv6中被指定为接口索引。如果其值为IPv4中的INADDR_ANY或IPv6中的

① 译者注: 原书中给出的IPv4的所有多播地址为224.0.0.0/8,但译者认为应是224.0.0.0/4(见19.2节)。224.0.0.0/8仅仅是链路局部多播地址的超集。

接口索引0,这将去掉以前通过这个选项指派给此套接口的接口,系统于是给每个外出数据报选择接口。

注意仔细区分当进程加入组时指定的(或由内核心选择的)本地接口(所到达的多播数据报将在该接口上接收)以及当送出多播数据报时指定的(或由内核心选择的)本地接口。

> 源自Berkeley的内核通过在通常的IP路由表中检索通往目的多播地址的路径来给多播数据报选择缺省接口。同样的技术也被用于当进程加入组时没有指定接口的情况下选择接收接口。这里假定如果存在通往给定的多播地址的路径(或许为路由表中的缺省路径),那么检索出的接口应被用于输入和输出。

IP_MULTICAST_TTL和IPV6_MULTICAST_HOPS

给外出的多播数据报设置IPv4的TTL或IPv6的跳限。如果不指定,那么缺省值都将为1,也就是限制数据报在本地子网。

IP_MULTICAST_LOOP和IPV6_MULTICAST_LOOP

打开或关闭多播数据报的本地自环即回馈。缺省时回馈是打开的:如果一个主机在发送接口上属于一个多播组,该主机上的进程发送的每个数据报的复本也会回馈,被主机当作接收的数据报处理。

这有点类似广播,一个主机上发送的广播数据报也被当作接收的数据报(图18.4)。(但在广播中是无法禁止回馈的。)这意味着如果一个进程属于它所发送数据报的多播组,它将会收到自己发送的东西。

> 在这里讨论的回馈是在IP层或更高层进行的内部回馈。如果接口听到了它自己发送的东西,RFC1112[Deering 1989]要求驱动程序忽略这些复本。该RFC还说明:回馈选项缺省情况下打开,目的是作为"一种某些上层协议的性能优化措施,而这些协议的特点是把一个组的成员限定到每台主机上的单个进程(例如路由协议)"。

前两对套接口选项(ADD_MEMBERSHIP和DROP_MEMBERSHIP)影响多播数据报的接收,后三对影响多播数据报的发送(外出接口、TTL或跳限及回馈)。我们以前提到过,发送多播数据报不需要什么特别的。如果在发送多播数据报之前没有指定多播套接口选项,数据报的外出接口将由内核来选择,TTL或跳限将为1,且有一个复本回馈回来。

进程要想接收多播数据报就必须加入那个多播组,并捆绑UDP套接口到某个端口,而这个端口将用作发到这个组的数据报的目的端口。这两个操作是不同的但都是必须的。加入这个组就是告诉所在主机的IP层和数据链路层接收发送到这个组的多播数据报。捆绑端口则是应用进程向UDP指示它想接收发送到此端口的数据报的手段。有些应用进程不仅捆绑端口,而且将多播地址也捆绑到套接口上,这样可防止此端口可能接收的其他数据报也递送到此套接口。

> 习惯上,源自Berkeley的实现仅要求主机上的某个套接口加入多播组,而不必给它捆绑端口就能接收多播数据报。但是这些实现有潜在的可能性,使多播数据报递送到并没有多播意识的应用进程。新的多播内核要求进程绑定端口并设置任意一个多播套接口选项,后一步指明应用进程意识到多播。最通常设置

的多播套接口选项是加入组。Solaris 2.5 与此有点不同，它仅递送收到的多播数据报到不但加入了组而且绑定了端口的套接口。为了移植性，所有的多播应用程序都应当加入组并捆绑端口。

有些老的能多播的主机不允许捆绑多播地址到套接口。为了移植性，应用程序可能需要忽略多播地址的 bind 调用出错。

19.6 mcast_join 和相关函数

尽管 5 对多播套接口选项对 IPv4 和 IPv6 是相似的，但仍有足够的区别使得使用多播的协议无关代码变得复杂，因为得加不少 #ifdef 语句。一个较好的解决方法是使用下面的 8 个函数隐藏这些区别。

```
#include "unp.h"
    int mcast_join(int sockfd,const struct sockaddr *sa,socklen_t salen,
                   const char *ifname,u_int ifindex);
    int mcast_leave(int sockfd,const struct sockaddr *sa,socklen_t salen);
    int mcast_set_if(int sockfd,const char *ifname,u_int ifindex);
    int mcast_set_loop(int sockfd,int flag);
    int mcast_set_ttl(int sockfd,int ttl);
                                        以上全部返回：成功时为 0，出错时为 -1
    int mcast_get_if(int sockfd);
                                        返回：成功时为非负接口索引，出错时为 -1
    int mcast_get_loop(int sockfd);
                                        返回：成功时为当前回馈标志，出错时为 -1
    int mcast_get_ttl(int sockfd);
                                        返回：成功时为当前 TTL 或跳限，出错时为 -1
```

mcast_join 函数加入一个多播组，这个组的 IP 地址在由 addr 指向的套接口地址结构中，它的长度由 salen 指定。我们可以指定在哪个接口上加入组：或者使用接口的名字（一个非空的 ifname），或者使用非零的接口索引（ifindex）。如果都没有给出，内核将选择在哪个接口上加入组。前面讲过，IPv6 中接口是通过索引指定给套接口选项的。如果已给 IPv6 套接口指定一个名字，我们就调用 if_nametoindex 函数获取其接口索引。在 IPv4 套接口选项中，接口是通过单播 IP 地址来指定的。如果已给 IPv4 套接口指定一个名字，我们就调用 SIOCGIFADDR 请求的 ioctl 函数获取其接口的单播 IP 地址。如果已给 IPv4 套接口指定一个索引，我们先调用 if_indextoname 函数获得名字，再像刚才描述的那样处理名字。

> 接口的名字（如 le0 或 ether0）是用户指定接口的通常方法，一般不直接用 IP 地址或索引。例如，tcpdump 是很少的允许用户指定接口的几个程序之一，它的 -i 选项要求以一个接口名字作为参数。

mcast_leave 离开一个多播组，这个组的 IP 地址在由 addr 指向的套接口地址结构中。

mcast_set_if 给外出多播数据报设置缺省的接口索引。如果 ifname 非空，那么它指定了接口的名字；否则，如果 ifindex 大于 0，那么它指定了接口索引。对于 IPv6，名字利用 if_nametoindex 函数映射为索引。对于 IPv4，从名字或索引到接口的单播 IP 地址的映射是通过前面描述 mcast_join 时给出的方法进行的。

mcast_set_loop 设置回馈选项为 1 或 0，mcast_set_ttl 设置 IPv4 的 TTL 或 IPv6 的跳限。另外三个 mcast_get_XXX 函数返回相应的值。

例子：mcast_join 函数

图 19.8 给出了 mcast_join 函数的前半部分。这部分处理 IPv4 的套接口。

处理索引

第 11～19 行　在套接口地址结构中的 IPv4 多播地址被复制到一个 ip_mreq 结构中。如果指定了索引，我们就调用 if_indextoname，把名字存到 ifreq 结构中。这成功后，我们往前跳转以调用 ioctl。

处理名字

第 20～27 行　调用者的名字被复制到 ifreq 结构中，SIOCGIFADDR 的 ioctl 调用返回与此名字相关联的单播地址。成功后，IPv4 地址被复制到 ip_mreq 结构中的 imr_interface 成员中。

指定缺省值

第 28～29 行　如果索引和名字都未指定，接口将被设置为通配地址，告诉内核去选择接口。

第 30～31 行　setsockopt 函数执行加入操作。

```
 1 #include    "unp.h"
 2 #include    <net/if.h>
 3 int
 4 mcast_join(int sockfd, const SA *sa, socklen_t salen,
 5              const char *ifname, u_int ifindex)
 6 {
 7     switch (sa->sa_family) {
 8     case AF_INET: {
 9             struct ip_mreq     mreq;
10             struct ifreq       ifreq;
11             memcpy(&mreq.imr_multiaddr,
12                    &((struct sockaddr_in *) sa)->sin_addr,
13                    sizeof(struct in_addr));
14             if (ifindex > 0) {
15                 if (if_indextoname(ifindex, ifreq.ifr_name) == NULL) {
16                     errno = ENXIO;     /* i/f index not found */
17                     return(-1);
18                 }
19                 goto doioctl;
20             } else if (ifname != NULL) {
21                 strncpy(ifreq.ifr_name, ifname, IFNAMSIZ);
22             doioctl:
23                 if (ioctl(sockfd, SIOCGIFADDR, &ifreq) < 0)
```

```
24                      return(-1);
25                  memcpy(&mreq.imr_interface,
26                         &((struct sockaddr_in *) &ifreq.ifr_addr)->sin_addr,
27                         sizeof(struct in_addr));
28              } else
29                  mreq.imr_interface.s_addr = htonl(INADDR_ANY);
30              return(setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
31                                &mreq, sizeof(mreq)));
32      }
```

图 19.8 加入一个多播组:IPv4 套接口[lib/mcast_join.c]

处理 IPv6 的后半部分函数在图 19.4 给出。

处理索引、名字或缺省值

第 36～50 行 首先,IPv6 的多播地址被从套接口地址结构中复制到 ipv6_mrep 结构中。如果指定了索引,这个索引将被复制到 ipv6mr_interface 成员中。否则,如果指定了名字,则通过调用 if_nametoindex 来获取索引。两者都没有指定时,把接口索引置为 0 调用 setsockopt,告诉内核去选择接口。这就加入了组。

```
33 #ifdef  IPV6
34      case AF_INET6: {
35          struct ipv6_mreq  mreq6;
36          memcpy(&mreq6.ipv6mr_multiaddr,
37                 &((struct sockaddr_in6 *) sa)->sin6_addr,
38                 sizeof(struct in6_addr));
39          if (ifindex > 0)
40              mreq6.ipv6mr_interface = ifindex;
41          else if (ifname != NULL) {
42              if ( (mreq6.ipv6mr_interface = if_nametoindex(ifname)) == 0) {
43                  errno = ENXIO;      /* i/f name not found */
44                  return(-1);
45              }
46          } else
47              mreq6.ipv6mr_interface = 0;
48          return(setsockopt(sockfd, IPPROTO_IPV6, IPV6_ADD_MEMBERSHIP,
49                            &mreq6, sizeof(mreq6)));
50      }
51 #endif
52      default:
53          errno = EPROTONOSUPPORT;
54          return(-1);
55      }
56 }
```

图 19.9 加入一个多播组:IPv6 套接口[lib/mcast_join.c]

例子:mcast_set_loop 函数

图 19.10 给出了我们的 mcast_set_loop 函数。

由于参数是一个套接口描述字,而不是一个套接口地址结构,我们于是调用自己的 sockfd_to_family 函数来获取套接口的地址族。随后设置相应的套接口选项。

我们不再给出所有剩余的 mcast_XXX 函数的源代码,但它们是免费可得的(见前言)。

19.7 使用多播的 dg_cli 函数

我们通过仅去掉图 18.5 中 dg_cli 函数对 setsockopt 的调用来修改此函数。如前所述，如果缺省的外出接口、TTL 和回馈选项设置合适，那么发送多播数据报就不必设置多播套接口选项。我们指定所有主机组作为目地地址来运行我们的程序。

```
1 #include    "unp.h"
2 int
3 mcast_set_loop(int sockfd, int onoff)
4 {
5     switch (sockfd_to_family(sockfd)) {
6     case AF_INET: {
7         u_char      flag;
8         flag = onoff;
9         return(setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_LOOP,
10                           &flag, sizeof(flag)));
11    }
12#ifdef      IPV6
13    case AF_INET6: {
14            u_int       flag;
15            flag = onoff;
16            return(setsockopt(sockfd, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
17                              &flag, sizeof(flag)));
18    }
19 #endif
20    default:
21        errno = EPROTONOSUPPORT;
22        return(-1);
23    }
24 }
```

图 19.10 设置多播回馈选项[lib/mcast_set_loop.c]

```
solaris % udpcli01 224.0.0.1
hi there
from 206.62.226.40: Fri Jul 18 13:02:41 1997   Linux
from 206.62.226.35: Fri Jul 18 13:02:41 1997   BSD/OS
from 206.62.226.43: Fri Jul 18 13:02:41 1997   AIX
from 206.62.226.34: Fri Jul 18 13:02:41 1997   BSD/OS
from 206.62.226.42: Fri Jul 18 13:02:41 1997   Digital Unix
```

有 5 个主机响应，因为每个主机都能多播，从而都加入了所有主机组，并且都有一个进程绑定了目的 UDP 端口(13，时间/日期服务器)。因此，到来的多播数据报被递送到这个套接口。通常，这个服务器是 inetd 的一部分。每个应答都是单播，因为请求的源地址是单播地址，而这个地址又被每个服务器用作应答的目的地址。

主机 sunos5 是子网上唯一一个能多播但没有响应的主机。我们以前提到过，Solaris 需要套接口加入组，这可不是使用标准时间/日期服务器的本例子的情况。

IP 分片和多播

我们在 18.4 节末尾提到，多数系统作为一种策略不允许对广播数据报进行分片。在多播中分片是没有问题的，我们可以用同样的含有一个 2000 字节长行的文件简单地验证。

```
bsdi % udpcli01 224.0.0.1 < 2000line
from 206.62.226.35: Fri Jul 18 14:31:50 1997
from 206.62.226.34: Fri Jul 18 14:31:50 1997
from 206.62.226.40: Fri Jul 18 14:31:50 1997
from 206.62.226.43: Fri Jul 18 14:31:50 1997
from 206.62.226.42: Fri Jul 18 14:31:50 1997
```

许多源自 4.4BSD 的实现在对多播数据报分片中存在缺陷：第一个片段外的其余片段不被作为链路层的多播帧来发送。

19.8 接收 MBone 会话声明

在 MBone(B.2 节)上，为了接收一个多媒体会议，一个站点仅需知道会议的多播地址和会议数据流(例如音频和视频)的 UDP 端口号。SAP(会话声明协议[Handley 1996])描述这样做的方法(声明被多播到 MBone 上所用的数据报头部和频率)，SDP(会话描述协议[Handley and Jacobson 1997])则描述声明的内容(如何指定多播地址和 UDP 端口号)。想在 MBone 上声明会话的站点周期性地向一个众所周知的多播组和 UDP 端口发送包含其会话描述信息的多播数据报。在 MBone 上的站点运行一个叫 sdr 的程序去接收这些声明。这个程序做许多工作：不仅接收会话声明，而且提供一个交互的用户界面，显示信息并让用户发送声明。

在这一节，我们开发一个简单的程序，它只接收那些会话声明，我们用它来作为简单的多播接收程序的例子。我们的目的是展示多播接收的简单性，而不是深入到这个应用程序的细节。

图 19.11 给出了我们的 main 程序，它接收定期的 SAP/SDP 声明。

众所周知的名字和众所周知的端口

第 2～3 行　分配给 SAP 声明的多播地址是 224.2.127.254，它的名字是 sap.mcast.net。所有的众所周知多播地址(见 ftp://ftp.isi.edu/in-notes/iana/assignments/multicast-addresses)都在 DNS 中的 mcast.net 层次下。众所周知的 UDP 端口号是 9875。

创建 UDP 套接口

第 12～17 行　我们调用自己的 udp_client 函数去查看名字和端口，它将信息填入合适的套接口地址结构中。如果没有指定命令行参数，我们就使用缺省值；否则我们从命令行参数中取得多播地址、端口号和接口名字。

```
1 #include    "unp.h"
2 #define     SAP_NAME      "sap.mcast.net"    /* default group name and port */
3 #define     SAP_PORT      "9875"
4 void  loop(int, socklen_t);
5 int
6 main(int argc, char **argv)
```

```
 7 {
 8     int          sockfd;
 9     const int on = 1;
10     socklen_t salen;
11     struct sockaddr *sa;
12     if (argc == 1)
13         sockfd = Udp_client(SAP_NAME, SAP_PORT, (void **) &sa, &salen);
14     else if (argc == 4)
15         sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);
16     else
17         err_quit("usage: mysdr <mcast-addr> <port#> <interface-name>");
18     Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
19     Bind(sockfd, sa, salen);
20     Mcast_join(sockfd, sa, salen, (argc == 4) ? argv[3] : NULL, 0);
21     loop(sockfd, salen);     /* receive and print */
22     exit(0);
23 }
```

图19.11 接收SAP/SDP声明的main程序[mysdr/main.c]

捆绑端口

第18～19行 我们设置SO_REUSEADDR套接口选项以允许这个程序的多个实例在单台主机上运行，然后捆绑端口到这个套接口上。通过捆绑多播地址到套接口上，我们防止这个套接口接收其他的可能被其端口接收的UDP数据报。捆绑这个多播地址不是必须的，但它提供了由内核来过滤我们不想要的分组的功能。

加入多播组

第20行 我们调用mcast_join函数来加入这个组。如果接口名字已被作为命令行参数指定，它将会传递给我们的函数；否则我们让内核去选择在哪个接口上加入组。

第21行 我们调用在图19.12中给出的loop函数来读取和输出所有的声明。

```
 1 #include    "unp.h"
 2 void
 3 loop(int sockfd, socklen_t salen)
 4 {
 5     char     buf[MAXLINE+1];
 6     socklen_t    len;
 7     ssize_t n;
 8     struct sockaddr    *sa;
 9     struct sap_packet {
10         uint32_t   sap_header;
11         uint32_t   sap_src;
12         char       sap_data[1];
13     } *sapptr;
14     sa = Malloc(salen);
15     for ( ; ; ) {
16         len = salen;
17         n = Recvfrom(sockfd, buf, MAXLINE, 0, sa, &len);
18         buf[n] = 0;    /* null terminate */
19         sapptr = (struct sap_packet *) buf;
20         if ( (n == 2 * sizeof(uint32_t)) <= 0)
```

```
21              err_quit("n = %d", n);
22          printf("From %s\n%s\n", Sock_ntop(sa, len), sapptr->sap_data);
23      }
24  }
```

图 19.12 接收并输出 SAP/SDP 声明的循环[mysdr/loop.c]

数据报格式

第 9～13 行 sap_packet 描述了 SDP 数据报：一个 32 位的 SAP 头部，后跟一个 32 位的源地址，再后跟实际的声明。声明仅仅是 ISO 8859-1 的文本行，不能超过 1024 字节。在每个 UDP 数据报中只允许有一个会话声明。

读入 UDP 数据报，输出发送者和内容

第 15～23 行 recvfrom 等待下一个到我们套接口的 UDP 数据报。当一个数据报到后，我们在缓冲区末端加一个空字节并且跳过两个头部字段，接着输出结果。我们还输出了发送多播声明的发送者的 IP 地址和端口号。

图 19.3 给出了我们程序的一些典型输出。

```
solaris % mysdr
From 128.102.84.134/2840
v=0
o=shuttle 3050400397 3051818822 IN IP4 128.102.84.134
s=NASA - Shuttle STS-79 Mission Coverage
i=Pre-launch and mission coverage of Shuttle Mission STS-79.  Launch expected 9/16/96
  with a mission duration anticipated of 9-10 days.  STS-79 is the 4th in a series of joint
  docking missions between the Shuttle and the MIR Space Station. This session is being
  offered by NASA - Ames Research Center as a public service to the MBone community.
u=http://www-pao.ksc.nasa.gov/kscpao/kscpao.htm
p=NASA ARC Digital Video Lab (415) 604-6145
e=NASA ARC Digital Video Lab <mallard@mail.arc.nasa.gov>
c=IN IP4 224.2.86.28/127
t=3051608400 3052472400
m=audio 19432 RTP/AVP 0
m=video 61192 RTP/AVP 31
```

图 19.13 典型的 SAP/SDP 声明

这个声明描述了在 MBone 上航天飞机任务的 NASA 报道。SDP 会话描述包含很多形如 type=value 的行，type 总是一个字符而且是区分大小写的。value 是一个依赖于 type 的有结构的文本串。等号两边不允许有空格。

v=0 是版本。

o=是信息源。shuttle 是用户名，3050400397 是会话 ID，3051818822 是这个声明的版本号，IN 是网络的类型，IP4 是地址类型，128.102.84.134 是地址。由用户名、会话 ID、网络类型、地址类型和地址组成的五元组为会话构成了一个唯一标识符。

s=定义会话名字，i=是有关会话的信息。我们把信息按每 80 字节折了行。u=为有关会话的详细信息提供一个 URI(统一资源标识符)，p=和 e=则提供会议负责人的电话号码和电子邮件地址。

c=提供连接信息，在这个例子中它表明了连接是基于 IP 的，用 IPv4，多播地址是 224.2.86.28，TTL 值为 127。t=提供起始时间和停止时间，都是 NTP 单位的，从 1900 年 1 月 1

日UTC时间以来的秒数。m=行是介质声明。这两行的第一行指明了音频在19432端口上，格式是RTP即实时传输协议(Real-time Transport Protocol)，使用AVP即音频/视频轮廓(Audio/Video Profile)，有效负荷类型为0(是一个PCM编码的信令通道音频，8Khz采样频率)。下一个m=行指明了视频在61192端口，RTP/AVP有效负荷类型31是ITU H.261视频格式。注意，音频和视频都多播到同一个地址(224.2.86.28)但是不同的端口。这意味着如果一台主机只想要一种数据，当它加入这个组后，两种数据都会被IP层接收，再由UDP层把不要的数据去掉。

19.9 发送和接收

前一节中的MBone会话声明程序只接收多播数据报。我们现在写一个简单的程序能发送和接收多播数据报。我们的程序包含两部分。第一部分每5秒种向指定的组发送多播数据报，其中有发送者的主机名和进程ID。第二部分是一个无限循环，加入由第一部分发往的多播组，输出接收到的每个数据报(包含发送者的主机名和进程ID)。这可以让我们在局域网的多台主机上启动该程序，很容易地看出哪个主机在接收哪个发送者的数据报。

图19.14给出了main函数。

```
1 #include    "unp.h"
2 void  recv_all(int, socklen_t);
3 void  send_all(int, SA *, socklen_t);
4 int
5 main(int argc, char **argv)
6 {
7     int     sendfd, recvfd;
8     const int on = 1;
9     socklen_t salen;
10    struct sockaddr *sasend, *sarecv;
11    if (argc != 3)
12        err_quit("usage: sendrecv <IP-multicast-address> <port#>");
13    sendfd = Udp_client(argv[1], argv[2], (void **) &sasend, &salen);
14    recvfd = Socket(sasend->sa_family, SOCK_DGRAM, 0);
15    Setsockopt(recvfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
16    sarecv = Malloc(salen);
17    memcpy(sarecv, sasend, salen);
18    Bind(recvfd, sarecv, salen);
19    Mcast_join(recvfd, sasend, salen, NULL, 0);
20    Mcast_set_loop(sendfd, 0);
21    if (Fork() == 0)
22        recv_all(recvfd, salen);          /* child -> receives */
23    send_all(sendfd, sasend, salen);      /* parent -> sends */
24 }
```

图19.14 创建套接口，fork并启动发送者与接收者[mcast/main.c]

我们创建两个套接口，一个用于发送，一个用于接收。我们希望给接收套接口捆绑多播组和端口，譬如说239.255.1.2端口8888。(回想一下，我们可以只捆绑通配IP地址和端口

8888,但捆绑多播地址可防止别的到端口 8888 的数据报也被本套接口接收。)我们接着让接收套接口加入多播组。发送套接口会向同一多播地址和端口(也就是 239.255.1.2 端口 8888)发送数据报。但是如果我们试着用单个套接口发送和接收,bind 的源协议地址是 239.255.1.2.8888(使用 netstat 记法),sendto 的目的协议地址也是 239.255.1.2.8888。但是,捆绑在套接口上的源协议地址将成为 UDP 数据报的源 IP 地址,而 RFC 1122[Braden 1989]却禁止源 IP 地址是多播地址或广播地址的 IP 数据报(见习题 19.2)。因此,我们创建了两个套接口:一个用于发送,一个用于接收。

创建发送套接口

第 13 行 我们的 udp_client 函数创建发送套接口,处理指定多播地址和端口号的两个命令行参数。这个函数也返回可用于调用 sendto 的一个套接口地址结构及它的长度。

创建接收套接口并捆绑多播地址和端口

第 14～18 行 我们用与用于发送套接口相同的地址族创建接收套接口。我们设置 SO_REUSEADDR 套接口选项以允许这个程序的多个实例同时在一台主机上运行。我们接着给这个套接口分配一个套接口地址结构的空间,从发送套接口地址结构中复制其内容(它的地址和端口是从命令行参数中取得的),然后把这个多播地址和端口号捆绑到接收套接口上。

加入多播组并关掉回馈

第 19～20 行 我们调用自己的 mcast_join 函数在接收套接口上加入多播组,并调用 mcast_set_loop 函数禁止发送套接口上的回馈特性。对于加入,我们指定接口名字为空指针,接口索引为 0,从而通知内核去选择接口。

fork 并调用相应函数

第 21～23 行 我们 fork 后,子进程就是接收循环,父进程就是发送循环。

我们的每 5 秒发送一个多播数据报的 send_all 函数在图 19.15 中给出。main 函数将套接口描述字、指向包含多播地址和端口的套接口地址结构的指针及该结构的长度作为参数传给 send_all。

获取主机名并形成数据报内容

第 9～11 行 我们从 uname 函数获得主机名并形成一个包含主机名和进程 ID 的输出行。

发送数据报,接着去 sleep

第 12～15 行 我们发送一个数据报后,接着 sleep 5 秒。

无限循环接收的 recv_all 函数在图 19.16 中给出。

分配套接口地址结构

第 9 行 分配一个套接口地址结构,用于接收调用 recvfrom 函数时返回的发送者的协议地址。

读入并输出数据报

第 10～15 行 每一个数据报由 recvfrom 读入,以空字符结尾后输出。

```
1 #include    "unp.h"
2 #include    <sys/utsname.h>
```

```
3 #define    SENDRATE    5        /* send one datagram every 5 seconds */
4 void
5 send_all(int sendfd, SA *sadest, socklen_t salen)
6 {
7     static char        line[MAXLINE];          /* hostname and process ID */
8     struct utsname     myname;
9     if (uname(&myname) < 0)
10        err_sys("uname error");
11    snprintf(line, sizeof(line), "%s, %d\n", myname.nodename, getpid());
12    for ( ; ; ) {
13        Sendto(sendfd, line, strlen(line), 0, sadest, salen);
14        sleep(SENDRATE);
15    }
16 }
```

图 19.15 每5秒发送一个多播数据报[mcast/send.c]

```
1 #include    "unp.h"
2 void
3 recv_all(int recvfd, socklen_t salen)
4 {
5     int          n;
6     char         line[MAXLINE+1];
7     socklen_t len;
8     struct sockaddr *safrom;
9     safrom = Malloc(salen);
10    for ( ; ; ) {
11        len = salen;
12        n = Recvfrom(recvfd, line, MAXLINE, 0, safrom, &len);
13        line[n] = 0;     /* null terminate */
14        printf("from %s: %s", Sock_ntop(safrom, len), line);
15    }
16 }
```

图 19.16 接收所有到达我们已加入的组的多播数据报[mcast/recv.c]

19.10 SNTP:简单网络时间协议

NTP(网络时间协议)是一个跨越广域网或局域网的复杂的同步时钟协议,它通常可获得毫秒级的精度。RFC 1305[Mills 1992]详细描述了这个协议,RFC 2030[Mills 1996]描述了SNTP(一个简化的版本),目的是为了那些不需要完整NTP实现复杂性的主机。通常让局域网上的若干台主机通过因特网与其他的NTP主机同步时钟,接着在局域网上用广播或多播来发布时间。

在这一节我们将开发一个SNTP客户程序,它监听与它连接的所有网络上的NTP广播或多播,接着输出NTP数据报中的时间和主机的当前时间之差。我们并不试着去修正当前时间,那要取得超级用户的权限,尽管那只是对代码附加的小改动。

ntp.h文件如图19.17所示,它包含了一些NTP数据报格式的基本定义。

```
1 #define    JAN_1970    2208988800UL    /* 1970 - 1900 in seconds */
2 struct l_fixedpt {                    /* 64-bit fixed-point */
```

```
3     uint32_t  int_part;
4     uint32_t  fraction;
5 };
6 struct s_fixedpt {              /* 32-bit fixed-point */
7     u_short   int_part;
8     u_short   fraction;
9 };
10 struct ntpdata {                /* NTP header */
11     u_char        status;
12     u_char        stratum;
13     u_char        ppoll;
14     int           precision:8;
15     struct s_fixedpt    distance;
16     struct s_fixedpt    dispersion;
17     uint32_t  refid;
18     struct l_fixedpt    reftime;
19     struct l_fixedpt    org;
20     struct l_fixedpt    rec;
21     struct l_fixedpt    xmt;
22 };
23 #define VERSION_MASK      0x38
24 #define MODE_MASK         0x07
25 #define MODE_CLIENT       3
26 #define MODE_SERVER       4
27 #define MODE_BROADCAST    5
```

图 19.17 ntp.h 头文件:NTP 数据报格式和定义[ssntp/ntp.h]

第 2～22 行 l_fixedpt 定义了 NTP 用于时间戳的 64 位定点值,s_fixedpt 定义了也是 NTP 用的 32 位定点值。ntpdata 结构是 48 字节的 NTP 数据报格式。

图 19.18 给出了 main 函数。

```
1 #include    "sntp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int       sockfd;
6     char      buf[MAXLINE];
7     ssize_t   n;
8     socklen_t salen, len;
9     struct ifi_info *ifi;
10    struct sockaddr *mcastsa, *wild, *from;
11    struct timeval now;
12    if (argc != 2)
13        err_quit("usage: ssntp <IPaddress>");
14    sockfd = Udp_client(argv[1], "ntp", (void **) &mcastsa, &salen);
15    wild = Malloc(salen);
16    memcpy(wild, mcastsa, salen);  /* copy family and port */
17    sock_set_wild(wild, salen);
18    Bind(sockfd, wild, salen);   /* bind wildcard */
19 #ifdef  MCAST
20        /* obtain interface list and process each one */
```

```
21      for (ifi = Get_ifi_info(mcastsa->sa_family, 1); ifi != NULL;
22           ifi = ifi->ifi_next)
23          if (ifi->ifi_flags & IFF_MULTICAST) {
24              Mcast_join(sockfd, mcastsa, salen, ifi->ifi_name, 0);
25              printf("joined %s on %s\n",
26                      Sock_ntop(mcastsa, salen), ifi->ifi_name);
27          }
28  #endif
29      from = Malloc(salen);
30      for ( ; ; ) {
31          len = salen;
32          n = Recvfrom(sockfd, buf, sizeof(buf), 0, from, &len);
33          Gettimeofday(&now, NULL);
34          sntp_proc(buf, n, &now);
35      }
36  }
```

图 19.18 main 函数[ssntp/main.c]

获得多播 IP 地址

第 12～14 行 执行本程序时,用户必须使用命令行参数指定要加入的多播地址。IPv4 里这将是 224.0.1.1 或名字 ntp.mcast.net。IPv6 里则是网点局部范围内 NTP 的 ff05::101。我们的 udp_client 函数给正确类型(IPv4 或 IPv6)的套接口地址结构分配空间,并将多播地址和端口号存入此结构。如果这个程序在不支持多播的主机上运行,那么可以指定任何 IP 地址,因为我们只用到了这个结构的地址族和端口号。注意,我们的 udp_client 函数并不把地址捆绑到套接口上;它只是创建套接口并填写套接口地址结构。

把通配地址捆绑到套接口

第 15～18 行 我们给另一个套接口地址结构分配空间,并用由 udp_dient 填充的结构通过拷贝填写它。这就设置了地址族和端口。我们调用自己的 sock_set_wild 函数设置 IP 地址为通配地址,然后调用 bind。0

获得接口列表

第 20～22 行 我们的 get_ifi_info 函数返回所有接口和地址的信息。我们查询的地址族是从由 udp_client 基于命令行参数填充的套接口地址结构中取得的。

加入多播组

第 23～27 行 我们调用 mcast_join 函数把每一个能多播的接口都加入由命令行参数指定的多播组。所有的加入操作都是在程序使用的套接口上进行的。我们以前提到过,通常一个套接口有 20 个加入的限制,但是几乎没有多宿主机有那么多的接口。

读入并处理所有的 NTP 数据报

第 29～35 行 另一个套接口地址结构被分配用于保存由 recvfrom 返回的地址后,程序接着进入了无穷循环,读入本主机收到的所有 NTP 数据报并调用 sntp_proc 函数去处理。由于套接口绑定了通配地址,又由于多播组在所有能多播的接口上都加入了,因此这个套接口应能收到本主机接收的任何单播、广播或多播 NTP 数据报。在调用 sntp_proc 前我们调用 gettimeofday 取得当前时间,因为 sntp_proc 需计算包含在数据报中的时间和当前时间之差。

在图19.19中给出的sntp_proc函数处理实际的NTP数据报。

检查数据报的合法性

第10～21行 我们首先检查数据报的大小，接着输出版本、模式和服务器层次。如果模式是MODE_CLIENT，那么这个数据报是一个客户请求，而不是一个应答，于是我们忽略它。

从NTP数据报获取发送时间

第22～34行 在NTP数据报中，我们感兴趣的是xmt(发送时间戳)，它是服务器发送这个数据报时的64位定点时间。由于NTP时间戳从1900年开始计秒数而Unix从1970年开始计秒数，因此我们首先从整数部分中减去JAN_1970(70年的秒数)。

小数部分是一个32位的0到4 294 967 295之间(包括上下限)的无符号整数。它从32位的整数(useci)复制到一个双精度的浮点变量(usecf)，接着被4 294 967 296(2^{32})除。结果大于等于0.0，小于1.0。我们再将结果乘以1 000 000即一秒中的微秒数，并将结果作为一个32位的无符号整数存在useci变量中。

```
1 #include    "sntp.h"
2 void
3 sntp_proc(char *buf, ssize_t n, struct timeval *nowptr)
4 {
5     int     version, mode;
6     uint32_t nsec, useci;
7     double  usecf;
8     struct timeval curr, diff;
9     struct ntpdata *ntp;
10    if (n < sizeof(struct ntpdata)) {
11        printf("\npacket too small: %d bytes\n", n);
12        return;
13    }
14    ntp = (struct ntpdata *) buf;
15    version = (ntp->status & VERSION_MASK) >> 3;
16    mode = ntp->status & MODE_MASK;
17    printf("\nv%d, mode %d, strat %d, ", version, mode, ntp->stratum);
18    if (mode == MODE_CLIENT) {
19        printf("client\n");
20        return;
21    }
22    nsec = ntohl(ntp->xmt.int_part) - JAN_1970;
23    useci = ntohl(ntp->xmt.fraction);    /* 32-bit integer fraction */
24    usecf = useci;                       /* integer fraction -> double */
25    usecf /= 4294967296.0;               /* divide by 2**32 -> [0, 1.0) */
26    useci = usecf * 1000000.0;           /* fraction -> parts per million */
27    curr = *nowptr;                      /* make a copy as we might modify it below */
28    if ((diff.tv_usec = curr.tv_usec - useci) < 0) {
29        diff.tv_usec += 1000000;
30        curr.tv_sec--;
31    }
32    diff.tv_sec = curr.tv_sec - nsec;
33    useci = (diff.tv_sec * 1000000) + diff.tv_usec; /* diff in microsec */
```

```
34      printf("clock difference = %d usec\n", usecl);
35  }
```

图 19.19 sntp_proc 函数：处理 SNTP 数据报[ssntp/sntp_proc.c]

这将是在 0～999,999 之间的微秒数(见习题 19.8)。我们把它变为微秒是因为由 gettimeofday 返回的 Unix 时间戳包含两个整数：从 1970 年 1 月 1 日 UTC 时间以来的秒数以及微秒数。我们然后计算并输出主机的当前时间与 NTP 服务器当前时间以微秒为单位的时间差。

我们程序没有考虑的一件事是服务器和客户之间的网络延迟。但是我们假定 NTP 数据报通常作为局域网上的广播或多播数据报接收，这种情况下，网络延迟应该只有几个毫秒。

我们在主机 solaris 上运行本程序，在主机 bsdi 上的 NTP 服务器每 64 秒多播 NTP 数据报到所在以太网的情况下，会有如下的输出：

```
solaris # ssntp 224.0.1.1
joined 224.0.1.1.123 on lo0
joined 224.0.1.1.123 on le0
v3, mode 5, start 3, clock difference = 621  usec
v3, mode 5, start 3, clock difference = 1205 usec
v3, mode 5, start 3, clock difference = 1664 usec
v3, mode 5, start 3, clock difference = 2291 usec
v3, mode 5, start 3, clock difference = 2942 usec
v3, mode 5, start 3, clock difference = 3558 usec
```

我们先停止本主机上正常的 NTP 服务器再运行该程序，这样程序开始运行时与服务器主机的时间很接近。我们看到这个主机每 64 秒约慢 600 微秒，即每天约慢 810 毫秒。

19.11 SNTP(续)

现在用额外的特性扩展我们的 SNTP 例子。首先，当程序启动时，它对每个单播地址、每个广播地址和所加入多播组的每个接口分别创建一个套接口，而不像在 19.10 节中那样只用一个套接口。这样做的目的是确定我们收到的数据报的目的地址。其次，当程序启动时，它从所有连接的接口上广播和多播一个 SNTP 客户请求，得到一个起始时间差估计。

警告：这个源代码不是编写 SNTP 客户程序推荐的方法。我们的目的是更多地了解广播和多播，特别是在多宿主机或路由器上，以及广播和多播的回馈特性。我们选 SNTP 来展示这些特性的原因是它是一个有用的现实存在的应用系统。我们的客户程序在启动时从所有能广播和多播的接口发送一个 SNTP 客户请求，其目的是给出某些程序在启动时是如何执行资源发现的。对于 SNTP 客户程序我们并不建议这样做，更好的技术是只去监听服务器的广播或多播数据报，就像在 19.10 节中的那样。

图 19.20 是组成我们程序的函数功能的概览。我们首先调用 16.6 节中的 get_ifi_info 函数来获得接口的列表。对每个接口，我们创建一个 UDP 套接口并 bind 该接口的单播地址，创建另一个 UDP 套接口并 bind 该接口的广播地址，再创建一个 UDP 套接口并 bind NTP 的多播组(224.0.1.1)，然后在这个接口上加入这个多播组。我们的 sntp_send 函数从

每个能广播的套接口向所连接子网上的所有NTP服务器发送一个广播请求，以获取它们的当前时间，接着从所有的能多播的套接口发送多播请求。这第一批发送的目的是找到所连接子网上的所有NTP服务器并得到一个当前时间的初始估计。

程序接着进入了一个无穷循环read_loop，读入到来的任何应答。我们首先期望sntp_send发出的数据报的应答，然后再接收从所连接子网上的NTP服务器定期发送的内容。大多数NTP服务器每隔64秒广播或多播一个数据报。sntp_proc同图19.19中的函数相同，处理接收到的NTP数据报。

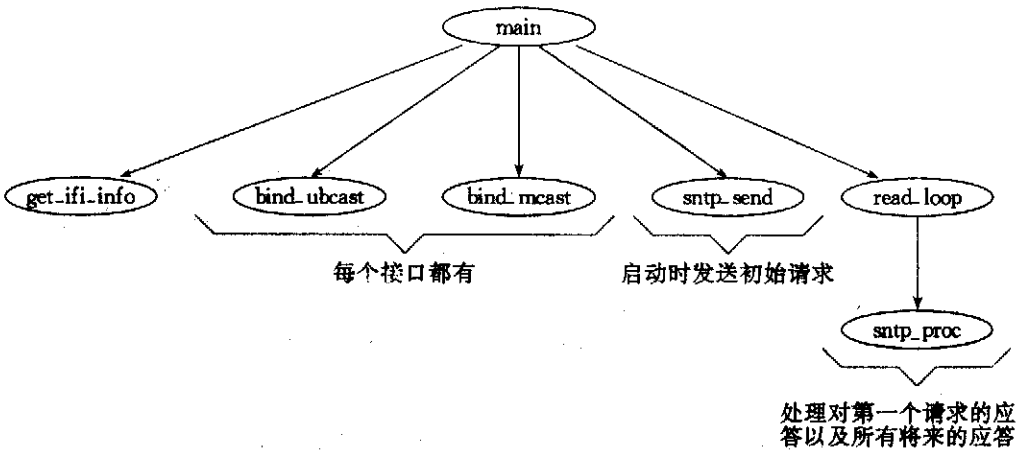


图19.20 SNTP客户程序各函数概览

我们从图19.21开始讲述我们的代码，图19.21是我们的头文件sntp.h，我们所有的程序都包含它。它包含的ntp.h头文件与图19.17中给出的相同。

定义Addrs结构

第3～12行 我们定义了一个结构类型Addrs，它包含我们需要的为给定接口返回的每个地址信息。由于我们的get_ifi_info函数返回的每个ifi_info结构可能有两个地址(如一个单播地址和一个广播地址)，因此我们要维护两个Addrs结构，每个地址一个。成员addr_sa指向由get_ifi_info返回的套接口地址结构，addr_salen是它的长度。成员addr_ifname中保存指向接口名的指针，它将用于多播。我们要给每个地址创建一个套接口，其描述字就存放在addr_fd中。我们还要知道一个套接口绑定的是广播地址还是多播地址，这个信息存放在成员addr_flags中。

```
1 #include    "unpifi.h"
2 #include    "ntp.h"
3 #define     MAXNADDRS   128          /* max # of addresses to bind() */
4 typedef struct {
5     struct sockaddr  *addr_sa;         /* ptr to bound address */
6     socklen_t        addr_salen;     /* socket address length */
7     const char       *addr_ifname;   /* interface name, for multicasting */
8     int              addr_fd;        /* socket descriptor */
9     int              addr_flags;     /* ADDR_xxx flags (see below) */
10 } Addrs;
11 Addrs addrs[MAXNADDRS];            /* the actual array of structs */
```

```
12 int          naddrs;                 /* index into the array */
13 #define      ADDR_BCAST   1
14 #define      ADDR_MCAST   2
15 const int    on;                     /* for setsockopt() */
16                                      /* function prototypes */
17 void   bind_mcast(const char *, SA *, socklen_t, int);
18 void   bind_ubcast(SA *, socklen_t, int, int, int);
19 void   read_loop(void);
20 void   sntp_proc(char *, ssize_t nread, struct timeval *);
21 void   sntp_send(void);
```

图19.21 sntp.h头文件[sntp/sntp.h]

图19.22给出了main函数。

获取众所周知的多播地址和端口

第10～14行 命令行参数通常是名字ntp.mcast.net,它被映射到多播地址224.0.1.1。服务的名字是ntp。我们的udp_client函数给正确类型(IPv4或者IPv6)的套接口地址结构分配空间,并将多播地址和端口号存入这个结构。如果这个程序在一个不支持多播的主机上运行,那么我们可以指定任意一个IP地址,因为只用到这个结构中的端口号。我们接着关闭所返回的套接口,因为我们调用udp_client的目的仅仅为了填充套接口地址结构。

获取接口列表

第15～17行 我们的get_ifi_info函数返回所有接口和地址的信息。我们使用的地址族是从套接口地址结构中得到的,这个结构是由udp_client基于命令行参数填入的。传给get_ifi_info的第二个参数是1,这是通知它返回有关别名地址的信息。我们接着传递IFI_ALIAS标志给我们的bind_XXX函数们,让每个函数决定如何处理别名地址,我们很快就会看到这种情形。

处理每个单播和广播地址

第18～22行 我们给每个地址调用函数bind_ubcast一次或两次。第一次调用的最后一个参数是0,表明第一个参数指向的是单播地址;第二次调用时为1,表明第一个参数指向的是广播地址。

```
1 #include    "sntp.h"
2 const int   on = 1;              /* for setsockopt() flags */
3 int
4 main(int argc, char **argv)
5 {
6     int     sockfd, port;
7     socklen_t salen;
8     struct ifi_info *ifi;
9     struct sockaddr *mcastsa, *wild;
10    if (argc != 2)
11        err_quit("usage: sntp <IPaddress>");
12    sockfd = Udp_client(argv[1], "ntp", (void **) &mcastsa, &salen);
13    port = sock_get_port(mcastsa, salen);
14    Close(sockfd);
15        /* obtain interface list and process each one */
```

```
16      for (ifi = Get_ifi_info(mcastsa->sa_family, 1); ifi != NULL;
17           ifi = ifi->ifi_next) {
18          bind_ubcast(ifi->ifi_addr, salen, port,
19                      ifi->ifi_myflags & IFI_ALIAS, 0);      /* unicast */
20          if (ifi->ifi_flags & IFF_BROADCAST)
21              bind_ubcast(ifi->ifi_brdaddr, salen, port,
22                          ifi->ifi_myflags & IFI_ALIAS, 1);   /* bcast */
23  #ifdef  MCAST
24          if (ifi->ifi_flags & IFF_MULTICAST)
25              bind_mcast(ifi->ifi_name, mcastsa, salen,
26                         ifi->ifi_myflags & IFI_ALIAS);      /* mcast */
27  #endif
28      }
29      wild = Malloc(salen);   /* socket address struct for wildcard */
30      memcpy(wild, mcastsa, salen);
31      sock_set_wild(wild, salen);
32      bind_ubcast(wild, salen, port, 0, 0);
33      sntp_send();      /* send first queries */
34      read_loop();      /* never returns */
35  }
```

图 19.22 main 函数[sntp/main.c]

加入多播组

第 23～27 行 如果接口能多播，我们就对每一个单播地址也调用 bind_mcast 函数，因为我们希望在每个接口上加入多播组。

捆绑通配地址

第 29～32 行 处理完所有接口信息后，我们分配另一个套接口地址结构并从由 udp_client 填入的那个套接口地址结构中拷入内容。我们接着调用 sock_set_wild 函数将合适的通配地址存入这个结构。bind_ubcast 创建一个套接口并将通配地址捆绑在其上。这用来处理那些目的地址是其他地址如 255.255.255.255 的数据报，这些地址我们无法捆绑。

发送初始请求并读入所有随后的应答

第 33～34 行 sntp_send 从所有能广播的接口上广播一个 SNTP 请求，并在所有能多播的接口上多播一个 SNTP 请求。read_loop 接着读入对这些请求的所有应答以及之后收到的任何 NTP 广播或多播数据报。

图 19.23 展示了将给主机 bsdi 创建的套接口。从图 1.16 中我们知道这个主机其实是有两个以太网接口的路由器。在图 16.6 后的例子中，我们给出了该主机的各个接口及它们的单播和广播 IP 地址，不过现在我们假设没有别名地址。我们将创建 9 个套接口，并 bind 同一个端口 9 次。

我们用 u、b 或 m 标记了底端的 8 个套接口，它们相应于捆绑了单播地址、广播地址或多播地址的套接口。我们给出了将由每个套接口接收的分组的目的 IP 地址，并指出最右边的捆绑 0.0.0.0 的套接口可以接收目的地址是其他 IP 地址(通常为 255.255.255.255)的任何接口上的分组。

我们的 bind_ubcast 函数在图 19.24 中给出。所有的单播地址、所有的广播地址和通配地址都调用它。在图 19.22 的三次调用中，只有在广播地址时，最后的参数才为 1。

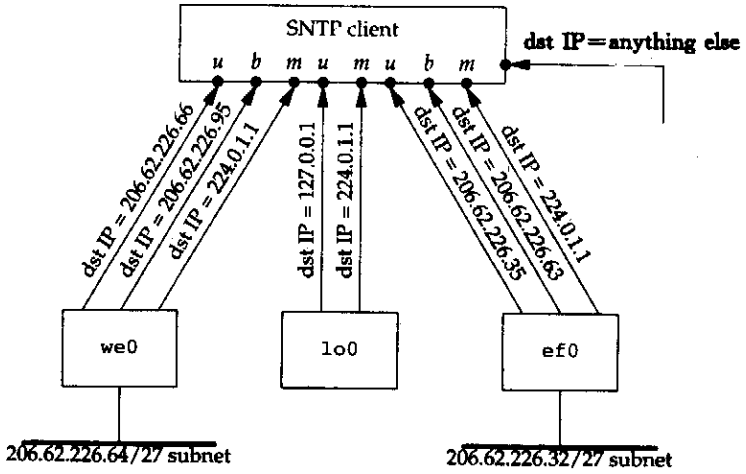


图 19.23 在主机 bsdi 上由 SNTP 客户创建的 9 个套接口

```
1 #include    "sntp.h"
2 void
3 bind_ubcast(struct sockaddr *sabind, socklen_t salen, int port,
4               int alias, int bcast)
5 {
6     int         i, fd;
7         /* first see if we've already bound this address */
8     for (i = 0; i < naddrs; i++) {
9         if (sock_cmp_addr(addrs[i].addr_sa, sabind, salen) == 0)
10            return;
11    }
12    fd = Socket(sabind->sa_family, SOCK_DGRAM, 0);
13    sock_set_port(sabind, salen, port);
14    Setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
15    printf("binding %s\n", Sock_ntop(sabind, salen));
16    if (bind(fd, sabind, salen) < 0) {
17        if (errno == EADDRINUSE) {
18            printf("  (address already in use)\n");
19            close(fd);
20            return;
21        } else
22            err_sys("bind error");
23    }
24    addrs[naddrs].addr_sa = sabind;      /* save ptr to sockaddr{} */
25    addrs[naddrs].addr_salen = salen;
26    addrs[naddrs].addr_fd = fd;
27    if (bcast)
28        addrs[naddrs].addr_flags = ADDR_BCAST;
29    naddrs++;
30 }
```

图 19.24 bind_ubcast 函数:给单播或多播地址创建套接口[sntp/bind_ubcast.c]

看地址是否已经被捆绑

第7～11行 我们首先查看这个地址是否已经被捆绑。这种情况对单播地址是不会发生的，然而别名共享同一个广播地址时，我们只想捆绑广播地址一次。

创建套接口并捆绑地址

第12～23行 我们创建一个UDP套接口，设置端口，接着设置SO_REUSEADDR套接口选项(由于我们给每个地址捆绑同样的端口)，然后bind地址到套接口。在其中保存了bind所用端口的套接口地址结构是由get_ifi_info填入后返回的。我们允许bind失败，这种情况只在NTP守护进程本身就在该主机上运行时才发生。

保存本套接口的信息

第24～29行 这些信息被存入了我们的Addrs结构，如果本地址是一个广播地址，则再包含一个标志。

对于能多播的接口，我们需要创建一个套接口，捆绑众所周知的端口，接着在这个接口上加入多播组。这由我们图19.25中所示的bind_mcast函数来完成。

```
1 #include    "sntp.h"
2 void
3 bind_mcast(const char *ifname, SA *mcastsa, socklen_t salen, int alias)
4 {
5 #ifdef  MCAST
6     int     fd;
7     struct sockaddr *msa;
8     if (alias)
9         return;             /* only one mcast join per interface */
10    printf("joining %s on %s\n", Sock_ntop_host(mcastsa, salen), ifname);
11    fd = Socket(mcastsa->sa_family, SOCK_DGRAM, 0);
12    Setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
13    Bind(fd, mcastsa, salen);
14    Mcast_join(fd, mcastsa, salen, ifname, 0);
15    addrs[naddrs].addr_sa = mcastsa;
16    addrs[naddrs].addr_salen = salen;
17    addrs[naddrs].addr_ifname = ifname;   /* save pointer, not string copy */
18    addrs[naddrs].addr_fd = fd;
19    addrs[naddrs].addr_flags = ADDR_MCAST;
20    naddrs++;
21 #endif
22 }
```

图19.25 bind_mcast函数：在接口上加入多播组[sntp/bind_mcast.c]

创建套接口并捆绑

第8～13行 如果这个地址是一个别名，我们立即返回，因为我们只需在每个接口上加入多播组一次而不管有多少单播地址是这个接口的别名。创建套接口后把多播组和众所周知的端口捆绑在其上。

加入多播组并保存信息

第14～20行 在这个接口上加入多播组后，我们在Addrs结构中保存信息。保存在成员addr_sa中的指针mcastsa指向一个套接口地址结构，它是由main函数中对udp_client函数的调用来分配的。每个能多播的接口的addr_sa成员指向同一个结构，这是没有问题的，因为这个结构从不修改。addr_ifname指向ifi_info结构中的接口名字字符串，这也是没有问题的，因为我们不会调用free_ifi_info函数去释放其内存空间。

下一个函数是sntp_send，它在图19.26中给出。它是在所有接口信息处理后和所有套接口创建后由main调用的。

```
1 #include    "sntp.h"
2 void
3 sntp_send(void)
4 {
5     int fd;
6     Addrs *aptr;
7     struct ntpdata    msg;
8         /* use the socket bound to 0.0.0.0/123 for sending */
9     fd = addrs[naddrs-1].addr_fd;
10    Setsockopt(fd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
11    bzero(&msg, sizeof(msg));
12    msg.status = (0 << 6) | (3 << 3) | MODE_CLIENT; /* see RFC 2030 */
13    for (aptr = &addrs[0]; aptr < &addrs[naddrs]; aptr++) {
14        if (aptr->addr_flags & ADDR_BCAST) {
15            printf("sending broadcast to %s\n",
16                   Sock_ntop(aptr->addr_sa, aptr->addr_salen));
17            Sendto(fd, &msg, sizeof(msg), 0,
18                   aptr->addr_sa, aptr->addr_salen);
19        }
20 #ifdef      MCAST
21            if (aptr->addr_flags & ADDR_MCAST) {
22                /* must first set outgoing i/f appropriately */
23                Mcast_set_if(fd, aptr->addr_ifname, 0);
24                Mcast_set_loop(fd, 0);          /* disable loopback */
25                printf("sending multicast to %s on %s\n",
26                       Sock_ntop(aptr->addr_sa, aptr->addr_salen),
27                       aptr->addr_ifname);
28                Sendto(fd, &msg, sizeof(msg), 0,
29                       aptr->addr_sa, aptr->addr_salen);
30        }
31 #endif
32    }
33 }
```

图19.26 sntp_send函数：广播和多播SNTP请求[sntp/sntp_send.c]

设置SO_BROADCAST选项并构造请求

第8～12行 我们调用sendto所用的是绑定了通配地址的套接口，而不是为此专门创建一个套接口。由于IP地址是通配的，内核于是使用外出接口的主单播地址作为UDP数据报的源IP地址。我们首先设置SO_BROADCAST套接口选项，然后构造SNTP请求：设置LI(闰年指示器)为0，版本为3，模式为MODE_CLIENT。这些是在客户请求中需要设置的

仅有的字段。

发送广播请求

第14～19行 如果地址是一个广播地址，请求就是广播请求。

发送多播请求

第21～30行 如果地址是一个多播地址，我们首先在套接口上指定多播数据报的外出接口（我们的mcast_set_if函数），接着禁止这个套接口上的回馈（我们的mcast_set_loop函数）。如果我们没有禁止回馈特性，我们会在所有的接收套接口上收到所发送数据报的多个复本（见习题19.11）。请求数据报然后被发送到多播地址。

这段代码给出了多播应用程序的一个典型范例，我们可以总结如下：

```
for(each interface){
    mcast_set_if(...);
    sendto(...);
}
```

数据报需从每个接口发出，因此在每个sendto之前必须用套接口选项设置发送接口。这对每一个需发送的数据报都要增加一个系统调用。另一个方法是给每个接口创建一个发送套接口，并在创建后就给每个套接口设置外出接口。我们将会在20.8节中看到，在调用sendmsg时，IPv6允许外出接口作为辅助数据指定，这样就会减少系统调用的数目。

也有一类多播应用程序并不关心外出接口，一次发送一个数据报，让内核去选择外出接口。

用图19.23作为我们的例子，应由sntp_send发送5个数据报：每个广播和多播套接口上一个。没有数据在单播套接口上发送。

下一个函数是read_loop，我们在图19.27中给出了其前半部分。这个函数在main函数中最后被调用，它只是从所有创建的套接口中读：出现在主机任一接口上的任何广播、多播或单播的NTP数据报。我们预期大多数NTP数据报是作为广播或多播数据报接收的，但是由sntp_send发送的查询对应的响应将是单播接收的。

分配缓冲区和套接口地址结构

第4～19行 我们分配两个缓冲区用于接收数据报，两个套接口地址结构用于存储相应的源地址，两个变量保存相应数据报的长度。相应于当前数据报的下标在currb中，相应于最近数据报的下标在lastb中。其中一个下标的值为0，另外一个为1。我们需要保存两个数据报的原因是，我们将会收到所有多播数据报的多份拷贝，我们要找出并忽略它们。即使在只有一个接口的主机上，一个多播NTP数据报能同时被绑定那个接口的套接口和绑定通配地址的套接口收到（见习题19.10）。

在这个多播的例子中会出现多份拷贝是因为我们多次捆绑同一个端口：每个接口一次再加通配地址一次。

```
1 #include    "sntp.h"
2 static int       check_loop(struct sockaddr *, socklen_t);
3 static int       check_dup(socklen_t);
```

```
 4 static char          buf1[MAXLINE], buf2[MAXLINE];
 5 static char         *buf[2] = { buf1, buf2 };
 6 struct sockaddr     *from[2];
 7 static ssize_t       nread[2] = { -1, -1 };
 8 static int          currb = 0, lastb = 1;

 9 void
10 read_loop(void)
11 {
12     int               nsel, maxfd;
13     Addrs             *aptr;
14     fd_set            rset, allrset;
15     socklen_t         len;
16     struct timeval    now;

17         /* allocate two socket address structures */
18     from[0] = Malloc(addrs[0].addr_salen);
19     from[1] = Malloc(addrs[0].addr_salen);

20     maxfd = -1;
21     for (aptr = &addrs[0]; aptr < &addrs[naddrs]; aptr++) {
22         FD_SET(aptr->addr_fd, &allrset);
23         if (aptr->addr_fd > maxfd)
24             maxfd = aptr->addr_fd;
25     }
```

图19.27 read_loop函数：前半部分[sntp/read_loop.c]

从第7章对SO_REUSEADDR套接口选项的讨论中知道，每个多播或广播数据报被递送到每个匹配的套接口，但是每个单播数据报只被递送到单个套接口。我们绑定通配地址的套接口匹配每一个收到的多播数据报。

如果我们不关心多播数据报的目的地址，我们可以仅创建一个套接口并捆绑多播地址(224.0.1.1)和众所周知的端口(123)，这样我们只接收单个拷贝。我们在图19.11中就是这样做的。使这个SNTP例子变复杂的原因是我们想知道每个数据报的目的地址。

给select准备描述字集

第20～25行 我们给select准备一个描述字集并计算我们能读的最大描述字。

read_loop函数的后半部分在图19.28中给出。它调用select等待一个或多个描述字可读，接着读数据报，处理SNTP数据报。

```
26     for ( ; ; ) {
27         rset = allrset;
28         nsel = Select(maxfd+1, &rset, NULL, NULL, NULL);

29         Gettimeofday(&now, NULL);       /* get time when select returns */

30         for (aptr = &addrs[0]; aptr < &addrs[naddrs]; aptr++) {
31             if (FD_ISSET(aptr->addr_fd, &rset)) {
32                 len = aptr->addr_salen;
33                 nread[currb] = recvfrom(aptr->addr_fd,
34                                         buf[currb], MAXLINE, 0,
35                                         from[currb], &len);
36                 if (aptr->addr_flags & ADDR_MCAST) {
37                     printf("%d bytes from %s", nread[currb],
38                             Sock_ntop(from[currb],len));
39                     printf(" multicast to %s", aptr->addr_ifname);
```

```
40                  } else if (aptr->addr_flags & ADDR_BCAST) {
41                      printf(" %d bytes from %s", nread[currb],
42                             Sock_ntop(from[currb], len));
43                      printf(" broadcast to %s",
44                             Sock_ntop(aptr->addr_sa, len));
45                  } else {
46                      printf(" %d bytes from %s", nread[currb],
47                             Sock_ntop(from[currb], len));
48                      printf(" to %s",
49                             Sock_ntop(aptr->addr_sa, len));
50                  }
51                  if (check_loop(from[currb], len)) {
52                      printf(" (ignored)\n");
53                      continue;       /* it's one of ours, looped back */
54                  }
55                  if (check_dup(len)) {
56                      printf(" (dup)\n");
57                      continue;       /* it's a duplicate */
58                  }
59                  sntp_proc(buf[lastb], nread[lastb], &now);
60                  if (--nsel <= 0)
61                      break;          /* all done with selectable descriptors */
62              }
63          }
64      }
65  }
```

图 19.28 read_loop 函数:后半部分[sntp/read_loop.c]

select 后接着获取当前时间

第 27～29 行 当 select 返回时,我们调用 gettimeofday 获取当前时间,这个时间用于计算与 SNTP 数据报中时间的差别。

确定哪些描述字可读

第 30～50 行 检查每个描述字看哪个是可读的,调用 recvfrom,输出数据报是如何接收的(广播、多播或单播)以及是从哪个接口上收到的。

检查回馈数据报

第 51～54 行 如果收到的数据报是所发送数据报的回馈,check_loop 函数就返回 1。我们在图 19.26 中禁止了多播的回馈,但是我们无法禁止我们发送的广播数据报的自动回馈。

检查重复数据报

第 55～58 行 由于我们能收到任何多播数据报和广播数据报的多份拷贝,我们的 check_dup 函数于是检查当前数据报是否与前一个数据报完全相同,如果是则返回 1。

处理 SNTP 数据报

第 59 行 这时,数据报既不是回馈拷贝也不是重复拷贝,所以我们调用 sntp_proc 函数(图 19.19)去处理 NTP 数据报。

我们的 check_loop 函数在图 19.29 中给出,它检查收到的数据报是不是我们发送的某个数据报的回馈拷贝。

```
66 int
67 check_loop(struct sockaddr *sa, socklen_t salen)
68 {
69     Addrs *aptr;
70     for (aptr = &addrs[0]; aptr < &addrs[naddrs]; aptr++) {
71         if (sock_cmp_addr(sa, aptr->addr_sa, salen) == 0)
72             return(1);      /* it is one of our addresses */
73     }
74     return(0);
75 }
```

图19.29 check_loop函数:如果数据报是我们发送的就返回1[sntp/read_loop.c]

检查发送者的地址

第70～74行 为了检查一个回馈拷贝,我们浏览一遍Addrs数组中的所有地址,并与收到的数据报的源地址比较。

我们的check_dup函数在图19.30中给出,它检查收到的数据报是否是前一个的完全拷贝。

检查长度、发送者地址和内容

第80～88行 如果两个数据报的长度、发送者的协议地址以及实际内容是相同的,我们就认为是同一个拷贝。如果不是一个重复数据报,就将下标currb和lastb对换。注意,在图19.28最后调用sntp_proc时传递的数据报是由lastb索引的,因为调用check_dup时,为了下一次调用recvfrom而对换了下标。

```
76 int
77 check_dup(socklen_t salen)
78 {
79     int         temp;
80     if (nread[currb] == nread[lastb] &&
81         memcmp(from[currb], from[lastb], salen) == 0 &&
82         memcmp(buf[currb], buf[lastb], nread[currb]) == 0) {
83         return(1);      /* it is a duplicate */
84     }
85     temp = currb;       /* swap currb and lastb */
86     currb = lastb;
87     lastb = temp;
88     return(0);
89 }
```

图19.30 check_dup函数:如果数据报重复则返回1[sntp/read_loop.c]

回想在图19.23中展示的9个套接口。图19.31给出了在主机bsdi上运行,并且主机soiaris上运行NTP服务器时的结果。前9行展示了创建的套接口、绑定在套接口上的IP地址以及多播组加入的接口。

接着的5行展示了由sntp_send发送的数据报:一个数据报到每个广播地址,一个数据报到每个多播地址。

再接着的6行展示了在不同套接口上收到的头5个数据报。这些数据报是作为sntp_send所发送请求的应答而收到的。第一个数据报被忽略了。因为它是我们发送的回馈拷贝(看一下源地址)。第二个数据报是从运行在主机solaris(206.62.26.33)上的NTP服务器来

的(模式4是图19.17中的MODE_SERVER),那是一个运行在第3层次的NTP版本3服务器。两个主机时间的差别是116毫秒。接着的3个数据报被忽略了:第一个是我们在另一个以太网上广播的某个数据报的回馈拷贝,另两个是在通配套接口上收到的两个广播数据报的回馈拷贝。我们发送的两个广播数据报都有一个拷贝被回馈,接着每个回馈数据报的一份拷贝被递送到绑定相应广播地址的套接口,另一份拷贝则被递送到通配套接口。两个广播数据报产生了4个回馈拷贝。我们可以关掉多播的回馈拷贝,但是无法对广播这样做。

最后的5行相应于4个收到的数据报,这4个数据报对应于从主机solaris上收到的一个NTP数据报。第一个数据报是作为一个多播数据报接收的,处理后产生一个117毫秒的时间差。后三个数据报是这个多播数据报的重复数据报,分别从另两个绑定相同地址和端口(224.0.1.1.123)的套接口和仅绑定相同端口(0.0.0.0.123)的通配套接口上收到的。

```
bsdi # sntp 224.0.1.1
binding 206.62.226.66.123          we0: Ethernet unicast
binding 206.62.226.95.123          we0: Ethernet broadcast
joining 224.0.1.1 on we0           we0: multicast
binding 206.62.226.35.123          ef0: Ethernet unicast
binding 206.62.226.63.123          ef0: Ethernet broadcast
joining 224.0.1.1 on ef0           ef0: multicast
binding 127.0.0.1.123              lo0: loopback
joining 224.0.1.1 on lo0           lo0: multicast
binding 0.0.0.0.123                wildcard
sending broadcast to 206.62.226.95.123
sending multicast to 224.0.1.1.123 on we0
sending broadcast to 206.62.226.63.123
sending multicast to 224.0.1.1.123 on ef0
sending multicast to 224.0.1.1.123 on lo0
48 bytes from 206.62.226.66.123 broadcast to 206.62.226.95.123 (ignored)
48 bytes from 206.62.226.33.123 to 206.62.226.35.123
v3,mode 4, strat 3, clock difference = -116013 usec
48 bytes from 206.62.226.35.123 broadcast to 206.62.226.63.123 (ignored)
48 bytes from 206.62.226.66.123 to 0.0.0.0.123 (ignored)
48 bytes from 206.62.226.35.123 to 0.0.0.0.123 (ignored)
48 bytes from 206.62.226.33.123 multicast to we0
v3,mode 5, strat 3, clock difference = -117043 usec
48 bytes from 206.62.226.33.123 multicast to ef0 (dup)
48 bytes from 206.62.226.33.123 multicast to lo0 (dup)
48 bytes from 206.62.226.33.123 to 0.0.0.0.123 (dup)
```

图19.31 sntp程序的输出

如果我们在这个环境下继续执行这个程序,我们会看到每64秒solaris上的NTP服务器多播一个NTP分组,我们的程序相应收到4个拷贝。第一个被处理,另外三个由于是重复的而被忽略。

19.12 小 结

多播应用进程从加入指派给它的多播组开始。这告诉IP层加入这个组,IP层再告诉数据链路层接收送到相应硬件层多播地址的多播帧。多播可以利用多数接口卡上有的硬件过

滤，而且过滤得越好，所收到的不需要的分组也越少。用这种硬件过滤减小了其他不参与同一应用系统的主机上的负载。

在广域网上的多播需要有多播能力的路由器和多播路由协议。在因特网上的所有路由器都能进行多播前，称为MBone(B.2节)的虚拟网络将一直被使用。

5对套接口选项提供了支持多播的API：

- 在一个接口上加入一个多播组
- 离开一个多播组
- 给外出的多播数据报设置缺省接口
- 给外出的多播数据报设置TTL或跳限
- 使能或禁止多播数据报的回馈。

前两个用于接收，后三个用于发送。IPv4和IPv6的各自5个套接口选项之间的区别足以使通过加入许多#ifdef语句实现的协议无关的多播编码很快变得凌乱不堪。我们自己开发了8个函数，都以mcast_开头，它们可以帮助编写在IPv4和IPv6中都能工作的多播应用程序。

19.13 习 题

19.1 构造图18.9中所示的程序，在命令行上指定IP地址224.0.0.1来运行它，会发生什么情况？

19.2 修改前一个例子中的程序，给它的套接口捆绑到224.0.0.1和端口0，然后运行它。你允许捆绑多播地址到套接口吗？如果你有像tcpdump的工具就用它观察网上的分组。你发送的数据报的源IP地址是什么？

19.3 知道子网上的哪些主机可以多播的一个方法是ping所有主机组224.0.0.1，试一下。

19.4 如果在不能多播的主机unixware上键入ping 224.0.0.1，我们得到如下的输出：

```
unixware % ping 224.0.0.1
PING 224.0.0.1: 56 data bytes
64 bytes from gw.kohala.com (206.62.226.62): icmp_seq=0. time=0. ms
64 bytes from gw.kohala.com (206.62.226.62): icmp_seq=1. time=0. ms
64 bytes from gw.kohala.com (206.62.226.62): icmp_seq=2. time=0. ms
```

发生了什么？

19.5 定位子网上的多播路由器的一个方法是ping所有路由器组224.0.0.2。试一下。

19.6 知道你的主机是否与MBone相连接的一个方法是运行19.8节中的程序，等待几分钟，看是否有任何会话声明出现。试一下看你是否能收到任何声明。

19.7 在图19.13的o=一行中会话ID和版本经常是NTP的时间戳。展示的这些值有意义吗？

19.8 当NTP时间戳的小数部分是1,073,741,824($\frac{1}{4}\times 2^{32}$)时，过一遍图19.19中的计算过程。对最大可能的整数小数($2^{32}-1$)重新做一下计算。

19.9 在图19.24和19.25中，我们设置SO_REUSEADDR套接口选项来允许多次捆绑同一个端口。但是在图19.25中，我们对多播地址(224.0.1.1)和端口123做了两次完全重复的捆绑，在图19.31中则有3次。在一个源自Berkeley的内核上不设置SO_REUSEPORT套接口选项以替代SO_REUSEADDR，我们怎样做到这一点呢？

19.10 在图19.31的最后一行展示了绑定通配地址的套接口接收的一个多播数据报。但是，如果在Solaris 2.5上运行我们的SNTP客户程序，这个套接口不会收到这个多播数据报，为什么？

19.11 在图19.31展示的例子中，如果不禁止图19.26中的多播回馈，我们会收到多少额外的拷贝？

19.12 按IPv4修改mcast_set_if的实现，让它记住已获取其IP地址的每个接口的名字，防止给那个接口再次调用ioctl。

第20章 高级UDP套接口编程

20.1 概 述

本章是各种影响应用程序使用UDP套接口话题的一个集合。首先是确定UDP数据报的目的地址以及是从哪个接口接收数据报的，因为一个绑定UDP端口和通配地址的套接口能在任何接口上接收单播、广播和多播数据报。

TCP是一个字节流协议，它使用一个滑动窗口，它没有像记录边界或者允许发送者用数据淹没接收者等事情需考虑。然而，对于UDP，每个输入操作对应一个UDP数据报(一个记录)，所以当接收的数据报比应用进程的输入缓冲区大时，就产生了如何处理的问题。

UDP是不可靠的，但对一些应用程序来说使用UDP而不用TCP是有意义的，我们会讨论使用UDP而不是TCP的影响因素。在那些UDP应用程序中，我们必须包含一些特性去补偿UDP的不可靠性：超时和重传，处理丢失分组，用序号匹配请求的应答。我们开发了一组函数集，在UDP应用程序中可调用这些函数去处理这些细节。

如果实现不支持IP_RECVDSTADDR套接口选项，那么一个确定UDP数据报目的IP地址的方法是捆绑所有的接口地址并使用select。我们在19.11节中给出了这样的一个例子，在本章还有对这个技术的更多使用。

多数UDP服务器程序是迭代执行的，但是有一些应用程序需在客户和服务器间交换多个UDP数据报，于是需要某种形式的并发。TFTP是一个通常的例子，我们会讨论在inetd下和无inetd下它是如何做的。

最后的话题是针对每个IPv6数据报的能被指定为辅助数据的信息：指定源IP地址、发送接口、外出跳限和下一跳地址。类似的信息(目的IP地址、接收接口和接收跳限)可随IPv6数据报返回。

20.2 接收标志、目的IP地址和接口索引

以往sendmsg和recvmsg只被用来跨越Unix域套接口传递描述字(14.7节)，即使这也很少用到。但是由于下面两个原因使得这两个函数的使用得以增加。

1. 在4.3BSD Reno中加入msghdr结构的msg_flags成员返回标志给应用进程。我们在图13.7中总结了这些标志。
2. 辅助数据被用来在应用进程和内核之间传递越来越多的信息。我们将在第24章看到IPv6继续了这种趋势。

作为recvmsg的一个例子，我们将要写一个名为recvfrom_flags的函数，它与recvfrom类似但它还返回：

1. 返回的 msg_flags 值。
2. 收到的数据报的目的地址(通过设置 IP_RECVDSTADDR 套接口选项)。
3. 接收数据报接口的索引(通过设置 IP_RECVIF 套接口选项)。

为了返回最后两项,我们在 unp.h 头文件中定义了下面的结构:

```
sturct in_pktinfo {
  struct in_addr ipi_addr;        /* destination IPv4 address */
  int            ipi_ifindex;     /* received interface index */
};
```

我们特意选择了结构和成员的名字,使它们与给 IPv6 套接口返回相同两项的 IPv6 in6_pktinfo 结构类似(20.8 节)。我们的 recvfrom_flags 函数将取一个 in_pktinfo 结构的指针作为参数,如果指针不为空,则通过它返回这个结构。

一个有关这个结构的设计问题是如果无法获取 IP_RECVDSTADDR 信息则返回什么(也就是说,实现不支持这个套接口选项)。接口索引容易处理,因为值 0 可以指示索引不可知。但是 IP 地址的所有 32 位值都是合法的。我们选择的是当实际值不可得时,返回一个全零值(0.0.0.0)作为目的地址。尽管这是一个合法的 IP 地址,但它从不允许作为目的 IP 地址(RFC 1122[Braden 1989])。只有当主机正在引导且还不知道自己的 IP 地址时,它作为源 IP 地址才是合法的。

> 非常不幸,源自 Berkeley 的内核接受目的地址为 0.0.0.0 的 IP 数据报(TCPv2 第 218～219 页)。这些是由源自 4.2BSD 的内核产生的过时的广播数据报。

我们在图 20.1 中给出 recvfrom_flags 函数的前半部分。这个函数意在用于 UDP 套接口。

```
 1 #include   "unp.h"
 2 #include   <sys/param.h>      /* ALIGN macro for CMSG_NXTHDR() macro */
 3 #ifdef   HAVE_SOCKADDR_DL_STRUCT
 4 #  include    <net/if_dl.h>
 5 #endif
 6 ssize_t
 7 recvfrom_flags(int fd, void *ptr, size_t nbytes, int *flagsp,
 8                SA *sa, socklen_t *salenptr, struct in_pktinfo *pktp)
 9 {
10     struct msghdr msg;
11     struct iovec iov[1];
12     ssize_t                 n;
13 #ifdef       HAVE_MSGHDR_MSG_CONTROL
14     struct cmsghdr *cmptr;
15     union {
16         struct cmsghdr   cm;
17         char             control[CMSG_SPACE(sizeof(struct in_addr)) +
18                                  CMSG_SPACE(sizeof(struct sockaddr_dl))];
19     } control_un;
20     msg.msg_control = control_un.control;
21     msg.msg_controllen = sizeof(control_un.control);
22     msg.msg_flags = 0;
```

```
23 #else
24     bzero(&msg, sizeof(msg));       /* make certain msg_accrightslen = 0 */
25 #endif
26     msg.msg_name = sa;
27     msg.msg_namelen = *salenptr;
28     iov[0].iov_base = ptr;
29     iov[0].iov_len = nbytes;
30     msg.msg_iov = iov;
31     msg.msg_iovlen = 1;
32     if ( (n = recvmsg(fd, &msg, *flagsp)) < 0)
33         return(n);
34     *salenptr = msg.msg_namelen;   /* pass back results */
35     if (pktp)
36         bzero(pktp, sizeof(struct in_pktinfo));   /* 0.0.0.0, i/f = 0 */
```

图 20.1 recvfrom_flags 函数：调用 recvmsg[advio/recvfromflags.c]

包含文件

第 2～5 行 为了使用宏 CMSG_NXTHDR，需要包含头文件<sys/param.h>。我们还要包含<net/if_dl.h>头文件，它定义了 sockaddr_dl 结构，接收接口的索引要从它返回。

函数参数

第 6～8 行 这个函数的参数与 recvfrom 类似，不过第四个参数现在是一个指向整数标志的指针(这样我们就可以返回由 recvmsg 返回的标志)，第七个参数则是新的：它是一个指向 in_pktinfo 结构的指针，这个结构将填入所接收数据报的目的 IPv4 地址和接收它的接口索引。

实现差别

第 13～25 行 当处理 msghdr 结构和各种 MSG_xxx 常值时，我们会遇到许多不同实现的差别。我们处理这些差别的方法是使用 C 的条件包含特性(#ifdef)。如果实现支持 msg_control 成员，则分配空间去储存由 IP_RECVDSTADDR 和 IP_RECVIF 套接口选项返回的值，并且初始化适当的成员。

填写 msghdr 结构并调用 recvmsg

第 26～36 行 填写完一个 msghdr 结构后调用 recvmsg。msg_namelen 和 msg_flags 的值必须传回给调用者；它们是值-结果参数。我们还初始化调用者的 in_pktinfo 结构，置 IP 地址为 0.0.0.0，置接口索引为 0。

图 20.2 给出了函数的后半部分。

第 37～40 行 如果实现不支持 msg_control 成员，我们只设置返回标志为 0 就返回。剩余的函数部分处理 msg_control 信息。

如果没有控制信息则返回

第 41～44 行 我们返回 msg_flags 的值，接着返回调用者，如果(a)没有控制信息，(b)控制信息被截断，(c)调用者不想要返回一个 in_pktinfo 结构。

处理辅助数据

第 45～46 行 我们使用 CMSG_FIRSTHDR 和 CMSG_NXTHDR 宏处理任意数目的

辅助数据对象。

处理 IP_RECVDSTADDR

第 47～54 行　如果目的 IP 地址是作为控制信息返回的(图 13.9),它就返回给调用者。

处理 IP_RECVIF

第 55～63 行　如果接收接口的索引是作为控制信息返回的,它就返回给调用者。图 20.3 展示了返回的辅助数据对象的内容。

```
37 #ifndef      HAVE_MSGHDR_MSG_CONTROL
38     *flagsp = 0;                        /* pass back results */
39     return(n);
40 #else
41     *flagsp = msg.msg_flags;            /* pass back results */
42     if (msg.msg_controllen < sizeof(struct cmsghdr) ||
43         (msg.msg_flags & MSG_CTRUNC) || pktp == NULL)
44         return(n);
45     for (cmptr = CMSG_FIRSTHDR(&msg); cmptr != NULL;
46           cmptr = CMSG_NXTHDR(&msg, cmptr)) {
47 #ifdef   IP_RECVDSTADDR
48           if (cmptr->cmsg_level == IPPROTO_IP &&
49               cmptr->cmsg_type == IP_RECVDSTADDR) {
50               memcpy(&pktp->ipi_addr, CMSG_DATA(cmptr),
51                       sizeof(struct in_addr));
52               continue;
53           }
54 #endif
55 #ifdef   IP_RECVIF
56           if (cmptr->cmsg_level == IPPROTO_IP &&
57               cmptr->cmsg_type == IP_RECVIF) {
58               struct sockaddr_dl *sdl;
59               sdl = (struct sockaddr_dl *) CMSG_DATA(cmptr);
60               pktp->ipi_ifindex = sdl->sdl_index;
61               continue;
62           }
63 #endif
64           err_quit("unknown ancillary data, len = %d, level = %d, type = %d",
65                    cmptr->cmsg_len, cmptr->cmsg_level, cmptr->cmsg_type);
66     }
67     return(n);
68 #endif   /* HAVE_MSGHDR_MSG_CONTROL */
69 }
```

图 20.2　recvfrom_flags 函数:返回标志和目的地址[advio/recvfromflags.c]

回想图 17.1 中的数据链路套接口地址结构,在上述辅助数据对象中返回的数据只是这些结构之一,但是三个长度成员都是 0(名字长度、地址长度和选择符长度)。因此,跟在这些长度后面不需要任何数据,所以结构的长度应是 8 字节,而不是图 17.1 中所示的 20。我们返回的信息是接口索引。

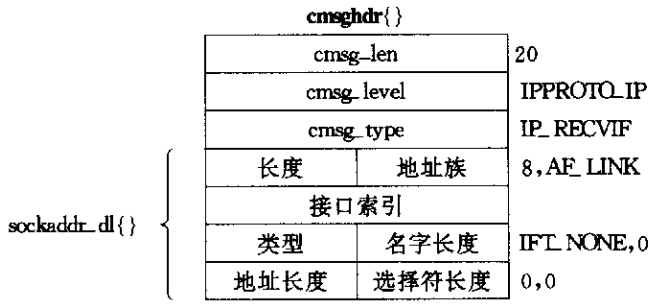


图 20.3 IP_RECVIF 返回的辅助数据对象

例子:输出目的 IP 地址和数据报截断标志

为了测试我们的函数,我们修改 dg_echo 函数(图 8.4)去调用 recvfrom_flags 而不是 recvfrom。我们在图 20.4 中给出这个新版本的 dg_echo 函数。

```
1 #include    "unpifi.h"
2 #undef    MAXLINE
3 #define    MAXLINE    20        /* to see datagram truncation */
4 void
5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     int      flags;
8     const int on = 1;
9     socklen_t len;
10    ssize_t n;
11    char     mesg[MAXLINE], str[INET6_ADDRSTRLEN], ifname[IFNAMSIZ];
12    struct in_addr in_zero;
13    struct in_pktinfo pktinfo;
14 #ifdef    IP_RECVDSTADDR
15    if (setsockopt(sockfd, IPPROTO_IP, IP_RECVDSTADDR, &on, sizeof(on)) < 0)
16        err_ret("setsockopt of IP_RECVDSTADDR");
17 #endif
18 #ifdef    IP_RECVIF
19    if (setsockopt(sockfd, IPPROTO_IP, IP_RECVIF, &on, sizeof(on)) < 0)
20        err_ret("setsockopt of IP_RECVIF");
21 #endif
22    bzero(&in_zero, sizeof(struct in_addr));  /* all 0 IPv4 address */
23    for ( ; ; ) {
24        len = clilen;
25        flags = 0;
26        n = Recvfrom_flags(sockfd, mesg, MAXLINE, &flags,
27                           pcliaddr, &len, &pktinfo);
28        printf("%d-byte datagram from %s", n, Sock_ntop(pcliaddr, len));
29        if (memcmp(&pktinfo.ipi_addr, &in_zero, sizeof(in_zero)) != 0)
30            printf(", to %s", Inet_ntop(AF_INET, &pktinfo.ipi_addr,
31                                        str, sizeof(str)));
32        if (pktinfo.ipi_ifindex > 0)
```

```
33              printf(", recv I/f = %s",
34                      If_indextoname(pktinfo.ipi_ifindex, ifname));
35 #ifdef       MSG_TRUNC
36          if (flags & MSG_TRUNC)
37              printf(" (datagram truncated)");
38 #endif
39#ifdef        MSG_CTRUNC
40          if (flags & MSG_CTRUNC)
41              printf(" (control info truncated)");
42 #endif
43 #ifdef       MSG_BCAST
44          if (flags & MSG_BCAST)
45              printf(" (broadcast)");
46 #endif
47 #ifdef       MSG_MCAST
48          if (flags & MSG_MCAST)
49              printf(" (multicast)");
50 #endif
51          printf("\n");
52          Sendto(sockfd, mesg, n, 0, pcliaddr, len);
53      }
54 }
```

图 20.4　调用 recvfrom_flags 函数的 dg_echo 函数[advio/dgechoaddr.c]

改变 MAXLINE

第 2～3 行　我们去掉出现在 unp.h 头文件中的 MAXLINE 的定义，把它重定义为 20。我们这样做的目的是看当我们收到一个比我们传给输入函数(该例中是 recvmsg)的缓冲区大的 UDP 数据报时会发生什么事情。

设置 IP_RECVDSTADDR 和 IP_RECVIF 套接口选项

第 14～21 行　如果 IP_RECVDSTADDR 套接口选项有定义，它就被打开。同样地，IP_RECVIF 套接口选项也被打开。

> 很不幸，这种形式的测试还不充分，因为一些系统(例如 Solaris 2.5)尽管不支持但定义了 IP_RECVDSTADDR。为此我们允许对 setsockopt 的调用失败，如果这种情况发生，我们只是输出一条信息接着继续运行。我们甚至不能检查一个特定的错误，因为当一个套接口选项没有实现时，不同的实现会返回不同的错误。例如，4.4BSD 的 getsockopt 对一个未知的选项返回 ENOPROTOOPT，但是 setsockopt 返回 EINVAL。可是 4.4BSD 的多播套接口选项对未知选项却返回 EOPNOTSUPP。

读数据报，输出源 IP 地址和端口号

第 24～28 行　数据报由调用 recvfrom_flags 读入。服务器应答的源 IP 地址和端口号由 sock_ntop 转换为表达格式。

输出目的 IP 地址

第 29～31 行　如果返回的 IP 地址不是 0，它就由 inet_ntop 转换为表达格式并输出。

输出接收接口的名字

第32～34行 如果返回的接口索引不是0，它的名字通过调用if_indextoname获得并输出。

测试各种标志

第35～51行 我们接着测试4个另外的标志，如果任何一个是打开的就输出一条消息。

如果在主机bsdi(它是多宿的)的BSD/OS 3.0下运行我们的服务器程序，我们就可以看到各种目的IP地址和标志：

```
bsdi % udpserv01
9-byte datagram from 206.62.226.33.41164, to 206.62.226.35, recv i/f = ef0
13-byte datagram from 206.62.226.65.1057, to 206.62.226.95, recv i/f = we0
    (broadcast)
4-byte datagram from 206.62.226.33.41176, to 224.0.0.1, recv i/f = ef0
    (multicast)
20-byte datagram from 127.0.0.1.4632,to 127.0.0.1, recv i/f = lo0
    (datagram truncated)
9-byte datagram from 206.62.226.33.41178, to 206.62.226.66, recv i/f = ef0
```

为便于阅读，我们将在括号中输出标志的行折了一下行。为了产生这5行的输出，我们在不同的主机上运行我们的sock程序(C.3节)，产生发送到服务器的数据报。

1. 第一行是从主机solaris到206.62.226.35，它是服务器主机bsdi的单播地址之一。
2. 第二行是从主机laptop到206.62.226.95，它是客户主机和服务器主机共享的以太网的广播地址(图1.16)。就同我们想像的，接口不同于第一行。我们的服务器接收该广播数据报并且设置MSG_BCAST标志，指明数据报是作为一个链路层广播接收的。
3. 第三行是从主机solaris到224.0.0.1，它是所有主机多播组地址。所有子网上的能多播的主机都必须属于这个组。我们的服务器接收该多播数据报，因为BSD/OS操作系统是能多播的，并且目的端口与我们的服务器端口匹配。它还设置MSG_MCAST标志，指明数据报是作为链路层多播接收的。
4. 第四行是从服务器主机本身到回馈地址127.0.0.1，接口是lo0。而且，这一次我们在客户键入了41字节的行(没有展示出来)。服务器只收到了该数据报的头20字节，MSG_TRUNC标志被设置，指明数据报被截断。
5. 最后一行是从主机solaris到206.62.226.66，它是服务器主机在另一个以太网(图1.16)的单播地址。服务器仍接收数据报(因为主机实现弱端系统模型，见8.8节)，但是目的IP地址(206.62.226.66)与数据报接收接口的地址不匹配。

当接收的数据报是广播或多播数据报时，早期的源自Berkeley的实现忽略IP_RECVDSTADDR套接口选项(TCPv2第776页)。新的版本修复了这个缺陷。

如果目的地址是255.255.255.255，BSD/OS就将它转换成接收接口的广播地址。

20.3　数据报截断

前一节的例子中表明，在 BSD/OS 环境下，当一个到来的 UDP 数据报的长度大于应用进程的缓冲区时，recvmsg 在 msghdr 结构（图 13.7）中的 msg_flags 成员上设置 MSG_TRUNC 标志。所有支持 msghdr 结构及其 msg_flags 成员的源自 Berkeley 的实现都提供这种通知。

> 这是一个必须从内核向进程返回标志的例子。我们在 13.3 节中提到过，一个与 recv 和 recvfrom 函数有关的设计问题是标志参数是一个整数，这允许从进程向内核传递标志，但反过来不行。

不幸的是，并非所有的实现都以这种方式处理超过预期长度的 UDP 数据报。这里有三种可能的情形。

1. 丢掉超出的字节并给应用进程返回 MSG_TRUNC 标志。这要求应用进程调用 recvmsg 来接收这个标志。
2. 丢掉超出的字节但不通知应用进程。
3. 保留超出的字节并在随后这个套接口上的读操作中返回这些数据。

我们已在 BSD/OS 上看到了第一种类型的行为。第二种类型的行为在 Solaris 2.5 上可看到：超出的字节被丢弃，但是由于它的 msghdr 结构不支持 msg_flags 成员，从而没有方法给应用进程返回错误。

> Posix.1g 指定了第一种类型的行为：丢弃超出的字节并设置 MSG_TRUNC 标志。早期的 SVR4 版本展现的是第三种类型的行为。

由于不同的实现在处理比应用进程接收缓冲区大的数据报时有很大差异，发现这种问题的方法之一就是分配一个比应用进程可能收到的最大数据报大 1 字节的应用缓冲区。如果收到长度等于该缓冲区的数据报，就认为这是一个错误。

20.4　何时使用 UDP 而不是 TCP

在 2.3 和 2.4 节中，我们描述了 UDP 和 TCP 的主要区别。既然 TCP 是可靠的而 UDP 不是，问题就产生了：何时该用 UDP 而不是 TCP 呢？为什么？我们首先列举 UDP 的优点。

- 就像在图 18.1 中展示的，UDP 支持广播和多播。其实，如果应用程序使用广播或多播，它必须用 UDP。我们在第 18 章和第 19 章讨论过这两种编址模式。
- UDP 没有连接建立和拆除。对于图 2.5，UDP 只需两个分组来交换一个请求和应答（假设两者的长度都小于两个端系统之间的最小 MTU）。假设为每个请求-应答交换建立一个新的 TCP 连接，TCP 需要大约 10 个分组。

  在分组数目分析中同样重要的是获得应答所需的分组往返次数。如果延迟超出了带宽，这会变得很重要，就像 TCPv3 的附录 A 中描述的那样。那段文字表明最小的事务处理时间（transaction time）对 UDP 请求-应答来说是 RTT+SPT，其中 RTT 是客

户与服务器之间的往返时间，SPT 是服务器处理请求的时间。然而对 TCP 来说，如果一个新的 TCP 连接用于请求-应答，那么最小的事务处理时间是 2×RTT+SPT，比 UDP 时间多一个 RTT。TCPv3 和本书 13.9 节还描述了对 TCP 的一个修改，称为 T/TCP 或“事务 TCP”，它通常避免了 TCP 的三路握手，允许 T/TCP 达到跟 UDP 的 RTT+SPT 事务处理时间相同。

对于第二点应该是明显的：如果一个 TCP 连接用于多个请求-应答交换，那么建立和拆除连接的负担就分担给了所有的请求和应答，这通常是比为每个请求-应答建一个新连接更好的设计。然而，有些应用程序给每个请求-应答用一个新的 TCP 连接（例如，HTTP），有的应用程序则客户和服务器交换一个请求-应答（例如 DNS）后，接着可能几个小时或几天没有交互。

我们现在列出 UDP 不能提供的 TCP 的特性，这意味着如果这些特性对应用程序是必需的，它们必须自己来提供。我们使用了限定词“必需”，因为并不是所有的应用程序都需要所有这些特性。例如，对于实时的音频应用系统，如果接收者能插值丢失的数据，则丢掉的分节可能不需要重传。同样，对于简单的请求-应答事务处理，如果双方事先统一最大的请求和应答大小，就可能不需要滑动窗口的流控。

- 正面确认，丢失分组重传，重复分组检测，给被网络打乱次序的分组排序。TCP 对所有数据都确认，可以发现丢失的分组。这些特性的实现要求每个 TCP 数据分节包含一个能被确认的序列号，还需要 TCP 为每个连接估计一个重传超时值，这个值应该随着两个端系统间的流量变化经常更新。
- 窗口式的流控。接收方 TCP 通知发送方它为接收数据分配了多少缓冲空间，发送方不能超过这个缓冲空间。也就是，发送方未被确认的数据量不能超过接收方告知的窗口。
- 慢启动和拥塞避免。这是流控的一种方式，由发送者确定当前网络容量和处理阵发的拥塞。所有现在的 TCP 必须支持这两个特性，我们从经验（在这些算法实现之前的 80 年代后期）知道，在拥塞面前不“退后”的协议只会使拥塞更糟糕（例如[Jacobson 1988]）。

作为总结，我们可以给出如下的建议：

- 对广播或多播应用程序必须使用 UDP。任何形式的期望的错误控制必须加入到客户和服务器程序中，但是当一定量（假定很少）错误无所谓时（例如音频或视频的丢失分组），应用程序经常使用广播和多播。多播应用系统需要建立可靠的递送（例如多播文件传输），但是我们必须确定使用多播获得的性能（发送一个分组到 N 个目的地，对比于通过 N 个 TCP 连接发送 N 个拷贝）是否超过为在应用系统中提供可靠通信要求附加的复杂性。
- UDP 可以用于简单的请求-应答式应用程序，但是应用程序内部必须有检查错误的功能。这至少涉及确认、超时和重传。流控对于合理长度的请求和响应并不重要。我们在 20.5 节的一个 UDP 应用程序中将提供这些特性的例子。这里要考虑的因素是客户和服务器间通信的频度（是否一个 TCP 连接可以在两个请求-应答之间保留？）和交换的数据量（如果通常需要多个分组，那么 TCP 连接建立和断开的开销就不成

为重要因素)。

- UDP 不应该用于海量数据的传输(例如文件传输)。因为它要求将窗口式流控、拥塞避免和慢启动以及上面一点提到的特性在应用程序中实现,这意味着我们要在应用程序中重建 TCP。我们应该让厂商注重考虑较好的 TCP 性能,而把我们的努力放在应用程序本身。

对于这些规则有例外,尤其在现存的应用程序中。例如 TFTP 就用 UDP 来海量传输数据。给 TFTP 选择 UDP 的原因是在引导代码中它比用 TCP 易于实现(例如,TCPv2 中使用 UDP 的 C 代码为 800 行,而使用 TCP 则为 4500 行),而且 TFTP 只用于 LAN 上以自举系统,而不是跨越 WAN 传输海量数据。但是这要求 TFTP 中包含它自己的用于确认的序列号字段,同时提供超时以及重传能力。

NFS 是这些规则的另一个例外:它也用 UDP 来海量数据传输(尽管有人声称它是一个请求-应答应用程序,只是使用了大量的请求和应答)。这是有一部分历史原因的,因为在 80 年代中期设计它的时候,UDP 的实现比 TCP 要快,而且 NFS 只用于 LAN,这里的丢分组率与 WAN 相比要少几个数量级。但是随着 NFS 在 90 年代早期被用于跨越 WAN,随着 TCP 的实现在海量数据传输性能上超过了 UDP,NFS 版本 3 被设计成支持 TCP,而且多数厂商现在同时提供 UDP 和 TCP 上的 NFS。同样的理由(在 80 年代中期 UDP 比 TCP 要快以及 LAN 的数量多于 WAN)导致了 DCE 远程过程调用(RPC)的前身软件包(Apollo NCS 软件包)选择了 UDP 而不是 TCP,虽然现在的实现同时支持 UDP 和 TCP。

随着好的 TCP 实现达到跟今天的网络同样快的性能以及越来越少的应用程序设计者愿意在 UDP 应用程序中再实现 TCP,我们可能要说与 TCP 相比 UDP 的使用会越来越少。但是下一个十年代预计在多媒体应用中的增长会看到 UDP 使用的增加,因为多媒体通常意味着多播,这需要 UDP。

20.5　给 UDP 应用程序增加可靠性

就像在前一节中提到的,如果我们想要在请求-应答式应用程序中使用 UDP,那么我们必须对我们的客户增加两个特性:

1. 超时和重传以处理丢失的数据报。
2. 序列号,这样客户可以验证一个应答是对应相应的请求的。

这两个特性是多数使用简单的请求-应答范例的现有 UDP 应用程序的一部分:例如 DNS 解析器、SNMP 代理、TFTP 和 RPC。我们不是试图用 UDP 于海量数据传输——我们的意图是为了剖析发送请求并等待应答的应用程序。

> 由定义,数据报是不可靠的;因此,我们故意没有称之为"可靠的数据报服务"。确实,"可靠的数据报"是一个矛盾的说法。我们将给出的是一个在不可靠的数据报服务(UDP)之上加入可靠性的应用程序。

加入序列号比较简单。客户给每个请求附加一个序列号,并且服务器必须在应答中给客户返回这个号。这样可以让客户验证给定的应答是对应所发请求的应答。

老式的处理超时和重传的方法是发送一个请求后等待N秒。如果没有收到应答,则重传并再等待另外N秒。这种情况发生一定的次数后放弃。这是一种线性重传定时器。(TCPv1的图6.8给出了使用这个方法的TFTP客户程序的例子。许多TFTP客户程序仍使用这种方法。)

这种方法的问题是数据报在一个互联网上的往返时间会从LAN上的远远不到一秒变化到WAN上的许多秒。影响往返时间(RTT)的因素是距离、网络速度和拥塞。另外,当网络条件变化时,客户和服务器之间的RTT会随着时间很快地变化。我们必须采用一个将实际测得的RTT以及RTT随着时间的变化考虑在内的超时和重传算法。许多工作已经集中在这个领域,多数是与TCP相关的,但是同样的想法可应用于任何网络应用程序。

我们想要计算用于我们发送的每个分组的重传超时(RTO)。为了计算它我们先测量RTT:分组的实际往返时间。每次我们测得一个RTT,我们就更新两个统计性估计因子:srtt是平滑了的RTT估计因子,rttvar是平滑了的平均偏差估计因子。后者只是标准偏差的一个很好的近似,但是由于不需要开方,所以容易计算。有了两个估计因子,使用的RTO就是srtt加上四倍的rttvar。[Jacobson 1988]提供了这些计算的所有细节,我们可以用下面的四个方程加以总结:

$$\begin{aligned} \text{delta} &= \text{measuredRTT} - \text{srtt} \\ \text{srtt} &\leftarrow \text{srtt} + g \times \text{delta} \\ \text{rttvar} &\leftarrow \text{rttvar} + h(|\text{delta}| - \text{rttvar}) \\ \text{RTO} &= \text{srtt} + 4 \times \text{rttvar} \end{aligned}$$

delta是测得的RTT和当前平滑了的RTT估计因子(srtt)之间的差。g是施加在RTT估计因子上的增益因子,值为1/8。h是施加在平均偏差估计因子上的增益因子,值为1/4。

> 在RTO计算中的两个增益因子和乘数4都故意是2的指数,这样用移位操作而不是乘除就可以计算。真正的TCP内核实现(TCPv2的25.7节)为了速度通常用定点算术进行的,但是为了简单起见,在我们后面的代码中使用了浮点计算。

在[Jacobson 1988]中指出的另一点是当重传定时器超时的时候,对下一个RTO要用一个指数回退(exponential backoff)。例如,如果RTO是2秒,这段时间内应答没有收到,则下一个RTO是4秒。如果仍没有应答,下一个RTO是8秒接着是16秒,就这样一直下去。

Jacobson的算法告诉我们每次测量RTT如何计算RTO以及当我们重传时如何增加RTO。但是当我们必须重传一个分组时接着收到了一个应答,问题出现了。这被称为"重传二义性问题"。图20.5展示了当重传定时器超时时,三种可能的情形。

- 请求丢失了
- 应答丢失了
- RTO太小

当客户收到已重传请求的应答时,它不能区分应答是对应于哪一个请求的。在右边的例子中,应答对应的是初始的请求。但是在另两个例子中,应答是对应于重传的请求。

Karn的算法[Karn and Partridge 1987]处理了这种情形,当收到应答对应于一个已重传的请求时,适用下面的规则。

- 如果测量了 RTT，由于我们不知道应答对应的请求，因此不用它去更新估计因子。

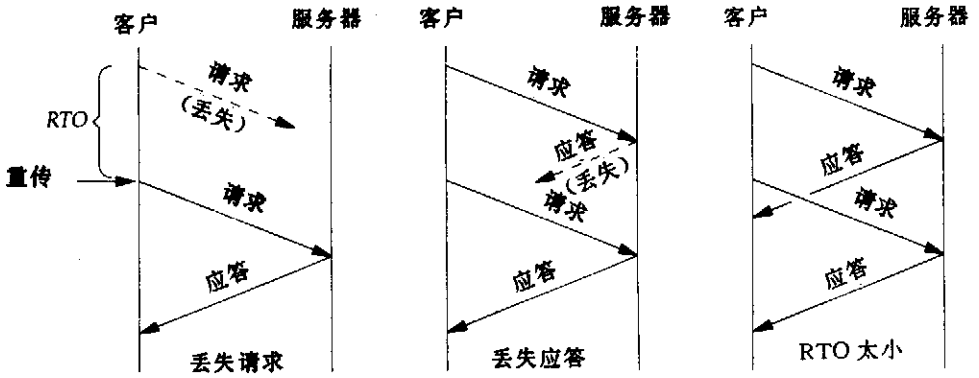


图 20.5　当重传定时器超时的三种情形

- 由于在重传定时器超时前应答到来，因此对下一个分组重复使用同一个 RTO。只有当我们收到了一个非重传请求的应答时，我们才去更新 RTT 估计因子并重新计算 RTO。

当我们编写自己的 RTT 函数时，将 Karn 的算法考虑在内并不困难，但结果是还有更好更精致的解法存在。这个解法是从对“长胖管道”(有高带宽或长 RTT 或两者都有的网络)的 TCP 扩展中来的，它在 RFC 1323[Jacobson，Braden，and Borman 1992]中描述。除了给每个请求加上一个服务器必须返回的序列号，我们还加上一个也要服务器返回的时间戳。每次我们发送一个请求，保存当前时间在时间戳中。当收到一个应答时，我们计算 RTT 为当前时间减去在应答中由服务器返回的时间戳。由于每个请求载有一个将由服务器返回的时间戳，因此我们可以计算我们收到的每个应答的 RTT。这就再也没有二义性了。而且，由于服务器所做的只是返回客户的时间戳，因此客户可以用任何想用的时间戳形式，并且不需要客户和服务器有同步的时钟。

例子

我们现在把所有这些都放在一个例子中。我们从图 8.7 中的 UDP 客户程序 main 函数开始，只是将端口号从 SERV_PORT 改为 7(标准的回射服务器，图 2.13)。

图 20.6 是 dg_cli 函数。仅有的对图 8.8 的改动是将对 sendto 和 recvfrom 的调用替换为调用我们的新函数 dg_send_recv。

在给出 dg_send_recv 函数和它调用的 RTT 函数之前，图 20.7 给出了我们如何给一个 UDP 客户程序增加可靠性的框架，所有以 rtt_打头的函数随后给出。

```
1 #include    "unp.h"
2 ssize_t    Dg_send_recv(int, const void *, size_t, void *, size_t,
3                         const SA *, socklen_t);
4 void
5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
6 {
7     ssize_t     n;
8     char    sendline[MAXLINE], recvline[MAXLINE + 1];
```

```
 9      while (Fgets(sendline, MAXLINE, fp) ! = NULL) {
10              n = Dg_send_recv(sockfd, sendline, strlen(sendline),
11                               recvline, MAXLINE, pservaddr, servlen);
12              recvline[n] = 0;  /* null terminate */
13              Fputs(recvline, stdout);
14      }
15 }
```

图 20.6 dg_cli 函数调用我们的 dg_send_recv 函数[rtt/dg_cli.c]

```
static sigjmp_buf jmpbuf;
{
    . . .
    form request
    signal(SIGALRM,sig_alrm);       /* establish signal handler */
    rtt_newpack();                  /* initialize rexmt counter to 0 */
sendagain:
    sendto();
    alarm(rtt_start());             /* set alarm for RTO seconds */
    if (sigsetjmp(jmpbuf, 1) ! = 0) {
        if (rtt_timeout())          /* double RTO, retransmitted enough? */
            give up
        goto sendagain;             /* retransmit */
    }
    do {
        recvfrom();
    } while (wrong sequence#)
    alarm(0);                       /* turn off alarm */
    rtt_stop();                     /* calculate RTT and update estimators */
    process reply
    . . .
}
void
sig_alrm(int signo)
{
    siglongjmp(jmpbuf, 1);
}
```

图 20.7 RTT 函数的框架以及它们何时被调用

当所接收的应答其序列号不是期待的时，我们再次调用 recvfrom，但是我们不重传请求，不重启运行着的重传定数器。注意图 20.5 中最右边的例子，相应于所重传请求的最后一个应答将会在下一次客户发送新请求时存入套接口接收缓冲区中。这不会有问题，因为客户将读入这个应答，注意到其序列号不是期待的，于是丢弃这个应答并再一次调用 recvfrom。

我们调用 sigsetjmp 和 siglongjmp 来避免我们在 18.5 节讨论过的由 SIGALRM 信号引起的竞争状态。

图 20.8 给出了 dg_send_recv 函数的前半部分。

第 1～5 行 我们包含一个新头文件 unprtt.h，在图 20.10 中给出，它定义给客户保存 RTT 信息的 rtt_info 结构。我们定义了这样的一个结构和很多其他变量。

定义 msghdr 结构和 hdr 结构

第 6～10 行 我们想把给每个分组附加序列号和时间戳的事实对调用者隐藏起来。最

简单的方法是使用 writev，写我们的头部(hdr 结构)跟着是调用者的数据，将它作为单独的一个 UDP 数据报。回想对 writev 在数据报套接口上的输出是单个数据报。这比让调用者在它的缓冲区前分配空间要简单，并且比为了单个 sendto 而复制我们的头部和调用者的数据到一个缓冲区(我们必须分配它的空间)要快。但是由于我们用的是 UDP 并且必须指定一个目的地址，因此我们必须使用 sendmsg 和 recvmsg 的 iovec 功能而不是用 sendto 和 recvfrom。回想 13.5 节，有的系统有一个能存放辅助数据的较新的 msghdr 结构，而老的系统仍在这个结构的末尾使用访问权力成员。为了避免由于用 #ifdef 伪代码处理这些差别使代码复杂，我们声明了两个 static 的 msghdr 结构，由 C 强制初始化它们为 0，然后就忽略这两个结构末尾没有用到的成员。

第一次被调用时进行初始化

第 20～24 行　第一次被调用时，我们调用 rtt_init 函数。

填写 msghdr 结构

第 25～41 行　我们填写用于输入输出的两个 msghdr 结构。我们给输出分组增加了发送序列号，但是直到发送这个分组时我们才设置发送时间戳(由于它可能被重传，而每次重传都需要当前时间戳)。

函数的后半部分和 sig_alrm 信号处理程序一起在图 20.9 中给出。

```
 1 #include    "unprtt.h"
 2 #include    <setjmp.h>
 3 #define     RTT_DEBUG
 4 static struct rtt_info   rttinfo;
 5 static int   rttinit = 0;
 6 static struct msghdr     msgsend, msgrecv;     /* assumed init to 0 */
 7 static struct hdr {
 8     uint32_t   seq;       /* sequence # */
 9     uint32_t   ts;        /* timestamp when sent */
10 } sendhdr, recvhdr;
11 static void sig_alrm(int signo);
12 static sigjmp_buf jmpbuf;
13 ssize_t
14 dg_send_recv(int fd, const void *outbuff, size_t outbytes,
15                     void *inbuff, size_t inbytes,
16                     const SA *destaddr, socklen_t destlen)
17 {
18     ssize_t n;
19     struct iovec     iovsend[2], iovrecv[2];
20     if (rttinit == 0) {
21         rtt_init(&rttinfo);     /* first time we're called */
22         rttinit = 1;
23         rtt_d_flag = 1;
24     }
25     sendhdr.seq++;
26     msgsend.msg_name = destaddr;
27     msgsend.msg_namelen = destlen;
28     msgsend.msg_iov = iovsend;
29     msgsend.msg_iovlen = 2;
```

```
30      iovsend[0].iov_base = &sendhdr;
31      iovsend[0].iov_len = sizeof(struct hdr);
32      iovsend[1].iov_base = outbuff;
33      iovsend[1].iov_len = outbytes;
34      msgrecv.msg_name = NULL;
35      msgrecv.msg_namelen = 0;
36      msgrecv.msg_iov = iovrecv;
37      msgrecv.msg_iovlen = 2;
38      iovrecv[0].iov_base = &recvhdr;
39      iovrecv[0].iov_len = sizeof(struct hdr);
40      iovrecv[1].iov_base = inbuff;
41      iovrecv[1].iov_len = inbytes;
```

图 20.8 dg_send_recv 函数:前半部分[rtt/dg_send_recv.c]

```
42      Signal(SIGALRM, sig_alrm);
43      rtt_newpack(&rttinfo);      /* initialize for this packet */
44  sendagain:
45      sendhdr.ts = rtt_ts(&rttinfo);
46      Sendmsg(fd, &msgsend, 0);
47      alarm(rtt_start(&rttinfo));    /* calc timeout value & start timer */
48      if (sigsetjmp(jmpbuf, 1) != 0) {
49          if (rtt_timeout(&rttinfo) < 0) {
50              err_msg("dg_send_recv: no response from server, giving up");
51              rttinit = 0;      /* reinit in case we're called again */
52              errno = ETIMEDOUT;
53              return(-1);
54          }
55          goto sendagain;
56      }
57      do {
58          n = Recvmsg(fd, &msgrecv, 0);
59      } while (n < sizeof(struct hdr) || recvhdr.seq != sendhdr.seq);
60      alarm(0);      /* stop SIGALRM timer */
61          /* calculate & store new RTT estimator values */
62      rtt_stop(&rttinfo, rtt_ts(&rttinfo) - recvhdr.ts);
63      return(n - sizeof(struct hdr));      /* return size of received datagram */
64  }
65  static void
66  sig_alrm(int signo)
67  {
68      siglongjmp(jmpbuf, 1);
69  }
```

图 20.9 dg_send_recv 函数:后半部分[rtt/dg_send_recv.c]①

建立信号处理程序

第 42～43 行 建立一个处理 SIGALRM 的信号处理程序,rtt_newpack 设置重传计数为 0。

① 作者注: 图 20.9 中存在一个非致命的竞争状态:如果 SIGALRM 是在某个成功的 recvmsg 调用之后的第 59 行和第 60 行之间递交,那么它将导致一次不必要的重传。

发送数据报

第 45～47 行　当前的时间戳由 rtt_ts 获得并被存入 hdr 结构中，而这个结构是要安在用户数据之前的。单个 UDP 数据报由 sendmsg 送出。rtt_start 返回这次超时值的秒数，我们以此调用 alarm 以调度 SIGALRM。

建立跳转缓冲区

第 48 行　我们用 sigsetjmp 给信号处理程序建立了一个跳转缓冲区。通过调用 recvmsg，我们等待下一个数据报。（我们在图 18.9 中讨论过 sigsetjmp、siglongjmp 及 SIGALRM 的用法）。如果 alarm 定时器超时，sigsetjmp 就返回 1。

处理超时

第 49～55 行　当超时发生时，rtt_timeout 计算下一个 RTO（指数回退），如果我们应放弃则返回－1，如果应重传则返回 0。如果放弃，我们就设置 errno 为 ETIMEDOUT 并返回调用者。

调用 recvmsg，比较序列号

第 57～59 行　我们通过调用 recvmsg 来等待一个数据报的到来。当它返回时，数据报的长度必须至少是我们的 hdr 结构的长度，并且序列号与发送序列号相同。如果有一个比较失败，则再一次调用 recvmsg。

关闭 alarm 并更新 RTT 估计因子

第 60～62 行　当期待的应答收到时，当前 alarm 被关闭并且 rtt_stop 更新 RTT 估计因子。rtt_ts 返回当前的时间戳，从它这儿减去所收到数据报中的时间戳得到 RTT。

SIGALRM 处理程序

第 65～69 行　siglongjmp 被调用，导致 dg_send_recv 的 sigsetjmp 返回 1。

我们现在看一下由 dg_send_recv 调用的各种 RTT 函数。图 20.10 给出了 unprtt.h 头文件。

```
 1 #ifndef   __unp_rtt_h
 2 #define   __unp_rtt_h
 3 #include   "unp.h"
 4 struct rtt_info {
 5     float   rtt_rtt;          /* most recent measured RTT, seconds */
 6     float   rtt_srtt;         /* smoothed RTT estimator, seconds */
 7     float   rtt_rttvar;       /* smoothed mean deviation, seconds */
 8     float   rtt_rto;          /* current RTO to use, seconds */
 9     int   rtt_nrexmt;         /* #times retransmitted: 0, 1, 2, ... */
10     uint32_t   rtt_base;      /* #sec since 1/1/1970 at start */
11 };
12 #define  RTT_RXTMIN  2       /* min retransmit timeout value, seconds */
13 #define  RTT_RXTMAX  60      /* max retransmit timeout value, seconds */
14 #define  RTT_MAXNREXMT  3    /* max #times to retransmit */
15                              /* function prototypes */
16 void    rtt_debug(struct rtt_info *);
17 void    rtt_init(struct rtt_info *);
18 void    rtt_newpack(struct rtt_info *);
19 int     rtt_start(struct rtt_info *);
```

```
20 void     rtt_stop(struct rtt_info *, uint32_t);
21 int      rtt_timeout(struct rtt_info *);
22 uint32_t rtt_ts(struct rtt_info *);
23 extern int   rtt_d_flag;     /* can be set nonzero for addl info */
24 #endif     /*  __unp_rtt_h  */
```

图 20.10 unprtt.h 头文件[lib/unprtt.h]

rtt_info 结构

第 4～11 行 这个结构含有用于给客户和服务器间的分组定时所必需的变量。头四个变量是从本节开始给出的方程中来的。

第 12～14 行 这些内容定义了最小和最大的重传超时和最大重传次数。

图 20.11 给出了一个宏和 RTT 函数的头两个。

```
 1 #include    "unprtt.h"
 2 int       rtt_d_flag = 0;     /* debug flag; can be set nonzero by caller */
 3 /*
 4  * Calculate the RTO value based on current estimators:
 5  *      smoothed RTT plus four times the deviation.
 6  */
 7 #define RTT_RTOCALC(ptr) ((ptr)->rtt_srtt + (4.0 * (ptr)->rtt_rttvar))
 8 static float
 9 rtt_minmax(float rto)
10 {
11     if (rto < RTT_RXTMIN)
12         rto = RTT_RXTMIN;
13     else if (rto > RTT_RXTMAX)
14         rto = RTT_RXTMAX;
15     return(rto);
16 }
17 void
18 rtt_init(struct rtt_info *ptr)
19 {
20     struct timeval     tv;
21     Gettimeofday(&tv, NULL);
22     ptr->rtt_base = tv.tv_sec;          /* #sec since 1/1/1970 at start */
23     ptr->rtt_rtt = 0;
24     ptr->rtt_srtt = 0;
25     ptr->rtt_rttvar = 0.75;
26     ptr->rtt_rto = rtt_minmax(RTT_RTOCALC(ptr));
27         /* first RTO at (srtt + (4 * rttvar)) = 3 seconds */
28 }
```

图 20.11 RTT_RTOCALC 宏,rtt_minmax 和 rtt_init 函数[lib/rtt.c]

第 3～7 行 RTT_RTOCALC 宏计算 RTO 为 RTT 估计因子加上 4 倍的平均偏差估计因子。

第 8～16 行 rtt_minmax 保证 RTO 在 unprtt.h 头文件中定义的上下界之中。

第 17～28 行 rtt_init 由 dg_send_recv 在任何分组第一次发送时调用。gettimeofday 返回当前的时间和日期,存放于 timeval 结构中,我们在用 select 时(6.3 节)见过这个结构。我们只保存自 Unix 纪元以来的秒数,也就是从 1970 年 1 月 1 日 00:00:00 协同世界时间

(UTC)开始计时。测得的 RTT 初置为 0,RTT 和平均偏差估计因子则分别置为 0 和 0.75,给出初始 RTO 为 3 秒。

> gettimeofday 函数还不是 Posix.1 的一部分,可能不被一些旧实现所支持。它是 Unix 98 要求的。我们使用它的原因是它已非常普遍并且在许多主机上它能提供毫秒的精度。另一个函数是 Posix.1 的 times 函数,但是它的精度依赖于内核使用的"时钟滴答",通常是每秒一百个滴答,给出 10 毫秒的精度。

图 20.12 给出了其余的三个 RTT 函数。

```
34 uint32_t
35 rtt_ts(struct rtt_info * ptr)
36 {
37     uint32_t  ts;
38     struct timeval    tv;
39     Gettimeofday(&tv, NULL);
40     ts = ((tv.tv_sec - ptr->rtt_base) * 1000) + (tv.tv_usec / 1000);
41     return(ts);
42 }
43 void
44 rtt_newpack(struct rtt_info * ptr)
45 {
46     ptr->rtt_nrexmt = 0;
47 }
48 int
49 rtt_start(struct rtt_info * ptr)
50 {
51     return((int) (ptr->rtt_rto + 0.5));     /* round float to int */
52             /* return value can be used as: alarm(rtt_start(&foo)) */
53 }
```

图 20.12　rtt_ts、rtt_newpack 和 rtt_start 函数[lib/rtt.c]

第 34～42 行　rtt_ts 返回当前的时间戳,让调用者把它作为一个无符号 32 位整数存入要发送的数据报中。我们从 gettimeofday 中获得当前时间和日期,接着减去调用 rtt_init 时的秒数(其值被保存在 rtt_base 中)。我们将它变为毫秒并且也将 gettimeofday 返回值变为毫秒。时间戳就是以毫秒为单位的前后这两个值之和。

两次 rtt_ts 调用之差是两次调用之间的毫秒数。但我们是在一个无符号 32 位整数中保存毫秒的时间戳,而不是在 timeval 结构中。

第 43～47 行　rtt_newpack 只是置重传计数为 0。当新的分组第一次发送时应调用这个函数。

第 48～53 行　rtt_start 返回当前的 RTO 秒数。这个返回值就可以作为 alarm 的参数。

rtt_stop 在图 20.13 中给出。在一个应答收到后,调用它去更新 RTT 估计因子并计算新的 RTO。

```
62 void
63 rtt_stop(struct rtt_info * ptr, uint32_t ms)
64 {
65     double     delta;
66     ptr->rtt_rtt = ms / 1000.0;     /* measured RTT in seconds */
```

```
67    /*
68     * Update our estimators of RTT and mean deviation of RTT.
69     * See Jacobson's SIGCOMM '88 paper, Appendix A, for the details.
70     * We use floating point here for simplicity.
71     */
72    delta = ptr->rtt_rtt - ptr->rtt_srtt;
73    ptr->rtt_srtt += delta / 8;        /* g = 1/8 */
74    if (delta < 0.0)
75        delta = -delta;                /* |delta| */
76    ptr->rtt_rttvar += (delta - ptr->rtt_rttvar) / 4;      /* h = 1/4 */
77    ptr->rtt_rto = rtt_minmax(RTT_RTOCALC(ptr));
78 }
```

图 20.13 rtt_stop 函数:更新 RTT 估计因子并计算新的 RTO[lib/rtt.c]

第 62～78 行 第二个参数是测得的 RTT,它是由调用者通过从当前时间戳(rtt_ts)中减去收到的应答中的时间戳获得的。接着应用本节开始处的方程,保存 rtt_srtt、rtt_rttvar 和 rtt_rto 的新值。

最后一个函数 rtt_timeout 在图 20.14 中给出。当重传定时器超时时,这个函数被调用。

```
83 int
84 rtt_timeout(struct rtt_info *ptr)
85 {
86    ptr->rtt_rto *= 2;          /* next RTO */
87    if (++ptr->rtt_nrexmt > RTT_MAXNREXMT)
88        return(-1);             /* time to give up for this packet */
89    return(0);
90 }
```

图 20.14 rtt_timeout 函数:应用指数回退[lib/rtt.c]

第 86 行 当前的 RTO 加倍:这就是指数回退。

第 87～89 行 如果已经达到了重传的上限,就给调用者返回－1,让其放弃,否则返回 0。

作为例子,我们的客户在一个工作日早上对两个跨越因特网的不同 echo 服务器运行了两次。500 行数据被发送到每个服务器。去往第一个服务器有 8 个分组丢失,去往第二个服务器 16 个分组丢失。去往第二个服务器丢失的分组中,有一个分组连着丢失两次:也就是在应答收到之前,这个分组不得不重传两次。所有其他的丢失分组都是一次重传处理的。我们可以通过输出每个收到分组的序列号来验证这些分组确实丢失了。如果一个分组仅是延迟了而并没有丢失,那么重传后,客户会收到两个应答:一个对应于延迟了的初始发送,另一个对应于重传。注意,当重传时,我们并不能区别是客户的请求还是服务器的应答丢失了。

在本书的第一版,作者编写了一个随机丢弃分组的 UDP 服务器程序来测试这个客户程序。现在不再需要了:我们只要对一个跨越因特网的服务器运行客户程序就可以了,我们几乎可以保证总有些分组会丢失!

20.6 捆绑接口地址

我们的 get_ifi_info 函数的一个典型用处是那些需监视主机上所有接口的 UDP 应用程

序，它们希望知道数据报是在何时及哪个接口上到达的。该函数允许接收程序获悉 UDP 数据报的目的地址，因为那个地址决定了数据报递送的套接口，即使主机不支持 IP_RECVDSTADDR 套接口选项也一样。

> 回想我们在 20.2 节末尾的讨论。如果主机使用了普遍的弱端系统模型，目的 IP 地址可能与接收接口的 IP 地址不同。这种情况下，我们所能确定的是数据报的目的地址，它不必是分配给接收接口的地址。要确定接收接口，需要 IP_RECVIF 或 IPV6_PKTINFO 套接口选项。
>
> 在 19.11 节中，我们曾用 SNTP 例子给出了捆绑所有接口地址的方法。

图 20.15 给出了使用该技术的一个简单例子的前半部分，这个例子中的 UDP 服务器捆绑所有单播地址、所有广播地址及通配地址。

调用 get_ifi_info 获取接口信息

第 11～12 行　get_ifi_info 获取所有接口的所有 IPv4 地址，包括别名。程序接着遍历每一次返回的 ifi_info 结构。

创建 UDP 套接口并捆绑单播地址

第 13～20 行　创建一个 UDP 套接口并在其上捆绑单播地址。我们还设置了 SO_REUSEADDR 套接口选项，因为我们要给所有的 IP 地址捆绑同一个端口(SERV_PORT)。

> 并非所有的实现需要设置这个套接口选项。例如，源自 Berkeley 的实现不需要这个选项，并且允许重新 bind 已经绑定的端口，只要新捆绑的 IP 地址(a)不是通配地址，(b)与跟此端口一块绑定的所有其他 IP 地址都不同。但是 Solaris 2.5 中第二次跟同一端口 bind 单播地址需要这个选项才会成功。

```
1 #include    "unpifi.h"
2 void  mydg_echo(int, SA *, socklen_t, SA *);
3 int
4 main(int argc, char **argv)
5 {
6     int       sockfd;
7     const int on = 1;
8     pid_t pid;
9     struct ifi_info *ifi, *ifihead;
10    struct sockaddr_in *sa, cliaddr, wildaddr;
11    for (ifihead = ifi = Get_ifi_info(AF_INET, 1);
12         ifi != NULL; ifi = ifi->ifi_next) {
13              /* bind unicast address */
14        sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
15        Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
16        sa = (struct sockaddr_in *) ifi->ifi_addr;
17        sa->sin_family = AF_INET;
18        sa->sin_port = htons(SERV_PORT);
19        Bind(sockfd, (SA *) sa, sizeof(*sa));
20        printf("bound %s\n", Sock_ntop((SA *) sa, sizeof(*sa)));
21        if ( (pid = Fork()) == 0) {          /* child */
22            mydg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr), (SA *) sa);
```

```
23              exit(0);        /* never executed */
24          }
```

图 20.15 捆绑所有地址的 UDP 服务器程序的前半部分[advio/udpserv03.c]

给这个地址 fork 子进程

第 21～24 行 fork 一个子进程并由这个子进程调用 mydg_echo 函数。这个函数等待任意的数据报到达这个套接口并且将它回送给发送者。

图 20.16 给出了 main 函数的后半部分,它处理广播地址。

捆绑广播地址

第 25～42 行 如果接口支持广播,则创建一个 UDP 套接口,并且将广播地址捆绑在其上。这次,我们允许 bind 的 EADDRINUSE 错误,因为如果在同一子网上的接口有多个地址(别名),那么每个不同的单播地址将会有相同的广播地址。我们在图 16.6 后面给出过一个例子。这种情形下,我们只希望第一次 bind 成功。

```
25          if (ifi->ifi_flags & IFF_BROADCAST) {
26                  /* try to bind broadcast address */
27              sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
28              Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
29              sa = (struct sockaddr_in *) ifi->ifi_brdaddr;
30              sa->sin_family = AF_INET;
31              sa->sin_port = htons(SERV_PORT);
32              if (bind(sockfd, (SA *) sa, sizeof(*sa)) < 0) {
33                  if (errno == EADDRINUSE) {
34                      printf("EADDRINUSE: %s\n",
35                             Sock_ntop((SA *) sa, sizeof(*sa)));
36                      close(sockfd);
37                      continue;
38                  } else
39                      err_sys("bind error for %s",
40                              Sock_ntop((SA *) sa, sizeof(*sa)));
41              }
42              printf("bound %s\n", Sock_ntop((SA *) sa, sizeof(*sa)));
43              if ( (pid = Fork()) == 0) {         /* child */
44                  mydg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr),
45                            (SA *) sa);
46                  exit(0);            /* never executed */
47              }
48          }
49      }
```

图 20.16 捆绑所有地址的 UDP 服务器程序的后半部分[advio/udpserv03.c]

fork 子进程

第 43～47 行 fork 一个子进程并由它调用 mydg_echo 函数。

main 函数的最后部分在图 20.17 中给出。这段代码 bind 通配地址来处理不是我们绑定的单播和广播地址。仅有的应该到达这个套接口的数据报应是那些到受限广播地址(255.255.255.255)的数据报。

创建套接口,并捆绑通配地址

第 50～62 行 创建一个 UDP 套接口,设置 SO_REUSEADDR 套接口选项,给它捆绑

通配 IP 地址。派生一个子进程，由它调用 mydg_echo 函数。

main 函数终止

第 63 行　main 函数终止，服务器作为派生的子进程继续执行。

由所有子进程执行的 mydg_echo 函数在图 20.18 中给出。

```
50          /* bind wildcard address */
51      sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
52      Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
53      bzero(&wildaddr, sizeof(wildaddr));
54      wildaddr.sin_family = AF_INET;
55      wildaddr.sin_addr.s_addr = htonl(INADDR_ANY);
56      wildaddr.sin_port = htons(SERV_PORT);
57      Bind(sockfd, (SA *) &wildaddr, sizeof(wildaddr));
58      printf("bound %s\n", Sock_ntop((SA *) &wildaddr, sizeof(wildaddr)));
59      if ( (pid = Fork()) == 0) {          /* child */
60          mydg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr), (SA *) sa);
61          exit(0);            /* never executed */
62      }
63      exit(0);
64  }
```

图 20.17　捆绑所有地址的 UDP 服务器程序的最后部分[advio/udpserv03.c]

```
65  void
66  mydg_echo(int sockfd, SA *pcliaddr, socklen_t clilen, SA *myaddr)
67  {
68      int     n;
69      char    mesg[MAXLINE];
70      socklen_t   len;
71      for ( ; ; ) {
72          len = clilen;
73          n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
74          printf("child %d, datagram from %s", getpid(),
75                  Sock_ntop(pcliaddr, len));
76          printf(", to %s\n", Sock_ntop(myaddr, clilen));
77          Sendto(sockfd, mesg, n, 0, pcliaddr, len);
78      }
79  }
```

图 20.18　mydg_echo 函数[advio/udpserv03.c]

新参数

第 65～66 行　传给这个函数的第四个参数是绑定在这个套接口上的 IP 地址。这个套接口应该只接收目的地址为那个 IP 地址的数据报。如果 IP 地址是通配的，这个套接口应只能接收与绑定同一端口的其他套接口匹配的数据报。

读入数据报并且返回应答

第 71～78 行　数据报用 recvfrom 读入，用 sendto 发回给客户。这个函数还输出客户绑定在套接口上的 IP 地址。

在创建主机 bsdi 上 ef0 以太网接口的三个别名地址之后，我们在其上运行这个程序。三个别名的主机 ID 分别为 50、51 和 52，但是所有别名都有相同的广播地址 206.62.266.63。

```
bsdi % udpserv03
bound 206.62.226.66.9877          we0接口的单播地址
bound 206.62.226.95.9877          we0接口的广播地址
bound 206.62.226.35.9877          ef0接口的主单播地址
bound 206.62.226.63.9877          ef0接口的广播地址
bound 206.62.226.50.9877          第一个单播别名
EADDRINUSE: 206.62.226.63.9877
bound 206.62.226.51.9877          第二个单播别名
EADDRINUSE: 206.62.226.63.9877
bound 206.62.226.52.9877          第三个单播别名
EADDRINUSE: 206.62.226.63.9877
bound 127.0.0.1.9877              回馈接口
bound 0.0.0.0.9877                通配地址
```

注意，正像我们期望的，三个别名的广播地址bind失败了。我们可以用netstat去检查所有这些套接口绑定了上面所指明的IP地址和端口号：

```
bsdi % netstat -na | grep 9877
udp        0      0  *.9877                 *.*
udp        0      0  127.0.0.1.9877         *.*
udp        0      0  206.62.226.52.9877     *.*
udp        0      0  206.62.226.51.9877     *.*
udp        0      0  206.62.226.50.9877     *.*
udp        0      0  206.62.226.63.9877     *.*
udp        0      0  206.62.226.35.9877     *.*
udp        0      0  206.62.226.95.9877     *.*
udp        0      0  206.62.226.66.9877     *.*
```

我们应该指出，给每个套接口创建一个子进程的设计是为了简单，也可能有另外的设计。例如，为了减少进程数，程序可以用select管理所有的描述字而不必调用fork。该设计的问题是增加了编码的复杂性。尽管对所有的描述字用select很简单，但是我们必须维护从每个描述字到绑定的IP地址的某种类型的映射(可能是一个结构数组)，这样当从一个套接口读入一个数据报时我们才可能输出其目的IP地址。通常，使用单独的进程或线程处理单个操作或单个描述字比用一个进程处理多个不同操作或多个描述字要简单。

20.7 并发UDP服务器

多数的UDP服务器程序是迭代执行的：服务器等待一个客户请求，读入请求，处理请求，送回应答，接着等待下一个客户请求。但是，当处理客户请求要很长时间时，就要有一定形式的并发性。

“长时间”的定义是指当服务于当前客户时另一个客户被认为等待了相当长的时间。例如，如果两个客户请求彼此相差10ms到达，处理一个客户要5秒的时间，那么第二个请求就要花大约10秒来等待应答，而不是请求一到就处理的5秒。

用TCP只是很简单地fork一个新的子进程(或者像我们将要在第23章中看到的创建一个新线程)，让子进程处理新客户。当使用TCP时，能够简化服务器并发性的根源在于每个客户连接都是唯一的：也就是说TCP套接口对于每个连接都是唯一的。但是在UDP中我们必须处理两种不同类型的服务器。

1. 第一种是简单的 UDP 服务器，它读入一个客户请求，发送应答，接着与这个客户就无关了。在这种情形里，读客户请求的服务器可以 fork 一个子进程去处理请求。“请求”(也就是数据报的内容和保存在客户协议地址中的套接口地址结构)通过从 fork 得来的内存映像传递给子进程。子进程接着直接给客户发送它的应答。
2. 第二种是与客户交换多个数据报的 UDP 服务器。问题是客户只知道服务器的端口是服务器众所周知的端口。客户发送请求的第一个数据报到这个端口，但是服务器又怎么能区分这是那个客户的后继数据报还是新请求呢？这种问题的典型解决方法是让服务器给每个客户创建一个新的套接口，bind 一个临时端口到那个套接口，并且对所有的回答都用这个套接口。这要求客户看一下服务器第一个应答中的端口号，并且向那个端口发送请求的后继数据报。

第二种类型的 UDP 服务器的一个例子是 TFTP(简化文件传输协议)。使用 TFTP 传送一个文件通常需要许多数据报(成百上千，依赖于文件长度)，因为该协议给每个数据报只发送 512 字节。客户向服务器的众所周知端口(69)发送一个数据报以指定要发送或接收的文件。服务器读入请求，但是从另外一个由它创建并绑定临时端口的套接口发送它的应答。所有这个文件的客户和服务器间的后继数据报使用这个新的套接口。这允许当这个文件传输进行时(可能 2 分钟或更长的时间)，主 TFTP 服务器继续去处理到达端口 69 的其他客户请求。

如果是一个独立的 TFTP 服务器(也就是说不是由 inetd 启动)，我们将情形展示在图 20.19 中。我们假设子进程给新套接口捆绑的临时端口是 2134。

如果用 inetd，这就要多一个步骤。回想图 12.6，多数 UDP 服务器指定等待标志为 wait。在图 12.10 后的叙述中，我们说这导致 inetd 停止在其套接口上选择，直到它的子进程终止，从而允许它的子进程读入到达这个套接口的数据报。图 20.20 展示了所需的步骤。

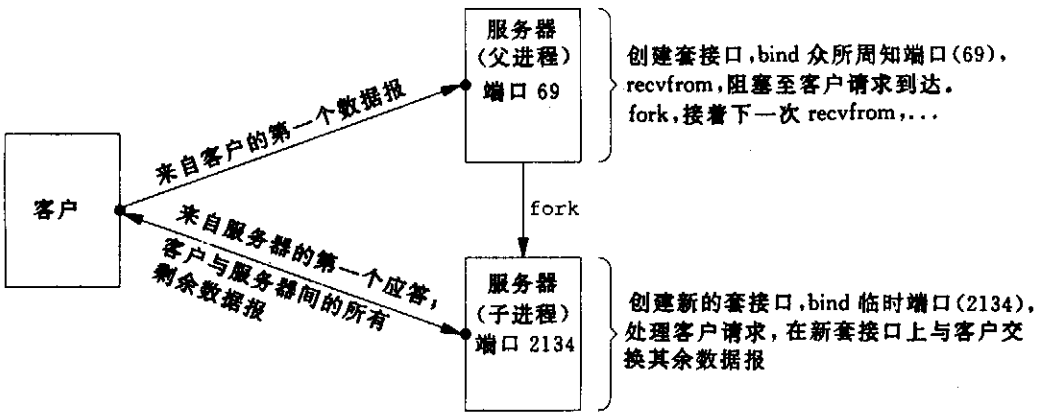


图 20.19　在独立运行的并发 UDP 服务器中所需的处理

作为 inetd 子进程的 TFTP 服务器调用 recvfrom 读入客户请求。它接着 fork 一个自己的子进程，这个子进程将处理客户请求。TFTP 服务器接着调用 exit，给 inetd 发 SIGCHLD 信号，通知 inetd 重新在绑定 UDP 端口 69 的套接口上 select。

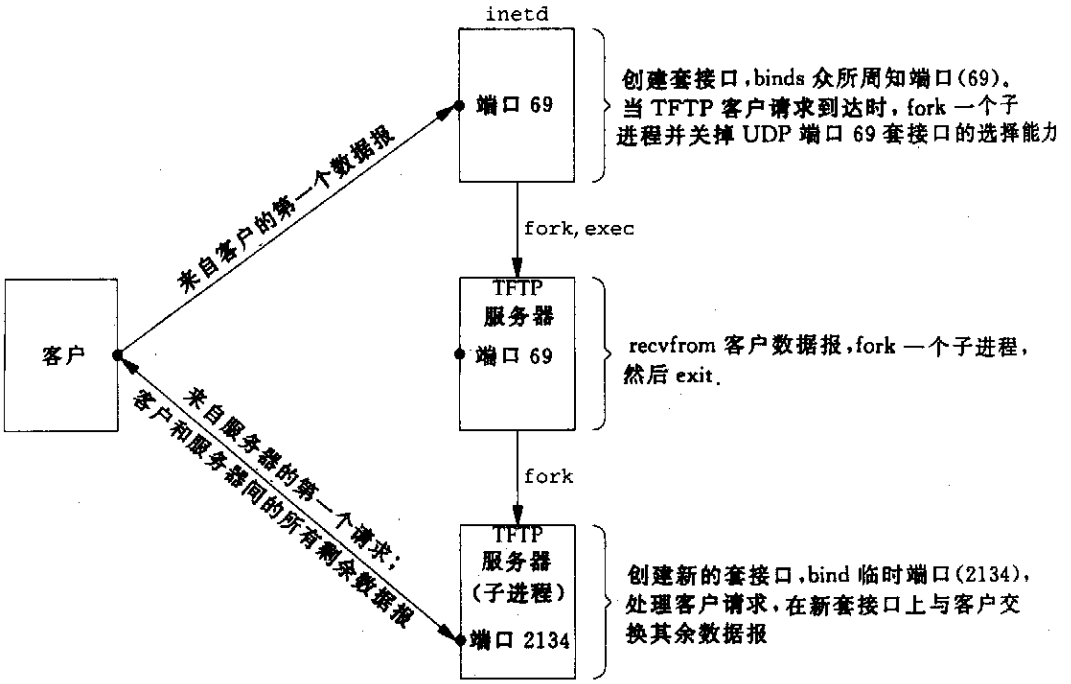


图 20.20 由 inetd 启动的 UDP 并发服务器

20.8 IPv6 分组信息

IPv6 允许应用程序对外出的数据报指定最多四条信息：

1. 源 IPv6 地址
2. 外出接口索引
3. 外出跳限
4. 下一跳地址

这些信息是作为辅助数据使用 sendmsg 发送的。对于收到的分组可以返回三条类似的信息，它们是作为辅助数据由 recvmsg 返回的：

1. 目的 IPv6 地址
2. 到达接口索引
3. 到达跳限

图 20.21 总结了这些辅助数据的内容，我们马上就会讨论到。

in6_pktinfo 结构含有发送数据报的源 IPv6 地址和外出接口索引，或者含有收到的数据报的目的 IPv6 地址和到达接口索引：

```
struct in6_pktinfo {
  struct in6_addr ipi6_addr;     /* src/dst IPv6 address */
  int             ipi6_ifindex;  /* send/recv interface index */
};
```

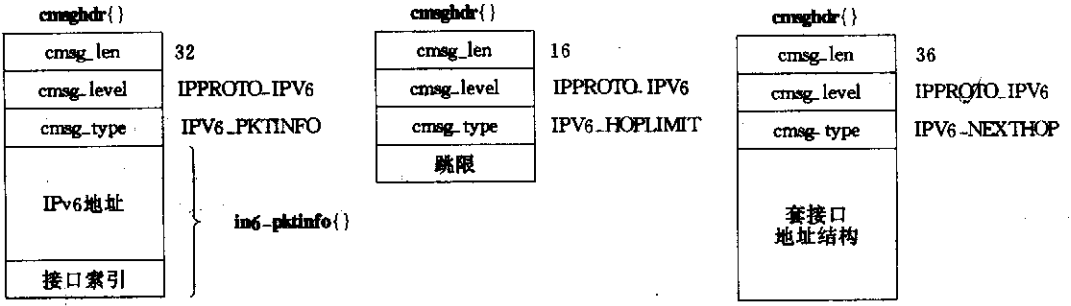


图 20.21　IPv6 分组信息的辅助数据

这个结构是通过包含<netinet/in.h>头文件而定义的。在包含这些辅助数据的 cmsghdr 结构中，cmsg_level 成员将是 IPPROTO_IPV6，cmsg_type 成员将是 IPV6_PKTINFO，数据的第一个字节将是 in6_pktinfo 结构的第一个字节。在图 20.21 的例子中，我们假设 cmsghdr 结构和数据之间没有填充字节，并且整数是 4 字节。

发送该信息不要什么特别的要求：只是给 sendmsg 指定作为辅助数据的控制信息。但是，只有应用进程打开了 IPV6_PKTINFO 套接口选项，这个信息才会由 recvmsg 作为辅助数据返回。

外出和到达接口

就像我们在 17.6 节中讨论的，IPv6 节点上的接口是由小正整数标识的。没有接口会被赋予索引 0。当指定外出接口时，如果 ipi6_ifindex 的值是 0，则由内核来选择发出接口。如果应用进程给多播数据报指定了外出接口，对这个数据报来说，由辅助数据指定的接口将覆盖由 IPV6_MULTICAST_IF 套接口选项指定的接口。

源和目的 IPv6 地址

源 IPv6 地址通常是调用 bind 指定的。但是，同数据一起指定源地址可能需要较少的开销。这个选项还允许服务器保证它的应答的源地址与客户请求的目的地址相同，有些客户需要这个特性，但用 IPv4(习题 20.4)很难实现。

当指定源 IPv6 地址作为辅助数据时，如果 in6_pktinfo 结构的 ipi6_addr 成员是 IN6ADDR_ANY_INIT，那么(a)如果有地址绑定在这个套接口上，它就被用作源地址或者(b)如果当前没有地址绑定在套接口上，内核将会选择源地址。如果 ipi6_addr 成员不是未指定地址，但是套接口已经绑定源地址，那么仅对这次输出操作，ipi6_addr 值覆盖绑定的源地址。内核将会验证要求的源地址确实是分配给本节点的单播地址。

当 in6_pktinfo 结构由 recvmsg 作为辅助数据返回时，ipi6_addr 成员含有取自所收到分组的目的 IPv6 地址。在概念上，这与 IPv4 的 IP_RECVDSTADDR 套接口选项相似。

指定和接收跳限

对于单播数据报，外出跳限通常由 IPV6_UNICAST_HOPS 套接口选项指定(7.8 节)；对于多播数据报，外出跳限通常由 IPV6_MULTICAST_HOPS 套接口选项指定(19.5 节)。对于单个的输出操作，给单播地址和多播地址指定跳限作为辅助数据会覆盖内核的缺省值或者以前指定的值。对于像 traceroute 的程序和一类需验证收到的跳限为 255 的 IPv6 应用

程序(例如分组没有被转发过),返回收到的跳限是有用的。

只有应用进程打开了 IPV6_HOPLIMIT 套接口选项,收到的跳限才会由 recvmsg 作为辅助数据返回。在包含辅助数据的 cmsghdr 结构中,cmsg_level 成员将是 IPPROTO_IPV6,cmsg_type 成员将是 IPV6_HOPLIMIT,并且数据的第一个字节将是整数跳限的第一个字节。我们在图 20.21 展示了它。要意识到,作为辅助数据返回的值是从收到的数据报中取得的实际值,而由 IPV6_UNICAST_HOPS 套接口选项的 getsockopt 返回的值是内核在套接口上给外出数据报使用的缺省值。

指定外出跳限没有什么特别的要做:只是给 sendmsg 指定控制信息。跳限的通常值是在 0～255 之间,包含 0 和 255,如果值为－1,则通知内核使用缺省值。

> 跳限没有包含在 in6_pktinfo 结构中有如下原因:一些 UDP 服务器想通过以下方式来响应客户请求,即发送应答的接口与接收请求的接口相同,而且所用 IPv6 源地址与请求的 IPv6 目的地址相同。为做到这一点,应用进程可以只打开 IPV6_PKTINFO 套接口选项,接着用使用 recvmsg 收到的控制信息作为 sendmsg 的外出控制信息。应用进程根本不用检查或修改 in6_pktinfo 结构。要是在该结构中含有跳限,应用进程将不得不分析收到的控制信息并改变跳限,因为收到的跳限并不是外出分组所希望的值。

指定下一跳地址

IPV6_NEXTHOP 辅助数据对象作为一个套接口地址结构给数据报指定了下一跳的地址。在包含这个辅助数据的 cmsghdr 的结构中,cmsg_level 成员是 IPPROTO_IPV6,cmsg_type 成员是 IPV6_NEXTHOP,数据的第一字节是套接口地址结构的第一个字节。

在图 20.21 中我们展示了这个辅助数据对象的一个例子,这里假设套接口地址结构是一个 24 字节的 sockaddr_in6 结构。在这种情况下,由那个地址指定的节点必须是发送主机的邻居。如果那个地址与数据报目的地址相同,这就与已有的 SO_DONTROUTE 套接口选项相同。设置这个选项需要超级用户权限。

20.9 小 结

有一些应用程序想知道一个 UDP 数据报的目的 IP 地址和接收接口。打开 IP_RECVDSTADDR 和 IP_RECVIF 套接口选项就可以随每个数据报返回作为辅助数据的这些信息。对于 IPv6 套接口,类似的信息同收到的跳限一起通过打开 IPV6_PKTINFO 套接口选项返回。

尽管 TCP 提供的所有特性 UDP 并不提供,有时还是要用 UDP 的。广播或多播必须用 UDP。UDP 可以用于简单的请求-应答情形,但是某种形式的可靠性必须加入应用程序中。UDP 不应用于海量数据传输。

在 20.5 节中,通过使用超时和重传发现丢失分组,从而增加了 UPD 客户程序的可靠性。通过对每个分组增加时间戳并保持两个估计因子:RTT 和它的平均偏差,我们动态地修改了重传超时值。我们还增加了一个序列号来验证指定的应答是所希望的应答。我们的客户程序仍使用简单的停等协议,但这是可使用 UDP 的应用程序类型。

20.10　习　题

20.1　在图 20.18 中为什么有两次对 printf 的调用？

20.2　dg_send_recv(图 20.8 和 20.9)能否返回 0？

20.3　重新编写 dg_send_recv，使用 select 和它的定时器，而不用 alarm、SIGALRM、sigsetjmp 和 siglongjmp。

20.4　IPv4 服务器如何保证它的应答的源地址与客户请求的目的地址相同(例如，类似于由 IPV6_PKTINFO 套接口选项的功能)？

20.5　图 20.6 中的 main 函数是跟 IPv4 协议相关的，重新编写它成为协议无关的。要求用户指定一个或两个命令行参数，第一个是一个可选的 IP 地址(例如，0.0.0.0 或 0::0)，第二个是要求的端口号。接着调用 udp_client 来获取套接口地址结构的地址族、端口号和长度。因为 udp_client 并没有给 getaddrinfo 指定 AI_PASSIVE 线索，如果像建议的那样不指定主机名参数调用 udp_client，将会发生什么事情？

20.6　更改 RTT 函数以输出每个 RTT，然后对跨越因特网的 echo 服务器运行图 20.6 中的客户程序。修改 dg_send_recv 函数以输出每个收到的序列号。绘出结果 RTT、RTT 的估计因子及其平均偏差的估计因子。

第21章 带外数据

21.1 概 述

许多传输层有带外数据(out-of-band data)的概念,有时也称为加速数据(expedited data)。这个想法是指连接双方中的一方发生重要事情,想要迅速地通知对方。这里"迅速"地意指这种通知应该在已经排队等待发送的任何"普通"(有时称为"带内")数据之前发送。也就是带外数据设计为比普通数据有更高的优先级。但是带外数据是映射到现有的连接中的,而不是在客户机和服务器间再用一个连接。

不幸的是,实际上几乎每个传输层都有一个不同的带外数据的实现。在本章,我们把注意力集中在TCP的带外数据模型,并提供了众多如何由套接口API处理带外数据的例子,接着用它去编写一些简单的客户-服务器心博函数,可用来检测何时对方进程崩溃或不可达。

21.2 TCP带外数据

TCP没有直正的带外数据,而是提供了一个我们要讨论的紧急模式(urgent mode)。假设一个进程已向一个TCP套接口写入了N字节数据,并且这些数据被TCP放入套接口发送缓冲区等待发送给对方。我们在图21.1中展示了这种状态,并且标记了从1到N的数据字节。

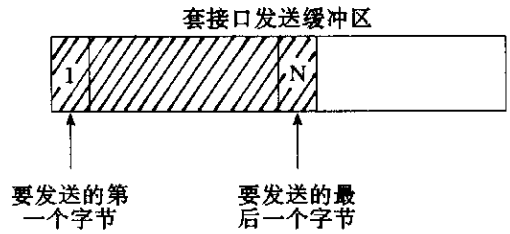


图21.1 包含要发送数据的套接口发送缓冲区

进程现在使用send函数和MSG_OOB标志发送一个包含ASCII字符a的带外数据字节:

```
send(fd,"a",1,MSG_OOB);
```

TCP将数据放置在套接口发送缓冲区的下一个可用位置,并设置这个连接的TCP紧急指针(urgent pointer)为下一个可用位置。我们在图21.2中展示了这种状态,并且标记带外字节为"OOB"。

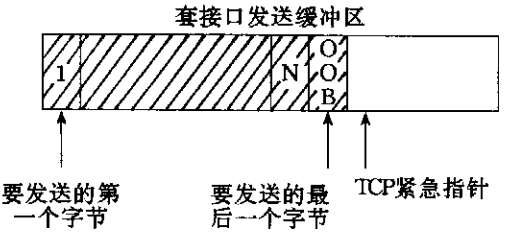


图 21.2　应用进程写入 1 字节带外数据后的套接口发送缓冲区

TCP 紧急指针有一个比用 MSG_OOB 标志写入的数据多一字节的序列号。正如 TCPv1 第 292～296 页讨论的，这是历史的既成事实，现在被所有的实现模仿。只要接收和发送 TCP 在 TCP 紧急指针的解释上达成一致，就没有问题。在 7.9 节我们提到过，新的 TCP_STDURG 套接口选项可以改变紧急指针指向哪个字节的解释。这个选项永远不必设置。

给定如图 21.2 所示的 TCP 套接口发送缓冲区状态，由 TCP 发送的下一个分节将会在 TCP 头部中设置 URG 标志，并且头部中的紧急偏移(urgent offset)字段也将指向带外字节后的字节。但是这个分节可以含有也可以不含有我们标记为 OOB 的字节。是否发送 OOB 字节取决于在套接口发送缓冲区中它前面的字节数、TCP 发送给对方的分节长度以及对方通告的当前窗口。

我们使用了紧急指针和紧急偏移这两个术语。在 TCP 的层次，它们是不同的。在 TCP 头部中的 16 位值被称为紧急指针，它必须加上头部中的序列号字段以获得 32 位的紧急指针。只有在被称为 URG 标志的位设置时，TCP 才会检查紧急偏移。从编程角度看，我们不需要担心这个细节，称呼 TCP 紧急指针就可以。

这是 TCP 紧急模式的一个重要特点：TCP 头部指出发送者进入了紧急模式(也就是与紧急偏移一起设置的 URG 标志)，但是由紧急指针所指的实际数据的字节却不一定被送出。其实，如果发送 TCP 因流控停止了(接收者的套接口接收缓冲区满了，所以它向发送者 TCP 通告一个为 0 的窗口)，就像我们将在图 21.10 和 21.11 中给出的那样，紧急通知不带任何数据地被送出(TCPv2 第 1016 页～1017 页)。这是为什么应用程序使用 TCP 的紧急模式(也就是带外数据)的一个原因：即便数据流因 TCP 的流控停止了，紧急通知也总会被发送到对方的 TCP。

如果我们发送多字节的带外数据又会如何呢，就像：

```
send(fd,"abc",3,MSG_OOB);
```

在这个例子中，TCP 的紧急指针指向最后那个字节后面；也就是说最后那个字节(字母 c)被认为是带外字节。

上面我们已经讲述了带外数据的发送。下面让我们从接收者的角度看一下。

1. 当 TCP 收到了一个设置 URG 标志的分节时，紧急指针被检查，看它是否指向新的带外数据。也就是说，这是否首次在从发送者到接收者的数据流中 TCP 的紧急模式

指向这个特殊字节。发送者TCP发送多个包含URG标志,但紧急指针却指向相同数据字节的分节(通常是在一小段时间内),这种情况相当普遍。这些分节中只有第一个导致接收进程被通知有新带外数据到达。

2. 当新紧急指针到达时,接收进程被通知。首先,SIGURG信号发送给套接口的属主,这里假设已调用fcntl或ioctl给套接口建立了属主(图7.15),还假设属主进程已为这个信号建立了信号处理程序。其次,如果进程阻塞在select调用中,等待这个套接口描述字有一个异常条件,那么select返回。

   当一个新紧急指针到达时,这两个对接收进程可能的通知就会发生,而不管是否由紧急指针指向的实际数据字节已经到达接收者TCP。

3. 当由紧急指针指向的实际数据字节到达接收者TCP时,这个数据字节可以被拉出带外或继续在线存放。SO_OOBINLINE套接口选项缺省时是不设置的,所以这个数据字节并不放入套接口接收缓冲区。相反,这个数据字节被放到这个连接的单独的1字节带外缓冲区(TCPv2第986～988页)。进程从这个特别的1字节缓冲区中读出的仅有方法是调用recv、recvfrom或者recvmsg并且指定MSG_OOB标志。

   然而如果进程设置了SO_OOBINLINE套接口选项,那么由TCP的紧急指针指向的单字节数据将被留在通常的套接口接收缓冲区里。在这种情况下,进程不能指定MSG_OOB标志去读带外数据字节。这个进程通过检查本连接的带外标记(out-of-band mark)来获悉何时它到达了这个数据字节,就像在21.3节中将描述的那样。

一些错误是可能的:

1. 如果进程请求带外数据(例如指定MSG_OOB标志)但是对方尚未发送,将会返回EINVAL。
2. 如果进程已被通知对方发送了带外字节(例如通过SIGURG或select),进程试图去读它,但是那个字节还没有到达,将会返回EWOULDBLOCK。进程在这时所能做的就是从套接口接收缓冲区中读(如果它没有空间保存读出的数据,可能就得丢弃),以便在缓冲区中腾出空间,这样对方TCP就可以发送带外数据。
3. 如果进程试图多次读相同的带外数据,将会返回EINAVL。
4. 如果进程已经设置了SO_OOBINLINE套接口选项,接着试图通过指定MSG_OOB读带外数据,将会返回EINVAL。

使用SIGURG的简单例子

我们现在给出一个发送和接收带外数据的小例子。图21.3给出了发送程序。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int             sockfd;
6     if (argc != 3)
7         err_quit("usage: tcpsend01 <host> <port#>");
8     sockfd = Tcp_connect(argv[1], argv[2]);
9     Write(sockfd, "123", 3);
10    printf("wrote 3 bytes of normal data\n");
```

```
11      sleep(1);
12      Send(sockfd, "4", 1, MSG_OOB);
13      printf("wrote 1 byte of OOB data\n");
14      sleep(1);
15      Write(sockfd, "56", 2);
16      printf("wrote 2 bytes of normal data\n");
17      sleep(1);
18      Send(sockfd, "7", 1, MSG_OOB);
19      printf("wrote 1 byte of OOB data\n");
20      sleep(1);
21      Write(sockfd, "89", 2);
22      printf("wrote 2 bytes of normal data\n");
23      sleep(1);
24      exit(0);
25  }
```

图 21.3 简单的带外发送程序[oob/tcpsend01.c]

发送 9 个字节，每个输出操作间有一个 1 秒的 sleep。间断的目的是让每次 write 或 send 被作为单个 TCP 分节传送。我们在后面讨论有关带外数据的定时考虑。当我们运行这个程序时，就会看到期待的输出：

```
solaris % tcpsend01 bsdi 9999
wrote 3 byte  of normal data
wrote 1 byte of OOB data
wrote 2 byte of normal data
wrote 1 byte of OOB data
wrote 2 byte of normal data
```

图 21.4 是接收程序。

建立信号处理程序和套接口属主

第 16～17 行 建立 SIGURG 的信号处理程序，并用 fcntl 设置已连接套接口的属主。

> 注意，直到 accept 返回后，我们才建立信号处理程序。带外数据可能在 TCP 完成三路握手后，accept 返回前到达。这样我们就会错过它。但是如果我们在调用 accept 前建立信号处理程序并设置监听套接口的属主(这会传递给已连接套接口)，那么如果带外数据在 accept 返回前到达，我们的信号处理程序中的 connfd 还没有获得数值。如果这种情况对应用程序来说重要，它应该初始化 connfd 为－1，在信号处理程序中检查这个值，并且如果为真，就给主循环置一个标志，在 accept 返回后来检查。

第 18～25 行 进程从套接口中读，输出每个由 read 返回的字符串。当发送者终止连接后，接收者也就终止了。

SIGURG 处理程序

第 27～36 行 我们的信号处理程序调用 printf，通过指定 MSG_OOB 标志读带外字节，然后输出返回的数据。注意，我们在 recv 的调用里请求最大 100 字节，但是我们不久就会看到，只有一个字节会作为带外数据返回。

如同前面所讲,从一个信号处理程序里调用不安全的printf是不受推荐的。我们这样做只是为看一下程序在干什么。

```
1 #include    "unp.h"
2 int         listenfd, connfd;
3 void       sig_urg(int);
4 int
5 main(int argc, char * * argv)
6 {
7     int         n;
8     char        buff[100];
9     if (argc == 2)
10        listenfd = Tcp_listen(NULL, argv[1], NULL);
11    else if (argc == 3)
12        listenfd = Tcp_listen(argv[1], argv[2], NULL);
13    else
14        err_quit("usage: tcprecv01 [ <host> ] <port#>");
15    connfd = Accept(listenfd, NULL, NULL);
16    Signal(SIGURG, sig_urg);
17    Fcntl(connfd, F_SETOWN, getpid());
18    for ( ; ; ) {
19        if ( (n = Read(connfd, buff, sizeof(buff)-1)) == 0) {
20            printf("received EOF\n");
21            exit(0);
22        }
23        buff[n] = 0;      /* null terminate */
24        printf("read %d bytes: %s\n", n, buff);
25    }
26 }
27 void
28 sig_urg(int signo)
29 {
30    int         n;
31    char        buff[100];
32    printf("SIGURG received\n");
33    n = Recv(connfd, buff, sizeof(buff)-1, MSG_OOB);
34    buff[n] = 0;                /* null terminate */
35    printf("read %d OOB byte: %s\n", n, buff);
36 }
```

图21.4 简单的带外接收程序[oob/tcprecv01.c]

以下是运行接收程序,并接着运行图21.3中的发送程序的输出。

```
bsdi % tcprecv01 9999
read 3 bytes: 123
SIGURG received
read 1 OOB byte: 4
read 2 bytes: 56
SIGURG received
read 1 OOB byte: 7
read 2 bytes: 89
received EOF
```

结果正像我们希望的，每次由发送者发送带外数据会对接收者产生 SIGURG，接收者接着读入一个字节的带外数据。

使用 select 的简单例子

我们现在不用 SIGURG 信号而用 select 来重写带外接收者程序。图 21.5 是接收程序。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int         listenfd, connfd, n;
6     char        buff[100];
7     fd_set      rset, xset;
8     if (argc == 2)
9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10    else if (argc == 3)
11        listenfd = Tcp_listen(argv[1], argv[2], NULL);
12    else
13        err_quit("usage: tcprecv02 [ <host> ] <port#>");
14    connfd = Accept(listenfd, NULL, NULL);
15    FD_ZERO(&rset);
16    FD_ZERO(&xset);
17    for ( ; ; ) {
18        FD_SET(connfd, &rset);
19        FD_SET(connfd, &xset);
20        Select(connfd + 1, &rset, NULL, &xset, NULL);
21        if (FD_ISSET(connfd, &xset)) {
22            n = Recv(connfd, buff, sizeof(buff)-1, MSG_OOB);
23            buff[n] = 0;      /* null terminate */
24            printf("read %d OOB byte: %s\n", n, buff);
25        }
26        if (FD_ISSET(connfd, &rset)) {
27            if ( (n = Read(connfd, buff, sizeof(buff)-1)) == 0) {
28                printf("received EOF\n");
29                exit(0);
30            }
31            buff[n] = 0;      /* null terminate */
32            printf("read %d bytes: %s\n", n, buff);
33        }
34    }
35 }
```

图 21.5　(不正确地)使用 select 得到带外数据通知的接收程序[oob/tcprecv02.c]

第 15～20 行　进程调用 select 等待普通数据(读集合 rset)或等待带外数据(异常集合 xset)。每种情况的接收数据都被输出。

当我们运行这个程序并接着运行像前面的同样的发送程序(图 21.3)时，我们遇到了如下错误：

```
bsdi % tcprecv02 8888
read 3 bytes: 123
read 1 OOB byte: 4
recv error: Invalid argument
```

问题是 select 一直指示一个异常条件，直到进程读越过带外数据(TCPv2 第 530～531 页)。因为我们第一次读过带外数据后，内核清除了 1 字节的带外缓冲区，所以我们不能多次读。当我们第二次指定 MSG_OOB 标志调用 recv 时，BSD/OS 返回 EINVAL，而 Solaris 2.5 返回 EAGAIN。(对这情况，Posix.1g 指定 EINVAL 作为错误。)

解决方法是只有在读过普通数据后，才去 select 异常条件。图 21.6 是对图 21.5 的一个修改版，它正确地处理了这种情形。

第 5 行 我们声明了一个叫 justreadoob 的变量来指示我们是否刚读过带外数据。这个标志决定是否 select 异常条件。

第 26～27 行 当我们设置 justreadoob 标志时，我们还必须清除这个描述字在异常集合中的那一位。

程序现在像希望的那样工作了。

21.3 sockatmark 函数

每当接收到带外数据时，就有一个相关联的带外标记。这是发送进程发送带外字节时在发送方普通数据流中的位置。接收进程读套接口时通过调用 sockatmark 函数确定是否在带外标记上。

```
#include <sys/socket.h>
int sockatmark(int sockfd);
                    返回值：如果在带外标记上为 1，不在标记上为 0，出错为－1
```

这个函数是 Posix.1g 创造的。Posix 正在把所有的 ioctl 请求替换为函数。

图 21.7 给出了使用通常的 SIOCATMARK ioctl 实现的这个函数。

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int           listenfd, connfd, n, justreadoob = 0;
 6     char    buff[100];
 7     fd_set        rset, xset;
 8     if (argc == 2)
 9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10     else if (argc == 3)
11         listenfd = Tcp_listen(argv[1], argv[2], NULL);
12     else
13         err_quit("usage: tcprecv03 [ <host> ] <port#>");
14     connfd = Accept(listenfd, NULL, NULL);
15     FD_ZERO(&rset);
16     FD_ZERO(&xset);
17     for ( ; ; ) {
18         FD_SET(connfd, &rset);
19         if (justreadoob == 0)
20             FD_SET(connfd, &xset);
```

```
21          Select(connfd + 1, &rset, NULL, &xset, NULL);
22          if (FD_ISSET(connfd, &xset)) {
23              n = Recv(connfd, buff, sizeof(buff)-1, MSG_OOB);
24              buff[n] = 0;                /* null terminate */
25              printf("read %d OOB byte: %s\n", n, buff);
26              justreadoob = 1;
27              FD_CLR(connfd, &xset);
28          }
29          if (FD_ISSET(connfd, &rset)) {
30              if ( (n = Read(connfd, buff, sizeof(buff)-1)) == 0) {
31                  printf("received EOF\n");
32                  exit(0);
33              }
34              buff[n] = 0;        /* null terminate */
35              printf("read %d bytes: %s\n", n, buff);
36              justreadoob = 0;
37          }
38      }
39  }
```

图 21.6　正确地 select 异常条件的图 21.5 程序修改版[oob/tcprecv03.c]

```
1 #include    "unp.h"
2 int
3 sockatmark(int fd)
4 {
5       int     flag;
6       if(ioctl(fd,SIOCATMARK, &flag) < 0)
7         return (-1);
8       return (flag ! = 0 ? 1 : 0);
9 }
```

图 21.7　使用 ioctl 实现的 sockatmark 函数[lib/sockatmark.c]

不管接收进程在线(SO_OOBINLINE 套接口选项)或是带外(MSG_OOB 标志)接收带外数据,带外标记都能应用。带外标记的常见用法之一是接收者特殊地对待所有的数据,直到标记通过。

例子

我们现在给出一个简单的例子来说明带外标记的以下两个特性。

1. 带外标记总是指向刚好越过普通数据最后一个字节的地方。这意味着,如果带外数据在线接收,那么如果下一个要读的字节是被用 MSG_OOB 标志发送的,sockatmark 就返回真。相反,如果 SO_OOBINLINE 套接口选项没有设置,那么如果下一个字节是跟在带外数据后发送的第一个字节,sockatmark 就返回真。
2. 读操作总是会停在带外标记上(TCPv2 第 519～520 页)。也就是,如果在套接口接收缓冲区中有 100 个字节,但带外标记前只有 5 个字节,进程执行 read 请求 100 字节,则只有标记前的 5 个字节返回。这种在标记处的强制停止允许进程调用 sockatmark 确定是否缓冲区指针在标记处。

图 21.8 是我们的发送程序。它发送 3 字节的普通数据,1 字节带外数据,接着另一字节的普通数据。在每个输出操作之间没有停顿。

图21.9是接收程序。它不使用SIGURG信号或select。相反，它调用sockatmark来确定何时遇到了带外字节。

```
1 #include    "unp.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int           sockfd;
6     if (argc ! = 3)
7         err_quit("usage: tcpsend04 <host> <port#>");
8     sockfd = Tcp_connect(argv [1], argv [2]);
9     Write(sockfd, "123", 3);
10    printf("wrote 3 bytes of normal data\n");
11    Send(sockfd, "4", 1, MSG_OOB);
12    printf("wrote 1 byte of OOB data\n");
13    Write(sockfd, "5", 1);
14    printf("wrote 1 byte of normal data\n");
15    exit(0);
16 }
```

图21.8 发送程序[oob/tcpsend04.c]

```
1 #include    "unp.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int           listenfd, connfd, n, on=1;
6     char  buff[100];
7     if (argc == 2)
8         listenfd = Tcp_listen(NULL, argv[1], NULL);
9     else if (argc == 3)
10        listenfd = Tcp_listen(argv[1], argv[2], NULL);
11    else
12        err_quit("usage: tcprecv04 [ <host> ] <port#>");
13    Setsockopt(listenfd, SOL_SOCKET, SO_OOBINLINE, &on, sizeof(on));
14    connfd = Accept(listenfd, NULL, NULL);
15    sleep(5);
16    for ( ; ; ) {
17        if (Sockatmark(connfd))
18            printf("at OOB mark\n");
19        if ( (n = Read(connfd, buff, sizeof(buff)-1)) == 0) {
20            printf("received EOF\n");
21            exit(0);
22        }
23        buff[n] = 0;        /* null terminate */
24        printf("read %d bytes: %s\n", n, buff);
25    }
26 }
```

图21.9 调用sockatmark的接收程序[oob/tcprecv04.c]

设置 SO_OOBINLINE 套接口选项

第 13 行　我们想在线接收带外数据，所以必须设置 SO_OOBINLINE 套接口选项。但是如果我们等到 accept 返回后再在已连接套接口上设置该选项，那么三路握手已经完成并且带外数据可能已经到达。所以我们必须给监听套接口设置这个选项，因为我们知道所有的套接口选项会从监听套接口传递给已连接套接口(7.4 节)。

连接接受后 sleep

第 14～15 行　连接接受后，接收者 sleep 以接收发送者送来的所有数据。这允许我们去展示 read 停在带外标记上，即使额外的数据已在套接口接收缓冲区中。

读入来自发送者的所有数据

第 16～25 行　程序循环调用 read，输出收到的数据。但是在读之前，用 sockatmark 检查是否缓冲区指针在带外标记处。

当我们运行这个程序时，得到如下输出：

```
bsdi % tcprecv04 6666
read 3 bytes: 123
at OOB mark
read 2 bytes: 45
recvived EOF
```

尽管所有的数据都已经由接收者 TCP 收到(因为接收进程调用了 sleep)，但是当第一次调用 read 时，只有 3 个字节返回，因为遇到了标记。下一次读的字节是带外数据(值为 4)，因为我们告诉过内核将带外数据在线放置。

例子

我们现在给出另一个简单例子来说明前面提到的带外数据的两个附加特性。

1. TCP 发送带外数据的通知(它的紧急指针)，即使它因流控停止了数据的发送。
2. 在带外数据到来之前，接收进程可得到指示：发送者已经送出了带外数据(用 SIGURG 信号或通过 select)。如果接收进程接着调用指定 MSG_OOB 的 recv，而带外数据却尚未到达，EWOULDBLOCK 错误就会返回。

图 21.10 是发送程序。

第 9～19 行　该进程设置它的套接口发送缓冲区大小为 32768，写 16384 字节的普通数据后接着 sleep 5 秒。我们不久将会看到接收者设置的套接口接收缓冲区大小为 4096，所以发送者的这些操作保证发送 TCP 填满接收者的套接口接收缓冲区。发送者接着发送 1 字节的带外数据，后面跟着 1024 字节的普通数据，然后终止。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int       sockfd, size;
6     char      buff[16384];
7     if (argc != 3)
8         err_quit("usage: tcpsend04 <host> <port#>");
9     sockfd = Tcp_connect(argv[1], argv[2]);
```

```
10      size = 32768;
11      Setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &size, sizeof(size));
12      Write(sockfd, buff, 16384);
13      printf("wrote 16384 bytes of normal data\n");
14      sleep(5);
15      Send(sockfd, "a", 1, MSG_OOB);
16      printf("wrote 1 byte of OOB data\n");
17      Write(sockfd, buff, 1024);
18      printf("wrote 1024 bytes of normal data\n");
19      exit(0);
20  }
```

图 21.10 发送程序[oob/tcpsend05.c]

图 21.11 给出了接收程序。

第 14～20 行 接收进程设置监听套接口的接收缓冲区大小为 4096。在连接建立后,这个大小会传递给已连接套接口。进程接着 accept 连接,给 SIGURG 建立一个信号处理程序,并设置套接口的属主。主进程然后在无穷循环中调用 pause。

第 22～31 行 信号处理程序调用 recv 来读带外数据。

我们先启动接收者接着启动发送者,下面是发送者的输出。

```
solaris % tcpsend05 bsdi 5555
wrote 16384 bytes of normal data
wrote 1 byte of OOB data
wrote 1024 bytes of normal data
```

正如所预期的,所有的数据放入了发送者的套接口发送缓冲区,它接着终止。下面是接收者的输出。

```
bsdi % tcprecv05 5555
SIGURG received
recv error: Resource temporarily unavailable
```

由我们的 err_sys 函数输出的出错消息串相应于 EAGAIN,这与 BSD_OS 中的 EWOULDBLOCK 相同。发送者 TCP 向接收者 TCP 发送带外通知,由此给接收进程产生 SIGURG 信号。但是当指定 MSG_OOB 标志调用 recv 时,带外字节不能读。

```
 1 #include    "unp.h"
 2 int         listenfd, connfd;
 3 void        sig_urg(int);
 4 int
 5 main(int argc, char * * argv)
 6 {
 7     int       size;
 8     if (argc == 2)
 9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10     else if (argc == 3)
11         listenfd = Tcp_listen(argv[1], argv[2], NULL);
12     else
13         err_quit("usage: tcprecv05 [ <host> ] <port#>");
14     size = 4096;
15     Setsockopt(listenfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
```

```
16      connfd = Accept(listenfd, NULL, NULL);
17      Signal(SIGURG, sig_urg);
18      Fcntl(connfd, F_SETOWN, getpid());
19      for ( ; ; )
20          pause();
21  }
22  void
23  sig_urg(int signo)
24  {
25      int             n;
26      char    buff[2048];
27      printf("SIGURG received\n");
28      n = Recv(connfd, buff, sizeof(buff)-1, MSG_OOB);
29      buff[n] = 0;            /* null terminate */
30      printf("read %d OOB byte\n", n);
31  }
```

图 21.11　接收程序[oob/tcprecv05.c]

解决方法是让接收者通过读入现有的普通数据，在套接口接收缓冲区中腾出空间。这将会导致 TCP 向发送者通告一个非零的窗口，最终让发送者发送带外字节。

> 我们注意源自 Berkeley 的实现中的两件相关事情(TCPv2 第 1016～1017 页)。首先，即使套接口发送缓冲区满，内核也会从发送进程接受带外字节。其次，当发送进程发送一个带外字节时，一个含有紧急通知的 TCP 分节会被立刻送出。所有通常的 TCP 输出检查(Nagle 算法、无义窗口避免、等)都会略过。

例子

我们下一个例子展示每个给定的 TCP 连接只有一个带外标记，如果在接收进程读入现有带外数据之前一个新的带外数据到达，那么先前的标记丢失。

图 21.12 是发送程序，与图 21.8 相似，附加了另一个带外数据的 send 和接着的普通数据的 write。

```
1  #include    "unp.h"
2  int
3  main(int argc, char **argv)
4  {
5      int     sockfd;
6      if (argc != 3)
7          err_quit("usage: tcpsend04 <host> <port#>");
8      sockfd = Tcp_connect(argv[1], argv[2]);
9      Write(sockfd, "123", 3);
10     printf("wrote 3 bytes of normal data\n");
11     Send(sockfd, "4", 1, MSG_OOB);
12     printf("wrote 1 byte of OOB data\n");
13     Write(sockfd, "5", 1);
14     printf("wrote 1 byte of normal data\n");
15     Send(sockfd, "6", 1, MSG_OOB);
16     printf("wrote 1 byte of OOB data\n");
17     Write(sockfd, "7", 1);
```

```
18      printf("wrote 1 byte of normal data\n");
19      exit(0);
20  }
```

图 21.12 前后紧凑地发送两个带外字节[oob/tcpsend06.c]

在发送中没有停顿,允许所有的数据迅速发送给接收者 TCP。

接收程序与图 21.9 中相同,它在接受连接之后 sleep 5 秒,以允许数据到达它的 TCP。这里是接收程序的输出:

```
bsdi % tcprecv06 5555
read 5 bytes: 12345
at OOB mark
read 2 bytes: 67
received EOF
```

第二个带外字节(6)的到来覆盖了第一个带外字节(4)到来时存储的标记。正像我们说的,每个 TCP 连接只有一个带外标记。

21.4 TCP 带外数据小结

到现在为止,我们使用带外数据的例子都是小例子。不幸的是,当我们考虑可能出现的定时问题时,带外数据会变得繁杂。要考虑的第一点是带外数据的概念实际上传递给接收者三条不同的信息。

1. 发送者进入紧急模式的事实,接收进程可以用 SIGURG 信号或者 select 得到通知。这种通知在发送者发送带外数据后立即传输,因为在图 21.11 中我们看到即使发送数据给接收者因 TCP 的流控而停止了,TCP 仍发送这种通知。这种通知可能导致接收者进入某种特殊处理模式,以处理收到的后继数据。
2. 带外字节的位置,也就是说相对于发送者的其余数据,它是在哪儿发出的:带外标记。
3. 带外字节的实际值。由于 TCP 是一个字节流协议,它不解释应用进程发送的数据,因此这可以是任何 8 位值。

对于 TCP 的紧急模式,我们可以认为 URG 标志是通知,紧急指针是标记,数据字节是其本身。

和这种带外数据相关的问题是(a)每个连接只有一个 TCP 紧急指针,(b)每个连接只有一个带外标记,(c)每个连接只有一个字节的带外缓冲区(这个缓冲区只有在数据非在线读入时才是一个问题)。我们在图 21.12 中看到,新到来的标记覆盖进程没有处理的先前标记。如果是在线读数据,当新的带外数据到来时,先前的带外字节不会丢失,但是标记丢失了。

带外数据的一个常见用途是在 Rlogin 程序中。当客户中断运行于服务器上的程序时(TCPv1 第 393~394 页),服务器需要通知客户忽略所有已排队的服务器的输出,因为从服务器到客户可能有最多一个窗口大小的输出在等待发送给客户。服务器向客户发送一个特殊字节,告诉它去清空所有的输出,这个字节就作为带外数据发送。当客户收到了 SIGURG 信号时,它只是从套接口中读直到遇见标记,并忽略直到标记的所有数据。(TCPv1 第 398~

401页含有这样使用带外数据的一个例子以及相应的tcpdump输出。)在这种情形中,如果服务器想前后紧凑地发送多个带外字节,那也不会影响客户,因为客户一直读到最后一个标记,忽略所有数据。

总之,带外数据的用处取决于应用程序使用它的目的。如果目的是告诉对方忽略直到标记的普通数据,那么丢失一个中间的带外字节和相应的标记就不太重要。但是如果不丢失带外数据是重要的,那么数据必须在线接收。而且,作为带外数据发送的数据字节应与普通数据不同,因为中间的标记会被新收到的标记覆盖,将带外数据与普通数据混在了一起。例如,telnet在客户和服务器之间发送命令是用普通数据流,每个命令前加一字节255。(要发送这个值作为数据,就要两个字节的255。)这使它区分了它的命令和普通的用户数据,但是要求客户和服务器处理每个字节以寻找命令。

21.5 客户-服务器心博函数

我们现在开发一些简单的心博函数用于我们的回射客户和服务器程序。这些函数可以发现对方主机或到对方的通信路径的早期失效。

在给出这些函数前,我们必须提出一些警告。首先,一些人想使用TCP的保持存活特性(SO_KEEPALIVE套接口选项)来提供这种功能,但是TCP只有在连接空闲2小时后才会发送一个保持存活探测分节。人们认识到这一点以后,他们的问题是如何修改保持存活参数使其达到一个更低的值(通常是秒的量级)以更快地发现失效。虽然在许多系统上可以缩短TCP的保持存活定时器参数(见TCPv1的附录E),但这些参数通常是以每个内核为基维持的,而不是以每个套接口为基维持的,所以改变这些参数会影响所有打开这个选项的套接口。另外,保持存活选项不是为这个目的(高频率的轮询)设置的。

第二,两个端系统之间短暂的连接性丢失并不总是一件坏事情。TCP就是设计来处理这种情况的,源自Berkeley的TCP实现会在夭折一个连接前重传8~10分钟。新的IP路由协议(例如OSPF)能发现链接的失效,并且可能会在短时间内(例如在几秒的量级上)使用另外一条路径。因此你必须检查你的应用程序,确定在没有听到对方信息5~10秒后终止连接是好事还是坏事。有些应用程序需要这类功能,但大多数并不需要。

我们将使用TCP的紧急模式来有规律地轮询对方;我们在下面的描述中假设每秒轮询一次,5秒无回应则认为对方不再存活,但这是可由应用程序改变的。图21.13展示了客户和服务器的关系。

在这个例子中,客户每隔一秒向服务器发送一个带外字节,服务器收到后向客户发回一个带外字节。双方都需要知道对方是否不存在了或者不可达。客户和服务器每秒增加它们的变量cnt,每收到一个带外字节就将这个变量重置为0。如果计数到了5(也就是说进程5秒内没有收到对方的带外字节),就认为失败了。客户和服务器都用SIGURG信号接收带外字节到达的通知。我们在该图的中间指出:数据、回射数据和带外字节都是通过一个TCP连接交换的。

我们的客户程序main函数来自图5.4,没有改动。我们的str_cli函数(我们没有给出)与图6.13中的相比只有3处简单的改动。

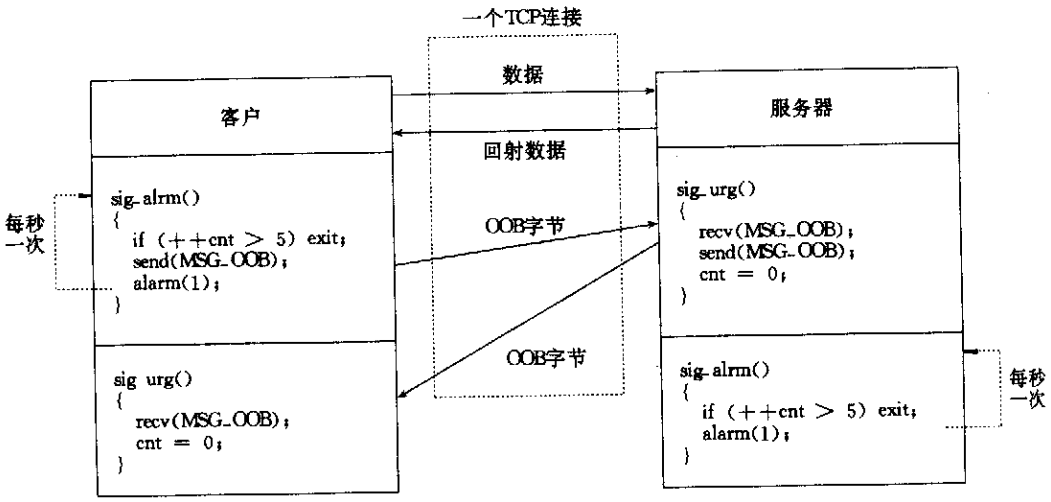


图 21.13 使用带外数据的客户-服务器心博函数

1. 在进入 for 循环前，调用 heartbeat_cli 函数来设置客户的心博特性：

```
heartbeat_cli(sockfd,1,5);
```

第二个参数是以秒为单位的轮询频率，第三个参数是放弃所在连接上的尝试前没有响应的秒数。

2. 如果 select 调用返回 EINTR 错误，我们 continue 到循环开始处，再次调用 select。注意，在图 21.13 中客户现在捕获两个信号：SIGALRM 和 SIGURG，所以我们必须准备处理被中断的系统调用。

3. 我们调用 writen 向标准输出写回射的行，而不是调用 fputs，原因是我们正在捕获两个信号，它们会中断慢的系统调用，而有些版本的标准 I/O 库不能正确地处理被中断的系统调用[Korn and Vo 1991]。

图 21.4 给出了提供客户心博功能的三个函数。

全程变量

第 2～5 行 前三个变量是给 heartbeat_cli 的参数的拷贝：套接口描述字(信号处理程序需用它来发送和接收带外数据)、SIGALRM 的频率、在客户认为服务器或连接死掉之前没有来自服务器的响应的 SIGALRM 的总数。变量 nprobes 记录从最近一次服务器应答以来的 SIGALRM 的数目。

```
1 #include    "unp.h"
2 static int     servfd;
3 static int     nsec;           /* #seconds betweeen each alarm */
4 static int     maxnprobes;     /* #probes w/no response before quit */
5 static int     nprobes;        /* #probes since last server response */
6 static void sig_urg(int), sig_alrm(int);
7 void
8 heartbeat_cli(int servfd_arg, int nsec_arg, int maxnprobes_arg)
9 {
10      servfd = servfd_arg;        /* set globals for signal handlers */
```

```
11      if ( (nsec = nsec_arg) < 1)
12          nsec = 1;
13      if ( (maxnprobes = maxnprobes_arg) < nsec)
14          maxnprobes = nsec;
15      nprobes = 0;
16      Signal(SIGURG, sig_urg);
17      Fcntl(servfd, F_SETOWN, getpid());
18      Signal(SIGALRM, sig_alrm);
19      alarm(nsec);
20  }
21  static void
22  sig_urg(int signo)
23  {
24      int     n;
25      char    c;
26      if ( (n = recv(servfd, &c, 1, MSG_OOB)) < 0) {
27          if (errno != EWOULDBLOCK)
28              err_sys("recv error");
29      }
30      nprobes = 0;            /* reset counter */
31      return;                 /* may interrupt client code */
32  }
33  static void
34  sig_alrm(int signo)
35  {
36      if (++nprobes > maxnprobes) {
37          fprintf(stderr, "server is unreachable\n");
38          exit(0);
39      }
40      Send(servfd, "1", 1, MSG_OOB);
41      alarm(nsec);
42      return;             /* may interrupt client code */
43  }
```

图 21.14　客户心博函数[oob/heartbeatcli.c]

heartbeat_cli 函数

第 7～20 行　heartbeat_cli 函数检查并且保存参数。给 SIGURG 和 SIGALRM 建立信号处理函数，将套接口的属主设为进程 ID。alarm 调度第一个 SIGALRM。

SIGURG 处理程序

第 21～32 行　当一个带外通知到来时，就会产生这个信号。我们试图去读带外字节，但如果还没有到(EWOULDBLOCK)也没有关系。由于系统不是在线接收带外数据，因此不会干扰客户读取它的普通数据。

既然服务器仍然存活，nprobes 就重置为 0。

SIGALRM 处理程序

第 33～43 行　这个信号以有规律间隔产生。计数器 nprobes 增 1，如果达到了 maxnprobes，我们认为服务器或者崩溃或者不可达。在这个例子中，我们结束客户进程，尽管其他的设计也可以使用：可以发送给主循环一个信号，或者作为另外一个参数给 heartbeat_cli 提供一个客户函数，当服务器看来死掉时调用它。

一个含有字符1的字节被作为带外数据发送(这个值没有什么含义)后,alarm调度下一个SIGALRM。

我们的服务器程序main函数与图5.12中的相同。与图5.3相比,对str_echo函数的仅有改动是在for循环前加入了下面一行:

```
heartbeat_serv(sockfd,1,5);
```

这个调用给服务器初始化心博函数。

图21.15给出了服务器心博函数。

heartbeat_serv函数

第7~19行 变量声明和函数heartbeat_serv几乎与客户的相同。

SIGURG处理程序

第20~32行 当一个带外通知收到时,服务器试图读入它。就像客户一样,如果带外字节没有到达没有什么关系。带外字节被作为带外数据返送给客户。注意,如果recv返回EWOULDBLOCK错误,那么自动变量c碰巧是什么就送给客户什么。由于我们不用带外字节的值,所以这没有关系。重要的是发送1字节的带外数据,而不管该字节是什么。由于刚收到通知,客户仍存活,所以重置nprobes为0。

SIGALRM处理程序

第33~42行 nprobes增1,如果它到达了调用者指定的值maxnalarms,服务器进程将被终止。否则调度下一个SIGALRM。

```
1 #include    "unp.h"
2 static int   servfd;
3 static int   nsec;           /* #seconds between each alarm */
4 static int   maxnalarms;     /* #alarms w/no client probe before quit */
5 static int   nprobes;        /* #alarms since last client probe */
6 static void sig_urg(int), sig_alrm(int);
7 void
8 heartbeat_serv(int servfd_arg, int nsec_arg, int maxnalarms_arg)
9 {
10    servfd = servfd_arg;   /* set globals for signal handlers */
11    if ( (nsec = nsec_arg) < 1)
12        nsec = 1;
13    if ( (maxnalarms = maxnalarms_arg) < nsec)
14        maxnalarms = nsec;
15    Signal(SIGURG, sig_urg);
16    Fcntl(servfd, F_SETOWN, getpid());
17    Signal(SIGALRM, sig_alrm);
18    alarm(nsec);
19 }
20 static void
21 sig_urg(int signo)
22 {
23    int         n;
24    char  c;
25    if ( (n = recv(servfd, &c, 1, MSG_OOB)) < 0) {
```

```
26          if (errno != EWOULDBLOCK)
27              err_sys("recv error");
28      }
29      Send(servfd, &c, 1, MSG_OOB); /* echo back out-of-band byte */
30      nprobes = 0;                  /* reset counter */
31      return;                       /* may interrupt server code */
32  }
33  static void
34  sig_alrm(int signo)
35  {
36      if (++nprobes > maxnalarms) {
37          printf("no probes from client\n");
38          exit(0);
39      }
40      alarm(nsec);
41      return;                       /* may interrupt server code */
42  }
```

图 21.15　服务器心博函数[oob/heartbeatserv.c]

21.6　小　结

TCP 没有真正的带外数据，一旦发送者进入紧急模式，它在 TCP 的头部中提供一个紧急指针发送给对方。连接另一端收到该指针即告诉接收进程发送者已进入了紧急模式，并且该指针指向紧急数据的最后一个字节。但是所有数据的发送仍然受 TCP 通常的流控支配。

套接口 API 把 TCP 的紧急模式映射为所谓的带外数据。通过在 send 调用中指定 MSG_OOB 标志，发送者进入紧急模式。在这个调用中最后的数据字节被认为是带外数据。接收者 TCP 接收到新的紧急指针后，通过发送 SIGURG 信号或者通过 select 指示套接口有异常条件待处理以通知接收者。TCP 缺省将带外字节从普通数据流中取出放入它自己的 1 字节带外缓冲区，由进程通过指定 MSG_OOB 标志调用 recv 读取。另外，接收者可以设置 SO_OOBINLINE 套接口选项，这种情况下，带外字节被留在普通的数据流中。不管接收者用的是哪种方法，套接口层在数据流中保存了一个带外标记，并且不会在一次输入操作中读过这个标记。接收者通过调用 sockatmark 函数确定它是否到达了标记。

带外数据没有广泛地使用。Telnet 和 Rlogin 使用它，还有 FTP，但是 FTP 使用它只是由于早期的实现没有提供 I/O 复用的特性。带外数据是在资源缺乏(也就是处理机内存和 CPU 时间)的时候设计的。当设计收发双方之间需要第二个高优先级、无流控信道的新应用程序时，我们应当考虑第二个 TCP 连接，而不是使用带外数据。

21.7　习　题

21.1　在一个函数调用

```
send(fd,"ab",2,MSG_OOB);
```

与两个函数调用

```
send(fd,"a",1,MSG_OOB);
```

```
send(fd,"b",1,MSG_OOB);
```

之间有什么区别?

21.2 重新编写21.6,使用poll而不是select。

21.3 修改图21.14和21.15中的sig_alrm函数,每次它被调用且nprobes已经大于0时向STDERR_FILENO写一条消息(例如,前一次探测丢失)。在同一个LAN的两台主机上上运行客户和服务器,看消息输出的频率。将客户的标准输入重定向到一个大文本文件,将它的标准输出重定向到一个临时文件。再运行客户,比较输入输出文件,确认没有数据丢失。当交换大量数据时,是否消息输出得更频繁?现在,在跨越WAN上两台主机上运行客户和服务器,并与LAN的结果进行比较。

21.4 用第二个TCP连接而不是紧急数据重写客户-服务器心博函数。运行与上个习题相同的测试,比较结果。

第 22 章　信号驱动 I/O

22.1　概　述

信号驱动是指当某个描述字上发生了某个事件时，让内核通知进程。这在历史上称为异步 I/O，但我们所描述的信号驱动不是真正的异步 I/O。后者通常定义为在进行 I/O 操作(读或写)的过程中，内核在启动 I/O 操作后立即返回，这样 I/O 操作的同时进程继续执行。当操作完成或遇到错误时以某种方式通知进程。在 6.2 节中我们比较了通常可用的各种 I/O 类型，并指出了信号驱动 I/O 和异步 I/O 的不同之处。

请注意，我们在第 15 章描述的非阻塞 I/O 同样不是异步 I/O。在非阻塞 I/O 中，启动 I/O 操作后内核并不像真正的异步 I/O 那样立即返回；它只有在进程非得睡眠才能完成操作时才立即返回。

> Posix.1 通过 aio_XXX 函数提供异步 I/O。这些函数让进程说明 I/O 完成时是否产生信号及产生什么信号。

源自 Berkeley 的实现用 SIGIO 信号支持套接口和终端设备的信号驱动器 I/O。SVR4 用 SIGPOLL 信号支持流设备上的信号驱动 I/O，而 SIGPOLL 等价于 SIGIO。

22.2　套接口上的信号驱动 I/O

使用套接口上的信号驱动 I/O(SIGIO)需要进程执行以下三个步骤：

1. 给 SIGIO 信号建立信号处理程序。
2. 设置套接口属主，通常使用 fcntl 的 F_SETOWN 命令(图 7.15)。
3. 激活套接口的信号驱动 I/O，通常使用 fcntl 的 F_SETFL 命令打开 O_ASYNC 标志(图 7.15)。

> O_ASYNC 标志是 posix.lg 新加的，图 1.16 中的系统没有一个支持该标志，所以在图 22.4 中我们用 FIOASYNC ioctl 代为激活信号驱动 I/O。注意，Posix.lg 选择了一个不太好的名字；O_SIGIO 也许是一个更好的选择。
>
> 应该在设置套接口属主之前建立信号处理程序。在源自 Berkeley 的实现中，相应的两个函数的调用顺序无关紧要，因为 SIGIO 的缺省行为是忽略该信号。因此如果我们颠倒这两个函数的调用顺序，就有可能在 fcntl 和 signal 两个调用之间产生一个信号，不过若如此该信号只是被丢弃。但是在 SVR4 中，SIGIO 在头文件<sys/signal.h>中被定义为 SIGPOLL，而 SLGPOLL 的缺省行为是终止进程。所以在 SVR4 中我们要确保在设置套接口属主之前安装信号处理程序。

虽然设置套接口上的信号驱动I/O很简单，但困难的是判断什么条件导致给套接口属主引发SIGIO信号，而这依赖于底层协议。

UDP套接口上的SIGIO信号

UDP上使用信号驱动I/O是简单的。当下述事件发生时产生SIGIO信号：

- 数据报到达套接口
- 套接口上发生异步错误

因此，当我们捕获到SIGIO信号时，我们调用recvfrom读取到达的数据报或者获取异步错误。我们在8.9节曾谈到UDP套接口的异步错误。我们知道只有当在UDP套接口已连接时才会产生这种错误。

> 当这两个条件满足时，内核调用sorwakeup产生SIGIO信号（见TCPv2第775、779页及784页）。

TCP套接口上的SIGIO信号

不幸的是，信号驱动I/O对TCP套接口几乎是没用的，原因是该信号产生得过于频繁，并且该信号的出现并没有告诉我们发生了什么事件。正如在TCPv2的第439页所指出的，下列条件均可在TCP套接口上产生SIGIO信号（假设信号驱动I/O是使能的）：

- 在监听套接口上有一个连接请求已经完成
- 发起了一个连接拆除请求
- 一个连接拆除请求已经完成
- 一个连接的一半已经关闭
- 数据到达了套接口
- 数据已从套接口上发出（即输出缓冲区有空闲空间）
- 发生了一个异步错误

*例如，如果一个进程既从一个TCP套接口读数据，又向其上写数据，当新数据到达或者以前所写数据得到确认后均会产生SIGIO信号，进程无法在信号处理程序中区分这两种情况。*如果在这种情况下使用SIGIO，TCP套接口应被设置为非阻塞方式以防止read或write发生阻塞。我们应该考虑只在监听TCP套接口上使用SIGIO，因为在监听套接口上产生SIGIO的唯一条件是一个新连接的完成。

作者能够找到的实际使用信号驱动I/O的程序是基于UDP的NTP（网络时间协议）服务器程序。NTP服务器的主循环从客户接收数据报并送回响应，但是每个客户请求都要进行相当数量的处理（比我们简单的回射服务器多得多）。对服务器来讲，很重要的一点是给每个收到的数据报记录精确的时间戳，因为这个值要返回给客户，客户要用它来计算到达该服务器的来回时间。图22.1展示了构建这样一个UDP服务器的两种方法。

大多数UDP服务器（包括第8章中的回射服务器）都被设计成了图中左边的方式。但是NTP服务器采用了图中右边的技术：当一个新数据报到达时，SIGIO处理程序读得该数据报，同时记录数据报到达的时刻，然后把它放入进程的另一个队列中，由主服务器循环移走和处理。虽然这种技巧使服务器代码变得复杂，但是它为到达的数据报提供了精确的时间戳。

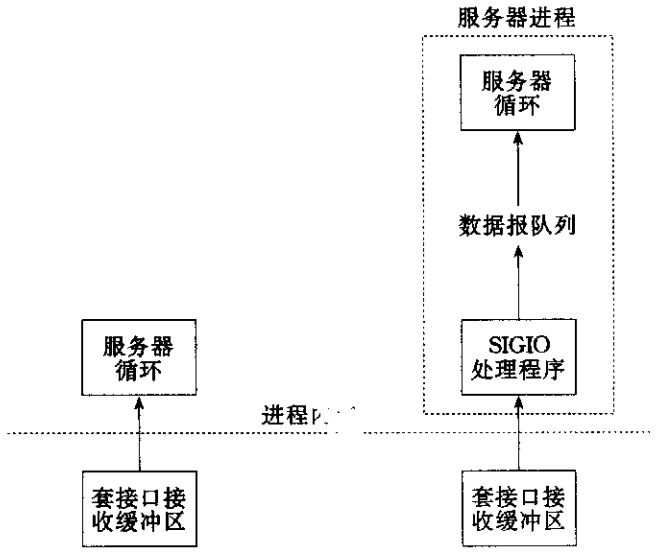


图 22.1　构建一个 UDP 服务器的两种方法

回想图 20.4，在那儿进程能够设置 IP_RECVDSTADDR 套接口选项以接收一个已收到的 UDP 数据报的目的地址。人们可能争论说同样应该接收该数据报的另外两条信息：收到的接口指示（如果采用普遍的弱端系统模型，那么收到的接口和目的地址就可能不同）和数据报到达的时刻。

> IPv6 的 IPV6_PKTINFO 套接口选项（20.8 节）可返回收到的接口。我们在 20.2 节中讨论了 IPv4 的 IP_RECVIF 套接口选项。
>
> FreeBSD 提供 SO_TIMESTAMP 套接口选项以作为辅助数据返回数据报收到的时刻（timeval 结构）。Linux 提供 SIOCGSTAMP ioctl 以返回一个记录数据报接收时刻的 timeval 结构。

22.3　使用 SIGIO 的 UDP 回射服务器程序

我们现在举一个类似图 22.1 右边的例子：一个使用 SIGIO 信号接收到达的数据报的 UDP 服务器程序。

我们使用图 8.7 和 8.8 中同样的客户程序以及图 8.3 中同样的服务器程序 main 函数。我们做的唯一修改是 dg_echo 函数，下边四张图给出了这些修改。图 22.2 给出了全局变量声明。

已收到数据报队列

第 3～12 行　SIGIO 信号处理程序将到达的数据报放入一个队列中。该队列是一个 DG 结构数组，我们将它处理成环形缓冲区。每个 DG 结构包括一个指向收到的数据报的指针、数据报的长度、一个指向包含客户协议地址的套接口地址结构的指针以及协议地址的大小。静态分配我们 QSIZE 个 DG 结构，从图 22.4 我们将看到 dg_echo 函数调用 malloc 给所有的数据报和套接口地址结构分配内存。我们还分配一个诊断用计数器 cntread，不久就会解

释到。图22.3展示了当第一项指向一个150字节数据报、与其关联的套接口地址结构长度为16时，DG结构数组的内容。

数组下标

第13～15行 iget是主循环将处理的下一个数组元素的下标，iput是信号处理程序将要存放的下一个数组元素的下标，nqueue是主循环将要处理的队列中数据报的总数目。

```
1 #include    "unp.h"
2 static int   sockfd;
3 #define     QSIZE     8          /* size of input queue */
4 #define     MAXDG 4096           /* maximum datagram size */
5 typedef struct {
6     void          *dg_data;      /* ptr to actual datagram */
7     size_t      dg_len;          /* length of datagram */
8     struct sockaddr   *dg_sa;    /* ptr to sockaddr{} w/client's address */
9     socklen_t dg_salen;          /* length of sockaddr{} */
10 } DG;
11 static DG    dg[QSIZE];        /* the queue of datagrams to process */
12 static long cntread[QSIZE+1];  /* diagnostic counter */
13 static int   iget;             /* next one for main loop to process */
14 static int   iput;             /* next one for signal handler to read into */
15 static int   nqueue;           /* #on queue for main loop to process */
16 static socklen_t clilen;       /* max length of sockaddr{} */
17 static void sig_io(int);
18 static void sig_hup(int);
```

图22.2 全局变量说明[sigio/dgecho01.c]

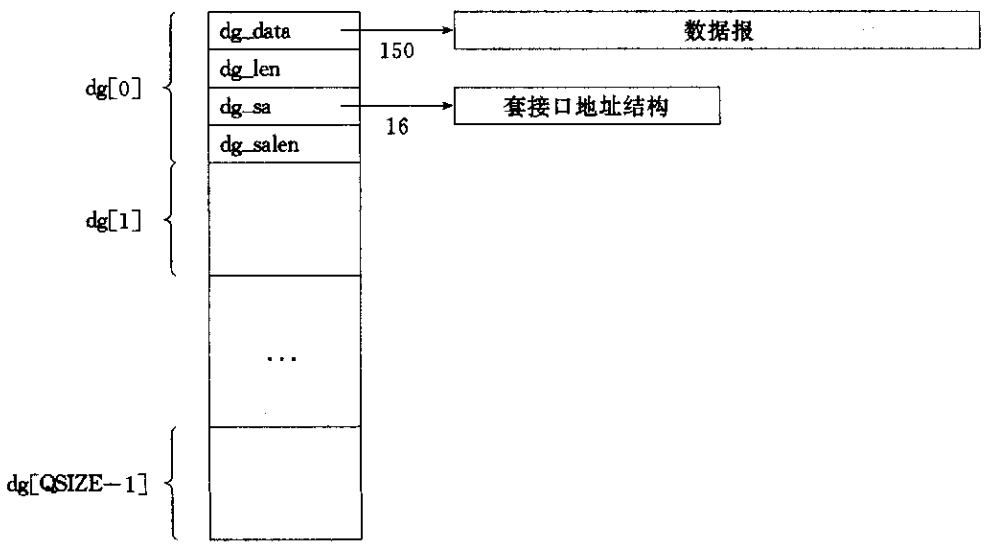


图22.3 用于存放接收到的数据报及其套接口地址结构的数据结构

```
19 void
20 dg_echo(int sockfd_arg, SA *pcliaddr, socklen_t clilen_arg)
21 {
22     int
```

```
23      const int   on = 1;
24      sigset_t    zeromask, newmask, oldmask;
25      sockfd = sockfd_arg;
26      clilen = clilen_arg;
27      for (i = 0; i < QSIZE; i++) { /* init queue of buffers */
28          dg[i].dg_data = Malloc(MAXDG);
29          dg[i].dg_sa = Malloc(clilen);
30          dg[i].dg_salen = clilen;
31      }
32      iget = iput = nqueue = 0;
33      Signal(SIGHUP, sig_hup);
34      Signal(SIGIO, sig_io);
35      Fcntl(sockfd, F_SETOWN, getpid());
36      Ioctl(sockfd, FIOASYNC, &on);
37      Ioctl(sockfd, FIONBIO, &on);
38      Sigemptyset(&zeromask);      /* init three signal sets */
39      Sigemptyset(&oldmask);
40      Sigemptyset(&newmask);
41      Sigaddset(&newmask, SIGIO);   /* the signal we want to block */
42      Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
43      for ( ; ; ) {
44          while (nqueue == 0)
45              sigsuspend(&zeromask);  /* wait for a datagram to process */
46              /* unblock SIGIO */
47          Sigprocmask(SIG_SETMASK, &oldmask, NULL);
48          Sendto(sockfd, dg[iget].dg_data, dg[iget].dg_len, 0,
49                 dg[iget].dg_sa, dg[iget].dg_salen);
50          if (++iget >= QSIZE)
51              iget = 0;
52              /* block SIGIO */
53          Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
54          nqueue--;
55      }
56  }
```

图 22.4　dg_echo 函数：服务器主处理循环[sigio/dgecho01.c]

初始化接收数据报队列

第 27～32 行　套接口描述字保存在一个全局变量中，因为信号处理程序要用到它。已收到数据报队列被初始化。

建立信号处理程序并设置套接口标志

第 33～37 行　给 SIGHUP(我们用作诊断)和 SIGIO 建立信号处理程序。用 fcntl 设置套接口属主，用 ioctl 设置信号驱动和非阻塞 I/O 标志。

> 我们在先前提到，fcntl 的 O_ASYNC 标志是 Posix.1g 指明使用信号驱动 I/O 的方法，但是因为大多数系统还不支持它，我们用 ioctl 来代替。尽管大多数系统确实支持用 fcntl 的 O_NONBLOCK 标志设置非阻塞 I/O，我们在这儿还是给出了 ioctl 方法。

初始化信号集

第38～41行 初始化三个信号集:zeromask(从不改变)、oldmask(记录我们阻塞SIGIO时的老信号掩码)和newmask。sigaddset打开newmask中与SIGIO对应的位。

阻塞SIGIO并且等待要做的事

第42～45行 sigprocmask将进程当前信号掩码存入oldmask中,然后将newmask与当前的信号掩码进行逻辑或。这将阻塞SIGIO并返回当前的信号掩码。然后我们进入for循环并测试nqueue计数器。只要这个计数器为0,进程就无事可做。这时我们调用sigsuspend。这个Posix函数自动保存当时的信号掩码并将当前信号掩码设置为其参数(zeromask)。因为zeromask是一个空信号集,所有的信号将被解阻塞。sigsuspend在捕获一个信号并在其信号处理程序返回后返回(sigsuspend是一个奇特的函数,它总是返回一个错误——EINTR)。但是在返回前,sigsuspend总是将信号掩码恢复为调用它时的信号掩码值,在我们的例子中,这个掩码值为newmask,所以我们能够保证sigsuspend返回后,SIGIO仍被阻塞。这就是为什么我们能够测试nqueue标志,因为当我们测试时,SIGIO信号不可能被递交。

> 试设想如果我们在测试nqueue这个主循环和信号处理程序共享的变量时,SIGIO未被阻塞会发生什么情况。我们可能测试nqueue时发现它为0,但刚刚测试完成,就有一个SIGIO递交并置nqueue为1。然后我们调用sigsuspend进入睡眠,这样实际上错过了了这个信号。除非另一个信号发生,否则我们将永远不能从sigsuspend调用中苏醒。这和我们在18.5节描述的竞争状态是相似的。

解除SIGIO阻塞,发送应答

第46～51行 我们调用sigprocmask将进程的信号掩码设置为先前保存在oldmask的旧值,从而解除了SIGIO阻塞。然后调用sendto发送应答。下标iget加1,如果其值等于数组元素的个数,则置iget为0,因为我们把数组当做环形缓冲区对待。注意:当修改iget时,我们不需要阻塞SIGIO,因为iget只被主循环使用,信号处理程序永远不会修改它。

阻塞SIGIO

第52～54行 阻塞SIGIO,nqueue减1。我们在修改nqueue时必须阻塞SIGIO,因为主循环和信号处理程序在共享这个变量。同样的理由,我们需要在循环顶部测试nqueue时阻塞SIGIO。

另外一个技巧是去掉for循环内的两个sigprocmask调用,避免解除信号阻塞后又阻塞它。然而,问题是这样的话,在整个循环执行的过程中,信号将被阻塞,这将降低信号处理程序的及时性。数据报不会因为这一改变而丢失(假设套接口接收缓冲区足够大),但是信号在整个阻塞期间将拖延递交给进程。在编写执行信号处理的应用程序时,目标之一就是尽量将信号阻塞时间减到最小。

图22.5给出的是SIGIO的信号处理程序。

```
57 static void
58 sig_io(int signo)
59 {
60     ssize_t          len;
```

```
61  int             nread;
62  DG              *ptr;
63  for (nread = 0; ; ) {
64      if (nqueue >= QSIZE)
65          err_quit("receive overflow");
66      ptr = &dg[iput];
67      ptr->dg_salen = clilen;
68      len = recvfrom(sockfd, ptr->dg_data, MAXDG, 0,
69                     ptr->dg_sa, &ptr->dg_salen);
70      if (len < 0) {
71          if (errno == EWOULDBLOCK)
72              break;          /* all done; no more queued to read */
73          else
74              err_sys("recvfrom error");
75      }
76      ptr->dg_len = len;
77      nread++;
78      nqueue++;
79      if (++iput >= QSIZE)
80          iput = 0;
81  }
82  cntread[nread]++;       /* histogram of #datagrams read per signal */
83 }
```

图 22.5　SIGIO 处理程序[sigio/dgecho01.c]

我们编写这段信号处理程序遇到的问题是 Posix 信号是不排队的。这意味着，如果我们在信号处理程序当中，由于信号保证在此期间阻塞，所以发生两次的信号将只被递交一次。

Posix.1 提供排队的实时信号，但其他信号如 SIGIO 通常是不排队的。

让我们考虑下述情形。一个数据报到达导致 SIGIO 被递交。它的信号处理程序读数据报并把它放入主循环的队列。但是当信号处理程序执行时，又有两个数据报到达，引起信号产生两次。然而因为信号被阻塞，当信号处理程序返回时，它将只被调用一次。信号处理程序第二次执行时，它读到的是第二个数据报，第三个数据报将留在套接口接收队列中。第三个数据报只有当第四个数据报到达时才能读到。当第四个数据报到达时，信号处理程序读入并放到主循环的队列中的是第三个而不是第四个数据报。

因为信号是不排队的，所以设置为信号驱动 I/O 的描述字也通常被设置为非阻塞方式。这样我们编写信号处理程序时用一个循环读数据报，直到读调用返回 EWOULDBLOCK 时才终止。

检查队列溢出

第 64～65 行　如果队列满，进程就终止。当然有其他方法(如分配额外缓冲区)处理这种情况。但在我们简单的例子中，我们终止执行。

读数据报

第 66～76 行　在非阻塞的套接口上调用 recvfrom。iput 做下标的数组项是数据报存储的地方。如果没有可读的数据报，则跳出 for 循环。

增加计数器和下标

第77～80行 nread是一个计数器，记录每个信号读的数据报数。nqueue是主循环将要处理的数据报数。

第82行 信号处理程序在返回前，将与每个信号读到的数据报数目对应的计数器加1。当SIGHUP递交后，该数组的内容做为诊断信息输出(见图22.6)。

```
84 static void
85 sig_hup(int signo)
86 {
87     int             i;
88     for (i = 0; i <= QSIZE; i++)
89         printf("cntread[%d] = %ld\n", i, cntread[i]);
90 }
```

图22.6 SIGHUP信号处理程序[sigio/dgecho01.c]

最后一个函数是SIGHUP信号处理程序(图22.6)，它输出cntread数组的内容。cntread数组统计每个信号读到的数据报数目。

为了说明信号是不排队的以及我们必须在设置信号驱动I/O标志之外，还要设置套接口的非阻塞方式，我们运行这个服务器并同时运行6个客户。每个客户发3645行数据让服务器回射，每个客户都从后台的shell脚本中启动，以便所有客户都能基本上同时启动。当所有客户都终止后，我们向服务器发SIGHUP信号，以使其输出cntread数组：

```
bsd: % udpserv01
cntread[0]=2
cntread[1]=21838
cntread[2]=12
cntread[3]=1
cntread[4]=0
cntread[5]=1
cntread[6]=0
cntread[7]=0
cntread[8]=0
```

信号处理程序大多数时间每次读到一个数据报，但有多次多于一个数据报就绪。cntread[0]非零表示，在信号处理程序运行的同时有信号产生，但在它返回之前，它读到了所有到达的数据报；当信号处理程序被再次调用时，已没有剩余可读的数据报。最后，我们证实数组元素的加权和(21838×1+12×2+1×3+1×5=21870)等于6(客户数)乘以3645行。

22.4 小 结

信号驱动I/O使内核在套接口上发生"某个事件"时用SIGIO信号通知进程：

- 对于已连接TCP套接口，有无数的条件可以引起这种通知，致使这个特性没有多少用处；
- 对于监听TCP套接口，一个连接已准备好接受时通知发生；
- 对于UDP，通知意味着或者一个数据报到达，或者一个异步错误到达，在这两种情况下，我们都调用recvfrom。

我们修改了我们的 UDP 回射服务器程序以使用信号驱动 I/O，使用类似于 NTP 使用的技巧：尽快读取到达的数据报，以得到到达时精确的时间戳，然后将它排队以备后续处理。

22.5　习　题

22.1　图 22.4 中的循环有下述另一种设计：

```
for( ; ; ) {
    Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
    while (nqueue == 0)
        sigsuspend(&zeromask);      /* wait for a datagram to process */
    nqueue--;
        /* unblock SIGIO */
    Sigprocmask(SIG_SETMASK,&oldmask,NULL);
        Sendto(sockfd,dg[iget],dg_data,dg[iget],dg_len,0,
        dg([iget].dg_sa,dg[iget].dg_salen);
    if (++iget >= QSIZE)
        iget = 0;
}
```

这个修改正确吗？

第23章 线 程

23.1 概 述

在传统的UNIX模型中,当一个进程需要由另一个实体执行某件事时,该进程派生(fork)一个子进程,让子进程去进行处理。Unix下的大多数网络服务器程序都是这么编写的,这在我们的并发服务器程序例子中可以看出:父进程接受连接,派生子进程,子进程处理与客户的交互。

虽然这种模式很多年来使用得很好,但是fork有一些问题:

- fork是昂贵的。内存映像要从父进程拷贝到子进程,所有描述字要在子进程中复制等等。目前的实现使用一种称做写时拷贝(copy-on-write)技术,可避免父进程数据空间向子进程的拷贝,除非子进程需要自己的拷贝。尽管有这种优化技术,fork仍然是昂贵的。
- fork子进程后,需要用进程间通信(IPC)在父子进程之间传递信息。fork之前的信息容易传递,因为子进程从一开始就有父进程数据空间及所有描述字的拷贝。但是从子进程返回信息给父进程需要做更多的工作。

线程有助于解决这两个问题。线程有时被称为轻权进程(lightweight process),因为线程比进程"权轻"。也就是说,创建线程要比创建进程快10～100倍。

一个进程中的所有线程共享相同的全局内存,这使得线程很容易共享信息,但是这种简易性也带来了同步(synchronization)问题。一个进程中的所有线程不仅共享全局变量,而且共享:

- 进程指令
- 大多数数据
- 打开的文件(如描述字)
- 信号处理程序和信号处置
- 当前工作目录
- 用户ID和组ID

但是每个线程有自己的:

- 线程ID
- 寄存器集合,包括程序计数器和栈指针
- 栈(用于存放局部变量和返回地址)
- errno
- 信号掩码
- 优先级

可把信号处理程序想像成一种线程，像我们在11.14中讨论的那样。在传统的UNIX模型中，我们有主执行流(一个线程)和信号处理程序(另一个线程)。如果当信号发生时主执行流正在更新一个链表，而且信号处理程序也试图更新链表，那么通常就会产生灾难性后果。主执行流和信号处理程序共享同样的全局变量，但它们各自有自己的栈。

在本章中，我们将讲述Posix线程，又称做Pthreads。这些内容在1995年作为Posix.1c标准的一部分得到了标准化，大多数UNIX版本将来都要支持它们。我们将会看到所有的pthread函数都以pthread_开头。本章只介绍线程，以便我们在网络程序中使用它们。如果想了解更多的有关线程的细节，请参看[Butenhof 1997]。

23.2 基本线程函数：创建和终止

本节讲述5个基本线程函数。在下两节中，我们将利用它们代替fork重新编写我们的TCP客户-服务器程序。

pthread_create函数

当一个程序由exec启动时，会创建一个称做初始线程(initial thread)或主线程(main thread)的单个线程。额外线程则由pthread_create函数创建。

```
#include <pthread.h>

int pthread_create (pthread_t *tid,const pthread_attr_t *attr,
                    void *(*func)(void *),void *arg);
```

返回：成功时为0，出错时为正Exxx值

一个进程中的每个线程都由一个线程ID(thread ID)标识，其数据类型是pthread_t(常常是unsigned int)。如果新的线程创建成功，其ID将通过tid指针返回。

每个线程都有很多属性(attribute)：优先级、起始栈大小、是否应该是一个守护线程，等等。当创建线程时，我们可通过初始化一个pthread_attr_t变量说明这些属性以覆盖缺省值。我们通常使用缺省值，在这种情况下，我们将attr参数说明为空指针。

最后，当创建一个线程时，我们要说明一个它将执行的函数。线程以调用该函数开始，然后或者显式地终止(调用pthread_exit)或者隐式地终止(让该函数返回)。函数的地址由func参数指定，该函数的调用参数是一个指针arg。如果我们需要多个调用参数，我们必须将它们打包成一个结构，然后将其地址当作唯一的参数传递给起始函数。

注意func和arg的声明。func函数取一个通用指针(void *)参数，并返回一个通用指针(void *)。这就使得我们可以传递一个指针(指向任何我们想要指向的东西)给线程，由线程返回一个指针(同样地，指向任何我们想要指向的东西)。

Pthread函数的返回值有两种：成功时返回为0，出错时返回为非0。它与套接口函数及大多数系统调用出错时返回-1，并置errno为正值不同，Pthread函数返回正值指示错误。例如，如果pthread_create因为超过了对系统线程数目的限制而不能创建新线程，将返回

EAGAIN。Pthread 函数不设置 errno。成功时返回 0，出错时返回非 0 的约定没有问题，因为所有的在<sys/errno.h>头文件中的 Exxx 值都大于 0。0 值永远不会赋给任何一个 Exxx 名字 。

pthread_join 函数

我们可以调用 pthread_join 等待一个线程终止。把线程和 UNIX 进程相比，pthread_create 类似于 fork，pthread_join 类似于 waitpid。

```
#include <pthread.h>
int pthread_join(pthread_t tid, void **status);
                                    返回：成功时为 0，出错时为正 Exxx 值
```

我们必须指定要等待线程的 tid。很可惜，我们没有办法等待任意一个线程结束(类似于 waitpid 的进程 ID 参数为－1 的情况)。我们在讨论图 23.14 时还将涉及这个问题。

如果，status 指针非空，线程的返回值(一个指向某个对象的指针)将存放在 status 指向的位置。

pthread_self 函数

每个线程都有一个 ID 以在给定的进程内标识自己。线程 ID 由 pthread_create 返回，我们也看到了它在 pthread_join 中的使用。线程用 pthread_self 取得自己的线程 ID。

```
#include <pthread.h>
pthread_t pthread_self(void);
                                    返回：调用线程的线程 ID
```

和 UNIX 进程相比，线程的 pthread_self 类似于 getpid。

pthread_detach 函数

线程或者是可汇合的(joinable)(缺省值)或者是脱离的(detached)。当可汇合的线程终止时，其线程 ID 和退出状态将保留，直到另外一个线程调用 pthread_join。脱离的线程则像守护进程：当它终止时，所有的资源都将释放，我们不能等待它终止。如果一个线程需要知道另一个线程什么时候终止，最好保留第二个线程的可汇合性。

pthread_detach 函数将指定的线程变为脱离的。

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
                                    返回：成功时为 0，出错时为正 Exxx 值
```

该函数通常被想脱离自己的线程调用，如：

```
pthread_detach (pthread_self());
```

pthread_exit 函数

终止线程的一种方法是调用 pthread_exit。

```
#include <pthread.h>
void pthread_exit(void *status);
                                                    不返回调用者
```

如果线程未脱离，其线程 ID 和退出状态将一直保留到调用进程中的某个其他线程调用 pthread_join。

指针 status 不能指向局部于调用线程的对象，因为线程终止时这些对象也消失。

有两种其他方法可使线程终止：

- 启动线程的函数(pthread_create 的第 3 个参数)返回。既然该函数必须说明为返回一个 void 指针，该返回值便是线程的终止状态。
- 如果进程的 main 函数返回或者任何线程调用了 exit，进程将终止，线程将随之终止。

23.3　使用线程的 str_cli 函数

我们使用线程的第一个例子是用线程代替 fork，重新编写图 15.9 中的 str_cli 函数。回想起来，我们提供了该函数的其他几个版本：最初使用停-等协议的图 5.5(我们展示该版本对批量输入而言远远不够优化)，使用阻塞 I/O 和 select 函数的图 6.13，以及从图 15.3 开始的使用非阻塞 I/O 的版本。图 23.1 示出我们线程版本的设计。

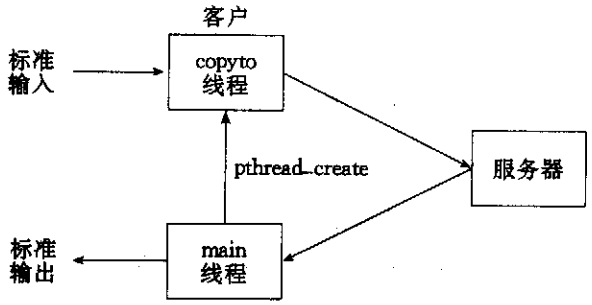


图 23.1　用线程重新编写 str_cli

图 23.2 给出了使用线程的 str_cli 函数代码。

unpthread.h 头文件

第 1 行　这是我们第 1 次碰到 unpthread.h 头文件。它包括我们通常的 unp.h，跟着是 Posix.1<pthread.h>头文件，然后定义我们为 pthread_XXX 函数编写的包裹版本(1.4 节)的函数原型，这些原型都以 Pthread_开头。

将参数存入外部变量中

第 10～11 行　我们将要创建的线程需要 str_cli 的两个参数：fp，输入文件的标准 I/O

FILE 指针;和 sockfd,连接到服务器的 TCP 套接口描述字。为简单起见,我们将这两个值存入外部变量中。另外一个技巧是将这两个值放入一个结构中,然后将指向该结构的指针做为参数传给我们将要创建的线程。

创建新线程

第 12 行 创建线程,新线程 ID 将存入 tid。新线程将要执行的函数是 copyto。没有参数传递给线程。

```
1 #include    "unpthread.h"
2 void    *copyto(void *);
3 static int   sockfd;          /* global for both threads to access */
4 static FILE *fp;
5 void
6 str_cli(FILE *fp_arg, int sockfd_arg)
7 {
8     char        recvline[MAXLINE];
9     pthread_t   tid;
10    sockfd = sockfd_arg;     /* copy arguments to externals */
11    fp = fp_arg;
12    Pthread_create(&tid, NULL, copyto, NULL);
13    while (Readline(sockfd, recvline, MAXLINE) > 0)
14        Fputs(recvline, stdout);
15 }
16 void *
17 copyto(void *arg)
18 {
19    char   sendline[MAXLINE];
20    while (Fgets(sendline, MAXLINE, fp) != NULL)
21        Writen(sockfd, sendline, strlen(sendline));
22    Shutdown(sockfd, SHUT_WR);     /* EOF on stdin, send FIN */
23    return(NULL);
24        /* return (i.e., thread terminates) when end-of-file on stdin */
25 }
```

图 23.2 使用线程的 str_cli 函数[threads/strclithread.c]

主线程循环:拷贝套接口输入到标准输出

第 13～14 行 主线程调用 readline 和 fputs,把从套接口读入的每一行拷贝到标准输出。

终止

第 15 行 当 str_cli 函数返回时,main 函数调用 exit 终止(5.4 节),从而进程中的所有线程都被终止。通常情况下,另外一个线程在读到标准输入上的文件结束符时已经终止。但是为了预防服务器过早终止(5.12 节),我们调用 exit 终止另一个线程,这正是我们想要的效果。

copyto 线程

第 16～25 行 这个线程只将从标准输入读入的每一行拷贝到套接口。当它在标准输入上读得文件结束符时,调用 shutdown 在套接口上发送 FIN,之后线程返回。从该函数(启动

线程的函数)返回终止了线程。

在第 15.2 节的最后,我们提供了 str_cli 函数使用的 5 种技术的测量性能。我们注意到刚刚给出的线程版本要用 8.5 秒,这比使用 fork 的版本稍稍快一点,但比非阻塞 I/O 要慢。然而,复杂的非阻塞 I/O 版本(15.2 节)和简单的线程版本相比,我们仍然推荐使用线程而不是非阻塞 I/O。

23.4 使用线程的 TCP 回射服务器程序

现在我们重新编写图 5.2 中的 TCP 回射服务器程序,我们采用一个客户对应一个线程而不是一个客户对应一个子进程。我们同样使用自己的 tcp_listen 函数使该程序与协议无关。图 23.3 给出的是服务器程序。

```
1 #include    "unpthread.h"
2 static void *doit(void *);     /* each thread executes this function */
3 int
4 main(int argc, char **argv)
5 {
6     int                listenfd, connfd;
7     socklen_t          addrlen, len;
8     struct sockaddr    *cliaddr;
9     pthread_t          tid;
10    if (argc == 2)
11        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12    else if (argc == 3)
13        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14    else
15        err_quit("usage: tcpserv01 [ <host> ] <service or port>");
16    cliaddr = Malloc(addrlen);
17    for ( ; ; ) {
18        len = addrlen;
19        connfd = Accept(listenfd, cliaddr, &len);
20        Pthread_create(&tid, NULL, &doit, (void *) connfd);
21    }
22 }
23 static void *
24 doit(void *arg)
25 {
26    Pthread_detach(pthread_self());
27    str_echo((int) arg);     /* same function as before */
28    Close((int) arg);        /* we are done with connected socket */
29    return(NULL);
30 }
```

图 23.3 使用线程的 TCP 回射服务器程序(参见习题 23.5)[threads/tcpserv01.c]

创建一个线程

第 17~21 行 当 accept 返回时,我们调用 pthread_create 而不是调用 fork。我们传给 doit 函数的唯一参数是已连接套接口描述字 connfd。

我们将整数描述字 connfd 强制成 void 指针。ANSI C 不保证该强制工作。只

有在整数大小小于或等于指针大小的系统上，该强制才工作。所幸的是大多数UNIX实现具有这个特性(图1.17)。我们不久还要谈到这一点。

线程函数

第23～30行 doit是线程执行的函数。线程脱离自己，因为主线程没有理由等待它创建的每个线程。str_echo函数和图5.3相同。该函数返回时，我们必须close已连接套接口，因为本线程和主线程共享所有的描述字。用fork时，子进程不必close已连接套接口，因为子进程终止后，其上所有打开的描述字都被关闭(参见习题23.2)。

另一值得注意的地方是主进程不关闭已连接套接口，而这是我们在调用fork的并发服务器程序中总是要做的。原因是一个进程中的所有线程共享描述字，如果主线程调用close，连接将被终止。创建线程并不影响打开的描述字的引用个数，这和fork是不同的。

该程序中有一个微小的错误，我们将在23.5节中详细描述。你能指出这个错误吗？(参见习题23.5)

给新线程传递参数

在图23.3中我们提到，我们将整数变量connfd强制成void指针并不能保证在所有系统上工作。要正确地处理这一点需要做额外的工作。

首先注意到，我们不能把connfd的地址传给新进程，也就是说，下列代码不工作：

```
int
main(int argc,char * * argv)
{
    int     listenfd, connfd;
    ...
    for ( ; ; ) {
        len = addrlen;
        connfd = Accept(listenfd,cliaddr,&len);
        Pthread_create(&tid,NULL,&doit,&connfd);
    }
}
static void *
doit(void * arg)
{
    int   connfd;
    connfd = * ((int * ) arg);
    Pthread_detach(pthread_self());
    str_echo(connfd);       /* same function as before */
    Close(connfd);          /* we are done with connected socket */
    return(NULL);
}
```

从ANSI C的角度看，这段代码没有问题：我们能够保证将整数指针强制为void *，然后再将该指针强制回到整型指针。问题是这个指针指向什么。

主线程中有一个整型变量connfd，每次调用accept时，新值(已连接描述字)都会覆盖该变量。因此可能发生下述情况：

- accept返回，新的已连接描述字(假设为5)存入connfd，主线程调用pthread_create，指向connfd的指针(不是connfd的内容)是pthread_create的最后一个参数。

- 创建一个线程，并调度 doit 函数以开始执行。
- 另一个连接准备好，主线程再次运行(在新创建线程之前)。accept 返回，新的描述字(假设为 6)存入 connfd，主线程调用 pthread_create。

即使在此情况下创建了两个线程，但是两个线程都将在已存入 connfd 的已连接描述字 6 上操作。问题的症结在于，多个线程同时存取一个共享变量(connfd 的整数值)而缺乏同步。在图23.3中，我们用给 pthread_create 传递 connfd 的值，而不是指向该值的指针来解决这个问题。C 将整数值传给被调用函数是很好的方法(即值的拷贝被推入被调函数的栈中)。

图 23.4 给出了解决这个问题的更好方法。

```
 1 #include    "unpthread.h"
 2 static void *doit(void *);     /* each thread executes this function */
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int    listenfd, *iptr;
 7     socklen_t   addrlen, len;
 8     struct sockaddr     *cliaddr;
 9     if (argc == 2)
10         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
11     else if (argc == 3)
12         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
13     else
14         err_quit("usage: tcpserv01 [ <host> ] <service or port>");
15     cliaddr = Malloc(addrlen);
16     for ( ; ; ) {
17         len = addrlen;
18         iptr = Malloc(sizeof(int));
19         *iptr = Accept(listenfd, cliaddr, &len);
20         Pthread_create(NULL, NULL, &doit, iptr);
21     }
22 }
23 static void *
24 doit(void *arg)
25 {
26     int         connfd;
27     connfd = *((int *) arg);
28     free(arg);
29     Pthread_detach(pthread_self());
30     str_echo(connfd);      /* same function as before */
31     Close(connfd);         /* we are done with connected socket */
32     return(NULL);
33 }
```

图 23.4 使用更具移植性参数传递的 TCP 回射服务程序的线程版本[threads/tcpserv02.c]

第 17～22 行 每当调用 accept 时，我们首先调用 malloc 给一个整型变量(已连接描述字)分配空间。这使得每个线程都有自己的已连接描述字拷贝。

第 28～29 行 线程获得已连接描述字的值，然后调用 free 释放内存空间。

malloc 和 free 函数历来是不可重入的。换句话说，从信号处理程序中调用这两个函数中的任何一个，同时主线程正在这两个函数中任何一个的处理过程中，那将会导致灾难性的后果，原因是这两个函数操纵相同的静态数据结构。那么我们如何在图 23.4 中调用这两个函数？Posix.1 要求这两个函数以及许多其他函数都是线程安全的(thread-safe)。这通常通过在库函数中进行某种形式的同步来达到，对我们是透明的。

线程安全函数

Posix.1 要求除了图 23.5 中列出的个别函数外，posix.1 和 ANS1 C 标准定义的所有函数都是线程安全的。

不必是线程安全的	必须是线程安全的	注释
asctime	asctime_r	
	ctermid	仅当参数非空时才是线程安全的
ctime	ctime_r	
getc_unlocked		
getchar_unblocked		
getgrid	getgrid_r	
getgrnam	getgrnam_r	
getlogin	getlogin_r	
getpwnam	getpwnam_r	
getpwuid	getpwuid_r	
gmtime	gmtime_r	
localtime	localtime_r	
putc_unlocked		
putchar_unlocked		
rand	rand_r	
readdir	readdir_r	
strtok	strtok_r	
	tmpnam	仅当参数非空时才是线程安全的
ttyname	ttyname_r	
gethostXXX		
getnetXXX		
getprotoXXX		
getservXXX		
inet_ntoa		

图 23.5 线程安全函数

可惜的是，关于网络 API 函数的线程安全性 Posix.1g 未做任何规定。表中最后 5 行摘自 Unix 98。我们在 11.4 节中讨论过 gethostbyname 和 gethostbyaddr 的不可重入特性。我们提到虽然一些厂商定义了以_r 结尾的线程安全版本，但是这些函数没有标准，因而应避免使用。所有不可重入的 getXXX 函数在图 9.9 中总结。

从该图我们看出，使一个函数线程安全的共同技巧是定义一个以_r 结尾的新函数。只有当一个函数的调用者给结果分配空间，并将其指针作为参数传给另外一个函数时，这两个函数才是线程安全的。

23.5 线程特定数据

当给第27章编写代码时，作者在将一个非线程应用程序转换成使用线程时犯了一个常见的编程错误。该错误只出现在运行图27.27中的服务器程序时，但是错误不在那个图中，而在被调用来处理每一个客户的readline函数中，readline也在图23.3中调用。

像其他与线程相关的编程错误一样，该错误导致的失败也是不确定的。实际上，图23.3和图27.27中的程序都能工作，但在给图27.29中线程版本程序运行定时测试时却发现了错误。不过当客户和服务器是在同一台主机上时，图27.29中的线程版本运行还是正确的，然而客户和服务器在不同主机上时，运行会在不同的时机失败，由web_child函数(图27.8)给出的错误是"client request for 0 bytes"(客户请求0个字节)。做了几小时的徒劳调试，作者突然想到在加快readline函数(图3.16)的过程中增加了静态变量(图3.17)。这个加速使该函数被单个进程中的不同线程调用时中断。

这是在将现成的函数转换成在线程环境中运行时常见的错误，并且有多种解决方法。

1. 使用线程特定数据。这种方法并不简单，它将函数转换成只能在支持线程的系统上工作。这种方法的优点是调用顺序不变，所有的改变都在库函数中而不在调用这些函数的应用程序中。本节后面将给定一个使用线程特定数据的readline版本，它是线程安全的。
2. 改变调用顺序使得调用者将所有参数以及图3.17中的静态变量都封装在一个结构里。这在以前也曾经做过，图23.6给定了新结构及新的函数原型。

```
typedef struct {
  int       read_fd;        /* caller's descriptor to read from */
  char      *read_ptr;      /* caller's buffer to read into */
  size_t    read_maxlen;    /* caller's max #bytes to read */
                /* next three are used internally by the function */
  int       rl_cnt;         /* initialize to 0 */
  char      *rl_bufptr;     /* initialize to rl_buf */
  char      rl_buf[MAXLINE];
} Rline;

void      readline_rinit(int, void *, size_t, Rline *);
ssize_t   readline_r(Rline *);
ssize_t   Readline_r(Rline *);
```

图23.6 readline可重入版本的数据结构及函数原型

这些新函数可在线程系统和大量非线程系统中运行，但调用readline的所有应用程序都要改变。

3. 忽略图3.17引入的加速代码，回到图3.16的老版本。

使用线程特定数据是使现成函数线程安全的常用技巧。在描述使用线程特定数据的Pthread函数之前，我们先描述概念和一个可能的实现，因为这些函数比它们实际看起来的要复杂得多。

> 作者看到的所有线程书籍都将线程特定数据描述得读起来像Pthreads标准，它们谈论键-值(key-value)对，键是透明对象，这增加了部分复杂性。我们以下

标和指针这两个术语刻划线程特定数据，因为在通常的实现中，键使用 Key 结构数组的小整数下标，以及键关联的值只是一个指向线程分配的存储区的指针。

每个系统支持有限数量的线程特定数据项。Posix.1 要求这个上限不小于 128(每个进程)，在后边的例子中我们假定这个上限存在。系统(很可能是线程库)为每个进程维护一个结构数组，我们称之为 Key 结构，如图 23.7 所示。

Key 结构中的标志指示这个数组元素是否正在使用，所有的标志初始化为“不在使用”。当一个线程调用 pthread_key_create 创建一个新线程特定数据项时，系统搜索其 Key 结构数组并找到第一个未被使用的项。其下标(0～127)称做键(key)，这个下标返回给调用线程。我们很快将谈到 Key 结构的另一个成员“析构函数指针”。

除了进程范围内的 Key 结构数组，系统还为进程的每个线程维护许多条信息。我们称之为 Pthread 结构，其中一部分是一个 128 个元素的指针数组，我们称之为 pkey 数组。图 23.8 展示了这些信息。

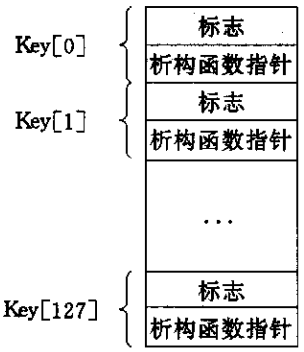


图 23.7 线程特定数据的可能实现

pkey 数组的所有项都被初始化为空指针。这 128 个指针是和进程的 128 个可能的键关联的值。

当我们用 pthread_key_create 创建一个键时，系统告诉我们它的键(下标)。每个线程随之能够给该键存储一个值(指针)，而每个线程通常又是通过 malloc 得到这个指针的。在线程特定数据中，部分易混淆的地方是：指针是键-值对中的值，但线程特定数据却是该指针指向的任何东西。

现在，我们完整地看一个如何使用线程特定数据的例子，这里假设我们的 readline 函数使用线程特定数据，跨越对该函数的连续调用维护每个线程的状态。我们很快将给出这个函数的代码，它是依循如下步骤修改原来的 readline 函数的结果。

1. 一个进程被启动，多个线程被创建。

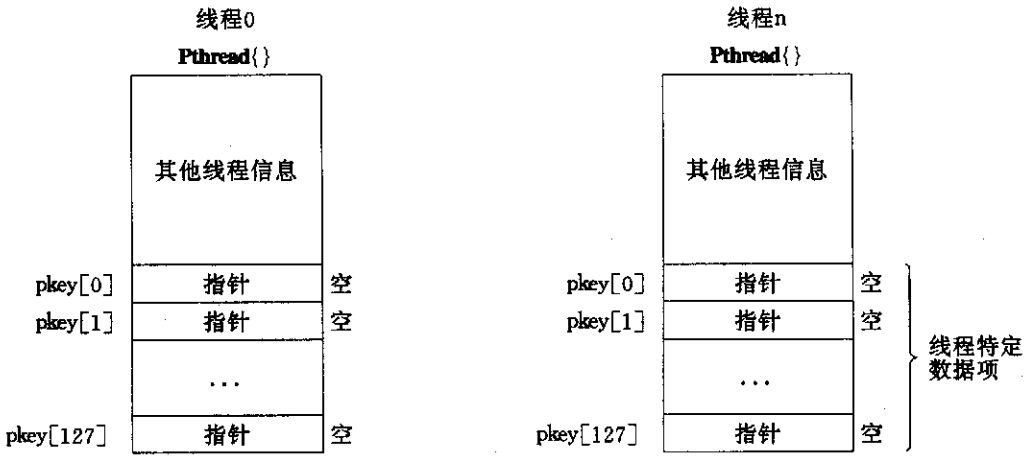


图 23.8 系统为每个线程维护的信息

2. 线程之一是第 1 个调用 readline 函数的线程，该函数再调用 pthread_key_create。系统在图23.7的 Key 结构数组中找到第 1 个未用的项并返回其下标(0～127)给调用者。假设找到的下标是 1。

   我们使用 pthread_once 函数来保证只有第一个调用 readline 的线程才调用 pthread_key_create。

3. readline 调用 pthread_getspecific 为该线程取得 pkey[1]的值(图 23.8 中与键 1 对应的“指针”)，返回值是一个空指针。于是 readline 调用 malloc 分配内存区域，该线程需要用它来跨越连续的 readline 调用保存本线程的线程特定的信息。readline 初始化该内存区域，并且调用 pthread_setspecific 将线程特定数据指针(pkey[1])指向它刚刚分配的内存区域。我们假设调用线程是进程中的线程 0。图 23.9 显示了这一过程。在该图中，我们注意到 Pthread 结构是系统(可能是线程库)维护的，但是我们 malloc 的线程特定数据是由我们的函数(本例中是 readline)维护的。pthread_setspecific 所做的全部工作是将这个 Pthread 结构中指定键指针指向我们刚刚分配的内存区域。类似地，pthread_getspecific 所做的全部工作是将这个指针返回给我们。

4. 另一个线程，假定是线程 n，调用 readline，这时线程 0 仍然在 readline 内执行。

   readline 调用 pthread_once 为这个线程特定数据项初始化键，但因为该函数已经被调用，所以不再被调用。

5. readline 调用 pthread_getspecific 为该线程取得 pkey[1]指针，返回空指针。该线程于是调用 malloc，像线程 0 一样为这个键(1)初始化线程特定数据。我们在图 23.10 中展示了这一点。

6. 线程 n 继续在 readline 中执行，使用和修改自己的线程特定数据。

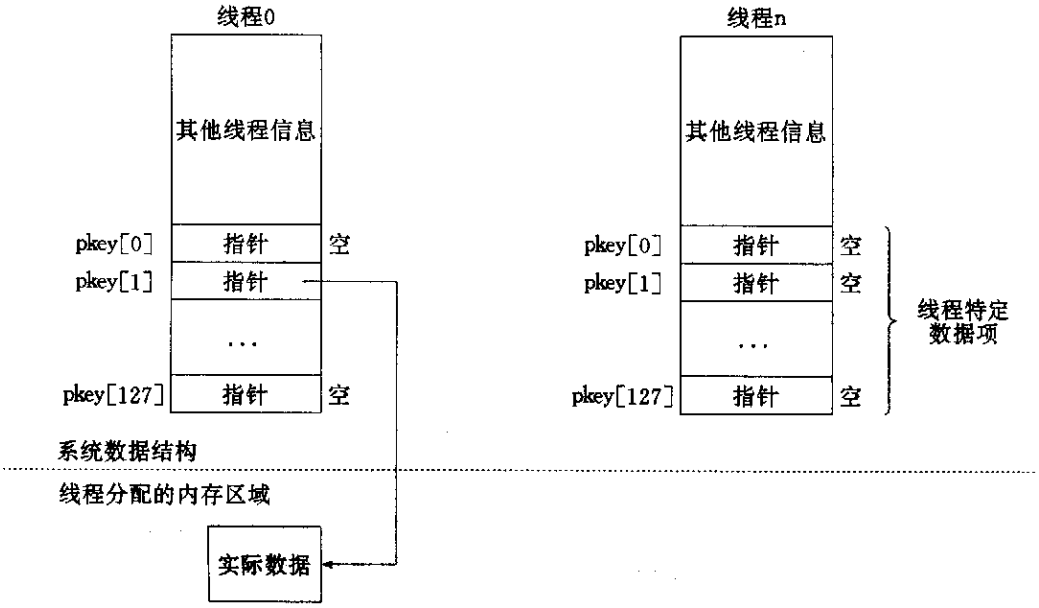


图 23.9 将分配的存储区域和线程特定数据指针关联

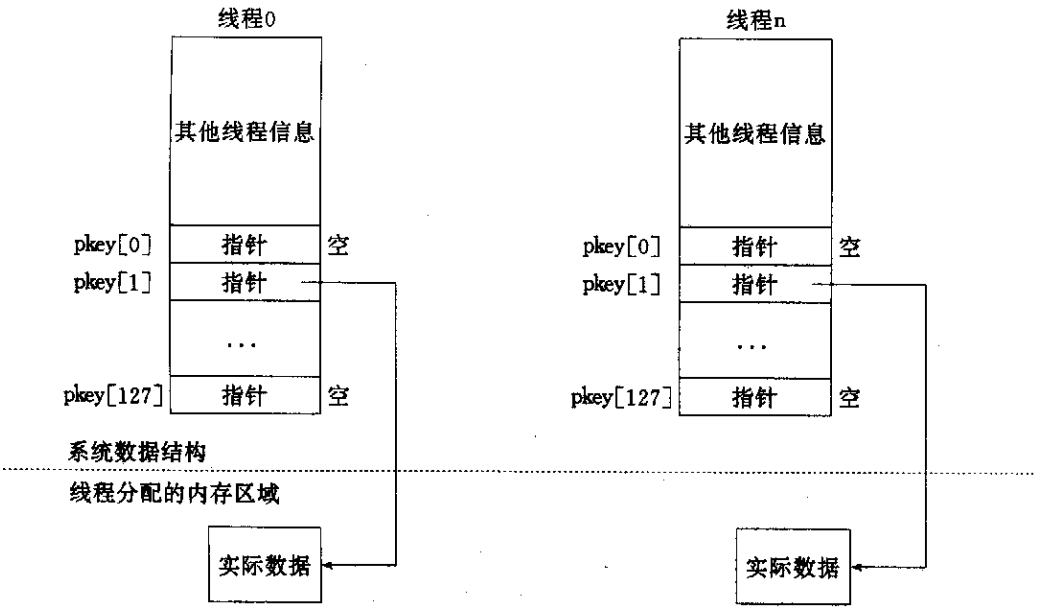


图 23.10 线程 n 初始化其线程特定数据后的数据结构

我们没有解决的一个问题是:当一个线程终止时会发生什么情况?如果该线程调用过我们的 readline 函数,该函数所分配的内存区就需要释放。这正是图 23.7 中"析构函数指针"的用武之地。线程调用 pthread_key_create 创建线程特定数据时,该函数的参数之一是指向一个析构函数(destructor)的指针。当一个线程终止时,系统扫描该线程的 pkey 数组,调用与每个非空 pkey 指针相应的析构函数,"相应的析构函数"是指在图 23.7 的 Key 数组中存放的函数指针。这是线程终止时线程特定数据的释放方法。

通常用来处理线程特定数据的头两个函数是:pthread_once 和 pthread_key_create。

```
#include <pthread.h>
int pthread_once(pthread_once_t * onceptr, void (*init)(void));
int pthread_key_create(pthread_key_t *keyptr,void (*destructor)(void *value));
                                        均返回:正常为 0,出错时为正 Exxx 值
```

通常每当一个使用线程特定数据的函数被调用时就要调用 pthread_once,但 pthread_once 使用 onceptr 所指的变量来保证每个进程只调用一次 init 函数。

对于一个进程内的给定键,pthread_key_create 只能被调用一次。键通过 keyptr 指针返回,如果相应 destructor 函数的参数不为空指针,而且线程为这个键存储了值,那么该线程终止时,destructor 函数将被调用。

这两个函数的用法如下所示(不考虑错误返回):

```
pthread_key_t    rl_key;
pthread_once_t   rl_once = PTHREAD_ONCE_INIT;

void
readline_destructor(void *ptr)
{
    free(ptr);
}

void
readline_once(void)
{
    pthread_key_create(&rl_key,readline_destructor);
}

ssize_t
readline( ... )
{
    ...
    pthread_once(&rl_once,readline_once);
    if ( (ptr = pthread_getspecific(rl_key)) == NULL) {
        ptr = Malloc( ... );
        pthread_setspecific(rl_key,ptr);
        /* initialize memory pointed to by ptr */
    }
    ...
    /* use the values pointed to by ptr */
}
```

每次 readline 被调用时,pthread_once 都要被调用。这个函数使用 onceptr 参数指向的值(变量 rl_once 的内容)确保 init 函数只被调用一次。初始化函数 readline_once 为存放在 rl_key 中的键创建线程特定数据,readline 的 pthread_getspecific 和 pthread_setspecific 将要用到该数据。

pthread_getspecific 和 pthread_setspecific 函数用来获取和存放与一个键关联的值。该值就是我们在图 23.8 中称做“指针”的东西。该指针指向什么由应用程序决定,但它通常指向动态分配内存区域。

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
                        返回:指向线程特定数据的指针(可能为非空指针)
int pthread_setspecific(pthread_key_t key, const void *value);
                        返回:成功时为0,出错时为正Exxx值
```

注意,phread_key_create 的参数是一个指向键的指针(因为该函数存放赋给键的值),而 get 和 set 函数的参数是键本身(很可能是我们前面讨论过的小整数下标)。

例子:使用线程特定数据的 readline 函数

现在我们举一个使用线程特定数据的例子。我们把图 3.17 中 readline 函数的优化版本转换为线程安全版本,而无需改变调用顺序。

图 23.11 是该函数的第 1 部分:pthread_key_t 变量、pthread_once_t 变量、readline_destructor 函数、readline_once 函数以及包含为每个线程维护的所有信息的 Rline 结构。

```
1  #include    "unpthread.h"
2  static pthread_key_t    rl_key;
3  static pthread_once_t   rl_once = PTHREAD_ONCE_INIT;
4  static void
5  readline_destructor(void *ptr)
6  {
7      free(ptr);
8  }
9  static void
10 readline_once(void)
11 {
12     Pthread_key_create(&rl_key, readline_destructor);
13 }
14 typedef struct {
15     int   rl_cnt;              /* initialize to 0 */
16     char        *rl_bufptr;   /* initialize to rl_buf */
17     char        rl_buf[MAXLINE];
18 } Rline;
```

图 23.11 线程安全的 readline 函数的第一部分[threads/readline.c]

析构函数

第 4～8 行 我们的析构函数只释放为该线程分配的内存区域。

一次性执行函数

第 9～13 行 我们将看到我们的一次性执行函数只被 pthread_once 调用一次,它只创建 readline 使用的键。

Rline 结构

第 14～18 行 Rline 结构包括三个变量，它们是图 3.17 声明为 static 从而引起问题的变量。Rline 结构之一将由每个线程动态分配，然后由析构函数释放。

图 23.12 给出了实际的 readline 函数及其调用的 my_read 函数。该图是图 3.17 的修改版。

```
19 static ssize_t
20 my_read(Rline *tsd, int fd, char *ptr)
21 {
22     if (tsd->rl_cnt <= 0) {
23     again:
24         if ( (tsd->rl_cnt = read(fd, tsd->rl_buf, MAXLINE)) < 0) {
25             if (errno == EINTR)
26                 goto again;
27             return(-1);
28         } else if (tsd->rl_cnt == 0)
29             return(0);
30         tsd->rl_bufptr = tsd->rl_buf;
31     }
32     tsd->rl_cnt--;
33     *ptr = *tsd->rl_bufptr++;
34     return(1);
35 }
36 ssize_t
37 readline(int fd, void *vptr, size_t maxlen)
38 {
39     int        n, rc;
40     char   c, *ptr;
41     Rline *tsd;
42     Pthread_once(&rl_once, readline_once);
43     if ( (tsd = pthread_getspecific(rl_key)) == NULL) {
44         tsd = Calloc(1, sizeof(Rline));          /* init to 0 */
45         Pthread_setspecific(rl_key, tsd);
46     }
47     ptr = vptr;
48     for (n = 1; n < maxlen; n++) {
49         if ( (rc = my_read(tsd, fd, &c)) == 1) {
50             *ptr++ = c;
51             if (c == '\n')
52                 break;
53         } else if (rc == 0) {
54             if (n == 1)
55                 return(0);      /* EOF, no data read */
56             else
57                 break;          /* EOF, some data was read */
58         } else
59             return(-1);         /* error, errno set by read() */
60     }
61     *ptr = 0;
62     return(n);
63 }
```

图 23.12 线程安全的 readline 函数的第二部分[threads/readline.c]

my_read 函数

第19～35行 该函数的第一个参数是指向 Rline 结构的指针，该指针是为这个线程分配的(实际的线程特定数据)。

分配线程特定数据

第42行 我们首先调用 pthread_once，使得进程中第一个调用 readline 的线程调用 pthread_once 创建线程特定数据键。

获取线程特定数据指针

第43～46行 pthread_getspecific 为线程返回指向 Rline 结构的指针。但是如果这是线程第一次调用 readline，其返回值将是一个空指针。在这种情况下，我们分配一个 Rline 结构的空间，并且初始化其 rl_cnt 成员为0。然后我们调用 pthread_setspecific 为线程存储这个指针。下一次线程调用 readline 时，pthread_getspecific 将返回这个刚存储的指针。

23.6 Web 客户与同时连接

现在让我们再来看一看15.5节中 Web 客户程序的例子。我们要用线程代替非阻塞 connect 重新编码。使用线程后，套接口就可以停留在缺省的阻塞方式，但必须为每个连接创建一个线程。每个线程阻塞在 connect 调用上也没有关系，因为内核会运行一些就绪的其他线程。

图23.13给出的是该程序的第一部分，即全局变量和 main 函数的开头部分。

```
 1 #include    "unpthread.h"
 2 #include    <thread.h>          /* Solaris threads */
 3 #define     MAXFILES    20
 4 #define     SERV        "80"    /* port number or service name */
 5 struct file {
 6     char *f_name;               /* filename */
 7     char *f_host;               /* hostname or IP address */
 8     int   f_fd;                 /* descriptor */
 9     int   f_flags;              /* F_xxx below */
10     pthread_t f_tid;            /* thread ID */
11 } file[MAXFILES];
12 #define F_CONNECTING    1       /* connect() in progress */
13 #define F_READING       2       /* connect() complete; now reading */
14 #define F_DONE          4       /* all done */
15 #define GET_CMD     "GET %s HTTP/1.0\r\n\r\n"
16 int     nconn, nfiles, nlefttoconn, nlefttoread;
17 void   *do_get_read(void *);
18 void    home_page(const char *, const char *);
19 void    write_get_cmd(struct file *);
20 int
21 main(int argc, char **argv)
22 {
23     int             i, n, maxnconn;
24     pthread_t       tid;
25     struct file    *fptr;
```

```
26    if (argc < 5)
27        err_quit("usage: web <#conns> <IPaddr> <homepage> file1 ...");
28    maxnconn = atoi(argv[1]);
29    nfiles = min(argc - 4, MAXFILES);
30    for (i = 0; i < nfiles; i++) {
31        file[i].f_name = argv[i + 4];
32        file[i].f_host = argv[2];
33        file[i].f_flags = 0;
34    }
35    printf("nfiles = %d\n", nfiles);
36    home_page(argv[2], argv[3]);
37    nlefttoread = nlefttoconn = nfiles;
38    nconn = 0;
```

图 23.13　全局变量和 main 函数的开头部分[threads/web01.c]

全局变量

第 1～16　我们在包括通常的<pthread.h>之外还包括了<thread.h>，因为除了用到 Pthreads，我们还需使用 Solaris 线程。我们不久还要描述这一点。

第 10 行　我们在 file 结构中增加了一个成员 f_tid，即线程 ID。其余代码和图 15.15 相似。在本线程版中，我们不使用 select，所以不需要任何描述字集或变量 maxfd。

第 36 行　被调用的 home_page 函数与图 15.16 相比没有改变。

图 23.14 给出的是 main 线程的主处理循环。

```
39    while (nlefttoread > 0) {
40        while (nconn < maxnconn && nlefttoconn > 0) {
41                /* find a file to read */
42            for (i = 0 ; i < nfiles; i++)
43                if (file[i].f_flags == 0)
44                    break;
45            if (i == nfiles)
46                err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
47            file[i].f_flags = F_CONNECTING;
48            Pthread_create(&tid, NULL, &do_get_read, &file[i]);
49            file[i].f_tid = tid;
50            nconn++;
51            nlefttoconn--;
52        }
53        if ( (n = thr_join(0, &tid, (void **) &fptr)) != 0)
54            errno = n, err_sys("thr_join error");
55        nconn--;
56        nlefttoread--;
57        printf("thread id %d for %s done\n", tid, fptr->f_name);
58    }
59    exit(0);
60 }
```

图 23.14　main 函数的主处理循环[threads/web01.c]

如果可能，创建另一个线程

第 40～52 行　如果允许我们创建另一个线程(nconn 小于 maxnconn)，我们就创建一个线程。新线程执行的函数是 do_get_read，函数参数是指向 file 结构的指针。

等待任何一个线程终止

第53～54行 调用Solaris的线程函数thr_join，第一个参数为0表示我们等待任何一个线程终止。不幸的是Pthreads没有提供等待任一线程终止的方法；pthread_join函数只能让我们显式地说明希望等待的线程。我们将在23.9节中看到，Pthreads解决这一问题的方法要复杂得多：它要求我们使用一个条件变量，线程终止时可以通过它通知主线程。

> 我们给出的使用Solaris线程函数thr_join的方案并不能在所有环境下可移植。即使可以，我们还是想举这个使用线程的Web客户程序的例子，而不必因为讨论条件变量和互斥锁而增加复杂性。幸运的是我们能够在Solaris中混用Pthreads和Solaris线程。
>
> 作者曾经在Usenet上抱怨pthread_join不能等待任何一个线程中止，一些曾经为Pthread标准工作过的人为这种设计决策的合理性辩解，他们宣称pthread_join不能为每个人去做每件事情。他们还宣称，在进程模型中有父-子进程关系，所以wait或waitpid等待任一子程序终止具有意义。但是在线程环境中，没有和父子进程类似的层次关系，所以等待任一线程终止并没有意义。其状态由等待任一线程的函数返回的线程不一定是调用线程创建的。如果有人真的需要等待任一线程，可以用条件变量实现(并不简单)，我们很快将给出这个方法。尽管有这些争论，作者还是认为这是pthread_join的一个设计缺陷。

图23.15给出的是每个线程都执行的do_get_read函数。该函数建立TCP连接，发送HTTP GET命令给服务器，并从服务器读到应答。

```
61 void *
62 do_get_read(void *vptr)
63 {
64     int             fd, n;
65     char            line[MAXLINE];
66     struct file     *fptr;
67     fptr = (struct file *) vptr;
68     fd = Tcp_connect(fptr->f_host, SERV);
69     fptr->f_fd = fd;
70     printf("do_get_read for %s, fd %d, thread %d\n",
71             fptr->f_name, fd, fptr->f_tid);
72     write_get_cmd(fptr);      /* write() the GET command */
73         /* Read server's reply */
74     for ( ; ; ) {
75         if ( (n = Read(fd, line, MAXLINE)) == 0)
76             break;            /* server closed connection */
77         printf("read %d bytes from %s\n", n, fptr->f_name);
78     }
79     printf("end-of-file on %s\n", fptr->f_name);
80     Close(fd);
81     fptr->f_flags = F_DONE;      /* clears F_READING */
82     return(fptr);           /* terminate thread */
83 }
```

图23.15 do_get_read函数[threads/web01.c]

创建 TCP 套接口，建立连接

第 68～71 行 由 tcp_connect 函数创建一个 TCP 套接口，建立一个连接。套接口是通常的阻塞方式套接口，所以线程在调用 connect 时会阻塞，直到连接建立。

写请求给服务器

第 72 行 write_get_cmd 构造 HTTP GET 命令并将它发送给服务器。我们不再给出这个函数的代码，它和图 15.18 的唯一区别是线程版本不调用 FD_SET，不使用 maxfd。

读取服务器的应答

第 73～82 行 服务器的应答随即被读取。当连接被服务器关闭时，设置 F_DONE 标志，函数返回，线程终止。

我们同样没有给出 home_page 函数，因为它和图 15.16 给出的版本完全一样。

我们还将回到这个例子，届时我们会用移植性更好的 Pthreads 方案替换 Solaris thr_join 函数。但首先让我们讨论一下互斥锁和条件变量。

23.7 互 斥 锁

注意，在图 23.14 中，当一个线程结束时，主循环将 nconn 和 nlefttoread 分别减 1。我们可以将这两个减 1 操作放到 do_get_read 函数中，以使每个线程在终止前的瞬间将这两个计数器减 1。但这是一个微妙而意义重大的并发程序设计错误。

将减 1 计数器的代码放到每个线程均执行的函数中的问题是：这两个计数器是全局变量，而不是线程特有的。如果一个线程在减 1 变量的当中被挂起，另一个线程执行并减 1 同一个变量便会引起错误。例如，假设 C 编译器将减 1 操作符用 3 条机器指令来实现：从内存装载到寄存器、寄存器减 1 及从寄存器存入内存。考虑下述可能的情形：

1. 线程 A 运行，它将 nconn 的值(3)装入一个寄存器。
2. 系统将线程从 A 切换到 B。A′的寄存器被存储，B′的寄存器被恢复。
3. 线程 B 执行与 C 表达式 nconn--相对应的三条指令存储新值 2；
4. 一段时间之后，系统将线程从 B 切回到 A。A′的寄存器被恢复，A 从原来离开的地方继续执行，即 3 条机器指令序列的第 2 条：寄存器中的值被减 1，即从 3 变为 2，并且 2 被存入 nconn。

最终的结果是 nconn 为 2，而不是正确的 1。这是错误的。

这类并行程序设计错误很难被发现，原因有多个：首先，错误很少出现。然而这是一个错误，早晚会引起失败(墨菲定律 Murphy's Law)；其次，错误很难重复，因为它依赖于许多非确定的事件时序。最后，在有些系统上，硬件指令可能是原子的，亦即存在一个硬件指令给内存中的整数减 1(而不是我们前面假设的 3 条指令序列)，而且这个指令在执行过程中不能被中断。但是不能保证所有系统都如此，所以代码在一个系统上工作，而在另一个系统上不工作。

我们称线程编程为并发编程(concurrent programming)或并行编程(parallel programming)，因为多个线程可并发(并行)运行并访问相同的变量。虽然我们刚刚讨论的错误情形以单 CPU 系统为前提，但是如果线程 A 和线程 B 在多处理器系统的不同 CPU 上同时运

行,潜在的错误同样存在。在通常的 Unix 编程中,我们没有遇到这种并发编程问题,因为用 fork 时,除了描述字外,父进程和子进程不共享任何东西。但是,当我们讨论进程间的共享内存时仍将遇到这类问题。

我们很容易用线程表现这个问题。图 23.17 是一个简单的程序,它创建两个线程,然后让每个线程执行 5000 次将一个全局变量加 1 的操作。

为了增加问题出现的可能性,我们先取得 counter 的值,输出新值,然后存储新值。如果我们运行这个程序,可得到图 23.16 所示的输出。

```
4: 1
4: 2
4: 3
4: 4
                线程 4 如此继续执行
4: 517
4: 518
4: 518          线程 5 现在执行
4: 519
4: 520
                线程 5 如此继续执行
5: 926
5: 927
4: 519          线程 4 现在执行,所存入值是错的
4: 520
```

图 23.16 图 23.17 中程序的输出[threads/example01.c]

```
 1 #include    "unpthread.h"
 2 #define     NLOOP 5000
 3 int         counter;        /* this is incremented by the threads */
 4 void    *doit(void *);
 5 int
 6 main(int argc, char **argv)
 7 {
 8     pthread_t   tidA, tidB;
 9     Pthread_create(&tidA, NULL, &doit, NULL);
10     Pthread_create(&tidB, NULL, &doit, NULL);
11         /* wait for both threads to terminate */
12     Pthread_join(tidA, NULL);
13     Pthread_join(tidB, NULL);
14     exit(0);
15 }
16 void *
17 doit(void *vptr)
18 {
19     int         i, val;
20     /*
21      * Each thread fetches, prints, and increments the counter NLOOP times.
22      * The value of the counter should increase monotonically.
23      */
24     for (i = 0; i < NLOOP; i++) {
25         val = counter;
```

```
26          printf("%d: %d\n", pthread_self(), val + 1);
27          counter = val + 1;
28      }
29      return(NULL);
30  }
```

图 23.17 两个线程不正确地对全局变量进行加 1 操作[threads/example01.c]

注意,第一次错误发生在系统从线程 4 切换到线程 5 的时候:值 518 被每个线程存储。这种错误在 10,000 行输出中发生了许多次。

这种类型问题的不确定性在我们运行几次该程序后同样明显:每次运行的最终结果都和前一次运行的不同。如果我们将输出重定向到磁盘文件,有时就不出现错误,这是因为程序运行得快,在线程间切换的机会少。当我们交互式地运行程序,将输入写到(慢)终端上,同时用 Unix script 程序(在 APUE 的第 19 章中详细讨论)将输出保存到一个文件中时,错误发生的次数最多。

我们刚刚讨论的问题,即多个线程修改一个共享变量,是最简单的问题。解决方法是用一个互斥锁(mutex,代表 mutual exclusion)保护共享变量;只有我们持有该互斥锁才能访问该变量。在 Pthreads 中,互斥锁是类型为 pthread_mutex_t 的变量。我们用下面两个函数为互斥锁加锁和解锁。

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_unlock(pthread_mutex_t *mptr);
                                    返回:正确时为 0,出错时为正 Exxx 值
```

如果我们试图为一个已被其他线程锁住的互斥锁加锁,程序便会阻塞直到该互斥锁被解锁。

如果互斥锁变量是静态分配的,我们必须将它初始化为常值 PTHREAD_MUTEX_INITIALIZER。我们将在 27.8 节中看到,如果在共享内存中分配一个互斥锁。我们必须在运行时调用 pthread_mutex_init 函数进行初始化。

> 有些系统(如 Solaris)将 PTHREAD_MUTEX_INITIALIZER 定义为 0,所以忽略这个初始化没有问题,因为静态分配的变量自动初始化为 0。但是这么做并不能保证绝对没有问题,更何况其他系统(如 Digital Unix)将初始化常值定义为非 0。

图 23.18 是图 23.17 的修正版本,使用一个互斥锁在两个线程之间锁住计数器。

```
1 #include    "unpthread.h"
2 #define NLOOP 5000
3 int     counter;            /* this is incremented by the threads */
4 pthread_mutex_t counter_mutex=PTHREAD_MUTEX_INITIALIZER;
5 void   *doit(void *);
6 int
7 main(int argc, char **argv)
8 {
9     pthread_t   tidA, tidB;
```

```
10      Pthread_create(&tidA, NULL, &doit, NULL);
11      Pthread_create(&tidB, NULL, &doit, NULL);
12          /* wait for both threads to terminate */
13      Pthread_join(tidA, NULL);
14      Pthread_join(tidB, NULL);
15      exit(0);
16  }
17  void *
18  doit(void *vptr)
19  {
20      int         i, val;
21      /*
22       * Each thread fetches, prints, and increments the counter NLOOP times.
23       * The value of the counter should increase monotonically.
24       */
25      for (i = 0; i < NLOOP; i++) {
26      Pthread_mutex_lock(&counter_mutex);
27          val = counter;
28          printf("%d: %d\n", pthread_self(), val + 1);
29          counter = val + 1;
30      Pthread_mutex_unlock(&counter_mutex);
31      }
32      return(NULL);
33  }
```

图 23.18 使用互斥锁保护共享变量的图 23.17 的修正版本[threads/example02.c]

我们声明一个名为 counter_mutex 的互斥锁，线程在操作 counter 变量之前必须锁住该互斥锁。当我们运行这个程序时，输出总是正确的：值单调增加，输出的最终值总是 10,000。

用互斥锁加锁的开销有多大？将图 23.17 和 23.18 的程序改变成循环 50,000 次，在输出定向到/dev/null 的条件下测量时间。没有互斥的不正确版本和使用互斥锁的正确版本之间的差别是 10%。这告诉我们，互斥锁加锁并没有太大开销。

23.8 条件变量

互斥锁适于阻止对共享变量的同时访问，但是我们需要某种东西以使我们能够睡眠等待某种条件出现。让我们用一个例子说明这一点。我们回到 23.6 节中 Web 客户程序的例子，用 pthread_join 代替 Solaris 的 thr_join。但是我们在知道哪个线程终止之前无法调用这个 Pthread 函数。我们首先说明一个统计终止线程数目的全局变量，并且用一个互斥锁来保护。

```
int             ndone     /* number of terminated threads */
pthread_mutex_t ndone_mutex=PTHREAD_MUTEX_INITIALIZER;
```

我们要求每个线程终止时将该计数器加 1，同时小心使用与之关联的互斥锁。

```
void *
do_get_read(void *vptr)
{
    ...
```

```
    Pthread_mutex_lock(&ndone_mutex);
    ndone++;
    Pthread_mutex_unlock(&ndone_mutex);
    return(fptr);          /* terminate thread */
}
```

这么做一点没有问题，但我们怎样编写主循环？它需要持续地锁住该互斥锁并检查是否有任何线程终止。

```
while (nlefttoread > 0) {
    while (nconn < maxnconn && nlefttoconn > 0) {
            /* find a file to read */
        ...
    }
        /* See if one of the threads is done */
    Pthread_mutex_lock(&ndone_mutex);
    if (ndone > 0) {
        for (i = 0; i < nfiles; i++) {
            if (file[i].f_flags & F_DONE) {
                Pthread_join(file[i].f_tid,(void **) &fptr);
                /* update file[i] for terminated thread */
                ...
            }
        }
    }
    Pthread_mutex_unlock(&ndone_mutex);
}
```

虽然这种方法正确，但是它意味着主循环永远不进入睡眠，而要进行循环，每次循环都要检查 ndone。这称做轮询(polling)，会严重浪费 CPU 时间。

我们需要一种方法使得主循环进入睡眠，直到有一个线程通知它某件事已就绪。条件变量(condition variable)加上互斥锁可以提供这种功能。互斥锁提供互斥机制，条件变量提供信号机制。

在 Pthreads 中，条件变量是一个 pthread_cond_t 类型的变量。条件变量使用下述两个函数：

```
#include <pthread.h>
int pthread_cond_wait (pthread_cond_t *cptr,pthread_mutext_t *mptr);
int pthread_cond_signal(pthread_cond_t *cptr);
                                        返回：成功时为 0，出错时为正的 Exxx 值
```

第二个函数的名字中“signal”一词不是指 Unix 的 SIGxxx 信号。

举例是解释这些函数的最容易的方法。回到我们的 Web 客户程序例子，现在我们给计数器 ndone 同时关联一个条件变量和一个互斥锁：

```
int             ndone;
pthread_mutex_t ndone_mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  ndone_cond=PTHREAD_COND_INITIALIZER;
```

线程在持有互斥锁的情况下通过增 1 计数器并且用条件变量发信号来通知主循环：

```
Pthread_mutex_lock(&ndone_mutex);
ndone++;
Pthread_cond_signal(&ndone_cond);
Pthread_mutex_unlock(&ndone_mutex);
```

主循环阻塞在调用 pthread_cond_wait 上，以等待即将终止线程发送条件变量信号：

```
while(nlefttoread > 0) {
    while (nconn < maxnconn && nlefttoconn > 0) {
            /* find a file to read */
        ...
    }
        /* Wait for one of the threads to terminate */
    Pthread_mutex_lock(&ndone_mutex);
    while(ndone == 0)
        Pthread_cond_wait(&ndone_cond,&ndone_mutex);
    for (i = 0; i < nfiles; i++) {
        if(file[i].f_flags & F_DONE) {
            Pthread_join(file[i].f_tid,(void **) &fptr);
            /* update file[i] for terminated thread */
            ...
        }
    }
    Pthread_mutex_unlock(&ndone_mutex);
}
```

注意，变量 ndone 仍然在持有互斥锁的情况下检查。然后，如果无事可做，pthread_cond_wait 将被调用。这使调用线程进入睡眠并且释放持有的互斥锁。当线程从 pthread_cond_wait 返回时(在其他线程给它发送条件变量信号之后)，它又重新持有该互斥锁。

为什么每个条件变量都要关联一个互斥锁？“条件”通常是线程间共享的某个变量的值，不同线程设置和测试该变量时要求由一个互斥锁控制。如果我们不用互斥锁，刚刚给出的例子代码中的主循环便会这样测试 ndone：

```
    /* Wait for one of the threads to terminate */
while (ndone == 0)
    Pthread_cond_wait(&ndone_cond,&ndone_mutex);
```

这样就存在一种可能，在主循环测试 ndone==0 之后、调用 pthread_cond_wait 之前，最后一个线程给 ndone 加 1，这样的话最后这个“信号”将丢失，因而主循环将永远阻塞在 pthread_cond_wait 上，等待永远不再发生的事件。

同样的原因，pthread_cond_wait 必须在关联的互斥锁加锁的情况下调用，而且这个函数将解锁互斥锁和使调用线程进入睡眠作为一个单一的原子操作。要是该函数不解锁互斥锁，在返回时再给它加锁，线程将不得不自己为互斥锁解锁和加锁，程序代码将变为：

```
    /* Wait for one of the threads to terminate */
Pthread_mutex_lock(&ndone_mutex);
while (ndone == 0) {
    Pthread_mutex_unlock(&ndone_mutex);
    Pthread_cond_wait(&ndone_cond,&ndone_mutex);
    Pthread_mutex_lock(&ndone_mutext);
}
```

但是,同样存在最后一个线程在调用 pthread_mutex_unlock 和 pthread_cond_wait 之间终止,并增 1 ndone 值的可能性。

通常地,pthread_cond_signal 唤醒一个等待条件变量的线程。有时,线程知道多个线程应被唤醒,在这种情况下,pthread_cond_broadcast 将唤醒阻塞在该条件变量上的所有线程。

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cptr);
int pthread_cond_timedwait (pthread_cond_t *cptr,pthread_mutex_t *mptr,const struct
                            timespec *abstime);
                                两者均返回:成功时为 0,出错时为正的 Exxx 值
```

pthread_cond_timedwait 允许一个线程设置阻塞时间的上限。abstime 是一个 timespec 结构(在 6.9 节中随 pselect 函数定义),指定函数必须返回的系统时间,即使条件变量信号还没有被发出。如果超时发生,则返回 ETIME 错误。

这个时间值是绝对时间,而不是时间差。换句话说,abstime 是函数应该返回时的系统时间——从 1970 年 1 月 1 日 UTC 时间以来的秒数和纳秒数——这与 select 和 pselect 都不同,它们指定从现在到未来函数应返回时的秒数和微秒数(pselect 为纳秒数)。常用的过程是:调用 gettimeofday 取得当时的时间(以 timeval 结构),将其拷贝到一个 timespec 结构,将希望的时限加到该结构中。例如:

```
struct timeval   tv;
struct timespec  ts;
if (gettimeofday(&tv,NULL) < 0)
   err_sys("gettimeofday error");
ts.tv_sec  = tv.tv_sec + 5;      /* 5 seconds in future */
ts.tv_nsec = tv,tv_usec * 1000;/* microsec to nanosec */
pthread_cond_timedwait( ... , &ts);
```

使用绝对时间而非相对时间的优点是,如果函数提前返回(很可能因为捕获了一个信号),还可以在不改变 timespec 结构内容的情况下再次调用该函数。缺点是在首次调用函数之前不得不调用 gettimeofday。

Posix.1 定义了一个新函数 clock_gettime,返回的是 timespec 结构的当前时间。

23.9 Web 客户与并发连接(续)

我们现在重新编写图 23.6 中的 Web 客户程序。我们去掉 Solaris 的 thr_join 函数,用 pthread_join 调用代替。在该节中我们讨论过,我们必须精确地指定要等待的线程。为了做到这一点,我们将使用 23.8 节中描述的条件变量。

全局变量(图 23.13)的唯一变化是增加了一个新标志和条件变量:

```
#define  F_JOINED        8   /* main has pthread_join'ed */
int                  ndone;    /* number of terminated threads */
```

```
pthread_mutex_t     ndone_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t      ndone_cond  = PTHREAD_COND_INITIALIZER;
```

do_get_read 函数(图 23.15)的唯一变化是增 1 ndone,在线程终止之前通知主循环:

```
    printf("end-of-file on %s\n",fptr->f_name);
    Close(fd);
    Pthread_mutex_lock(&ndone_mutex);
    fptr->f_flags = F_DONE;          /* clears F_READING */
    ndone++;
    Pthread_cond_signal(&ndone_cond);
    Pthread_mutex_unlock(&ndone_mutex);
    return(fptr);          /* terminate thread */
}
```

大多数变化是在主循环中(图 23.14),图 23.19 给出的是主循环的新版本。

```
43    while (nlefttoread > 0) {
44        while (nconn < maxnconn && nlefttoconn > 0) {
45                /* find a file to read */
46            for (i = 0 ; i < nfiles; i++)
47                if (file[i].f_flags == 0)
48                    break;
49            if (i == nfiles)
50                err_quit("nlefttoconn = %d but nothing found", nlefttoconn);
51            file[i].f_flags = F_CONNECTING;
52            Pthread_create(&tid, NULL, &do_get_read, &file[i]);
53            file[i].f_tid = tid;
54            nconn++;
55            nlefttoconn--;
56        }
57            /* Wait for one of the threads to terminate */
58        Pthread_mutex_lock(&ndone_mutex);
59        while (ndone == 0)
60            Pthread_cond_wait(&ndone_cond, &ndone_mutex);
61        for (i = 0; i < nfiles; i++) {
62            if (file[i].f_flags & F_DONE) {
63                Pthread_join(file[i].f_tid, (void **) &fptr);
64                if (&file[i] != fptr)
65                    err_quit("file[i] != fptr");
66                fptr->f_flags = F_JOINED;      /* clears F_DONE */
67                ndone--;
68                nconn--;
69                nlefttoread--;
70                printf("thread %d for %s done\n", fptr->f_tid, fptr->f_name);
71            }
72        }
73        Pthread_mutex_unlock(&ndone_mutex);
74    }
72    exit(0);
76 }
```

图 23.19 main 函数的主处理循环[threads/web03.c]

如果可能，创建另一个线程

第44～56行 这几行代码没有改变。

等待线程终止

第57～60行 为了等待一个线程终止，我们等待ndone变为非0。像23.8节讨论的那样，测试必须在互斥锁上锁情况下进行，睡眠由pthread_cond_wait执行。

处理终止的线程

第61～73行 当一个线程终止时，我们扫描所有的file结构以找出相应的线程，调用pthread_join，然后设置新的F_JOINED标志。

图15.20给出了该版本的时间性能和使用非阻塞connect版本的时间性能。

23.10 小 结

创建一个新线程通常比用fork创建一个新进程要快得多。仅这一点就能在繁重使用的网络服务器中成为优点。但是线程编程是一个新的模式，需要更多的训练。

一个进程中的所有线程共享全局变量和描述字，从而允许这些信息被不同的线程共享。但是这种共享引入了同步问题，我们必须使用的Pthread同步原语是互斥锁和条件变量。共享数据的同步是几乎每个线程应用程序必需的部分。

编写能够被线程应用程序调用的函数时，这些函数必须是线程安全的。线程特定数据是一个可以帮助我们解决这个问题的技巧。我们用readline函数给出了一个例子。

我们将在第27章中回到线程模式，届时，我们将用到另一种服务器程序设计：服务器在启动时创建一个线程池，该池中一个可用的线程将处理下一个客户请求。

23.11 习 题

23.1 假设同时有100个客户需要得到服务，比较用fork的服务器和用线程的服务器中描述字的用量。

23.2 在图23.3中，如果str_echo返回时不关闭已连接套接口会发生什么情况？

23.3 在图5.5和6.13中，当我们期望从服务器收到一个回射行，但却收到文件结束符时（回忆5.12节），我们输出“server terminated prematurely（服务器过早终止）”。修改图23.2以在合适的时候也输出这条消息。

23.4 修改图23.11和23.12以便它们能在不支持线程的系统上通过编译。

23.5 为了观察图23.4中使用的readline函数的错误，请构造该程序并启动服务器。然后从图6.13中构造能够以批处理方式正确工作的TCP回射客户程序。从系统中找到一个大的文本文件并在批模式下三次启动客户，让它们从这个大文本文件中读并且将输出写入临时文件。如果可能，在与服务器主机不同的机器上运行客户。如果三个客户正确终止（它们常常挂起），观察它们临时输出文件的内容，并和输入文件进行比较。

现在构造一个使用23.5节中正确的readline函数的服务器程序版本。用三个客户重新运行测试，现在这三个客户应该工作。你还应该在readline_destructor函数、readline_once函数以及malloc的调用中各加入一个printf。这可以证实键只被创建一次，但是每个线程都要分配内存和调用析构函数。

第 24 章　IP 选项

24.1　概　述

IPv4 允许在固定的 20 字节头部之后跟以 40 字节的选项。虽然定义了 10 个不同的选项，但最常用的是源路径选项。这些选项是通过 IP_OPTIONS 套接口选项访问的，我们将用一个使用源路由的例子展示该访问方法。

IPv6 允许在固定的 40 字节 IPv6 头部和传输层头部（如 ICMPv6、TCP 或 UDP）之间出现扩展头部（extension header）。目前定义了 6 种不同的扩展头部。和 IPv4 不同的是，访问 IPv6 扩展头部的途径不是强迫用户去理解头部如何出现在 IPv6 分组中的实际细节，而是通过函数接口进行。

24.2　IPv4 选项

在图 A.1 中，我们展示了 20 字节 IPv4 头部后面的选项。在那儿，我们注意到，4 位的头部长度字段将总的 IPv4 头部的长度限制为 15 个 32 位字（60 字节），所以 IP 选项长度的上限为 40 字节。IPv4 定义了 10 种不同的选项：

1. NOP（no-operation）：1 字节，典型用法是为后续的选项做填充以使其对准 4 字节的边界。
2. EOL（end-of-list）：1 字节，终止选项处理。因为 IP 选项总的长度必须为 4 字节的倍数，所以 EOL 字节跟在最后一个选项之后。
3. LSRR（loose source and record route）（TCPv1 的 8.5 节）：我们将很快举例说明。
4. SSRR（strict source and record route）（TCPv1 的 8.5 节）：我们将很快举例说明。
5. Timestamp（TCPv1 的 7.4 节）。
6. Record route（TCPv1 的 7.3 节）。
7. Basic security。
8. Extended security。
9. Stream identifier（已过时）。
10. Router alert：这是一个新选项，在 RFC 2113 中描述[Katz 1997]。该选项包含在转发数据报的所有路由器都应该查看的 IP 数据报中。

TCPv2 的第 9 章提供了内核对前 6 个选项进行处理的进一步的细节，所指出的 TCPv1 的相关章节给出了如何使用它们的例子。RFC 1108[Kent 1991]有关于两个不太常用的安全选项的细节。

getsockopt 和 setsockopt（level 参数为 IPPROTO_IP，optname 参数为 IP_OPTIONS）读取和设置 IP 选项。getsockopt 和 setsockopt 的第四个参数是指向一个缓冲区（大小小于等于

44字节)的指针,第五个参数是该缓冲区的大小。该缓冲区的大小之所以可以比选项的最大总长度还大4个字节是源路经选项的处理方法使然,我们将很快描述到这一点。除了两个源路径选项之外,该缓冲区的格式就是IP数据报中放置的选项的格式。

当用setsockopt设置了IP选项后,在该套接口上发出的所有IP数据报都将包括这些选项。套接口可以是TCP、UDP或原始IP套接口。为了清除这些选项,可以调用setsockopt,置第四个参数为空指针,或者置第五个参数(长度)为0。

> 如果一个原始IP套接口上设置了IP_HDRINCL选项,给该套接口设置IP选项的操作就并不一定能在所有的实现上工作(我们将在下一章描述IP_HDRINCL)。许多源自Berkeley的实现在IP_HDRINCL选项打开时不发送用IP_OPTIONS设置的选项,因为应用进程可以在它构造的IP头部中设置自己的IP选项(TCPv2第1056~1057行)。别的系统(如FreeBSD)允许应用进程或者用IP_OPTIONS套接口选项设置IP选项,或者设置IP_HDRINCL,再将IP选项包括在自己构造的IP头部中,但不能两者都用。

当调用getsockopt获取一个由accept创建的已连接TCP套接口的IP选项时,返回的是在监听套接口上收到的客户SYN中源路径选项的倒序值(TCPv2第931页)。源路径自动地被TCP倒序,因为客户说明的是从客户到服务器的源路径,服务器需要使用该路径的倒序值,以发送从服务器到客户的数据报。如果SYN中没有源路径,那么getsockopt通过第五个参数返回的值-结果参数长度将为0。对所有其他的TCP套接口、所有的UDP套接口以及原始IP套接口而言,getsockopt只返回用setsockopt设置的IP选项的拷贝。注意,对一个原始IP套接口来说,收到的IP头部,包括任何IP选项,总是由输入函数返回的,所以收到的IP选项总是可得的。

> 源自Berkeley的内核从来都不为UDP套接口返回源路径选项或其他任何选项。TCPv2第775页所示的返回IP选项的的代码从BSD4.3 Reno以来已经存在,但一直被注释掉,因为它不能工作。这使得一个UDP应用进程不可能使用收到的路径的倒序值将数据报发回发送者。
>
> 许多源自Berkeley的内核在为原始IP套接口调用getsockopt和setsockopt时发生恐慌(panic),即系统停机。但是只有用超级用户权限才能创建原始IP套接口,而拥有超级用户权限的人本来就可以对系统进行更为恶意的活动。

24.3 IP源路径选项

源路径是IP数据报的发送者说明的一组IP地址。如果源路径是严格的(strict),那么数据报必须且只能经过所列的节点。如果源路径是宽松的(loose),数据报必须经过所列的节点但也可以经过未在源路径中列出的节点。

> IPv4的源路由是有争议的。[Cheswick and Bellovin 1994,第26页]倡议在所有的路由器上去掉这个特性,许多组织和业务供应商这么做了。对源路由的一种合法使用是用Traceroute程序检测非对称的路径,就像TCPv1第108~109页展示的那样。但是因特网上越来越多的路由器不支持源路由,所以即使这种

用法也将消失。但是，指定和接收源路径是套接口 API 的一部分，我们仍有必要对此进行描述。

IPv4 源路径称为源和记录路径(source and record routes，SRR)，宽松源路径选项称为 LSRR，严格源路径选项称为 SSRR，原因是当数据报穿过所列的全部节点到来时，每个节点都将自己的所列地址替换为自己的外出接口地址。这允许接收者将新的列表倒序以沿逆向的路径回到发送者。这两个源路径的例子以及相应的 tcpdump 输出在 TCPv1 的 8.5 节。

我们将源路径说明为一个 IPv4 地址数组，前边附以三个 1 字节字段(如图 24.1 所示)。这是我们传递给 setsockopt 的缓冲区的格式。

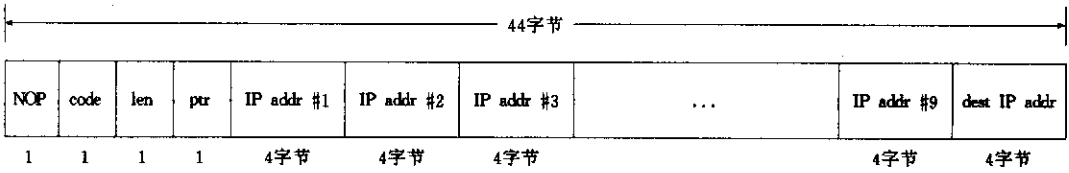


图 24.1 传递源路径给内核

在源路径选项之前，我们放置一个 NOP，这使得所有的 IP 地址与 4 字节的边界对齐。这不是必须的，但并不用额外的空间(IP 选项总是填充成 4 字节的倍数)，而且对齐了地址。

在本图中，我们展示了路径上的 10 个 IP 地址，但是所列的第 1 个地址被从源路径选项中挪出，在 IP 数据报离开源主机时作为它的目的地址。虽然在 40 字节的 IP 选项空间中只有 9 个 IP 地址的位置(别忘记我们刚要讨论的 3 字节选项头部)，但是加上目的地址(不是指源路径的最终目的地址，而是按源路径转发的各个 IP 数据报的目的地址字段地址)，IPv4 头部中实际上有 10 个 IP 地址。

code 对 LSRR 为 0x83，对 SSRR 为 0x89。len 用于说明选项的字节长度，包括 3 字节头部和尾部额外的目的地址。对包括 1 个 IP 地址的路径，len 为 11，2 个 IP 地址的路径，len 为 15，以此类推，len 最大为 43。NOP 不是选项的一部分，所以不包括在 len 字段，但是却包括在给 setsockopt 指定的缓冲区的大小中。当源路径选项的第一个地址挪走并被置入 IP 头部的目的地址字段后，这个 len 值将减 4(TCPv2 中的图 9.32 和 9.33)。ptr 是一个指向路径中下一个要处理的 IP 地址的指针或偏移量，其初始值为 4，即指向第一个 IP 地址。该字段的值每经过一个所列节点将加 4。

现在，我们开发三个函数以初始化、创建和处理一个源路径选项。我们的函数只处理一个源路径选项。虽然有可能将源路径和其他 IP 选项(例如 timestamp)结合起来，但是除了两个源路径选项之外，其他选项很少使用。图 24.2 是第一个函数 inet_srcrt_init 以及构造选项要用到的一些静态变量。

```
1 #include    "unp.h"
2 #include    <netinet/in_systm.h>
3 #include    <netinet/ip.h>

4 static u_char  *optr;          /* pointer into options being formed */
5 static u_char  *lenptr;        /* pointer to length byte in SRR option */
6 static int ocnt;               /* count of # addresses */

7 u_char *
```

```
 8 inet_srcrt_init(void)
 9 {
10     optr = Malloc(44);          /* NOP, code, len, ptr, up to 10 addresses */
11     bzero(optr, 44);            /* guarantees EOLs at end */
12     ocnt = 0;
13     return(optr);               /* pointer for setsockopt() */
14 }
```

图 24.2 inet_srcrt_init 函数:存放源路径前进行初始化[ipopts/sourceroute.c]

初始化

第 10～13 行 我们分配一个最大为 44 字节的缓冲区并且将其初置为 0。EOL 选项的值为 0,所以这将整个选项初始化为 EOL 字节。指向选项的指针返回给调用者以便作为第四个参数传给 setsockopt。

下一个函数 inet_srcrt_add,把一个 IPv4 地址加到正在构造的源路径上。

```
15 int
16 inet_srcrt_add(char *hostptr, int type)
17 {
18     int          len;
19     struct addrinfo *ai;
20     struct sockaddr_in *sin;

21     if (ocnt > 9)
22         err_quit("too many source routes with: %s", hostptr);

23     if (ocnt == 0) {
24         *optr++ = IPOPT_NOP;        /* NOP for alignment */
25         *optr++ = type ? IPOPT_SSRR : IPOPT_LSRR;
26         lenptr = optr++;            /* we fill in the length later */
27         *optr++ = 4;                /* offset to first address */
28     }

29     ai = Host_serv(hostptr, "", AF_INET, 0);
30     sin = (struct sockaddr_in *) ai->ai_addr;
31     memcpy(optr, &sin->sin_addr, sizeof(struct in_addr));
32     freeaddrinfo(ai);

33     optr += sizeof(struct in_addr);
34     ocnt++;
35     len = 3 + (ocnt * sizeof(struct in_addr));
36     *lenptr = len;
37     return(len + 1);   /* size for setsockopt() */
38 }
```

图 24.3 inet_srcrt_add 函数:加入一个 IPv4 地址到源路径[ipopts/sourceroute.c]

参数

第 15～16 行 第一个参数指向一个主机名或者点分十进制数形式的 IP 地址。第二个参数对宽松源路径为 0,对严格源路径为非 0。我们将看到,加到路径中的第一个地址的类型决定路径是松散的还是严格的。

检查溢出，然后初始化

第 21～28 行　我们检查没有指定太多的地址，如果是第一个地址则进行初始化。我们已经提到过，源路径选项前总是放一个 NOP。我们保存一个指向 len 字段的指针，当每个地址加入列表时，存入这个长度值。

取得二进制的 IP 地址并存入路径

第 29～37 行　我们的 host_serv 函数处理主机名或者点分十进制数串，并将最终的二进制地址存入列表。修改 len 字段的值，并返回缓冲区的大小（包括 NOP）以便调用者将其传给 setsockopt。

当一个收到的源路径通过 getsockopt 返回给应用进程时，其格式和图 24.1 不同。图 24.4 展示了这一格式。

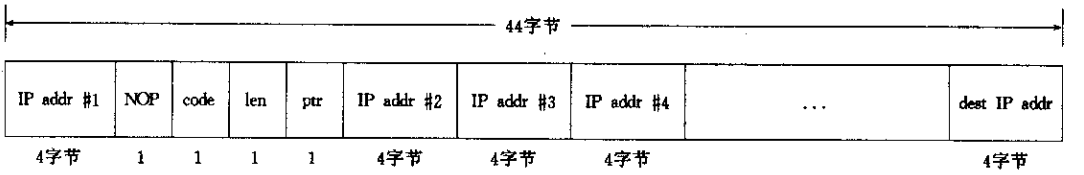


图 24.4　getsockopt 返回的源路径选项格式

首先，地址的顺序是收到的源路径被内核颠倒后的顺序。“颠倒”是指如果收到的源路由包括四个地址 A，B，C，D，则颠倒的顺序为 D，C，B，A。前 4 字节是列表中的第 1 个 IP 地址，后面是 1 字节的 NOP（为了对齐），再后面是 3 字节源路径选项头部，最后跟以其余的 IP 地址。最多有 9 个 IP 地址可以跟在 3 字节的头后，返回的 len 域的最大值为 39。因为 NOP 总是出现的，所以 getsockopt 返回的长度总是 4 的倍数。

图 24.4 所示的格式在<netinet/ip_var.h>中定义为下述结构：

```
#define MAX_IPOPTLEN 40
struct ipoption{
    struct in_addr ipopt_dst;   /* first-hop dst if source routed */
    char           ipopt_list[MAX_IPOPTLEN];   /* Options proper */
};
```

在图 24.5 中，我们觉得自己分析数据更容易，所以没有使用该结构。

这个返回的格式与我们传递给 setsockopt 的格式是不同的。如果我们想把图 24.4 中的格式转化为图 24.1 的格式，我们必须将头 4 个字节和随后的 4 个字节对换，再给 len 字段加 4。所幸的是我们不必这样做，因为源自 Berkeley 的实现自动地使用在 TCP 套接口收到的源路径的倒序值。换句话说，图 24.4 展示的信息是 getsockopt 返回的纯粹信息，我们不必调用 setsockopt 告诉内核使用该路径传送该 TCP 连接上的 IP 数据报，内核会自动这么做。我们很快就会在我们的 TCP 服务器程序例子中看到这一点。

下一个源路径函数是取得收到的图 24.4 格式的源路径，并输出该信息。图 24.5 给出的是我们的这个函数 inet_srcrt_print。

保存第一个IP地址,跳过任何NOP

第45～47行 将第1个IP地址存放在缓冲区,跳过任何后面的NOP。

```
39 void
40 inet_srcrt_print(u_char *ptr, int len)
41 {
42     u_char      c;
43     char        str[INET_ADDRSTRLEN];
44     struct in_addr hop1;
45     memcpy(&hop1, ptr, sizeof(struct in_addr));
46     ptr += sizeof(struct in_addr);
47     while ( (c = *ptr++) == IPOPT_NOP);      /* skip any leading NOPs */
48     if (c == IPOPT_LSRR)
49         printf("received LSRR: ");
50     else if (c == IPOPT_SSRR)
51         printf("received SSRR: ");
52     else {
53         printf("received option type %d\n", c);
54         return;
55     }
56     printf("%s ", inet_ntop(AF_INET, &hop1, str, sizeof(str)));
57     len = *ptr++ - sizeof(struct in_addr);      /* subtract dest IP address */
58     ptr++;          /* skip over pointer */
59     while (len > 0) {
60         printf("%s ", inet_ntop(AF_INET, ptr, str, sizeof(str)));
61         ptr += sizeof(struct in_addr);
62         len -= sizeof(struct in_addr);
63     }
64     printf("\n");
65 }
```

图24.5 inet_srcrt_print函数:输出收到的源路径[ipopts/sourceroute.c]

检查源路径选项

第48～64行 我们只输出源路径信息,从3字节头部中,我们检查code,取出len,并跳过ptr。然后我们输出3字节头部后的所有IP地址,最终的目的地址除外。

例子

现在我们修改TCP回射客户程序以说明源路径,修改TCP回射服务器程序以输出收到的源路径。图24.6是我们的客户程序。

处理命令行参数

第8～23行 我们调用inet_srcrt_init函数初始化源路径。路径的每跳由-g选项(宽松)或-G选项(严格)指定。第1个IP地址的类型(宽散或严格)说明了源路径的类型。(我们为了简化才这么做。显然我们可以增加代码以检查所有跳都为同一类型)。inet_srcrt_add函数将每个地址加到路径中。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int          c, sockfd, len = 0;
6     u_char       *ptr;
7     struct addrinfo    *ai;
8     if (argc < 2)
9         err_quit("usage: tcpcli01 [ -[gG] <hostname> ... ] <hostname>");
10    ptr = inet_srcrt_init();
11    opterr = 0;      /* don't want getopt() writing to stderr */
12    while ( (c = getopt(argc, argv, "g:G:")) != -1) {
13        switch (c) {
14        case 'g':              /* loose source route */
15            len = inet_srcrt_add(optarg, 0);
16            break;
17        case 'G':              /* strict source route */
18            len = inet_srcrt_add(optarg, 1);
19            break;
20        case '?':
21            err_quit("unrecognized option: %c", c);
22        }
23    }
24    if (optind != argc-1)
25        err_quit("missing <hostname>");
26    ai = Host_serv(argv[optind], SERV_PORT_STR, AF_INET, SOCK_STREAM);
27    sockfd = Socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
28    if (len > 0) {
29        len = inet_srcrt_add(argv[optind], 0);      /* dest at end */
30        Setsockopt(sockfd, IPPROTO_IP, IP_OPTIONS, ptr, len);
31        free(ptr);
32    }
33    Connect(sockfd, ai->ai_addr, ai->ai_addrlen);
34    str_cli(stdin, sockfd);      /* do it all */
35    exit(0);
36 }
```

图 24.6　指定源路径的 TCP 回射客户程序[ipopts/tcpcli01.c]

这是我们第一次碰到 Posix.2 getopt 函数。该函数的第三个参数是一个字符串，说明我们允许的命令行参数字节，本例为 g 和 G。每个字母后跟一个冒号，指明该选项有一个参数。这个函数与四个定义在<unistd.h>中的全局变量一起工作：

```
extern char *optarg;
extern int optind,opterr,optopt;
```

在调用 getopt 之前，将 opterr 设置为 0 以阻止该函数将出错消息(如果有

的话)写到标准错误输出上,因为我们自己处理错误。Posix.2规定,如果第三个参数的第1个字节是冒号,这同样会阻止函数将出错消息写到标准错误输出上,但并非所有的实现都支持这一点。

处理目的地址,创建套接口

第24～27行 最后一个命令行参数是主机名或者服务器的点分十进制数地址,由函数host_serv处理。我们不能够调用tcp_connect函数,因为在socket和connect调用之间必须指定源路径。connect启动了三路握手,但是我们想要最初的SYN和后续的分组都使用这个源路径。

第28～34行 如果指定了一个源路径,我们必须将服务器的IP地址加到IP地址列表的末尾(图24.1)。setsockopt给套接口安装源路径。然后我们调用connect,紧接着是str_cli函数(图5.5)。

我们的TCP服务器程序几乎和图5.12给出的代码相同,只有以下几点变化。第一,我们给选项分配空间:

```
int     len;
u_char *opts;

opts = Malloc(44);
```

然后在调用accept之后、fork之前获取IP选项:

```
len = 44;
Getsockopt(connfd,IPPROTO_IP,IP_OPTIONS,opts,&len);
if (len>0){
    printf("received IP options,len=%d\n",len);
    inet_srcrt_print(opts,len);
}
```

如果从客户收到的SYN未包括任何IP选项,从getsockopt返回的len变量将为0(len是一个值-结果参数)。我们早先提到,我们不必做任何事情使TCP使用收到的源路径的倒序值,这是TCP自动做的(TCPv2第931页)。我们调用setsockopt所做的全部工作就是取得一份收到的源路径的拷贝。如果我们不想让TCP使用这条路径,我们可以在accept返回之后调用setsockopt,将第五个参数(长度)置为0,这将取消正在使用的IP选项。TCP在三路握手(图2.5)的第二个参数已经使用了源路径,但如果我们取消了该选项,IP将给送往客户的以后的分组使用计算出的任何路径。

现在,我们给出指定源路径时的客户-服务器例子。我们在主机solaris上如下运行客户:

```
solaris %tcpcli01 -g gw -g sunos5 bsdi
```

IP数据报将从solaris传到路由器gw(图1.16),再传到主机sunos5,然后传到运行服务器的主机bsdi。两个中间系统必须转发源路由的数据报,以使本例工作。

当连接在服务器方建立时,有如下的输出:

```
bsdi % tcpserv01
received IP options,len=16
received LSRR:206.62.226.36  206.62.226.62  206.62.226.33
```

输出的第一个 IP 地址是反方向路径的第一跳(sunos5,如图 24.4 所示),下两个地址的顺序是服务器用来将数据报传回给客户的。如果我们用 tcpdump 观察客户-服务器的交互,我们就可以看到两个方向上每个数据报中的源路径选项。

> 不幸的是,IP_OPTIONS 套接口选项的操作从来没有正式文档,所以在不是源自 Berkeley 源代码的系统上,可能会有所改变。例如,在 Solaris 2.5 下,getsockopt 返回的缓冲区中的第一个地址(图 24.4)不是返回路径的第一跳的地址,而是对方主机的地址。但是,TCP 使用的反方向路径是正确的。另外,Solaris 2.5 的源路径选项前有 4 个 NOP,限制选项中只能有 8 个 IP 地址,而不是真正的上限 9。

删除收到的源路径

不幸的是源路径存在一个安全漏洞,从 Net/1 版本(1989)开始,rlogind 和 rshd 服务器程序有类似下述的代码:

```
u_char    buf[44];
char      lbuf[BUFSIZ];
int       optsize;

optsize = sizeof(buf);
if (getsockopt(0, IPPROTO_IP,IP_OPTIONS,
               buf,&optsize) == 0 && optsize != 0) {
    /* format the options as hex numbers to print in lbuf[] */
    syslog(LOG_NOTICE,
           "Connection received using IP options (ignored): %s",lbuf);
    setsockopt(0,ipproto,IP_OPTIONS, NULL, 0);
}
```

如果一个连接到达时有 IP 选项(getsockopt 返回的 optsize 值不为 0),那么 syslog 会登记一条信息,setsockopt 则被调用来清除这些选项。这可阻止将来在这个连接上发送的 TCP 分节使用接收到的源路径的反转路径。这个技巧现在知道是不够的,因为应用进程接受连接时,TCP 三路握手已经完成。三路握手的第二个分节(图 2.5 中服务器的 SYN-ACK)已经通过源路径的反转路径回到了客户(或者至少回到了源路径中所列的某个中间节点,黑客可能就在该节点上)。因为黑客已经看到了两个方向 TCP 的序列号,即使没有分组再用源路径发送,黑客仍然可以用正确的序列号发送分组给服务器。

解决这个潜在问题的唯一方法是当你使用源 IP 地址进行某种形式认证(像 rlogind 和 rshd 所做的那样)时,禁止使用源路径到达的所有 TCP 连接。在刚给出的代码片段中,将 setsockopt 替换为关闭刚刚接受的连接并终止新派生的服务器。三路握手的第二个分节尽管已经发出,但是连接不应该仍然开着。

24.4　IPv6 扩展头部

在图 A.2 中,我们没有随 IPv6 头部(它的长度总是 40 字节)一起展示任何选项,但是 IPv6 头部可以跟以可选扩展头部。

1. 步跳(hop_by_hop)选项必须紧跟40字节的IPv6头部。目前没有定义这种可供应用程序使用的选项。
2. 目的(destination)选项:目前没有定义这种可供应用程序使用的选项。
3. 路由头部(routing header):这是一个源路由选项,在概念上类似于我们在24.3节中描述的IPv4源路径选项。
4. 分片头部(fragmentation header):该头部由将IPv6数据报分片的主机自动生成;由最终的目的主机重组片段时处理。
5. 认证头部(authentication header,AH):该头部的用法由RFC 1826[Atkinson 1995a]和[Kent and Atkinson 1997a]说明。
6. 封装安全有效负载(encapsulating security payload,ESP)头部:该头部的用法在RFC 1827[Atkinson 1995b]和[Kent and Atkinson 1997b]中说明。

我们说过分片头部全部由内核处理,[McDonald 1997]建议用套接口选项处理AH和ESP头部。这样只剩下了前三个选项,我们将在下两节中讨论。

24.5 IPv6步跳选项和目的选项

步跳选项和目的选项有类似的格式(如图24.7所示)。8位的下一个头部字段标识本扩展头部后边的下一个头部。8位的头部扩展长度是该扩展头部的长度,以8字节为单位,但不包括第1个8字节。例如,如果该扩展头部占用8字节,其头部扩展长度就为0。如果扩展头部占用16字节,其头部扩展长度就为1,以此类推。这两个头部都被填充成8字节的整数倍。填充值有pad1选项或padN选项,我们很快就会描述这两个选项。

步跳选项头部和目的选项头部各自拥有任意数目的单个选项,其格式如图24.8所示。

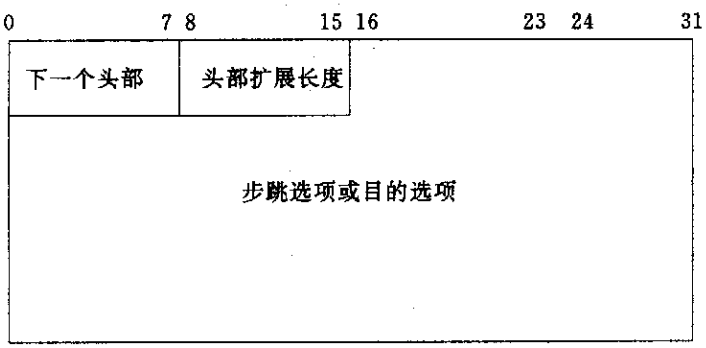


图24.7 步跳选项和目的选项的格式

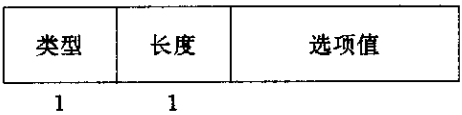


图24.8 步跳选项和目的选项的单个选项的格式

这称为 TLV 编码，因为：每个选项以类型(type)、长度(length)和值(value)的形式出现。8 位的类型字段标识选项类型。另外，该字段的高序两位指定 IPv6 节点在不理解该选项的情况下如何处理这个选项：

00　跳过这个选项，继续处理该头部。

01　扔掉分组。

10　扔掉分组并发送一个 ICMP 参数问题类型 2 错误(图 A.16)给发送者，不管分组的目的地址是不是一个多播地址。

11　扔掉分组并发送一个 ICMP 参数问题类型 2 错误(图 A.16)给发送者，但只有在分组的目的地址不是一个多播地址时才发送错误。

下一个高序位说明选项数据在途中有无变化：

0　选项数据在途中无变化。

1　选项数据在途中可能变化。

低 5 位说明选项本身。

8 位的长度字段说明选项数据的字节长度，类型字段和本长度字段不包括在长度之内。

两个填充选项在 RFC 1883 [Deering and Hinden 1995]中定义，可以用在步跳选项头部中或目的选项头部中。特大有效负载长度(jumbo payload length)是一个步跳选项，也在 RFC 1883 中定义。它按照需要产生，内核在全部收到后处理。[Katz et/al. 1997]提出了一种新的 IPv6 步跳选项，和 IPv4 路由器警告(rounter alert)类似。我们用图 24.9 展示这些选项。

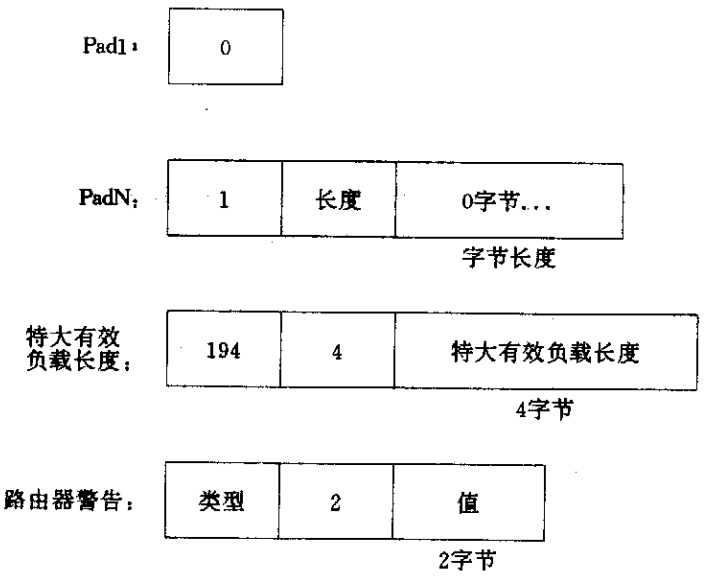


图 24.9　IPv6 步跳选项

pad1 字节是唯一一个没有长度和值的选项。它提供 1 字节的填充。padN 选项用于需要 2 个或更多字节填充的场合。对于 2 字节填充，其长度为 0，该选项只包括类型字段和长度字段。对于 3 字节填充，长度为 1，1 字节的 0 值紧跟在长度字段后。特大有效负载长度选项提供一个 32 位的数据报长度，当图 A.2 中的 16 位有效负载长度字段不够时使用。

我们展示这些选项还因为每一个步跳选项或目的选项都有一个对齐要求(alignment requirement),写作 xn+y。其意思是该选项必须出现在距离头部开始位置 x 字节整数倍加 y 字节的地方。例如:特大有效负载长度选项的对齐要求是 4n+2,这便迫使 4 字节的选项值(特大有效负载长度)在 4 字节的边界上。y 值为 2 的原因是出现在每个步跳选项和目的选项头部开始处的 2 字节(图 24.8)。

步跳选项和目的选项通常指定成 sendmsg 的辅助数据,由 recvmsg 作为辅助数据返回。应用程序不必做任何特别的事情来发送其中一种选项和全部两种选项,只需要在调用 sendmsg 时指定它们即可。但是为了接收这些选项,对应的套接口选项必须打开:步跳选项是 IPV6_HOPOPTS,目的选项是 IPV6_DSTOPTS。例如,为了使这两种选项都返回:

```
const int on=1;
setsockopt(sockfd,IPPROTO_IPV6,IPV6_HOPOPTS,&on,sizeof(on));
setsockopt(sockfd,IPPROTO_IPV6,IPV6_DSTOPS,&on,sizeof(on));
```

图 24.10 展示了用来发送和接收步跳选项和目的选项的辅助数据对象的格式。

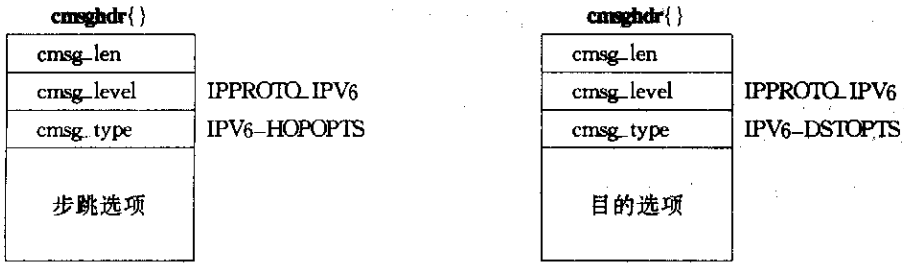


图 24.10 步跳选项和目的选项的辅助数据对象

和我们描述过的其他 IPv6 辅助数据对象(图 20.21)不同,这些对象由每个实现决定通过它们的 cmsg_data 部分在进程和内核之间传递什么内容。虽然这些内容没有定义,但却定义了 6 个创建和处理这些辅助数据对象的函数。下述四个函数构造一个待发送的选项。

```
#include <netinet/in.h>
int inet6_option_space(int nbytes);
                                                    返回:存放选项所需的字节数
int inet6_option_init (void *buf,struct cmsghdr **cmsgp,int type);
                                                    返回:成功时为0,出错时为-1
int inet6_option_append (struct cmsghdr *cmsg, const uint8_t *typep, int multx, int
                         plusy);
                                                    返回:成功时为0,出错时为-1
uint8_t *inet6_option_alloc (struct cmsghdr *cmsg,int datalen,int multx,int plusy);
                                返回:成功时为指向选项类型字段的指针,出错时为NULL
```

inet6_option_space 返回容纳一个选项需要的字节数,包括开头的 cmsghdr 结构和末尾的填充字节。参数 nbytes 是定义选项的结构的大小,必须包括开头的任何填充字节(对齐要求 xn+y 中的 y 值)、选项类型、长度和数据。

inet6_option_init 对于每个将容纳步跳选项或目的选项的辅助数据对象只调用一次。buf 指向将容纳辅助数据对象的缓冲区。cmsgp 是指向 cmsghdr 结构的指针地址，该函数用 buf 指向的缓冲区初始化这个结构，并在 * cmsgp 中返回一个指向该结构的指针。type 或者是 IPV6_HOPOPTS，或者是 IPV6_DSTOPTS，其值被存入所构建的 cmsghdr 结构的 cmsg_type 成员中。

inet6_option_append 将步跳选项或目的选项加进由 inet6_option_init 初始化的辅助数据对象中。cmsg 是一个指针，指向由 inet6_option_init 初始化的 cmsghdr 结构。typep 是一个指向 8 位选项类型的指针，选项类型之后必须跟以 8 位的选项长度以及选项数据(TLV)。调用者在调用该函数前设置好这些值。multx 和 plusy 是该选项对齐要求中的 x 项和 y 项。

前边描述的函数要求调用者构造选项的 TLV 并且用 typep 参数传递指针，选项便被拷入辅助数据对象中。另一种方法是，inet6_option_alloc 函数返回一个指向辅助数据对象的指针，调用者必须自己将选项 TLV 存入辅助数据对象中。cmsg 是一指针，指向由 inet6_option_init 初始化的 cmsghdr 结构。datalen 是该选项的长度字节的值(TLV 中的 L 值)，它是计算是否必须给该选项添加填充所需的。multx 和 plusy 是该选项对齐要求中的 x 项和 y 项。

剩下的两个函数处理收到的选项。

```
#include <netinet/in.h>
int inet6_option_next (const struct cmsghdr *cmsg,uint8_t **tptrp);
                    返回：有另一个选项要处理时为 0，到达选项末端或出错时为 -1
int inet6_option_find (const struct cmsghdr *cmsg,uint8_t *tptrp,int type);
                                        返回：成功时为 0，出错时为 -1
```

inet6_option_next 处理缓冲区中的下一个选项。cmsg 指向一个 cmsghdr 结构，该结构的 cmsg_level 必须为 IPPROTO_IPV6，cmsg_type 必须为 IPV6_HOPOPTS 或者 IPV6_DSTOPTS。当第一次给某个给定的辅助数据对象调用该函数时，* tptrp 必须为空指针，然后每次该函数返回时，* tptrp 就指向下一个欲处理的选项的 8 位选项类型。* tptrp 的值被该函数用来记忆从一次调用到下次调用中在辅助数据对象中的位置。当最后一个选项处理完后，返回值是 -1，* tptrp 是空指针。如果出现错误，则返回值为 -1，* tptrp 为非空指针。

inet6_option_find 和上一个函数类似，但是它让调用者说明要搜索的选项类型(type 参数)，而不是总是返回下一个选项。

24.6　IPv6 路由头部

IPV6 路由头部用于 IPv6 的源路由。该头部的前 2 个字节和我们在图 24.7 中展示的相同：一个下一个头部(next header)字段，后跟以头部扩展长度(header extension length)。下两个字节指定路由类型(routing type)和剩余网段(segments left)的数目(即还有多少列出的节点未被访问)。只有一种类型的路由头部(类型 0)被说明，在图 24.11 中展示的是它的格式。

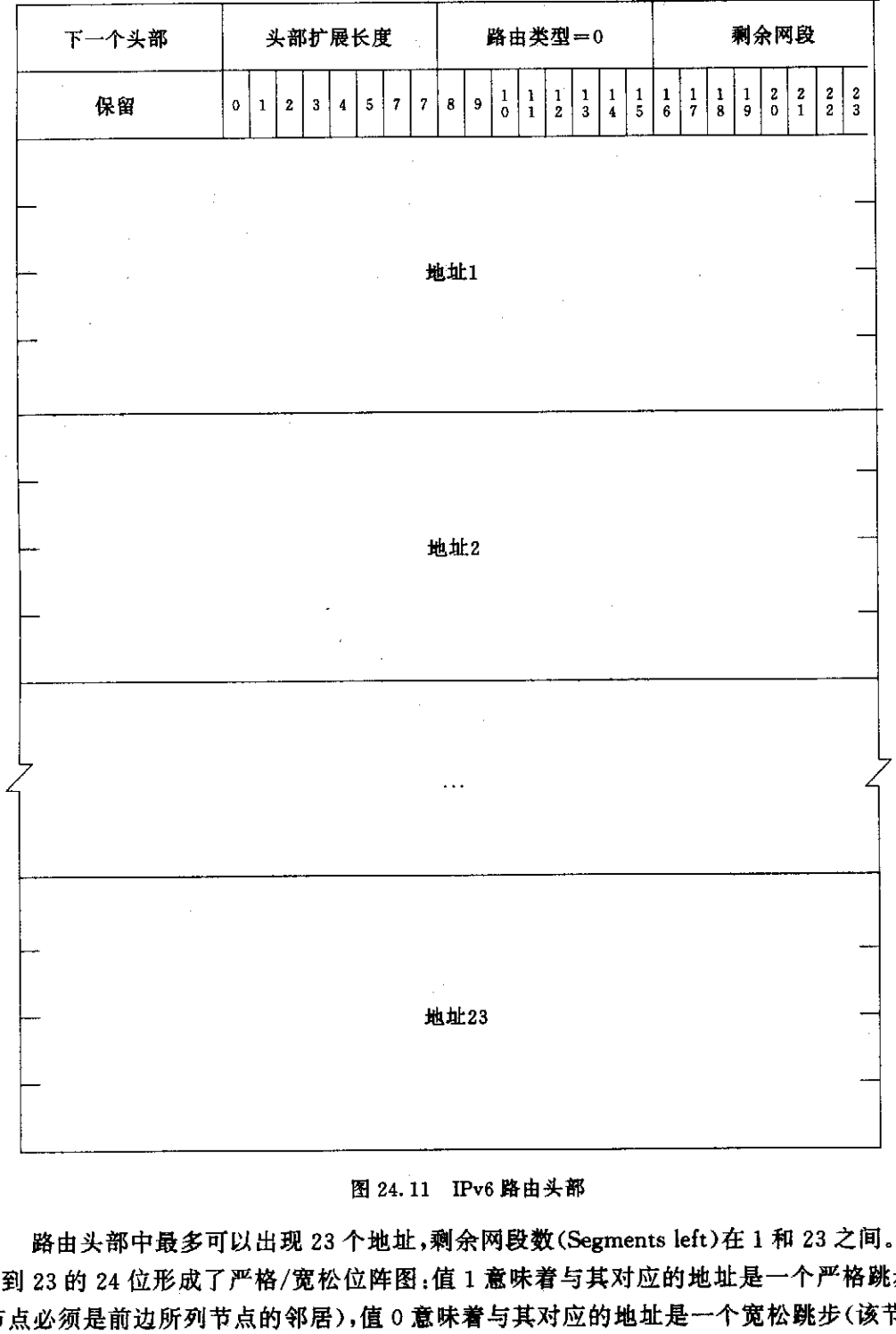


图 24.11 IPv6 路由头部

路由头部中最多可以出现 23 个地址，剩余网段数(Segments left)在 1 和 23 之间。标以 0 到 23 的 24 位形成了严格/宽松位阵图：值 1 意味着与其对应的地址是一个严格跳步(该节点必须是前边所列节点的邻居)，值 0 意味着与其对应的地址是一个宽松跳步(该节点不一定是前边所列节点的邻居)。标为 1 的位对应于地址 1，标为 2 的位对应于地址 2，以此类推。标为 0 的位对应最后一跳步。RFC 1883[Deering and Hinden 1995]说明了在分组到达最

终目的节点的过程中，如何处理路由头部，并给出了详细的例子。

路由头部通常作为辅助数据在调用 sendmsg 时指定，又由 recvmsg 作为辅助数据返回。应用程序不必为发送该头部做任何事情：只在调用 sendmsg 时指定就行。但要接收路由头部，IPV6_RTHD 套接口选项必须打开，如下所示：

```
const int on = 1;
setsockopt(sockfd,IPPROTO_IPV6,IPV6_RTHDR,&on,sizeof(on));
```

图 24.12 给出了用于发送和接收路由头部的辅助数据对象的格式。与步跳选项和目的选项的辅助数据对象(图 24.10)类似，该对象也是由每个实现决定通过它的 cmsg_data 部分在进程和内核之间传递什么内容。为创建和处理路由头部定义了 8 个函数。以下 4 个函数用于构造一个待发送的选项。

```
#include <netinet/in.h>
size_t inet6_rthdr_space(int type,int segments);
                                        返回：成功时为正数字节，出错时为 0
struct cmsghdr * inet6_rthdr_init(void * buf, int type);
                                        返回：成功时为非空指针，出错时为 NULL
int inet6_rthdr_add (struct cmsghdr * cmsg, const struct in6_addr * addr,
                     unsigned int flags);
                                        返回：成功时为 0，出错时为 -1
int inet6_rthdr_lasthop(struct cmsghdr * cmsg,unsigned int flags);
                                        返回：成功时为 0，出错时为 -1
```

inet6_rthdr_space 返回容纳一个路由头部辅助数据对象所需的字节数，该头部具有所指定的类型 type(通常指定成 IPV6_RTHDR_TYPE_0)，并具有所指定网段数 segments 所需的空间。这个大小还包括 cmsghdr 结构。

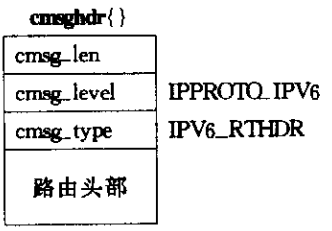


图 24.12　IPv6 路由头部的辅助数据对象

inet6_rthdr_space 和 inet6_option_space 都返回某种类型的辅助数据对象所需的字节数，但两者都不分配内存。这是为了提防调用者希望分配更大的缓冲区以同时容纳其他辅助数据对象的情况。

inet6_rthdr_init 初始化由 buf 指向的缓冲区以容纳一个辅助数据对象，它是类型为指定的 type 的路由头部。返回值是指向在这个缓冲区中构造的 cmsghdr 结构的指针，该指针然后用作下面两个函数的一个参数。cmsg_level 和 cmsg_type 成员也被初始化。

inet6_rthdr_add 把由 addr 指向的 IPv6 地址加到正在构造中的路由头部的末端。flags

参数要么是IPV6_RTHDR_LOOSE,要么是IPV6_RTHDR_STRICT。调用成功时cmsg_len成员更新成计入这个新地址。

inet6_rthdr_lasthop指定最后一跳的flags(IPV6_RTHDR_LOOSE或IPV6_RTHDR_STRICT)。我们知道具有N个地址的路由头部有N+1跳。这需要调用inet6_rthdr_add共N次,外加调用inet6_rthdr_lasthop一次。

下面是处理收到的路由头部的4个函数。

```
#include <netinet/in.h>
int inet6_rthdr_reverse(const struct cmsghdr *in, struct cmsghdr *out);
                                                    返回:成功时为0,出错时为-1
int inet6_rthdr_segments(const struct cmsghdr *cmsg);
                                          返回:成功时为路由头部中的网段数,出错时为-1
struct in6_addr *inet6_rthdr_getaddr(struct cmsghdr *cmsg, int index);
                                          返回:成功时为非空指针,出错时为NULL
int inet6_rthdr_getflags(const struct cmsghdr *cmsg, int index);
                                          返回:成功时宽松/严格标志,出错时为-1
```

inet6_rthdr_reverse根据作为辅助数据收到的路由头部(由参数in所指向)建立一个新的路由头部(建在由out指向的缓冲区中),这样数据报可以沿着这条反转了的路径发送出去。路径的反转可以在同一位置发生;也就是说,in和out这两个指针可以指向同一个缓冲区。

inet6_rthdr_segments返回由cmsg描述的路由头部中网段的数目。调用成功时返回值在1~23之间(包括1和23)。

inet6_rthdr_getaddr返回一个指针,它指向由cmsg描述的路由头部中由index指定的IPv6地址。index的值必须在1和由inet6_rthdr_segments返回的值之间(包括这两个值)。

inet6_rthdr_getflags返回由cmsg描述的路由头部中相应于index的标志值IPV6_RTHDR_LOOSE或IPV6_RTHDR_STRICT。index的值同样必须在1和由inet6_rthdr_segments返回的值之间(包括这两个值)。

> 地址索引从1开始,宽松/严格标志索引从0开始,如图24.11所示。这和RFC 1883 [Decring and Hinden 1995]中的表示方法一致。

24.7 IPv6粘附选项

我们已经描述了用sendmsg和recvmsg的辅助数据发送和接收六种不同的辅助数据对象:

1. IPv6分组信息:in6_pktinfo结构或者包含目的地址和外出接口索引,或者包含源地址和到达接口索引(图20.21)。
2. 外出跳限或收到的跳限(图20.21)。
3. 下一跳地址(图20.21)。

4. 步跳选项(图 24.10)。
5. 目的选项(图 24.10)。
6. 路由头部(图 24.12)。

我们在图 13.11 中总结了这些对象的 cmsg_level 值和 cmsg_type 值,以及其他辅助数据对象的值。

不用每次调用 sendmsg 时都发送这些选项,我们可以代以设置 IPV6_PKTOPTIONS 套接口选项。当设置这一选项时,第四个参数指向一个缓冲区,该缓冲区包含了该套接口上应随所有分组发送的所有辅助数据对象。缓冲区的格式就像是指定该缓冲区为 sendmsg 的辅助数据一样。这些选项称为"粘附"的,因为一旦它们设置后,就永远设置了,直到清除为止(用长度为 0 设置相应套接口选项)。但是对于 UDD 套接口和原始 IP 套接口,这些选项可以用给 sendmsg 调用指定辅助数据的方法,针对每个分组覆盖这些粘附选项。如果在 sendmsg 调用中指定了任何辅助数据,任何粘贴选项都不随数据报发送。

由于在 TCP 套接口上从来不用 sendmsg 和 recvmsg 发送和接收辅助数据,因此同样适用于 TCP 的粘附选项概念是实现这一目的的唯一手段。TCP 应用程序可以设置 IPV6_PKTOPTIONS 套接口选项,并指定本节开始时提到过的 6 种辅助数据对象的任何一个或多个。这些对象可以影响在该套接口上发送的所有分组。

TCP 应用程序也能给 IPV6_PKTOPTIONS 套接口选项调用 getsockopt,以检索这些辅助数据对象。这种情况下内核仅维护来自最近接收到的分节中的选项,所返回的是应用程序显式指明的那些选项(通过打开相应的套接口选项)。

24.8 小 结

在 10 个 IPv4 选项中,最常用的就数源路径选项了,不过由于安全性的原因,它的使用日益受限。可通过 IP_OPTIONS 套接口选项访问 IPv4 头部中的选项。

IPv6 定义了 6 个扩展头部,虽然目前对它们的支持几乎没有。访问 IPv6 的扩展头部要通过函数接口,这也屏蔽了它们在分组中的具体格式。这些扩展头部通过 sendmsg 和 recvmsg 作为辅助数据收发。

24.9 习 题

24.1 对 24.3 节末尾的源路径例子,如果我们不是给客户指定一个主机名 gw,而是指定该路由器的另一个 IP 地址 206.85.40.74,服务器会有什么变化?

24.2 在 24.3 节末尾的源路径例子中,如果我们使用 -G 选项而不是-g 选项给客户指定各个中间节点,会有什么变化?

24.3 给 IP_OPTIONS 套接口选项调用 setsockopt 时指定的缓冲区长度必须是 4 字节的倍数。如果我们不像图 24.1 那样在缓冲区头部插入一个 NOP,该怎么办?

24.4 当使用 IP 记录路径(Record Route)选项时(在 TCPv1 的 7.3 节中叙述),ping 如何接收一个源路径?

24.5 在24.3节末尾出自rlogind服务器程序的用于清除接收到的源路径的示例代码中，为何getsockopt和setsockopt的套接口描述字参数为0。

24.6 很多年以来，24.3节末尾给出的用于清除接收到的源路径的代码大体如下：

```
optsize=0;
setsockopt(0,ipproto,IP_OPTIONS,NULL,&optsize);
```

这段代码有什么问题？会出事么？

第 25 章　原始套接口

25.1　概　述

原始套接口提供以下三种 TCP 及 UDP 套接口一般不提供的功能。

1. 使用原始套接口可以读/写 ICMPv4、IGMPv4 及 ICMPv6 分组。例如：Ping 程序(详见 25.5 节)，就使用原始套接口发送 ICMP 回射请求，并接受 ICMP 回射应答。用于多播路由的守护进程：mrouted，同样利用原始套接口来发送和接收 IGMPv4 分组。上述功能同样允许使用 ICMP 或 IGMP 构造的应用程序完全作为用户进程处理，而不必再增加过多的内核编码。例如，路由器发现守护进程(在 Solaris 2.x 下名为 in.rdisc，TCPv1 的附录 F 说明如何获得公开可得版本的源代码)即以这种方式构造。它处理内核完全不知道的两个 ICMP 消息(路由器通告和路由器征求)。
2. 使用原始套接口可以读/写特殊的 IPv4 数据报，内核不处理这些数据报的 IPv4 协议字段。回想一下图 A.1 所给出的 IPv4 的 8 位协议字段。大多数内核只处理值为 1(ICMP)、2(IGMP)、6(TCP)和 17(UDP)的数据报。但协议字段还可能为其他值：RFC 1700[Reynolds and Postel 1994]列出了所有值。例如，OSPF 路由协议就不使用 TCP 或 UDP，而直接用 IP，将 IP 数据报的协议字段设为 89。因此，由于这些数据报包含内核完全不知道的协议字段，实现 OSPF 协议的 gated 程序就必须使用原始套接口来读写它们。此项同样适用于 IPv6。
3. 利用原始套接口，使用 IP_HDRINCL 套接口选项可以构造自己的 IPv4 头部。我们可以使用这个特性来构造自己的 TCP 或 UDP 分组。26.6 节将给出一个这样的例子。

25.2　原始套接口创建

创建一个原始套接口涉及以下几步：

1. 当第二个参数是 SOCK_RAW 时，调用 socket 函数创建一个原始套接口。第三个参数(协议)一般不应为 0。例如，为了创建一个 IPv4 原始套接口，我们可以这样写：

```
int sockfd;
sockfd = socket(AF_INET, SOCK_RAW, protocol);
```

其中 protocol 参数值为形如 IPPROTO_xxx 的常值，由<netinet/in.h>头文件定义，如 IPPROTO_IGMP。但要注意，头文件里定义了一个协议名，如 IPPROTO_EGP，并不意味着内核肯定支持它。

为了防止普通用户向网络写自己的 IP 数据报，只有超级用户才有权创建原始套接口。

2. 可以设置 IP_HDRINCL 套接口选项：

```
const int on=1;
if (setsockopt (sockfd, IPPROTO_IP,IP_HDRINCL,&on, sizeof(on))< 0)
    出错
```

我们将在下一节介绍这个选项的效果。

3. 可以对原始套接口调用 bind 函数，但并不常用。该函数仅用来设置本地地址：对于一个原始套接口而言端口号没有什么意义。当进行输出的时候，bind 设置在原始套接口上所发送的数据报中将用到的源 IP 地址(仅当 IP_HDRINCL 套接口选项未设置时)；若不调用 bind，则由内核将源 IP 地址设成外出接口的主 IP 地址。
4. 在原始套接口上可调用 connect 函数，但也不常用。connect 函数仅设置目的地址；再重申一遍：端口号对原始套接口而言没有意义。对于输出而言，调用 connect 之后，由于目的地址已经指定，我们可以调用 write 或 send，而不是 sendto 了。

25.3 原始套接口输出

原始套接口的输出遵循以下规则：

1. 普通输出通过调用 sendto 或 sendmsg 并指定目的 IP 地址来完成。如果套接口已经连接，也可以调用 write、writev 或 send。
2. 如果 IP_HDRINCL 选项未设置，则内核写的数据起始地址指 IP 头部之后的第一个字节。因为这种情况下，内核将构造 IP 头部，并将它安在来自进程的数据之前。内核将 IPv4 头部的协议字段设置成用户在调用 socket 函数时所给的第三个参数。
3. 如果 IP_HDRINCL 选项已设置，则内核写的数据起始地址指 IP 头部的第一个字节。用户所提供的数据大小值必须包括头部的字节数。此时进程构造除了以下两项以外的整个 IP 头部：(a)IPv4 标识字段可以设为 0，要求内核设置该值。(b)IPv4 头部的校验和由内核来计算和存储。
4. 对于超出外出接口 MTU 的分组，内核将其分片。

非常不幸，IP_HDRINCL 选项从未正式地说明过，特别是 IP 头部中的字节序的问题。对于源自 Berkeley 的内核而言，ip_len 和 ip_off 采用主机字节序，其他字段使用网络字节序(TCPv2 第 233 页和第 1057 页)。而对 Linux，所有字段均需使用网络字节序。

在 4.3BSD Reno 引入 IP_HDRINCL 之前，应用程序在原始 IP 套接口上给出自己的 IP 头部的唯一途径是：使用 Van Jacobson 于 1988 年提供的一个内核补丁(用于支持 Traceroute)。此补丁要求应用进程创建一个协议为 IPPROTO_RAW 的原始 IP 套接口。IPPROTO_RAW 实际值为 255(作为一个保留值，该值不能在 IP 头部的协议字段里出现)。

原始套接口上的输入和输出函数是内核中最简单的一部分。举个例子，在 TCPv2 中，每个函数只需 40 行左右的 C 代码(第 1054～1057 页)，而普通 TCP 的输入函数长达约 2000 行，输出函数也有约 700 行。

本书对 IP_HDRINCL 套接口选项的描述基于 4.4BSD。早期版本，如 Net/2，在使用该选项时，在 IP 头部中要填入更多的字段。

对于 IPv4 而言，用户进程还必须计算和设置 IPv4 头部之后所跟的所有头部的校验和。例如，在我们的 Ping 程序中(图 25.13)，我们必须计算 ICMPv4 的校验和，并在调用 sendto 之前将它存储在 ICMPv4 头部中。

IPv6 的差异

IPv6 的原始套接口与 IPv4 相比有以下几点不同[Stevens and Thomas 1997]：

- IPv6 原始套接口发送和接收的协议头部内，所有字段均使用网络字节序。
- IPv6 不存在类似于 IPv4 中 IP_HDRINCL 这样的套接口选项。使用 IPv6 原始套接口无法读写整个 IPv6 分组(包括扩展头部)，不过通过套接口选项及辅助数据(习题 25.1)，我们几乎可取得 IPv6 头部所有的字段和所有扩展头部。但要读/写整个 IPv6 数据报，仍必须使用数据链路访问(第 26 章)。
- IPv6 原始套接口的校验和处理有差异，很快我们将谈到这一点。

IPV6_CHECKSUM 套接口选项

对于 ICMPv6 原始套接口而言，总是由内核来计算并在 ICMPv6 头部内存储其校验和。这一点与 ICMPv4 原始套接口有所不同。ICMPv4 原始套接口由应用进程来计算并存储校验和(试比较图 25.13 和图 25.15)。此外，虽然 ICMPv4 和 ICMPv6 都要求发送方计算校验和，ICMPv6 在其校验和中还包括一个伪头部(pseudoheader)(当我们在图 26.13 中计算 UDP 的校验和时，我们将讨论伪头部的概念)。伪头部所包含的字段中有一个源 IPv6 地址字段，应用进程一般让内核来设置其值。为了避免应用进程仅仅为了计算校验和而设置该地址，由内核来计算校验和更为方便。

对于其他 IPv6 原始套接口(即调用 socket 时第三个参数不为 IPPROTO_ICMPV6 的那些原始套接口)而言，可以使用一个套接口选项来通知内核，是否为外出分组计算和存储校验和，以及验证接收分组的校验和。缺省情况下，此选项是关闭的，如果给予一个非负的值，则此选项将打开。例如：

```
int offset=2;
if(setsockopt (sockfd,IPPROTO_IPV6,IPV6_CHECKSUM,&offset, sizeof(offset)) < 0)
    出错
```

上例不但设置套接口的校验和选项，而且通知内核该 16 位校验和的字节偏移量：本例中为自应用数据起始位置起，偏移 2 个字节。为了清除此选项，可以设置偏移量为 −1。当此选项设置时，内核将为输出分组计算和存储校验和，并对输入分组的校验和进行验证。

25.4　原始套接口输入

对于原始套接口输入，我们要回答的第一个问题是：接收到的哪些 IP 分组将传递给原始套接口。这将遵循如下规则：

1. 接收到的 TCP 分组和 UDP 分组决不会传递给任何原始套接口，如果一个进程希望

读取包含 TCP 或 UDP 分组的 IP 数据报，那么它们必须在数据链路层读入，见第 26 章。

2. 当内核处理完 ICMP 消息之后，绝大部分 ICMP 分组将传递给原始套接口。对源自 Berkeley 的实现而言，除了回射请求、时间戳请求和地址掩码请求将完全由内核处理以外(见 TCPv2 第 302～303 页)，所有收到的 ICMP 分组都将传递给某个原始套接口。
3. 当内核处理完 IGMP 消息之后，所有 IGMP 分组都将传递给某个原始套接口。
4. 所有带有内核不能识别的协议字段的 IP 数据报都将传递给某个原始套接口。内核对这些分组唯一做的就是检验 IP 头部中的某些字段：IP 版本、IPv4 头部校验和、头部长度以及目的 IP 地址(TCPv2 第 213～220 页)。
5. 如果数据报以片段形式到达，则该分组将在所有片段到达并重组后才传给原始套接口。

当内核准备好一个待传递的数据报之后，内核将对所有进程的原始套接口进行检查，以寻找所有匹配的套接口。每个匹配的套接口都将收到一个该 IP 数据报的拷贝。以下是对每个原始套接口所做的三个测试，只有当这三个测试都为真时，数据报才会递送给该套接口。

1. 如果在创建原始套接口时，所指定的 protocol 参数不为零(socket 的第三个参数)，则接收到的数据报的协议字段应与该值匹配。否则该数据报将不递送给该套接口。
2. 如果此原始套接口之上绑定了一个本地 IP 地址，那么接收到的数据报的目的 IP 地址应与该绑定地址相匹配，否则该数据报将不递送给该套接口。
3. 如果此原始套接口通过调用 connect 指定了一个对方 IP 地址，那么接收到的数据报的源 IP 地址应与该已连接地址相匹配，否则该数据报将不递送给该套接口。

注意如果一个原始套接口以 protocol 参数为 0 的方式创建，并且未调用 connect 和 bind，那么对于内核传递给原始套接口的每一个原始数据报，该套接口都会收到一份拷贝。

此外，当一个接收到的数据报传递给 IPv4 原始套接口时，整个数据报(包括 IP 头部)都将传递给进程。对于原始 IPv6 套接口，除扩展头部外的整个数据报内容都传递给套接口(例如图 25.11 和 25.21)。

在传递给应用进程的 IPv4 头部内，ip_len、ip_off 和 ip_id 是主机字节序的，其他字段是网络字节序的。在 Linux 下，所有字段均为网络字节序。

上一节我们还提过，在原始 IPv6 套接口上收到的数据报中，所有字段均为网络字节序。

ICMPv6 类型过滤

原始 ICMPv6 套接口接收绝大部分内核收到的 ICMPv4 消息。但是 ICMPv6 是 ICMPv4 的一个超集，包括 ARP 和 IGMP(2.2 节)的功能。因此，原始 ICMPv6 套接口有可能收到比原始 ICMPv4 套接口多得多的分组。然而大多数应用程序只对所有的 ICMP 消息中的一个小子集感兴趣。

为了减少内核通过原始 ICMPv6 套接口向应用进程传递的分组的数量，提供了一个由应用程序指定的过滤器。过滤器用 struct icmp6_filter 数据类型声明，该数据类型由<netinet/icmp6.h>定义。一个原始 ICMPv6 套接口的当前过滤器可用 setsockopt 和 getsock-

opt 来设置和获取，其中 level 参数为 IPPROTO_ICMPV6，optname 参数为 ICMP6_FILTER。

以下是操作 icmp6_filter 结构的 6 个宏：

```
#include <netinet/icmp6.h>

void ICMP6_FILTER_SETPASSALL (struct icmp6_filter * filt);
void ICMP6_FILTER_SETBLOCKALL (struct icmp6_filter * filt);

void ICMP6_FILTER_SETPASS (int msgtype,struct icmp6_filter * filt);
void ICMP6_FILTER_SETBLOCK (int msgtype,struct icmp6_filter * filt);

int ICMP6_FILTER_WILLPASS (int msgtype,const struct icmp6_filter * filt);
int ICMP6_FILTER_WILLBLOCK (int msgtype,const struct icmp6_filter * filt);
                        返回值：如果过滤器传递(阻塞)相应消息类型为 1，否则为 0
```

以上这些宏调用中的 filt 参数是一个指向某个 icmp6_filter 变量的指针，前四个宏修改此 icmp6_filter 变量，后两个宏检查它。msgtype 参数值在 0～255 之间，指定 ICMP 消息类型。

SETPASSALL 宏设定所有消息类型均可传递给应用进程。SETBLOCKALL 宏设定没有消息类型可传递。作为缺省，当一个 ICMPv6 原始套接口创建时，所有 ICMPv6 消息类型可传递给其应用进程。

SETPASS 宏打开某个消息类型向该应用进程的传递，而 SETBLOCK 宏阻塞某个消息类型的传递。如果给定消息类型可以由过滤器传递时，WILLPASS 宏返回 1，否则返回 0。如果给定消息类型被过滤器所阻塞时，WILLBLOCK 宏返回 1，否则返回 0。

作为例子，假想一个只接收 ICMPv6 路由器通告的应用程序：

```
struct icmp6_filter myfilt;
fd = Socket(AF_INET6,SOCK_RAW, IPPROTO_ICMPV6);
ICMP6_FILTER_SETBLOCKALL(&myfilt);
ICMP6_FILTER_SETPASS(ND_ROUTER_ADVERT,&myfilt);
Setsockopt(fd,IPPROTO_ICMPV6,ICMP6_FILTER,&myfilt,sizeof(myfilt));
```

这个例子首先阻塞所有消息类型的传递(既然缺省是传递所有消息类型)，然后只传递路由器通告消息。

25.5　Ping 程序

本章我们将开发一个同时支持 IPv4 和 IPv6 的 Ping 程序。此程序与公开可得版本有两点不同。首先，此程序忽略了公开可得 Ping 程序所支持的大量选项。我们设计这个 Ping 程序的目的在于展示网络编程原理和技术，因此无须为这些选项偏离主要目标。我们这个只支持一个选项的程序，是公开可得程序大小的五分之一。其次，本程序同时支持 IPv6 和 IPv4，而公开可得 Ping 程序只支持 IPv4。

Ping 的操作非常简单，向某些 IP 地址发送一个 ICMP 回射请求，接着该节点返回一个

ICMP 回射应答。这两个 ICMP 消息在 IPv4 和 IPv6 下均得到支持。图 25.1 为 ICMP 消息的格式：

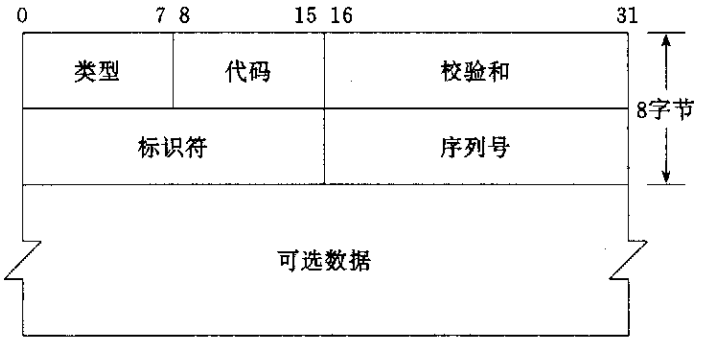


图 25.1 ICMPv4 及 ICMPv6 回射请求和回射应答消息的格式

图 A.15 和图 A.16 给出了这些消息的类型值并指出它们的代码为 0。标识符字段将设为 Ping 程序的进程 ID，序列号随每个分组的发送而增 1。除此之外，我们还将存入一个分组发送时间的 8 字节时间戳作为可选数据。ICMP 规定回射应答需返回标识符、序列号和所有可选数据。存储时间戳使得我们可以在收到应答时计算 RTT。

图 25.2 展示了程序运行的几个例子，第一个使用 IPv4，第二个使用 IPv6。注意一下 # 号提示符，它表示超级用户，因为只有超级用户才有权创建原始套接口。

```
solaris # ping gemini.tuc.noao.edu
PING gemini.tuc.noao.edu (140.252.4.54): 56 data bytes
64 bytes from 140.252.4.54: seq=0, ttl=248, rtt=37.542 ms
64 bytes from 140.252.4.54: seq=1, ttl=248, rtt=34.596 ms
64 bytes from 140.252.4.54: seq=2, ttl=248, rtt=29.204 ms
64 bytes from 140.252.4.54: seq=3, ttl=248, rtt=52.630 ms

solaris # ping 6bone-router.cisco.com
PING 6bone-router.cisco.com (5f00:6d00:c01f:700:1:60:3ell:6700): 56 data bytes
64 bytes from 5f00:6d00:c01f:700:1:60:3ell:6770: seq=0, hlim=255,rtt=116.802 ms
64 bytes from 5f00:6d00:c01f:700:1:60:3ell:6770: seq=1, hlim=255,rtt=129.321 ms
64 bytes from 5f00:6d00:c01f:700:1:60:3ell:6770: seq=2, hlim=255,rtt=109.297 ms
64 bytes from 5f00:6d00:c01f:700:1:60:3ell:6770: seq=3, hlim=255,rtt=78.216 ms
```

图 25.2 Ping 程序的输出示例

图 25.3 为组成整个 Ping 程序的函数的概貌。

程序分两大部分：一部分读取一个原始套接口上收到的所有消息，并输出 ICMP 回射应答，另一部分每隔一秒发送一个回射请求。另一部分由 SIGALRM 信号每秒驱动一次。

图 25.4 给出所有程序都包括的头文件 ping.h。

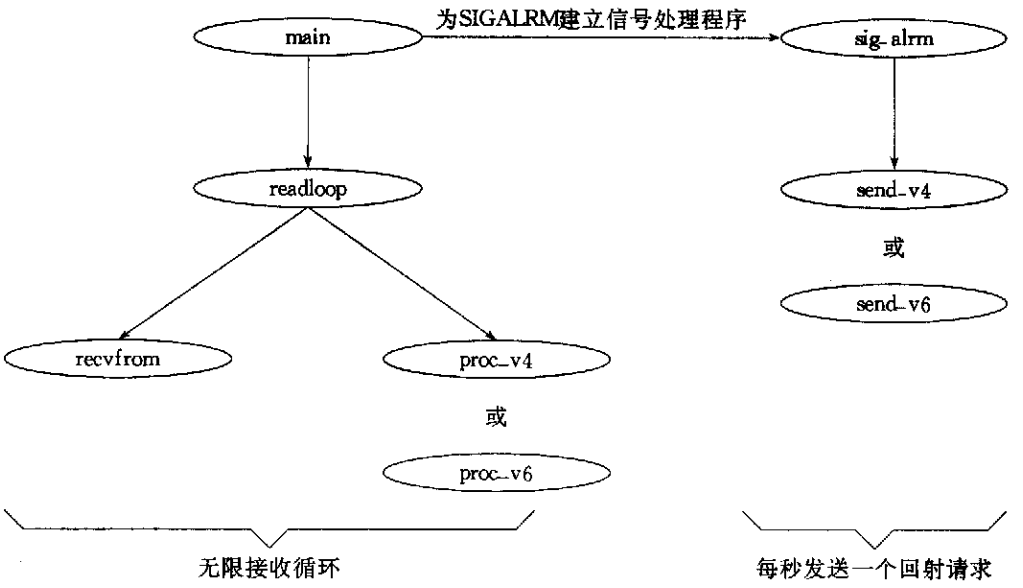


图 25.3 Ping 程序的函数概貌

```
 1 #include    "unp.h"
 2 #include    <netinet/in_systm.h>
 3 #include    <netinet/ip.h>
 4 #include    <netinet/ip_icmp.h>

 5 #define     BUFSIZE         1500

 6               /* globals */
 7 char      recvbuf[BUFSIZE];
 8 char      sendbuf[BUFSIZE];

 9 int       datalen;        /* #bytes of data, following ICMP header */
10 char      *host;
11 int       nsent;          /* add 1 for each sendto() */
12 pid_t     pid;            /* our PID */
13 int       sockfd;
14 int       verbose;

15              /* function prototypes */
16 void      proc_v4(char *, ssize_t, struct timeval *);
17 void      proc_v6(char *, ssize_t, struct timeval *);
18 void      send_v4(void);
19 void      send_v6(void);
20 void      readloop(void);
21 void      sig_alrm(int);
22 void      tv_sub(struct timeval *, struct timeval *);

23 struct proto {
24     void      (*fproc)(char *, ssize_t, struct timeval *);
25     void      (*fsend)(void);
26     struct sockaddr   *sasend;     /* sockaddr{} for send, from getaddrinfo */
27     struct sockaddr   *sarecv;     /* sockaddr{} for receiving */
28     socklen_t      salen;          /* length of sockaddr{}s */
29     int       icmpproto;          /* IPPROTO_xxx value for ICMP */
30 } *pr;
```

```
31 #ifdef    IPV6
32 #include  "ip6.h"          /* should be <netinet/ip6.h> */
33 #include  "icmp6.h"        /* should be <netinet/icmp6.h> */
34 #endif
```

图 25.4 ping.h 头文件[ping/ping.h]

包括 IPv4 和 ICMPv4 头文件

第 1～22 行 包括 IPv4 和 ICMPv4 的基本头文件，定义部分全程变量以及函数原型。

定义 proto 结构

第 23～30 行 我们使用 proto 结构来处理 IPv4 与 IPv6 的差异。此结构包含两个函数指针、两个套接口地址结构指针、这两个套接口地址结构的大小以及 ICMP 的协议值。全程变量指针 pr 将指向为 IPv4 或 IPv6 初始化的某个 proto 结构。

包括 IPv6 和 ICMPv6 头文件

第 31～34 行 我们包括两个定义 IPv6 和 ICMPv6 结构和常值的头文件([Stevens and Thomas 1997])。

```
 1 #include    "ping.h"
 2 struct proto     proto_v4 =
 3 { proc_v4, send_v4, NULL, NULL, 0, IPPROTO_ICMP };
 4 #ifdef    IPV6
 5 struct proto     proto_v6 =
 6 { proc_v6, send_v6, NULL, NULL, 0, IPPROTO_ICMPV6 };
 7 #endif
 8 int    datalen = 56;      /* data that goes with ICMP echo request */
 9 int
10 main(int argc, char **argv)
11 {
12     int      c;
13     struct addrinfo    *ai;
14     opterr = 0;      /* don't want getopt() writing to stderr */
15     while ( (c = getopt(argc, argv, "v")) != -1) {
16         switch (c) {
17         case 'v':
18             verbose++;
19             break;
20         case '?':
21             err_quit("unrecognized option: %c", c);
22         }
23     }
24     if (optind != argc-1)
25         err_quit("usage: ping [ -v ] <hostname>");
26     host = argv[optind];
27     pid = getpid();
28     Signal(SIGALRM, sig_alrm);
29     ai = Host_serv(host, NULL, 0, 0);
30     printf("PING %s (%s): %d data bytes\n", ai->ai_canonname,
31             Sock_ntop_host(ai->ai_addr, ai->ai_addrlen), datalen);
```

```
32          /* initialize according to protocol */
33      if (ai->ai_family == AF_INET) {
34          pr = &proto_v4;
35  #ifdef      IPV6
36      } else if (ai->ai_family == AF_INET6) {
37          pr = &proto_v6;
38          if (IN6_IS_ADDR_V4MAPPED(&(((struct sockaddr_in6 *)
39                                                  ai->ai_addr)->sin6_addr)))
40              err_quit("cannot ping IPv4-mapped IPv6 address");
41  #endif
42      } else
43          err_quit("unknown address family %d", ai->ai_family);
44      pr->sasend = ai->ai_addr;
45      pr->sarecv = Calloc(1, ai->ai_addrlen);
46      pr->salen = ai->ai_addrlen;
47      readloop();
48      exit(0);
49  }
```

图 25.5　main 函数[ping/main.c]

给 IPv4 和 IPv6 定义 proto 结构

第 2～7 行　给 IPv4 和 IPv6 分别定义 proto 结构，由于不知道最终使用的是 IPv4 还是 IPv6，套接口地址结构指针被初始化为 NULL。

可选数据长度

第 8 行　在第 8 行，我们设置随同回射请求一起发送的可选数据长度为 56 个字节，这样，如产生 IPv4 数据报，则总长为 84 个字节(20 字节 IPv4 头部，8 个字节 ICMP 头部)；如产生 IPv6 数据报，则总长 104 字节。回射请求中所包含的任何数据，都要由回射应答返回。在本例中，我们将在回射请求数据区的前 8 个字节内存储该回射请求发出的时间，以便在接收到回射应答后使用它来计算和输出 RTT。

处理命令行参数

第 14～28 行　本程序唯一的命令行选项是 -v，它决定是否输出收到的大多数 ICMP 消息(当然，我们不输出属于运行中的另一个 Ping 进程的回射应答)。此外，第 28 行还为 SIGALRM 信号建立了一个处理程序，我们将看到该信号每秒产生一次，并导致一个 ICMP 回射请求的发出。

处理主机名参数

第 29～46 行　在命令行参数中必须有一个主机名或主机 IP 地址。这里，我们调用 host_serv 函数来处理它，返回的 addrinfo 结构含有地址族：或者是 AF_INET，或者是 AF_INET6。接着，我们将全局变量 pr 初始化为正确的 proto 结构。我们还要通过调用 IN6_IS_ADDR_V4MAPPED 来确认一个 IPv6 地址不是一个 IPv4 映射的 IPv6 地址，这是因为即使由 host_serv 返回的的地址是一个 IPv6 地址，发送给其主机的仍然是 IPv4 地址。(这种情况下我们可以改用 IPv4 地址。)已由 getaddrinfo 函数分配的套接口地址结构将用来发送，另一个同样大小的套接口地址结构分配用于接收。

第 47 行　由 readloop 处理接收分组事宜，见图 25.6。

```
1 #include    "ping.h"
2 void
3 readloop(void)
4 {
5     int                 size;
6     char            recvbuf[BUFSIZE];
7     socklen_t   len;
8     ssize_t     n;
9     struct timeval   tval;
10    sockfd = Socket(pr->sasend->sa_family, SOCK_RAW, pr->icmpproto);
11    setuid(getuid());       /* don't need special permissions any more */
12    size = 60 * 1024;       /* OK if setsockopt fails */
13    setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
14    sig_alrm(SIGALRM);          /* send first packet */
15    for ( ; ; ) {
16        len = pr->salen;
17        n = recvfrom(sockfd, recvbuf, sizeof(recvbuf), 0, pr->sarecv, &len);
18        if (n < 0) {
19            if (errno == EINTR)
20                continue;
21            else
22                err_sys("recvfrom error");
23        }
24        Gettimeofday(&tval, NULL);
25        (*pr->fproc)(recvbuf, n, &tval);
26    }
27 }
```

图 25.6 readloop 函数[ping/readloop.c]

创建套接口

第10～11行 第10行创建一个拥有合适协议的原始套接口。第11行的setuid主要为安全性考虑。将进程的有效用户ID设成进程的实际用户ID。我们知道，只有超级用户才能创建原始套接口。因此，这个ping程序通常属于root，并且有SUID的属性。现在套接口已经建成，我们可以放弃额外的特权。在额外的特权不再需要时放弃它，总是这类程序最好的处理SUID属性的方式。这可以防止某些人利用程序中潜在的错误来攻击系统。

设置套接口接收缓冲区的大小

第12～13行 我们将套接口接收缓冲区设为61,440字节(60×1024)，这要比缺省值大得多。这样做主要是为了减少接收缓冲区溢出的可能性，因为，如果我们无意中ping一个IPv4广播地址或一个多播地址，将会引发大量的应答。

发送第一个分组

第14行 我们调用信号处理程序来发送一个分组，并调度下一次SIGALRM为一秒之后。直接调用一个信号处理程序并不常见；不过也没什么，虽然它们一般由系统内核调用，但终归也是一个C语言函数。

读入所有ICMP消息的无限循环

第15～26行 程序的主循环是一个无限循环，它读取返回给ICMP原始套接口的每个

分组，调用 gettimeofday 来记录分组收到的时间，并调用恰当的协议函数(proc_v4 或 proc_v6)来处理这些 ICMP 消息。

图 25.8 给出了 proc_v4 函数，用于处理所有收到的 ICMPv4 消息。如果想了解 IPv4 头部的格式，见图 A.1。顺便提一句，当进程在原始套接口上收到 CMPv4 消息时，内核已经检查过 IPv4 头部及 ICMPv4 头部中各个基本字段的有效性了(TCPv2 第 214～311 页)。

获取 ICMP 头部指针

第 10～14 行　IPv4 头部长度字段是以 4 字节为一个单位计数的，将它乘以 4 才是以字节为单位的头部长度。(记住 IPv4 头部可以包含选项。)这使我们可以正确设置 icmp 以指向 ICMP 头部的起始地址。图 25.9 标出了本段代码所用的各种头部、指针及长度。

检查 ICMP 回射应答

第 15～19 行　如果所收到的消息是一个 ICMP 回射应答，那么我们必须检查标识符字段，看看这个应答是不是对我们这个进程发出的请求的响应。如果在这个主机上进程被运行了多次，每个进程都将获得本主机所有接收到的 ICMP 消息的一份拷贝。

第 20～25 行　通过将消息发出时间(包含在 ICMP 应答的可选数据部分内)与当前时间(由函数参数 tvrecv 给出)相减，我们可以计算出 RTT。RTT 从微秒转换成毫秒并输出，同时输出的还有序列号和收到的 TTL。序列号用来检查分组是否丢失、被重新排序或重复，TTL 用来指示收发主机间的跳数。

输出所有收到的 ICMP 消息

第 26～30 行　如果用户指定了 -v(详尽输出)命令行选项，这段程序将输出所收到的所有其他 ICMP 消息的类型字段和代码字段。

图 25.7 为 tv_sub 函数，用于将两 timeval 结构相减，并将结果存入第一个 timeval 结构中。

```
 1 #include    "unp.h"
 2 void
 3 tv_sub(struct timeval *out, struct timeval *in)
 4 {
 5     if ( (out->tv_usec -= in->tv_usec) < 0) { /* out -= in */
 6         --out->tv_sec;
 7         out->tv_usec += 1000000;
 8     }
 9     out->tv_sec -= in->tv_sec;
10 }
```

图 25.7　tv_sub 函数：将两个 timeval 结构相减[lib/tv_sub.c]

```
 1 #include    "ping.h"
 2 void
 3 proc_v4(char *ptr, ssize_t len, struct timeval *tvrecv)
 4 {
 5     int      hlen1, icmplen;
 6     double   rtt;
 7     struct ip *ip;
 8     struct icmp *icmp;
 9     struct timeval *tvsend;
```

```
10      ip = (struct ip *) ptr;          /* start of IP header */
11      hlen1 = ip->ip_hl << 2;          /* length of IP header */
12      icmp = (struct icmp *) (ptr + hlen1);      /* start of ICMP header */
13      if ( (icmplen = len - hlen1) < 8)
14          err_quit("icmplen (%d) < 8", icmplen);
15      if (icmp->icmp_type == ICMP_ECHOREPLY) {
16          if (icmp->icmp_id != pid)
17              return;          /* not a response to our ECHO_REQUEST */
18          if (icmplen < 16)
19              err_quit("icmplen (%d) < 16", icmplen);
20          tvsend = (struct timeval *) icmp->icmp_data;
21          tv_sub(tvrecv, tvsend);
22          rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;
23          printf("%d bytes from %s: seq=%u, ttl=%d, rtt=%.3f ms\n",
24                  icmplen, Sock_ntop_host(pr->sarecv, pr->salen),
25                  icmp->icmp_seq, ip->ip_ttl, rtt);
26      } else if (verbose) {
27          printf("  %d bytes from %s: type = %d, code = %d\n",
28                  icmplen, Sock_ntop_host(pr->sarecv, pr->salen),
29                  icmp->icmp_type, icmp->icmp_code);
30      }
31  }
```

图 25.8 proc_v4 函数：处理 ICMPv4 消息[ping/proc_v4.c]

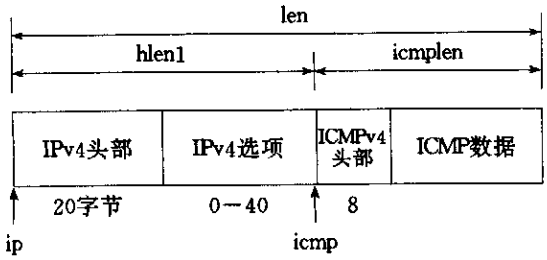


图 25.9 ICMPv4 应答结构：头部、指针及长度

ICMPv6 消息由函数 proc_v6 处理，见图 25.10，它与 proc_v4 函数很相近。

```
1  #include    "ping.h"
2  void
3  proc_v6(char *ptr, ssize_t len, struct timeval *tvrecv)
4  {
5  #ifdef  IPV6
6      int          hlen1, icmp6len;
7      double       rtt;
8      struct ip6_hdr  *ip6;
9      struct icmp6_hdr  *icmp6;
10     struct timeval  *tvsend;
11     ip6 = (struct ip6_hdr *) ptr;      /* start of IPv6 header */
12     hlen1 = sizeof(struct ip6_hdr);
13     if (ip6->ip6_nxt != IPPROTO_ICMPV6)
14         err_quit("next header not IPPROTO_ICMPV6");
15     icmp6 = (struct icmp6_hdr *) (ptr + hlen1);
```

```
16      if ( (icmp6len = len - hlen1) < 8)
17          err_quit("icmp6len (%d) < 8", icmp6len);
18      if (icmp6->icmp6_type == ICMP6_ECHO_REPLY) {
19          if (icmp6->icmp6_id != pid)
20              return;         /* not a response to our ECHO_REQUEST */
21          if (icmp6len < 16)
22              err_quit("icmp6len (%d) < 16", icmp6len);
23          tvsend = (struct timeval *) (icmp6 + 1);
24          tv_sub(tvrecv, tvsend);
25          rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;
26          printf("%d bytes from %s: seq=%u, hlim=%d, rtt=%.3f ms\n",
27                  icmp6len, Sock_ntop_host(pr->sarecv, pr->salen),
28                  icmp6->icmp6_seq, ip6->ip6_hlim, rtt);
29      } else if (verbose) {
30          printf("  %d bytes from %s: type = %d, code = %d\n",
31                  icmp6len, Sock_ntop_host(pr->sarecv, pr->salen),
32                  icmp6->icmp6_type, icmp6->icmp6_code);
33      }
34  #endif      /* IPV6 */
35  }
```

图 25.10　proc_v6 函数：处理收到的 ICMPv6 消息[ping/proc_v6.c]

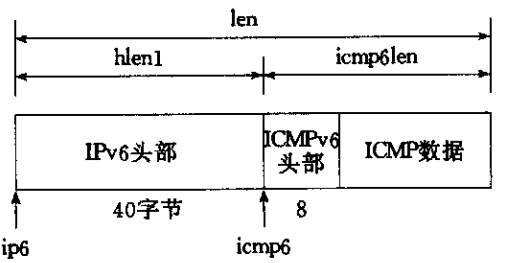


图 25.11　ICMPv6 应答结构：头部、指针及长度

获取 ICMPv6 头部的指针

第 11～17 行　IPv6 头部长度固定为 40 字节，接下来是 ICMPv6 头部(回忆一下，扩展头部永远不会作为普通数据返回，而是作为附加数据)。图 25.11 展示了代码使用的各种头部、指针和长度。

检查 ICMP 回射应答

第 18～28 行　如果 ICMP 消息类型为回射应答，则检查标识符字段，看是不是给我们的应答。如果一切正常，则计算 RTT，并和序列号及 IPv6 跳限一起输出。

输出所有其他 ICMP 消息

第 29～33 行　如果用户指定了-v(详尽输出)命令行选项，则输出接收到的所有其他 ICMP 消息的类型和代码字段。

> 如果不指定-v 选项，我们可以建立 ICMP 过滤器(ICMP6_FILTER 套接口选项见 25.4 节)，使得内核在我们的套接口上只返回回射应答。

我们的 SIGALRM 信号处理函数是 sig_alrm 函数，见图 25.12。在图 25.6 的 readloop

函数中，我们就在开头调用过它一次，发出第一个分组。此函数仅调用协议相关的函数来发送一个 ICMP 回射请求(send_v4 或 send_v6)，然后调度下一次 SIGALRM 在一秒后产生。

```
1 #include    "ping.h"
2 void
3 sig_alrm(int signo)
4 {
5     (*pr->fsend)();
6     alarm(1);
7     return;        /* probably interrupts recvfrom() */
8 }
```

图 25.12 sig_alrm 函数：SIGALRM 信号处理程序[ping/sig_alrm.c]

函数 send_v4(图 25.13)构造一个 ICMPv4 回射请求消息并写入原始套接口。

构造 ICMPv4 消息

第 7～12 行 构造 ICMPv4 消息，用进程 ID 设置标识符字段，用全程变量 nsent 设置序列号，并为下一个分组将 nsent 增 1。当前时间存入 ICMP 消息的数据部分。

计算校验和

第 13～15 行 为了计算 ICMP 校验和，我们先设置校验和字段为 0，然后调用函数 in_cksum 来计算校验和，将结果存入校验和字段。ICMPv4 校验和的计算涉及 ICMPv4 头部及其后的所有数据。

```
1 #include    "ping.h"
2 void
3 send_v4(void)
4 {
5     int            len;
6     struct icmp *icmp;
7     icmp = (struct icmp *) sendbuf;
8     icmp->icmp_type = ICMP_ECHO;
9     icmp->icmp_code = 0;
10    icmp->icmp_id = pid;
11    icmp->icmp_seq = nsent++;
12    Gettimeofday((struct timeval *) icmp->icmp_data, NULL);
13    len = 8 + datalen;          /* checksum ICMP header and data */
14    icmp->icmp_cksum = 0;
15    icmp->icmp_cksum = in_cksum((u_short *) icmp, len);
16    Sendto(sockfd, sendbuf, len, 0, pr->sasend, pr->salen);
17 }
```

图 25.13 send_v4 函数构造并发送 ICMPv4 回射请求消息[ping/send_v4.c]

发送数据报

第 16 行 ICMP 消息通过原始套接口发送。由于我们没有使用 IP_HDRINCL，内核将为我们构造 IPv4 分组的头部并将其安在我们的缓冲区前。

网际校验和是被校验数据 16 位值的反码和(ones-complement sum)。如果数据字节长度为奇数，则尾部补一个字节的 0 以凑成偶数。此算法适用于 IPv4、ICMPv4、IGMPv4、ICMPv6、UDP 和 TCP 校验和。更详细的信息和几个例子请见 RFC 1071[Braden，Borman，and

Partridge 1988]。TCPv2 的 8.7 节详细讨论了这个算法，并给出一个高效的实现。图 25.14 中的 in_cksum 函数也是一个实现。

网际校验和生成算法

第 1～27 行　第一个 while 循环计算所有 16 位值之和，如果长度为奇数，则最后一个字节加入和中。图 25.14 的算法是个简单的算法，对 Ping 程序而言还不错，但对内核使用的大数据量的校验和计算不太适用。

这个函数是从公开可得版本 Ping 程序中获取的。

```
1 unsigned short
2 in_cksum(unsigned short *addr, int len)
3 {
4     int         nleft = len;
5     int         sum = 0;
6     unsigned short *w = addr;
7     unsigned short answer = 0;
8     /*
9      * Our algorithm is simple, using a 32 bit accumulator (sum), we add
10     * sequential 16 bit words to it, and at the end, fold back all the
11     * carry bits from the top 16 bits into the lower 16 bits.
12     */
13    while (nleft > 1)  {
14        sum += *w++;
15        nleft -= 2;
16    }
17        /* mop up an odd byte, if necessary */
18    if (nleft == 1) {
19        *(unsigned char *)(&answer) = *(unsigned char *)w ;
20        sum += answer;
21    }
22        /* add back carry outs from top 16 bits to low 16 bits */
23    sum = (sum >> 16) + (sum & 0xffff);     /* add hi 16 to low 16 */
24    sum += (sum >> 16);             /* add carry */
25    answer = ~sum;                  /* truncate to 16 bits */
26    return(answer);
27 }
```

图 25.14　in_cksum 函数：计算网际校验和[libfree/in_cksum.c]

我们这个 Ping 程序的最后一个函数是 send_v6，见图 25.15，它构造并发送 ICMPv6 回射请求。

send_v6 函数与 send_v4 很相像，不过要注意，它不计算 ICMPv6 校验和。在本章早些时候我们提过，ICMPv6 校验和的计算要涉及 IPv6 头部中的源地址，因此，该校验和在内核选择源地址后，由内核来计算并设置。

```
1 #include    "ping.h"
2 void
3 send_v6()
4 {
5 #ifdef    IPV6
6     int         len;
```

```
 7     struct icmp6_hdr  *icmp6;
 8     icmp6 = (struct icmp6_hdr *) sendbuf;
 9     icmp6->icmp6_type = ICMP6_ECHO_REQUEST;
10     icmp6->icmp6_code = 0;
11     icmp6->icmp6_id = pid;
12     icmp6->icmp6_seq = nsent++;
13     Gettimeofday((struct timeval *) (icmp6 + 1), NULL);
14     len = 8 + datalen;          /* 8-byte ICMPv6 header */
15     Sendto(sockfd, sendbuf, len, 0, pr->sasend, pr->salen);
16         /* kernel calculates and stores checksum for us */
17 #endif      /* IPV6 */
18 }
```

图 25.15 send_v6 函数:构造并发送 ICMPv6 回射请求消息[ping/send_v6.c]

25.6 Traceroute 程序

本节我们将开发一个自己的 Traceroute 程序。与上一节开发的 Ping 程序一样,我们开发自己的版本,而不是给出公开可得版本。这么做,一是由于我们要同时支持 IPv4 和 IPv6,二是因为我们不想与大量和网络编程无关的选项处理打太多的交道。

Traceroute 的用途是确定从我们的主机到目的地之间 IP 数据报行进的路径。它的操作比较简单,TCPv1 的第 8 章对此进行了详细的解释,并给了几个应用实例。Traceroute 使用 IPv4 的 TTL 字段或 IPv6 的跳限字段以及两个 ICMP 消息。一开始,它向目的主机发送一个 TTL(或跳限)为 1 的 UDP 数据报。这个数据报导致头一跳的路由器返回一个"time exceeded in transmit(传输中超时)"错。接着每次对 TTL 加 1,并分别发出 UDP 数据报,这样可以一次一次确定下一个路由器。当 UDP 数据报终于到达目的主机时,希望目的主机能返回一个"port unreachable(端口不可达)"错,这通过向一个期望未被使用的随机端口发送 UDP 数据报来实现。

以前版本的 Traceroute 为了设置 IPv4 的 TTL 字段,只能先设置 IP_HDRINCL 套接口选项,然后直接构造自己的 IPv4 头部。如今的系统提供 IP_TTL 套接口选项,通过该选项就可以置数据报的 TTL。(该选项由 4.3BSD Reno 版本引入)。设置该选项要比构造整个 IPv4 头部容易得多(虽然我们在 26.6 节将会说明如何构造自己的 IPv4 头部及 UDP 头部)。IPv6 的 IPV6_UNICAST_HOPS 套接口选项也提供对 IPv6 数据报跳限字段的控制。

图 25.16 是程序将要使用的 trace.h 头文件

```
1 #include    "unp.h"
2 #include    <netinet/in_systm.h>
3 #include    <netinet/ip.h>
4 #include    <netinet/ip_icmp.h>
5 #include    <netinet/udp.h>
6 #define     BUFSIZE     1500
7 struct rec {                        /* format of outgoing UDP data */
8     u_short   rec_seq;              /* sequence number */
9     u_short   rec_ttl;              /* TTL packet left with */
10    struct timeval  rec_tv;         /* time packet left */
11 };
```

```
12              /* globals */
13 char          recvbuf[BUFSIZE];
14 char          sendbuf[BUFSIZE];
15 int           datalen;            /* #bytes of data, following ICMP header */
16 char          *host;
17 u_short      sport, dport;
18 int           nsent;              /* add 1 for each sendto() */
19 pid_t        pid;                /* our PID */
20 int           probe, nprobes;
21 int           sendfd, recvfd;     /* send on UDP sock, read on raw ICMP sock */
22 int           ttl, max_ttl;
23 int           verbose;
24              /* function prototypes */
25 char      *icmpcode_v4(int);
26 char      *icmpcode_v6(int);
27 int       recv_v4(int, struct timeval *);
28 int       recv_v6(int, struct timeval *);
29 void      sig_alrm(int);
30 void      traceloop(void);
31 void      tv_sub(struct timeval *, struct timeval *);
32 struct proto {
33     char      *(*icmpcode)(int);
34     int   (*recv)(int, struct timeval *);
35     struct sockaddr  *sasend;     /* sockaddr{} for send, from getaddrinfo */
36     struct sockaddr  *sarecv;     /* sockaddr{} for receiving */
37     struct sockaddr  *salast;     /* last sockaddr{} for receiving */
38     struct sockaddr  *sabind;     /* sockaddr{} for binding source port */
39     socklen_t         salen;      /* length of sockaddr{}s */
40     int      icmpproto;           /* IPPROTO_xxx value for ICMP */
41     int      ttllevel;            /* setsockopt() level to set TTL */
42     int      ttloptname;          /* setsockopt() name to set TTL */
43 } *pr;
44 #ifdef   IPV6
45 #include   "ip6.h"               /* should be <netinet/ip6.h> */
46 #include   "icmp6.h"             /* should be <netinet/icmp6.h> */
47 #endif
```

图 25.16　trace.h 头文件[traceroute/trace.h]

第 1～11 行　我们包括标准 IPv4 头文件，该头文件定义了 IPv4、ICMPv4 和 UDP 的结构及常值。rec 结构用于定义我们将要发送的 UDP 数据报的数据部分。不过我们将发现，其实我们并不需要检查这些数据，这是用于调试目的的。

定义 proto 结构

第 32～43 行　与前一节的 Ping 程序一样，我们通过定义一个 proto 结构来处理 IPv4 和 IPv6 两协议之间的不同，该结构含有函数指针、指向套接口地址结构的指针及其他在两个 IP 版本之间有区别的常值。当 main 函数处理完目的地址以后，全程变量 pr 指针指向某个初始化为 IPv4 或 IPv6 类型的 proto 结构(一旦目的地址确定，使用 IPv4 还是 IPv6 也就确定了)。

包括 IPv6 头文件

第 44～47 行 包括定义了 IPv6 及 ICMPv6 结构及常值的头文件。

图 25.17 为 main 函数。它处理命令行参数，给 IPv4 或 IPv6 初始化 pr 指针，并调用 traceloop 函数。

```
 1 #include    "trace.h"
 2 struct proto     proto_v4 =
 3 { icmpcode_v4, recv_v4, NULL, NULL, NULL, NULL, 0,
 4 IPPROTO_ICMP, IPPROTO_IP, IP_TTL };
 5 #ifdef    IPV6
 6 struct proto     proto_v6 =
 7 { icmpcode_v6, recv_v6, NULL, NULL, NULL, NULL, 0,
 8 IPPROTO_ICMPV6, IPPROTO_IPV6, IPV6_UNICAST_HOPS };
 9 #endif
10 int     datalen = sizeof(struct rec); /* defaults */
11 int     max_ttl = 30;
12 int     nprobes = 3;
13 u_short dport = 32768 + 666;
14 int
15 main(int argc, char **argv)
16 {
17     int     c;
18     struct addrinfo *ai;
19     opterr = 0;          /* don't want getopt() writing to stderr */
20     while ( (c = getopt(argc, argv, "m:v")) != -1) {
21         switch (c) {
22         case 'm':
23             if ( (max_ttl = atoi(optarg)) <= 1)
24                 err_quit("invalid -m value");
25             break;
26         case 'v':
27             verbose++;
28             break;
29         case '?':
30             err_quit("unrecognized option: %c", c);
31         }
32     }
33     if (optind != argc-1)
34         err_quit("usage: traceroute [ -m <maxttl> -v ] <hostname>");
35     host = argv[optind];
36     pid = getpid();
37     Signal(SIGALRM, sig_alrm);
38     ai = Host_serv(host, NULL, 0, 0);
39     printf("traceroute to %s (%s): %d hops max, %d data bytes\n",
40             ai->ai_canonname,
41             Sock_ntop_host(ai->ai_addr, ai->ai_addrlen),
42             max_ttl, datalen);
43         /* initialize according to protocol */
44     if (ai->ai_family == AF_INET) {
45         pr = &proto_v4;
46 #ifdef    IPV6
```

```
47      } else if (ai->ai_family == AF_INET6) {
48          pr = &proto_v6;
49          if (IN6_IS_ADDR_V4MAPPED(&(((struct sockaddr_in6 *)ai->ai_addr)->
            sin6_addr)))
50              err_quit("cannot traceroute IPv4-mapped IPv6 address");
51  #endif
52      } else
53          err_quit("unknown address family %d", ai->ai_family);
54      pr->sasend = ai->ai_addr;           /* contains destination address */
55      pr->sarecv = Calloc(1, ai->ai_addrlen);
56      pr->salast = Calloc(1, ai->ai_addrlen);
57      pr->sabind = Calloc(1, ai->ai_addrlen);
58      pr->salen = ai->ai_addrlen;
59      traceloop();
60      exit(0);
61  }
```

图 25.17　Traceroute 程序的 main 函数[traceroute/main.c]

定义 proto 结构

第 2～9 行　我们给 IPv4 和 IPv6 分别定义一个 proto 结构，不过直到函数尾，才为套接口地址结构的指针分配地址。

设置缺省值

第 10～13 行　最大 TTL 或跳限缺省设为 30，不过 -m 命令行参数可让用户修改此值。对于每一个 TTL 值，我们将发三个控测分组，同样，可通过命令行参数修改此值。目的端口缺省设为 32768+666，且每发一个 UDP 数据报就增 1。我们希望当数据报最终到达目的主机时，相应端口是未被使用的。当然，这一点无法保证。

处理命令行参数

第 19～37 行　-v 参数使程序输出收到的大多数 ICMP 消息。

处理主机名或 IP 地址参数并结束初始化

第 38～58 行　使用 host_serv 函数处理目的主机名或 IP 地址。该函数返回一个指向 addrinfo 结构的指针。根据返回地址类型的不同(IPv4 或 IPv6)，初始化 proto 结构。将该结构的指针存入全程变量 pr。

使用适当的大小，分配套接口地址结构

第 59 行　图 25.18 中的 traceloop 函数发送数据报并读取返回的 ICMP 消息。这是程序的主循环。

```
1  #include    "trace.h"
2  void
3  traceloop(void)
4  {
5      int          seq, code, done;
6      double       rtt;
7      struct rec  *rec;
8      struct timeval tvrecv;
9      recvfd = Socket(pr->sasend->sa_family, SOCK_RAW, pr->icmpproto);
```

```
10      setuid(getuid());      /* don't need special permissions any more */
11      sendfd = Socket(pr->sasend->sa_family, SOCK_DGRAM, 0);
12      pr->sabind->sa_family = pr->sasend->sa_family;
13      sport = (getpid() & 0xffff) | 0x8000;      /* our source UDP port# */
14      sock_set_port(pr->sabind, pr->salen, htons(sport));
15      Bind(sendfd, pr->sabind, pr->salen);
16      sig_alrm(SIGALRM);
17      seq = 0;
18      done = 0;
19      for (ttl = 1; ttl <= max_ttl && done == 0; ttl++) {
20          Setsockopt(sendfd, pr->ttllevel, pr->ttloptname, &ttl, sizeof(int));
21          bzero(pr->salast, pr->salen);
22          printf("%2d  ", ttl);
23          fflush(stdout);
24          for (probe = 0; probe < nprobes; probe++) {
25              rec = (struct rec *) sendbuf;
26              rec->rec_seq = ++seq;
27              rec->rec_ttl = ttl;
28              Gettimeofday(&rec->rec_tv, NULL);
29              sock_set_port(pr->sasend, pr->salen, htons(dport + seq));
30              Sendto(sendfd, sendbuf, datalen, 0, pr->sasend, pr->salen);
31              if ( (code = (*pr->recv)(seq, &tvrecv)) == -3)
32                  printf(" *");          /* timeout, no reply */
33              else {
34                  char   str[NI_MAXHOST];
35                  if (sock_cmp_addr(pr->sarecv, pr->salast, pr->salen) != 0) {
36                      if (getnameinfo(pr->sarecv, pr->salen, str, sizeof(str),
37                                      NULL, 0, 0) == 0)
38                          printf(" %s (%s)", str,
39                                  Sock_ntop_host(pr->sarecv, pr->salen));
40                      else
41                          printf(" %s",
42                                  Sock_ntop_host(pr->sarecv, pr->salen));
43                      memcpy(pr->salast, pr->sarecv, pr->salen);
44                  }
45                  tv_sub(&tvrecv, &rec->rec_tv);
46                  rtt = tvrecv.tv_sec * 1000.0 + tvrecv.tv_usec / 1000.0;
47                  printf("  %.3f ms", rtt);
48                  if (code == -1)          /* port unreachable; at destination */
49                      done++;
50                  else if (code >= 0)
51                      printf(" (ICMP %s)", (*pr->icmpcode)(code));
52              }
53              fflush(stdout);
54          }
55          printf("\n");
56      }
57  }
```

图 25.18 traceloop 函数：主处理循环[traceroute/traceloop.c]

创建两个套接口

第 9～11 行 我们需要两个套接口：一个原始套接口，用于读取所有返回的 ICMP 消

息;一个 UDP 套接口,用于发送 TTL 不断递增的探测分组。在创建原始套接口之后,我们将有效用户 ID 重置为实际用户 ID,此时我们已不需要超级用户权限了。

给 UDP 套接口捆绑源端口

第 12～15 行　由于用户可能在同一时间运行两个 traceroute,对于收到的 ICMP 消息,必须判别是不是本程序所发数据报引起的。为此,在为发送 UDP 分组的套接口捆绑源端口时,我们使用本进程的低序 16 位作为端口号,但是高序位总是置为 1。我们在 UDP 头部中使用源端口号来标识发送进程,因为返回的 ICMP 消息总是包含引起 ICMP 错误的数据报的 UDP 头部。

建立 SIGALRM 信号处理程序

第 16 行　对于 SIGALRM 信号建立信号处理程序 sig_alrm。每发出一个 UDP 数据报后,在发送下一个探测分组以前,我们将等待 ICMP 消息 3 秒。

主循环:设置 TTL 或跳限,发送三个探测分组

第 17～28 行　函数的主循环是一个双层嵌套的 for 循环。外循环从 TTL(或跳限)等于 1 开始,每次加 1。内循环为每个 TTL 向目的主机发送 3 个探测分组(UDP 数据报)。每当 TTL 变化时,我们都用 IP_TTL 或 IPV6_UNICAST_HOPS 参数调用 setsockopt 设置新值。

在外部循环的每一轮中,我们都要将由 salast 指向的套接口地址结构初始化成 0。这个结构将与 recvfrom 读取 ICMP 消息时返回的套接口地址结构相比较,如果两者不同,则输出新结构的 IP 地址。使用这种技术,对于每个 TTL 都可以输出第一个探测分组获得的 IP 地址。如果同一个 TTL 对应的 IP 地址发生变动(譬如说程序运行中,所路由的路径可能发生变化),则新的 IP 地址也输出。

设置目的端口,发送 UDP 数据报

第 29～30 行　每次发送探测分组时,我们都要调用 soct_set_port 函数来修改 sasend 套接口地址结构中的目的端口号。理由是在探测分组到达目的主机时,我们希望发送给一个未被使用的端口。而发送三个端口不同的探测分组,则至少有一个端口未被使用的机会就大大增加了。sendto 发送 UDP 数据报。

读 ICMP 消息

第 31～53 行　使用 recv_v4 或 recv_v6 函数之一调用 recvfrom 来读取并处理返回的 ICMP 消息。这两个函数如果超时则返回-3(通知我们,如果对当前 TTL 没有发够三个探测分组,就发送下一个探测分组)。如果收到 ICMP"time exceeded in transmit(传输中超时)"错则返回-2,如果收到 ICMP"port unreachable(端口不可达)"错则返回-1(意味着已到达目的主机)。如果收到其他 ICMP 目的地不可达错误则返回一个非负数。

输出应答

第 33～53 行　如前所述,对于所收到的 ICMP 消息,如果它是给定 TTL 的第一个应答,或者就本 TTL 而言,发送 ICMP 消息的节点其 IP 地址已发生变化,那么我们将输出发送应答主机的主机名和 IP 地址(如果 getnameinfo 无法返回主机名则只输出 IP 地址)。同时根据发出探测分组的时间和收到 ICMP 消息的时间计算 RTT 并输出。

图 25.19 为 recv_v4 函数。

```
 1 #include    "trace.h"
 2 /*
 3  * Return: -3 on timeout
 4  *         -2 on ICMP time exceeded in transit (caller keeps going)
 5  *         -1 on ICMP port unreachable (caller is done)
 6  *         >= 0 return value is some other ICMP unreachable code
 7  */
 8 int
 9 recv_v4(int seq, struct timeval *tv)
10 {
11     int     hlen1, hlen2, icmplen;
12     socklen_t len;
13     ssize_t n;
14     struct ip *ip, *hip;
15     struct icmp *icmp;
16     struct udphdr *udp;
17     alarm(3);
18     for ( ; ; ) {
19         len = pr->salen;
20         n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0, pr->sarecv, &len);
21         if (n < 0) {
22             if (errno == EINTR)
23                 return(-3);      /* alarm expired */
24             else
25                 err_sys("recvfrom error");
26         }
27         Gettimeofday(tv, NULL);      /* get time of packet arrival */
28         ip = (struct ip *) recvbuf;    /* start of IP header */
29         hlen1 = ip->ip_hl << 2;      /* length of IP header */
30         icmp = (struct icmp *) (recvbuf + hlen1); /* start of ICMP header */
31         if ( (icmplen = n - hlen1) < 8)
32             err_quit("icmplen (%d) < 8", icmplen);
33         if (icmp->icmp_type == ICMP_TIMXCEED &&
34             icmp->icmp_code == ICMP_TIMXCEED_INTRANS) {
35             if (icmplen < 8 + 20 + 8)
36                 err_quit("icmplen (%d) < 8 + 20 + 8", icmplen);
37             hip = (struct ip *) (recvbuf + hlen1 + 8);
38             hlen2 = hip->ip_hl << 2;
39             udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);
40             if (hip->ip_p == IPPROTO_UDP &&
41                 udp->uh_sport == htons(sport) &&
42                 udp->uh_dport == htons(dport + seq))
43                 return(-2);      /* we hit an intermediate router */
44         } else if (icmp->icmp_type == ICMP_UNREACH) {
45             if (icmplen < 8 + 20 + 8)
46                 err_quit("icmplen (%d) < 8 + 20 + 8", icmplen);
47             hip = (struct ip *) (recvbuf + hlen1 + 8);
48             hlen2 = hip->ip_hl << 2;
49             udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);
50             if (hip->ip_p == IPPROTO_UDP &&
51                 udp->uh_sport == htons(sport) &&
```

```
52                  udp->uh_dport == htons(dport + seq)) {
53                  if (icmp->icmp_code == ICMP_UNREACH_PORT)
54                      return(-1);  /* have reached destination */
55                  else
56                      return(icmp->icmp_code);     /* 0, 1, 2, ... */
57              }
58          } else if (verbose) {
59              printf(" (from %s: type = %d, code = %d)\n",
60                      Sock_ntop_host(pr->sarecv, pr->salen),
61                      icmp->icmp_type, icmp->icmp_code);
62          }
63          /* Some other ICMP error, recvfrom() again */
64      }
65  }
```

图 25.19　recv_v4 函数：读取并处理 ICMPv4 消息[traceroute/recv_v4.c]

设置报警时钟(alarm)并读取每个 ICMP 消息

第 17～27 行　设定一个 3 秒的报警时钟，并进入一个调用 recvfrom 的循环，读取所有返回到原始套接口的 ICMPv4 消息。

该函数同样面临 18.5 节讨论过的竞争状态：SIGALRM 中断读操作。

获取 ICMP 头部指针

第 28～32 行　ip 指向 IPv4 头部的开始处(回忆一下，原始套接口上的读操作总是返回 IP 头部)。icmp 指向 ICMP 头部的开始处。图 25.20 给出了本段代码所用的各种头部、指针和长度的图示。

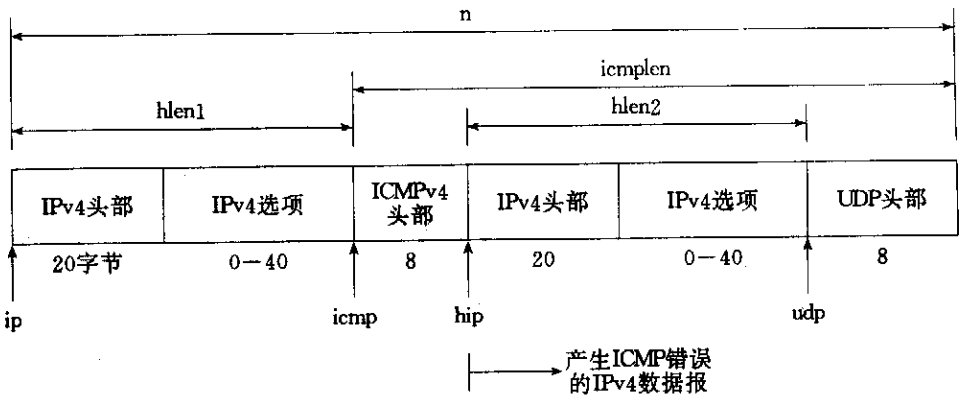


图 25.20　ICMPv4 出错消息的处理：头部、指针和长度

处理 ICMP 传输中超时错

第 33～43 行　如果返回的 ICMP 消息是“time exceeded in transmit”，那么很可能是我们的探测分组引起的。hip 指向 ICMP 消息中返回的 IPv4 头部，跟在 8 个字节的 ICMP 头部后面。udp 指向其后的 UDP 头部。如果 ICMP 消息的确是由一个 UDP 数据报引起的，且源和目的端口与我们发送时记下的值一样，那么这的确是某个中间路由器对我们的探测分组的应答。

处理ICMP端口不可达错

第44～57行 如果返回的ICMP消息为"destination unreachable",那么我们检查一下UDP头部,看一下该ICMP消息是不是由我们的探测分组引起的。如果是,且ICMP代码为端口不可达,则返回－1表示我们到达了目的主机。如果ICMP消息的确是由我们的探测分组引起的,但不是端口不可达,那么直接返回ICMP代码值。这种情况的常见例子是由防火墙为我们探测的目的主机返回除端口不可达以外的其他某个不可达ICMP代码。

处理其他ICMP消息

第58～62行 如果指定了-v参数,则输出其他ICMP消息。

下一个函数recv_v6由图25.22给出。除了用于IPv6之外,与前一个函数(指recv_v4)基本相同,不过使用不同的常量名和不同的结构成员名。此外IPv6头部固定为40字节,而对于IPv4,我们必须读取头部长度字段,并乘以4,从而把可能有的IP选项也计在内。图25.21给出了本段代码所用的各种头部、指针和长度的图示。

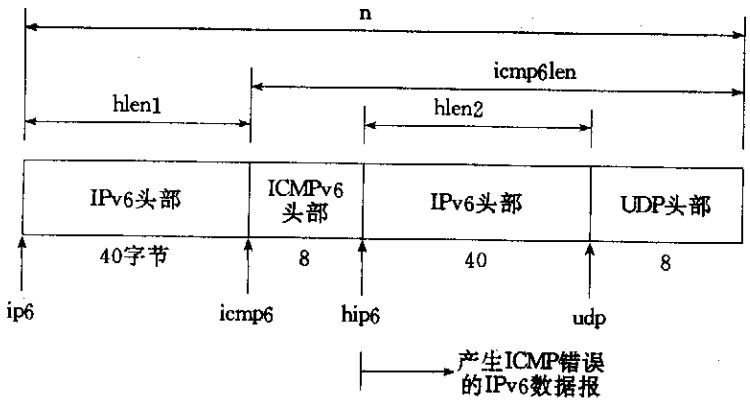


图25.21 ICMPv6出错消息的处理:头部、指针和长度

我们定义了两个函数icmpcode_v4和icmpcode_v6,由traceloop的最后几条语句来调用,对于ICMP目的地不可达错误输出相应的描述串。图25.23给出了其中的IPv6函数,IPv4函数类似,不过稍长一点,因为ICMPv4目的地不可达错误要多几个(图A.15)。

我们的Traceroute程序的最后一个函数是SIGALRM处理程序,见图25.24给出的sig_alrm。该函数所做的仅仅是返回,使recv_v4或recv_v6中的recvfrom返回一个EINTR错误。

```
1 #include    "trace.h"
2 /*
3  * Return:  -3 on timeout
4  *          -2 on ICMP time exceeded in transit (caller keeps going)
5  *          -1 on ICMP port unreachable (caller is done)
6  *          >= 0 return value is some other ICMP unreachable code
7  */
8 int
9 recv_v6(int seq, struct timeval *tv)
10 {
```

```
#ifdef  IPV6
    int     hlen1, hlen2, icmp6len;
    ssize_t n;
    socklen_t len;
    struct ip6_hdr *ip6, *hip6;
    struct icmp6_hdr *icmp6;
    struct udphdr *udp;
    alarm(3);
    for ( ; ; ) {
        len = pr->salen;
        n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0, pr->sarecv, &len);
        if (n < 0) {
            if (errno == EINTR)
                return(-3);     /* alarm expired */
            else
                err_sys("recvfrom error");
        }
        Gettimeofday(tv, NULL);     /* get time of packet arrival */
        ip6 = (struct ip6_hdr *) recvbuf;   /* start of IPv6 header */
        hlen1 = sizeof(struct ip6_hdr);
        icmp6 = (struct icmp6_hdr *) (recvbuf + hlen1); /* ICMP hdr */
        if ( (icmp6len = n - hlen1) < 8)
            err_quit("icmp6len (%d) < 8", icmp6len);
        if (icmp6->icmp6_type == ICMP6_TIME_EXCEEDED &&
            icmp6->icmp6_code == ICMP6_TIME_EXCEED_TRANSIT) {
            if (icmp6len < 8 + 40 + 8)
                err_quit("icmp6len (%d) < 8 + 40 + 8", icmp6len);
            hip6 = (struct ip6_hdr *) (recvbuf + hlen1 + 8);
            hlen2 = sizeof(struct ip6_hdr);
            udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);
            if (hip6->ip6_nxt == IPPROTO_UDP &&
                udp->uh_sport == htons(sport) &&
                udp->uh_dport == htons(dport + seq))
                return(-2);     /* we hit an intermediate router */
        } else if (icmp6->icmp6_type == ICMP6_DST_UNREACH) {
            if (icmp6len < 8 + 40 + 8)
                err_quit("icmp6len (%d) < 8 + 40 + 8", icmp6len);
            hip6 = (struct ip6_hdr *) (recvbuf + hlen1 + 8);
            hlen2 = 40;
            udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);
            if (hip6->ip6_nxt == IPPROTO_UDP &&
                udp->uh_sport == htons(sport) &&
                udp->uh_dport == htons(dport + seq)) {
                if (icmp6->icmp6_code == ICMP6_DST_UNREACH_NOPORT)
                    return(-1); /* have reached destination */
                else
                    return(icmp6->icmp6_code);      /* 0, 1, 2, ... */
            }
        } else if (verbose) {
            printf(" (from %s: type = %d, code = %d)\n",
                    Sock_ntop_host(pr->sarecv, pr->salen),
                    icmp6->icmp6_type, icmp6->icmp6_code);
        }
```

```
64          /* Some other ICMP error, recvfrom() again */
65      }
66 #endif
67 }
```

图 25.22 recv_v6 函数：读取并处理 ICMPv6 消息[traceroute/recv_v6.c]

```
1 #include    "trace.h"
2 char *
3 icmpcode_v6(int code)
4 {
5     switch (code) {
6     case  ICMP6_DST_UNREACH_NOROUTE:
7         return("no route to host");
8     case  ICMP6_DST_UNREACH_ADMIN:
9         return("administratively prohibited");
10    case  ICMP6_DST_UNREACH_NOTNEIGHBOR:
11        return("not a neighbor");
12    case  ICMP6_DST_UNREACH_ADDR:
13        return("address unreachable");
14    case  ICMP6_DST_UNREACH_NOPORT:
15        return("port unreachable");
16    default:
17        return("[unknown code]");
18    }
19 }
```

图 25.23 返回与 ICMPv6 不可达代码相对应的字符串[traceroute/icmpcode_v6.c]

```
1 #include    "trace.h"
2 void
3 sig_alrm(int signo)
4 {
5     return;             /* just interrupt the recvfrom() */
6 }
```

图 25.24 sig_alrm 函数[traceroute/sig_alrm.c]

例子

下面先给出使用 IPv4 的例子：

```
solaris # traceroute gemini.tuc.noao.edu
traceroute to gemini.tuc.noao.edu (140.252.3.54):30 hops max, 12 data bytes
1 gw.kohala.com(206.62.226.62) 3.839 ms 3.595 ms 3.722 ms
2 tuc-1-s1-9.rtd.net (206.85.40.73) 42.014 ms 21.078 ms 18.826 ms
3 frame-gw.ttn.ep.net (198.32.152.9) 39.283 ms 24.598 ms 50.037 ms
4 tucson-nap-1.arizona.edu (198.32.152.248) 44.350 ms 78.109 ms 47.003 ms
5 Butch-ENET-BONE.Telcom.Arizona.EDU (128.196.11.5) 29.849 ms 46.664 ms 83.571
  ms
6 gateway.tuc.noao.edu (140.252.104.1) 37.376 ms 36.430 ms 30.555 ms
7 gemini.tuc.noao.edu (140.252.3.54) 70.476 ms 43.555 ms 88.716 ms
```

下面是使用 IPv6 的例子，对过长的行进行了折行。

```
solaris # traceroute ipng9.ipng.nist.gov
traceroute to ipng9.ipng.nist.gov (5f00:3100:8106:3300:0:c0:3302:5a),
30 hops max, 12 data bytes
```

```
1  6bone-router.cisco.inner.net (5f00:6d00:c01f:700:1:60:3e11:6770)
                185.869 ms  *  127.082ms
2  buzzcut.ipv6.nr1.navy.mil (5f00:3000:84fa:5a00::5)
                187.736 ms 199.455 ms 172.839 ms
3  ipng9.ipng.nist.gov (5f00:3100:8106:3300:0:c0:3302:5a)
                206.762 ms  *  441.081 ms
```

在本例子中，跳限为 1 的第二个探测分组超时，跳限为 3 的第二个探测分组也一样。

25.7　一个 ICMP 消息守护进程

在 UDP 套接口上接收异步 ICMP 错误(即 ICMP 出错消息)一直是个问题。主要原因在于，尽管内核接收了 ICMP 错误，它们也很少递送给需要的应用进程。在以前所学套接口 API 中我们看到，为了获取这些错误，我们必须将 UDP 套接口与一个 IP 地址建立连接(8.11节)。这主要是由于 recvfrom 调用只返回整数 errno 代码，而如果一个应用进程向多个目的地发送数据报并调用 recvfrom，则 recvfrom 函数难以告知应用进程到底哪个数据报出了问题。

在 31.4 节介绍的 XTI 稍稍改善了这个问题。应用程序在调用等价于套接口 API 中的 recvfrom 获得错误码后，还必须调用另一个函数(t_rcvuderr)来获得真正的错误以及引起错误的数据报的目的地址和端口号。然而这种办法还是有问题：内核同时只能容纳单个这样的异步错误的有关信息。举例来说，如果应用进程发送的 3 个数据报中有 2 个引发了 ICMP 错误，则只有一个错误会返回给应用程序。

在本节中，我们将给出另一种不涉及内核的解决方案。我们提供一个 ICMP 消息守护进程：icmpd，它分别创建一个 ICMPv4 的原始套接口和一个 ICMPv6 的原始套接口，并接收内核传递给这两个原始套接口的所有 ICMP 消息。同时，它还将创建一个基于流的 Unix 域套接口，将路径名/tmp/icmpd 捆绑在其上，并在此套接口上监听客户对此路径名的 connect 请求。见图 25.25。

UDP 应用进程(也是 icmpd 守护进程的客户)首先必须创建用于接收异步错误的 UDP 套接口，并 bind 一个临时端口到其上，原因一会儿会提到。其次，它必须创建一个 Unix 域套接口，并与 icmpd 众所周知的路径名(/tmp/icmpd)连接。见图 25.26。

在以上两步完成后，应用进程使用 Unix 域套接口的描述字传递机制，将它的 UDP 套接口描述字“传递”给 icmpd(见 14.7 节)。这样 icmpd 就得到这个套接口描述字的一个拷贝，并可以调用 getsockname 来获取绑定在这个套接口上的端口号。这个套接口描述字传递过程见图 25.27。

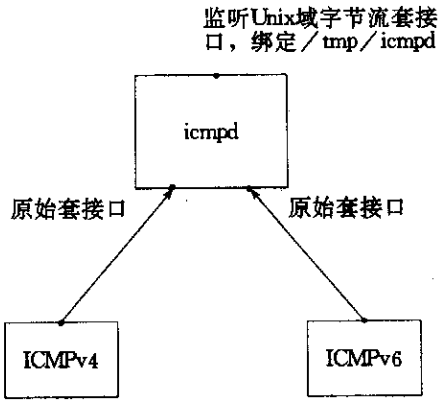


图 25.25 icmpd 守护进程，初始套接口建立

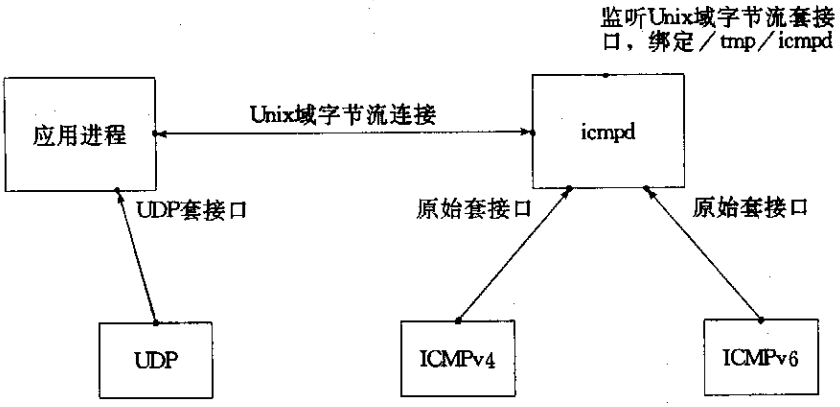


图 25.26 应用进程创建它的 UDP 套接口并与守护进程建立 Unix 域连接

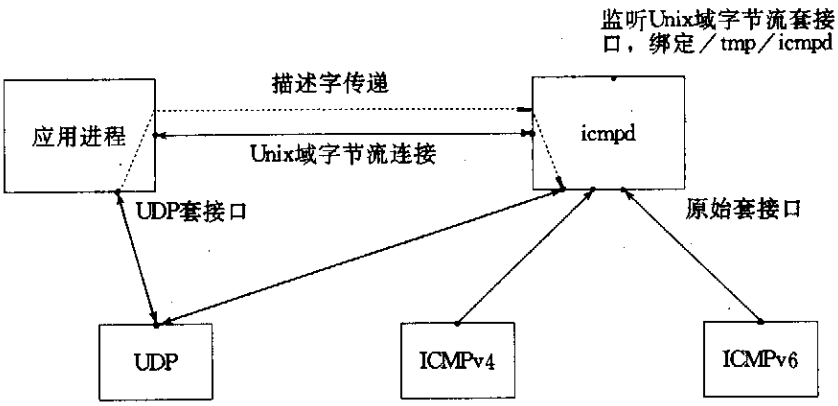


图 25.27 跨越 Unix 域连接，将 UDP 套接口描述字传递给 icmpd

在 icmpd 获取绑定在 UDP 套接口上的端口号之后，它关闭所获取的套接口拷贝，使我们返回图 25.26 所在的布局。

如果主机支持凭证传递(credential passing)机制的话(14.8 节),应用进程还应发送它的"凭证"给 icmpd,以便 icmpd 确定执行客户的用户是否有权使用这个 ICMP 消息获取机制。

从这时起,由该应用进程在其绑定所传递端口的 UDP 套接口上发送的 UDP 数据报所引起的 ICMP 错误,一旦由 icmpd 从内核处获取,则 icmpd 将通过 Unix 域套接口给该应用进程发送一个消息(我们很快会谈到)。当然,该应用进程因此必须使用 select 或 poll 来同时监视两个端口的输入。

下面我们查看使用 icmpd 的应用程序的源代码,接着查看 icmpd 的程序本身。图 25.28 是应用程序和 icmpd 程序都包括的头文件。

```
1 #ifndef   __unpicmp_h
2 #define   __unpicmp_h
3 #include   "unp.h"
4 #define   ICMPD_PATH  "/tmp/icmpd"   /* server's well-known pathname */
5 struct icmpd_err {
6   int     icmpd_errno;   /* EHOSTUNREACH, EMSGSIZE, ECONNREFUSED */
7   char    icmpd_type;    /* actual ICMPv[46] type */
8   char    icmpd_code;    /* actual ICMPv[46] code */
9   socklen_t  icmpd_len;  /* length of sockaddr{} that follows */
10  struct sockaddr icmpd_dest;   /* may be bigger */
11  char    icmpd_fill[MAXSOCKADDR - sizeof(struct sockaddr)];
12 };
14 #endif   /*  __unpicmp_h */
```

图 25.28　unpicmpd.h 头文件[icmpd/unpicmpd.h]

第 4～12 行　我们定义 icmpd 的众所周知路径名以及 icmpd_err 结构。当 icmpd 获取 ICMP 出错消息后,它将使用该结构来向相应的应用进程传递出错消息。

第 6～8 行　一个问题是 ICMPv4 消息类型和 ICMPv6 消息类型有不少差异(而且有时候是在概念上,见图 A.15 和 A.16)。除返回真正的 ICMP 类型和代码值外,我们还把这两个值映射成一个 errno 值(icmpd_errno),它跟图 A.15 和图 A.16 的最后一栏类似。应用进程可以直接利用这个值,而无需直接处理协议相关的 ICMPv4 和 ICMPv6 值。图 25.29 给出了所处理的 ICMP 消息以及它们映射到的 errno 值。icmpd 返回的 5 种 ICMP 错误如下:

1. 端口不可达:指示在目的 IP 地址上,没有绑定目的端口的套接口。
2. 分组过大错:该 ICMP 消息一般用于 MTU 发现。目前尚无 API 允许 UDP 应用程序自行进行路径 MTU 发现。在支持 UDP 路径 MTU 发现机制的内核上通常是这么做的:收到分组过大错导致内核在路由表里记录新的路径 MTU 值,但已发送分组因过大而丢弃这件事本身通知相应的 UDP 应用进程。该应用进程必须等待超时,然后重发该分组,而这时内核将根据路由表中新的路径 MTU 来对数据报进行分片。如果应用进程能获取本错误,则它就可以自行调整数据报的大小,并及时重传数据报。
3. 超时:本错误的 ICMP 代码一般为 0,指示 IPv4 的 TTL 或 IPv6 的跳限已达 0。一般而言,这标志着路由循环,可能是暂时的。
4. ICMPv4 源熄灭:尽管 RFC 1812[Baker 1995]反对使用本错误,它们仍可能由路由

器(或者配置不当的用作路由器的主机)发出。它们指示分组已被丢弃,因此我们像目的地不可达错误那样对付它们。注意IPv6并没有源熄灭错。

5. 其他不可达错误:这些错误指示分组已被丢弃。

icmpd_errno	ICMPv4错误	ICMPv6错误
ECONNREFUSED	端口不可达	端口不可达
EMSGSIZE	需分片但DF位已设置	分组过大
EHOSTUNREACH	超时	超时
EHOSTUNREACH	源熄灭	
EHOSTUNREACH	其他不可达错误	其他不可达错误

图25.29 icmpd_errno与ICMPv4及ICMPv6错误的映射关系

第10行 icmpd_dest成员是一个套接口地址结构,用于存放产生ICMP错误的目的IP地址及端口。该成员可为IPv4的sockaddr_in结构,也可为IPv6的sockaddr_in6结构。如果应用进程向多个目的主机发送数据报,那么每个目的主机都有一个对应的套接口地址结构。通过在某个套接口地址结构中返回这些信息,应用进程可以将它和它自己的各个结构相比较,判断到底是哪个产生了错误。

第11行 icmpd_fill成员用于填充空间,它将icmpd_err填充成能容纳最大可能的套接口地址结构。

使用icmpd的UDP回射客户程序

针对以上提供的机制,新的UDP回射客户程序的dg_cli函数将修改如下。图25.30给出了函数的前半部分。

```
1 #include    "unpicmpd.h"
2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int          icmpfd, maxfdp1;
6     char         sendline[MAXLINE], recvline[MAXLINE + 1];
7     fd_set    rset;
8     ssize_t n;
9     struct timeval tv;
10    struct icmpd_err icmpd_err;
11    Sock_bind_wild(sockfd, pservaddr->sa_family);
12    icmpfd = Tcp_connect("/unix", ICMPD_PATH);
13    Write_fd(icmpfd, "1", 1, sockfd);
14    n = Read(icmpfd, recvline, 1);
15    if (n != 1 || recvline[0] != '1')
16        err_quit("error creating icmp socket, n = %d, char = %c",
17                 n, recvline[0]);
18    FD_ZERO(&rset);
19    maxfdp1 = max(sockfd, icmpfd) + 1;
```

图25.30 dg_cli函数前半部分[icmpd/dgcli01.c]

第2~3行 dg_cli函数的参数保持不变。

捆绑通配地址及临时端口

第 11 行　我们调用 sock_bind_wild 给 UDP 套接口捆绑通配 IP 地址和一个临时端口。这样，在该套接口的拷贝传递到 icmpd 时，它已绑定一个端口，而 icmpd 确定需要知道这个端口号。

> 当 icmpd 接收的套接口上未绑定一个本地端口时，该守护进程也可以调用 bind，然而这个方法并不是对所有环境都有效。在 SVR4 实现（例如 Solaris 2.5）中套接口并不是内核的一部分，当一个进程在一个共享套接口上捆绑一个端口时，拥有这个套接口拷贝的其他进程就会在使用这个套接口时遇到一些奇怪的错误。因此，最简单的处理办法就是由客户在将套接口描述字传递给 icmpd 前预先捆绑本地端口。

与 icmpd 建立 Unix 域连接

第 12 行　调用 tcp_connect 函数创建一个 Unix 域字节流套接口，并 connect 到 icmpd 的众所周知路径。（回忆一下 11.5 节，我们的 getaddrinfo 实现支持 Unix 域套接口。）

向 icmpd 发送 UDP 套接口，等待它的应答

第 13～17 行　我们调用 write_fd 函数（图 14.13）向 icmpd 发送 UDP 套接口的拷贝，同时附带一个字节的数据即字符“1”，因为某些实现要求在传递描述字时同时附带数据。此后，icmpd 发回一个字节的数据：字符“1”指示发送成功，其他应答意味着失败。

第 18～19 行 初始化描述字集，并计算 select 的第一个参数（两个套接口描述字的较大值再加 1）。

图 25.31 是客户程序的后半部分。它是一个循环，在循环内，应用进程反复从标准输入读取一行，将该行发送给服务器，读服务器的应答，并将应答写到标准输出。

```
20    while (Fgets(sendline, MAXLINE, fp) != NULL) {
21        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
22        tv.tv_sec = 5;
23        tv.tv_usec = 0;
24        FD_SET(sockfd, &rset);
25        FD_SET(icmpfd, &rset);
26        if ( (n = Select(maxfdp1, &rset, NULL, NULL, &tv)) == 0) {
27            fprintf(stderr, "socket timeout\n");
28            continue;
29        }
30        if (FD_ISSET(sockfd, &rset)) {
31            n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
32            recvline[n] = 0;  /* null terminate */
33            Fputs(recvline, stdout);
34        }
35        if (FD_ISSET(icmpfd, &rset)) {
36            if ( (n = Read(icmpfd, &icmpd_err, sizeof(icmpd_err))) == 0)
37                err_quit("ICMP daemon terminated");
38            else if (n != sizeof(icmpd_err))
39                err_quit("n = %d, expected %d", n, sizeof(icmpd_err));
40            printf("ICMP error: dest = %s, %s, type = %d, code = %d\n",
41                    Sock_ntop(&icmpd_err.icmpd_dest, icmpd_err.icmpd_len),
```

```
42                  strerror(icmpd_err.icmpd_errno),
43                  icmpd_err.icmpd_type, icmpd_err.icmpd_code);
44          }
45      }
46  }
```

图 25.31 dg_cli 函数后半部分[icmpd/dgcli01.c]

调用 select

第 22～29 行 既然我们是在调用 select，因此可以很容易地给等待服务器的应答设置超时。我们设置超时时限为 5 秒，打开所创建的两个套接口的描述字可读条件，然后调用 select。一旦超时，就输出一个消息，并返回循环头部(continue 语句)。

输出服务器的应答

第 30～34 行 如果服务器返回数据报，则将它写到标准输出上。

处理 ICMP 错误

第 35～44 行 如果与 icmpd 守护进程之间的 Unix 域套接口可读的话，我们试图读取一个 icmpd_err 结构。如果成功，则输出守护进程返回的相关信息。

> strerror 是一个缺乏移植性的简单函数的例子。首先，ANSI C 和 Posix.1 没有规定该函数出错时的返回。Solaris 2.5 手册页面中说，一旦参数超出可行范围，该函数应返回空指针。这意味着形如：
>
> ```
> printf("%s",strerror(arg));
> ```
>
> 这样的语句是不正确的，因为即使参数未超出可行范围，strerror 也可能返回空指针。不过 UnixWare 2.1 以及作者所见到的其他源代码中，对于错误参数一般通过返回形如"Unknown error"的字符串来指示。这使得上面的语句输出会好看些。不过 Unix 98 又作了些改动。由于没有任何值被保留用来指示一个错误，如果参数值超出可行范围的话，该函数将 errno 设为 EINVAL(对出错情况下返回的指针未做任何规定)。这意味着正规的代码应先设置 errno 为 0，调用 strerror 函数，再检查 errno 是否为 EINVAL，如果有错则再输出某个其他消息。

UDP 回射客户程序使用实例

在说明 icmpd 源代码前，我们先看一下客户程序的使用实例。

先给一个未连接到因特网上的 IP 地址发送数据报：

```
solaris % udpcli01 192.3.4.5 echo
hi there
socket timeout
and hello
socket timeout
```

我们假设 icmpd 正在运行，并预期将由某个路由器返回主机不可达 ICMP 错误，不过什么也没有收到，代之以应用进程自己的超时提示。举这个例子是为了强调 ICMP 主机不可达错并不一定发生，因此应用程序自己的超时设置不可省略。过 30 秒再次运行该程序，我们的确收到了所希望的 ICMP 错误。

```
solaris % udpcli01 192.3.4.5 echo
hello
```

```
ICMP error: dest = 192.3.4.5.7, No route to host, type = 3,code = 1
```

我们的下一个例子是向一个不在运行标准回射服务器的主机发送数据报，得到一个ICMPv4 端口不可达错，与所期望的一样。

```
solaris % udpcli01 gemini.tuc.noao.edu echo
hello, world
ICMP error: dest = 140.252.4.54.7, Connection refused, type = 3, code = 3
```

icmpd 守护进程

我们先从头文件 icmpd.h 开始叙述，如图 25.32 所示。

client数组

第 2～17 行　由于 icmpd 可同时服务多个客户，因此我们使用一个 client 结构数组来分别记录与每个客户相关的信息。这有点像 6.8 节里所用的数据结构。除了到客户的 Unix 域已连接描述字以外，我们还记录 UDP 套接口的地址族(AF_INET 或 AF_INET6)以及绑定在该套接口上的端口号。除此以外，在头文件里还说明了各函数的原型和几个全程变量。

图 25.33 为 main 函数的前半部分。

```
 1 #include    "unpicmpd.h"
 2 struct client {
 3   int connfd;          /* Unix domain stream socket to client */
 4   int family;          /* AF_INET or AF_INET6 */
 5   int lport;           /* local port bound to client's UDP socket */
 6   /* network byte ordered */
 7 } client[FD_SETSIZE];
 8                        /* globals */
 9 int          fd4, fd6, listenfd, maxi, maxfd, nready;
10 fd_set    rset, allset;
11 socklen_t addrlen;
12 struct sockaddr *cliaddr;
13               /* function prototypes */
14 int      readable_conn(int);
15 int      readable_listen(void);
16 int      readable_v4(void);
17 int      readable_v6(void);
```

图 25.32　icmpd 守护进程程序的 icmpd.h 头文件[icmpd/icmpd.h]

```
 1 #include    "icmpd.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, sockfd;
 6     if (argc != 1)
 7         err_quit("usage: icmpd");
 8     maxi = -1;                      /* index into client[] array */
 9     for (i = 0; i < FD_SETSIZE; i++)
10         client[i].connfd = -1;   /* -1 indicates available entry */
11     FD_ZERO(&allset);
12     fd4 = Socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
13     FD_SET(fd4, &allset);
14     maxfd = fd4;
```

```
15 #ifdef    IPV6
16     fd6 = Socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);
17     FD_SET(fd6, &allset);
18     maxfd = max(maxfd, fd6);
19 #endif
20     listenfd = Tcp_listen("/unix", ICMPD_PATH, &addrlen);
21     FD_SET(listenfd, &allset);
22     maxfd = max(maxfd, listenfd);
23     cliaddr = Malloc(addrlen);
```

图 25.33 main 函数前半部分：创建套接口[icmpd/icmpd.c]

初始化 client 数组

第 9～10 行 通过设置 client 数组的已连接套接口成员为－1 以初始化整个数组。

创建套接口

第 12～23 行 创建三个套接口：一个原始 ICMPv4 套接口、一个原始 ICMPv6 套接口和一个 Unix 域字节流套接口。该 Unix 域字节流套接口通过调用我们的 tcp_listen 函数创建，该函数还把众所周知的路径名捆绑在其上并调用 listen。该套接口也是客户 connect 的套接口。我们还计算 select 所用的最大描述字号，并分配由 accept 使用的套接口地址结构。

图 25.34 为 main 函数的第二部分。它是一个无限循环，在循环中调用 select，等待任意描述字成为可读。

```
24     for ( ; ; ) {
25         rset = allset;
26         nready = Select(maxfd+1, &rset, NULL, NULL, NULL);
27         if (FD_ISSET(listenfd, &rset))
28             if (readable_listen() <= 0)
29                 continue;
30         if (FD_ISSET(fd4, &rset))
31             if (readable_v4() <= 0)
32                 continue;
33 #ifdef    IPV6
34         if (FD_ISSET(fd6, &rset))
35             if (readable_v6() <= 0)
36                 continue;
37 #endif
38         for (i = 0; i <= maxi; i++) { /* check all clients for data */
39             if ( (sockfd = client[i].connfd) < 0)
40                 continue;
41             if (FD_ISSET(sockfd, &rset))
42                 if (readable_conn(i) <= 0)
43                     break;                  /* no more readable descriptors */
44         }
45     }
46     exit(0);
47 }
```

图 25.34 main 函数后半部分：处理可读描述字[icmpd/icmpd.c]

检查监听 Unix 域套接口

第 27～29 行 首先测试的是 Unix 域套接口，它如果准备好了就调用 readable_listen。

select 返回的可读描述字数(nready)是一个全程变量。每一个形如 readable_XXX 的函数都将这个变量减 1,并将这个新值作为函数的返回值。当这个值等于 0 时,所有的可读描述字都已被处理完毕,于是再次调用 select。

检查原始 ICMP 套接口

第 30～37 行　分别测试 ICMPv4 和 ICMPv6 套接口。

检查已连接 Unix 域套接口

第 38～44 行　检查所有已连接 Unix 域套接口是否可读,如果可读,则意味着客户已发送描述口,或者客户已终止。

图 25.35 为当 icmpd 的监听套接口可读时调用的 readable_listen 函数,用于接受一个新的来自客户的连接。

```
1 #include    "icmpd.h"
2 int
3 readable_listen(void)
4 {
5     int             i, connfd;
6     socklen_t    clilen;
7     clilen = addrlen;
8     connfd = Accept(listenfd, cliaddr, &clilen);
9         /* find first available client[] structure */
10    for (i = 0; i < FD_SETSIZE; i++)
11        if (client[i].connfd < 0) {
12            client[i].connfd = connfd;      /* save descriptor */
13            break;
14        }
15    if (i == FD_SETSIZE)
16        err_quit("too many clients");
17    printf("new connection, i = %d, connfd = %d\n", i, connfd);
18    FD_SET(connfd, &allset);      /* add new descriptor to set */
19    if (connfd > maxfd)
20        maxfd = connfd;             /* for select() */
21    if (i > maxi)
22        maxi = i;                   /* max index in client[] array */
23    return(--nready);
24 }
```

图 25.35　处理新客户连接[icmpd/readable_listen.c]

第 7～23 行　接受客户的连接请求,并选用 client 数组中第一个可用项。此源代码是从图 6.22 前半部分拷贝过来的。

当一个已连接套接口可读时,我们调用 readable_conn 函数(图 25.36),其参数为相应客户在 client 数组中的下标。

```
1 #include    "icmpd.h"
2 int
3 readable_conn(int i)
4 {
5     int          unixfd, recvfd;
6     char         c;
```

```
7    ssize_t     n;
8    socklen_t   len;
9    union {
10     char                buf[MAXSOCKADDR];
11     struct sockaddr sock;
12   } un;
13   unixfd = client[i].connfd;
14   recvfd = -1;
15   if ( (n = Read_fd(unixfd, &c, 1, &recvfd)) == 0) {
16       err_msg("client %d terminated, recvfd = %d", i, recvfd);
17       goto clientdone;    /* client probably terminated */
18   }
19       /* data from client; should be descriptor */
20   if (recvfd < 0) {
21       err_msg("read_fd did not return descriptor");
22       goto clienterr;
23   }
```

图 25.36 读取来自客户的数据及可能有的描述字[icmpd/readable_conn.c]

读取客户发送的数据及可能有的描述字

第13～18行 我们调用图14.11中的read_fd函数来读取客户发送的数据以及可能有的描述字。如果返回0,则意味着客户已经关闭它所在的连接端,很可能是进程终止引起。

> 为了在应用进程和icmpd之间传递描述字,我们可以使用Unix域字节流套接口,也可以使用Unix域数据报套接口。应用进程的UDP套接口描述字可以用任意一种类型的Unix域套接口传递。之所以采用字节流套接口是为了检测客户何时终止。当客户终止时,它的所有描述字(包括与icmpd间的Unix域连接)都将自动关闭,这就通知icmpd从client数组中将这个客户去掉。要是使用数据报套接口,我们就无法知道客户何时终止。

第19～23行 如果客户未关闭连接,那我们期待一个描述字。

图25.37为readable_conn函数的另一部分。

```
24   len = sizeof(un.buf);
25   if (getsockname(recvfd, (SA *) un.buf, &len) < 0) {
26       err_ret("getsockname error");
27       goto clienterr;
28   }
29   client[i].family = un.sock.sa_family;
30   if ( (client[i].lport = sock_get_port(&un.sock, len)) == 0) {
31       client[i].lport = sock_bind_wild(recvfd, client[i].family);
32       if (client[i].lport <= 0) {
33           err_ret("error binding ephemeral port");
34           goto clienterr;
35       }
36   }
37   Write(unixfd, "1", 1);    /* tell client all OK */
38   FD_SET(unixfd, &allset);
39   if (unixfd > maxfd)
40       maxfd = unixfd;
41   if (i > maxi)
```

```
42          maxi = i;
43      Close(recvfd);              /* all done with client's UDP socket */
44      return(--nready);
45  clienterr:
46      Write(unixfd, "0", 1);   /* tell client error occurred */
47  clientdone:
48      Close(unixfd);
49      if (recvfd >= 0)
50          Close(recvfd);
51      FD_CLR(unixfd, &allset);
52      client[i].connfd = -1;
53      return(--nready);
54  }
```

图 25.37　获取客户捆绑在 UDP 套接口上的端口号[icmpd/readable_conn.c]

获取绑定在 UDP 套接口上的端口号

第 24～28 行　守护进程通过调用 getsockname 来获取绑定在套接口上的端口号。由于我们不知道到底要给套接口地址结构申请多大的缓冲区，因此我们声明一个包含一个字符数组和一个通用套接口地址结构的联合。这样能够保证该字符数组按照套接口地址结构适当对齐；如果我们仅仅声明一个字符数组就不能满足对齐要求。我们在图 11.7 里讨论过这个问题，那儿我们调用 malloc 来做到对齐。

第 29～36 行　套接口的地址族存储在 client 结构里，同时存储的还有端口号。如果端口号为 0，则 icmpd 调用 sock_bind_wild 来给套接口捆绑通配地址和一个临时端口。不过如前面所述，这在 SVR4 实现上不可行。

通知客户一切正常

第 37～42 行　将字符“1”回送给客户，并将获取的新描述字加进 select 所用的描述字集中，同时根据需要更新 maxfd 和 maxi。

关闭客户的 UDP 套接口

第 43 行　现在客户的 UDP 套接口拷贝已经完成了它的任务，于是将其关闭。当然，这个描述字是客户传递过来的一个拷贝，因此在客户中，UDP 套接口还是打开的。

处理错误和客户终止

第 45～53 行　当出错时，向客户发送一个字符“0”。当客户终止时，关闭 Unix 域套接口连接的服务器端，并从 select 的描述字集中去除该描述字。我们还将 client 数组中对应元素的 connfd 成员置为-1，以指示该项可作他用。

当原始 ICMPv4 套接口可读时，icmpd 调用 readable_v4 函数。它的前半部分源代码见图 25.38，这与图 25.8 和图 25.19 中处理 ICMPv4 的代码有些类似。

```
1 #include    "icmpd.h"
2 #include    <netinet/in_systm.h>
3 #include    <netinet/ip.h>
4 #include    <netinet/ip_icmp.h>
5 #include    <netinet/udp.h>
6 int
7 readable_v4(void)
8 {
```

```
 9    int        i, hlen1, hlen2, icmplen, sport;
10    char       buf[MAXLINE];
11    char       srcstr[INET_ADDRSTRLEN], dststr[INET_ADDRSTRLEN];
12    ssize_t n;
13    socklen_t len;
14    struct ip  *ip, *hip;
15    struct icmp  *icmp;
16    struct udphdr  *udp;
17    struct sockaddr_in from, dest;
18    struct icmpd_err icmpd_err;
19    len = sizeof(from);
20    n = Recvfrom(fd4, buf, MAXLINE, 0, (SA *) &from, &len);
21    printf("%d bytes ICMPv4 from %s:",
22           n, Sock_ntop_host((SA *) &from, len));
23    ip = (struct ip *) buf;          /* start of IP header */
24    hlen1 = ip->ip_hl << 2;          /* length of IP header */
25    icmp = (struct icmp *) (buf + hlen1);      /* start of ICMP header */
26    if ( (icmplen = n - hlen1) < 8)
27        err_quit("icmplen (%d) < 8", icmplen);
28    printf(" type = %d, code = %d\n", icmp->icmp_type, icmp->icmp_code);
```

图 25.38 处理接收到的 ICMPv4 数据报的前半部分[icmpd/readable_v4.c]

这个函数输出每一个接收到的 ICMPv4 消息的部分信息。这些为开发 icmpd 时调试之用的,并可由命令行参数来控制。

图 25.39 为 readable_v4 函数的后半部分。

```
29    if (icmp->icmp_type == ICMP_UNREACH ||
30        icmp->icmp_type == ICMP_TIMXCEED ||
31        icmp->icmp_type == ICMP_SOURCEQUENCH) {
32        if (icmplen < 8 + 20 + 8)
33            err_quit("icmplen (%d) < 8 + 20 + 8", icmplen);
34        hip = (struct ip *) (buf + hlen1 + 8);
35        hlen2 = hip->ip_hl << 2;
36        printf("\tsrcip = %s, dstip = %s, proto = %d\n",
37               inet_ntop(AF_INET, &hip->ip_src, srcstr, sizeof(srcstr)),
38               inet_ntop(AF_INET, &hip->ip_dst, dststr, sizeof(dststr)),
39               hip->ip_p);
40        if (hip->ip_p == IPPROTO_UDP) {
41            udp = (struct udphdr *) (buf + hlen1 + 8 + hlen2);
42            sport = udp->uh_sport;
43                /* find client's Unix domain socket, send headers */
44            for (i = 0; i <= maxi; i++) {
45                if (client[i].connfd >= 0 &&
46                    client[i].family == AF_INET &&
47                    client[i].lport == sport) {
48                    bzero(&dest, sizeof(dest));
49                    dest.sin_family = AF_INET;
50 #ifdef     HAVE_SOCKADDR_SA_LEN
51                    dest.sin_len = sizeof(dest);
52 #endif
53                    memcpy(&dest.sin_addr, &hip->ip_dst,
54                           sizeof(struct in_addr));
```

```
55                      dest.sin_port = udp->uh_dport;
56                      icmpd_err.icmpd_type = icmp->icmp_type;
57                      icmpd_err.icmpd_code = icmp->icmp_code;
58                      icmpd_err.icmpd_len = sizeof(struct sockaddr_in);
59                      memcpy(&icmpd_err.icmpd_dest, &dest, sizeof(dest));
60                          /* convert type & code to reasonable errno value */
61                      icmpd_err.icmpd_errno = EHOSTUNREACH;   /* default */
62                      if (icmp->icmp_type == ICMP_UNREACH) {
63                          if (icmp->icmp_code == ICMP_UNREACH_PORT)
64                              icmpd_err.icmpd_errno = ECONNREFUSED;
65                              else if (icmp->icmp_code == ICMP_UNREACH_
                                NEEDFRAG)
66                              icmpd_err.icmpd_errno = EMSGSIZE;
67                      }
68                      Write(client[i].connfd, &icmpd_err, sizeof(icmpd_err));
69                  }
70              }
71          }
72      }
73      return(--nready);
74  }
```

图 25.39　处理接收到的 ICMPv4 数据报的后半部分[icmpd/readable_v4.c]

检查消息类型，通知应用进程

第 29～31 行　仅当 ICMPv4 消息为目的地不可达、超时或源熄灭时，才通知客户(图 25.29)。

检查是否为 UDP 错误，查找对应客户

第 34～42 行　hip 指向所返回消息中 ICMP 头部后的 IP 头部。这是引发 ICMP 错误的数据报的 IP 头部。我们验证这个 IP 数据报是一个 UDP 数据报，并从 IP 头部后的 UDP 头部中获取源 UDP 端口号。

第 43～55 行　搜索整个 client 数组，寻找地址族和端口都匹配的客户。如果找到，则构造一个 IPv4 套接口 地址结构，其中含有引发错误的那个 UDP 数据报的目的 IP 地址和端口号。

构造 icmpd_err 结构

第 56～70 行　构造一个 icmpd_err 结构，跨越与客户的 Unix 域连接发送给客户。其中 ICMPv4 消息类型和代码先被映射为某个 errno 值(见图 25.29)。

ICMPv6 错误由 readable_v6 函数处理，其前半部分见图 25.40。ICMPv6 的处理与图 25.10 和图 25.22 的代码类似。

readable_v6 函数的后半部分见图 25.41，代码与图 25.39 类似，检查 ICMP 错误的类型，检查引起错误的数据报是否是 UDP 数据报，然后构造发送给客户的 icmpd_err 结构。

```
1 #include    "icmpd.h"
2 #include    <netinet/in_systm.h>
3 #include    <netinet/ip.h>
4 #include    <netinet/ip_icmp.h>
5 #include    <netinet/udp.h>
6 #ifdef    IPV6
```

```
 7 #include    "ip6.h"                 /* should be <netinet/ip6.h> */
 8 #include    "icmp6.h"               /* should be <netinet/icmp6.h> */
 9 #endif
10 int
11 readable_v6(void)
12 {
13 #ifdef    IPV6
14     int          i, hlen1, hlen2, icmp6len, sport;
15     char         buf[MAXLINE];
16     char         srcstr[INET6_ADDRSTRLEN], dststr[INET6_ADDRSTRLEN];
17     ssize_t n;
18     socklen_t len;
19     struct ip6_hdr *ip6, *hip6;
20     struct icmp6_hdr *icmp6;
21     struct udphdr *udp;
22     struct sockaddr_in6 from, dest;
23     struct icmpd_err icmpd_err;
24     len = sizeof(from);
25     n = Recvfrom(fd6, buf, MAXLINE, 0, (SA *) &from, &len);
26     printf("%d bytes ICMPv6 from %s:",
27             n, Sock_ntop_host((SA *) &from, len));
28     ip6 = (struct ip6_hdr *) buf;       /* start of IPv6 header */
29     hlen1 = sizeof(struct ip6_hdr);
30     if (ip6->ip6_nxt != IPPROTO_ICMPV6)
31         err_quit("next header not IPPROTO_ICMPV6");
32     icmp6 = (struct icmp6_hdr *) (buf + hlen1);
33     if ( (icmp6len = n - hlen1) < 8)
34         err_quit("icmp6len (%d) < 8", icmp6len);
35     printf(" type = %d, code = %d\n", icmp6->icmp6_type, icmp6->icmp6_code);
```

图 25.40 处理收到的 ICMPv6 数据报的前半部分[icmpd/readable_v6.c]

```
36     if (icmp6->icmp6_type == ICMP6_DST_UNREACH ||
37       icmp6->icmp6_type == ICMP6_PACKET_TOO_BIG ||
38       icmp6->icmp6_type == ICMP6_TIME_EXCEEDED) {
39       if (icmp6len < 8 + 40 + 8)
40           err_quit("icmp6len (%d) < 8 + 40 + 8", icmp6len);
41       hip6 = (struct ip6_hdr *) (buf + hlen1 + 8);
42       hlen2 = sizeof(struct ip6_hdr);
43       printf("\tsrcip = %s, dstip = %s, next hdr = %d\n",
44           Inet_ntop(AF_INET6, &hip6->ip6_src, srcstr, sizeof(srcstr)),
45           Inet_ntop(AF_INET6, &hip6->ip6_dst, dststr, sizeof(dststr)),
46           hip6->ip6_nxt);
47       if (hip6->ip6_nxt == IPPROTO_UDP) {
48           udp = (struct udphdr *) (buf + hlen1 + 8 + hlen2);
49           sport = udp->uh_sport;
50               /* find client's Unix domain socket, send headers */
51           for (i = 0; i <= maxi; i++) {
52               if (client[i].connfd >= 0 &&
53                   client[i].family == AF_INET6 &&
54                   client[i].lport == sport) {
55                   bzero(&dest, sizeof(dest));
56                   dest.sin6_family = AF_INET6;
57 #ifdef    HAVE_SOCKADDR_SA_LEN
```

```
58                      dest.sin6_len = sizeof(dest);
59 #endif
60                      memcpy(&dest.sin6_addr, &hip6->ip6_dst,
61                             sizeof(struct in6_addr));
62                      dest.sin6_port = udp->uh_dport;
63                      icmpd_err.icmpd_type = icmp6->icmp6_type;
64                      icmpd_err.icmpd_code = icmp6->icmp6_code;
65                      icmpd_err.icmpd_len = sizeof(struct sockaddr_in6);
66                      memcpy(&icmpd_err.icmpd_dest, &dest, sizeof(dest));
67                          /* convert type & code to reasonable errno value */
68                      icmpd_err.icmpd_errno = EHOSTUNREACH;  /* default */
69                      if (icmp6->icmp6_type == ICMP6_DST_UNREACH) {
70                          if (icmp6->icmp6_code == ICMP_UNREACH_PORT)
71                              icmpd_err.icmpd_errno = ECONNREFUSED;
72                                  else if (icmp6->icmp6_code == ICMP_UNREACH_
                                    NEEDFRAG)
73                              icmpd_err.icmpd_errno = EMSGSIZE;
74                      }
75                      Write(client[i].connfd, &icmpd_err, sizeof(icmpd_err));
76                  }
77              }
78          }
79      }
80      return(--nready);
81 #endif
82 }
```

图 25.41　处理收到的 ICMPv6 数据报的后半部分[icmpd/readable_v6.c]

25.8　小　结

原始套接口提供三种功能：

1. 我们可以读/写 ICMPv4、IGMPv4 和 ICMPv6 分组。
2. 我们可以读/写具有内核不处理的协议字段的 IP 数据报。
3. 我们可以构造我们自己的 IPv4 头部，这一般用于诊断目的(当然黑客们也利用这个功能)。

Ping 和 Traceroute 这两个最常用的诊断工具使用了原始套接口，我们开发了两者的我们自己的版本，它们同时支持 IPv4 和 IPv6。除此之外，我们还开发了自己的 icmpd 守护进程程序，以支持在 UDP 套接口上对 ICMP 错误的访问。它还是一个通过 Unix 域套接口在无亲缘关系的客户和服务器之间传递描述字的例子。

25.9　习　题

25.1　我们提到，IPv6 头部和扩展头部中几乎所有的字段都可以通过套接口选项或辅助数据得到，那么应用程序不能得到 IPv6 数据报中的哪些消息？

25.2 在图25.39中，如果由于某种原因，客户停止从到icmpd守护进程的Unix域连接中读取数据，而同时有关该客户的大量ICMP错误到达，那么会发生什么情况？最简单的解决办法是什么？

25.3 如果自己使用我们的Ping程序来ping一个子网广播地址，那么即使我们不设置SO_BROADCAST套接口选项，广播ICMP回射请求仍会作为一个链路层的广播帧发出，为什么？

25.4 如果使用我们的Ping程序在一台多宿主机上ping所有主机多播组(其地址为244.0.0.1)，会有什么情况发生？

第 26 章　数据链路访问

26.1　概　述

目前大多数操作系统都为应用程序提供了访问数据链路层的手段，它使得应用程序拥有如下功能：

- 监视数据链路层上所收到的分组，这使得我们可以在普通计算机系统上通过像 tcpdump 这样的程序来监视网络，而无需使用特殊的硬件设备。如果结合使用网络接口的混杂模式(promiscuous mode)，我们甚至可以侦听本地电缆上的所有分组，而不只是以程序运行所在主机为目的地的分组。
- 作为普遍应用进程而不是内核的一部分运行某些程序。例如：大多数 Unix 系统的 RARP 服务器是普通的应用进程，它们从数据链路读取 RARP 请求(RARP 请求不是 IP 数据报)，并把应答写回数据链路。

Unix 上三种最常用的数据链路层访问方法是 BSD 的 BSD 分组过滤器(BPF)、SVR4 的数据链路提供者接口(DLPI)和 Linux 的 SOCK_PACKET 接口。这里，我们先简要介绍一下这三种方法，然后介绍 libpcap，它是公开可得的分组截获函数库。该函数库在所有三种接口上都可使用，因此使用它的程序独立于操作系统提供的实际数据链路访问。我们将通过一个向名字服务器发送 DNS 请求的程序来说明这个库(我们构造自己的 UDP 数据报并把它们写到一个原始套接口)，该程序使用 libpcap 从数据链路获取名字服务器的应答，以此判断名字服务器是否使能 UDP 校验和。

26.2　BPF：BSD 分组过滤器

4.4BSD 以及许多其他源自 Berkeley 的实现使用 BPF 作为数据链层访问手段，其实现方式见 TCPv2 的第 31 章。BPF 的历史、BPF 伪机器以及与 SunOS 4.1.x 上的 NIT 分组过滤器的比较见[McCanne and Jacobson 1993]。

在支持 BPF 的系统上，每个数据链路驱动程序就在收到一个分组之后或发送一个分组之前调用 BPF，见图 26.1。

这些调用的例子参见 TCPv2 的图 4.11 和图 4.19(以太网接口)。在收到后立即调用 BPF 以及在发送前最后一刻调用 BPF 的原因是为了提供精确的时间戳。

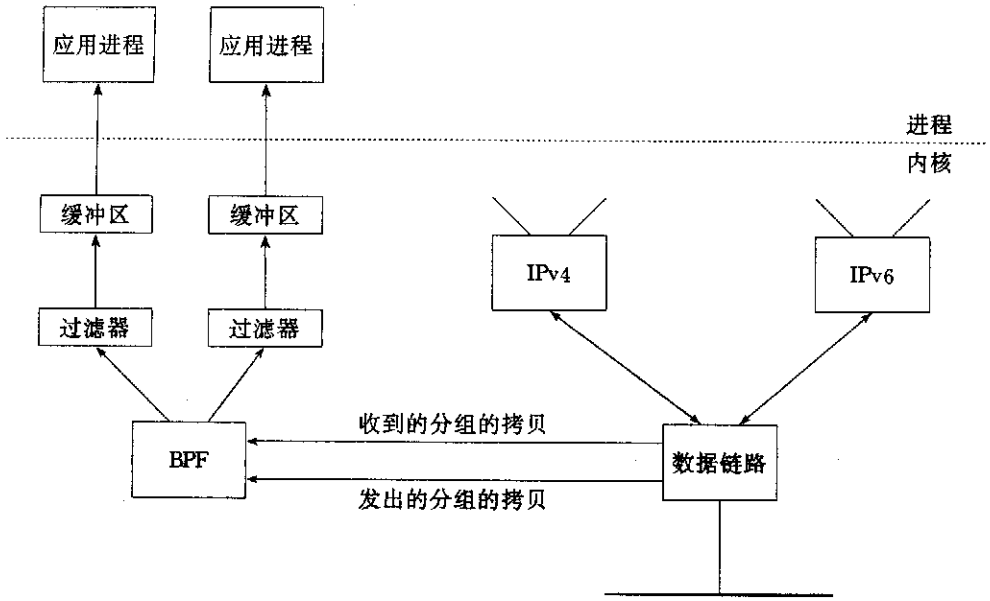


图 26.1 使用 BPF 截获分组

虽然在数据链路嵌入一个分组捕获机制并不困难，BPF 的强大威力却在于它的过滤功能。任何一个打开 BPF 设备的应用进程都可以装入自己的过滤器，该过滤器然后就由 BPF 应用于各个分组。过滤器可以简单(udp or tcp：只接收 UDP 或 TCP 分组)，也可以复杂(检查分组头部各个字段是否为特定值)，如下例：

```
tcp and port 80 and tcp[13:1] & 0x7 ! = 0
```

该过滤器在 TCPv3 的第 14 章使用，它仅收集来往于端口 80 的设置了 SYN、FIN 或 RST 标志的 TCP 分节。表达式 tcp[13:1]指自 TCP 头部开始位置起偏移量为 13 个字节的那个字节的值。

BPF 实现一个基于寄存器的过滤器机，该机器对每个收到的分组应用一个特定于应用进程的过滤器。程序员可以使用这个伪机器的机器语言编写自己的过滤器程序(这在 BPF 的手册页面中说明)，但最简单的接口却是使用我们将在 26.6 节叙述的 pcap_compile 函数把 ASCII 字符串(例如刚才给出的 tcp 开头的那个字符串)编译成伪机器的机器语言。

为了减少开销，BPF 使用以下三种技术：

1. BPF 过滤由内核完成，以减少从 BPF 拷贝到应用进程的数据量。从系统空间到用户空间的拷贝是昂贵的，如果每个分组都这么拷贝，那么 BPF 将跟不上快速的数据链路。
2. 每个分组只有部分数据由 BPF 传递给应用进程，这被称为捕获长度(capture length)。大多数应用进程只需要分组头部，而不需要分组数据。这同样减少了内核到用户空间的数据拷贝量。譬如：tcpdump 将该值缺省设为 68，允许 1 个 14 字节的以太网头部、一个 20 字节的 IP 头部、一个 20 字节的 TCP 头部以及 14 字节数据。当然，如果需要获取其他协议(例如 DNS 和 NFS)的更多的信息，那么用户在运行

tcpdump 时应增大该值。

3. BPF 缓冲递送给应用进程的数据。该缓冲区只有在已满或读超时(read timeout)发生时才拷贝给应用进程。超时值可由应用进程指定。例如 tcpdump 把超时设为 1000ms;而 RARP 守护进程则把它设为 0(RARP 分组并不多,而且 RARP 服务器一接收到请求就得发送响应)。缓冲的目的在于减少系统调用的次数。BPF 和应用进程之间拷贝的分组数目仍然一样,但每个系统调用都需要一定开销,减少系统调用则可减少开销(例如,APUE 的图 3.1 比较了使用在 1 字节到 131,072 字节之间变化的不同数据块大小读一个给定文件时 read 系统调用的开销)。

   尽管图 26.1 中只画了一个缓冲区,BPF 其实给每个应用进程维护两个缓冲区,当一个缓冲区在向应用进程拷贝数据时,启用另一个缓冲区来装填数据。这就是标准的双缓冲(double buffering)技术。

图 26.1 我们只给出了 BPF 对分组的接收,包括:由数据链路从下面(网络)接收的分组以及从上面(IP)接收的分组。应用进程也可以向 BPF 写,使得分组从数据链路往外(向上和向下)发出,但大多数应用进程仅仅从 BPF 读而已。没有理由通过向 BPF 写来发送 IP 数据报,因为 IP_HDRINCL 套接口选项允许我们写期望的任何类型的 IP 数据报,包括 IP 头部在内。(我们将在 26.6 节给出一个这样的例子。)向 BPF 写的唯一原因是为了发送不是 IP 数据报的我们自己的网络分组。例如 RARP 守护进程就这么做来发送 RARP 应答,它们不是 IP 数据报。

为了访问 BPF,应用进程需要打开一个未使用的 BPF 设备。例如可以试着打开/dev/bpf0,如果返回 EBUSY 错,则尝试打开/dev/bpf1,如此一直到成功地打开某个 BPF 设备。当设备打开后,要使用一系列的 ioctl 命令来设置设备属性:装入过滤器、设置读超时、设置缓冲区大小、与某个数据链路附接、使能混杂模式,等等。这一切完成之后,就可以用 read 和 write 来完成 I/O。

26.3　DLPI:数据链路提供者接口

SVR4 通过 DLPI 来实现数据链路访问。DLPI 是 AT&T 设计的独立于协议的访问数据链路层所提供服务的接口[Unix International 1991],其访问通过发送和接收流消息来实现。

应用进程介入数据链路层只需打开设备(例如 le0)并使用 DLIP 的 DL_ATTACH_REQ 请求将它与 DLPI 附接就可以了。不过为了提高效率,一般还需压入两个流模块:pfmod(在内核中进行分组过滤)和 bufmod(缓冲递送给应用进程的数据)。见图 26.2。

从原理上说,这与上一节所讲的 BPF 相类似:pfmod 在内核中使用伪机器实现分组过滤,bufmod 则通过支持捕获长度和读超时来减少系统调用次数和拷贝的数据量。

BPF 与 pfmod 过滤器虽然都使用伪机器来实现包过滤,但它们所支持的伪机器类型却存在有趣的差别。BPF 过滤器使用有向无环控制流图(CFG),而 pfmod 使用布尔表达式树。前者自然映射成寄存器型机器代码,后者自然映射成堆栈式机器代码[McCanne and Jacobson 1993]。该论文证明根据过滤器的复杂程度,CFG 实现方式可比布尔表达式树这种实现方式快 3～20 倍。

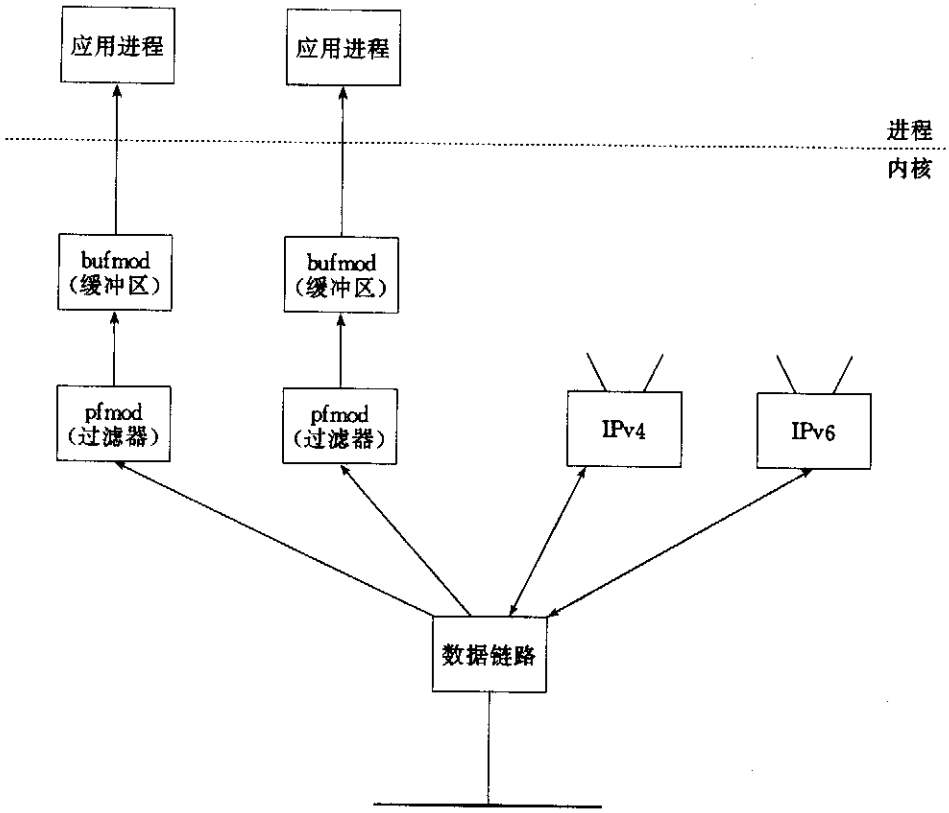


图 26.2 使用 DLPI、pfmod 和 bufmod 捕获分组

26.4 Linux:SOCK_PACKET

在 Linux 下读取数据链路层分组需要创建 SOCK_PACKET 类型的套接口。与原始套接口类似,这也需要超级用户权限,而且调用 socket 的第三个参数必须是指定以太网帧类型的某个非 0 值。例如,从数据链路获取所有帧应如下创建套接口:

```
fd = socket (AF_INET, SOCK_PACKET, htons(ETH_P_ALL));
```

这将返回数据链路接收的所有协议的帧。如果需要捕获 IPv4 帧,则可以如下调用 socket:

```
fd = socket (AF_INET, SOCK_PACKET, htons(ETH_P_IP));
```

最后一个参数使用的其他常值还有 ETH_P_ARP、ETH_P_IPV6 等。

指定协议 ETH_P_xxx 通知数据链路将它收到的哪些帧类型传递给套接口。如果数据链路支持混杂模式(如以太网),那么如果需要还必须将设备设为混杂模式。这是通过 SIOCGIFFLAGS 的 ioctl 获取标志,设置 IFF_PROMISC 标志,再用 SIOCSIFFLAGS 存储标志完成的。

与 BPF 以及 DLPI 相比较,Linux 有以下几点不同。

1. Linux 不提供基于核心的缓冲和分组过滤机制。它提供普通的套接口接收缓冲区,但

多个帧不能缓冲在一起，一次性地由应用进程读取。这么一来从内核向应用进程拷贝大量数据的开销势必增长。

2. Linux 不提供针对设备的过滤。如果调用 socket 时指定 ETH_P_IP，那么从任何设备（例如以太网、PPP 链路、SLIP 链路和回馈设备）来的 IPv4 分组都将传递给该套接口。recvfrom 将返回一个通用套接口地址结构，其中的 sa_data 成员含有设备名（如 eth0）。应用进程必须自己丢弃来自它不感兴趣的任何设备的数据。问题仍然是可能会给应用进程返回太多的数据，这在监视高速网络时可能真成问题。

26.5　libpcap：分组捕获函数库

libpcap 是一个与实现无关的访问操作系统所提供的分组捕获机制的分组捕获函数库。目前它只支持分组的读取（当然增加一些代码行之后，也可以写数据链路分组）。

它同时支持源自 Berkeley 内核下的 BPF、Solaris 2.x 下的 DLPI、SunOS 4.1x 下的 NIT、Linux 的 SOCK_PACKET 套接口以及其他若干操作系统，tcpdump 就用到它。libpcap 大约由 25 个函数组成。我们将不单独地逐一介绍这些函数，而是在下节的完整例子中给出常用函数的真正使用。所有函数都以 pcap_ 前缀开头。详见 pcap 手册页面。

该函数库可以 ftp://ftp.ee.lbl.gov/libpcap.tar.z 公开取得。

26.6　检查 UDP 的校验和字段

下面让我们来开发一个例子，这个例子使用 UDP 数据报向名字服务器发送一个 DNS 查询，然后使用 libpcap 读取应答。程序的目的是检查名字服务器是否计算 UDP 校验和。在 IPv4 里，计算 UDP 校验和是可选的。绝大多数目前的系统缺省是计算校验和的，不过一些老系统（如 SunOS 4.1X）缺省禁止这个功能。目前所有系统，特别是运行名字服务器的系统，都应该使能 UDP 校验和，否则，错误的数据报就有可能破坏服务器的数据库。

使能和禁止 UDP 校验和一般是在系统范围的基础上做的，如 TCPv1 的附录 E 所述。

我们构造自己的 UDP 数据报（DNS 查询），并把它写到原始套接口。实际上只需要通过普通 UDP 套接口就可以发送查询，但我们想展示如何使用 IP_HDRINCL 套接口选项构造一个完整的 IP 数据报。但是当我们从普通 UDP 套接口读取数据时，我们无法获得校验和，使用原始套接口也无法读取 TCP 或 UDP 分组（25.4 节）。唯一的办法就是使用分组捕获机制获取完整的含有名字服务器应答的 UDP 数据报。我们检查 UDP 头部内的校验和字段，如果校验和为 0，则服务器没有使能 UDP 校验和。

图 26.3 总结了本程序的的操作。我们向原始套接口写自己构造的 UDP 数据报，并使用 libpcap 读取应答。需要注意的是：UDP 模块也会读取我们所发请求的应答，然后响应一个 ICMP 端口不可达错误给名字服务器（我们的 UDP 分组是自己构造的，未通过 UDP 模块，它当然不知道端口号）。名字服务器将忽略这个 ICMP 错误。通过了解这种情况，我们还应注意到，用以上这种测试 UDP 的方式来测试 TCP 是比较困难的，虽然我们可像发送 UDP

数据报那样简单地发送 TCP 分节，但是对于我们构造的 TCP 分节所引发的应答，我们的 TCP 通常会响应一个 RST，其目的地就是应答分节的发送者。

> 绕过以上问题的一个方法是发送以所连接子网上某个未被使用的 IP 地址为源地址的 TCP 分节。给发送主机增加一个该 IP 地址的 ARP 表项，这样发送主机就会回答对这个新地址的 ARP 请求。然而，不要把该 IP 地址配置成一个 IP 别名。这将导致发送主机上的 IP 协议栈丢弃所接收到的目的地址为这个新 IP 地址的分组，前提是发送主机并不用作路由器。

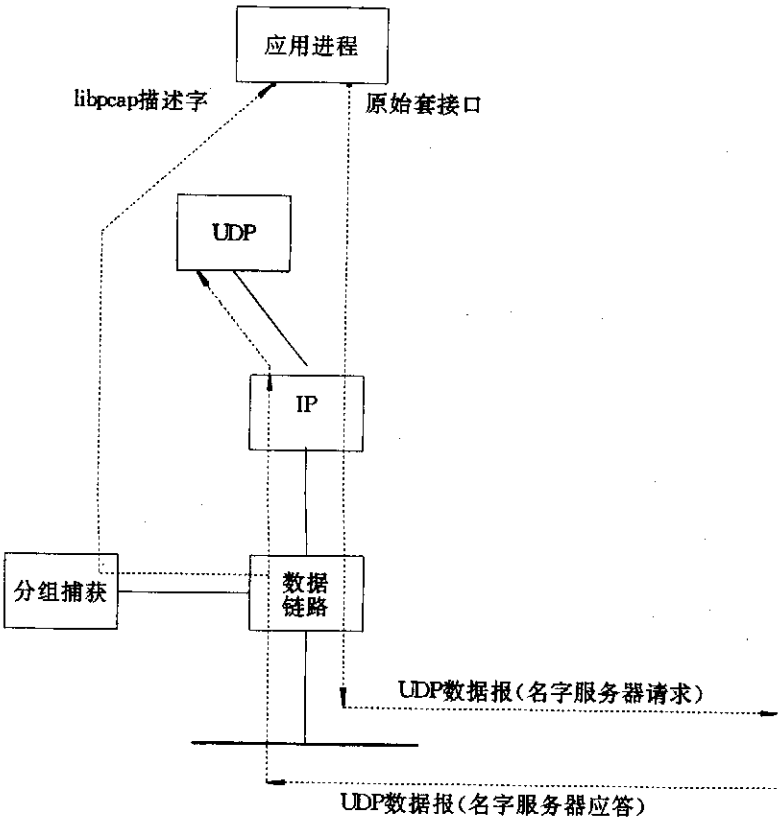


图 26.3 检查名字服务器是否计算 UDP 校验和的应用程序

图 26.4 是本程序中使用函数的汇总。

图 26.5 为头文件 udpcksum.h，它内部还包括了我们自定义的基本头文件 unp.h 以及一些系统头文件，它们是访问 IP 及 UDP 分组头部的结构定义所必需的。

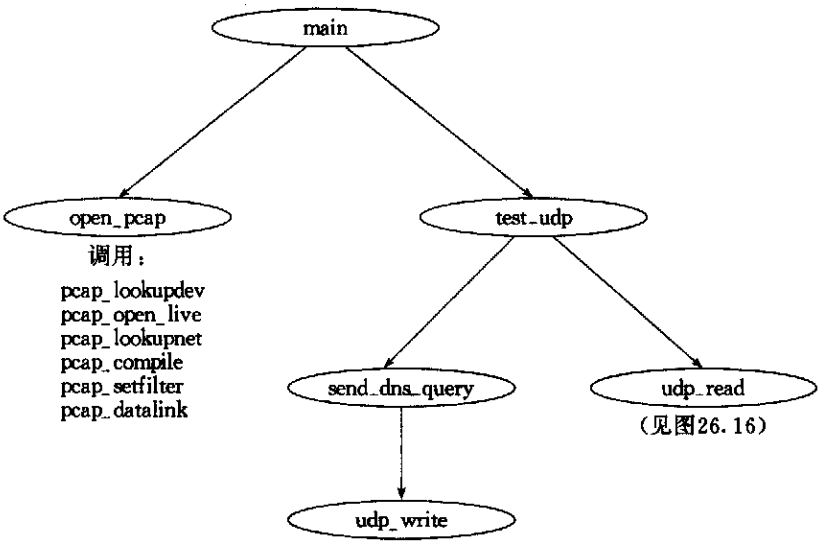


图 26.4　udpcksum 程序中的函数

```
1  #include    "unp.h"
2  #include    <pcap.h>
3  #include    <netinet/in_systm.h>     /* required for ip.h */
4  #include    <netinet/in.h>
5  #include    <netinet/ip.h>
6  #include    <netinet/ip_var.h>
7  #include    <netinet/udp.h>
8  #include    <netinet/udp_var.h>
9  #include    <net/if.h>
10 #include    <netinet/if_ether.h>
11 #define     LOCALPORT   "39123"      /* source port (default) */
12 #define     TTL_OUT     64           /* outgoing TTL */
13                          /* declare global variables */
14 extern struct sockaddr  *dest, *local;
15 extern socklen_t destlen, locallen;
16 extern int datalink;
17 extern char  *device;
18 extern pcap_t  *pd;
19 extern int rawfd;
20 extern int snaplen;
21 extern int verbose;
22 extern int zerosum;
23                          /* function prototypes */
24 void      cleanup(int);
25 char      *next_pcap(int *);
26 void      open_pcap(void);
27 void      test_udp(void);
28 void      udp_write(char *, int);
29 struct udpiphdr  *udp_read(void);
```

图 26.5　udpcksum.h 头文件[udpcksum/udpcksum.h]

第3～10行 处理IP和UDP头部字段需要额外的因特网头文件。

第11～29行 定义程序中所用的函数原型及全程变量。

图26.6为main函数的第一部分。

```
1 #include    "udpcksum.h"
2             /* define global variables */
3 struct sockaddr    *dest, *local;
4 socklen_t          destlen, locallen;
5 int          datalink;          /* from pcap_datalink(), in <net/bpf.h> */
6 char         *device;           /* pcap device */
7 int          fddipad;           /* HACK; for libpcap if FDDI defined */
8 pcap_t       *pd;               /* packet capture struct pointer */
9 int          rawfd;             /* raw socket to write on */
10 int         snaplen = 200;     /* amount of data to capture */
11 int         verbose;
12 int         zerosum;           /* send UDP query with no checksum */
13 static void usage(const char *);
14 int
15 main(int argc, char *argv[])
16 {
17     int                 c, on=1;
18     char              *ptr, localname[1024], *localport;
19     struct addrinfo    *aip;
20     if (argc < 2)
21         usage("");
22     /*
23      * Need local IP address for source IP address for UDP datagrams.
24      * Can't specify 0 and let IP choose, as we need to know it for
25      * the pseudo-header to calculate the UDP checksum.
26      * Both localname and localport can be overridden by -l option.
27      */
28     if (gethostname(localname, sizeof(localname)) < 0)
29         err_sys("gethostname error");
30     localport = LOCALPORT;
```

图26.6 main函数:定义[udpcksum/main.c]

检查命令行参数个数

第20～21行 本程序至少需要两个参数:运行DNS服务器的主机名或IP地址以及服务器的服务名(domain)或端口号(53)。这里我们就不给出usage函数了,它的作用是输出命令使用格式,然后终止。

获取本地主机名

第28～30行 源IP地址是UDP伪头部(我们很快要谈到)的一部分,而伪头部是用来计算UDP校验和的。现在我们需要构造自己的IP头部和UDP头部,为此在写UDP数据报时我们必须知道源IP地址,我们不能让它等于0,然后由IP模块来选择地址。于是我们调用gethostname来获取主机名。此外,我们把源端口号缺省成udpcksum.h中定义的值。

图 26.7 是 main 函数的下一部分，它处理命令行参数。

```
31    opterr = 0;      /* don't want getopt() writing to stderr */
32    while ( (c = getopt(argc, argv, "0i:l:v")) != -1) {
33        switch (c) {
34        case '0':
35            zerosum = 1;
36            break;
37        case 'i':
38            device = optarg;            /* pcap device */
39            break;
40        case 'l':            /* local IP address and port#: a.b.c.d.p */
41            if ( (ptr = strrchr(optarg, '.')) == NULL)
42                usage("invalid -l option");
43            *ptr++ = 0;                 /* null replaces final period */
44            localport = ptr;            /* service name or port number */
45            strncpy(localname, optarg, sizeof(localname));
46            break;
47        case 'v':
48            verbose = 1;
49            break;
50        case '?':
51            usage("unrecognized option");
52        }
53    }
```

图 26.7　main 函数：处理命令行参数[udpcksum/main.c]

处理命令行参数

第 31～36 行　我们调用 getopt 来处理命令行参数，-0 表示在发送 UDP 查询时不置校验和，用来查看对于有无校验和两种情况，服务器会有什么不同的处理。

第 37～39 行　-i 选项用于指定接收服务器应答的接口。如果我们不直接给出的话，libpcap 将自动选择一个，不过在多宿主机上，这种自动选择未必正确。从分组捕获设备读与从普通套接口读的差别之一是：使用套接口的话我们可以通配本地地址，从而允许我们接收到达任意接口的分组；但使用分组捕获设备时，我们接收只到达单个接口的分组。

> 有一点需要指出，Linux 的 SOCK_PACKET 并未限制只从一个设备进行数据链路捕获。不过 libpcap 通过-i 选项或缺省值提供了这样的过滤。

第 40～46 行　-l 选项指定源 IP 地址及端口号。服务名或端口号是作为本选项参数的字符串中跟在最后一个“.”号后面的部分，而源 IP 地址则是该字符串中在最后一个“.”号前的部分。

图 26.8 是 main 函数的最后一部分。

```
54    if (optind != argc-2)
55        usage("missing <host> and/or <serv>");
56        /* convert destination name and service */
57    aip = host_serv(argv[optind], argv[optind+1], AF_INET, SOCK_DGRAM);
```

```
58      dest = aip->ai_addr;          /* don't freeaddrinfo() */
59      destlen = aip->ai_addrlen;
60          /* convert local name and service */
61      aip = host_serv(localname, localport, AF_INET, SOCK_DGRAM);
62      local = aip->ai_addr;         /* don't freeaddrinfo() */
63      locallen = aip->ai_addrlen;
64      /*
65       * Need a raw socket to write our own IP datagrams to.
66       * Process must have superuser privileges to create this socket.
67       * Also must set IP_HDRINCL so we can write our own IP headers.
68       */
69      rawfd = Socket(dest->sa_family, SOCK_RAW, 0);
70      Setsockopt(rawfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
71      open_pcap();          /* open packet capture device */
72      setuid(getuid()); /* don't need superuser privileges any more */
73      Signal(SIGTERM, cleanup);
74      Signal(SIGINT, cleanup);
75      Signal(SIGHUP, cleanup);
76      test_udp();
77      cleanup(0);
78  }
```

图 26.8 main 函数:转换主机名和服务名,创建套接口[udpcksum/main.c]

先处理目的主机名和端口,再处理本地主机名和端口

第 54～63 行 我们处理最后两个命令行参数:目的主机名和服务名。我们调用 host_serv 将其转换为套接口地址结构的形式,并将指向该结构的指针存入 dest 中。接着,我们对本地主机名和端口号作同样的转换工作,并将指向相应套接口地址结构的指针存入 local 中。

创建原始套接口,并打开分组捕获设备

第 64～71 行 我们创建一个原始套接口并打开 IP_HDRINCL 套接口选项,这样我们可以构造完整的 IP 数据报,包括 IP 头部。open_pcap 函数用于打开分组捕获设备,很快我们将介绍该函数。

改变权限并建立信号处理程序

第 72～75 行 我们需要超级用户权限来创建原始套接口;一般而言,我们在打开分组捕获设备时也需要超级用户权限,不过这依赖于实现。例如对于 BPF,管理员可以根据需要设置/dev/bpf 设备的访问权限。假设这是一个需经设置用户 ID 的程序(也就是说程序文件具有 SUID 访问权限),现在我们放弃这个额外权限。如果这个进程拥有超级用户权限,调用 setuid 将把它的实际用户 ID、有效用户 ID 和保存的经设置用户 ID 设置成我们的实际用户 ID(getuid)。我们还为预防用户中途终止程序运行而建立有关的信号处理程序。

测试与清理

第 76～77 行 函数 test_udp(图 26.10)进行测试后返回。函数 cleanup 输出来自分组捕

获函数库的统计结果后终止进程。

图 26.9 为 open_pcap 函数，该函数在 main 函数内用来打开分组捕获设备。

选择分组捕获设备

第 10～14 行　如果未通过 -i 命令行选项给出分组捕获设备，则 open_pcap 将通过 pcap_lookupdev 函数来寻找一个设备。该函数发出 SIOCGIFCONF ioctl 请求取得所有设备的信息，并从中选择一个处于工作状态的编号最小的设备，不过回馈接口除外。很多 pcap 库函数都在出错时填写一个出错消息串，传给 pcap_lookupdev 的唯一参数就是一个用于填写该字符串的字符数组。

打开设备

第 15～17 行　pcap_open_live 打开设备。函数名中的"live"指一个确确实实被打开的设备，而不是一个含有先前保存的分组的保存文件。该函数第一个参数是设备名，第二个参数是从每个分组获取的数据长度(snaplen，我们在图 26.6 中初始化成 200)，第三个参数是混杂标志，第四个参数是以毫秒为单位的超时时限，第五个参数是出错消息串字符指针。

如果我们设置了混杂标志则接口将进入混杂模式，从而能够接收出现在电缆上的所有分组，tcpdump 通常就使用该模式。而对于我们的例子而言，DNS 服务器的应答是会发送到我们的主机的(出就是说不必进入混杂模式)。

超时参数是指读超时。要是每收到一个分组就向应用进程返送，那会引发从内核心到应用进程的各个分组的大量拷贝，效率是比较低的。因此我们仅当分组捕获设备的读缓冲区已满时或超时时，才返送数据。如果把超时时限设为 0，那么一接收到一个分组就立即返送。

获取网络地址与子网掩码

第 18～23 行　pcap_lookupnet 返回分组捕获设备的网络地址和子网掩码。因为分组过滤器需要确定一个 IP 地址是否为一个子网广播地址，因此接下来调用的 pcap_compile 函数必须指定子网掩码。

编译分组过滤器

第 24～30 行　pcap_compile 将一个过滤器字符串(我们存储在 cmd 数组内)编译为一个过滤器程序(存储在 fcode 内)。它将选择我们希望接收的分组。

```
1 #include    "udpcksum.h"
2 #define     CMD         "udp and src host %s and src port %d"
3 void
4 open_pcap(void)
5 {
6     uint32_t              localnet, netmask;
7     char                  cmd[MAXLINE], errbuf[PCAP_ERRBUF_SIZE],
8                             str1[INET_ADDRSTRLEN], str2[INET_ADDRSTRLEN];
9     struct bpf_program    fcode;
10    if (device == NULL) {
11        if ( (device = pcap_lookupdev(errbuf)) == NULL)
12            err_quit("pcap_lookup: %s", errbuf);
13    }
14    printf("device = %s\n", device);
```

```
15          /* hardcode: promisc=0, to_ms=500 */
16      if ( (pd = pcap_open_live(device, snaplen, 0, 500, errbuf)) == NULL)
17          err_quit("pcap_open_live: %s", errbuf);
18      if (pcap_lookupnet(device, &localnet, &netmask, errbuf) < 0)
19          err_quit("pcap_lookupnet: %s", errbuf);
20      if (verbose)
21          printf("localnet = %s, netmask = %s\n",
22                 inet_ntop(AF_INET, &localnet, str1, sizeof(str1)),
23                 inet_ntop(AF_INET, &netmask, str2, sizeof(str2)));
24      snprintf(cmd, sizeof(cmd), CMD,
25               Sock_ntop_host(dest, destlen),
26               ntohs(sock_get_port(dest, destlen)));
27      if (verbose)
28          printf("cmd = %s\n", cmd);
29      if (pcap_compile(pd, &fcode, cmd, 0, netmask) < 0)
30          err_quit("pcap_compile: %s", pcap_geterr(pd));
31      if (pcap_setfilter(pd, &fcode) < 0)
32          err_quit("pcap_setfilter: %s", pcap_geterr(pd));
33      if ( (datalink = pcap_datalink(pd)) < 0)
34          err_quit("pcap_datalink: %s", pcap_geterr(pd));
35      if (verbose)
36          printf("datalink = %d\n", datalink);
37  }
```

图 26.9 open_pcap 函数:打开并初始化分组捕获设备[udpcksum/pcap.c]

装入过滤器程序

第31～32行 pcap_setfilter 将我们刚才编译获得的过滤器程序装入分组捕获设备。这将初始化我们用该过滤器选择的分组的捕获。

确定数据链路类型

第33～36行 pcap_datalink 返回分组捕获设备的数据链路类型。这么做是为了确定数据链路头部的大小,它们将出现在我们所读的每个分组的起始处(图 26.14)。

在调用 open_pcap 之后,main 函数调用 test_udp(图 26.10)。该函数发送一个 DNS 查询,并读取服务器的应答。

```
47  void
48  test_udp(void)
49  {
50      volatile int    nsent = 0, timeout = 3;
51      struct udpiphdr    *ui;
52      Signal(SIGALRM, sig_alrm);
53      if (sigsetjmp(jmpbuf, 1)) {
54          if (nsent >= 3)
55              err_quit("no response");
56          printf("timeout\n");
57          timeout *= 2;       /* exponential backoff: 3, 6, 12 */
58      }
59      canjump = 1;        /* siglongjmp is now OK */
```

```
60      send_dns_query();
61      nsent++;
62      alarm(timeout);
63      ui = udp_read();
64      canjump = 0;
65      alarm(0);
66      if (ui->ui_sum == 0)
67          printf("UDP checksums off\n");
68      else
69          printf("UDP checksums on\n");
70      if (verbose)
71          printf("recevied UDP checksum = %x\n", ui->ui_sum);
72  }
```

图 26.10　test_udp 函数：发送请求并读取应答[udpcksum/udpcksum.c]

volatile 变量

第 50 行　具体的实现允许在 siglongjmp 之后将自动变量恢复成调用 sigsetjmp 时的值(APUE 第 178 页)，而我们则希望自动变量 nsent 和 timeout 在从信号处理程序 siglongjmp 到本函数之后仍维持它们的值，给它们加上 volatile 声明就是为了保证做到这一点。

建立信号处理程序以及跳转缓冲区

第 52～53 行　调用 signal 建立一个 SIGALRM 信号处理程序，再调用 sigsetjmp 给 siglongjmp 准备一个跳转缓冲区(APUE 的 10.15 节对这两个函数有详细的描述)。sigsetjmp 的第二个参数为 1，它通知 sigsetjmp 保存当前信号掩码，因为我们将从信号处理程序中调用 siglongjmp。

处理 siglongjmp

第 54～48 行　只有当从信号处理程序中调用 siglongjmp 之后，才会执行这段代码。它表示发生了超时，也就是说我们发出了一个请求，但未收到应答。如果我们已经连续发送了三个请求而未获任何应答，那么终止执行。否则，我们输出一行信息，把超时时限加倍，并再次等待任何应答。这是指数后退，我们在 20.5 节提到过。可以算一下，如果始终没有应答的话，三次超时应该分别为 3、6 和 12 秒。

我们在这个例子里使用 sigsetjmp 和 siglongjmp，而不是仅仅捕获 EINTR(如图 13.1)，其原因在于分组捕获函数库 libpcap 的读函数(它们由 udp_read 函数调用)在返回 EINTR 错误时将重新启动一次读操作。由于我们不想为了返回 EINTR 而修改库函数，我们就得使用自己的方式捕获 SIGALRM 信号，并执行一个非本地的长跳转，将控制从库代码转到我们自己的代码里来。

发送 DNS 查询并读取应答

第 60～65 行　send_dns_query(图 26.12)给名字服务器发送一个 DNS 查询。udp_read(图 26.14)读取应答。接着，我们使用 alarm 来防止读操作永久阻塞。当超时时限到达时，将会产生一个 SIGALRM 信号，于是我们的信号处理程序调用 siglongjmp。

检查收到的 UDP 分组的校验和

第 66～71 行　如果收到的 UDP 分组的校验和等于 0，则表示该服务器未计算并发送

校验和。

图26.11是我们的信号处理程序sig_alrm，用于处理SIGALRM信号。

```
1 #include    "udpcksum.h"
2 #include    <setjmp.h>
3 static sigjmp_buf jmpbuf;
4 static int canjump;
5 void
6 sig_alrm(int signo)
7 {
8     if (canjump == 0)
9         return;
10    siglongjmp(jmpbuf, 1);
11 }
```

图26.11 sig_alrm函数：处理SIGALRM信号[udpcksum/udpcksum.c]

第8～10行 在图26.10里，当跳转缓冲区被sigsetjmp初始化之后，canjump标志被设置。如果该标志已设置，siglongjmp将使控制流转向，仿佛图26.10里的sigsetjmp返回1一样。

图26.12为send_dns_query函数，它构造了一个应用数据(DNS查询)，并向名字服务器发送这个UDP查询。

```
16 void
17 send_dns_query(void)
18 {
19     size_t      nbytes;
20     char        buf[sizeof(struct udpiphdr) + 100], *ptr;
21     short       one;
22     ptr = buf + sizeof(struct udpiphdr);  /* leave room for IP/UDP headers */
23     *((u_short *) ptr) = htons(1234);     /* identification */
24     ptr += 2;
25     *((u_short *) ptr) = htons(0x0);      /* flags */
26     ptr += 2;
27     *((u_short *) ptr) = htons(1);        /* #questions */
28     ptr += 2;
29     *((u_short *) ptr) = 0;               /* #answer RRs */
30     ptr += 2;
31     *((u_short *) ptr) = 0;               /* #authority RRs */
32     ptr += 2;
33     *((u_short *) ptr) = 0;               /* #additional RRs */
34     ptr += 2;
35     memcpy(ptr, "\001a\014root-servers\003net\000", 20);
36     ptr += 20;
37     one = htons(1);
38     memcpy(ptr, &one, 2);                 /* query type = A */
39     ptr += 2;
40     memcpy(ptr, &one, 2);                 /* query class = 1 (IP addr) */
41     ptr += 2;
42     nbytes = 36;
```

```
43      udp_write(buf, nbytes);
44      if (verbose)
45          printf("sent: %d bytes of data\n", nbytes);
46  }
```

图 26.12　send_dns_query 函数：向 DNS 服务器发送一个查询[udpcksum/udpcksum.c]

初始化缓冲区指针

第 20～22 行　缓冲区 buf 足以存放 20 字节的 IP 头部、8 字节的 UDP 头部以及 100 字节的用户数据。经过初始化，指针 ptr 指向用户数据区的第一个字节。

构造 DNS 查询

第 23～24 行　要详细了解本函数所构造 UDP 数据报的内容，需要知道 DNS 消息格式。可参见 TCPv1 的 14.3 节。这里我们设置标识符字段为 1234，标志为 0，问题数为 1，然后把回答资源记录数、权威资源记录数和附加资源记录数都设置为 0。

> <arpa/nameser.h>头文件定义了一个 HEADER 数据类型，用来填写查询头部的前 12 个字节。在我们这个简单的查询中，自己填写这 12 个字节也挺方便。

第 35～41 行　我们接着构造本消息中头部之后的单个问题：查询主机 a.root-servers.net 的 IP 地址。这个域名用 20 字节存储，包含 4 个标签：单字节的标签 a、12 字节的标签 root-servers(\014 是一个八进制常值)、3 字节的标签 net 以及长度为 0 的根标签。查询类型为 1(称为 A 查询)，查询类别也为 1。

写 UDP 数据报

第 42～45 行　这个消息共由 36 个字节组成：8 个双字节字段和一个 20 字节长的域名。我们调用 udp_write 函数构造 UDP 和 IP 头部，并通过原始套接口将 IP 数据报发送出去。

图 26.13 给出了函数 udp_write，它构造 IP 和 UDP 头部并把数据报写入原始套接口。

初始化分组头部指针

第 11～13 行　ip 指向 IP 头部(一个 ip 结构)的开始，ui 指向同一位置，但它的结构 udpiphdr 包含 IP 头部和 UDP 头部。

更新长度

第 14～17 行　ui_len 是 UDP 长度，等于用户数据长度加上 UDP 头部长度(8 个字节)。userlen(UDP 头部后跟着的用户数据字节数)加上 28(20 字节的 IP 头部和 8 字节的 UDP 头部)就是整个 IP 数据报的大小。

填充 UDP 头部并计算 UDP 校验和

第 18～35 行　计算 UDP 校验和不仅要涉及 UDP 头部和 UDP 数据，还要涉及 IP 头部的一部分字段，这些出自 IP 头部的额外字段构成所谓的伪头部(pseudoheader)。引入伪头部对于如果校验和正确那么数据报是递送给了正确的主机和协议代码这一点提供了额外的验证。这些语句初始化构成伪头部的 IP 字段。当然这些语句有些难懂，不过 TCPv2 的 23.6 节

有相应的解释。最终的结果是:如果 zerosum 标志(对应-0 命令行参数)未设置,UDP 校验和就存入 ui_sum 成员中。

如果计算得到校验和为 0,则改为存入 0xffff。虽然在反码算术(ones-complement arithmetic)中它们是相同的,但在 UDP 中,如果校验和为 0,则表示未设置校验和。回想一下图 25.13,我们并未检查校验和是否为 0,因为 ICMPv4 的校验和是必需的,其值为 0 并不指示没有校验和。

> 我们注意到 Solaris 2.x(x<6)有一个跟校验和相关的缺陷。在设置 IP_HDRINCL 套接口选项情况下,在原始套接口上发送 TCP 分节或 UDP 数据报时,内核会计算校验和,但我们必须把 ui_sum 成员设置成 UDP 长度。

填写 IP 头部

第 36~49 行 由于使用了 IP_HDRINCL 套接口选项,我们必须填写 IP 头部内的大部分字段(25.3 节已有讨论)。这里,我们设置标识符字段为 0(ip_id),它通知 IP 模块设置本字段。同样,IP 模块将计算 IP 头部校验和。sendto 用来发送 IP 数据报。

```
 6 void
 7 udp_write(char *buf, int userlen)
 8 {
 9     struct udpiphdr *ui;
10     struct ip *ip;

11         /* Fill in and checksum UDP header */
12     ip = (struct ip *) buf;
13     ui = (struct udpiphdr *) buf;
14             /* add 8 to userlen for pseudo-header length */
15     ui->ui_len = htons((u_short) (sizeof(struct udphdr) + userlen));
16             /* then add 28 for IP datagram length */
17     userlen += sizeof(struct udpiphdr);

18     ui->ui_next = 0;
19     ui->ui_prev = 0;
20     ui->ui_x1 = 0;
21     ui->ui_pr = IPPROTO_UDP;
22     ui->ui_src.s_addr = ((struct sockaddr_in *) local)->sin_addr.s_addr;
23     ui->ui_dst.s_addr = ((struct sockaddr_in *) dest)->sin_addr.s_addr;
24     ui->ui_sport = ((struct sockaddr_in *) local)->sin_port;
25     ui->ui_dport = ((struct sockaddr_in *) dest)->sin_port;
26     ui->ui_ulen = ui->ui_len;
27     ui->ui_sum = 0;
28     if (zerosum == 0) {
29 # ifdef notdef          /* change to ifndef for Solaris 2.x, x < 6 */
30         if ( (ui->ui_sum = in_cksum((u_short *) ui, userlen)) == 0)
31             ui->ui_sum = 0xffff;
32 # else
33         ui->ui_sum = ui->ui_len;
34 # endif
35     }

36         /* Fill in rest of IP header; */
37         /* ip_output() calcuates & stores IP header checksum */
```

```
38      ip->ip_v = IPVERSION;
39      ip->ip_hl = sizeof(struct ip) >> 2;
40      ip->ip_tos = 0;
41  #ifdef  linux
42      ip->ip_len = htons(userlen);    /* network byte order */
43  #else
44      ip->ip_len = userlen;           /* host byte order */
45  #endif
46      ip->ip_id = 0;                  /* let IP set this */
47      ip->ip_off = 0;                 /* frag offset, MF and DF flags */
48      ip->ip_ttl = TTL_OUT;
49      Sendto(rawfd, buf, userlen, 0, dest, destlen);
50  }
```

图 26.13　udp_write 函数：构造 UDP 头部和 IP 头部，并向原始套接口发送 IP 数据报[udpcksum/udpwrite.c]

下一个要说明的函数是图 26.13 给出的 udp_read，它在图 26.10 中被调用。

```
 7  struct udpiphdr *
 8  udp_read(void)
 9  {
10      int           len;
11      char          *ptr;
12      struct ether_header   *eptr;
13      for ( ; ; ) {
14          ptr = next_pcap(&len);
15          switch (datalink) {
16          case DLT_NULL:      /* loopback header = 4 bytes */
17              return(udp_check(ptr+4, len-4));
18          case DLT_EN10MB:
19              eptr = (struct ether_header *) ptr;
20              if (ntohs(eptr->ether_type) != ETHERTYPE_IP)
21                  err_quit("Ethernet type %x not IP", ntohs(eptr->ether_type));
22              return(udp_check(ptr+14, len-14));
23          case DLT_SLIP:      /* SLIP header = 24 bytes */
24              return(udp_check(ptr+24, len-24));
25          case DLT_PPP:       /* PPP header = 24 bytes */
26              return(udp_check(ptr+24, len-24));
27          default:
28              err_quit("unsupported datalink (%d)", datalink);
29          }
30      }
31  }
```

图 26.14　udp_read 函数：从分组捕获设备读取下一个分组[udpcksum/udpread.c]

第 14～29 行　函数 next_pcap(图 26.15)返回一个从分组捕获设备获得的分组。由于数据链路头部取决于实际的设备类型，这里我们根据函数 pcap_datalink 返回的不同值分别跳转处理。

TCPv2的图31.9展示了函数udp_read中出现的4、14和24这几个神奇的偏移量。SLIP和PPP对应的24字节偏移量适用于BSD/OS 2.1。

尽管名字DLT_EN10MB中有"10M"的限字词，实际上100M以太网也使用相同的数据链路类型。

函数udp_check(图26.17)对分组进行检查，并验证IP及UDP头部内的字段。

图26.15给出next_pcap函数，该函数从分组捕获设备返回下一个分组。

```
38 char *
39 next_pcap(int *len)
40 {
41     char    *ptr;
42     struct pcap_pkthdr hdr;

43         /* keep looping until packet ready */
44     while ((ptr = (char *) pcap_next(pd, &hdr)) == NULL);

45     *len = hdr.caplen;      /* captured length */
46     return(ptr);
47 }
```

图26.15 next_pcap函数：返回下一个分组[udpcksum/pcap.c]

第43～44行 调用pcap_next获取下一个分组。该函数通过返回值返回一个指向该分组的指针，同时它的第二个参数所指向的pcap_pkthdr结构也在返回时被填写：

```
struct pcap_pkthdr {
  struct timeval ts; /* timestamp */
  bpf_u_int32 caplen; /* length of portion captured */
  bpf_u_int32 len; /* length this packet (off wire) */
};
```

时间戳(ts)记录的是分组捕获时间，而不是递送时间，后者总要迟于前者。caplen为实际获取的(部分)分组长度。(回忆一下，我们在图26.6中将变量snaplen设为200，而它又是图26.9中的pcap_open_live函数的第二个参数。)分组捕获是用来获取分组头部而不是分组内数据的，所以无需获取整个分组。len是分组在电缆上的实际长度，caplen总小于等于len。

第45～46行 分组捕获长度通过指针参数len返回给函数调用者，指向分组的指针通过函数的返回值返回。记住，返回的指针指向的是数据链路头部，它可能是以太网帧(包含14个字节的以太网头部)，也可能是回馈接口帧(包含4字节的伪链路头部)。

如果我们看一下pcap_next的实现，可以发现它是由多个函数协同完成的，见图26.16。我们的应用程序所调用的pcap_函数，其中一部分是设备无关的，而另一部分则是设备相关的。譬如所展示的pcap_read在BPF下调用read，在DLPI下调用getmsg，在Linux下调用recvfrom。

图26.17为函数udp_check，它检验IP及UDP头部内的几个字段。这是必须的，因为与原始套接口不同，当我们通过分组捕获设备获得这些分组时，IP层还未见到它们。

第44～61行 分组长度必须包括IP和UDP头部。IP版本、IP包头长度以及IP头部

校验和都要检查，如果协议字段指示该分组为 UDP 数据报，那么本函数将返回指向组合的 IP/UDP 头部的指针。由于我们在图 26.9 中调用 pcap_setfilter 时设置的过滤器应保证无其他类型分组被捕获，一旦发现其他类型的分组，程序就终止。

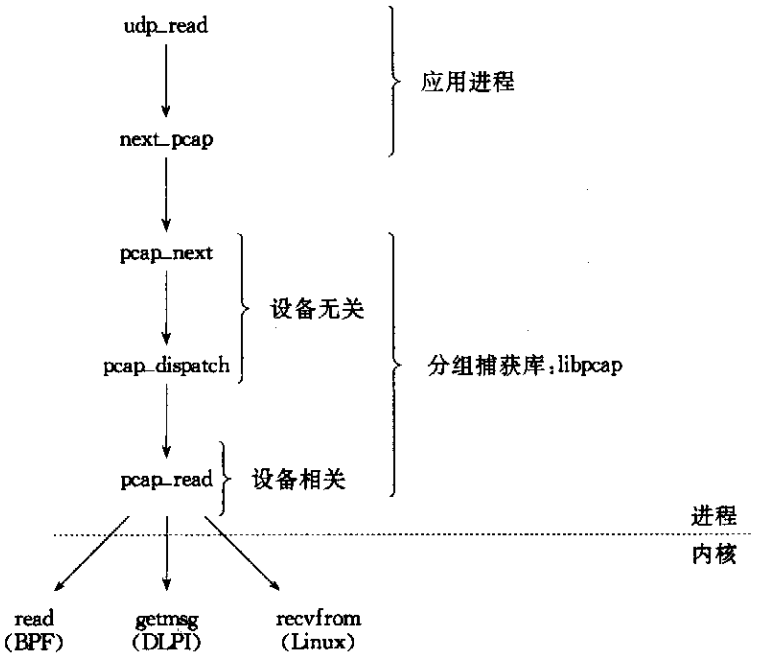


图 26.16　分组捕获库内与读取相关的函数调用

```
38 struct udpiphdr *
39 udp_check(char *ptr, int len)
40 {
41     int     hlen;
42     struct ip *ip;
43     struct udpiphdr *ui;
44     if (len < sizeof(struct ip) + sizeof(struct udphdr))
45         err_quit("len = %d", len);
46         /* minimal verification of IP header */
47     ip = (struct ip *) ptr;
48     if (ip->ip_v != IPVERSION)
49         err_quit("ip_v = %d", ip->ip_v);
50     hlen = ip->ip_hl << 2;
51     if (hlen < sizeof(struct ip))
52         err_quit("ip_hl = %d", ip->ip_hl);
53     if (len < hlen + sizeof(struct udphdr))
54         err_quit("len = %d, hlen = %d", len, hlen);
55     if ( (ip->ip_sum = in_cksum((u_short *) ip, hlen)) != 0)
56         err_quit("ip checksum error");
57     if (ip->ip_p == IPPROTO_UDP) {
58         ui = (struct udpiphdr *) ip;
59         return(ui);
```

```
60      } else
61          err_quit("not a UDP packet");
62  }
```

图 26.17 udp_check 函数:检验 IP 头部和 UDP 头部[udpcksum/udpread.c]

```
 2 void
 3 cleanup(int signo)
 4 {
 5     struct pcap_stat  stat;

 6     fflush(stdout);
 7     putc('\n', stdout);

 8     if (verbose) {
 9         if (pcap_stats(pd, &stat) < 0)
10             err_quit("pcap_stats: %s\n", pcap_geterr(pd));
11         printf("%d packets received by filter\n", stat.ps_recv);
12         printf("%d packets dropped by kernel\n", stat.ps_drop);
13     }

14     exit(0);
15 }
```

图 26.18 cleanup 函数[udpcksum/cleanup.c]

图 26.18 为函数 cleanup,该函数在程序即将终止时由 main 函数调用。同时在用户中断程序时(图 26.8)作为信号处理程序。

获取分组捕获统计数据并输出

第 8～13 行 使用 pcap_stats 获取统计信息,输出以下两项:由过滤器收到的分组的总数、由内核丢弃的分组的数目。

例子

首先我们使用 -0 和 -v 命令行选项来检查名字服务器对无校验和数据报的响应。

```
solaris # udpcksum -0  -v connix.com domain
device = le0
localnet = 206.62.226.32, netmask = 255.255.255.224
cmd = udp and src host 198.69.10.4 and src port 53
datalink = 1
sent: 36 bytes of data
UDP checksums on
recevied UDP checksum = ad39

2 packets received by filter
0 packets dropped by kernel
```

接着,我们向一个不计算 UDP 校验和的名字服务器发送请求。

```
solaris # udpcksum -v gw.pacbell.com domain
device = le0
localnet = 206.62.226.32, netmask = 255.255.255.224
cmd = udp and src host 192.150.170.2 and src port 53
datalink = 1
sent: 36 bytes of data
```

```
UDP checksums off
recevied UDP checksum = 0
1 packets received by filter
0 packets dropped by kernel
```

26.7　小　结

通过原始套接口我们可以读写内核不能识别的 IP 数据报。而通过数据链路层访问，我们可以把这种能力扩展到读写任意类型的数据链路帧，而不光是 IP 数据报。tcpdump 也许是直接访问数据链路层的最常用程序。

不同操作系统的数据链路层的访问方式是不同的，我们已看过源自 Berkeley 的 BPF、SVR4 的 DLPI 以及 Linux 的 SOCK_PACKET。不过，如果使用免费可得的分组捕获函数库 libpcap，我们可以不管这些区别，写出可移植的代码。

26.8　习　题

26.1　图 26.11 中的 canjump 标志有什么用？

26.2　对于我们的 udpcksum 程序，常见出错应答是 ICMP 端口不可达（目的主机没有运行名字服务器）或 ICMP 主机不可达。其实，这些 ICMP 错误已经是对我们的 DNS 查询的应答了，图 26.10 中的 udp_read 无需进一步等待超时。请修改一下程序，以捕获这些 ICMP 错误。

第27章 客户-服务器程序其他设计方法

27.1 概 述

当开发一个Unix服务器程序时，我们有如下类型的进程控制可供选择：

- 本书的第一个服务器程序即图1.9是一个迭代服务器程序。这种方式并不常用，它的主要缺点是在当前客户被处理完之前，新到达的客户无法被服务。
- 图5.2则是本书的第一个并发服务器程序，它为每个客户fork一个子进程提供服务，这是Unix服务器程序通常的做法。
- 在6.8节，我们开发了使用select在一个进程内同时服务多个客户的TCP服务器程序。
- 在图23.3中，我们再次修改我们的并发服务器程序，以线程来代替进程。

本章，我们将引入两种新的并发服务器程序设计方法：

- 预先派生子进程(preforking)。服务器启动后就派生一组子进程，形成一个子进程池。每当来一个客户请求，就从进程池内取一个可用子进程为它服务。
- 预先创建线程(prethreading)。服务器启动后就创建一组线程，形成一个线程池。每个客户由池中的一个线程服务。

关于预先派生子进程和预先创建线程，本章将要讨论很多问题。譬如：如果池中的进程和线程不够怎么办？如果池中进程和线程过多怎么办？父进程与子进程及线程之间怎么同步等等。

客户程序写起来相对容易一些，因为很少有什么进程控制问题。不过，既然我们已查看了书写自己的简单回射客户程序的各种方法，我们就在27.2节对此作个总结。

本章我们将查看9种不同的服务器程序设计，并使用同一客户来作比较。我们的客户-服务器情形在Web很典型：客户向服务器发送一个小请求，服务器响应以返回给客户的数据。有几种服务器类型已经讨论得很详细了(譬如为每个客户fork一个子进程的并发服务器)，而预先派生子进程和预先创建线程的服务器则是新的，将在本章详细讨论。

我们针对每个服务器运行同一客户程序的多个实例，测量服务固定数目的客户请求所需的CPU时间，图27.1是测量结果。请注意，图中的时间测量的是仅仅用于进程控制的那部分CPU时间，而迭代服务器是我们的基准，我们把它从实际的CPU时间中减去就得到用于进程控制的那部分CPU时间，因为迭代服务器没有进程控制开销。我们在图中包含0.0的基准时间就是为了强调这一点。本章我们用进程控制CPU时间(process control CPU time)来称谓给定系统CPU时间与基准之差。

行号	服务器描述	进程控制 CPU 时间(秒,与基准之差)		
		Solaris	DUnix	BSD/OS
0	迭代服务器(测量基准,无进程控制)	0.0	0.0	0.0
1	简单并发服务器,为每个客户请求 fork 一个进程	504.2	168.9	29.6
2	预先派生子进程,每个子进程调用 accept		6.2	1.8
3	预先派生子进程,用文件上锁方式保护 accept	25.2	10.0	2.7
4	预先派生子进程,用线程互斥锁上锁方式保护 accept	21.5		
5	预先派生子进程,由父进程向子进程传递套接口描述字	36.7	10.9	6.1
6	并发服务器,为每个客户请求创建一个线程	18.7	4.7	
7	预先创建线程,用互斥锁上锁方式保护 accept	8.6	3.5	
8	预先创建线程,由主线程调用 accept	14.5	5.0	

图 27.1　本章所讨论各种服务器的耗时比较

我们分别在三个主机上运行多种服务器:sunos5(Solaris 2.5.1)、alpha(Digital Unix 4.0b)和 bsdi(BSD/OS 3.0)。注意:并非所有的服务器都能在这三种平台上运行。譬如第 2 行的服务器就不能在大多数 SVR4 主机上运行(27.7 节中有讨论),而 BSD/OS 则不支持线程(因为内核不支持)。此外,由于这三种主机系统结构不同,因此无法横向比较,而只能纵向比较;看看不同的服务器程序设计在同样的主机上会有什么样的不同结果。譬如第 7 行在 Solaris 和 Digital Unix 下都是最快的,而第 2 行是 BSD/OS 平台上最快的。

所有这些数据都是在与服务器主机处于同一子网的两台不同的主机上运行同一客户程序(图 27.4)的前提下测得的。每个客户派生 5 个子进程,对服务器开 5 个连接,因此服务器在任意时刻最多有 10 个连接。每个客户请求从服务器返回 4000 字节的数据量。预先派生子进程型或预先创建线程型服务器所预先创建的子进程数或线程数为 15 个。

子进程数或线程数	进程控制 CPU 时间(秒,与基准之差)					
	预先派生子进程,accept 无上锁保护(第 2 行)		预先派生子进程,accept 有文件上锁保护(第 3 行)			预先创建线程,accept 有互斥锁上锁保护
	DUnix	BSD/OS	Solaris	DUnix	BSD/OS	Solaris
15	6.2	1.8	25.2	10.0	2.7	8.6
30	7.8	3.5	27.3	11.2	5.6	10.0
45	8.9	5.5	29.7	13.1	8.7	19.6
60	10.1	6.9	34.2	14.3	11.2	28.6
75	11.4	8.7	39.8	16.0	13.7	29.3
90	12.6	10.9	130.1	17.6	15.5	28.6
105	13.2	12.0		19.7	17.6	30.4
120	15.7	13.5		22.0	19.2	29.4

图 27.2　过多子进程或线程对服务器 CPU 时间的影响

子进程编号或线程编号	所服务客户数									
	预先派生子进程，accept 无上锁保护（第2行）		预先派生子进程，accept 有文件上锁保护（第3行）			预先派生子进程，描述字传递（第5行）			预先创建线程，accept 有线程上锁保护（第7行）	
	DUnix	BSD/OS	Solaris	DUnix	BSD/OS	Solaris	DUnix	BSD/OS	Solaris	DUnix
0	318	333	347	335	335	1006	718	530	333	335
1	343	340	328	334	335	950	647	529	323	337
2	326	335	332	334	332	720	589	509	333	338
3	317	335	335	333	333	582	554	502	328	311
4	309	332	338	333	331	485	526	501	329	345
5	344	331	340	335	335	457	501	495	322	332
6	340	333	335	330	332	385	447	488	324	355
7	337	333	343	334	333	250	389	484	360	322
8	340	332	324	333	334	105	314	460	341	336
9	309	331	315	333	336	32	208	443	348	337
10	356	334	326	333	331	14	62	59	358	334
11	354	333	340	334	338	9	18	0	331	340
12	356	334	330	333	333	4	14	0	321	317
13	302	332	331	333	331	1	12	0	329	326
14	349	332	336	333	331	0	1	0	320	335
	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000

图27.3 15个子进程或线程中每一个所服务的客户数的分布

有些服务器程序设计涉及子进程池或线程池的创建，我们要讨论的另一个问题就是过多线程或过多子进程产生的影响。图27.2汇总了结果，我们将在恰当的章节讨论。

当我们有一个子进程集或线程集来服务客户时，还有一个待讨论的问题是：客户请求在子进程或线程池内的分布。图27.3给出了总结，我们也将在恰当的章节讨论。

27.2 TCP客户程序其他设计方法

我们已经查看了客户程序的各种设计方法，这里作一个小结，看看它们的优缺点。

1. 图5.5给出了最基本的TCP客户程序。不过这个程序有两个问题。首先，当它等待用户输入而阻塞时，无法监视网络事件（譬如对方关闭连接）。其次，它工作在停-等模式，批处理时效率极低。
2. 图6.9是一个改进了的客户程序。它改用select来同时监视用户输入与网络事件。它的问题是不能正确地处理批模式。图6.13利用shutdown函数解决了这个问题。
3. 图15.3使用非阻塞式I/O实现客户程序。
4. 我们超越单进程单线程设计的第一个客户程序是图15.9，它用fork派生一个子进程，父子进程中一个处理从服务器到客户的数据，另一个处理从客户到服务器的数据。
5. 图23.2使用双线程而不是双进程。

在15.2节，我们总结了这些方法的时间耗费。非阻塞式I/O是最快的，不过代码也比较复杂，而使用双进程或双线程则可以大大简化代码。

27.3　TCP 测试用客户程序

图 27.4 是用来测试所有服务器变种的客户程序。

第 10～12 行　每次运行客户程序时，需指定服务器的主机名或 IP 地址、服务器的端口号、客户 fork 的子进程数（关系到同时对一个服务器建立的连接数）、每个子进程向服务器发送的请求数以及每个请求要求服务器返回的数据量。

```
1 #include    "unp.h"
2 #define     MAXN   16384      /* max #bytes to request from server */
3 int
4 main(int argc, char **argv)
5 {
6     int           i, j, fd, nchildren, nloops, nbytes;
7     pid_t         pid;
8     ssize_t       n;
9     char    request[MAXLINE], reply[MAXN];
10    if (argc != 6)
11        err_quit("usage: client <hostname or IPaddr> <port> <#children> "
12                 "<#loops/child> <#bytes/request>");
13    nchildren = atoi(argv[3]);
14    nloops = atol(argv[4]);
15    nbytes = atol(argv[5]);
16    snprintf(request, sizeof(request), "%d\n", nbytes); /* newline at end */
17    for (i = 0; i < nchildren; i++) {
18        if ( (pid = Fork()) == 0) {         /* child */
19            for (j = 0; j < nloops; j++) {
20                fd = Tcp_connect(argv[1], argv[2]);
21                Write(fd, request, strlen(request));
22                if ( (n = Readn(fd, reply, nbytes)) != nbytes)
23                    err_quit("server returned %d bytes", n);
24                Close(fd);          /* TIME_WAIT on client, not server */
25            }
26            printf("child %d done\n", i);
27            exit(0);
28        }
29        /* parent loops around to fork() again */
30    }
31    while (wait(NULL) > 0)   /* now parent waits for all children */
32        ;
33    if (errno != ECHILD)
34        err_sys("wait error");
35    exit(0);
36 }
```

图 27.4　用来测试各种服务器的 TCP 客户程序[server/client.c]

第 17～30 行　父进程调用 fork 派生子进程，每个子进程与服务器建立指定次数的连接。在每条连接上，相应子进程向服务器发送一个请求行，申明服务器需要返回多少数据量，然后就从服务器读取这个数量的数据。而父进程则等待所有子进程终止。要注意的是，这里

是由客户关闭每个 TCP 连接，这样 TCP 的 TIME_WAIT 状态将出现在客户方，而不是服务器方。这是与通常的 HTTP 连接的差别之一。

本章我们使用以下命令行测试各种服务器：

```
%client 206.62.226.36 8888 5 500 4000
```

这将产生 2500 条 TCP 连接：一共 5 个子进程，每个子进程连接 500 次。每个子进程每次连接发送 5 个子节("4000\n")给服务器，而服务器为之返回 4000 字节。我们同时在两台主机上启动客户程序，最终产生 5000 条 TCP 连接，服务器同时可能会有 10 个连接。

> http://www.sgi.com/Products/WebFORCE/WebStone 下有称为 WebStone 的测试 Web 服务器的精致复杂的测试程序。对于各种可选服务器程序设计方法的一般性比较，还用不上如此精致复杂的测试程序。

下面，我们逐个分析 9 种不同的服务器程序设计。

27.4 TCP 迭代服务器程序

迭代服务器总是在完全处理了一个客户的请求后，才响应下一个客户的请求。这样的服务器程序很少，不过图 1.9 给出了一个简单的时间/日期服务器程序。

迭代服务器在本章有一个用处：提供各种服务器相互之间比较的基准。如果我们针对迭代服务器以如下方式运行客户：

```
% client 206.62.226.36 8888 1 5000 4000
```

那么同样是 5000 条连接，每条连接收发的字节数也一样。不过，由于服务器程序是迭代的，因此谈不上进程控制时间。这就给我们提供了处理如此数目的客户所需 CPU 时间的基准值，我们可以从其他服务器的 CPU 时间中减去该值，得到它们的进程控制时间。从进程控制的角度来说，迭代服务器是最快的，因为它不进行进程控制。我们在图 27.1 中对相对于该基准的各个差值进行了比较。

我们未给出我们的迭代服务器程序，不过它只需对下一节给出的并发服务器程序做少量改动。

27.5 TCP 并发服务器程序，每个客户一个子进程

传统上，并发服务器调用 fork 派生一个子进程来处理每个客户。这使得服务器可在同一时间为多个客户提供服务。唯一的限制是操作系统对同一用户同时可拥有的进程数的限制。图 5.12 就是一个并发服务器程序的例子，绝大多数 TCP 服务器程序也是这么编写的。

并发服务器的问题在于 fork 子进程时所消耗的 CPU 时间。20 世纪 80 年代后期，当一个服务器一天只需处理几百、几千个客户时，这没什么。然而到了 Web 时代，一个重负荷的 Web 服务器一天要经受数以百万计的访问。这是就单台主机而言，对于最繁忙的站点人们往往运行多台主机来分摊负荷 (TCPv3 的 14.2 节谈到通常用于散布负载的 DNS 轮流法)。以后各节将会谈及多种技术来避免简单的并发服务器所需的每个客户一次 fork，不过

简单的并发服务器还是很普遍的。

图 27.5 是我们的并发服务器程序的 main 函数。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int         listenfd, connfd;
6     pid_t       childpid;
7     void        sig_chld(int), sig_int(int), web_child(int);
8     socklen_t clilen, addrlen;
9     struct sockaddr *cliaddr;
10    if (argc == 2)
11        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12    else if (argc == 3)
13        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14    else
15        err_quit("usage: serv01 [ <host> ] <port#>");
16    cliaddr = Malloc(addrlen);
17    Signal(SIGCHLD, sig_chld);
18    Signal(SIGINT, sig_int);
19    for ( ; ; ) {
20        clilen = addrlen;
21        if ( (connfd = accept(listenfd, cliaddr, &clilen)) < 0) {
22            if (errno == EINTR)
23                continue;       /* back to for() */
24            else
25                err_sys("accept error");
26        }
27        if ( (childpid = Fork()) == 0) {    /* child process */
28            Close(listenfd);  /* close listening socket */
29            web_child(connfd);    /* process the request */
30            exit(0);
31        }
32        Close(connfd);          /* parent closes connected socket */
33    }
34 }
```

图 27.5　TCP 并发服务器程序 main 函数[server/serv01.c]

这个函数与图 5.12 类似，它为每个客户连接 fork 一个子进程并处理子进程终止信号 SIGCHLD。不过，本函数已通过调用 tcp_listen 实现了协议无关性。此外，我们没有给出 sig_chld 信号处理程序，它与图 5.11 相同，只不过去掉了 printf 语句。

我们还处理键入终端中断键时产生的 SIGINT 信号。我们在客户完成后这么做以输出服务器程序运行所需 CPU 时间。图 27.6 为 SIGINT 信号处理程序。注意，这个信号处理程序不返回，直接调用 exit 终止进程。

```
35 void
36 sig_int(int signo)
37 {
38     void   pr_cpu_time(void);
```

```
39      pr_cpu_time();
40      exit(0);
41  }
```

图 27.6 SIGINT 信号处理程序[server/serv01.c]

图 27.7 是信号处理程序调用的 pr_cpu_time 函数。

```
1  #include    "unp.h"
2  #include    <sys/resource.h>
3  #ifndef HAVE_GETRUSAGE_PROTO
4  int     getrusage(int, struct rusage *);
5  #endif
6  void
7  pr_cpu_time(void)
8  {
9      double  user, sys;
10     struct rusage   myusage, childusage;
11     if (getrusage(RUSAGE_SELF, &myusage) < 0)
12         err_sys("getrusage error");
13     if (getrusage(RUSAGE_CHILDREN, &childusage) < 0)
14         err_sys("getrusage error");
15     user = (double) myusage.ru_utime.tv_sec +
16         myusage.ru_utime.tv_usec/1000000.0;
17     user += (double) childusage.ru_utime.tv_sec +
18         childusage.ru_utime.tv_usec/1000000.0;
19     sys = (double) myusage.ru_stime.tv_sec +
20         myusage.ru_stime.tv_usec/1000000.0;
21     sys += (double) childusage.ru_stime.tv_sec +
22         childusage.ru_stime.tv_usec/1000000.0;
23     printf("\nuser time = %g, sys time = %g\n", user, sys);
24 }
```

图 27.7 pr_cpu_time 函数:输出总 CPU 时间[server/pr_cpu_time.c]

getrusage 函数被调用了两次,分别返回调用进程的资源使用情况(RUSAGE_SELF)和它的所有已终止子进程的资源使用情况(RUSAGE_CHILDREN)。输出的值包括两部分:总的用户时间(用户进程耗费 CPU 时间)和总的系统时间(内核代表调用进程执行所耗费的时间)。

回过头来看一下图 27.5,它为每个客户请求调用一次 web_child 函数。图 27.8 给出了这个函数。

```
1  #include    "unp.h"
2  #define     MAXN   16384      /* max #bytes that a client can request */
3  void
4  web_child(int sockfd)
5  {
6      int             ntowrite;
7      ssize_t         nread;
8      char        line[MAXLINE], result[MAXN];
9      for ( ; ; ) {
10         if ( (nread = Readline(sockfd, line, MAXLINE)) == 0)
11             return;          /* connection closed by other end */
```

```
12              /* line from client specifies #bytes to write back */
13          ntowrite = atol(line);
14          if ((ntowrite <= 0) || (ntowrite > MAXN))
15              err_quit("client request for %d bytes", ntowrite);
16          Writen(sockfd, result, ntowrite);
17      }
18  }
```

图 27.8　处理每个客户请求的 web_child 函数[server/web_child.c]

当建立了与服务器的连接之后，客户向服务器写出一行以指定需由它返回给客户的数据字节数。这与 HTTP 有些类似：客户发送一个小请求，服务器返回所需的信息(HTML 文件、GIF 图像，等等)。不过在 HTTP 应用系统中，一般总由服务器在发回所请求的数据后关闭连接。当然，新版本允许使用持续连接(persistent connection)，保持 TCP 连接打开以便接受额外的客户请求。我们的 web_child 函数中服务器允许来自客户的额外请求，但在图 27.4 中我们看到我们的客户每次连接只发送一个请求，然后就自己关闭连接。

图 27.1 中第 1 行给出了简单并发服务器的时间耗费。与同一图表中其他几种并发服务器相比较，它的 CPU 时间最大，这是由于对每个客户 fork 一个子进程造成的。

> 我们没有测量的服务器程序设计之一是由 inetd 激活的服务器(见 12.5 节)。从进程控制角度看，如果服务器是由 inetd 激活的，那么每个子进程不仅要 fork 还要 exec，所以 CPU 时间会比图 27.1 中第 1 行所示的还大。

27.6　TCP 预先派生子进程服务器程序，accept 无上锁保护

我们的第一个“增强”型服务器程序运用了预先派生子进程技术。预先派生子进程服务器程序不再为每个客户请求 fork 一个子进程，而是在服务器启动时就预先派生一组子进程，做好为接入的客户作服务的准备。图 27.9 给出了一个预先派生 N 个子进程的服务器正在为 2 个客户同时服务的情形。

这种技术的优点在于：不需要引入父进程执行 fork 的开销，新客户就能得到处理。而缺点在于：每次启动服务器时，父进程必须猜测到底需要预先派生多少子进程。除此以外，如不考虑再派生子进程，一旦所有子进程都为客户请求所占用，再么此时新到的请求就将被暂时忽略，直到有一个进程可用。不过回忆一下 4.5 节，我们知道这些请求也并非被完全忽略。对于每个新到的客户，内核将通过三路握手与客户建立连接，直到达到相应套接口上 listen 调用的 backlog 数为止；当服务器调用 accept 时，这些已完成的连接就传递给它。不过客户仍能觉察到服务器响应时间变慢，因为即使它的 connect 调用可能立即返回，但它的第一个请求得过一段时间才被处理。

增加一些代码可以解决以上问题。由父进程来监视可用子进程数，一旦低于某个阈值就再派生额外的子进程。同样，如果空闲子进程数大于某个阈值，则父进程终止部分新派生的子进程，因为在对图 27.2 进行讨论时，我们将会发现过多的可用子进程也会降低性能。

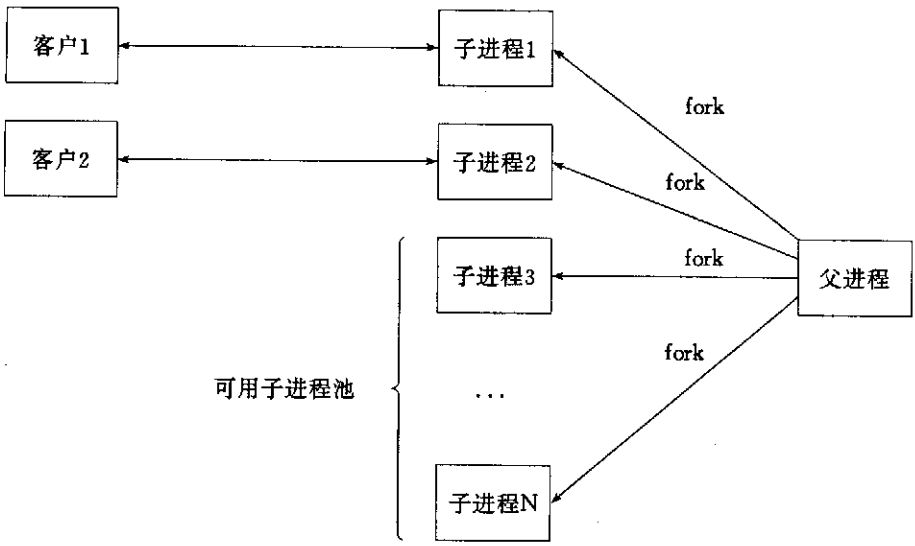


图 27.9 服务器预先派生子进程

不过,在考虑这些增强之前,先看一下这类服务器程序的基本结构。图 27.10 给出了我们的预先派生子进程服务器程序第一版的 main 函数。

```
1 #include    "unp.h"
2 static int       nchildren;
3 static pid_t     *pids;
4 int
5 main(int argc, char **argv)
6 {
7     int       listenfd, i;
8     socklen_t   addrlen;
9     void      sig_int(int);
10    pid_t     child_make(int, int, int);
11    if (argc == 3)
12        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13    else if (argc == 4)
14        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15    else
16        err_quit("usage: serv02 [ <host> ] <port#> <#children>");
17    nchildren = atoi(argv[argc-1]);
18    pids = Calloc(nchildren, sizeof(pid_t));
19    for (i = 0; i < nchildren; i++)
20        pids[i] = child_make(i, listenfd, addrlen);     /* parent returns */
21    Signal(SIGINT, sig_int);
22    for ( ; ; )
23        pause();    /* everything done by children */
24 }
```

图 27.10 预先派生子进程服务器程序 main 函数[server/serv02.c]

第 11～18 行 从命令行参数获取预先派生子进程的个数(可选),分配一个记录子进程进程 ID 的数组,用于在父进程将终止时,由 main 函数终止所有子进程。

第 19～20 行 使用 child_make 创建各个子进程,参见图 27.12。

如图 27.11 所示的 SIGINT 信号处理程序与图 27.6 有些不同。

第 30～34 行　由于 getrusage 汇报的是已终止进程的资源使用统计，因此在调用 pr_cpu_time 之前必须终止所有子进程。我们通过给每个子进程发送 SIGTERM 信号做到这一点，然后就 wait 所有子进程。

图 27.12 给出了 child_make 函数，它在 main 里被用于派生各个子进程。

第 7～9 行　fork 派生子进程后只有父进程返回，子进程则调用图 27.13 中的函数 child_main，它是个无限循环。

```
25 void
26 sig_int(int signo)
27 {
28     int     i;
29     void    pr_cpu_time(void);
30         /* terminate all children */
31     for (i = 0; i < nchildren; i++)
32         kill(pids[i], SIGTERM);
33     while (wait(NULL) > 0)        /* wait for all children */
34         ;
35     if (errno != ECHILD)
36         err_sys("wait error");
37     pr_cpu_time();
38     exit(0);
39 }
```

图 27.11　SIGINT 信号处理程序[server/serv02.c]

```
 1 #include    "unp.h"
 2 pid_t
 3 child_make(int i, int listenfd, int addrlen)
 4 {
 5     pid_t pid;
 6     void  child_main(int, int, int);
 7     if ( (pid = Fork()) > 0)
 8         return(pid);          /* parent */
 9     child_main(i, listenfd, addrlen);   /* never returns */
10 }
```

图 27.12　child_make 函数：派生各个子进程[server/child02.c]

```
11 void
12 child_main(int i, int listenfd, int addrlen)
13 {
14     int         connfd;
15     void        web_child(int);
16     socklen_t clilen;
17     struct sockaddr  *cliaddr;
18     cliaddr = Malloc(addrlen);
19     printf("child %ld starting\n", (long) getpid());
20     for ( ; ; ) {
21         clilen = addrlen;
```

```
22          connfd = Accept(listenfd, cliaddr, &clilen);
23          web_child(connfd);          /* process the request */
24          Close(connfd);
25      }
26  }
```

图 27.13 child_main 函数：每个子进程执行的无限循环[server/child02.c]

第 20～25 行 每个子进程均调用 accept，当它返回后，调用 web_child(图 27.8)处理客户请求，然后关闭连接，并等待下一个客户。此过程不断重复，直到被父进程终止。

4. 4BSD 上的实现

如果你没有见过这种运行方式(多个进程在同一个监听描述字上调用 accept)，你可能奇怪这居然能工作。这里，我们暂且离开一下正题，看看在源自 Berkeley 的内核上是怎么实现这一点的(如 TCPv2 所给出的)。

父进程在派生子进程之前创建监听套接口，而每次 fork 子进程时，各个子进程复制父进程的全部描述字。图 27.14 展示了 proc 结构(每个进程一个)、监听描述字的单个 file 结构以及单个 socket 结构的关系。

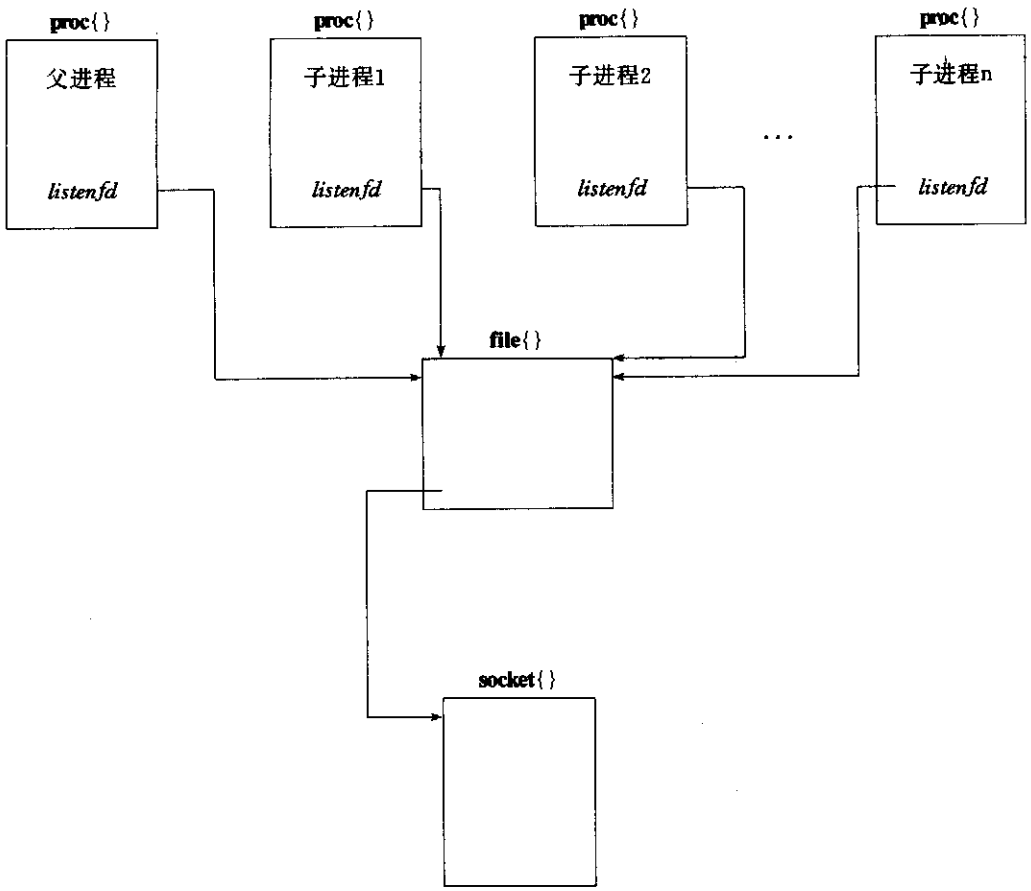


图 27.14 proc、file 和 socket 结构的关系

在 proc 结构中，描述字只是某个数组的一个下标，用于引用一个 file 结构。而 fork 派生

子进程时，子进程复制描述字的特性之一就是：子进程中给定的描述字所引用的 file 结构与父进程中同一描述字所引用的 file 结构一致。每个 file 结构有一个访问计数，它在文件或套接口打开时为 1，而每当调用 fork 或 dup 本描述字时，它就增 1。在我们的具有 N 个子进程的例子中，file 结构的访问计数为 N+1（别忘了虽然父进程从不调用 accept，但它并未关闭该监听描述字）。

当程序启动后，N 个子进程被派生，它们分别调用 accept 并由内核置入睡眠状态（TCPv2 第 458 页 140 行）。当第一个客户连接到来时，N 个睡眠进程均被唤醒。这是由于这 N 个进程共享一个 socket 结构，导致它们睡眠在同一等待通道（wait channel）即 socket 结构的 so_timeo 成员上。不过，虽然 N 个进程同时醒来，只有最先被调度的进程才能获得客户连接，而当其他 N−1 个进程执行到 TCPv2 第 458 页 135 行时，它们会发现队列长度为 0（连接已被取走了），因而再次被投入睡眠。

这通常被称为惊群（thundering herd）问题，因为尽管只有一个进程可获得连接，但所有（N 个）进程都被唤醒。不过这的确可以工作，只是每次有连接、待接受时唤醒太多的等待进程会导致系统性能下降。下面让我们来看一下对性能的影响。

过多子进程的影响

看一下图 27.1 第 2 行，在预先派生 15 个子进程，同时最多有 10 个客户的条件下，基于 BSD/OS 系统的服务器的 CPU 时间为 1.8。为了测量惊群问题的影响，我们保持最大客户数为 10 不变，单纯增加子进程数。在图 27.2 中我们给出了本例子以及将要讨论的两个其他例子的 CPU 时间。这里我们只讨论 accept 阻塞，把其他各栏留在以后几节讨论。

从图 27.2 可以看出，子进程每增加（不必要的）15 个，CPU 时间就有较大的增加。因此，为了避免惊群问题的发生，我们不希望有太多的额外子进程闲着不干事。

> 某些 Unix 内核有一个名为 wakeup_one 的函数，对于等待某个事件的一群进程，它只唤醒一个等待进程，而不是所有进程。不过 BSD/OS 内核没有这个函数。

连接在子进程中的分布

我们另一个关心的问题是，对于进程池中阻塞在 accept 上的可用子进程，客户连接如何分布。为了统计这个分布，我们修改了 main 函数，在共享内存中分配一个长整数计数器数组，每个子进程一个计数器，代码如下：

```
long    *cptr, *meter(int);     /* for counting #clients/child */
cptr = meter(nchildren);        /* before spawning children */
```

图 27.15 是函数 meter，用于在共享内存中分配计数器数组。

在 meter 中，如果系统支持（如 4.4BSD），我们就使用匿名内存映射（anonymous memeory mapping），否则使用 /dev/zero 映射（如 SVR4）。由于数组是由 mmap 在子进程派生之前分配的，因此该数组由该进程（父进程）和所有后来 fork 的子进程一起共享。

除此以外，我们还修改了 child_main 函数（图 27.13）：当 accept 返回后，每个子进程就对各自的计数器加 1；而 SIGINT 信号处理程序则在所有子进程终止后输出各计数器的内容。

```
 1 #include    "unp.h"
 2 #include    <sys/mman.h>
 3 /*
 4  * Allocate an array of "nchildren" longs in shared memory that can
 5  * be used as a counter by each child of how many clients it services.
 6  * See pp. 467-470 of "Advanced Programming in the Unix Environment".
 7  */
 8 long *
 9 meter(int nchildren)
10 {
11     int         fd;
12     long        *ptr;
13 #ifdef  MAP_ANON
14     ptr = Mmap(0, nchildren * sizeof(long), PROT_READ | PROT_WRITE,
15                MAP_ANON | MAP_SHARED, -1, 0);
16 #else
17     fd = Open("/dev/zero", O_RDWR, 0);
18     ptr = Mmap(0, nchildren * sizeof(long), PROT_READ | PROT_WRITE,
19                MAP_SHARED, fd, 0);
20     Close(fd);
21 #endif
22     return(ptr);
23 }
```

图 27.15 在共享内存分配数组的 meter 函数[server/meter.c]

图 27.3 给出分布结果。当可用子进程阻塞在 accept 调用上时，内核调度算法使得连接的分布是大致均匀的。

select 冲突

在观察 4.4BSD 下的本例子的同时，我们看一下另外一个难以理解又罕见的现象。TCPv2 的 16.13 节提到过 select 函数的“冲突”问题以及内核怎么处理它。当多个进程在同一描述字上 select 时就会发生 select 冲突，因为当一个描述字就绪(ready)后，相应的 socket 结构中只能够存放单个将被唤醒的进程 ID；而内核又无法知道哪些进程将受刚就绪的描述字影响，因此它只能唤醒在 select 上等待的所有进程。

对我们的程序作一点改动，我们就可以强迫发生 select 冲突。我们只需在图 27.13 中调用 accept 前加上一个 select 调用，等待监听套接口的可读条件。子进程就将阻塞在 select 而不是 accept 上。图 27.16 给出了改动后的 child_main 函数，被改动的语句前均标有“+”号。

```
        printf("child %ld starting\n", (long) getpid());
+       FD_ZERO(&rset);
        for( ; ; ) {
+           FD_SET(listenfd, &rset);
+           Select(listenfd+1,&rset,NULL,NULL,NULL);
+           if(FD_ISSET(listenfd, &rset) == 0)
+               err_quit("listenfd readable");
+
            clilen = addrlen;
            connfd = Accept(listenfd, cliaddr, &clilen);
            web_chile(connfd);      /* process the request */
```

```
        Close(connfd);
    }
```

图 27.16　改动图 27.13 以阻塞在 select 而不是 accept

如果我们作此改动并检查 BSD/OS 内核的 nselcoll 计数器在运行服务器前后的变化，我们发现第一次运行时有 1814 个冲突，第二次运行时有 2045 个冲突。由于两个客户共产生 5000 条连接，因此这两个结果相当于约有 35～40%的 select 调用发生了冲突。

如果从 BSD/OS 服务器的 CPU 时间上看，加入 select 调用后它由图 27.1 的 1.8 上升为 2.9。这有两个原因：其一是增加了一个系统调用，由 accept 变成 select 加 accept；其二是内核为处理冲突而增加了开销。

从以上讨论我们可以得出一点：如果有多个进程阻塞在同一描述字，那么阻塞在诸如 accept 这样的函数要比阻塞在 select 上好。

27.7　TCP 预先派生子进程服务器程序，accept 使用文件锁保护

我们刚叙述的 4.4BSD 上允许多个进程在同一监听描述字上调用 accept 的实现也仅仅适用于在内核中实现 accept 的源自 Berkeley 的内核。系统 V 内核以库函数形式实现 accept，就不允许这种实现。如果我们使用 Solaris 2.5(基于 SVR4 的内核)运行以上例子，那么客户开始连接到该服务器后不久，某个子进程的 accept 就会返回 EPROTO 错误，表示协议错。

> 造成问题的原因在于 SVR4 的流实现机制(第 33 章)，以及库函数 accept 并非一个原子操作的事实。Solaris 2.6 解决了这个问题，不过大多数其他 SVR4 实现仍存在这个问题。

解决问题的方法就是让应用进程在调用 accept 前后设置某种形式的锁(lock)，这样每次只有一个子进程阻塞在 accept 调用，其他子进程则在阻塞在试图获取提供调用 accept 权力的锁上。

这种锁的实现方法有很多，我们在本系列丛书的第二卷叙述。本节我们调用 fcntl 函数来使用 Posix 文件上锁功能。

main 函数的唯一改动是在子进程的循环前增加一个函数调用：my_lock_init。

```
+   my_lock_init("/tmp/lock.XXXXXX");   /* one lock file for all children */
    for(i = 0; i < nchildren; i++)
        pids[i] = child_make(i, listenfd,addrlen);   /* parent returns */
```

child_make 函数不变，与图 27.12 一样。child_main 函数(图 27.13)的唯一改动是在调用 accept 前获取文件锁，在 accept 返回后释放文件锁。

```
    for ( ; ; ) {
        clilen = addrlen;
+       my_lock_wait ();
        connfd = Accept (listenfd,cliaddr,&clilen);
+       my_lock_release ();
        web_child(connfd);          /* process the request */
        Close(connfd);
    }
```

图 27.17 给出了使用 Posix 文件上锁功能的 my_lock_init 函数。

```
1 #include    "unp.h"
2 static struct flock lock_it, unlock_it;
3 static int lock_fd = -1;
4                       /* fcntl() will fail if my_lock_init() not called */
5 void
6 my_lock_init(char *pathname)
7 {
8     char    lock_file[1024];
9         /* must copy caller's string, in case it's a constant */
10    strncpy(lock_file, pathname, sizeof(lock_file));
11    Mktemp(lock_file);
12    lock_fd = Open(lock_file, O_CREAT | O_WRONLY, FILE_MODE);
13    Unlink(lock_file);              /* but lock_fd remains open */
14    lock_it.l_type = F_WRLCK;
15    lock_it.l_whence = SEEK_SET;
16    lock_it.l_start = 0;
17    lock_it.l_len = 0;
18    unlock_it.l_type = F_UNLCK;
19    unlock_it.l_whence = SEEK_SET;
20    unlock_it.l_start = 0;
21    unlock_it.l_len = 0;
22 }
```

图27.17 使用Posix.1文件上锁功能的my_lock_init函数[server/lock_fcntl.c]

第9～13行 调用者作为参数给my_lock_init指定一个路径名模板，mktemp函数就根据该模板创建一个唯一的路径名。具有该路径名的文件随后创建并立即unlink掉。这样，一旦程序崩溃，该文件也自动消失，然而只要有一个或多个进程打开着该文件(也就是说它的访问计数大于0)，那么该文件本身并不会真正删除。(这也是关闭打开的文件与从目录中删除路径名的基本差别。)

第14～21行 初始化两个flock结构：一个用于上锁文件，一个用于解锁文件。从字节偏移量0开始锁(l_whence设为SEEK_SET，l_start设为0)，由于l_len设为0，因此整个文件被锁。当然，我们并不往文件里写什么(它的长度总为0)，但这无所谓，内核仍能正确地处理这个建议性质的锁(相对于强制性质的锁)。

本来，作者在声明这些结构时使用如下语句初始化它们：

```
static struct flock lock_it = {F_WRLCK, 0, 0, 0, 0 };
static struct flock unlock_it = {F_UNLCK, 0, 0, 0, 0 };
```

然而，这样做有两个问题。首先，我们无法保证SEEK_SET就是常值0。其次，我们无法保证flock结构内成员的顺序。在Solaris和Digital Unix上，l_type是第一个成员，而在BSD/OS上它不是。Posix只规定结构内必须有哪些成员，却未规定它们的前后顺序，它还允许在结构中有额外的非Posix成员。因此，除非把一个结构初始化为全0，否则必须总是以真正的C代码完成初始化，而不是在分配结构时使用初始化常量(initilizer)。

这条规则的例外是当结构初始化常值是由具体的实现提供时。例如在第23章初始

化 Pthread 互斥锁时，我们是这么写的：

```
pthread_mutex_t  mlock = PTHREAD_MUTEX_INITIALIZER;
```

其中 pthread_mutex_t 数据类型通常为一个结构，但初始化常量是由实现提供的，不同实现可以不一样。

图 27.18 给出了上锁和解锁文件的函数，它们仅仅调用 fcntl，并使用图 27.17 中初始化过的结构。

这个新版本预先派生子进程服务器程序可以运行在 SVR4 系统上，因为它保证每次只有一个子进程在 accept 上等待。不过对比图 27.1 中 Digital Unix 和 BSD/OS 服务器的第 2 行和第 3 行，可以看出，文件上锁增加了服务器的进程控制 CPU 时间。

> Apache Web 服务器程序 1.1 版(http://www.apache.org)利用了以上两节介绍的技术。预先派生子进程后，如果实现允许所有子进程都阻塞在 accept 调用上，那么使用上一节介绍的技术，否则就使用本节介绍的在 accept 前后的文件上锁技术。

过多子进程的影响

我们可以检查一下使用文件锁是否会有上节所述的惊群现象。图 27.2 给出了增加不必要的子进程以后的结果。在 accept 前后使用文件上锁的 Solaris 一栏中，当子进程数由 75 变成 90 以后，CPU 时间有相当大的增加。这可能是由于进程过多导致内存耗尽，从而开始对换引起的。

连接在子进程中的分布

我们也用图 27.15 中叙述的函数对客户在可用子进程池中的分布进行统计。结果如图 27.3 所示，所有三种操作系统都均匀地把文件锁分布到等待进程中。

```
23 void
24 my_lock_wait()
25 {
26     int             rc;
27     while ( (rc = fcntl(lock_fd, F_SETLKW, &lock_it)) < 0) {
28         if (errno == EINTR)
29             continue;
30         else
31             err_sys("fcntl error for my_lock_wait");
32     }
33 }
34 void
35 my_lock_release()
36 {
37     if (fcntl(lock_fd, F_SETLKW, &unlock_it) < 0)
38         err_sys("fcntl error for my_lock_release");
39 }
```

图 27.18　使用 fcntl 的 my_lock_wait 和 my_lock_redease 函数[server/lock_fcntl.c]

27.8 TCP预先派生子进程服务器程序，accept使用线程互斥锁保护

前面我们提过，实现进程间上锁有多种方式。前面所提及的Posix文件上锁方式虽然在所有Posix兼容系统都能用，但却要涉及文件系统操作，这是很耗时的。本节我们将使用线程上锁，因为这种方式不仅适用于同一进程内各线程间的上锁，也适用于不同进程间的上锁。

为了使用线程上锁，我们的main、child_make和child_main函数都不用动，只需要修改3个上锁函数就可以了。为了在多个进程之间使用线程锁，我们应做到(1)互斥锁变量必须存储在为所有进程所共享的内存中；(2)必须通知线程函数库互斥锁是在不同进程间共享的。

> 这同样要求线程库支持PTHREAD_RPOCESS_SHARED属性。Digitial Unix 4.0b不支持这个属性，也就无法运行这个新版本的服务器程序。

在不同进程间共享内存有多种实现方式(见本系列丛书第二卷)。在本节中，我们将使用mmap函数和/dev/zero设备，它在Solaris和其他SVR4内核下均可运行。图27.19给出了新版的my_lock_init函数。

```
1 #include    "unpthread.h"
2 #include    <sys/mman.h>
3 static pthread_mutex_t   *mptr;     /* actual mutex will be in shared memory */
4 void
5 my_lock_init(char *pathname)
6 {
7     int         fd;
8     pthread_mutexattr_t     mattr;
9     fd = Open("/dev/zero", O_RDWR, 0);
10    mptr = Mmap(0, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE,
11                MAP_SHARED, fd, 0);
12    Close(fd);
13    Pthread_mutexattr_init(&mattr);
14    Pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
15    Pthread_mutex_init(mptr, &mattr);
16 }
```

图27.19 在进程间使用Pthread上锁的my_lock_init函数[server/lock_pthread.c]

第9～12行 打开/dev/zero，并调用mmap。映射的字节数为pthread_mutex_t类型变量的大小。紧接着关闭文件，这不会有问题，因为描述字已经内存映射了。

第13～15行 在以前的例子里，我们一般用常值PTHREAD_MUTEX_INITIALIZER(例如图23.18)来初始化全局或静态互斥锁变量。然而，对于一个共享内存中的互斥锁，我们必须调用某些Pthread库函数来通知该库：该互斥锁位于共享内存中，且将用于不同进程之间的上锁。因此，我们先使用缺省属性初始化一个pthread_mutexattr_t结构，然后赋予

PTHREAD_PROCESS_SHARED 属性(本属性缺省值为 PTHREAD_PROCESS_PRIVATE,只允许在单个进程内使用)。最后使用 pthread_mutex_init 函数以这些属性初始化互斥锁变量(mptr)。

图 27.20 是新的 my_lock_wait 和 my_lock_release 函数。各自调用一个 Pthread 函数来加锁和解锁互斥锁。

比较一下图 27.1 的第 3 行和第 4 行,可以看出,线程互斥锁上锁方式要快于文件上锁方式。

```
17 void
18 my_lock_wait()
19 {
20      Pthread_mutex_lock(mptr);
21 }
22 void
23 my_lock_release()
24 {
25      Pthread_mutex_unlock(mptr);
26 }
```

图 27.20　使用 Pthread 上锁的 my_lock_wait 和 my_lock_release 函数[server/lock_pthread.c]

27.9　TCP 预先派生子进程服务器程序,传递描述字

对预先派生子进程服务器程序的最后一种改动就是由父进程调用 accept,然后再将所接受的已连接描述字传递给子进程。这样绕过了在所有子进程中调用 accept 可能需要的上锁保护,但父进程必须以某种形式向子进程传递描述字。这种技术比较复杂,因为父进程必须跟踪子进程的忙闲状态,以便给空闲子进程传递新的描述字。

在以前的预先派生子进程的例子中,进程无需关心哪个子进程接收连接,而由操作系统处理这个细节,给某个子进程以首先调用 accept 的权力或给予它文件锁或互斥锁。图 27.3 的前 5 栏表明我们测量的三种操作系统对此类操作具有公平性。

而在本例子中,我们必须为每个子进程维护一个信息结构,用来管理各子进程。图 27.21 中的 child.h 头文件定义了我们的 Child 结构。

```
1 typedef struct {
2     pid_t     child_pid;    /* process ID */
3     int       child_pipefd; /* parent's stream pipe to/from child */
4     int       child_status; /* 0 = ready */
5     long      child_count;  /* #connections handled */
6 } Child;
7 Child *cptr;               /* array of Child structures; calloc'ed */
```

图 27.21　Child 结构[server/child.h]

我们将存储子进程进程 ID、连接到子进程的父进程字节流管道描述字、子进程状态以及子进程已处理客户数。在程序终止时,SIGINT 信号处理程序将输出这些计数值,以便观察客户请求在各子进程间的分布。

让我们先看一下图27.22中的child_make函数。在调用fork之前，先创建一个字节流管道，它是Unix域的字节流套接口(第14章)。当子进程派生之后，父进程关闭一个描述字(sockfd[1])，子进程关闭另一个描述字(sockfd[0])。此外子进程将流管道的自己所在端(sockfd[1])复制到标准错误输出，这样每个子进程光通过读/写标准错误输出就和父进程通信。父子进程间关系如图27.23。

```
1 #include    "unp.h"
2 #include    "child.h"
3 pid_t
4 child_make(int i, int listenfd, int addrlen)
5 {
6     int         sockfd[2];
7     pid_t pid;
8     void   child_main(int, int, int);
9     Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);
10    if ( (pid = Fork()) > 0) {
11        Close(sockfd[1]);
12        cptr[i].child_pid = pid;
13        cptr[i].child_pipefd = sockfd[0];
14        cptr[i].child_status = 0;
15        return(pid);          /* parent */
16    }
17    Dup2(sockfd[1], STDERR_FILENO);      /* child's stream pipe to parent */
18    Close(sockfd[0]);
19    Close(sockfd[1]);
20    Close(listenfd);              /* child does not need this open */
21    child_main(i, listenfd, addrlen);   /* never returns */
22 }
```

图27.22 描述字传递型预先派生子进程服务器程序的child_make函数[server/child05.c]

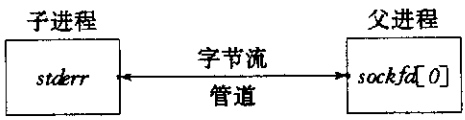


图27.23 父进程和子进程各关闭一端后的字节流管道

当所有子进程都派生之后，布局就如图27.24所示。我们关闭每个子进程中的监听描述字，因为只有父进程调用accept。父进程使用select同时监视监听套接口和各个字节流套接口。真如你猜想的那样，父进程使用select来选择各个描述字。

图27.25是新的main函数。与以前各个版本不同之处在于：它分配描述字集，并设置其中对应于监听套接口以及到各个子进程的字节流管道的位。此外，我们还给Child结构数组分配空间。整个函数由一个select调用来驱动。

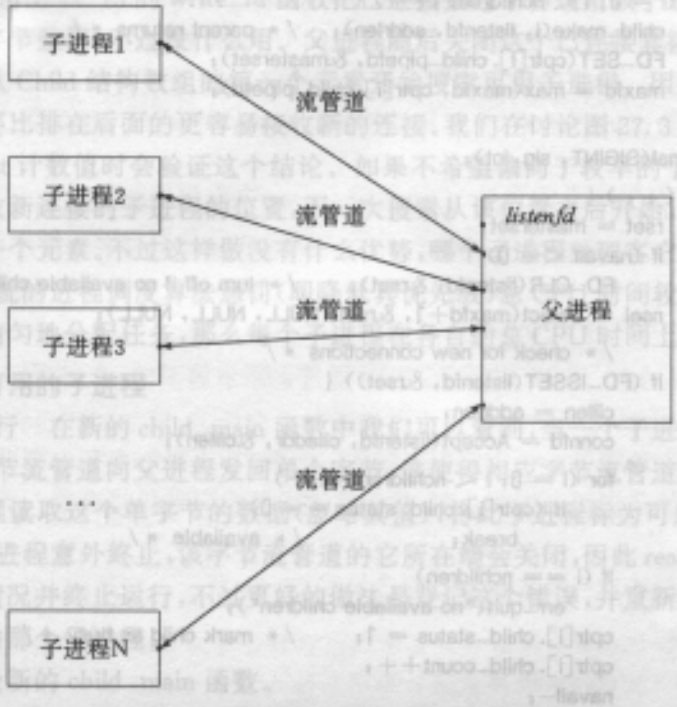


图 27.24　所有子进程都派生后的各个字节流管道

```
1 #include    "unp.h"
2 #include    "child.h"
3 static int nchildren;
4 int
5 main(int argc, char **argv)
6 {
7     int         listenfd, i, navail, maxfd, nsel, connfd, rc;
8     void        sig_int(int);
9     pid_t       child_make(int, int, int);
10    ssize_t     n;
11    fd_set      rset, masterset;
12    socklen_t   addrlen, clilen;
13    struct sockaddr *cliaddr;
14    if (argc == 3)
15        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
16    else if (argc == 4)
17        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
18    else
19        err_quit("usage: serv05 [ <host> ] <port#> <#children>");
20    FD_ZERO(&masterset);
21    FD_SET(listenfd, &masterset);
22    maxfd = listenfd;
23    cliaddr = Malloc(addrlen);
24    nchildren = atoi(argv[argc-1]);
25    navail = nchildren;
26    cptr = Calloc(nchildren, sizeof(Child));
27    /* prefork all the children */
```

```
28      for (i = 0; i < nchildren; i++) {
29          child_make(i, listenfd, addrlen);    /* parent returns */
30          FD_SET(cptr[i].child_pipefd, &masterset);
31          maxfd = max(maxfd, cptr[i].child_pipefd);
32      }
33      Signal(SIGINT, sig_int);
34      for ( ; ; ) {
35          rset = masterset;
36          if (navail <= 0)
37              FD_CLR(listenfd, &rset);      /* turn off if no available children */
38          nsel = Select(maxfd+1, &rset, NULL, NULL, NULL);
39              /* check for new connections */
40          if (FD_ISSET(listenfd, &rset)) {
41              clilen = addrlen;
42              connfd = Accept(listenfd, cliaddr, &clilen);
43              for (i = 0; i < nchildren; i++)
44                  if (cptr[i].child_status == 0)
45                      break;                /* available */
46              if (i == nchildren)
47                  err_quit("no available children");
48              cptr[i].child_status = 1;     /* mark child as busy */
49              cptr[i].child_count++;
50              navail--;
51              n = Write_fd(cptr[i].child_pipefd, "", 1, connfd);
52              Close(connfd);
53              if (--nsel == 0)
54                  continue;    /* all done with select() results */
55          }
56              /* find any newly-available children */
57          for (i = 0; i < nchildren; i++) {
58              if (FD_ISSET(cptr[i].child_pipefd, &rset)) {
59                  if ( (n = Read(cptr[i].child_pipefd, &rc, 1)) == 0)
60                      err_quit("child %d terminated unexpectedly", i);
61                  cptr[i].child_status = 0;
62                  navail++;
63                  if (--nsel == 0)
64                      break;     /* all done with select() results */
65              }
66          }
67      }
68  }
```

图 27.25 使用描述字传递的 main 函数[server/serv05.c]

如果无可用子进程则关掉监听套接口可读条件

第36～37行 计数器 navail 用来记录当前可用的子进程数。如果其值为0，则将监听套接口从 select 的描述字集中除去。这样可以防止在无可用子进程的情况下 accept 新的客户连接。内核仍将外来连接排入队列，直到达到 listen 的 backlog 数为止，但我们在没有可用子进程前不想 accept 它们。

accept 新连接

第39～55行 如果监听套接口可读则有一个新连接等待 accept。我们找出第一个可用

的子进程，并使用图 14.13 的 write_fd 函数把已连接描述字传递给该子进程，与描述字同时传递的还有 1 字节数据，不过没什么用。父进程随后关闭这个已连接套接口。

我们总是从 Child 结构数组的第一个元素开始搜索可用子进程。因此排在该数组前面的子进程总是要比排在后面的更容易接收新的连接。我们在讨论图 27.3 和查看服务器终止后的 child_count 计数值时会验证这个结论。如果不希望偏向于较早的子进程，我们可以记录最近一次接收新连接的子进程的位置，下一次搜索从该位置之后开始，如果到达数组的尾部则再回到第一个元素。不过这样做没有什么优势，哪个子进程处理客户请求并没有什么关系，除非操作系统的进程调度算法惩罚（即降低其优先级）总 CPU 时间较长的进程。如果在各个子进程间均匀地分配任务，那么每个子进程在各自的总 CPU 时间上也趋于一致。

处理新近可用的子进程

第 56～66 行　在新的 child_main 函数中我们可以看到，当一个子进程处理完客户请求后，它就通过字节流管道向父进程发回单个字节。这使得相应字节流管道的父进程所在端变为可读。父进程读取这个单字节的数据（忽略其值），将此子进程标为可用，然后递增 navail 计数器。如果子进程意外终止，该字节流管道的它所在端会关闭，因此 read 的返回值为 0。父进程捕获这种情况并终止运行，不过更好的做法是登记这个错误，并重新派生一个子进程来替代意外终止的那个子进程。

图 27.26 是新的 child_main 函数。

```
23 void
24 child_main(int i, int listenfd, int addrlen)
25 {
26     char        c;
27     int         connfd;
28     ssize_t     n;
29     void            web_child(int);
30     printf("child %ld starting\n", (long) getpid());
31     for ( ; ; ) {
32         if ( (n = Read_fd(STDERR_FILENO, &c, 1, &connfd)) == 0)
33             err_quit("read_fd returned 0");
34         if (connfd < 0)
35             err_quit("no descriptor from read_fd");
36         web_child(connfd);              /* process the request */
37         Close(connfd);
38         Write(STDERR_FILENO, "", 1);   /* tell parent we're ready again */
39     }
40 }
```

图 27.26　描述字传递型预先派生子进程服务器程序的 child_main 函数[server/child05.c]

等待来自父进程的描述字

第 32～33 行　这个函数与以前有所不同。子进程不再调用 accept，而是用 read_fd 等待父进程传递的已连接套接口描述字。

通知父进程已准备好

第 38 行　完成客户请求后，子进程通过字节流管道向父进程写一个字节，通知父进程本进程已空闲。

在图27.1中,通过比较Solaris服务器的第4和第5行可以看出,目前这个服务器要比在子进程间使用线程上锁的服务器慢。比较Digtial Unix和BSD/OS服务器的第3行和第5行,可以得出类似结论,即通过字节流管道传递描述字给每个子进程,由子进程通过字节流管道写回1字节的数据以表示可用,要比无论是共享内存中的互斥锁还是文件锁的上锁和解锁更费时。

图27.3给出Child结构中child_count计数值的分布,它是在终止服务器时由SIGINT信号处理程序输出的。与对图27.25的讨论一致,越是排在前头的子进程所处理的客户请求就越多。

27.10 TCP并发服务器程序,每个客户一个线程

最近5节里我们着眼于由一个进程来服务一个客户,或者每个客户调用一次fork,或者预先派生某个数目的子进程。下面几节,我们将试图用线程来代替进程。当然,系统要能支持线程才行。

我们的第一个创建线程版本服务器程序如图27.27所示。它是图27.5的修改版,而且与图23.3很相似。

```
1 #include    "unpthread.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int       listenfd, connfd;
6     void      sig_int(int);
7     void      * doit(void * );
8     pthread_t tid;
9     socklen_t clilen, addrlen;
10    struct sockaddr * cliaddr;
11    if (argc == 2)
12        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
13    else if (argc == 3)
14        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
15    else
16        err_quit("usage: serv06 [ <host> ] <port#>");
17    cliaddr = Malloc(addrlen);
18    Signal(SIGINT, sig_int);
19    for ( ; ; ) {
20        clilen = addrlen;
21        connfd = Accept(listenfd, cliaddr, &clilen);
22        Pthread_create(&tid, NULL, &doit, (void * ) connfd);
23    }
24 }
25 void *
26 doit(void * arg)
27 {
28    void  web_child(int);
29    Pthread_detach(pthread_self());
30    web_child((int) arg);
```

```
31      Close((int) arg);
32      return(NULL);
33  }
```

图 27.27　创建线程 TCP 服务器程序的 main 函数[server/serv06.c]

主线程循环

第 19～23 行　主线程循环阻塞在 accept 调用上，一旦有新连接到达，就使用 pthread_create 创建一个新线程。新线程执行的函数为 doit，其参数就是已连接描述字。

每个线程的函数

第 25～33 行　doit 函数先将自己与主线程脱离，使得主线程不必等待它，然后调用 web_client 函数(图 27.4)。该函数返回后关闭已连接套接口。

通过图 27.1 我们可以发现，这个简单的创建线程版本在 Solaris 和 Digital Unix 上都要快于所有的预先派生子进程版本。当然也快于第 1 行的每个客户一个子进程的那个版本。

> 在 23.5 节，我们曾经提到过将非线程安全函数转变成线程安全函数的三种可选方法。我们的 web_child 函数调用 readline 函数，而图 3.17 的 readline 函数版本是非线程安全的。我们针对图 27.27 的例子应用 23.5 节中第 2 和第 3 种转变方法并计时。结果从第 3 种方法到第 2 种方法的加速比少于 1%，这也许是 readline 仅仅用来读来自客户的 5 字节计数值的缘故。因此为了简单起见，我们给本章中预先创建线程的服务器程序使用图 3.16 所给的效率稍低的版本。

27.11　TCP 预先创建线程服务器程序，每个线程各自 accept

从本章前面我们发现，预先派生一个子进程池要比给每个客户派生一个子进程来得快。对于支持线程的系统而言，预先创建一个线程池来提高速度看起来也是合理的。基本设计方法是由服务器预先创建一组线程，每个线程各自调用 accept 接受连接。不过我们不是让每个线程都阻塞在 accept 调用上，而是直接使用互斥锁(类似于 27.8 节)来保证线程间互斥地调用 accept。这里就不用文件锁了，因为对于单个进程中的多个线程来说，互斥锁总是可用。

我们通过 pthread07.h 头文件定义了用于维护线程信息的 Thread 结构，见图 27.28。

```
1 typedef struct {
2       pthread_t thread_tid;        /* thread ID */
3       long      thread_count;      /* #connections handled */
4 } Thread;
5 Thread        *tptr;               /* array of Thread structures; calloc'ed */
6 int           listenfd,nthreads;
7 socklen_t addrlen;
8 pthread_mutex_t mlock;
```

图 27.28　pthread07.h 头文件[server/pthread07.h]

当然，该头文件里还有一些其他全程变量定义，譬如监听套接口描述字和各线程共享的互斥锁变量等。

图 27.29 给出了 main 函数。

```
1 #include    "unpthread.h"
2 #include    "pthread07.h"
3 pthread_mutex_t    mlock = PTHREAD_MUTEX_INITIALIZER;
4 int
5 main(int argc, char * * argv)
6 {
7     int         i;
8     void   sig_int(int), thread_make(int);
9     if (argc == 3)
10        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
11    else if (argc == 4)
12        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
13    else
14        err_quit("usage: serv07 [ <host> ] <port#> <#threads>");
15    nthreads = atoi(argv[argc-1]);
16    tptr = Calloc(nthreads, sizeof(Thread));
17    for (i = 0; i < nthreads; i++)
18        thread_make(i);          /* only main thread returns */
19    Signal(SIGINT, sig_int);
20    for ( ; ; )
21        pause();      /* everything done by threads */
22 }
```

图 27.29 预先创建线程 TCP 服务器程序的 main 函数[server/serv07.c]

图 27.30 给出了函数 thread_make 和 thread_main。

```
1 #include    "unpthread.h"
2 #include    "pthread07.h"
3 void
4 thread_make(int i)
5 {
6     void   * thread_main(void * );
7     Pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void * ) i);
8     return;          /* main thread returns */
9 }
10 void *
11 thread_main(void * arg)
12 {
13    int         connfd;
14    void   web_child(int);
15    socklen_t clilen;
16    struct sockaddr * cliaddr;
17    cliaddr = Malloc(addrlen);
18    printf("thread %d starting\n", (int) arg);
19    for (; ;) {
20        clilen = addrlen;
21        Pthread_mutex_lock(&mlock);
22        connfd = Accept(listenfd, cliaddr, &clilen);
23        Pthread_mutex_unlock(&mlock);
24        tptr[(int) arg].thread_count++;
25        web_child(connfd);          /* process the request */
```

```
26          Close(connfd);
27      }
28  }
```

图 27.30 thread_make 和 thread_main 函数[server/pthread07.c]

创建线程

第 7 行　创建线程并使之执行 thread_main 函数，它的唯一的参数为本线程在 Thread 结构数组 tptr 中的下标。

第 21～23 行　thread_main 函数在调用 accept 前后调用 pthread_mutex_lock 和 pthread_mutex_unlock 进程保护。

在图 27.1 中比较第 6 行与第 7 行，目前这个版本在 Solaris 和 Digital Unix 上的确比每个客户一个线程方式要快一点。我们期望也是这样，毕竟我们只需要一次性创建线程池，而无需每来一个客户创建一个线程。事实上，在 Solaris 和 Digital Unix 上这个版本是最快的。

图 27.3 给出了 Thread 结构内的 thread_count 计数值的分布。SIGINT 信号处理程序在服务器终止前输出这些信息。由于线程调度算法表现为均匀调度各线程，因此各线程均匀地得到互斥锁，客户也呈均衡分布。

> 对于像 Digital Unix 这样的源自 Berkeley 的内核，我们在调用 accept 前后可以不上锁，从而构造一个没有互斥锁上锁和解锁的图 27.30 的版本。不过这样做，将使图 27.1 第 7 行的进程控制 CPU 时间由 3.5 增长为 3.9。如果你同时注意用户时间(user time)和系统时间(system time)两个部分的话，你会发现用户时间减少的同时，系统时间增加了。用户时间减少是因为上锁是由在用户空间执行的线程库完成的；系统时间增长是因为一个连接到达时所有阻塞在 accept 上的线程都被唤醒，造成内核的惊群问题。既然为把每个连接返回给单个线程，某种形式的互斥是必需的，让线程自己来完成这件工作要比由内核来完成快一些。

27.12　TCP 预先创建线程服务器程序，主线程统一 accept

最后一种服务器程序设计也预先创建一个线程池，但接着由主线程单独调用 accept 并将每个客户连接传递给池中某个可用线程。这与 27.9 节的描述字传递有些类似。

主要的设计问题在于主线程如何将已连接描述字传递给线程池中可用的线程。这有多种实现方法，包括以前使用的描述字传递法。不过现在几个线程都在一个进程之内，所以无需传递描述字，所需知道的只是描述字号。图 27.31 给出了 pthread08.h 头文件，其中定义了一个 Thread 结构，与图 27.28 一样。

```
1 typedef struct {
2     pthread_t thread_tid;          /* thread ID */
3     long      thread_count;        /* #connections handled */
4 } Thread;
5 Thread *tptr;      /* array of Thread structures; calloc'ed */
6 #define MAXNCLI 32
7 int       clifd[MAXNCLI], iget, iput;
8 pthread_mutex_t clifd_mutex;
```

```
9 pthread_cond_t clifd_cond;
```

图 27.31 pthread08.h 头文件[server/pthread08.h]

定义存放已连接描述字的共享数组

第 6～9 行 我们还定义 clifd 数组，由主线程在某中记录已连接套接口描述字。池中的可用线程从该数组内获取一个已连接描述字，并为相应的客户服务。iput 是主线程在 clifd 数组中放置下一项的下标，iget 是池中某个线程从同一数组中取走下一项的下标。当然，由于这个数组由所有线程共享，这些操作必须保证互斥。我们使用互斥锁和条件变量来实现这一点。

图 27.32 是 main 函数。

```
1 #include    "unpthread.h"
2 #include    "pthread08.h"
3 static int             nthreads;
4 pthread_mutex_t        clifd_mutex = PTHREAD_MUTEX_INITIALIZER;
5 pthread_cond_t         clifd_cond = PTHREAD_COND_INITIALIZER;
6 int
7 main(int argc, char **argv)
8 {
9     int               i, listenfd, connfd;
10    void        sig_int(int), thread_make(int);
11    socklen_t   addrlen, clilen;
12    struct sockaddr   *cliaddr;
13    if (argc == 3)
14        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
15    else if (argc == 4)
16        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
17    else
18        err_quit("usage: serv08 [ <host> ] <port#> <#threads>");
19    cliaddr = Malloc(addrlen);
20    nthreads = atoi(argv[argc-1]);
21    tptr = Calloc(nthreads, sizeof(Thread));
22    iget = iput = 0;
23        /* create all the threads */
24    for (i = 0; i < nthreads; i++)
25        thread_make(i);         /* only main thread returns */
26    Signal(SIGINT, sig_int);
27    for ( ; ; ) {
28        clilen = addrlen;
29        connfd = Accept(listenfd, cliaddr, &clilen);
30        Pthread_mutex_lock(&clifd_mutex);
31        clifd[iput] = connfd;
32        if (++iput == MAXNCLI)
33            iput = 0;
34        if (iput == iget)
35            err_quit("iput = iget = %d", iput);
36        Pthread_cond_signal(&clifd_cond);
```

```
37          Pthread_mutex_unlock(&clifd_mutex);
38      }
39  }
```

图 27.32　预先创建线程服务器程序 main 函数[server/serv08.c]

创建线程池

第 23～25 行　使用 thread_make 创建线程池。

等待客户连接

第 27～38 行　主线程阻塞在 accept 上，等待客户连接的到达。一旦某个客户连接到达，就将已连接描述字存入 clifd 数组的下一项中，当然，事先必须获得保护该数组的互斥锁。我们还要检查 iput 没有赶上 iget，如果这样说明数组太小。然后发出条件变量信号并释放 mutex，以允许线程池中某个线程为该客户服务。

图 27.33 给出了 thread_make 和 thread_main 函数。前者与图 27.30 中的版本相同。

```
 1 #include    "unpthread.h"
 2 #include    "pthread08.h"
 3 void
 4 thread_make(int i)
 5 {
 6     void    *thread_main(void *);
 7     Pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i);
 8     return;             /* main thread returns */
 9 }
10 void *
11 thread_main(void *arg)
12 {
13     int         connfd;
14     void    web_child(int);
15     printf("thread %d starting\n", (int) arg);
16     for ( ; ; ) {
17     Pthread_mutex_lock(&clifd_mutex);
18         while (iget == iput)
19             Pthread_cond_wait(&clifd_cond, &clifd_mutex);
20         connfd = clifd[iget];   /* connected socket to service */
21         if (++iget == MAXNCLI)
22             iget = 0;
23         Pthread_mutex_unlock(&clifd_mutex);
24         tptr[(int) arg].thread_count++;
25         web_child(connfd);          /* process the request */
26         Close(connfd);
27     }
28 }
```

图 27.33　thread_make 和 thread_main 函数[server/pthread08.c]

等待待服务的客户描述字

第 17～26 行　线程池中每个线程都力图获得保护 clifd 数组的互斥锁。获得之后就测试 iput 与 iget，如果两者相等则无事可做，于是调用 pthread_cond_wait 并进入睡眠。主线程在接受新连接后将调用 pthread_cond_signal 唤醒睡眠线程。如果 iput 与 iget 不等则从 clifd 数组取走下一项，得到相应的客户连接，然后调用 web_child。

图27.1指示最后这个版本比前一节中的要慢一些。原因可能在于本节的例子同时需要互斥锁和条件变量，而图27.30中只用到了互斥锁。

如果我们检查池中各个线程的客户分布直方图的话，我们会发现它与图27.3的最后一栏相类似。这意味着主线程调用pthread_cond_signal时，线程库是基于条件变量在所有可用线程中循环唤醒某一个的。

27.13 小 结

本章我们讨论了9种服务器程序设计方法，并用Web风格的客户程序对它们分别进行了测试，还对各类服务器用于进程控制的CPU时间进行了比较。考察的设计方法有以下几种：

0. 迭代服务器(无进程控制，用作测量基准)。
1. 简单并发服务器，每个客户fork一次。
2. 预先派生子进程，每个子进程相互独立调用accept。
3. 预先派生子进程，使用文件上锁保护accept。
4. 预先派生子进程，使用线程互斥锁上锁保护accept。
5. 预先派生子进程，父进程向子进程传递套接口描述字。
6. 并发服务器，每个客户请求创建一个线程。
7. 预先创建线程服务器，使用互斥锁上锁保护accept。
8. 预先创建线程服务器，由主线程调用accept。

经过实践，可以得到以下几点结论：

- 当系统负载较轻时，传统的并发服务器程序模型就够了，也就是每来一个客户请求，派生一个子进程为之服务。它甚至可以与inetd结合使用，由inetd负责接受每个连接。其他结论适用于重负载情况，譬如Web服务器。
- 相对于传统的每个客户一次fork型设计，预先创建一个进程池或线程池可以减少进程控制CPU时间，大约可减少10倍或以上。编码并不复杂，不过对于实用系统而言，还须监视空闲子进程数，并随所服务客户数的动态变化而增加或减少这个数目。
- 某些实现允许多个子进程或线程阻塞在accept调用上。然而在另一些实现上，我们必须使用文件锁、线程互斥锁或其他类型的锁来确保每次只有一个子进程或线程在accept。
- 一般来讲，所有子进程或线程都调用accept要比父进程或主线程调用accept后将描述字传递给子进程或线程来得简单而且快。
- 由于select冲突的存在，让所有子进程或线程阻塞在同一监听套接口的accept调用上要比让它们阻塞在select调用上更为可取。
- 使用线程通常要比使用进程快。不过选择每个客户一个子进程还是每个客户一个线程取决于操作系统提供什么以及需什么其他程序(如果有的话)来服务各个客户。例如，如果accept客户连接的服务器调用fork和exec，那么fork一个单线程的进程要比fork一个多线程的进程快。

27.14　习　题

27.1　在图 27.14 中，为什么父进程总使监听套接口保持打开状态，而不在子进程全部创建后关闭它？

27.2　试修改 27.9 节的服务器程序，用 Unix 域数据报套接口代替 Unix 域字节流套接口。需要做哪些改动？

27.3　运行客户，并按你的环境支持极限尽可能多地运行服务器，把你的结果与本章所汇报的结果作比较。

第4部分 XTI:X/Open传输接口编程

第28章 XTI:TCP客户程序

28.1 概 述

套接口API是在1983年随4.2BSD引入的,一开始工作在TCP/IP协议族和Unix域协议上。在80年代中期Posix.1完成之前,AT&T Unix与Berkeley Unix仍存在巨大差异。而在网络界也纷纷盛传,OSI协议"不久"将取代TCP/IP。

AT&T于1986年随SystemV版本3.0(SVR3)推出称为TLI(传输层接口)的另外一种网络编程API。TLI与套接口编程有许多相似之处,不过TLI是按照OSI传输服务定义建模的。除此以外,SVR3还第一次以商业版的形式推出流子系统,我们将在第33章详细讨论。然而,SVR3并未提供任何诸如TCP/IP之类的协议,它只包括了流以及TLI构建模块。这就导致一些软件厂商给SystemV提供第三方的TCP/IP协议以及一些OSI协议的初步实现。1990年的SystemV版本4(SVR4)终于将TCP/IP协议作为基本操作系统的一部分提供。

在1.10节我们提到过X/Open组织。1998年她发布了一个TLI的修正版XTI the X/Open Transport Interface XTI X/Open传输接口)。XTI基本上是TLI的一个超集,并已经历若干个版本。鉴于Posix.1g使用XTI而非TLI,我们这里就只介绍XTI。我们在以后几章叙述的是Unix 98的XTI[Open Group 1997],它与Posix.1g的XTI几乎没什么区别。

XTI使用术语"通信提供者"(communications provider)来描述协议的实现。普遍可用的通信提供者提供的是网际协议,也就是TCP和UDP。另一个术语"通信端点"(communications endpoint)指的是由通信提供者创建和维护并由应用进程使用的对象,这些端点使用描述字来表示。为了简单起见,我们通常将这两者简称为提供者(provider)和端点(endpoint)。

> TLI的相应术语是传输提供者(transport provider)和传输端点(transport endpoint)。

所有的XTI函数名均以t_开始。应用程序需包括的头文件为<xti.h>。部分特定于网际协议的定义在头文件<xti_inet.h>内。

我们将按如下次序介绍XTI:

- TCP客户程序
- 名字与地址函数
- TCP服务器程序
- UDP客户与服务器程序

- 选项
- 流
- 附加函数

对 XTI 的讨论相对于套接口而言要短一些,因为这两者所用的网络编程技术非常相近。主要区别在于函数名、函数参数以及一些本质上的细节(譬如如何接受 TCP 连接),但我们不必把每个套接口编程例子都用 XTI 复述一遍。

28.2　t_open 函数

建立通信端点的第一步是打开代表特定通信提供者的 Unix 设备。该函数返回一个描述字(小整数),它将被其他 XTI 函数所使用。

```
#include <xti.h>
#include <fcntl.h>

int t_open (const char *pathname, int oflag,struct t_info *info);
                                                返回:0——成功,-1——出错
```

实际使用的 pathname 依赖于不同的实现,不过适合于 TCP/IP 端点的典型值为/dev/tcp、/dev/udp 或/dev/icmp。适合于回馈端点的典型值为/dev/ticots、/dev/ticotsord 和/dev/ticlts。

oflag 参数指定打开标志,其值为 O_RDWR。对于非阻塞方式的端点,标志 O_NONBLOCK 应跟 O_RDWR 逻辑相或。

> 这个 XTI 函数与 socket 函数很类似。两者都接受用户给定的协议并返回一个相关的描述字。

t_info 结构由一组整数组成,用来描述提供者的各种与协议有关的特性。如果 info 参数不为空,那么该结构通过指针 info 返回给调用者。这也是我们所遇到的第一个以 t_开头的 XTI 结构。一共有 7 个这样的结构,在 28.4 节我们将再详细介绍。

```
struct t_info {
t_scalar_t addr;     /* max #bytes of communications protocol address */
t_scalar_t options;  /* max #bytes of protocol-specific options */
t_scalar_t tsdu;     /* max #bytes of transport service data unit (TSDU) */
t_scalar_t etsdu;    /* max #bytes of expedited TSDU (ETSDU) */
t_scalar_t connect;  /* max #bytes of data on conn. establishment */
t_scalar_t discon;   /* max #bytes of data on t_XXXdis() & t_XXXrcvdata() */
t_scalar_t servtype; /* service type supported */
t_scalar_t flags;    /* other information (new with XTI) */
```

> 这是我们第一次遇到 t_scalar_t 数据类型,它是由 Unix 98 新引入的。较早的实现使用长整数作为结构中所有成员的数据类型,不过在 64 位体系结构上这么定义存在问题(参见 1.11 节)。t_scalar_t 和 t_uscalar_t 因此被分别定义为

int32_t和uint32_t。

在解释t_info结构的各个成员之前,我们先给出TCP和UDP的一些典型值。见图28.1和图28.2。

	AIX 4.2	DUnix 4.0B	HP-UX 10.30	Solaris 2.6	UnixWare 2.1.2
addr	16	16	16	16	16
options	512	4096	1024	504	360
tsdu	0	0	0	0	0
etsdu	−1	−1	−1	−1	−1
connect	−2	−2	−2	−2	−2
discon	−2	−2	−2	−2	−2
servtype	T_COTS_ORD	T_COTS_ORD	T_COTS_ORD	T_COTS_ORD	T_COTS_ORD

图28.1 TCP的t_info值

	AIX 4.2	DUnix 4.0B	HP-UX 10.30	Solaris 2.6	UnixWare 2.1.2
addr	16	16	16	16	16
options	512	768	256	468	328
tsdu	8192	9216	65508	65508	65508
etsdu	−2	−2	−2	−2	−2
connect	−2	−2	−2	−2	−2
discon	−2	−2	−2	−2	−2
servtype	T_CLTS	T_CLTS	T_CLTS	T_CLTS	T_CLTS

图28.2 UDP的t_info值

对于t_info结构中前6个成员变量的值,我们感兴趣的是三种情况:≥0、−1(也称为T_INFINITE)和−2(也称及T_INVALID)。

addr addr用来给出协议地址所用的最大字节数。−1表示大小无限,−2表示无用户访问本协议地址。

TCP和UDP的这个值为16,它表示sockaddr_in结构的大小。对于IPv6端点,该值也许会是sockaddr_in6结构的大小。

options options用来给出协议特定选项所用的最大字节数。−1表示大小无限,−2表示无用户访问本选项。我们在第32章将进一步讨论XTI选项。

从图28.1和图28.2给出的例子中可看出,XTI选项在不同实现下缺乏一致性,其大小在256～1024字节之间变化。

tsdu TSDU代表传输服务数据单元(transport service data unit),tsdu用来规定端点之间传输数据所用记录的最大字节数。0表示提供者不支持TSDU的概念,但支持字节流数据(也就是说没有记录边界)。−1表示记录大小不限,−2表示不支持传输普通数据(很少见)。

对于 TCP 来说,由于使用字节流方式,所以无所谓记录边界,tsdu 总是为 0。而 UDP 的流行值为 65508,却是错误的:IP 数据报最大长度为 65535 字节(附录中的图 A.1 中 16 位总长度字段),扣去 20 字节 IP 头部以及 8 字节 UDP 头部后剩下的 UDP TSDU 最大大小应为 65507 字节。

etsdu　ETSDU 代表经加速的传输服务数据单元(expedited transport service data unit),etsdu 规定 ETSDU 的最大字节数。在第 21 章,我们称之为带外数据。值 0 表示通信提供者不支持 ETSDU 概念,但支持字节流的带外数据(也就是说在带外数据中不保留记录边界)。−1 表示大小没有限制,−2 表示不支持经加速数据的传输。

正如我们所想,UDP 不支持任何形式的带外数据。而 TCP 则支持 ETSDU,且对应用进程能够发送的带外数据量没有限制(回忆一下 21.2 节中有关 TCP 紧急模式的讨论)。

connect　一些面向连接的协议允许随连接请求附带发送用户数据。connect 成员规定附带发送数据的最大字节数。−1 表示大小无限,−2 表示通信提供者不支持这种特性。

TCP 就不支持这种特性,所以它的值总是−2。而 UDP 不是面向连接,所以它的值也是−2。面向连接的 OSI 传输层支持这种特性。

> 注意,TCP 允许随 SYN 附带数据(见 TCPv3 第 14～16 页)。然而套接口和 XTI API 都不对 TCP 的这种特性提供支持。实际上,t_info 结构里的这个成员指的是不同的东西(例如由 OSI 传输层提供的能力)。

discon　一些面向连接的协议支持在断连请求中附带用户数据。我们将在本章后面讨论 t_snddis 和 t_rcvdis 函数时看到这种可能性。discon 成员给出这份数据的最大字节数。−1 表示对大小没有限制,−2 表示通信提供者不支持这种特性。这个成员同时指定在有序释放时可附带发送的用户数据量。在 34.10 节中,我们将讨论有序释放所用的两个函数:t_sndreldata 和 t_rcvreldata。

TCP 不支持这种特性,该特性由 OSI 传输层提供。

servtype　这个成员规定通信提供者所提供的服务类型。如图 28.3 所示有三种:

servtype	说　明
T_COTS	面向连接的服务,非有序释放方式
T_COTS_ORD	面向连接的服务,有序释放方式
T_CLTS	无连接的服务

图 28.3　通信提供者提供的服务类型

TCP 提供有序释放方式的面向连接服务,而 UDP 则是无连接的。

flags　本成员是 XTI 新加入的,用于指定通信提供者的一些附加标志。图28.4给出的两个常值由<xti.h>头文件定义,它们可作为本成员的返回值。

flags	说 明
T_SENDZERO	提供者支持0长度写
T_ORDRELDATA	提供者支持有序释放数据(t_sndreldata和t_recreldata)

图28.4 t_info结构flags成员的值

TCP不支持0长度写,而UDP支持(导致一个28字节的IP数据报,仅包括一个IP头部和一个UDP头部,不带数据)。TCP同样不支持T_ORDRELDATA。

28.3 t_error和t_strerror函数

回忆一下大多数套接口函数(例如socket、bind、connect,等等),它们在出错时通常返回-1,并通过设置errno变量来提供有关错误的额外信息。XTI函数出错时也通过返回-1来通知调用者,并通过设置t_errno变量来提供有关错误的额外信息。回想23.1节有关errno的讨论,它是一个每个线程一个的变量。在线程环境下,t_errno也必须每个线程有一个。t_errno与errno非常相像,它们仅当错误发生时才被置入一个相应的值,而当函数调用成功返回时,它们的当前值并不清除。

XTI错误码在<xti.h>头文件内定义,并以大写的T开头,例如TBADADDR(错误地址格式)、TBADF(非法传输描述字)等等。

TSYSERR是特殊的错误值,当它在t_errno中返回时,应用进程需要查看errno的值来获取系统错误指示。

函数t_error和t_strerror用来帮助格式化由XTI函数导致的出错消息。

```
#include <xti.h>
int t_error(const char *msg);
                                                    返回:0
const char *t_strerror(int errnum);
                                          返回:出错消息的指针
```

t_error在标准错误输出上输出一条消息。这条消息由参数msg(假设非空)所指的字符串后跟一个冒号和一个空格,再后跟t_errno的当前值对应的出错消息串构成。如果t_error等于TSYSERR,则相应于errno当前值的消息串也输出。最后输出一个换行符。

t_strerror则返回一个描述errnum错误码的字符串,其中errnum假定是某个可能的t_errno值。与t_error不同,当errnum为TSYSERR时,t_strerror不做特殊的处理。

图28.5给出了以上两个XTI函数的用法。此外还使用了err_xti函数(我们将在D.4节内讨论)。

```
1 #include    "unpxti.h"
2 int
3 main(int argc, char **argv)
4 {
5     printf("%s\n", t_strerror(TPROTO));
6     errno = ETIMEDOUT;
7     printf("%s\n", t_strerror(TSYSERR));
8     t_errno = TSYSERR;
9     errno = ETIMEDOUT;
10    t_error("t_error says");
11    t_errno = TSYSERR;
12    errno = ETIMEDOUT;
13    err_xti("err_xti says");
14    exit(0);
15 }
```

图 28.5　t_error 和 t_strerror 函数使用例子[xtiintro/strerror.c]

以下是这个程序的输出：

```
aix % strerror
XTI protocol error
system error
t_error says: system error,Connection timed out
err_xti says: system error,Connection timed out
```

28.4　netbuf 结构和 XTI 结构

为了在应用程序与 XTI 函数之间传递信息，XTI 定义了 7 个结构。我们在 28.2 节已讨论的 t_info 结构就是其中之一。t_info 结构的成员皆为整型值，描述与提供者的协议相关的特性。余下的 6 个结构每个都包含 1～3 个 netbuf 结构。netbuf 结构定义的缓冲区用于在应用程序和 XTI 函数之间交换数据。

```
struct netbuf {
    unsigned int maxlen;    /* maximum size of buf */
    unsigned int len;       /* actual amount of data in buf */
    void        *buf;       /* data (char * before Posix.1g) */
};
```

图 28.6 给出的 6 个 XTI 结构含有 1 个或以上 netbuf 结构，再加上各自的其他成员。

数据类型	XTI 结构					
	t_bind	t_call	t_discon	t_optmgmt	t_uderr	t_unitdata
struct netbuf	addr	addr			addr	addr
struct netbuf		opt		opt	opt	opt
struct netbuf		udata	udata			udata
t_scalar_t					error	
t_scalar_t				flags		
unsigned int	qlen					
int			reason			
int		sequence	sequence			

图 28.6 6个 XTI 结构及它们的成员

这6个包含 netbuf 结构的 XTI 结构总是以引用方式在 XTI 函数与应用程序之间进行传递。也就是说,将 XTI 结构的地址作为参数传给 XTI 函数。这样 XTI 函数就总能读取和修改 netbuf 的3个成员(当然,没有函数会修改 maxlen 这个成员)。

netbuf 结构的三个成员的用途依赖于数据传递的方向:从应用程序到 XTI 函数,还是反过来,如图 28.7 所示。我们还要注意 XTI 是读取成员的值还是向成员内写入值。

成员	数据从应用程序到 XTI	数据从 XTI 到应用程序
maxlen	忽略	只读,表示 buf 所指缓冲区的大小。XTI 函数将存储数据限制在 maxlen 之内(截尾)。如果为 0,则不返回任何数据,len 和 buf 成员被忽略
len	只读,应用程序将这个成员的值设成 buf 所指缓冲区内的数据量	只写,XTI 函数将该成员设为实际存储在 buf 中的数据量,其值总是小于或等于 maxlen
buf	提向数据的指针,该数据由应用程序存储,然后由 XTI 函数处理	指向数据的指针,该数据由 XTI 函数存储,然后由应用程序处理

图 28.7 netbuf 三个成员的处理

如果 XTI 有多于 maxlen 的数据待返回,那么 XTI 调用失败,并置 t_errno 为 TBUFOVFLW。

由于 netbuf 结构是通过其地址传给 XTI 函数的,而且该结构内包含了存储在缓冲区内的数据量(len)以及缓冲区本身的大小(maxlen),所以 XTI 就不再需要套接口 API 使用的变参。

28.5 t_bind 函数

此函数将一个本地地址赋给一个端点,并激活该端点。在 TCP 和 UDP 中,本地地址指 IP 地址和端口号。

```
#include <xti.h>
int t_bind (int fd, const struct t_bind *request, struct t_bind *return);
                                                返回:0——成功,-1——出错
```

t_bind 函数的第二、第三个参数均为指向 t_bind 结构的指针:

```
struct t_bind {
  struct netbuf addr;   /* protocol-specific address */
  unsigned int qlen;    /* max# of outstanding connections (if server) */
};
```

fd 为端点。request 参数有三种情况:

request == NULL:

调用者不关心赋给端点的具体本地地址,由提供者来选择一个地址。qlen 成员假定为 0(见下面)。

request != NULL,但 request->addr.len == 0

调用者不关心赋给端点的具体本地地址,同样由提供者来选择一个地址。但与上一种情况不同的是,调用者可以给 request 的 qlen 成员指定一个非 0 值。

request != NULL,且 request->addr.len > 0

由调用者指定应由提供者赋给端点的本地地址。

无论是由调用者给定本地地址,还是由系统来选择地址,t_bind 都在 return 结构内返回由提供者赋给端点的地址。如果 return 为空指针,则提供者不返回实际地址。

qlen 的值仅对面向连接的服务器有效:它给出了本端点可排队的最大连接数。通信提供者有可能修改这个值,这种情况下在 return 结构的 qlen 成员给出的是提供者实际支持的 qlen 值。在图 30.14 中,我们将进一步讨论这个成员,并对不同的 qlen 值测量实际排队的连接数。

注意:t_bind 结构中的 addr 是一个实实在在的 netbuf 结构,而不是指向这种结构的指针。这对于 XTI 结构很常见:大多数 t_XXX 结构里含有一个或多个 netbuf 结构。

如果 XTI 未能成功地捆绑所请求的地址,它将返回 TADDRBUSY 错误。如果 TLI 遇上这种问题,它将给端点捆绑另一个本地地址,这需要调用者在请求值和返回值之间作比较。

调用者通知提供者选择一个合适地址的 XTI 方法要比 bind 使用的套接口方法更具有通用性。拿 IPv4 上的 TCP 和 UDP 来说,我们必须指定一个 INADDR_ANY 的地址和一个为 0 的端口号才能让系统选择本地地址和端口号。而这是与 IPv4 相关的,对 bind 来说并不通用。

qlen 值与 listen 的 baclog 参数相对应。对于面向连接的服务器而言,t_bind 函数与 bind 及 listen 所做的工作完全相同。

面向连接的 XTI 客户必须先调用 t_bind 再调用 t_connect(下一节叙述),这

与connect有所不同,如果套接口未绑定,后者在内部调用bind。

28.6 t_connect函数

面向连接的客户通过调用t_connect来启动与服务器的连接。客户给出服务器的协议地址(例如TCP服务器的IP地址和端口)。

```
#include <xti.h>
int t_connect(int fd, const struct t_call *sendcall,struct t_call *recvcall);
                                                返回:0——成功,-1——出错
```

第二、三两个参数为指向t_call结构的指针。

```
struct t_call {
  struct netbuf  addr;      /* protocol-specific address */
  struct netbuf  opt;       /* protocol-specific options */
  struct netbuf  udata;     /* user data to accompany connection request */
  int            sequence;  /* for t_listen() & t_accept() functions */
};
```

参数sendcall所指向的t_call结构指定传输提供者建立连接所需的信息:addr结构指定服务器地址,opt指定特定于协议的任选项,udata为建立连接期间将传给服务器的用户数据。(回忆一下28.1节,TCP不支持连接请求附带数据。)sequence结构在t_connect函数中没什么特殊含义,它是由t_accept使用的。

当函数调用返回时,recvcall所指向的t_call结构里含有通信提供者就已建立起的连接所返回的信息。addr为对方进程的地址,opt包含任何特定于协议的相应于该连接的可选数据,udata包含在连接建立期间由对方传输提供者返回的所有用户数据。与sendcall一样,这里的sequence成员没什么意义。

opt结构的内容是与协议相关的,调用者可将该结构的len字段设为0,通知通信提供者使用各连接选项的缺省值。我们将在第32章详细讨论XTI选项。

如果调用者无需知道有关连接的返回信息,recvcall就可以指定为空指针。

缺省情况下,t_connect函数直到连接完成或发生错误时才返回。在34.3节,我们将讨论如何进行非阻塞的连接。

从4.3节我们得知,建立TCP连接时最常见的错误为:收到RST、收到ICMP目的地不可达及超时。不幸的是,当以上常见错误出现时,t_connect只是返回-1,并置t_errno为TLOOK,具体原因还需额外编码检查。在28.9和28.10节我们将讨论这个问题,并在图28.13中给出一个例子。

t_connect函数与connect函数相似。

28.7 t_rcv和t_snd函数

缺省情况下,XTI应用程序无法调用通常的read和write函数(除非tirdwr模块被压入

流中,见 28.12 节)。XTI 应用程序必须调用 t_rcv 和 t_snd 函数:

```
#include <xti.h>
int t_rcv(int fd, void *buff, unsigned int nbytes, int *flagsp);
int t_snd(int fd, const void *buff, unsigned int nbytes ,int flags);
                                两者均返回:读或写的字节数——成功,-1——出错
```

前三个参数与 read 和 write 相似,分别是:描述字、缓冲区指针和准备读/写的字节数。

套接口 API 中的输入输出函数都使用 size_t 作为缓冲区大小的数据类型,使用 ssize_t 作为返回值的数据类型。而 XTI 函数则分别使用 unsigned int 和 int。

t_snd 函数的 flags 参数要么为 0,要么是图 28.8 中常值的组合。

flag	描　述
T_EXPEDITED	收发经加速(带外)数据
T_MORE	有过量数据待收发

图 28.8　t_rcv 和 t_snd 的 flags 参数

如果要使用 t_snd 发送带外数据(34.12 节),则设置 T_EXPEDITED。如果收到带外数据,则 t_rcv 在返回时设置该值。

当使用多个 t_snd 或 t_rcv 调用来发送一个大数据量记录时(仅适用于支持记录概念的协议),需设置 T_MORE。我们将在图 31.7 随 t_rcvudata 函数和面向记录的 UDP 协议给出一个使用这个标志的例子。读 TCP 带外数据时也可使用这个标志(我们将在 34.12 节叙述),但 TCP 的普通数据从不用该标志。

XTI 定义了 T_PUSH 标志,通知提供者将所有累积未发送的数据发送出去。不过该标志是由 SNA(IMB 的 Systems Network Architecture)上的 XTI 使用的,TCP 上的 XTI 并不使用,说得更明确些就是它不引起 TCP 的 PUSH 标志的设置。

注意,t_snd 函数的 flags 参数是一个整数值,t_rcv 的相应参数则是一个整数指针。但 t_rcv 的 flagsp 参数却不是一个真正的变参,因为 t_rcv 函数并不检查它的值,它只是在返回时才被设置。

t_snd 和 t_rcv 都返回实际写出或读到的字节数。如果端点是非阻塞的,或者发送中途进程捕获一个信号,则 t_snd 的返回值可能小于 nbytes。

以上两个函数对应于 send 和 recv 函数,而 XTI 的 T_EXPEDITED 标志对应于 MSG_OOB 标志,不过对于 XTI 我们无法给 t_rcv 设置该标志。

回想一下,当 TCP 套接口收到 FIN 时,read 返回 0,而收到 RST 时,read 返回 -1,errno 设置为 ECONNRESET。当 XTI 端点上发生这些情况时,t_rcv 的反应有所不同:

- 当XTI端点收到TCP FIN后,t_rcv返回−1,并置t_errno为TLOOK。应用程序需要接着调用XTI函数t_look,该函数返回T_ORDREL。这称为顺序释放指示(orderly release indication)。
- 当XTI端点收到TCP RST后,t_rcv同样返回−1,并置t_errno为TLOOK。应用程序也需要接着调用t_look,不过此时它返回的是T_DISCONNECT。这称为断连(disconnect)或夭折性释放(abortive release)。

下面我们将先讨论t_look函数,接着讨论顺序释放和夭折性释放函数。

28.8 t_look函数

在XTI端点可能异步地发生许多事件。也就是说,当应用进程在端点上执行某个任务时,很可能有一件不相关的事件发生。它们有可能指示一个错误(譬如T_UDERR,先前发送数据报中的一个错误),也可能不是错误(如T_EXDATA,经加速数据的到达)。

例如,假设应用进程调用t_snd向对方发送数据,然而就在此前,由于对方发生某事而导致对方进程发送一个RST并终止。这个意外事件(在调用t_snd时收到RST)将通过让t_snd返回−1,并置t_errno为T_LOOK传递给应用进程。应用进程接着调用t_look来确定到底在端点发生了什么。本例中所发生的事件为T_DISCONNECT,表示收到断连指示(RST)。

```
#include <xti.h>
int t_look (int fd);
                                返回:事件(图28.9)——成功,−1——出错
```

t_look函数返回的整数值参见图28.9中的9种事件。

当在XTI端点发生一个事件时,它被认为是未处理的(outstanding),直到被消费掉为止。图28.10给出了消费(conswning)各个XTI事件的XTI函数,可看出调用t_look消费两个事件。

事 件	描 述
T_CONNECT	收到连接确认
T_DATA	收到普通数据
T_DISCONNECT	收到断连
T_EXDATA	收到经加速数据
T_GODATA	普通数据流量控制解除
T_GOEXDATA	经加速数据流量控制解除
T_LISTEN	收到连接指示
T_ORDREL	收到顺序释放指示
T_UDERR	先前发送数据报出错

图28.9 XTI端点的事件

事　件	由 t_look 清除?	消费函数
T_CONNECT		t_connect,t_rcvconnect
T_DATA		t_rcv,t_rcvv,t_rcvudata,t_rcvvudata
T_DISCONNECT		t_rcvdis
T_EXDATA		t_rcv,t_rcvv
T_GODATA	是	t_snd,t_sndv,t_sndudata,t_sndvudata
T_GOEXDATA	是	t_snd,t_sndv
T_LISTEN		t_listen
T_ORDREL		t_rcvrel
T_UDERR		t_rcvuderr

图 28.10　XTI 事件及消费这些事件的函数

在阻塞方式的端点上(缺省),T_CONNECT 事件由 t_connect 函数自身处理,应用进程是看不到的。对于我们以前考虑的例子(调用 t_snd 时收到 FIN),我们必须调用 t_rcvrel 来清除相应事件(T_ORDREL)。

> 在本节的开始,我们描述了一个 RST 信号是如何引起端点的 T_DISCONNECT 事件的。直到 Unix 98 之前,收到 FIN 总是导致端点上产生 T_ORDREL 事件,而 Unix 98 使之成为可选的。

28.9　t_sndrel 和 t_rcvrel 函数

XTI 支持两种断连方式:顺序释放和夭折性释放。其区别在于,夭折性型释放不保证所有累积未处理数据的递送,而顺序释放则保证这一点。任何通信提供者都必须支持夭折性释放,而顺序释放则是可选的。不过幸运的是,TCP 支持顺序释放。

我们可以使用以下函数来进行顺序释放的发送和接收。

```
#include <xti.h>
int t_sndrel(int fd);
int t_rcvrel(int fd);
```

二者均返回:0——成功,−1——出错

为了理解顺序释放的语义,我们有必要提醒一下,面向连接的协议通常是两个进程间的全双工连接。一个方向上传输的数据与另一个方向上的无关。图 28.11 给出了使用 TCP 时各函数的一种用法,它利用了 TCP 的半关闭功能(关闭一个方向上的连接)。

进程通过调用 t_sndrel 来发起顺序释放。这通知提供者应用进程已无其他数据需在端点发送。对于 TCP 端点,它将给对方发送一个 FIN(等所有积压在队列中的数据发送出去以后)。调用 t_sndrel 的进程仍可继续接收数据,但它不能再发送数据。

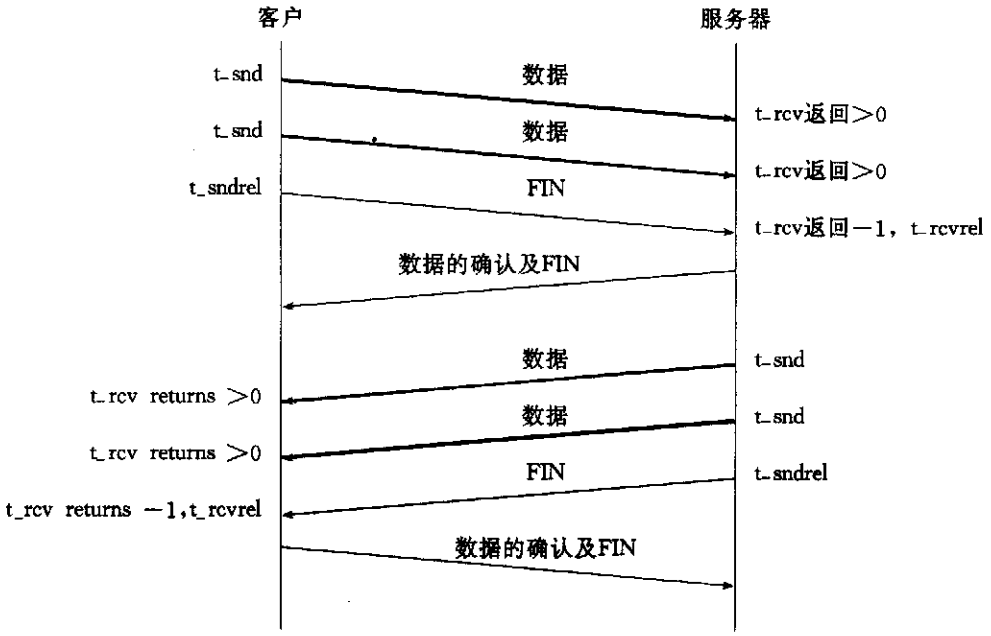


图 28.11 使用 XTI 的 TCP 半关闭

该函数与 TCP 套接口上第二参数为 SHUT_WR(1)时的 shutdown 功能相同。

进程通过调用 t_rcvrel 来确认收到连接释放。此后该进程只能发送数据而不能再接收数据。

套接口 API 中并无与 t_rcvrel 相对应的函数，收到的 FIN 将被视为文件结束符递送给进程，例如 read 返回 0。

XTI 的这个特性强迫应用程序处理全双工的顺序释放，即使应用程序无意使用该特性也不行，在图 28.13 我们将看到这一点。

28.10 t_snddis 和 t_rcvdis 函数

以下函数处理夭折性释放(断连)：

```
#include <xti.h>
int t_snddis(int fd, const struct t_call *call);
int t_rcvdis(int fd,struct t_discon *discon);
                                    二者均返回:0——成功,-1——出错
```

t_snddis 函数有两种用途：

- 对于已有连接进行夭折性释放，如果是 TCP 则导致发送一个 RST
- 拒绝连接请求

对于已有连接的夭折性释放,参数 call 可以是空指针,这种情况下不给对方进程发送任何信息。否则,对 t_call 结构内各成员的解释参见图 28.12。

成员	断开已有连接	拒绝新的连接
addr	忽略	忽略
opt	忽略	忽略
udata	可选	可选
sequence	忽略	必需

图 28.12　t_snddis 使用的 t_call 结构

可选的 udata 成员指定伴随断连发送的用户数据,不过从图 28.1(t_info 结构的 discon 成员)我们可以看出 TCP 并不支持这种数据。

> 对于套接口应用程序,夭折性释放由如下操作产生:设置 SO_LINGER 套接口选项,将 l_onoff 设为非 0 值,l_linger 设为 0,接着关闭套接口(第 7 章)。

当 T_DISCONNECT 事件在 XTI 端点发生之后(譬如 TCP 接收到 RST),应用进程必须调用 t_rcvdis 来接收夭折性释放。如果 discon 参数为非空指针,则其 t_discon 结构将被填入夭折性释放的原因。

```
struct t_discon {
struct netbuf   udata;      /* user data */
  int           reason;     /* protocol-specific reason code */
  int           sequence;
};
```

udata 成员含有伴随断连而来的可选用户数据,reason 是与协议相关的断连原因,sequence 只对接收连接的服务器适用。

> 套接口 API 中没有什么函数对应于 t_rcvdis 函数。RST 的接收将作为输入错误(例如 read 返回 −1)递送给进程,同时置 errno 为 ECONNRESET。再向这样的套接口写入将引发 SIGPIPE 信号。

28.11　XTI TCP 时间/日期客户程序

下面我们用 XTI 函数重写图 1.5 的 TCP 时间/日期客户程序。请看图 28.13。

```
1 #include    "unpxti.h"
2 int
3 main(int argc, char **argv)
4 {
5     int           tfd, n, flags;
6     char          recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     struct t_call tcall;
9     struct t_discon tdiscon;
10    if (argc != 2)
11        err_quit("usage: daytimecli01 <IPaddress>");
12    tfd = T_open(XTI_TCP, O_RDWR, NULL);
```

```
13    T_bind(tfd, NULL, NULL);
14    bzero(&servaddr, sizeof(servaddr));
15    servaddr.sin_family = AF_INET;
16    servaddr.sin_port   = htons(13);     /* daytime server */
17    inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
18    tcall.addr.maxlen = sizeof(servaddr);
19    tcall.addr.len = sizeof(servaddr);
20    tcall.addr.buf = &servaddr;
21    tcall.opt.len = 0;        /* no options with connect */
22    tcall.udata.len = 0;      /* no user data with connect */
23    if (t_connect(tfd, &tcall, NULL) < 0) {
24        if (t_errno == TLOOK) {
25            if ( (n = T_look(tfd)) == T_DISCONNECT) {
26                tdiscon.udata.maxlen = 0;
27                T_rcvdis(tfd, &tdiscon);
28                errno = tdiscon.reason;
29                err_sys("t_connect error");
30            } else
31                err_quit("unexpected event after t_connect: %d", n);
32        } else
33            err_xti("t_connect error");
34    }
35    for ( ; ; ) {
36        if ( (n = t_rcv(tfd, recvline, MAXLINE, &flags)) < 0) {
37            if (t_errno == TLOOK) {
38                if ( (n = T_look(tfd)) == T_ORDREL) {
39                    T_rcvrel(tfd);
40                    break;
41                } else if (n == T_DISCONNECT) {
42                    tdiscon.udata.maxlen = 0;
43                    T_rcvdis(tfd, &tdiscon);
44                    errno = tdiscon.reason; /* probably ECONNRESET */
45                    err_sys("server terminated prematurely");
46                } else
47                    err_quit("unexpected event after t_rcv: %d", n);
48            } else
49                err_xti("t_rcv error");
50        }
51        recvline[n] = 0;   /* null terminate */
52        fputs(recvline, stdout);
53    }
54    exit(0);
55 }
```

图 28.13 使用XTI的时间/日期客户程序[xtiintro/daytimecli01.c]

unpxti.h头文件

第1行 我们定义自己的unpxti.h头文件,我们的所有XTI程序都#include该头文件。我们在D.3节给出该头文件。

创建端点,bind任意本地地址

第12～13行 使用t_open创建XTI端点,通过将第二参数置为空指针调用t_bind,让系统为该端点选择本地协议地址。

指定服务器地址和端口

第 14～22 行　我们将服务器的 IP 地址和端口填入一个网际套接口地址结构,并将它的地址指针存入一个 t_call 结构(tcall)中,并设置其中的 opt 结构和 udata 结构的 len 成员均为 0,以表示无选项和无用户数据。

> XTI 并未规定 t_call 结构必须指向 IPv4 的 sockaddr_in 结构。但几乎所有的 Unix 系统在实现网际协议上的 XTI 时,都使用 sockaddr_in 结构作为应用进程和提供者之间传递协议地址的手段。如果要做到协议无关,我们应该将该结构的使用对应用程序隐藏起来。下一章我们将介绍如何做。

建立连接

第 23～34 行　调用 t_connect 建立连接,即完成 TCP 的三路握手过程。如我们前面所提到的,如果连接建立因某个常见错误而失败,那么 t_connect 返回 TLOOK,我们于是调用 t_look 找出事件,如果是 T_DISCONNECT 事件则调用 t_rcvdis。这种情况下我们还将断连原因存入 errno,并调用 err_sys 函数输出相应出错消息。

从服务器读取信息,再输出至标准输出

第 35～53 行　使用 t_rcv 从服务器读取数据,并将它写到标准输出上,直至连接结束。

处理顺序释放和断连

第 37～47 行　当 t_rcv 返回错误时,查看 t_errno,如果是 TLOOK,则调用 t_look 函数获取本端点的当前事件。如果事件为 T_ORDREL,则调用 t_rcvrel;如果是 T_DISCONNECT,则调用 t_rcvdis。

如果我们像 4.3 节一样运行我们的程序,将看到如下输出。首先与运行时间/日期服务器程序的主机相连。

```
unixware % daytimecli01 206.62.226.35
Tue Feb 4 15:00:26 1997
```

其次,指定一台不存在的本地子网主机。

```
unixware % daytimecli01 206.62.226.55
t_connect error: Connection timed out
```

然后,给定一台未运行时间/日期服务器程序的路由器,它对我们的 SYN 响应以一个 RST。

```
unixware % daytimecli01 140.252.1.4
t_connect error: Connection refused
```

最后,给定一个未连接到因特网上的 IP 地址,作为对我们的多个 SYN 的响应的是一个 ICMP 主机不可达错误。

```
unixware % daytimecli01 192.3.4.5
t_connect error: No route to host
```

XTI 与套接口 API 的互操作性

在第一个例子中,我们的 XTI 时间/日期客户与主机 bsdi(206.62.226.35)上的服务器建立连接。然而,我们的客户程序是用 XTI API 编写的,而服务器程序是用套接口 API 编写

的,不过客户与服务器的通信不成问题。类似地,用XTI API编写服务程序,用套接口API编写客户程序,也工作得很好。

其实互操作性是由网际协议族提供的,而与套接口和XTI无关。一个使用TCP或UDP的客户可以和使用同样传输协议的服务器很好地互操作,只要两者采用相同的应用协议(application protocol),而与开发客户程序和服务器程序的API无关。应用协议(例如HTTP、FTP、Telnet,等等)和传输层(TCP或UDP)决定互操作性,采用什么API编写客户或服务器程序则无关紧要。

28.12 xti_rdwr函数

正如前一节所述,作为缺省配置,我们无法在指代某个XTI端点的描述字上使用read和write。试着修改一下图28.13的程序,用read和write代替t_rcv和t_snd拷贝图1.5的循环,运行结果就可能是这样的:

```
unixware % daytimecli03 206.62.226.35
read error: Not a data message
```

在AIX下有相同的结果。不过HP-UX和Solaris服务器的响应是先读入客户数据,再返回错误:

```
hpux % daytimecli03 198.69.10.4
Wed Apr     2 18:59:40   1997
read error: Bad message
```

我们将在图33.11中解释这两种情形的差异。

幸运的是,我们通常可以绕过这个问题。如果我们有一个基于流机制实现的XTI(通常如此),我们可以向流内压入tirdwr流模块,然后就可以使用read和write了。图33.3展示了相关的流模块。不过,既然该能力是与实现相关的,我们并不把实际的ioctl命令直接放到程序中,而是定义一个简单的函数。这样,一旦系统实现有变化,只需要修改库函数就可以了。

```
#include "unpxti.h"
int xti_rdwr(int fd);
                                        返回:0——成功,-1——出错
```

图28.14给出了该函数的一个简单实现,仅压入流模块tirdwr。

```
1 #include    "unpxti.h"
2 int
3 xti_rdwr(int fd)
4 {
5     return (ioctl(fd,I_PUSH,"tirdwr"));
6 }
```

图28.14 xti_rdwr:向流内压入tirdwr流模块[libxti/xti_rdwr.c]

在使用时有几点要注意:

- 流模块仅可在端点处于数据传输阶段使用。对于我们的例子程序(图 28.13),我们将 xti_rdwr 调用放在 t_connect 成功返回之后。
- 当流模块压入之后,所有 XTI 函数(t_打头)将无法调用。
- 该模块不能被使用带外数据的应用程序使用， 因为带外数据的到达无法用 read 处理。
- 收到一个顺序释放后,read 返回 0(即文件结束符)。
- 不幸的是,当收到一个断连时,read 也返回 0,从而无法区别(正常的)FIN 接收和(异常的)RST 接收。RST 的接收还导致所有的后续 write 失败。

使用套接口情况下,RST 的接收引起 read 返回 −1,并置 errno 为 ECONNRESET。

28.13　小　结

XTI 客户程序和套接口客户程序很类似,t_open 对应于 socket,t_connect 对应于 connect。两者使用同样的网际套接口地址结构来指定服务器协议地址,不过 XTI 中该结构用另外一个 netbuf 结构来描述。缺省情况下,XTI 使用 t_rcv 和 t_snd 代替 read 和 write 来读写数据,而后两者的使用则取决于环境,我们已看到在流环境下,这两个函数的使用需向流内压入另外一个流模块。

XTI 在端点上定义了 9 种事件。当事件发生时,XTI 函数会返回一个 TLOOK 错误。通过调用 t_look 函数,我们可以确定是什么事件并处理它们。相对于套接口编程,处理这些事件往往会增加必须编写的代码量。

28.14　习　题

28.1　在 t_bind 的第二种情况下,我们说 request 参数非空,但该结构的 addr.len 成员又为 0,从而允许调用者给 qlen 成员指定一个非 0 值。这种情况有用吗?

28.2　当针对不可达主机 192.3.4.5 运行图 28.13 的程序时,我们说过接收到了作为对我们的多个 SYN 的响应的一个 ICMP 主机不可达错误。为什么我们说是多个 SYN 呢?

28.3　在 28.8 节的开头我们描述了这样一种情况:当应用进程调用 t_snd 时已在连接上收到一个 RST。把这种情况跟已在连接上接收到一个 RST 时套接口 API 如何处理 write 作比较。

28.4　编写一个 xti_read 函数,它接受与 read 一样的参数,不过用 t_rcv 来实现,并处理图 28.13 的两种情况:(1)收到一个顺序释放则返回 0;(2)收到一个断连则返回 −1,并置 errno 为原因。

第29章 XTI:名字与地址函数

29.1 概 述

XTI本身并未在名字与地址之间的转换上作什么规定。Unix 98在这方面所需的仅仅是在第9章中讨论过的一些函数:gethostbyname、gethostbyaddr、getservbyname,等等。然而,由于大多数XTI是在源自SVR4的系统上实现的,它们大多提供源自SVR4的地址与名字函数:netconfig和netdir。它们在SVR4中被称为"网络选择及名字到地址映射函数"。

在28.11节的客户程序例子里,我们用IP地址和端口号填写一个网际套接口地址结构,这就与协议相关了。本章的目的就是把存放地址的netbuf结构作为一个不透明结构来处理,避免知道其内容。我们应该从主机名与服务名开始,调用一些函数,结果就是一个已准备好的netbuf结构,譬如说在TCP客户程序中马上就能用它来调用t_connect。这与11.2节的getaddrinfo函数的用法类似。

> 我们将叙述过的函数不在任何标准的范围之内,这样导致的问题之一就是缺乏关于它们如何操作的明确说明。譬如说,对于大多数实现而言,netdir_getbyname既接受端口号,也接受TCP或UDP服务名;然而有些实现却只接受名字,而不接受端口号。

29.2 /etc/netconfig文件与netconfig函数

XTI名字与地址映射机制的起始点在/etc/netconfig文件,它是一个文本文件,每行对应一种受支持的协议。图29.1给出了各种协议每一栏的典型值。

网络ID	语义	标志	协议族	协议名	设备
tcp	tpi_cots_ord	v	inet	tcp	/dev/tcp
udp	tpi_clts	v	inet	udp	/dev/udp
icmp	tpi_raw	-	inet	icmp	/dev/icmp
rawip	tpi_raw	-	inet	-	/dev/rawip
ticlts	tpi_clts	v	loopback	-	/dev/ticlts
ticots	tpi_cots	v	loopback	-	/dev/ticots
ticotsord	tpi_cots_ord	v	loopback	-	/dev/ticotsord
spx	tpi_cots_ord	v	netware	spx	/dev/nspx2
ipx	tpi_clts	v	netware	ipx	/dev/ipx

图29.1 /etc/netconfig文件的典型项

这个文件每行实际有 7 栏，我们没有给出的是最后一栏，这一栏指定用于本网络目录查寻的一个或多个函数库。网际协议最后一栏的典型值为/use/lib/tcpip. so 和/usr /lib/resolv. so。这些函数库通常都是动态链接库，提供名字到地址转换的网络特定部分。

网际协议的网络 ID 正是我们预期的 4 个。下面 3 行是回馈项(其中 ti 代表“传输独立”)，最后 2 行是 Novell Netware 协议的(本书我们不讨论)。

网际协议的网络语义值跟图 28.1 所示的服务类型是对应的，例外是 tpi_raw，它既用于 ICMP，也用于原始 IP。注意 TCP 提供具有顺序释放的面向连接服务，这跟图 28.1 是一致的。

当前定义了的唯一标志是 v，它意味着该标志所在项对于 NETPATH 库例程(稍后叙述)是可见的。

设备名用作 t_open 的参数。

网络服务函数库提供读 netconfig 文件的许多函数。其中函数 setnetconfig 打开该文件，函数 getnetconfig 然后读该文件中的下一项。endnetconfig 关闭该文件，并释放已分配的任何内存。

网络服务函数库(network services library)这个称谓出自系统 V，通常指的是以-lnsl 命令行参数指定给链接器的那个库。该库(譬如说/usr/lib/libnsl. so)包含所有的 XTI 库函数及我们将叙述的其他函数。

```
#include <netconfig.h>
void *setnetconfig(void);
                                        返回：非空指针——成功，NULL——出错
struct netconfig *getnetconfig(void *handle);
                                        返回：非空指针——成功，NULL——文件结束
int endnetconfig(void *handle);
                                        返回：0——成功，-1——出错
```

setnetconfig 返回一个指针(称为句柄)，它作为下两个函数的参数。文件中的每一项都被作为一个 netconfig 结构返回。

```
struct netconfig {
  char          *nc_netid;        /* "tcp", "udp", etc. */
  unsigned long nc_semantics;     /* NC_TPI_CLTS, etc. */
  unsigned long nc_flag;          /* NC_VISIBLE, etc. */
  char          *nc_protofmly;    /* "inet", "loopback", etc. */
  char          *nc_proto;        /* "tcp", "udp", etc. */
  char          *nc_device;       /* device name for network id */
  unsigned long nc_nlookups;      /* #of entries in nc_lookups */
  char          **nc_lookups;     /* list of lookup libraries */
  unsigned long nc_unused[8];
};
```

前 6 个成员对应于图 29.1 的 6 列。如果我们编写如下轮廓的程序：

```
void *handle;
struct netconfig *nc;

handle = setnetconfig();
while ((nc = getnetconfig(handle)) != NULL) {
    /* print netconfig structure */
}
endnetconfig(handle);
```

并假设netconfig文件如图29.1所示,那么我们将按文件中的顺序输出9个netconfig结构。

29.3 NETPATH环境变量与netpath函数

通过getnetconfig函数,我们可以逐行遍历整个netconfig文件。然而对于交互式程序(典型的是客户程序),用户只关心特定的几种协议。通过设置名为NETPATH的环境变量和调用以下函数,可以做到这一点(无需使用netconfig函数)。

```
#include <netconfig.h>
void *setnetpath(void);
                                        返回:非空指针——成功,MULL——出错
struct netconfig *getnetpath(void *handle);
                                    返回:非空指针——成功,NULL——文件结束
int endnetpath(void *handle);
                                                返回:0——成功,-1——出错
```

例如,在Korn Shell下如下设置该环境变量:

```
export NETPATH=udp:tcp
```

然后如下编程:

```
void *handle;
struct netconfig *nc;

handle = setnetpath();
while ((nc = getnetpath(handle)) !=NULL) {
    /* print netconfig structure */
}
endnetpath(handle);
```

结果只输出两项,一项是TCP协议的,一项是UDP协议的;先后次序与两者在环境变量内出现的次序相同,与在netconfig文件内出现的次序无关。

如果没有设置NETPATH这个环境变量,所有可见项都被返回,且按netconfig文件内的次序输出。

29.4　netdir 函数

netconfig 和 netpath 函数可以用来寻找所需要的协议。我们还需要根据使用 netconfig 或 netpath 函数获得的协议来查找主机名和服务名。这是由 netdir_getbyname 函数提供的。

```
#include <netdir.h>
int netdir_getbyname (const struct netconfig *ncp,
                      const struct nd_hostserv *hsp,
                      struct nd_addrlist **alpp);
                                                返回:0——成功,非 0——出错
void netdir_free(void *ptr, int type);
```

netdir_getbyname 将一个主机名和服务名转换成一个地址。ncp 指向一个由 getnetconfig 或 getnetpath 返回的 netconfig 结构。而 hsp 则指向一个已由应用进程填入主机名和服务名的 nd_hostserv 结构。

```
struct nd_hostserv {
  char    *h_host;          /* hostname */
  char    *h_serv;          /* service name */
};
```

第三个参数是指向 nd_addrlist 结构的指针,如果成功返回,*alpp 将含有一个指向如下结构之一的指针。

```
struct nd_addrlist {
  int           n_cnt;     /* number of netbufs */
  struct netbuf *n_addrs;  /* array of netbufs containing the addrs */
};
```

注意,nd_addrlist 结构指向一个含有一个或多个 netbuf 结构的数组,每个 netbuf 结构都含有相应主机的一个地址。(回忆一下主机可以是多宿的。)

例如对于图 11.1 的例子,主机名为 bsdi(拥有两个 IP 地址),服务名为 domain(TCP 及 UDP 端口 53),图 29.2 给出了 netdir_getbyname 返回的信息,前提是用作该函数第一个参数的 netconfig 结构是 TCP 协议项。

> 我们再次假设提供者用来表示网际地址的结构为 sockaddr_in 结构。虽然这很普通,但这并不是必须的。

本例中 netdir_getbyname 的最后一个参数是指向 alp 变量的指针。

当我们使用完这些动态分配的结构之后,需要释放它们。netdir_free 提供这个功能,我们将 type 置成 ND_ADDRLIST,它将释放指向 nd_addrlist 结构的 ptr 指针。

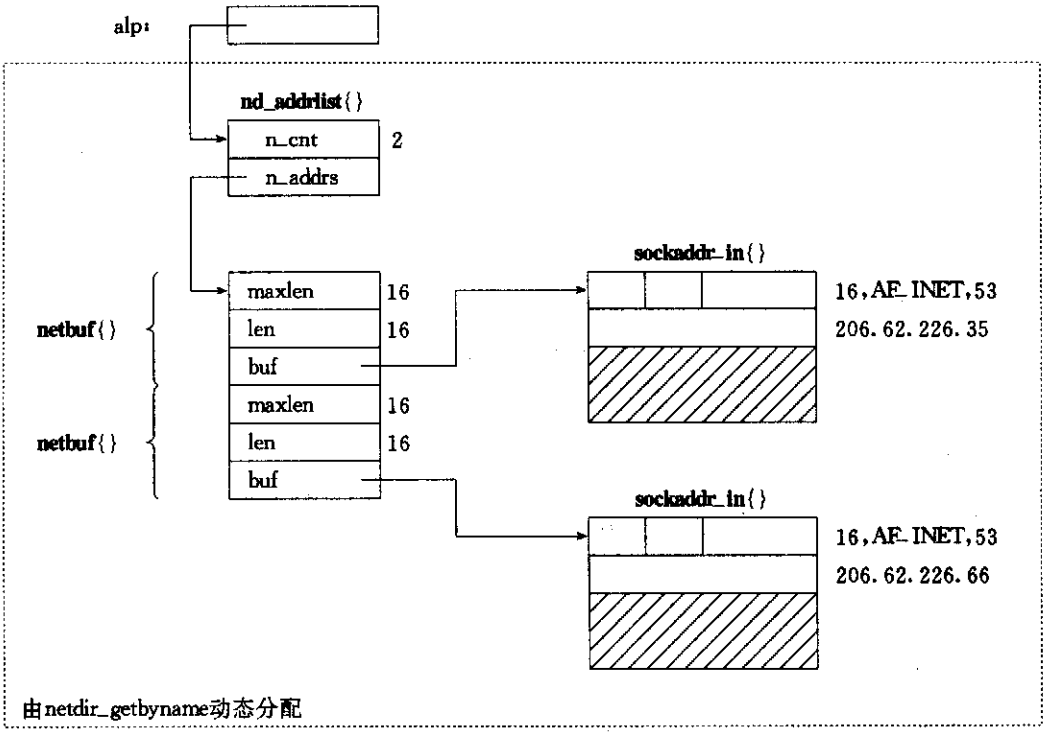


图 29.2 netdir_getbyname 返回的数据结构

逆向转换,即给定一个含有地址的 netbuf 结构,返回主机名与服务名,由 netdir_getbyaddr 函数提供。

```
#include <netdir.h>
int netdir_getbyaddr(const struct netconfig * ncp,
                     struct nd_hostservlist * * hslpp,
                     const struct netbuf * addr);
                                          返回:0——成功,非0——出错
```

该函数的第一、第三参数用来向函数提供输入:一个是指向 netconfig 结构的指针,一个是指向 netbuf 结构的指针。而结果则是存放在 * hslpp 中的指向 nd_hostservlist 结构的指针。

```
struct nd_hostservlist {
int      h_cnt;                    /* number of nd_hostservs */
struct nd_hostserv * h_hostservs;  /* the hostname/service-name pairs */
};
```

该结构指向由一个或多个 nd_hostserv 结构组成的数组。无论是 nd_hostservlist 结构、它所指的 nd_hostserv 结构数组还是各个 nd_hostserv 结构中的主机名及服务名所在的字符串,都是动态分配的。在用完之后,设置 type 为 ND_HOSTSERVLIST 调用 netdir_free 来释放它们。

29.5　t_alloc 和 t_free 函数

作为一个 API，对于协议独立性的要求之一就是在无需知道地址格式的情况下，获得协议地址的大小。对于套接口 API，我们通过调用 getaddrinfo 函数(11.2 节)，从它返回的 addrinfo 结构中的 ai_addrlen 成员获得这项信息。对于 XTI API，t_open 函数返回的 t_info 结构中的 addr 成员提供地址大小。

在获得地址大小后，下一步就是动态地分配相应的数据单元。对于套接口 API 而言，我们只需使用套接口地址结构，并在适当时候调用 malloc 动态分配就可以了(例如图 27.5)。而对于 XTI API，我们有 6 个结构(图 28.6)，每个结构有一至多个 netbuf 结构，而每个 netbuf 结构又指向一个其大小取决于协议地址大小的缓冲区(譬如 28.6 节里 t_call 结构的 addr 成员)。为了简化这些 XTI 结构及它们所包含的 netbuf 结构的动态分配，XTI 提供了 t_alloc 和 t_free 函数。

```
#include <xti.h>
void *t_alloc(int fd, int structtype, int fields);
                                    返回：非空指针——成功，NULL——出错
int t_free(void *ptr, int structtype);
                                    返回：0——成功，-1——出错
```

structtype 参数需为图 29.3 所示的某个常值，用于指定分配或释放的是 7 种 XTI 结构中的哪一种。

fields 参数用于设定哪些 netbuf 结构需同时分配并初始化。fields 是图 29.4 所示常值的按位或。回忆一下图 28.6，netbuf 结构作为成员在 7 种 XTI 结构中总是命名为 addr、opt 或 udata。

structtype	结构类型
T_BIND	struct t_bind
T_CALL	struct t_call
T_DIS	struct t_discon
T_INFO	struct t_info
T_OPTMGMT	struct t_optmgmt
T_UDERROR	struct t_uderr
T_UNITDATA	struct t_unitdata

图 29.3　t_alloc 和 t_free 使用的 structtype 参数

fields	分配及初始化
T_ALL	所给定结构中所有相关的字段
T_ADDR	t_bind、t_call、t_uderr 或 t_unitdata 中的 addr 字段
T_OPT	t_optmgmt、t_call、t_uderr 或 t_unitdata 中的 opt 字段
T_UDATA	t_call、t_discon 或 t_unitdata 中的 udata 字段

图 29.4 t_alloc 中 fields 参数

对 fields 参数给出不同值的原因在于 XTI 结构一般含有多个 netbuf 结构，而我们有时并不想给所有的缓冲区分配空间。譬如 28.6 节所给的 t_call 结构就含有三个 netbuf 结构。

```
struct t_call {
  struct netbuf  addr;      /* protocol-specific address */
  struct netbuf  opt;       /* protocol-specific options */
  struct netbuf  udata;     /* user data */
  int            sequence;  /* applies only to t_listen() func */
};
```

组合使用 T_ADDR、T_OPT 及 T_UDATA 给予我们对空间分配的完全控制。不过为了简单起见，我们的例子通常就用 T_ALL(参见习题 29.2)。

给定图 28.1 中有关 AIX 4.2 的各个值，如果我们置 fd 参数为某个 TCP 端点的描述字，structtype 为 T_CALL，fields 为 T_ALL 来调用 t_call，我们将得到如图 29.5 的返回值。

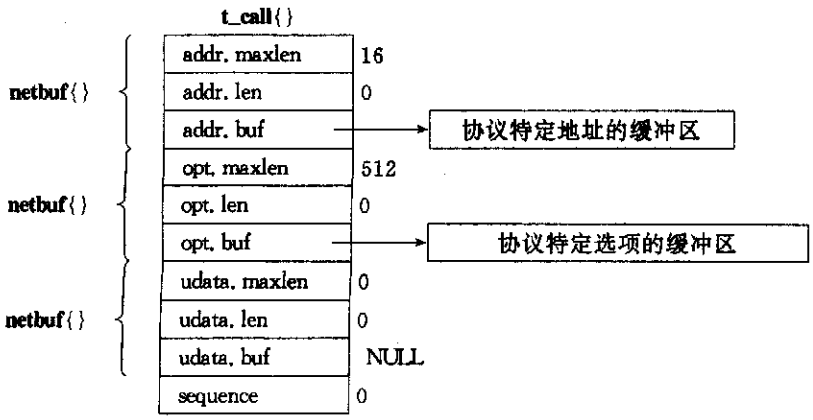


图 29.5 使用 t_alloc 分配结构和缓冲区

这个 t_alloc 调用分配一个 t_call 结构，它包含三个 netbuf 结构。其中有两个 netbuf 结构分配了缓冲区：一个是存放协议特定地址的(对应 addr 成员)，另一个是存放协议特定选项的(对应 opt 成员)。这两个 netbuf 结构(t_call 结构的 addr 成员和 opt 成员)由 t_alloc 初始化 buf 指针及 maxlen 成员，len 成员则设成 0。第三个 netbuf 结构在 TCP 中用不到(TCP 不支持 udata)，所以 udata 成员的 maxlen 和 len 均被置为 0，缓冲区指针则设成空指针。

t_free 函数用于释放 t_alloc 所分配的结构。structtype 参数指定结构类型(使用图 29.3 的常值)。t_free 在释放为由 structtype 指定的结构所分配的空间前，首先检查相应 XTI 结构中包含的 netbuf 结构，并释放这些缓冲区所占的空间。如图 29.5 的例子，t_free 首先释放

两个缓冲区，再释放 t_call 结构。

29.6　t_getprotaddr 函数

t_getprotaddr 可用于返回与端点相关的本地及对方协议地址。

```
#include <xti.h>
int t_getprotaddr(int fd, struct t_bind *localaddr, struct t_bind *peeraddr);
                                        返回：0——成功，-1——出错
```

该函数使用两个 t_bind 结构的 addr 成员（一个 netbuf 结构）作为参数。当函数被调用时，netbuf 结构的 maxlen 和 buf 成员给出结果存在何处。netbuf 结构的 maxlen 成员用于给出是否需要返回地址，如 maxlen 为 0 则表示无需返回该地址。返回时 netbuf 的 len 成员含有存储在 buf 成员中的地址的大小。对于本地地址，如 len 为 0 则表示端点尚未绑定；对于对方地址，则表示端点尚未连接。

> TLI 有一个未写入正式文档的函数 t_getname，它也返回本地协议地址和对方协议地址。
>
> t_getprotaddr 相当于 getsockname 和 getpeername 的组和。
>
> 如果我们对两个地址之一不感兴趣，我们仍需为那个地址分配一个 t_bind 的结构，只是 maxlen 设为 0。更简单的设计应允许我们直接用空指针参数来表示对它不感兴趣。

29.7　xti_ntop 函数

为了简化 XTI 协议地址的输出（比 netdir_getbyaddr 简单），我们编写一个自己的函数 xti_ntop，它与 3.8 节的 sock_ntop 函数类似。大多数 XTI 实现提供名为 taddr2uaddr 和 uaddr2taddr 的两个函数。taddr 表示传输地址（transport address），存储在 netbuf 结构中，uaddr 表示通用地址（universal address），是一个可读的文本字符串，存在以空字符结尾的 C 字符串里。IPv4 通用地址是一个用 5 个点号分隔开的 6 个十进制数组成的串。前 4 个数是点分十进制数 IPv4 地址，后两个数组成 2 字节的 TCP 或 UDP 端口号。

这两个函数的问题在于它们需要一个 netconfig 结构作参数，由该结构给出其地址正待转换的协议的配置信息。而 XTI 的 IPv4 及 IPv6 协议地址是自定义的。例如，在 netbuf 结构中存储 IPv4 地址实质上用的是套接口地址结构，它的第一个成员是 IPv4 的地址族 AF_INET。这种 netbuf 结构的长度为 16 字节（例如图 28.1 和图 28.2 中的 addr 行）。如果是 IPv6 地址，我们预期使用的是 sockaddr_in6 结构，地址族为 AF_INET6，长度为 24 字节。我们不能肯定，所有存储在 netbuf 中的 XTI 地址都是如此自定义的，但 IPv4 和 IPv6 的确是这样的。

> 自定义的说法也许措词过强。另外某个协议族也使用 16 字节的地址结构，而

且其前2个字节碰巧等于常值AF_INET的情况是可能的。不过现实地说,这不应该是个问题。

接下去,必须选择如何向函数传递协议地址。头一个选择是使用形如t_XXX的XTI结构。不过对于客户而言,服务器的协议地址是放在t_call结构的addr成员里的(28.6节);对于服务器而言,客户的协议地址是存在t_bind结构的addr成员里的(30.2节);对于任意端点而言,由t_getprotaddr返回的本地和对方地址都是存在t_bind结构里的。由于这里没有一致性(如果我们传递这三个结构中某一个的指针,我们还得同时传递一个标志位指明其结构类型),因此我们将跳过XTI结构,直接给我们的函数传递一个netbuf结构的指针。

```
#include "unpxti.h"
char *xti_ntop(const struct netbuf *np);
                                返回:非空指针——成功,NULL——出错
```

参数为指向含有地址的netbuf结构的指针,结果存储在函数内的静态存储区中。如果成功,返回值就是指向含有所给地址表达格式字符串的指针。

另一个函数xti_ntop_host使用同样的调用序列,不过只格式化IP地址,忽略端口号。

这两个函数与图3.13的函数类似。这里我们就不给出源码了,不过它们是可以免费获得的(见前言)。

29.8 tcp_connect函数

现在我们将getnetpath函数(返回一个或多个协议的信息)和netdir_getbyname函数(查找主机名和服务名)相结合,使用XTI API重写11.8节的tcp_connect函数,代替使用套接口API和getaddrinfo。图29.6给出了这个函数。

```
1  #include    "unpxti.h"
2  int
3  tcp_connect(const char *host, const char *serv)
4  {
5      int     tfd, i;
6      void    *handle;
7      struct t_call tcall;
8      struct t_discon tdiscon;
9      struct netconfig *ncp;
10     struct nd_hostserv hs;
11     struct nd_addrlist *alp;
12     struct netbuf *np;
13     handle = Setnetpath();
14     hs.h_host = (char *) host;
15     hs.h_serv = (char *) serv;
16     while ( (ncp = getnetpath(handle)) != NULL) {
17         if (strcmp(ncp->nc_proto, "tcp") != 0)
18             continue;
```

```
19          if (netdir_getbyname(ncp, &hs, &alp) != 0)
20              continue;
21              /* try each server address */
22          for (i = 0, np = alp->n_addrs; i < alp->n_cnt; i++, np++) {
23              tfd = T_open(ncp->nc_device, O_RDWR, NULL);
24              T_bind(tfd, NULL, NULL);
25              tcall.addr.len  = np->len;
26              tcall.addr.buf  = np->buf;      /* pointer copy */
27              tcall.opt.len   = 0;            /* no options */
28              tcall.udata.len = 0;            /* no user data with connect */
29              if (t_connect(tfd, &tcall, NULL) == 0) {
30                  endnetpath(handle);         /* success, connected to server */
31                  netdir_free(alp, ND_ADDRLIST);
32                  return(tfd);
33              }
34              if (t_errno == TLOOK && t_look(tfd) == T_DISCONNECT) {
35                  t_rcvdis(tfd, &tdiscon);
36                  errno = tdiscon.reason;
37              }
38              t_close(tfd);
39          }
40          netdir_free(alp, ND_ADDRLIST);
41      }
42      endnetpath(handle);
43      return(-1);
44  }
```

图 29.6　使用 XTI 的 tcp_connect 函数[libxti/tcp_connect.c]

初始化

第 13～15 行　使用 setnetpath 打开 netconfig 文件，用 host(主机名)和 serv(服务名)两参数初始化 nd_hostserv 结构。

获取 netconfig 文件的下一项

第 16～18 行　getnetpath 获取 NETPATH 变量中下一个协议在 netconfig 文件内的相应项。如果协议不是 TCP，则忽略之。其实，既然我们只寻找 TCP 的 netconfig 项，我们可以改用如下调用来定位该项：

```
ncp = getnetconfigent ("tcp")
```

若要释放 getnetconfigent 分配的空间，则可调用

```
freenetconfigent(ncp);
```

不过，为了做到与 IPv6 的兼容，我们还是逐个搜索 netconfig 项。目前我们还不知道在 netconfig 文件内如何存放 IPv6 上的 TCP 协议项，也不知道 XTI 名字函数如何处理 IPv6。

搜索主机名和服务名

第 19～20 行　使用 getnetpath 返回的 netconfig 结构，调用 netdir_getbyname 查找主机名和服务名。

尝试所有服务器地址

第21～28行 本循环尝试前面返回的每个服务器地址,逐个调用t_open、t_bind和t_connect,直到建立连接,或者所有地址都试过为止。使用netdir_getbyname返回的netbuf结构初始化t_call结构。

连接成功

第29～33行 如果连接建立成功,则进行清理工作并返回已连接描述字。endnetpath释放为netconfig结构分配的空间,并关闭netconfig文件。netdir_free释放从nd_addrlist结构开始的所有动态分配的空间(图29.2)。

处理t_connect错误

第34～38行 如果t_connect失败,则检查TLOOK。如果连接被拒绝则调用t_rcvdis来处理。我们用一个协议相关的错误码置errno,以供调用者查询。相应端点随后关闭。

试完所有地址

第40～41行 如果尝试了所有地址,则使用netdir_free来释放nd_addrlist结构及由它指向的netbuf结构数组。while循环将继续搜索netconfig文件以求获得待尝试的其他协议。

> getaddrinfo相当于把getnetpath调用、测试正确协议或语义及netdir_getbyname调用结合起来。
>
> XTI端点在连接建立失败后,仍然可用于另一次t_connect调用。也就是说,我们可以把t_open和t_bind调用统统移到for循环外面,只在外面的while循环中每循环一次调用它们一次。自然t_close循环也可移出for循环。而对于套接口,一旦连接失败,套接口也就没有用了,必须关闭掉(例如图11.6)。
>
> 不过这么做也带来一个问题。假如我们试图连接的主机有多个地址,而连接建立又失败了,那么要是我们使用老端点的话,本地端口并不改变,而且由于所有连接建立都出自同一本地端点,因此每次t_connect相对于上一次t_connect就将有一个指数回退延时。也就是说,如果第一次t_connect失败,第二次t_connect可能要延时1秒;如果再次失败,第三次t_connect可能要延时2秒……。为了避免这种情况的发生,我们还是在t_connect失败时t_close当前端点,然后为下一次t_connect调用创建一个新端点。

例子

现在,我们改用XTI API及新版tcp_connect函数改写图11.7中使用套接口API的协议无关时间/日期客户程序。图29.7给出了XTI版本的这个函数。

```
1 #include    "unpxti.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int       tfd, n, flags;
6     char      recvline[MAXLINE + 1];
7     struct t_bind * bound, * peer;
```

```
 8    struct t_discon tdiscon;
 9    if (argc != 3)
10        err_quit("usage: daytimecli02 <hostname/IPaddress> <service/port#>");
11    tfd = Tcp_connect(argv[1], argv[2]);
12    bound = T_alloc(tfd, T_BIND, T_ALL);
13    peer = T_alloc(tfd, T_BIND, T_ALL);
14    T_getprotaddr(tfd, bound, peer);
15    printf("connected to %s\n", Xti_ntop(&peer->addr));
16    for ( ; ; ) {
17        if ( (n = t_rcv(tfd, recvline, MAXLINE, &flags)) < 0) {
18            if (t_errno == TLOOK) {
19                if ( (n = T_look(tfd)) == T_ORDREL) {
20                    T_rcvrel(tfd);
21                    break;
22                } else if (n == T_DISCONNECT) {
23                    T_rcvdis(tfd, &tdiscon);
24                    errno = tdiscon.reason; /* probably ECONNRESET */
25                    err_sys("server terminated prematurely");
26                } else
27                    err_quit("unexpected event after t_rcv: %d", n);
28            } else
29                err_xti("t_rcv error");
30        }
31        recvline[n] = 0;   /* null terminate */
32        fputs(recvline, stdout);
33    }
34    exit(0);
35 }
```

图 29.7　协议无关的时间/日期客户程序[xtiintro/daytimecli02.c]

建立连接

第 11 行　使用图 29.6 的 tcp_connect 函数查找主机名和服务名,并建立连接。

输出对方协议地址

第 12～15 行　分配两个 t_bind 结构,使用 t_getprotaddr 得到本地协议地址和对方协议地址。调用 xti_ntop 函数输出对方地址。

从服务器读取数据

第 16～33 行　从服务器读取数据的部分与 28.11 节中代码相同。

运行程序结果如下:

```
unixware % daytimecli02 aix daytime
connected to 206.62.226.43.13
Fri Feb 7 13:28:24 1997
```

29.9 小 结

在 XTI 的 SVR4 实现中,网络的选择通常是使用/etc/netconfig 文件完成的,函数 netdir_getbyname 随后查找主机名和服务名,返回一个 netbuf 结构的数组,每个地址和服务一个结构。这与第 11 章的 getaddrinfo 函数类似。逆映射过程用 netdir_getbyaddr 函数完成,与 getmameinfo 函数类似。

由于 XTI 使用数量不少的 7 种 t_XXX 结构,且有数量不等的 netbuf 结构包含其中,因此提供了两个函数:t_alloc 和 t_free 以动态分配和释放这些结构。

29.10 习 题

29.1 getnetconfig 返回一个填充了的结构的指针,这与 gethostbyname 相似。但我们说过,gethostbyname 不是线程安全的。getnetconfig 是线程安全的吗?如果是,它是如何做到这一点的?

29.2 试写一个调用两次 t_alloc 来为某个端点分配 t_call 结构的程序。头一次以 T_ALL 为第三参数,第二次用 T_ADDR|T_OPT|T_UDATA。会发生什么事?

29.3 为什么 t_free 需要 structtype 参数?

29.4 在图 29.6 中,我们为什么不用以下方式来初始化 nd_hostserv 结构:

```
struct nd_hostserv hs = { host, serv };
```

第 30 章　XTI:TCP 服务器程序

30.1　概　述

毫无疑问,XTI 最易混淆的方面是其面向连接服务器对外来连接的处理。使用套接口 API 时,我们只需调用 accept,所有细节都交给内核或套接口函数库去处理。对于 TCP 而言,客户的 SYN 到达后先被放入相应套接口(相当于 XTI 的端点)的未完成连接队列之中(图 4.6)。当三路握手完成之后,accept 返回(图 2.5)。如果已完成连接队列中有多个连接,则以 FIFO 次序由 accept 返回。在现实中(图 4.9),未完成连接数通常大于 0,而已完成连接数通常为 0。

XTI 模型的"意图"(基于 OSI 传输服务的设计)是允许传输层在 SYN 到达时通知服务器进程(称为连接指示),将客户协议地址(IP 地址和端口号)传给服务器。服务器随后既可以接受连接请求,也可以拒绝它。在这种模型下,服务器所在 TCP 模块在服务器进程告诉它怎么做之前并不发送 SYN/ACK 或 RST。这种模型如图 30.1 所示。

注意一下服务器的函数调用:首先以非 0 的 qlen 来调用 t_bind,指示该端点准备接受外来连接。接着调用 t_listen,它在有连接到来后返回(我们一会就讨论)。最后调用 t_open、t_bind 和 t_accept 来接受连接。LISTEN 指示端点当前的 TCP 状态(图 2.4)。

当服务器进程获得连接请求指示后,它也可以选择不接受连接,这时调用 t_snddis 来拒绝请求,见图 30.2。这儿的"意图"在于服务器在客户 SYN 到达(即连接指示)时被通知到,但它选择不接受连接(也许基于客户的 IP 地址或端口号;要是协议支持的话,也许基于随连接请求发送的用户数据)。应用进程随后调用 t_snddis,导致发送 RST 而不是完成三路握手。这使得客户的 t_connect 调用返回一个错误。

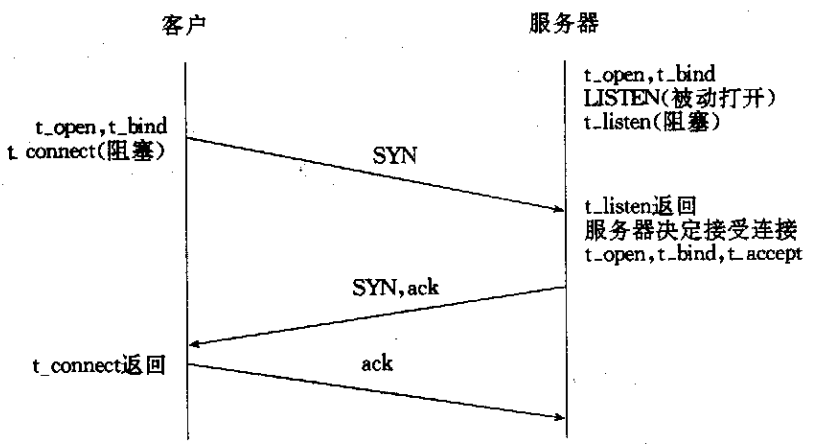


图 30.1　XTI 服务器接收连接请求的预期模型

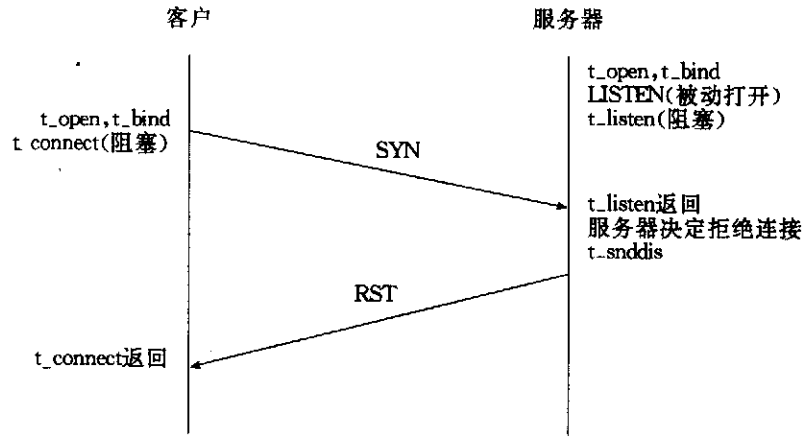


图30.2 XTI服务器拒绝连接请求的预期模型

回忆一下我们就套接口的讨论和图2.5的时间线图,我们知道套接口服务器绝不会导致客户connect的失败,因为accept总是在三路握手完成时即connect返回后再过半个RTT返回。如果套接口服务器不喜欢某个客户(也许是基于由accept返回的该客户的IP地址和端口号),该服务器所能做的就是终止(无法拒绝的)连接,所用方法或者是正常的close(这时发送的是FIN),或者是在设置SO_LINGER套接口选项后再调用close(这时发送的是RST)。

上面叙述XTI情形时我们使用带引号的"意图"字眼是因为这不是现实的情形。这种情形是OSI协议的意图,但大多数现有的TCP实现都自动接受外来的连接请求(只要已完成和未完成连接队列还没有满就行),然后到三路握手完成时才通知服务器进程。

当SYN到达时通知应用进程,然后在应用进程指示它是否接受或拒绝连接请求后再决定是否继续完成三路握手的技术有时称为惰性接受(lazy accept)。历史上至少有两个TLI的实现(分别出自Wollongone Group和Sequent Computer Systems)执行惰性接受。两者都对三路握手完成时从t_listen返回的既成事实"标准"方法作了改动。这种改动的原因之一是惰性接受导致大多数FTP实现不工作。

Posix.1g也要求TCP端点的t_listen成功返回本身表示的是一个已完成的连接,而不是一个连接指示。

[Jacobson 1994]指出4.4BSD本来打算提供一个针对每个套接口的选项以允许在TCP的套接口API中使用惰性接受,但这一点从未实现过。不过4.4BSD支持OSI协议的惰性接受。

30.2 t_listen函数

对于面向连接的XTI服务器而言,以下是常见的函数调用情形:

```
listenfd = t_open( ... );              /* create listening endpoint */
t_bind(listenfd, ... );                /* t_bind.qlen > 0 */

for ( ; ; ) {
```

```
    t_listen(listenfd, ... );              /* blocks awaiting connection */
    connfd = t_open( ... );                /* create new fd for connected endpoint */
    t_bind(connfd, NULL, NULL);            /* any local addr */
    t_accept(listenfd,connfd, ...);        /* accept on new fd */
    ...          /* process connected endpoint */
    t_close(connfd);
}
```

t_listen 是其中通常阻塞的函数,它等待来自客户的连接。

```
#include <xti.h>
int t_listen(int fd,struct t_call *call);
                                                    返回:0——成功,-1——出错
```

我们在说明 t_connect 函数时已经给过 t_call 结构,这里再给一遍:

```
struct t_call {
    struct netbuf  addr;      /* protocol-specific address */
    struct netbuf  opt;       /* protocol-specific options */
    struct netbuf  udata;     /* user data to accompany connection request */
    int            sequence;  /* for t_listen() & t_accept() functions */
};
```

通过 call 指针返回的结构含有本连接的相关参数:addr 含有客户的协议地址,opt 含有特定于协议的选项,udata 含有伴随连接请求一起发送过来的用户数据(TCP 不支持)。sequence 含有标识此连接请求的唯一值,该值由 t_accept(或 t_snddis)使用,用来鉴别接受(或拒绝)的连接请求。

> 虽然该函数看起来类似于 accept,但两者是不同的,t_listen 仅等待连接的到来,而不接受连接。要接受连接,需调用 XTI 的 t_accept 函数。
>
> 虽然 sequence 是一个整型值,但有些实现存储的是一个地址。不可将其假定为一个类似于描述字的小整数。

30.3　tcp_listen 函数

下面,我们给出一个函数,它创建一个监听端点,用来接受外来的连接。该函数的调用方式与图 11.8 的同名函数一样。

```
1 #include    "unpxti.h"
2 #include    <limits.h>        /* PATH_MAX */
3 char  xti_serv_dev[PATH_MAX + 1];
4 int
5 tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
6{
7     int       listenfd;
8     void      *handle;
9     char      *ptr;
```

```
10  struct t_bind tbind;
11  struct t_info tinfo;
12  struct netconfig *ncp;
13  struct nd_hostserv hs;
14  struct nd_addrlist *alp;
15  struct netbuf *np;
16  handle = Setnetconfig();
17  hs.h_host = (host == NULL) ? HOST_SELF : (char *) host;
18  hs.h_serv = (char *) serv;
19  while ( (ncp = getnetconfig(handle)) != NULL &&
20          strcmp(ncp->nc_proto, "tcp") != 0);
21  if (ncp == NULL)
22      return(-1);
23  if (netdir_getbyname(ncp, &hs, &alp) != 0) {
24      endnetconfig(handle);
25      return(-2);
26  }
27  np = alp->n_addrs;          /* use first address */
28  listenfd = T_open(ncp->nc_device, O_RDWR, &tinfo);
29  strncpy(xti_serv_dev, ncp->nc_device, sizeof(xti_serv_dev));
30  tbind.addr = *np;       /* copy entire netbuf{} */
31      /* 4can override LISTENQ constant with environment variable */
32  if ( (ptr = getenv("LISTENQ")) != NULL)
33      tbind.qlen = atoi(ptr);
34  else
35      tbind.qlen = LISTENQ;
36  T_bind(listenfd, &tbind, NULL);
37  netdir_free(alp, ND_ADDRLIST);
38  endnetconfig(handle);
39  if (addrlenp)
40      *addrlenp = tinfo.addr; /* size of protocol addresses */
41  return(listenfd);
42 }
```

图30.3 XTI tcp_listen函数:创建监听端点[libxti/tcp_listen.c]

初始化

第16～18行 setnetconfig打开/etc/netconfig文件。如果host参数为空指针,则使用特殊字符串HOST_SELF作为netdir_getbyname的参数。这使得监听端点被捆绑以通配地址(IPv4的通配地址为0.0.0.0)。

查找匹配的协议

第19～22行 逐行处理/etc/netconfig文件,查找TCP协议。注意,我们给服务器程序使用getnetconfig函数,这与图29.6给客户程序使用getnetpath有所不同。这主要是由于服务器程序和客户程序的区别。服务器往往是由初始化脚本或从命令行启动,所以我们无法保证NETPATH环境变量设置了有意义的值。相反,客户通常是从代表一个用户的交互式shell启动的,因此可以假设NETPATH变量已由该用户设置。

寻找主机名和服务名

第 23～27 行　在获得 netconfig 结构内的协议项之后，调用 netdir_getbyname 寻找主机名和服务名。

打开设备

第 28～29 行　t_open 打开相应的设备(譬如:/dev/tcp)，然后我们将设备名存入一个外部变量 xti_serv_dev。这么做是由于 tcp_listen 的调用者需要对每一次连接再次调用 t_open，这时得使用该设备名来保证协议无关性。对于套接口版本的 tcp_listen 就不用这么费事了，因为 accept(而不是由进程)自动为每个连接创建一个新的套接口。

> 本技术是非线程安全的。这是要求应用进程在调用监听描述字的 t_open 和以后调用每个已连接描述字的 t_open 之间维护状态信息(设备名)的负效应。使得本操作线程安全的方法之一是调用 strdup 把设备名拷入动态分配存储区，然后就通过 tcp_listen 的另一个参数返回其指针，调用者可凭此 free 之。

进入 TCP 的 LISTEN 状态

第 30～36 行　调用 t_bind，将 netdir_getbyname 返回的地址捆绑到端点。将 t_bind 结构的 qlen 成员设为非 0，这将指示本端点为监听端点，于是在 TCP 情况下该访问点进入 LISTEN 状态(XTI 称之为 T_IDLE 状态)。外来连接请求现在即可被传输提供者接受。这里我们使用 LISTENQ 环境变量覆盖 unp.h 头文件中所给的缺省值。我们在套接口 API 的 listen 函数的包裹函数 Listen(图 4.8)中有类似的代码。

释放内存空间，返回值

第 37～41 行　通过调用 netdir_free 和 endnetconfig 释放分配的内存空间，然后返回协议地址的大小(如果需要)，函数本身的返回值则为监听端点的描述字。

注意我们这里没有调用 t_listen 函数，因为那是服务器阻塞在其上等待外来连接的函数。

30.4　t_accept 函数

一旦 t_listen 函数指示有连接到达，我们就可以决定是否接受该请求。准备接受则调用 t_accept 函数：

```
#include <xti.h>
int t_accept(int listenfd, int connfd, struct t_call *call);
                                        返回:0——成功,-1——出错
```

listenfd 指定连接到达的端点，也就是说这是曾作为 t_listen 的参数的端点。connfd 指定将建立连接的端点。通常，服务器总是创建一个新的端点 connfd 来接收连接。

call 参数确定到底接受哪个连接(因为一个端点上可能有多个连接等待接受，我们很快就会谈到)，其值是此前 t_listen 的返回值。

注意现在是由服务器本身为自己创建新的端点。这一般是通过在 t_listen 和 t_accept 之间调用 t_open 完成的。

> 我们可以使用相同的 listenfd 和 connfd,也就是直接在监听端点上接受连接。不过如果我们真这么做,那么直到该连接完成前,提供者都无法接受下一个连接(也就是说这是迭代服务器)。这种情形下 qlen 只宜设为 1。由于大多数现实的服务器都需要同时处理多个连接,这种迭代情形的服务器程序我们就不再举例了。

30.5 xti_accept 函数

现在我们编写一个简单的名为 xti_accept 的函数来完成使用 XTI 接受一个连接的步骤。一般情况下,我们的 XTI 服务器应用程序的代码应为如下形式:

```
listenfd = TCP_listen( ... ); /* create listening endpoint */
for ( ; ; ) {
    connfd = xti_accept(listenfd, ...); /* block ,then accept */
    ...     /* process connfd */
    t_close (connfd);
}
```

这段代码与套接口 API 代码类似,差别仅仅是以 xti_accept 取代 accept。

```
#include "unpxti.h"
int xti_accept(int listenfd, struct netbuf *cliaddr, int rdwr);
                                        返回:非负描述字——成功,-1——出错
```

如果成功返回,则返回值为新的已连接描述字,客户地址则在 cliaddr 所指向的 netbuf 结构内返回。如果 rdwr 参数不为 0,则对已连接端点调用 xti_rdwr 函数。

图 30.4 给出了 xti_accept 的简单版本。我们说"简单"是指该函数在同时遇到多个连接时会失败。在30.8节我们将修复这个缺陷。

```
1  #include    "unpxti.h"
2  int
3  xti_accept(int listenfd, struct netbuf *cliaddr, int rdwr)
4  {
5      int         connfd;
6      u_int       n;
7      struct t_call *tcallp;
8      tcallp = T_alloc(listenfd, T_CALL, T_ALL);
9      T_listen(listenfd, tcallp);          /* blocks */
10         /* following assumes caller called tcp_listen() */
11     connfd = T_open(xti_serv_dev, O_RDWR, NULL);
12     T_bind(connfd, NULL, NULL);
13     T_accept(listenfd, connfd, tcallp);
```

```
14    if (rdwr)
15        Xti_rdwr(connfd);
16    if (cliaddr) {          /* return client's protocol address */
17        n = min(cliaddr->maxlen, tcallp->addr.len);
18        memcpy(cliaddr->buf, tcallp->addr.buf, n);
19        cliaddr->len = n;
20    }
21    T_free(tcallp, T_CALL);
22    return(connfd);
23 }
```

图 30.4　简单版本的 xti_accept 函数[libxti/xti_accept_simple.c]

等待连接

第 8～9 行　分配一个 t_call 结构，用来存放客户连接的信息，调用 t_listen 等待客户连接。

创建新的端点并捆绑任意本地地址

第 10～12 行　使用 tcp_listen 存储在外部变量 xti_serv_dev 中的路径名调用 t_open 创建新的端点，并调用 t_bind 给它捆绑任意本地地址。这个 t_bind 调用是可选的。如果未调用，那么在调用 t_accept 时该访问点就处于未绑定状态，通信提供者将自动给它捆绑某个合适的地址。

接受连接

第 13 行　t_accept 根据 t_call 结构中的 sequence 成员接受特定连接，该结构是由 t_listen 填写以标识这个特定连接的。

如果需要则允许 read 和 write

第 14～15 行　如果调用者将 rdwr 参数设为非 0 的话，我们就调用 xti_rdwr 函数，将 tirdwr 流模块压入流中，以允许调用者使用 read 和 write 取代 t_rcv 和 t_snd。

返回客户协议地址

第 16～20 行　调用者可以指定一个指向某个 netbuf 结构的非空指针参数 cliaddr。该结构必须由调用者初始化，再由本函数在返回时填入客户的协议地址。我们保证不溢出调用者的缓冲区，然后将其 len 成员设置成将返回的地址的大小。

清除并返回

第 21～22 行　释放 t_call 结构，并返回已连接描述字。

30.6　简单的时间/日期服务器程序

现在，我们使用 XTI 重写图 11.10 的时间/日期服务器程序，其中使用了 tcp_listen 和 xti_accept 函数。

```
1 #include    "unpxti.h"
2 int
3 main(int argc, char **argv)
```

```
4 {
5     int             listenfd, connfd;
6     char            buff[MAXLINE];
7     time_t          ticks;
8     socklen_t addrlen;
9     struct NETBUF CLIADDR;
10    if (argc == 2)
11        listenfd = Tcp_listen(NULL, argv[1], NULL);
12    else if (argc == 3)
13        listenfd = Tcp_listen(argv[1], argv[2], NULL);
14    else
15        err_quit("usage: daytimetcpsrv01 [ <host> ] <service or port>");
16    cliaddr.buf = Malloc(addrlen);
17    cliaddr.maxlen = addrlen;
18    for ( ; ; ) {
19        connfd = Xti_accept(listenfd,&cliaddr,0);
20        printf("connection from %s\n", Xti_ntop(&cliaddr));
21        ticks = time(NULL);
22        snprintf(buff,sizeof(buff),"%.24s\r\n",ctime(&ticks));
23        T_snd(connfd, buff, strlen(buff),0);
24        T_close(connfd);
25    }
26 }
```

图30.5 使用XTI的时间/日期服务器程序[xtiintro/daytimesrv01.c]

创建端点

第10～17行 使用tcp_listen创建监听端点。为客户协议地址分配空间,并为此初始化netbuf结构。

等待连接并接受之

第19～20行 使用xti_accept函数等待连接,创建一个新端点并返回已连接描述字、客户IP地址和端口号。使用xti_ntop输出客户协议地址。

产生时间/日期输出

第21～24行 调用time取得当前时间,再调用ctime将它转换成直观可读格式。调用t_snd将它发回给客户后,调用t_close关闭端点。

注意:在发送完数据后,我们简单地调用t_close。由于TCP提供顺序释放,因此这将导致发送一个FIN,并经历通常的四分组连接终止序列(图2.5),不过t_close立即返回。

这跟对TCP套接口调用close等价。

如果希望等待对方TCP收到我们的数据后再发送FIN,我们必须调用t_sndrel来发送我们的FIN,然后调用t_rcvrel等待对方的FIN。也就是用以下部分代替图30.5结束处的T_close。

```
T_sndrel(connfd);
while ( (n = t_rcv(connfd, buff, MAXLINE, &flags)) >= 0)
    ;
```

```
if (t_errno == TLOOK) {
    if ( (n = T_look(connfd)) == T_ORDREL) {
        T_rcvrel(connfd);
    } else if (n == T_DISCONNECT) {
        T_rcvdis (connfd, NULL);
    } else
        err_quit ("unexpected event after t_rcv: %d", n);
} else
    err_xti("t_rcv error");
T_close (connfd);
```

t_sndrel 发送 FIN。此后我们应等待一个顺序释放指示，等到之后调用 t_rcvrel 接收它。为了做到这一点，我们调用 t_rcv，并忽略所有可能到达的数据。

这种情形跟对套接口调用 shutdown，然后等待直到 read 返回文件结束符(图 7.8)类似。

对于 XTI，如果仍有数据排队等待发送到对方，我们也可以促成 t_close 或 close 的延迟，而不是立即返回。这是通过设置 XTI_LINGER 选项完成的，我们将在 32.3 节讨论。这与 SO_LINGER 套接口选项类似。

30.7　多个待处理连接

如果差不多同时有多个连接到达监听端点，事情就有点复杂了。为了说明这个问题，我们先回过头来看一下图 27.5 的 TCP 服务器程序。我们用该程序测量多种类型服务器所需的进程控制时间。我们可以运行为该服务器程序编写的客户程序(图 27.4)并指定 fork 的子进程数，从而跟服务器建立多个连接。

图 30.6 给出的服务器程序与图 27.5 类似，差别仅在改用 XTI 代替套接口 API。

```
1  #include    "unpxti.h"
2  int
3  main(int argc, char **argv)
4  {
5      int         listenfd, connfd;
6      pid_t       childpid;
7      void        sig_chld(int), sig_int(int), web_child(int);
8      socklen_t addrlen;
9      struct netbuf cliaddr;
10     if (argc == 2)
11         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
12     else if (argc == 3)
13         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
14     else
15         err_quit("usage: serv01 [ <host> ] <port#>");
16     cliaddr.buf = Malloc(addrlen);
17     cliaddr.maxlen = addrlen;
18     Signal(SIGCHLD, sig_chld);
19     Signal(SIGINT, sig_int);
```

```
20      for ( ; ; ) {
21          connfd = Xti_accept(listenfd, &cliaddr, 1);
22          printf("connection from %s\n", Xti_ntop(&cliaddr));
23          if ( (childpid = Fork()) == 0) {      /* child process */
24              Close(listenfd);   /* close listening socket */
25              web_child(connfd);      /* process the request */
26              exit(0);
27          }
28          Close(connfd);              /* parent closes connected socket */
29      }
30  }
31  void
32  sig_int(int signo)
33  {
34      void   xti_accept_dump(void);
35      xti_accept_dump();
36      exit(0);
37  }
```

图30.6 说明多连接问题的TCP并发服务器程序[xtiserver/serv01.c]

第10～22行 调用本章早些时候开发的tcp_listen和xti_accept函数。

SIGINT信号处理程序

第31～37行 该信号处理程序调用xti_accept_dump函数,我们以此收集图30.13所示的各个计数器。该函数输出每个cli结构(图30.7)的count成员。

如果我们启动该服务器在TCP端口9999上监听:

```
unixware % serv01 9999
```

并在另一主机上运行客户程序:

```
solaris % client unixware 9999 1 600 4000
```

那么一切正常。(1为fork的子进程数,600为每个子进程建立连接的次数,4000是每次连接请求的字节数。)这是由于我们让客户只派生一个子进程,因此各个连接串行到达服务器。

如果我们将第三个命令行参数由1改为2,则服务器会立即给出如下错误信息:

```
t_accept error: event requires attention
```

问题出在有两个连接请求几乎同时到达服务器(每个子进程一个连接),在服务器的TCP建立连接之前,这两个连接请求同时引发三路握手过程。

当服务器的TCP正在建立连接时,我们的服务器进程阻塞在xti_accept发起的t_listen调用上。当第一个连接完成三路握手后,t_listen返回,接着调用t_open和t_accept。但是当给第一个连接调用t_accept时,第二个连接已完成三路握手,也准备好被接受。按照XTI的规则,此时t_accept不是完成第一个连接,而是返回一个错误,并置t_errno为TLOOK。待处理的事件为T_LISTEN(有一个连接指示待处理),因为现在有另外一个连接(即第二个连接)待处理。

这儿的关键在于,如果有另外一个连接准备好,t_accept 总是返回一个错误。这时我们不得不做的是调用 t_listen 接收所有的连接指示,保存每个连接的 t_call 结构,然后给每个连接调用 t_accept。

> 为什么 XTI 要以这种奇怪的方式来接受连接呢?要是 t_listen 确实在客户的 SYN 到达时返回(图 30.1),那么在给任何单个连接调用 t_accept 前,强制服务器对所有已到达的 SYN 调用 t_listen 给了服务器进程以选择待处理连接接受顺序的机会。服务器也许会对这些连接按排优先级,例如基于由 t_listen 返回的 IP 地址或端口号,或者基于伴随连接请求的用户数据(TCP 不支持)。然而在 TCP 的 t_listen 到三路握手完成后才返回的情况下,这种特性纯粹是给服务器增加(不必要的)复杂性。
>
> 我们刚才叙述的对应于 Unix 95 中的 XTI。Posix.1g 和 Unix 98 修改了 t_accept 的说明,说它也许失败并置 t_errno 为 TLOOK。不过我们必须准备应付 t_accept 确实失败的情况。

30.8　xti_accept 函数(修订版)

现在,我们对图 30.4 的简单版本 xti_accept 进行修改,使之更健壮。为此要处理以下两个问题:

- t_accept 在另有一个连接待处理时将出错(t_look 返回 T_LISTEN)。
- t_accept 在一个待处理连接收到 RST 时将出错(t_look 返回 T_DISCONNECT)。

为了解决以上问题,我们将维护一个待处理连接的队列,其长度不超过调用监听端点的 t_bind 时的 qlim 值。对待处理的连接进行跟踪时,我们可使用很多种数据结构;简单起见,我们只用一个 cli 结构的栈(数组)。每个结构内包含三个信息:一个已连接描述字、一个本结构使用频度的诊断用计数器以及一个指向 t_call 结构的指针。

假设同时有三个客户与服务器建立连接,并完成三路握手过程。当调用 t_listen 返回第一个连接后,我们使用数组内的 cli[0]结构,如图 30.7 所示。

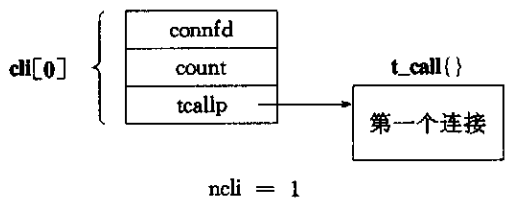


图 30.7　t_listen 返回头一个客户连接后的数据结构

connfd 是调用 t_open 创建的新描述字,用来接受连接。count 是测试程序所用的诊断用计数器。tcallp 是一个指向 t_call 结构的指针,该结构由 t_listen 填写并将传给 t_accept。它含有客户的协议地址(addr 成员)和连接标识符(sequence 成员)。我们还使用一个计数器 ncli 来记录数组中 cli 结构的数目,现在是 1。

假设现在调用 t_accept 去接受第一个连接,但是 t_accept 返回 TLOOK 错误,而 t_look

则返回T_LISTEN。这说明我们必须再调用t_listen来获取下一个连接。我们就给cli数组增加另外一项,并将ncli增1,见图30.8。

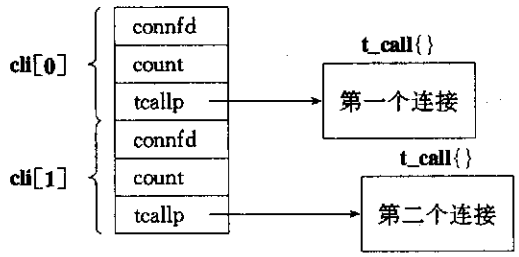


图30.8 t_listen返回第二个客户连接后的数据结构

我们总是对数组中最后一个cli结构(其下标为ncli-1)调用t_accept。这样,我们按后进先出(LIFO)顺序处理连接,而不是按人们也许期待的先进先出(FIFO)顺序。需要的话改成FIFO顺序也不困难,不过增加了复杂性。

> 图4.9表明,即使在适度繁忙的Web服务器上,同时有多个已完成连接等待应用进程接受的情况也不常见,因此这种简化设计是可行的。

现在我们给cli[1]调用t_accept,但仍然假设它返回TLOOK错误,t_look返回T_LISTEN。我们必须给cli数组再增加一项(即cli[2]),见图30.9。

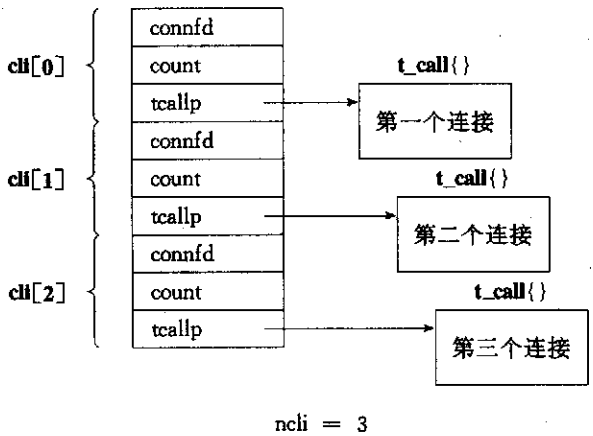


图30.9 t_listen返回第三个客户连接后的数据结构

现在我们有三个等待接受的连接,我们对cli[2]调用t_accept。但假设此时第一个客户(cli[0])通过发送一个RST而刚刚夭折了它的连接,那么t_accept仍以TLOOK错误失败,不过t_look返回的是T_DISCONNECT。对于这种情况,我们必须调用t_rcvdis来接收断连,回忆一下,该函数使用的t_discon结构中有一个sequence成员,它是用来标识已夭折的连接的。我们必须用该标识符搜索我们的cli结构数组,确定谁的t_call结构有匹配的sequence,然后将该项去掉。这里就是cli[1]代替cli[0],cli[2]代替cli[1],ncli减1。图30.10给出了现在的数据结构情况:

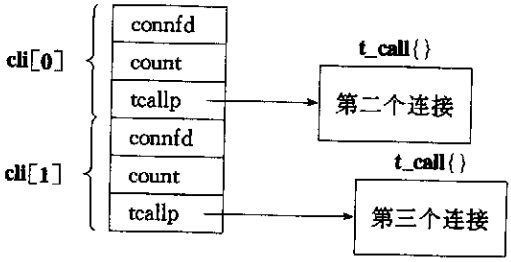


图 30.10 第一个连接夭折后的数据结构

接着,我们再次给 cli[1]调用 t_accept。这次假设调用成功,于是我们将 cli[1]从数组(栈)中移出,ncli 再减 1,并将所得到的第三个连接返回给 xti_accept 的调用者。记住,我们刚才所描述的一切,从第一个 t_listen 调用起,都发生在 xti_accept 函数的内部。现在是第一次由 t_listen 函数给调用者返回一个已连接描述字。

下次再调用 xti_accept 时 ncli 为 1,并对 cli[0]调用 t_accept。假设成功,返回的是第二个连接。

下面是新 xti_accept 函数的前半部分。

```
1 #include    "unpxti.h"
2 static int ncli = -1, ndisconn;
3 static struct cli {
4     int     connfd;      /* connected fd or -1 if disconnected */
5     int     count;
6     struct t_call    *tcallp;/* ptr to t_alloc'ed structure */
7 } *cli;                   /* cli[0], cli[1], ..., cli[ncli-1] are in use */
8 int
9 xti_accept(int listenfd, struct netbuf *cliaddr, int rdwr)
10 {
11     int          i, event;
12     u_int        n;
13     char         *ptr;
14     struct t_discon tdiscon;
15     if (ncli == -1) {          /* initialize first time through */
16         if (cli != NULL)
17             err_quit("already initialized");
18         if ( (ptr = getenv("LISTENQ")) != NULL)
19             n = atoi(ptr);
20         else
21             n = LISTENQ;
22         cli = Calloc(n, sizeof(struct cli));
23         for (i = 0; i < n; i++)
24             cli[i].tcallp = T_alloc(listenfd, T_CALL, T_ALL);
25         ncli = 0;
26     }
```

图 30.11 xti_accept 函数:前半部分[libxti/xti_accept.c]

声明静态变量

第2～7行 ncli计数器、夭折连接计数器ndisconn以及指向cli结构数组的指针都声明为static变量。

首次调用时初始化

第15～26行 在本函数第一次调用时我们动态分配一个cli结构数组，其元素数目根据LISTENQ常值或同名环境变量值来确定。由于tcp_listen中也用LISTENQ调用t_bind，这两者是一致的。对于数组中每一项，再分别调用t_alloc分配一个t_call结构，把返回的指针存放在cli结构中。

在线程环境下，cli数组和ncli计数器必须被保护起来，以保证各线程能同时调用xti_accept。

图30.12是函数的后半部分。

```
27    for ( ; ; ) {
28        if (ncli == 0) {   /* need to wait for a connection */
29            T_listen(listenfd, cli[ncli].tcallp);      /* block here */
30                /* following assumes caller called tcp_listen() */
31            cli[ncli].connfd = T_open(xti_serv_dev, O_RDWR, NULL);
32            T_bind(cli[ncli].connfd, NULL, NULL);
33            cli[ncli].count++;
34            ncli++;
35        }
36        if (t_accept(listenfd, cli[ncli-1].connfd,
37                     cli[ncli-1].tcallp) == 0) {
38            ncli--;                /* success */
39            if (rdwr)
40                Xti_rdwr(cli[ncli].connfd);
41            if (cliaddr) {          /* return client's protocol address */
42                n = min(cliaddr->maxlen, cli[ncli].tcallp->addr.len);
43                memcpy(cliaddr->buf, cli[ncli].tcallp->addr.buf, n);
44                cliaddr->len = n;
45            }
46            return(cli[ncli].connfd);
47        } else if (t_errno == TLOOK) {
48            if ( (event = T_look(listenfd)) == T_LISTEN) {
49                T_listen(listenfd, cli[ncli].tcallp);      /* won't block */
50                cli[ncli].connfd = T_open(xti_serv_dev, O_RDWR, NULL);
51                T_bind(cli[ncli].connfd, NULL, NULL);
52                cli[ncli].count++;
53                ncli++;
54            } else if (event == T_DISCONNECT) {
55                T_rcvdis(listenfd, &tdiscon);
56                for (i = 0; i < ncli; i++) {
57                    if (cli[i].tcallp->sequence == tdiscon.sequence) {
58                        T_close(cli[i].connfd);
59                        ndisconn++;
60                        ncli--;
```

```
61                          if ( (n = ncli - i) > 0)
62                              memmove(&cli[i], &cli[i+1],
63                                      n * sizeof(struct cli));
64                          break;
65                      }
66                  }
67              } else
68                  err_quit("unexpected t_look event %d", event);
69          } else
70              err_xti("unexpected t_accept error");
71      }
72  }
```

图 30.12　xti_accept 函数:后半部分[libxti/xti_accept.c]

等待连接

第 28～35 行　如果数组为空,则调用 t_listen 等待客户连接。这也是监听服务器花费时间较多的地方。当 t_listen 返回时,t_call 结构中将含有客户的协议地址以及连接标识符。在这之后,我们调用 t_open 创建一个新的端点以接受连接,并给它捆绑任意本地地址。

调用 t_accept;成功则返回

第 36～46 行　对 cli[ncli－1]对应的连接调用 t_accept。如果成功返回则向调用者返回已连接描述字。在返回之前还根据调用者的要求调用 xti_rdwr 以支持 read 和 write,需要的话还返回客户的协议地址。

处理额外的待处理连接

第 47～53 行　如果 t_accept 返回 TLOOK 且 t_look 返回 T_LISTEN,那么有另一个连接待处理,于是我们必须调用 t_listen 来接收有关这个新连接的信息。这个调用是不会阻塞的,因为本端点已经有一个待处理的 T_LISTEN 事件。在 t_listen 返回之后,我们同样调用 t_open 和 t_bind,并将相应信息存入 cli[ncli]中。

处理待处理连接的断连

第 54～66 行　如果 t_accept 返回 TLOOK,而 t_look 返回 T_DISCONNECT,那么某个已由 t_listen 返回的连接被其客户夭折。我们调用 t_rcvdis 获取这个已夭折连接的信息(也就是 sequence),并在数组内搜索匹配的项。当我们在数组内找到该项时,将其去除,并将其后的各项依次前移。这里我们使用 memmove 而不是 memcpy,这是由于前者能够正确地处理重叠区域(见习题 30.3)。

图 30.6 的服务器程序使用新版 xti_accept 之后,我们再用多个客户进行测试,结果正常。我们还想测一下 t_accept 由于额外的待处理连接而出错的频度。为此我们特意编写了一个函数来输出 cli 结构的 count 成员以及 ndisconn 计数器,并在服务器程序的 SIGINT 信号处理程序(图 30.6)内调用它。图 30.13 给出在 UnixWare 2.1.2 上运行服务器程序,在 Solaris 2.5.1 运行客户程序后得出的结果。我们让客户数在 1～4 内变化,且总是从所有子进程发出总共 600 个连接请求。

服务器计数器	客户子进程数			
	1	2	3	4
cli[0].count	600	309	95	102
cli[1].count		291	286	121
cli[2].count			219	235
cli[3].count				142
总计	600	600	600	600

图 30.13 t_accept 返回 T_LISTEN 的频度计数器

即使只是两个客户,每个客户几乎同时一个接一个地建立 300 个连接,一个接一个,几乎同时,t_accept 也有一半时间返回 T_LISTEN 事件的 TLOOK。

> 我们在讨论图 4.9 时提过,即使比较繁忙的 Web 服务器,这种多个已完成连接准备好等待服务器接受的情形也不常出现,而我们的测试例子里却出现得如此之频繁。这里有三个原因:(1)我们在同一 LAN 的主机上运行客户和服务器,以迫使出现图 30.13 的情形。(2)大约在 12 秒内就发起 600 个连接的速率相当于 1 天 4 百万个连接。(3)我们有意在一台慢速服务器主机(75-Mhz Pentium CPU)上运行本例子,目的是测试 xti_accept 函数对多个待处理连接的处理。结论是在现实世界里这种情形并不常见,因此按我们的 xti_accept 函数使用的 LIFO 处理顺序就足够了,不过这种情形的确会出现,所以还得对付它。

为了测试客户在连接建立后立即夭折连接的情况,我们对图 27.4 中的客户程序作如下修改:

```
  for (j = 0; j < nloops; j++) {
      fd = Tcp_connect(argv[1], argv[2]);
+     if (i == 2 && (j % 3) == 0) {
+         struct linger ling;
+
+         ling.l_onoff = 1;
+         ling.l_linger = 0;
+         Setsockopt(fd, SOL_SOCKET, SO_LINGER, &ling, sizeof (ling));
+         Close(fd);
+
+         /* and just continue on for this client connection ... */
+         fd = Tcp_connect(argv[1],argv[2]);
+     }
      Write (fd,request, strlen(request));
      if ( (n = Readn(fd,reply, nbytes)) != nbytes)
          err_quit("server returned %d bytes", n);
      Close(fd);      /* TIME_WAIT on client, not server */
  }
```

带加号的行是新加的。以上修改使得第三个子进程(i 等于 2)每三个刚完成的连接就放

弃一个。为了发送 RST,我们在相应设置 SO_LINGER 套接口选项后再 close 套接口。我们接着建立另外一个连接并继续循环。具体效果可能与时间有关:RST 可能在服务器调用 t_listen 和 t_accept 之间到达(我们用 ndisconn 计数器计数以验证相关代码的执行);可能在 t_listen 返回之前到达;也可能在连接被接受后到达。

XTI 队列长度与 listen 的 backlog

XTI 队列长度与 listen 函数的 backlog 类似,但并不完全相同。首先 listen 的 backlog 从来就没有什么明确的规范;我们从图 4.10 可以看出,不同的系统对它的解释是有差别的。

在 Posix.1g 中,XTI 队列长度指定提供者对于给定端点应支持的待处理连接指示数。待处理的连接指示指已由提供者传递给应用进程但尚未被接受或拒绝的连接指示。提供者可以给超出限制个数的连接指示排队,但必须保证任何时刻已递送给应用进程但仍未被处理的连接指示数不超过 qlen。

如果 XTI 具体实现在连接指示到达(即客户的 SYN 到达服务器)时将它们传递给应用进程,那么这种形式的应用进程排队手段也许是明智的。然而在应用进程得到通知前 TCP 连接就已经完全建立的情况下,让应用进程去给这些连接排队并没有实际的需要。

我们照常需要观察在不同的 qlen 情况下各种实现提供的实际效果。我们对图 E.15 做了些修改,使之在 XTI 而不是套接口 API 上工作,接着在 5 种支持 XTI 的系统上设置不同的 qlen 运行该程序。结果如图 30.14 所示。

请求 qlen	AIX 4.2		DUnix 4.0B		HP-UX 10.30		Solaris 2.6		UWare2.1.2	
	返回 qlen	实际的连接数	返回 qlen	实际的连接数	返回 qlen	实际的连接数	返回 qlen	实际的连接数	返回 qlen	实际的连接数
0	0	0	0	0	0	0	0	0		0
1	1	3	1	2	1	1	1	1		1
2	2	6	2	4	2	2	2	2		2
3	3	8	3	6	3	3	3	3		3
4	4	11	4	8	4	4	4	4		4
5	5	13	5	10	5	5	5	5		5
6	5	13	6	12	6	6	6	6		6
7	5	13	7	14	7	7	7	7		7
8	5	13	8	16	8	8	8	8		8
9	5	13	9	18	9	9	9	9		9
10	5	13	10	20	10	10	10	10		10
11	5	13	11	22	11	11	11	11		11
12	5	13	12	24	12	12	12	12		12
13	5	13	13	26	13	13	13	13		13
14	5	13	14	28	14	13	14	14		14

图 30.14　不同的 XTI qlen 值下实际排队的连接数

回忆一下图 28.5,t_bind 的第三个参数是一个指向 t_bind 结构的指针,该结构由提供者在返回时填写。看一下该结构的 qlen 成员,我们获悉提供者把它设置成什么值。(XTI 称之为协商值。)我们看到大多数系统返回指定的值,除非只支持较小的值(AIX)。不过有个系统(UnixWare)不返回该值。

任何系统在 qlen 为 0 时均不接收连接(这跟图 4.10 中 listen 的 backlog 为 0 不一样),

其中两个系统(HP-UX和Solaris)在调用t_connect时返回错误。一些系统(AIX和Digital Unix)支持超出qlen的连接队列,不过余下的三个不支持。

> 这里我们测量的是由提供者排队的连接数,而不是由应用进程排队的连接数,不过提供者执行的排队才是我们最关心的。

将服务器监听队列长度设为1

避免接受XTI连接所牵涉的复杂性的方法之一是将t_bind结构的qlen成员设成1。不过这也带来一个问题:许多系统将只排队一个客户连接(图30.14),在这个连接被应用进程接受之前,所有其他已到达的SYN都被忽略掉。

我们可以用本节的服务器程序测试这个特性。在UnixWare 2.1.2上运行服务器程序,这种系统在指定qlen为1时只排队一个连接。与图30.13一样,我们让客户子进程从1变化到4,不过这次我们测量600个连接的总流逝时间(单位为秒)。

队列长度	客户子进程数			
	1	2	3	4
1	10.6	12.2	15.6	13.2
1024	10.6	10.2	10.3	10.4

图30.15 总共600个连接的流逝时间,变动子进程数和队列长度

对于长度为1的队列,流逝时间随子进程数增长而增长。这是由于许多客SYN因已有一个连接填入队列而被服务器忽略,从而客户必须重传SYN的缘故。而当队列长度大于可能同时出现的连接数时,对于我们在这儿测试的小数目子进程来说,流逝时间反而减少。

这些数字表明一个问题:长度为1的队列对于现实中的服务器是不实际的。

30.9 小 结

使用XTI接受客户连接要比使用套接口麻烦得多。所增加的复杂性的根源在于XTI允许协议提供惰性接受,即在连接请求到来时就通知应用进程,而不是在连接建立完成以后再通知。TCP实现不提供惰性接受,Unix98也不再要求t_accept遇到另外一个待处理连接时返回T_LISTEN事件,但为了向后兼容,XTI服务器程序必须处理这种情况。

30.10 习 题

30.1 我们在30.2节里提到,有些系统在由t_listen填写的t_call结构里给sequence成员存放一个指针。在64位体系结构中会发生什么情况?

30.2 为什么在图30.3的xti_serv_dev声明中要对PATH_MAX加1?

30.3　在图 30.12 中我们调用 memmove,并提到这是由于源和目的区域有重迭而必须的。假设一个 4 字节数组,其元素从 x[0]到 x[3](从左到右画)。我们要从中删去 X[1],将其后 2 个元素依次左移 1 个字节,留下 3 个元素。试画一下源区域和目的区域,并说明从源区域头部开始拷贝(从右到左拷贝)和从源区域尾部开始拷贝(从左到右拷贝)各是什么样的。memcpy 保证哪个方向的拷贝结果正确?

30.4　使用 XTI API 重新编写图 E.15 的程序。

30.5　使用一个 cli 结构的链表代替 cli 结构数组重新编写图 30.11 和图 30.12 的程序。动态分配各 cli 结构。

第31章 XTI:UDP客户和服务器程序

31.1 概 述

XTI为无连接的客户和服务器程序提供3个函数:t_sndudata发送数据报;t_rcvudata接收数据报;t_rcvuderr获取异步错误信息。对于套接口API,我们有给UDP应用程序调用connect的选项,不过XTI不提供这种选择。

31.2 t_rcvudata和t_sndudata函数

这两个函数用来为无连接协议(例如UDP)提供收发数据报的能力。

```
#include <xti.h>
int t_rcvudata(int fd, struct t_unitdata *unitdata, int *flagsp);
int t_sndudata(int fd, struct t_unitdata *unitdata);
                                        二者均返回:0——成功,-1——出错
```

对于t_sndudata,t_unitdata结构指定目的地址、任选项和实际要发送的数据。

```
struct t_unitdata {
struct netbuf addr;          /* protocol-specific address */
struct netbuf opt;           /* protocol-specific options */
struct netbuf udata;         /* user data */
};
```

对于t_rcvudata,该结构含有发送者协议地址、收到的选项以及收到的数据。

如果成功的话,两函数均返回0,否则均返回-1。这与大多数读写函数不同,它们通常返回实际收发的字节数。对于t_rcvudata,读取的字节数可从t_unitdata结构的udata.len成员得到。而对于t_sndudata,我们无法从函数的返回值中得到到底发送了多少字节,它只在成功时返回0(也就是说整个数据报被拷贝到内核缓冲区了)。

由flagsp所指向的整数很像t_rcv的最后一个参数:它不是一个变参,因为函数不检查它的值,而仅由函数在返回时设置其值。如果有必要再次调用t_rcvudata来读取当前数据报的剩余部分(也就是说当前数据报超出了接收缓冲区的长度),那么将返回T_MORE标志。在31.6节我们将给出一个例子。

这两个XTI函数与sendto和recvfrom相对应。

31.3 udp_client函数

在给出使用XTI的UDP程序例子之前,我们编写一个名叫udp_client的函数,其调用序列与11.10节所示的一致,它给UDP客户创建一个XTI端点。图31.1给出了该函数,其中涉及了第29章所讨论的主机名及服务名转换。

```
1 #include    "unpxti.h"
2 int
3 udp_client(const char *host, const char *serv, void **vptr, socklen_t *lenp)
4 {
5     int         tfd;
6     void        *handle;
7     struct netconfig *ncp;
8     struct nd_hostserv hs;
9     struct nd_addrlist *alp;
10    struct netbuf *np;
11    struct t_unitdata *tudptr;
12    handle = Setnetpath();
13    hs.h_host = (char *) host;
14    hs.h_serv = (char *) serv;
15    while ( (ncp = getnetpath(handle)) != NULL) {
16        if (strcmp(ncp->nc_proto, "udp") != 0)
17            continue;
18        if (netdir_getbyname(ncp, &hs, &alp) != 0)
19            continue;
20        tfd = T_open(ncp->nc_device, O_RDWR, NULL);
21        T_bind(tfd, NULL, NULL);
22        tudptr = T_alloc(tfd, T_UNITDATA, T_ADDR);
23        np = alp->n_addrs;          /* use first server address */
24        tudptr->addr.len = min(tudptr->addr.maxlen, np->len);
25        memcpy(tudptr->addr.buf, np->buf, tudptr->addr.len);
26        endnetpath(handle);
27        netdir_free(alp, ND_ADDRLIST);
28        *vptr = tudptr;          /* return pointer to t_unitdata{} */
29        *lenp = tudptr->addr.maxlen;      /* and size of addresses */
30        return(tfd);
31    }
32    endnetpath(handle);
33    return(-1);
34 }
```

图 31.1　使用 XTI 的 udp_client 函数[libxti/udp_client.c]

搜索主机名与服务名

第 12～19 行　getnetpath 和 netdir_getbyname 的调用与图 29.6 类似。

打开设备,捆绑任意本地地址

第 20～21 行　t_open 打开适当的设备,t_bind 捆绑任意本地地址到端点。

分配 t_unitdata 结构

第 22 行　t_alloc 分配一个 t_unitdata 结构,不过结构内只分配 addr 这个 netbuf 结构的成员,而没有 opt 和 udata。不分配 opt 是由于 UDP 很少有针对某个数据报的选项(第 32 章)。而不分配 udata 是考虑到 UDP 数据报长度范围很宽,最大有 65507 字节,然而绝大多数应用程序却不使用最大长度的数据报。既然大多数应用程序处理较小的数据报(最多几千字节),不如让应用进程根据需要自行分配数据缓冲区。

使用头一个返回的服务器地址

第 23～25 行　用 netdir_getbyname 返回的第一个服务器地址填写上面的 addr 结构。图 31.2 展示了所涉及的数据结构,其中假设对于指定的主机名和服务名返回 2 个 netbuf 结

构,并假设使用 sockaddr_in 结构表示 IPv4 地址。

addr_maxlen 值应与 netdir_getbyname 返回的结构中的 maxlen 值相同(IPv4 为 16),不过为了保证 memcpy 不溢出其目的区域,我们还是用了宏 min。在图 31.2 里我们还给出 opt 和 udata 结构的 4 个 0 值长度和 2 个空指针,这是 t_alloc 根据我们以 T_ADDR 参数告诉它只分配和初始化 addr 结构的结果。

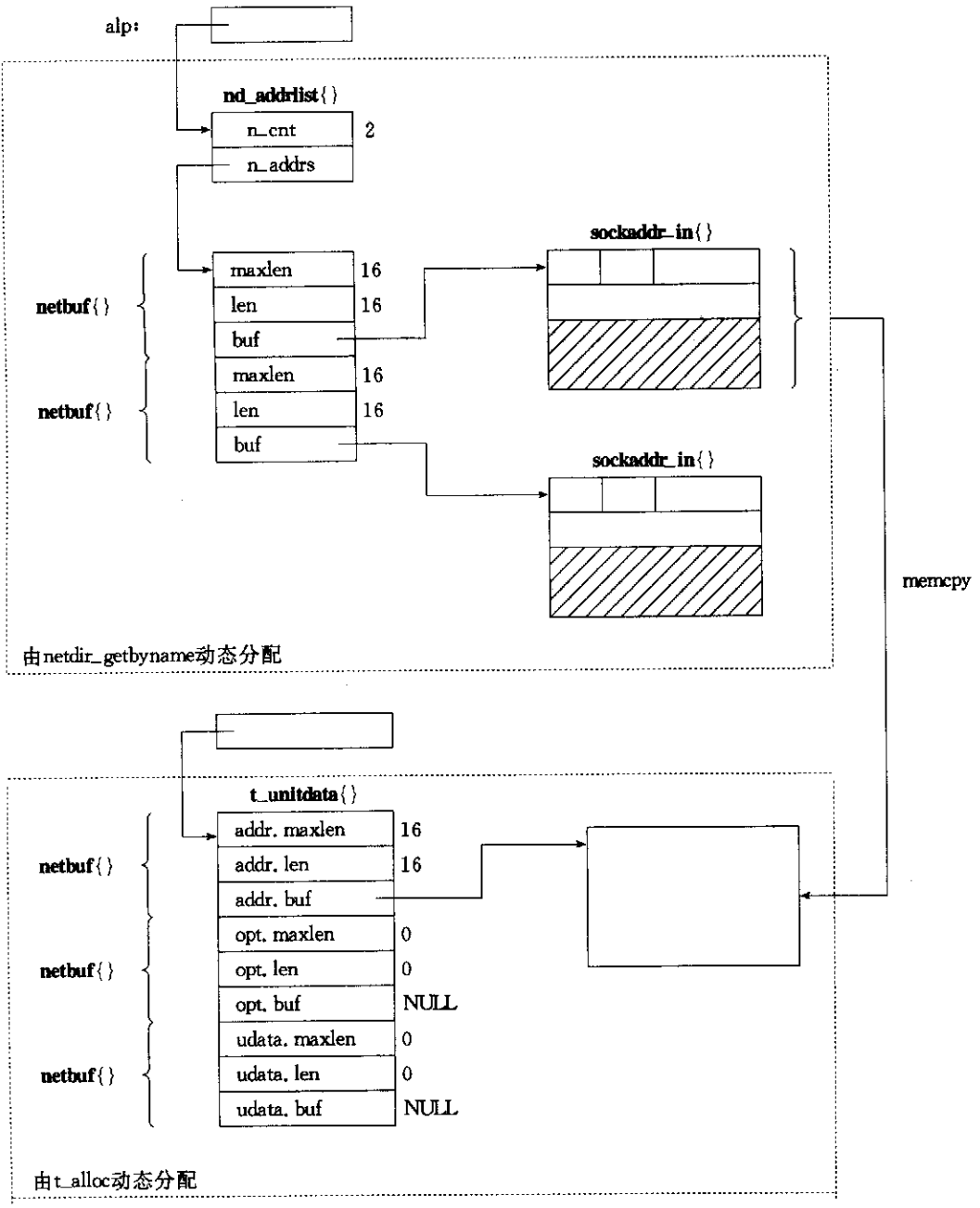


图 31.2 调用 udp_client 期间的数据结构

释放空间并返回

第 26～33 行　endnetpath 释放 netconfig 结构的空间，netdir_free 释放由 netdir_getbyname 分配的空间(图 31.2)。指向 t_unitdata 结构的指针被返回给调用者，同时返回的还有协议地址的大小和端点的描述字。这里，我们使用 addr.maxlen 而不是 addr.len 作为返回地址的大小，因为该值将被用来作为 malloc 的参数。要是地址可变长，使用地址本身长度是不行的，应该使用最大长度。

例子:时间/日期客户程序

现在我们使用 XTI 版本的 udp_client 函数来重新编写图 11.2 的协议无关的时间/日期客户程序，如图31.3所示。

```
1 #include    "unpxti.h"
2 int
3 main(int argc, char **argv)
4 {
5     int             tfd, flags;
6     char            recvline[MAXLINE + 1];
7     socklen_t addrlen;
8     struct t_unitdata *sndptr, *rcvptr;
9     if (argc != 3)
10        err_quit("usage: daytimeudpcli <hostname> <service>");
11    tfd = Udp_client(argv[1], argv[2], (void **) &sndptr, &addrlen);
12    rcvptr = T_alloc(tfd, T_UNITDATA, T_ADDR);
13    printf("sending to %s\n", Xti_ntop_host(&sndptr->addr));
14    sndptr->udata.maxlen = MAXLINE;
15    sndptr->udata.len = 1;
16    sndptr->udata.buf = recvline;
17    recvline[0] = 0;        /* 1-byte datagram containing null byte */
18    T_sndudata(tfd, sndptr);
19    rcvptr->udata.maxlen = MAXLINE;
20    rcvptr->udata.buf = recvline;
21    T_rcvudata(tfd, rcvptr, &flags);
22    recvline[rcvptr->udata.len] = 0;    /* null terminate */
23    printf("from %s: %s", Xti_ntop_host(&rcvptr->addr), recvline);
24    exit(0);
25 }
```

图 31.3　使用 XTI 及新版 udp_client 函数的 UDP 时间/日期客户程序[xtiudp/daytimeudpcli1.c]

创建端点

第 11～13 行　调用 udp_client 创建 XTI 端点，并分配一个用于发送数据报的 t_unitdata 结构。再为接收应答分配另一个 t_unitdata 结构。最后调用 xti_ntop_host 函数输出服务器的 IP 地址。

发送数据报

第 14～18 行　初始化 t_unitdata 结构的 udata 结构，使之指向 recvline 缓冲区，且让它包含一个字节的 0。t_sndudata 向服务器发送数据报。

经过图28.4的讨论,我们知道发送0字节UDP数据报是可行,不过许多XTI的实现不支持这一点。

读取应答

第19~23行 我们给接收t_unitdata结构初始化它的udata结构,然后调用t_rcvudata读取服务器的应答。应答为空字符结尾的字符串,它跟服务器的地址一起输出到标准输出。

与图11.2的套接口API版本UDP程序例子一样,本例子也享有UDP的所有不可靠特性。一旦没有应答,客户将永远阻塞在t_rcvudata调用上。

下面针对一台运行daytime服务器的主机运行本客户程序,结果正如我们所希望的那样:

```
unixware % daytimeudpcltl bsdi daytime
sending to 206.62.226.35
from 206.62.226.35: Fri Feb 28 17:23:40 1997
```

如果我们向同一台主机发送数据报,但这次是向一个未被任何进程绑定的UDP端口发送,会有什么后果?我们预期是一个ICMP端口不可达错。回忆一下套接口客户程序,如果客户不调用connect,则该错误不会返回给客户。对于XTI的访问点,没什么与connect相对应的操作,不过我们看到,错误的确返回给客户,并由T_rcvudata包裹函数输出。

```
unixware % daytimeudpcltl bsdi 9999
sending to 206.62.226.35
t_rcvudata error:event requires attention
```

当接收到一个UDP端点的异步错误时,t_rcvudata返回一个TLOOK错误。我们必须调用t_rcvuderr来确定真正的错误。我们在下一节讨论这个问题。

31.4 t_rcvuderr函数:异步错误

对于无连接协议,错误可以异步地返回。也就是说,一个数据报可以正常地由协议栈发送出去,但在网络别的地方却发现该数据报有错。UDP数据报的通常错误引发的是由目的主机发回的端口不可达ICMP错误,或者由中间路由器发回的主机不可达ICMP错误。一旦提供者收到此类ICMP错误,应该以某种方式通知进程,并提供进程检查实际错误的手段。正如前一节所述,XTI通过使t_rcvudata设置t_errno为TLOOK来指示某个先前发送的数据报出错。我们于是可调用t_rcvuderr函数来确定到底发生了什么,并清除出错状态。

```
#include <xti.h>
    int t_rcvuderr(int fd, struct t_uderr *uderr);
                                                    返回:0——成功,-1——出错
```

如果uderr为非空指针,本函数将用有关错误的信息填写这个t_uderr结构。

```
struct t_uderr {
  struct netbuf addr;          /* protocol-specific address */
  struct netbuf opt;           /* protocol-specific options */
  t_scalar_t error;            /* protocol-specific error */
};
```

addr 结构含有引起错误的数据报的目的地址,opt 结构含有引起错误的数据报的协议特定选项,error 含有协议特定错误码。对于 UDP,error 一般为<sys/errno.h>中的 errno 值之一。

如果 uderr 为空指针,则本函数只清除出错误状态而不返回任何信息。

例子:ICMP 端口不可达

现在,我们修改图 31.3 的客户程序来处理异步错,如图 31.4 所示。

```
1 #include    "unpxti.h"
2 int
3 main(int argc, char **argv)
4 {
5     int         tfd, flags;
6     char        recvline[MAXLINE + 1];
7     socklen_t addrlen;
8     struct t_unitdata *sndptr, *rcvptr;
9     struct t_uderr *uderr;
10    if (argc != 3)
11        err_quit("usage: a.out <hostname or IPaddress> <service or port#>");
12    tfd = Udp_client(argv[1], argv[2], (void **) &sndptr, &addrlen);
13    rcvptr = T_alloc(tfd, T_UNITDATA, T_ADDR);
14    uderr = T_alloc(tfd, T_UDERROR, T_ADDR);
15    printf("sending to %s\n", Xti_ntop_host(&sndptr->addr));
16    sndptr->udata.maxlen = MAXLINE;
17    sndptr->udata.len = 1;
18    sndptr->udata.buf = recvline;
19    recvline[0] = 0;        /* 1-byte datagram containing null byte */
20    T_sndudata(tfd, sndptr);
21    rcvptr->udata.maxlen = MAXLINE;
22    rcvptr->udata.buf = recvline;
23    if (t_rcvudata(tfd, rcvptr, &flags) == 0) {
24        recvline[rcvptr->udata.len] = 0;      /* null terminate */
25        printf("from %s: %s", Xti_ntop_host(&rcvptr->addr), recvline);
26    } else {
27        if (t_errno == TLOOK) {
28            T_rcvuderr(tfd, uderr);
29            printf("error %ld for datagram sent to %s\n",
30                    uderr->error, Xti_ntop_host(&uderr->addr));
31        } else
32            err_xti("t_rcvudata error");
33    }
34    exit(0);
35 }
```

图 31.4　使用 XTI 处理异步错的 UDP 客户程序[xtiudp/daytimeudpcli2.c]

处理异步错误

第23～33行 与图31.3不同之处在于:对包裹函数T_rcvudata的调用被替换成了对t_rcvudata的调用,而且调用t_rcvuderr处理异步错误,并输出返回的错误码。

如果我们运行这个程序,并向一个不支持daytime协议的主机发送数据报,我们将从t_rcvuderr收到ICMP端口不可达错。

```
unixware % daytimeudpcli2 gateway.tuc.noao.edu daytime
sending to 140.252.104.1
error 146 for datagram sent to 140.252.104.1

unixware % grep 146 /usr/include/sys/errno.h
#define ECONNREFUSED 146 /* Connection refused */
```

我们看到在t_uderr结构中返回的error值就是相应ICMP错误的errno值(图A.15)。

非常不幸,虽然这样的设计看来非常完美(给XTI UDP端点返回ICMP错误),但实际上有不少问题。首先,并没有什么要求规定提供者必须将这些错误返回给应用进程。就UnixWare 2.1.2而言,ICMP端口不可达会返回给应用进程,而ICMP主机不可达却不会。其次,如果我们修改客户程序以发送3个数据报给3个不同的服务器,然后读取所有应答,然而其中有2个数据报引发了ICMP端口不可达错,那么只有第一个错误会由t_rcvuderr返回给应用进程。这是由于提供者对每个端点只维护一个错误的缘故。所有这些问题促成我们开发一种通知数据报应用进程异步错误的独立方法:25.7节的icmpd守护进程。

注意,我们收到的只是错误码以及引起错误的数据报的目的地址。未返回的信息之一是返回本错误的主机的源地址(例如ICMP出错消息的源地址)。

31.5 udp_server函数

下面我们也用XTI重新编写图11.4的udp_server函数,如图31.5所示。

```
1 #include    "unpxti.h"
2 int
3 udp_server(const char *host, const char *serv, socklen_t *addrlenp)
4 {
5     int             tfd;
6     void            *handle;
7     struct t_bind tbind;
8     struct t_info tinfo;
9     struct netconfig *ncp;
10    struct nd_hostserv hs;
11    struct nd_addrlist *alp;
12    struct netbuf *np;
13    handle = Setnetconfig();
14    hs.h_host = (host == NULL) ? HOST_SELF : (char *) host;
```

```
15    hs.h_serv = (char *) serv;
16    while ( (ncp = getnetconfig(handle)) != NULL &&
17            strcmp(ncp->nc_proto, "udp") != 0);
18    if (ncp == NULL)
19        return(-1);
20    if (netdir_getbyname(ncp, &hs, &alp) != 0)
21        return(-2);
22    np = alp->n_addrs;          /* use first address */
23    tfd = T_open(ncp->nc_device, O_RDWR, &tinfo);
24    tbind.addr = *np;           /* copy entire netbuf{} */
25    tbind.qlen = 0;             /* not used for connectionless server */
26    T_bind(tfd, &tbind, NULL);
27    endnetconfig(handle);
28    netdir_free(alp, ND_ADDRLIST);
29    if (addrlenp)
30        *addrlenp = tinfo.addr; /* size of protocol addresses */
31    return(tfd);
32 }
```

图 31.5　使用 XTI 的 udp_server 函数[libxti/udp_server.c]

查找协议、主机和服务

第 13～22 行　函数的开始部分类似 tcp_listen 函数(图 30.3),调用 getnetconfig 来找出协议,再调用 netdir_getbyname 来查找主机名和服务名。

打开设备,捆绑服务器的 IP 地址和端口

第 23～28 行　调用 t_open 打开正确设备,再调用 t_bind 捆绑服务器的 IP 地址(如果 host 参数为空指针则为通配地址)和端口。netdir_getbyname 所分配的空间由 netdir_free 释放。

返回地址长度和描述字

第 29～31 行　如果最后一个参数为非空指针则返回地址长度,端点的描述字则作为返回值返回。

例子:时间/日期服务器程序

现在,使用 XTI 重新编写图 11.15 的简单时间/日期服务器程序,见图 31.6。

```
1 #include    "unpxti.h"
2 #include    <time.h>
3 int
4 main(int argc, char **argv)
5 {
6     int         tfd, flags;
7     char        buff[MAXLINE];
8     time_t      ticks;
9     struct t_unitdata *tud;
10    if (argc == 2)
11        tfd = Udp_server(NULL, argv[1], NULL);
```

```
12      else if (argc == 3)
13          tfd = Udp_server(argv[1], argv[2], NULL);
14      else
15          err_quit("usage: daytimeudpsrv [ <host> ] <service or port>");
16      tud = T_alloc(tfd, T_UNITDATA, T_ADDR);
17      for ( ; ; ) {
18          tud->udata.maxlen = MAXLINE;
19          tud->udata.buf = buff;
20          if (t_rcvudata(tfd, tud, &flags) == 0) {
21              printf("datagram from %s\n", Xti_ntop(&tud->addr));
22          ticks = time(NULL);
23          snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
24              tud->udata.len = strlen(buff);
25          T_sndudata(tfd, tud);
26          } else if (t_errno == TLOOK)
27              T_rcvuderr(tfd, NULL);   /* just clear error */
28          else
29              err_xti("t_rcvudata error");
30      }
31  }
```

图31.6 使用XTI的UDP时间/日期服务器程序[xtiudp/daytimeudpsrv2.c]

创建XTI端点

第10～16行 使用udp_server函数创建端点,并捆绑本地IP地址和端口。以T_ADDR参数调用t_alloc分配一个t_unitdata结构,因此只为协议地址分配缓冲区。

读入请求,发送应答

第17～30行 程序进入循环,使用t_rcvudata读取客户请求,再使用t_sndudata发送应答。如果发送的某个应答引起一个异步错误,那么t_rcvudata将返回一个TLOOK错误,而我们则调用t_rcvuderr来处理该错误。注意,t_rcvuderr的最后一个参数为空指针,也就是说只清除出错状态,而不返回任何信息(我们也没什么好处理)。如果我们不像这里所示的那样处理这些错误,而是一旦t_rcvudata返回一个错误就放弃服务器运行的话,那么任何一个客户都有可能导致服务器的崩溃。假设一个客户发送一个数据报后立即终止,那么当它的应答到达客户主机时,该主机会响应以一个ICMP端口不可达错,从而引起服务器的t_rcvudata返回一个错误。因此,处理这些异步错误是使用XTI编写的每个UDP服务器程序必须做的。

31.6 分片读取数据报

回忆一下20.3节有关对数据报进行截短的讨论,以及在UDP套接口上读入的数据报长度超过应用进程请求的字节数时的若干不同情形。XTI的处理方式有所不同。

回忆一下t_rcvudata的最后一个参数flagsp。如果应用进程的缓冲区过小,以至于无法全部装入队列中下一个数据报时,返回的字节数将等于udata.maxlen,且由flagsp参数指向的整数中的T_MORE位将被设置。该标志位通知应用进程再次调用t_rcvudata以读取当前数据报的剩余部分。发送者地址和选项仅在对给定的数据报首次调用t_rcvudata时返回。对于所有后续的t_rcvudata调用(读取同一数据报剩余部分),addr.len和opt.len成员都将

返回 0。

我们通过将图 31.4 的客户程序修改成图 31.7 以展示这种特性的用法。

```
 1 #include    "unpxti.h"
 2 #undef  MAXLINE
 3 #define MAXLINE   2
 4 int
 5 main(int argc, char **argv)
 6 {
 7     int           tfd, flags;
 8     char          recvline[MAXLINE + 1];
 9     socklen_t addrlen;
10     struct t_unitdata *sndptr, *rcvptr;
11     struct t_uderr *uderr;
12     if (argc != 3)
13         err_quit("usage: a.out <hostname or IPaddress> <service or port#>");
14     tfd = Udp_client(argv[1], argv[2], (void **) &sndptr, &addrlen);
15     rcvptr = T_alloc(tfd, T_UNITDATA, T_ADDR);
16     uderr = T_alloc(tfd, T_UDERROR, T_ADDR);
17     printf("sending to %s\n", Xti_ntop_host(&sndptr->addr));
18     sndptr->udata.maxlen = MAXLINE;
19     sndptr->udata.len = 1;
20     sndptr->udata.buf = recvline;
21     recvline[0] = 0;        /* 1-byte datagram containing null byte */
22     T_sndudata(tfd, sndptr);
23     do {
24         rcvptr->udata.maxlen = MAXLINE;
25         rcvptr->udata.buf = recvline;
26         flags = 0;
27         if (t_rcvudata(tfd, rcvptr, &flags) == 0) {
28             recvline[rcvptr->udata.len] = 0;     /* null terminate */
29             if (rcvptr->addr.len > 0)
30                 printf("from %s: ", Xti_ntop_host(&rcvptr->addr));
31             printf("%s\n", recvline);
32         } else {
33             if (t_errno == TLOOK) {
34                 T_rcvuderr(tfd, uderr);
35                 printf("error %ld from %s\n",
36                         uderr->error, Xti_ntop_host(&uderr->addr));
37             } else
38                 err_xti("t_rcvudata error");
39             flags = 0;
40         }
41     } while (flags & T_MORE);
42     exit(0);
43 }
```

图 31.7　使用 XTI 的分片读取返回的数据报的 UDP 客户程序[xtiudp/daytimeudpcli4.c]

重定义MAXLINE

第2～3行 将udp.h头文件中的MAXLINE重定义为2,这也是recvline缓冲区的大小。

创建端点,向服务器发送数据报

第12～22行 这段代码与图31.4相比没有变化。

读取应答,一次2字节

第23～41行 使用一个循环读取应答,直至flags变量的T_MORE位不再被置位。当addr.len不为0即第一次调用t_rcvudata时,输出服务器IP地址。

现在,对某个daytime服务器运行这个客户程序

```
unixware % daytimeudpcli4 bsdi daytime
sending to 206.62.226.35
from 206.62.226.35:Su
n
Ma
r
2
1
1:
53
:5
0
19
97
```

如果我们从recvline的printf格式化字符串中去掉换行符(第31行,我们用它来展示t_rcvudata一次返回多少数据),那么输出结果如下:

```
unixware % daytimeudpcli4 bsdi daytime
sending to 206.62.226.35
from 206.62.226.35: Sun Mar 2 12:04:48 1997
```

31.7 小 结

两个XTI函数t_rcvudata和t_sndudata类似于recvfrom和sendto。不过相对于套接口函数有个新功能,即允许分片读取一个数据报,并通过T_MORE标志来指出当前数据报是否还有剩余数据可读。

当发生异步错误时,t_rcvudata和t_sndudata会返回一个TLOOK错误。我们接着调用t_rcvuderr来获取关于这个错误更详细的协议相关信息。这比套接口的处理方式(仅当套接口已连接时才返回错误)要好。不过即使这样仍可能丢失异步错误,且依赖于协议栈来决定返回哪些ICMP错误。一个较好的解决办法是使用类似icmpd的守护进程(25.7节),在一个分离的通道上返回所有错误。

第 32 章　XTI 选项

32.1　概　述

XTI 的另一个神秘领域是选项处理。虽然标准和手册都用大量篇幅来介绍选项处理和选项协商，但都没有给出具体例子，总以“具体细节协议相关”来结尾。

协商(negotiation)这个词对 XTI 选项而言很常用。选项不是设定的，而是协商定的，也就是说提供者可能不完全按照我们的请求设置选项。不过在协商完成后，提供者返回最终结果，因此我们能够看到由它实际使用的值。

图 32.1 给出了所有标准的 XTI 选项，包括通用选项(XTI_开头的)和 IPv4 专用选项。

> Unix98 给所有的 INET_、IP_、TCP_和 UDP_名字加了个 T_前缀，不过 Posix.1g 不这么做。例如，在 Posix.1g 里，UDP 的校验和选项叫 UDP_CHEKCKSUM。Unix98 认可 Posix.1g 格式的名字为合法名字，但我们在本书中将使用新的命名法。

XTI 将选项分为两类：本地的(local)或者端到端的(end-to-end)。端到端类的选项通常导致某种类型的信息被传送到对方。例子之一是 IPv4 的服务类型(type-of-service)字段，它可以由一个(TCP 或 UDP)端点设置，由 IPv4 头部携带，并在对方端点可用。图 32.1 中的 IPv4 包头部选项及 UDP 校验和是另外两个端到端选项。本地选项的一个例子是 T_IP_REUSEADDR，该选项影响调用进程捆绑一个已被其端点使用的端口号的能力，但对对方端点没有什么影响。

级别	名字	数据类型	端到端	绝对要求	说明
XTI_GENERIC	XTI_DEBUG	t_uscalar_t[]		•	使能调试跟踪
	XTI_LINGER	t_linger{}		•	有数据待发送则延时关闭
	XTI_RCVBUF	t_uscalar_t			接收缓冲区大小
	XTI_RCVLOWAT	t_uscalar_t			接收缓冲区低潮限度
	XTI_SNDBUF	t_uscalar_t			发送缓冲区大小
	XTI_SNDLOWAT	t_uscalar_t			发送缓冲区低潮限度
T_INET_IP	T_IP_BROADCAST	u_int		•	允许发送广播消息
	T_IP_DONTROUTE	u_int		•	绕过路由表查找
	T_IP_OPTIONS	u_char[]	•	•	IP 头部选项
	T_IP_REUSEADDR	u_int		•	允许本地地址重用
	T_IP_TOS	u_char	•	•	服务类型和优先级
	T_IP_TTL	u_char		•	存活时间
T_INET_TCP	T_TCP_KEEPALIVE	u_kpalive{}		•	周期性测试连接是否存活
	T_TCP_MAXSEG	t_uscalar_t			TCP MSS(只读)
	T_TCP_NODELAY	t_uscalar_T		•	禁止 Nagle 算法
T_INET_UDP	T_UDP_CHECKSUM	t_uscalar_t	•	•	使能 UDP 校验和

图 32.1　XTI 选项

有些XTI选项被归为绝对要求(absolute requirement),我们在图32.1中也给出了这种属性。当我们设置具有该属性的选项时,如果系统无法按要求设置,操作就会失败。而对于那些没有该属性的选项,如果我们设置了一个不在系统支持范围内的值,提供者将会用一个可接受的值来代替。后者的一个例子是接收缓冲区的大小XTI_RCVBUF,因为大多数系统对其值施加一个上限和一个下限。如果我们请求一个小于下限或大于上限的值,其实际值将会被改成相应的下限值或上限值,返回值于是是"部分成功"。

XTI选项以如下方法指定和接收:

1. 调用t_optmgmt函数可以指定任意期望的选项(端到端和本地)。我们也可以调用该函数来获取某个选项的当前值或缺省值。
2. 对于UDP端点,使用t_unitdata结构的opt成员,我们可以随每个t_sndudata调用指定期望的选项(端到端和本地)。
3. 对于UDP端点,所有伴随数据报而来的端到端选项都可由t_rcvudata通过t_unitdata结构中的opt成员返回。
4. 对于TCP客户,我们可以在调用t_connect时作为t_call结构的opt成员指定期望的选项(端到端和本地)。
5. 对于TCP服务器,所有伴随连接而来的端到端选项都可由t_listen通过t_call结构中的opt成员返回。

t_optmgmt函数是getsockopt和setsockopt的混合。不过在套接口API下,在收发UDP数据报时,或者在发起或接受连接时,无法指定选项。sendmsg和recvmsg函数提供了指定和接收辅助数据的能力,IPv6选项使用这种能力指定和接收。

图32.2总结了XTI函数发送和接收选项的情况。

端点	函数	只返回端到端选项	返回端到端和局部选项	指定端到端和局部选项
任意端点	t_optmgmt		•	•
TCP端点	t_accept			•
	t_connect		•	•
	t_listen	•		
	t_rcvconnect		•	
UDP端点	t_rcvudata	•		
	t_rcvvudata	•		
	t_rcvuderr		•	
	t_sndudata			•
	t_sndvudata			•

图32.2 可指定和返回选项的XTI函数

由图32.2可以得出,我们可以在t_accept时指定选项。对于TCP这条不适用,因为t_

listen 返回时，连接已经建立完毕。因此，所有期望对三路握手有影响的选项都必须指定在监听端点上。

32.2　t_opthdr 结构

XTI 选项总是通过一个名字为 opt 的 netbuf 结构来设置和返回的，该结构为 t_call、t_optmgmt、t_uderr 和 t_unitdata 结构的成员之一(图 28.6)。选项缓冲区的内容是一个或多个 t_opthdr 结构，每个结构后跟一个可选的值。

```
struct t_opthdr {
  t_uscalar_t len;          /* total length of option:
                               sizeof(struct t_opthdr) + length of value */
  t_uscalar_t level;        /* protocol affected */
  t_uscalar_t name;         /* option name */
  t_uscalar_t status;       /* status value */
          /* followed by the option value, and then possible padding */
};
```

XTI 和 TLI 的区别之一就是 TLI 对选项缓冲区的格式不做任何规定，只是说它依赖于具体实现。许多 TLI 实现使用一个名为 opthdr 的结构，该结构只有三个成员：level、name 和 len。

我们在图 32.3 中展示了两个这样的 XTI 结构，指向它们的 netbuf 结构是一个 t_unitdata 结构的一部分。

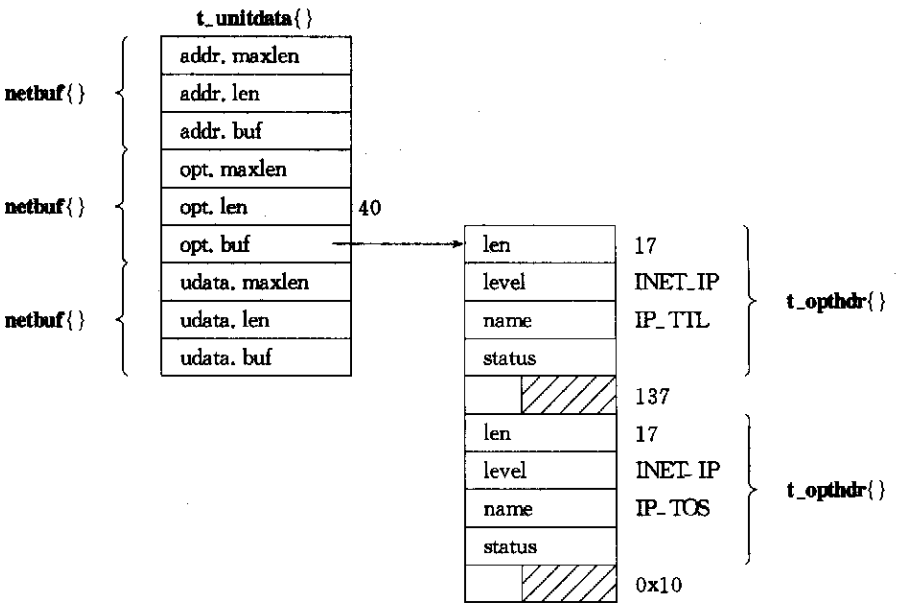


图 32.3　由一个 netbuf 结构指向的两个选项的例子

在图 32.3 中我们设 IP 的 TTL 为 137，设 IP 的服务类型为 0x10(常规优先级与低延时)。每个选项值是一个 1 字节 u_char(图 32.1)，我们在每个值后各补了 3 个字节。我们同

样假设这里的t_uscalar_t占4个字节;因此整个选项缓冲区为40个字节。

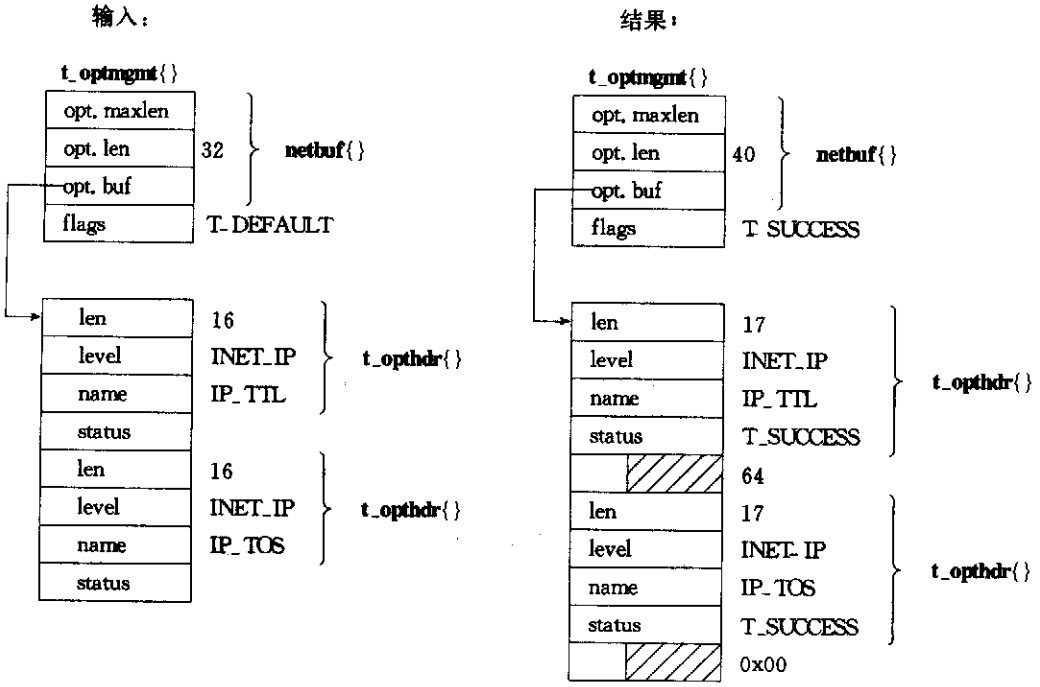


图32.4 从t_optmgmt请求两个选项的缺省值

图32.4是另一例子,这次是对t_optmgmt函数的调用参数(见32.4节)。该函数使用两个指向t_optmgmt结构的指针作为参数:一个是输入,另外一个是结果。

在这个例子中,我们要获取IP TTL和TOS选项的缺省值(flags为T_DEFAULT),因此我们只给每个选项指定一个不带任何值的t_opthdr结构。结果是输入的拷贝,不过缺省值已返回到每个t_opthdr结构的后面。在结果结构中status成员也被填写。

> Unix98(而不是Posix.1g)定义了三个宏来处理t_opthdr结构及其后的数据:T_OPT_FIRSHDR、T_OPT_NEXTHDR和T_OPT_DATA。它们与套接口API中处理辅助数据的CMSG_FIRSTHDR、CMSG_NXTHDR和CMSG_DATA类似(见13.6节)。

32.3 XTI选项

大多数XTI选项可以和第7章所讲的某个套接口选项建立直接对应关系,因此我们只对它们进行简略描述。此外我们应注意到,为了引入IP、TCP和UDP选项所用的常值,需要包括<xti_inet.h>头文件。

> 注意,XTI未定义任何多播方式。

XTI_DEBUG 选项

该选项与 SO_DEBUG 套接口选项类似，通常只由 TCP 支持。如果光指定一个选项头部而不给出其值，则清除该选项。不给出选项值的意思是 t_opthdr 结构的 len 成员正好是结构本身的大小(例如图 32.4 中为 16 字节)。

XTI_LINGER 选项

该选项类似于 SO_LINGER 套接口选项，且由 TCP 支持。它指示在关闭端点时的动作。t_linger 结构如下：

```
struct t_linger {
  t_scalar_t l_onoff;       /* T_NO, T_YES */
  t_scalar_t l_linger;      /* T_UNSPEC (use default), T_INFINITE,
                               or linger time in seconds */
};
```

我们在图 32.1 中已经指明，该选项是一个“绝对”值，不过只有 l_onoff 是绝对要求的，l_linger 并不是。也就是说，具体实现可以在延迟时间上设上下限。

与 SO_LINGER 套接口选项不同，XTI_LINGER 不用来发送 RST。发送 RST 由 t_snddis 完成。

XTI_RCVBUF 和 XTI_RCVLOWAT 选项

这两个选项类似于 SO_RCVBUF 和 SO_RCVLOWAT 套接口选项。第一个选项指定端点接收缓冲区的大小，第二个选项指定用于 select 或 poll 的接收缓冲区低潮限度。

图 32.1 未将 XTI_RCVBUF 归为端到端选项，不过如果使用 TCP 的长胖管道支持(RFC 1323[Jacobson, Braden and Borman 1992])，该选项的确有端到端的重要性，因为它影响三路握手时对 TCP 窗口规模选项的协商。

XTI_SNDBUF 和 XTI_SNDLOWAT 选项

这两个选项与 SO_SNDBUF 和 SO_SNDLOWAT 套接口选项类似。头一个选项指定端点发送缓冲区的大小，第二个选项指定由 select 或 poll 使用的发送缓冲区低潮限度。

T_IP_BROADCAST 选项

该选项与 SO_BROADCAST 套接口选项类似，选项值可以是 T_YES 或者 T_NO。

T_IP_DONTROUTE 选项

该选项与 SO_DONTROUTE 套接口选项类似，选项值可以是 T_YES 或 T_NO。

T_IP_OPTIONS 选项

该选项与 IP_OPTIONS 套接口选项类似。该选项值用作 IPv4 头部选项。IPv4 头部选项的例子见第 24 章。如果不指定选项值(即只有一个 t_opthdr 结构)，则表示清除该选项。

使用 T_CURRENT 请求调用 t_optmgmt 可得到将用在外出数据报上的各 IP 选项的当前值。

T_IP_REUSEADDR选项

该选项类似于SO_REUSEADDR套接口选项,它的值可以是T_YES或T_NO。

T_IP_TOS选项

该选项类似于IP_TOS套接口选项。选项值是如图32.5所示的IPv4优先级字段和如图32.6所示的IPv4服务类型字段的组合。

常 数	值
T_ROUTINE	0
T_PRIORITY	1
T_IMMEDIATE	2
T_FLASH	3
T_OVERRIDEFLASH	4
T_CRITIC_ECP	5
T_INETCONTROL	6
T_NETCONTROL	7

图32.5 T_IP_TOS选项使用的IPv4优先级值

常 数	描 述
T_NOTOS	普通
T_LDELAY	最小延迟
T_HITHRPT	最大吞吐量
T_HIREL	最高可靠性
T_LOCOST	最小成本

图32.6 T_IP_TOS选项使用的IPv4服务类型值

宏SET_TOS(在<xti.h>头文件中定义)将其第一个参数(图32.5的优先级值)与第二个参数(图32.6的服务类型值)进行组合,结果可用于T_IP_TOS选项。

使用T_CURRENT请求调用t_optmgmt可返回将用于外出数据报的该选项的当前值。

T_IP_TTL选项

该选项类似于IP_TTL套接口选项。选项的值对应IPv4的存活时间。该选项可用于设置外出数据报的TTL。然而我们无法得到所收到的数据报的TTL。

T_TCP_KEEPALIVE选项

该选项类似于SO_KEEPALIVE套接口选项,它控制TCP连接中保持存活分组的发送。此XTI选项使用如下结构:

```
struct t_kpalive {
  t_scalar_t kp_onoff;     /* T_NO (disable),T_YES (enable), or
                              T_YES |T_GARBAGE(enable & send garbage byte) */
  t_scalar_t kp_timeout;   /* timeout in minutes; T_UNSPEC means default */
};
```

该选项与 XTI_LINGER 选项有一点相同，即 kp_onoff 值是绝对要求的，而 kp_timeout 则不是。

> 发送一个垃圾字节不应该是必须的，事实上 Unix98 已经取消了 T_GARBAGE。TCPv1 第 335 页讨论了垃圾字节的使用。

T_TCP_MAXSEG 选项

该选项与 TCP_MAXSEG 套接口选项类似。它是只读的，返回一个 TCP 端点的最大分节大小(MSS)。由于是只读选项，其值不可能是绝对要求的。

T_TCP_NODELAY 选项

该选项与 TCP_NODELAY 套接口选项类似。选项值为 T_YES(禁止 Nagle 算法)或 T_NO(缺省值，使能 Nagle 算法)。在 7.9 节我们对 Nagle 算法有详细的介绍。

T_UDP_CHECKSUM 选项

这个 XTI 选项是一个端到端选项，因此，如果请求接收到的选项的话(也就是说 opt.maxlen 不为 0)，它总是由 t_rcvudata 返回。选项值为 T_YES 或 T_NO。

> 该选项永远不该设置，而且提供应用程序以禁止在某个端点上发送 UDP 校验和的能力也是一个错误。已有例子表明，当 UDP 校验和功能被禁止时数据会被破坏，而且也没有什么不使用校验和的理由。该选项唯一的用途是检查对方使能 UDP 校验和了没有。

32.4　t_optmgmt 函数

t_optmgmt 函数可执行如下 XTI 操作：

- 检查一个或多个选项是否被支持。
- 获取一个或多个选项的缺省值。
- 获取一个或多个选项的当前值。
- 协商一个或多个选项的值。

```
#include <xti.h>
int t_optmgmt(int fd, const struct t_optmgmt *request, struct t_optmgmt *result);
                                                返回：0——成功，-1——出错
```

我们通过一个 t_optmgmt 结构指定我们的要求,结果则通过另一个 t_optmgmt 结构返回。如果我们对结果不感兴趣,可将该结构的 maxlen 成员设为 0。图 32.4 给出了 t_optmgmt 结构的两个例子。

```
struct t_optmgmt {
  struct netbuf opt;      /* one or more t_opthdr structures */
  t_scalar_t flags;       /* action on input ,result on output */
};
```

request 结构的 flags 成员指定调用者期望的行为:

T_CHECK 检验选项是否受支持
T_DEFAULT 返回选项缺省值
T_CURRENT 返回选项当前值
T_NEGOTIATE 协商选项值

我们将在以下各节逐个检查这四个操作。

如图 32.4 所示,我们可以在一个 t_optmgmt 调用中指定多个选项,但必须是同一级别(t_opthdr 结构的 level 成员)的。图 32.4 之所以可行是因为那两个选项的级别均为 T_INET_IP。在同一个调用中协商多个选项的新值还存在一个问题:尽管每个选项都可以含有自己的 status 返回值,总的返回 flags 只能取最坏情况的那个单值。为了避免以上问题,还是每次调用 t_optmgmt 处理一个选项为好。

该函数相应于 getsockopt 和 setsockopt 函数。

32.5 检查选项是否受支持并获取缺省值

我们的第一个例子是检查系统支持图 32.1 中的哪些选项,然后输出每个受支持选项的缺省值,图 32.7 为该程序。

```
1 #include    "unpxti.h"
2 struct xti_opts {
3   char              *opt_str;
4   t_uscalar_t       opt_level;
5   t_uscalar_t       opt_name;
6   char          *(*opt_val_str)(struct t_opthdr *);
7 } xti_opts[] = {
8     "XTI_DEBUG",          XTI_GENERIC,   XTI_DEBUG,          xti_str_uscalard,
9     "XTI_LINGER",         XTI_GENERIC,   XTI_LINGER,         xti_str_linger,
10    "XTI_RCVBUF",         XTI_GENERIC,   XTI_RCVBUF,         xti_str_uscalard,
11    "XTI_RCVLOWAT",       XTI_GENERIC,   XTI_RCVLOWAT,       xti_str_uscalard,
12    "XTI_SNDBUF",         XTI_GENERIC,   XTI_SNDBUF,         xti_str_uscalard,
13    "XTI_SNDLOWAT",       XTI_GENERIC,   XTI_SNDLOWAT,       xti_str_uscalard,
14    "T_IP_BROADCAST",     T_INET_IP,     T_IP_BROADCAST,     xti_str_uiyn,
15    "T_IP_DONTROUTE",     T_INET_IP,     T_IP_DONTROUTE,     xti_str_uiyn,
16    "T_IP_OPTIONS",       T_INET_IP,     T_IP_OPTIONS,       xti_str_uchard,
17    "T_IP_REUSEADDR",     T_INET_IP,     T_IP_REUSEADDR,     xti_str_uiyn,
18    "T_IP_TOS",           T_INET_IP,     T_IP_TOS,           xti_str_ucharx,
```

```
19      "T_IP_TTL",          T_INET_IP,     T_IP_TTL,          xti_str_uchard,
20      "T_TCP_KEEPALIVE",   T_INET_TCP,    T_TCP_KEEPALIVE,   xti_str_kpalive,
21      "T_TCP_MAXSEG",      T_INET_TCP,    T_TCP_MAXSEG,      xti_str_uscalard,
22      "T_TCP_NODELAY",     T_INET_TCP,    T_TCP_NODELAY,     xti_str_usyn,
23      "T_UDP_CHECKSUM",    T_INET_UDP,    T_UDP_CHECKSUM,    xti_str_usyn,
24      NULL,                0,             0,                 NULL
25  };
26  int
27  main(int argc, char **argv)
28  {
29      int         fd;
30      struct t_opthdr *topt;
31      struct t_optmgmt *req, *ret;
32      struct xti_opts *ptr;
33      if (argc != 2)
34          err_quit("usage: checkopts <device>");
35      fd = T_open(argv[1], O_RDWR, NULL);
36      T_bind(fd, NULL, NULL);
37      req = T_alloc(fd, T_OPTMGMT, T_ALL);
38      ret = T_alloc(fd, T_OPTMGMT, T_ALL);
39      for (ptr = xti_opts; ptr->opt_str != NULL; ptr++) {
40          topt = (struct t_opthdr *) req->opt.buf;
41          topt->level = ptr->opt_level;
42          topt->name = ptr->opt_name;
43          topt->len = sizeof(struct t_opthdr);
44          req->opt.len = topt->len;
45          req->flags = T_CHECK;
46          printf("%s: ", ptr->opt_str);
47          if (t_optmgmt(fd, req, ret) < 0) {
48              err_xti_ret("t_optmgmt error");
49          } else {
50              topt = (struct t_opthdr *) ret->opt.buf;
51              printf("%s", xti_str_flags(topt->status));
52              if (topt->status == T_SUCCESS || topt->status == T_READONLY) {
53                  req->flags = T_DEFAULT;
54                  if (t_optmgmt(fd, req, ret) < 0) {
55                      err_xti_ret("t_optmgmt error for T_DEFAULT");
56                  } else {
57                      topt = (struct t_opthdr *) ret->opt.buf;
58                      printf(", default = %s", (*ptr->opt_val_str)(topt));
59                  }
60              }
61              printf("\n");
62          }
63      }
64      exit(0);
65  }
```

图 32.7　检查都支持哪些 XTI 选项[xtiopt/checkopts.c]

第2～25行 定义和初始化一个结构来包含图32.1中的各XTI选项。数组内各元素的最后一个成员是一个输出本选项值的函数的指针。对于每个不同的选项类型,我们分别使用不同的函数。我们未给出这些函数的源码。

打开设备

第35～38行 我们取作为命令行参数之一的设备名并打开该设备。这使得我们可以运行该程序两次,一次使用/dev/tcp,一次使用/dev/udp,因为我们预期不同提供者提供对不同选项的支持。由于大多数对t_optmgmt的调用要求端点已绑定,因此我们给它捆绑任意本地地址。我们分配两个t_optmgmt结构,一个用于请求,一个用于函数应答。

使用T_CHECK请求调用t_optmgmt

第39～48行 对xti_opts数组中的每个选项使用T_CHECK请求标志调用t_optmgmt。我们填写req结构,在opt缓冲区(32.2节)中构造单个t_opthdr结构,不带任何数据(与图32.4中左边类似)。

使用T_DEFAULT请求调用t_optmgmt

第49～62行 如果第一次t_optmgmt调用成功,我们输出选项的status。如果status为T_SUCCESS或T_READONLY,那么再次调用t_optmgmt,这次的请求是T_DEFAULT。如果第二次调用也成功,则调用图32.7中当前xti_opts结构中的opt_var_str成员所指向的函数来输出当前选项缺省值。当第二次调用t_optmgmt时,我们只改动请求所在结构的flags成员。既然指向请求所在结构的指针在t_optmgmt的函数原型中有const限定词,我们肯定该结构不会被第一次调用所修改。

下面我们在AIX4.2上分别对TCP和UDP运行两次程序。注意,AIX使用的设备名是/dev/xti/tcp和/dev/xti/udp。

```
aix % checkopts /dev/xti/tcp
XTI_DEBUG:T_SUCCESS, default = 0
XTI_LINGER: T_SUCCESS, default = T_NO, 0 sec
XTI_RCVBUF: T_SUCCESS, default = 16384
XTI_RCVLOWAT: T_SUCCESS, default = 1
XTI_SNDBUF: T_SUCCESS, default = 16384
XTI_SNDLOWAT: T_SUCCESS, default = 4096
T_IP_BROADCAST: T_SUCCESS, default = T_NO
T_IP_DONTROUTE: T_SUCCESS, default = T_NO
T_IP_OPTIONS: T_SUCCESS, default = 0 (length of value)
T_IP_REUSEADDR: T_SUCCESS, default = T_NO
T_IP_TOS: T_SUCCESS, default = 0x00
T_IP_TTL: T_SUCCESS, default = 0
T_TCP_KEEPALIVE: T_SUCCESS, default = T_NO, T_UNSPEC
T_TCP_MAXSEG: T_READONLY,default = 512
T_TCP_NODELAY: T_SUCCESS, default = T_NO
T_UDP_CHECKSUM: t_optmgmt error: incorrect option format

aix % checkopts /dev/xti/udp
XTI_DEBUG: T_SUCCESS, default = 0
XTI_LINGER: T_SUCCESS, default = T_NO, 0 sec
XTI_RCVBUF: T_SUCCESS, default = 41600
```

```
XTI_RCVLOWAT: T_SUCCESS, default = 1
XTI_SNDBUF: T_SUCCESS, default = 9216
XTI_SNDLOWAT: T_SUCCESS, default = 4096
T_IP_BROADCAST: T_SUCCESS, default = T_NO
T_IP_DONTROUTE: T_SUCCESS, default = T_NO
T_IP_OPTIONS: T_SUCCESS, default = 0 (length of value)
T_IP_REUSEADDR: T_SUCCESS, default = T_NO
T_IP_TOS: T_SUCCESS, default = 0x00
T_IP_TTL: T_SUCCESS, default = 0
T_TCP_KEEPALIVE: t_optmgmt error: incorrect option format
T_TCP_MAXSEG: t_optmgmt error: incorrect option format
T_TCP_NODELAY: t_optmgmt error: incorrect option format
T_UDP_CHECKSUM: T_NOTSUPPORT
```

除了 T_IP_TTL 外，受支持的值都在预料之中。TCP 不支持的 T_UDP_CHECKSUM 以及 UDP 不支持的 3 个 TCP 选项则使 t_optmgmt 返回 TBADOPT 错误。UDP 提供者理解 T_UDP_CHECKSUM，但返回 T_NOTSUPPORT，因为它不受支持。给 T_IP_OPTIONS 选项输出的字符串"(length of value)"表示返回的 len 成员为 0，所以没有什么值可以输出。

32.6　获取和设置 XTI 选项

现在我们给出获取和设置 XTI 选项的例子。我们定义两个自己的函数 xti_getopt 和 xti_setopt。它们与 getsockopt 和 setsockopt(7.2 节)的调用序列相同。

```
#include "unpxti.h"
int xti_getopt(int fd, int level, int name, void *optval, socklen_t *optlen);
int xti_setopt(int fd, int level, int name, const void *optval, socklen_t optlen);
                                        二者均返回：0——成功，-1——出错
```

这些函数每一个都包含 20～30 行 C 代码，可以简化我们的 XTI 程序。

xti_getopt 函数

将请求结构的 flags 成员置为 T_CURRENT 来调用 t_optmgmt 将返回指定选项的当前值。图 32.8 为 xti_getopt 函数。

```
 1 #include    "unpxti.h"
 2 int
 3 xti_getopt(int fd, int level, int name, void *optval, socklen_t *optlenp)
 4 {
 5     int     rc, len;
 6     struct t_optmgmt  *req, *ret;
 7     struct t_opthdr       *topt;
 8     req = T_alloc(fd, T_OPTMGMT, T_ALL);
 9     ret = T_alloc(fd, T_OPTMGMT, T_ALL);
10     if (req->opt.maxlen == 0)
```

```
11          err_quit("xti_getopt: opt.maxlen == 0");
12      topt = (struct t_opthdr *) req->opt.buf;
13      topt->level = level;
14      topt->name = name;
15      topt->len = sizeof(struct t_opthdr);      /* just a t_opthdr{} */
16      req->opt.len = topt->len;
17      req->flags = T_CURRENT;
18      if (t_optmgmt(fd, req, ret) < 0) {
19          T_free(req, T_OPTMGMT);
20          T_free(ret, T_OPTMGMT);
21          return(-1);
22      }
23      rc = ret->flags;
24      if (rc == T_SUCCESS || rc == T_READONLY) {
25              /* 4copy back value and length */
26          topt = (struct t_opthdr *) ret->opt.buf;
27          len = topt->len - sizeof(struct t_opthdr);
28          len = min(len, *optlenp);
29          memcpy(optval, topt+1, len);
30          *optlenp = len;
31      }
32      T_free(req, T_OPTMGMT);
33      T_free(ret, T_OPTMGMT);
34      if (rc == T_SUCCESS || rc == T_READONLY)
35          return(0);
36      return(-1);      /* T_NOTSUPPORT */
37  }
```

图32.8 xti_getopt函数:获取一个XTI选项的当前值[libxti/xti_getopt.c]

分配请求和应答结构

第8～11行 调用t_alloc分配两个结构,分别用于请求和应答。同时验证选项缓冲区的大小不为0。

> 老版本的TLI经常使用值0作为TCP选项的大小,表示应用程序必须自己分配缓冲区。

填写t_opthdr结构

第12～16行 填写t_opthdr结构的level和name成员。我们不在请求结构中指定任何值,因为在获取选项当前值时这没有什么用。

调用t_optmgmt并返回选项值

第17～31行 调用t_optmgmt并保存返回的应答结构中的flags成员。如果返回值为T_SUCCESS或T_READONLY,则拷回选项的值和它的大小。(指针表达式topt+1指向返回的选项值,因为它紧跟在t_opthdr结构之后)。本函数的最后一个参数是变参,我们必须小心,不要使调用者的缓冲区溢出(以防它万一太小)。

释放空间并返回

第 32～36 行　释放由 t_alloc 分配的内存，成功则返回 0，失败则返回－1。

xti_setopt 函数

为了设置 XTI 选项，可将请求结构的 flags 成员设为 T_NEGOTIATE 来调用 t_optmgmt。图 32.9 给出了 xti_setopt 函数，该函数与图 32.8 很相似。

拷贝调用者的值

第 12～19 行　我们把调用者给定的选项值拷入调用 t_alloc 分配的缓冲区，紧紧放在 t_opthdr 结构之后。

调用 t_optmgmt

第 20～26 行　t_optmgmt 的请求结构的 flags 成员现在置为 T_NEGOTIATE。

释放空间并返回

第 27～31 行　如果选项值不是绝对要求的，则返回值为 T_PARTSUCCESS 也是允许的。

XTI 允许我们在一次 t_optmgmt 调用中设置一个选项并取回选项值。这对其值非绝对要求的选项（譬如发送和接收缓冲区的大小）可能有用。使用我们的 xti_setopt 和 xti_getopt 函数则需要一前一后调用两次函数。我们也可以写一个函数来一次完成这两个操作，不过额外调用 xti_getopt 不太可能会成为应用程序的瓶颈。

```
1 #include    "unpxti.h"
2 int
3 xti_setopt(int fd, int level, int name, void *optval, socklen_t optlen)
4 {
5     int           rc;
6     struct t_optmgmt *req, *ret;
7     struct t_opthdr *topt;
8     req = T_alloc(fd, T_OPTMGMT, T_ALL);
9     ret = T_alloc(fd, T_OPTMGMT, T_ALL);
10    if (req->opt.maxlen == 0)
11        err_quit("xti_setopt: req.opt.maxlen == 0");
12    topt = (struct t_opthdr *) req->opt.buf;
13    topt->level = level;
14    topt->name = name;
15    topt->len = sizeof(struct t_opthdr) + optlen;
16    if (topt->len > req->opt.maxlen)
17        err_quit("optlen too big");
18    req->opt.len = topt->len;
19    memcpy(topt+1, optval, optlen);      /* copy option value */
20    req->flags = T_NEGOTIATE;
21    if (t_optmgmt(fd, req, ret) < 0) {
22        T_free(req, T_OPTMGMT);
23        T_free(ret, T_OPTMGMT);
24        return(-1);
25    }
```

```
26      rc = ret->flags;
27      t_free(req, T_OPTMGMT);
28      t_free(ret, T_OPTMGMT);
29      if (rc == T_SUCCESS || rc == T_PARTSUCCESS)
30          return(0);
31      return(-1);     /* T_FAILURE, T_NOTSUPPORT, T_READONLY */
32  }
```

图 32.9 xti_setopt 函数:设置一个 XTI 选项的值[libxti/xti_setopt.c]

例子

我们现在使用我们刚给的两个函数。图 32.10 中的程序获取 TCP 最大分节大小(MSS)的当前值,将发送缓冲区大小设为 65536,然后获取并输出发送缓冲区的大小。如果我们编译并执行这个程序,其输出为:

```
aix % getsetopt
TCP mss = 512
send buffer size = 65536
```

```
 1  #include    "unpxti.h"
 2  int
 3  main(int argc, char **argv)
 4  {
 5      int         fd;
 6      socklen_t optlen;
 7      t_uscalar_t mss, sendbuff;
 8      fd = T_open(XTI_TCP, O_RDWR, NULL);
 9      T_bind(fd, NULL, NULL);
10      optlen = sizeof(mss);
11      Xti_getopt(fd, T_INET_TCP, T_TCP_MAXSEG, &mss, &optlen);
12      printf("TCP mss = %d\n", mss);
13      sendbuff = 65536;
14      Xti_setopt(fd, XTI_GENERIC, XTI_SNDBUF, &sendbuff, sizeof(sendbuff));
15      optlen = sizeof(sendbuff);
16      Xti_getopt(fd, XTI_GENERIC, XTI_SNDBUF, &sendbuff, &optlen);
17      printf("send buffer size = %d\n", sendbuff);
18      exit(0);
19  }
```

图 32.10 使用 xti_getopt 和 xti_setopt 的例子[xtiopt/getsetopt.c]

32.7 小 结

XTI 选项是协商定的,并存在提供者返回的值不同于我们的请求值的可能性。虽然 XTI 选项处理很通用,但最简单的方法是定义两个类似于 getsockopt 和 setsockopt 的基本函数,然后从应用程序中调用它们。

第 33 章　流

33.1　概　述

在介绍 XTI 的附加特性如信号驱动 I/O 和带外数据之前，我们需要了解一下实现的细节。与大多数源自 SVR4 的内核上的终端 I/O 系统一样，XTI 和网络协议一般是用流系统实现的。

本章我们将提供流系统的概貌以及应用程序用来访问流的函数。我们的目的是：了解在流框架下实现网络协议的机制。我们同样要使用 TPI(传输提供者接口)开发一个简单的 TCP 客户程序。TPI 是在基于流的系统上 XTI 和套接口通常使用的传输层接口。其他有关流的信息(包括如何使用流开发内核例程)参见[Rago 1993]。

> 流由 Dennis Ritchie [Ritchie 1984]设计，并于 1986 年随 SVR3 首次得以广泛使用。Posix 未对它进行标准化。Unix98 要求的基本流函数包括：getmsg、getpmsg、putmsg、putpmsg、fattach，以及所有的流 ioctl 命令。XTI 通常用流实现。所有源自系统 V 的系统都应提供流，但各种各样的 4.xBSD 版本并不提供流。
>
> 流有时写成 STREAMS，但它不是一个首字母缩写词，所以我们仅将它写为 streams。
>
> 请注意我们本章将要介绍的流 I/O 系统和“标准 I/O 流”的区别，后者在论及标准 I/O 函数库(例如 fopen、fgets、printf 等等函数)时用到。

33.2　概　貌

流在进程和驱动程序(driver)之间提供一个全双工的连接，如图 33.1。虽然我们将底部方框描绘成驱动程序，但它并不一定与硬件设备相关联；它可以是个伪设备驱动程序(例如软件驱动程序)。

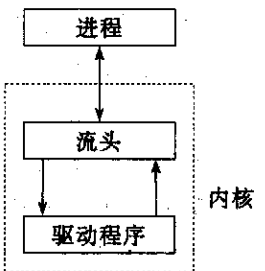


图 33.1　进程和驱动程序之间的一个流

流头(stream head)由一组内核例程构成,这些例程在应用进程对流描述字进行系统调用时(例如 read、putmsg、ioctl,等等)激活执行。

进程可动态地在流头和驱动程序之间加入或删除中间处理模块(module)。这些模块对顺着流上行或下行的消息施行某种类型的过滤。如图 33.2 所示。

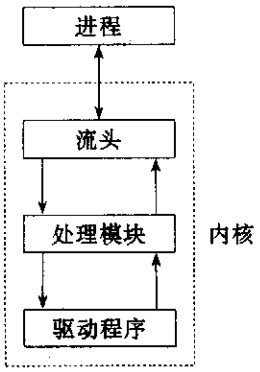


图 33.2 一个有处理模块的流

在一个流中可以压入(push)任意数量的模块。之所以说"压入"是由于每个新模块都插在流头下面的第一个位置。

多路复用器(multiplexor)是一种特殊类型的伪设备驱动程序,它从多个源接受数据。图 33.3 给出了一个基于流的 TCP/IP 协议族的实现,举例来说它可以在 SVR4 上找到。

- 当套接口创建之后,Sockmod 模块被套接口函数库压入流。套接口函数库和 sockmod 流模块共同向进程提供套接口 API。
- 当 XTI 端点创建之后,timod 模块被 XTI 函数库压入流。XTI 函数库和 timod 流模块一起向进程提供 XTI API。
- 我们在 28.12 节曾提到,要使用 read 和 write 来访问 XTI 端点,必须压入 tirdwr 模块。图 33.3 中,中间那个使用 TCP 的进程就做了该动作。该进程这么做也许是放弃了对 XTI 的使用,因此我们去掉了 XTI 函数库。
- 三个服务接口定义了顺着流上行和下行交换的网络消息的格式。TPI(Transport Provider Interface,传输提供者接口)[Unix International 1992b]定义了传输层提供者(例如 TCP 和 UDP)向其上层模块提供的接口。NPI(Network Provider Interface,网络提供者接口)[Unix International 1992a]定义了网络层提供者(例如 IP)向其上层模块提供的接口。DLPI 是数据链路提供者接口(Data Link Provider Interface)[Unix International 1991],[Rago 1993]也提供了对 TPI 和 DLPI 的参考和 C 代码例子。

> 在 Usenet 上经常有人声称"在流环境下,套接口在 TLI(XTI)之上实现"。这是不对的。从图 33.3 可以看出,套接口和 XTI 都在 TPI 上实现。这种声称接着是"因此,TLI(XTI)比套接口要快",这也是不对的。TCP、UDP 和 IP 无论使用 XTI 还是套接口都是一样的,所改变的不过是用户函数库以及流中是否有 timod 或 sockmod 模块而已。不过作者不清楚比较这些库和模块的任何数字。

对于大多数应用程序的瓶颈(数据传送)而言,XTI 和套接口两者的编码路径(code path)也许类似,因此除非一方做特别优化而另一方不做优化,否则很难有大的区别。

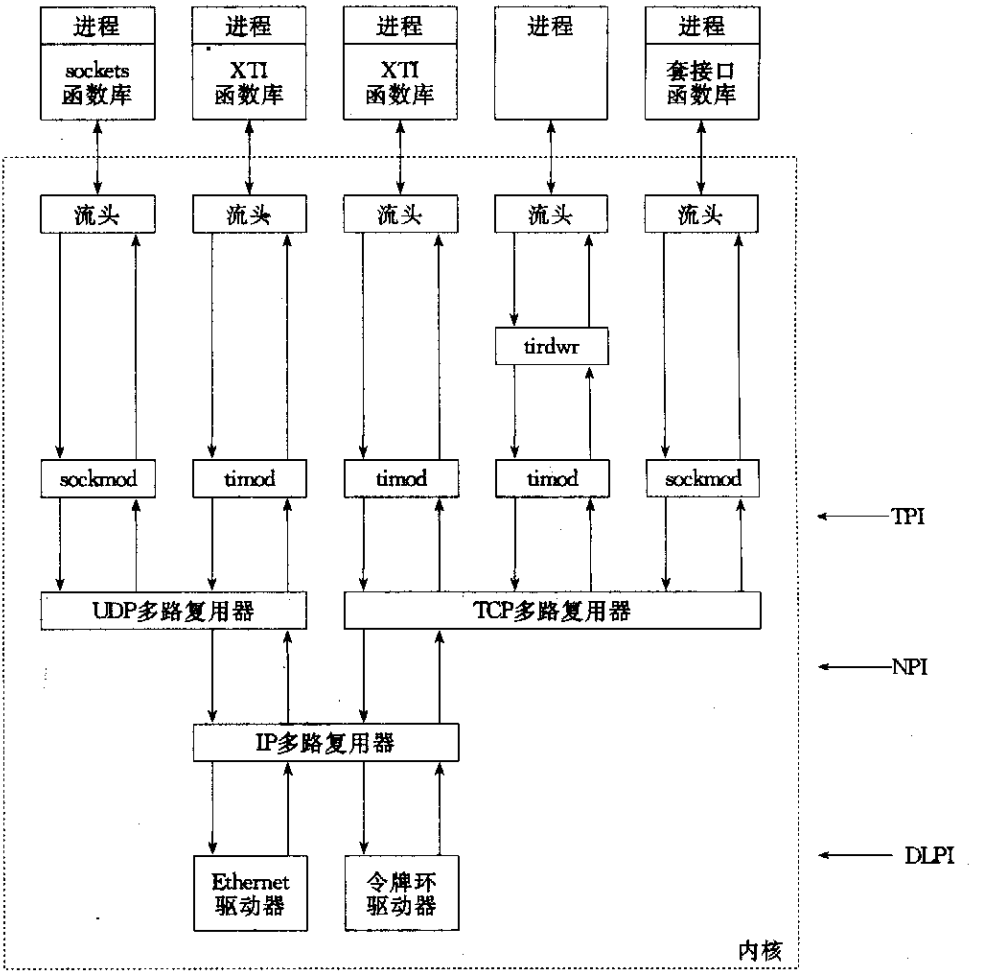


图 33.3　使用流的 TCP/IP 的简化实现

流里的每个部件——流头、所有处理模块及驱动程序——至少包括一对队列(queue):一个写队列和一个读队列。如图 33.4 所示。

消息类型

流消息可分类成:高优先级(high priority)消息、优先级带(priority band)消息和普通消息(normal)三类。共有 256 种不同的优先级带(0～255),普通消息在带 0 中。流消息的优先级用于排队和流控。按约定,高优先级的消息不受流控影响。

图 33.5 给出了一个给定队列中消息的优先级顺序。

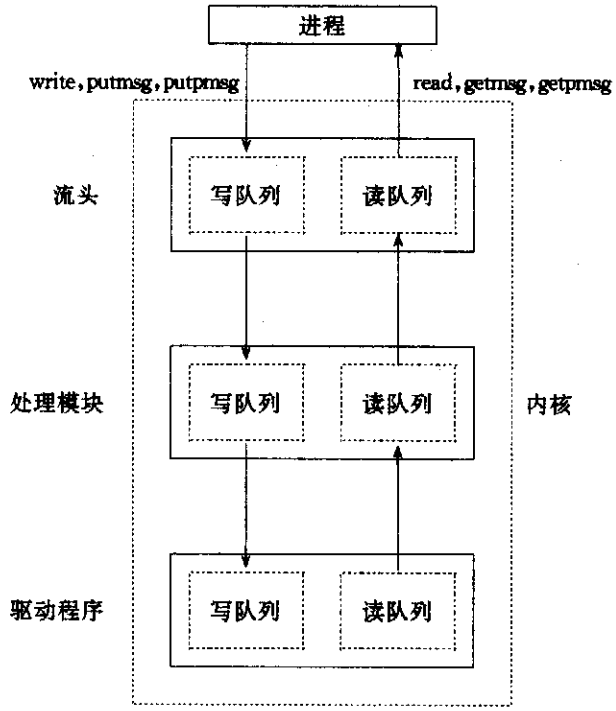


图 33.4 流中每个部件至少有一对队列

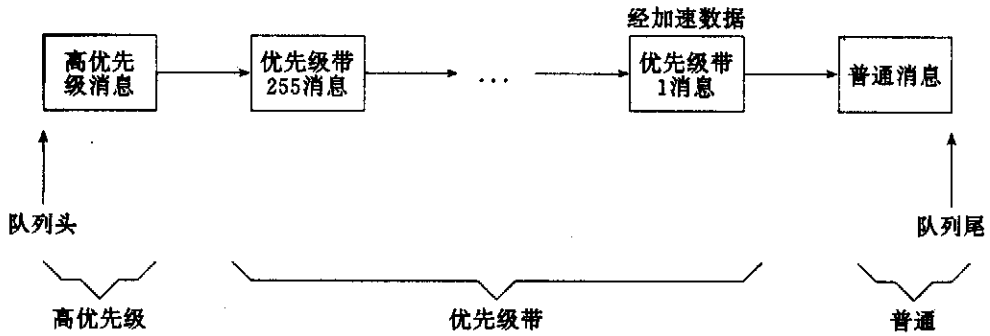


图 33.5 一个队列中的流消息基于优先级的顺序

虽然流系统支持256个不同的优先级带,网络协议一般把带1用于经加速数据,把带0用于普通数据。

TCP带外数据在TPI看来不是真正的经加速数据。实际上TCP对普通数据和带外数据都使用带0(正如我们将在附录的图C.4中验证的一样)。只有那些将经加速数据(并非像TCP中只是一个紧急指针)在普通数据之前传送的协议才利用带1传送加速数据。

注意普通这个词。在SVR4之前没有优先级带的概念,只有普通消息和优先级消息。SVR4实现了优先级带,提供了getpmsg和putpmsg这两个函数(我们很快要讲到)。老的优先级消息换名成高优先级消息。问题是有了优先级带1~255后如何称呼新的消息。通常的方法是将所有非高优先级消息的任何消

息称为普通优先级消息，然后将这些普通优先级消息按某种原则分入各个优先级带[Rago 1993]。普通消息总是指处于带0的消息。

尽管我们只讨论普通优先级消息和高优先级消息两个大类，普通优先级消息和高优先级消息却分别有约12和18种类型。从应用程序的角度以及我们将要讲到的getmsg和putmsg两函数的角度来看，我们只对三种不同类型的消息感兴趣：M_DATA、M_PROTO和M_PCPROTO(PC代表"priority control(优先级控制)"，因此隐指高优先级消息)。图33.6给出了这三种消息类型如何由write和putmsg函数生成。

函数	控制?	数据?	标志	生成消息的类型
write		是		M_DATA
putmsg	不是	是	0	M_DATA
putmsg	是	无关	0	M_PROTO
putmsg	是	无关	MSG_HIPRI	M_PCPROTO

图33.6 由write和putmsg生成的流消息的类型

我们将在下一节描述putmsg函数时解释控制、数据和标志。

33.3 getmsg和putmsg函数

顺着流上传和下传的数据由消息构成，每个消息含有控制、数据或两者都有。如果我们在流上使用read和write，所传送的就只是数据。为了让进程读和写数据和控制信息两部分内容，增加了如下两个函数：

```
#include <stropts.h>
int getmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *flagsp);
int putmsg(int fd, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flags);
                                    二者均返回：非负值——成功，-1——出错
```

消息的控制和数据部分均由strbuf结构描述：

```
struct strbuf {
int maxlen;        /* maximum size of buf */
int len;           /* actual amount of data in buf */
char *buf;         /* data */
};
```

注意strbuf和netbuf的相似之处，结构内的三个成员名也是相同的。

不过netbuf结构的两个长度成员是无符号整数，而strbuf结构的两个长度成员是有符号整数。原因在于有些流函数用len或maxlen的-1值表示特殊意义。

使用putmsg可以单独发送控制信息、数据或者两者都有。为了指示某个部分没有,可以将相应的指针(ctlptr或dataptr)置成空指针,也可以将相应的长度成员(cllptr->len或dataptr->len)设成-1。

如果没有控制信息,putmsg将产生一个M_DATA消息(图33.6);否则产生M_PROTO或M_PCPROTO消息(根据flags的不同)。putmsg的flags参数为0表示普通消息,为RS_HIPRI则表示高优先级消息。

getmsg的最后一个参数是变参。当调用该函数时,如果flagsp指向的整数为0,则流中的第一个消息被返回(可以是普通或高优先级)。如果该整数值为RS_HIPRI,则函数等待一个高优先级消息的到来。这两种情况下,根据返回消息类型的不同,存入flagsp所指整型变量的值为0或RS_HIPRI。

假设我们给getmsg传递了一个非空的ctlptr和dataptr值,如果没有控制信息返回(也就是说返回的是M_DATA消息),函数在返回时设置ctlptr->len为-1。同样,如果没有数据返回,则dataptr->len将被设置为-1。

putmsg返回0表示成功,-1表示出错。而getmsg仅在整个消息都返回时返回0。如果控制缓冲区太小,以至于无法容纳整个控制信息,则返回MORECTL(非负)。同样,如果数据缓冲区太小,则返回MOREDATA。如果两个缓冲区都不够,则返回这两个标志的逻辑或。

33.4 getpmsg和putpmsg函数

当SVR4开始支持多种优先带时,引入了getmsg和putmsg的如下两个变种函数:

```
#include <stropts.h>
int getPmsg(int fd, struct strbuf *ctlptr,
            struct strbuf *dataptr, int *bandp, int *flagsp);
int putPmsg(int fd, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flags);
                                    二者均返回:非负值——成功,-1——出错
```

putpmsg的band参数必须在0~255之间。如果flags参数为MSG_BAND,则将产生一个指定优先带的消息。设置flags为MSG_BAND,并将band设为0,则功能与putmsg相同。如果flags为MSG_HIPRI,则band必须为0,此时产生高优先级消息(注意:putmsg用的是RS_HIPRI标志,这里用的是MSG_HIPRI标志)。

由bandp和flagsp指向的两个整数是getpmsg的变参。flagsp所指的整数可以是MSG_HIPRI(读取高优先消息)、MSG_BAND(读取一个优先级大于等于bandp参数所指整数的消息)和MSG_ANY(读取任意消息)。在返回时,由bandp指向的整数含有读入消息的优先级带,由flagsp指向的整数则变为MSG_HIPRI(如果读入一个高优先消息)或MSG_BAND(读入其他消息)。

33.5　ioctl 函数

为了讨论流，我们又遇到了第 16 章讲过的 ioctl 函数。

```
#include <stropts.h>
int ioctl(int fd, int request, ... /* void *arg */);
                                        返回：0——成功，-1——出错
```

与 16.2 节相比，函数原型的唯一变化是处理流时必须包括的头文件。

大约有 30 个请求影响流头，每个请求都以 I_开头，可以查看 streamio 的手册页面具体了解。我们在图 28.14 给出了 I_PUSH 请求，它是我们在压入 tirdwr 模块到流中时使用的。

当我们在 34.11 节讨论 XTI 的信号驱动 I/O 时，会讨论 I_SETSIG 请求。

33.6　TPI：传输提供者接口

我们在图 33.3 指出，TPI 是传输层向其上层提供的服务接口。套接口和 XTI 在流环境下均使用 TPI。在图 33.3 中，套接口函数库和 sockmod 流模块的组合跟 TCP 和 UDP 交换 TPI 消息，XTI 函数库和 timod 流模块的组合也一样。

TPI 是一个基于消息的接口。它定义了应用进程（例如 XTI 函数库或套接口函数库）和传输层之间顺着流上行和下行交换的消息：消息的格式和每个消息执行的操作。在许多实例中，应用进程向提供者发送一个请求（譬如“捆绑这个本地地址”），而提供者则发回一个响应（“成功”或“出错”）。一些事件在提供者异步地发生（譬如对某个服务器的连接请求的到达），它们导致顺着流向上发送的消息或信号。

我们可以绕过 XTI 和套接口，直接使用 TPI。本节我们将使用 TPI 重新编写我们的简单时间/日期客户程序。用编程语言进行类比，用套接口或 XTI 好比用诸如 C 或 Pascal 等高级语言，而直接用 TPI 好比用汇编语言编程。在实际应用程序中，我们不鼓励直接使用 TPI。不过查看 TPI 如何工作并开发本例子有助于我们更好地理解在流环境下套接口函数库和 XTI 函数库的工作原理。

图 33.7 是我们的 tpi_daytime.h 头文件。

```
1 #include    "unpxti.h"
2 #include    <sys/stream.h>
3 #include    <sys/tihdr.h>
4 void  tpi_bind(int, const void *, size_t);
5 void  tpi_connect(int, const void *, size_t);
6 ssize_t      tpi_read(int, void *, size_t);
7 void  tpi_close(int);
```

图 33.7　我们的 tpi_daytime.h 头文件[streams/tpi_daytime.h]

这里，我们包括一个额外的流头文件<sys/stream.h>以及<sys/tihdr.h>，后者含有全部 TPI 消息的结构定义。

图33.8是时间/日期客户程序的main函数。

```
1 #include    "tpi_daytime.h"
2 int
3 main(int argc, char **argv)
4 {
5     int         fd, n;
6     char        recvline[MAXLINE + 1];
7     struct sockaddr_in myaddr, servaddr;
8     if (argc != 2)
9         err_quit("usage: tpi_daytime <IPaddress>");
10    fd = Open(XTI_TCP, O_RDWR, 0);
11        /* bind any local address */
12    bzero(&myaddr, sizeof(myaddr));
13    myaddr.sin_family = AF_INET;
14    myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15    myaddr.sin_port = htons(0);
16    tpi_bind(fd, &myaddr, sizeof(struct sockaddr_in));
17        /* fill in server's address */
18    bzero(&servaddr, sizeof(servaddr));
19    servaddr.sin_family = AF_INET;
20    servaddr.sin_port   = htons(13);    /* daytime server */
21    inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
22    tpi_connect(fd, &servaddr, sizeof(struct sockaddr_in));
23    for ( ; ; ) {
24        if ( (n = tpi_read(fd, recvline, MAXLINE)) <= 0) {
25            if (n == 0)
26                break;
27            else
28                err_sys("tpi_read error");
29        }
30        recvline[n] = 0;        /* null terminate */
31        fputs(recvline, stdout);
32    }
33    tpi_close(fd);
34    exit(0);
35 }
```

图33.8 TPI时间/日期客户程序的main函数[streams/tpi_daytime.c]

打开传输提供者,捆绑本地地址

第10～16行 我们打开与传输提供者对应的设备(一般为/dev/tcp)。在网际套接口地址结构内填入INADDR_ANY和端口0,通知TCP给端点捆绑任意本地地址。捆绑工作由我们马上给出的函数tpi_bind完成。

填入服务器地址,建立连接

第17～22行 我们在另一个网际套接口地址结构中填入服务器的IP地址(从命令行获取)和端口(13),然后调用tpi_connect建立连接。

从服务器读取数据,拷贝至标准输出

第23～33行 与我们的其他时间/日期客户程序一样,我们简单地将从连接中读取的数据拷贝至标准输出,并在接收到来自服务器的文件结束符(如FIN)时结束。这个循环写得

有点像我们的套接口客户程序(图 1.5),而不像 XTI 客户程序(图 28.13),因为我们的 tpi_read 函数能把来自服务器的顺序释放转化成返回值为 0。我们随后调用 tpi_close 函数关闭端点。

我们的 tpi_bind 函数见图 33.9。

```
1  #include    "tpi_daytime.h"
2  void
3  tpi_bind(int fd, const void *addr, size_t addrlen)
4  {
5      struct {
6        struct T_bind_req    msg_hdr;
7        char     addr[128];
8      } bind_req;
9      struct {
10       struct T_bind_ack    msg_hdr;
11       char     addr[128];
12     } bind_ack;
13     struct strbuf          ctlbuf;
14     struct T_error_ack     *error_ack;
15     int      flags;
16     bind_req.msg_hdr.PRIM_type = T_BIND_REQ;
17     bind_req.msg_hdr.ADDR_length = addrlen;
18     bind_req.msg_hdr.ADDR_offset = sizeof(struct T_bind_req);
19     bind_req.msg_hdr.CONIND_number = 0;
20     memcpy(bind_req.addr, addr, addrlen);      /* sockaddr_in{} */
21     ctlbuf.len = sizeof(struct T_bind_req) + addrlen;
22     ctlbuf.buf = (char *) &bind_req;
23     Putmsg(fd, &ctlbuf, NULL, 0);
24     ctlbuf.maxlen = sizeof(bind_ack);
25     ctlbuf.len = 0;
26     ctlbuf.buf = (char *) &bind_ack;
27     flags = RS_HIPRI;
28     Getmsg(fd, &ctlbuf, NULL, &flags);
29     if (ctlbuf.len < (int) sizeof(long))
30         err_quit("bad length from getmsg");
31     switch(bind_ack.msg_hdr.PRIM_type) {
32     case T_BIND_ACK:
33         return;
34     case T_ERROR_ACK:
35         if (ctlbuf.len < (int) sizeof(struct T_error_ack))
36             err_quit("bad length for T_ERROR_ACK");
37         error_ack = (struct T_error_ack *) &bind_ack.msg_hdr;
38         err_quit("T_ERROR_ACK from bind (%d, %d)",
39                  error_ack->TLI_error, error_ack->UNIX_error);
40     default:
41         err_quit("unexpected message type: %d", bind_ack.msg_hdr.PRIM_type);
42     }
43 }
```

图 33.9　tpi_bind 函数:给端点捆绑一个本地地址[streams/tpi_bind.c]

填充 T_bind_req 结构

第16～20行 头文件<sys/tihdr.h>定义了T_bind_req结构：

```
struct T_bind_req {
long            PRIM_type;      /* T_BIND_REQ */
long            ADDR_length;    /* address length */
long            ADDR_offset;    /* address offset */
unsigned long   CONIND_number;  /* connect indications requested */
        /* followed by the protocol address for bind */
};
```

所有TPI请求都定义成一个以长整数类型成员开头的结构。我们在函数中定义了自己的bind_req结构，该结构以自己的T_bind_req结构作第一个成员，后面跟着一个用于保存待捆绑本地地址的缓冲区。TPI对该缓冲区的内容未做任何规定，它由提供者定义。TCP提供者期望它包含一个sockaddr_in结构。

我们填写T_bind_req结构，将ADDR_length成员置成地址的大小(一个网际套接口地址结构为16字节)，将ADDR_offset置成地址字节偏移量(紧跟在T_bind_req结构之后)。我们不保证该地址能正确对齐，因此我们调用memcpy将调用者传入的地址拷入bind_req结构。由于我们是客户而不是服务器，因此置CONIND_number为0。

调用 putmsg

第21～23行 TPI要求以上结构作为一个M_PROTO消息传给提供者。因此，我们将这个bind_req结构指定成控制信息来调用putmsg，同时指定标志为0，且没有数据。

调用 getmsg 读取高优先级消息

第24～30行 对于T_BIND_REQ请求，响应或者是T_BIND_ACK消息，或者是T_ERROR_ACK消息。这些确认消息以高优先级消息的形式(M_PCPROTO)传送，所以我们使用RS_HIPRI标志调用getmsg。由于应答是高优先级的，这将绕过流上所有普通优先级消息。

下面是这两个消息：

```
struct T_bind_ack {
  long          PRIM_type;      /* T_BIND_ACK */
  long          ADDR_length;    /* address length */
  long          ADDR_offset;    /* address offset */
  unsigned long CONIND_number;  /* connect ind to be queued */
        /* followed by the bound address */
};
struct T_error_ack {
  long          PRIM_type;      /* T_ERROR_ACK */
  long          ERROR_prim      /* primitive in error */
  long          TLI_error;      /* TLI error code */
  long          UNIX_error;     /* UNIX error code */
};
```

所有这些消息都以一个类型字段打头，因此我们可以先假定这是一个T_BIND_ACK消息，再查看其类型，然后相应地处理消息。我们不期望来自提供者的任何数据，因此将

getmsg 的第三个参数设置成空指针。

> 当我们验证所返回的控制信息量至少是一个长整数的大小时，必须小心把 sizeof 的值类型强制转换成一个整数。sizeof 运算符返回的是一个无符号整型，而返回的 len 成员可能是 −1。然而由于小于比较运算符左边是一个有符号值，右边是一个无符号值，因此 C 编译器把有符号值类型转换成无符号值。在补码(twos-complement)体系结构上，−1 作为无符号值非常之大，导致 −1 大于 4(假设长整数占 4 个字节)。

处理应答

第 31～33 行　如果应答是 T_BIND_ACK，则捆绑成功，我们于是返回。端点实际绑定的地址由 bind_ack 结构的 addr 成员返回。

第 34～39 行　如果应答是 T_ERROR_ACK，则先验证整个消息都已收到，然后输出结构中的三个返回值。在我们这个小程序里，如果遇到错误就直接终止，不再返回到调用者。

通过修改我们的 main 函数以捆绑某个非 0 端口，我们就可以看到这种来自捆绑的错误。例如，如果捆绑端口 1(1024 以内端口号要求超级用户特权才能使用)，我们会得到：

```
aix % tpi_daytime 206.62.226.33
T_ERROR_ACK from bind (3, 0)
```

错误 EACCES 在该系统上的值为 3。如果我们选一个 1023 以上的但正被另外一个 TCP 端点使用的端口，我们会得到：

```
aix % tpi_daytime 206.62.226.33
T_ERROR_ACK from bind (23, 0)
```

错误 EADDRBUSY 在该系统上的值为 23。

> 该错误类型是 TPI 为支持 XTI 新引入的。如果请求捆绑的是一个已被使用的端口，支持 TLI 的旧版本的 TPI 就会自动选择另外一个未被使用的端口来捆绑。这意味着当一个服务器捆绑一个众所周知端口时，它必须比较返回的地址(如果 t_bind 的第三个参数是非空指针的话，该地址由 t_bind 在 T_bind_ack 消息中返回)和请求的地址，如果不一致则放弃。

下一个函数是图 33.10 的 tpi_connect，负责建立与服务器的连接。

```
1 #include    "tpi_daytime.h"
2 void
3 tpi_connect(int fd, const void *addr, size_t addrlen)
4 {
5     struct {
6       struct T_conn_req msg_hdr;
7       char      addr[128];
8     } conn_req;
9     struct {
10      struct T_conn_con msg_hdr;
11      char      addr[128];
12    } conn_con;
13    struct strbuf ctlbuf;
14    union T_primitives rcvbuf;
```

```
15  struct T_error_ack  * error_ack;
16  struct T_discon_ind  * discon_ind;
17  int      flags;
18  conn_req.msg_hdr.PRIM_type = T_CONN_REQ;
19  conn_req.msg_hdr.DEST_length = addrlen;
20  conn_req.msg_hdr.DEST_offset = sizeof(struct T_conn_req);
21  conn_req.msg_hdr.OPT_length = 0;
22  conn_req.msg_hdr.OPT_offset = 0;
23  memcpy(conn_req.addr, addr, addrlen);      /* sockaddr_in{} */
24  ctlbuf.len = sizeof(struct T_conn_req) + addrlen;
25  ctlbuf.buf = (char  * ) &conn_req;
26  Putmsg(fd, &ctlbuf, NULL, 0);
27  ctlbuf.maxlen = sizeof(union T_primitives);
28  ctlbuf.len = 0;
29  ctlbuf.buf = (char  * ) &rcvbuf;
30  flags = RS_HIPRI;
31  Getmsg(fd, &ctlbuf, NULL, &flags);
32  if (ctlbuf.len < (int) sizeof(long))
33      err_quit("tpi_connect: bad length from getmsg");
34  switch(rcvbuf.type) {
35  case T_OK_ACK:
36      break;
37  case T_ERROR_ACK:
38      if (ctlbuf.len < (int) sizeof(struct T_error_ack))
39          err_quit("tpi_connect: bad length for T_ERROR_ACK");
40      error_ack = (struct T_error_ack  * ) &rcvbuf;
41      err_quit("tpi_connect: T_ERROR_ACK from conn (%d, %d)",
42              error_ack->TLI_error, error_ack->UNIX_error);
43  default:
44      err_quit("tpi_connect: unexpected message type: %d", rcvbuf.type);
45  }
46  ctlbuf.maxlen = sizeof(conn_con);
47  ctlbuf.len = 0;
48  ctlbuf.buf = (char  * ) &conn_con;
49  flags = 0;
50  Getmsg(fd, &ctlbuf, NULL, &flags);
51  if (ctlbuf.len < (int) sizeof(long))
52      err_quit("tpi_connect2: bad length from getmsg");
53  switch(conn_con.msg_hdr.PRIM_type) {
54  case T_CONN_CON:
55      break;
56  case T_DISCON_IND:
57      if (ctlbuf.len < (int) sizeof(struct T_discon_ind))
58          err_quit("tpi_connect2: bad length for T_DISCON_IND");
59      discon_ind = (struct T_discon_ind  * ) &conn_con.msg_hdr;
60      err_quit("tpi_connect2: T_DISCON_IND from conn (%d)",
61              discon_ind->DISCON_reason);
62  default:
```

```
63          err_quit("tpi_connect2: unexpected message type: %d",
64                   conn_con.msg_hdr.PRIM_type);
65      }
66  }
```

图 33.10　tpi_connect 函数：建立与服务器的连接[streams/tpi_connect.c]

填写请求结构并发送给提供者

第 18～26 行　TPI 定义了一个 T_conn_req 结构，用来放置连接的协议地址和选项：

```
struct T_conn_req {
long    PRIM_type;          /* T_CONN_REQ */
long    DEST_length;        /* destination address length */
long    DEST_offset;        /* destination address offset */
long    OPT_length;         /* options length */
long    OPT_offset;         /* options offset */
        /* followed by the protocol address and options for connection */
};
```

与 tpi_bind 函数一样，我们也定义了自己的 conn_req 结构，它含有 T_conn_req 结构，以及用于存放协议地址的空间。然后填写该结构，设置处理选项的两个成员为 0。最后仅以控制信息调用 putmsg，同时设置的标志为 0，意味着顺着流下行发送的是 M_PROTO 消息。

读入响应

第 27～45 行　调用 getmsg 等待收到 T_OK_ACK 消息(如果连接建立启动)或者 T_ERROR_OK 消息。

```
struct T_ok_ack {
  long    PRIM_type;        /* T_OK_ACK */
  long    CORRECT_prim;     /* correct primitive */
};
```

如果出错则终止。由于不知道收到的是什么类型的消息，因此我们定义一个名为 T_primitives 的所有可能的请求和应答的联合，再(静态)分配这样的一个联合(rcvbuf)，然后在调用 getmsg 时将它用作控制信息的输入缓冲区。

等待连接建立完成

第 46～65 行　T_OK_ACK 消息表示连接已开始建立，现在等待 T_CONN_CON 消息以确信对方已经确认该连接请求。

```
struct T_conn_con {
  long    PRIM_type;        /* T_CONN_CON */
  long    RES_length;       /* responding address length */
  long    RES_offset;       /* responding address offset */
  long    OPT_length;       /* option length */
  long    OPT_offset;       /* option offset */
        /* followed by peer's protocol address and options */
};
```

再次调用 getmsg，不过预期的消息是作为 M_PROTO 消息而不是 M_PCPROTO 消息发送的，因此置标志为 0。如果接收到 T_CONN_CON 消息，则连接建立完毕，函数返回。但是如果连接未能建立(对方进程未运行、超时或其他原因)，就会收到一个 T_DISCON_IND

消息：

```
struct T_discon_ind {
  long    PRIM_type;        /* T_DISCON_IND */
  long    DISCON_reason;    /* disconnect reason */
  long    SEQ_number;       /* sequence number */
};
```

下面，我们看看由提供者返回的各种错误，我们先指定一个未运行时间/日期服务器程序的主机的IP地址：

```
solaris26 % tpi_daytime 140.252.1.4
tpi_connect2: T_DISCON_IND from conn (146)
```

错误146表示ECONNREFUSED。接着我们指定一个未接入因特网的IP地址：

```
solaris26 % tpi_daytime 192.3.4.5
tpi_connect2: T_DISCON_IND from conn (145)
```

错误145表示ETIMEDOUT。再次对该IP地址运行此程序，指定同样的IP地址，得到一个不同的结果：

```
solaris26 % tpi_daytime 192.3.4.5
tpi_connect2: T_DISCON_IND from conn (148)
```

这次错误是EHOSTUNREACH。最后两个错误的区别在于，第一次未返回ICMP主机不可达错，第二次则返回这个错误。

图33.11为我们的另一个TPI函数tpi_read，它从一个流中读取数据。

```
 1 #include    "tpi_daytime.h"
 2 ssize_t
 3 tpi_read(int fd, void *buf, size_t len)
 4 {
 5     struct strbuf   ctlbuf;
 6     struct strbuf   datbuf;
 7     union T_primitives rcvbuf;
 8     int     flags;
 9     ctlbuf.maxlen = sizeof(union T_primitives);
10     ctlbuf.buf = (char *) &rcvbuf;
11     datbuf.maxlen = len;
12     datbuf.buf = buf;
13     datbuf.len = 0;
14     flags = 0;
15     Getmsg(fd, &ctlbuf, &datbuf, &flags);
16     if (ctlbuf.len >= (int) sizeof(long)) {
17         if (rcvbuf.type == T_DATA_IND)
18             return(datbuf.len);
19         else if (rcvbuf.type == T_ORDREL_IND)
20             return(0);
21         else
22             err_quit("tpi_read: unexpected type %d", rcvbuf.type);
23     } else if (ctlbuf.len == -1)
24         return(datbuf.len);
```

```
25      else
26          err_quit("tpi_read: bad length from getmsg");
27  }
```

图 33.11　tpi_read 函数：从流中读取数据[streams/tpi_read.c]

读控制信息和数据，处理应答

第 9～26 行　这次我们调用 getmsg 来同时读取控制信息和数据。返回数据的 strbuf 结构指向调用者的缓冲区。在读取时可能会在流上出现 4 种不同情形：

- 数据作为一个 M_DATA 消息到达，这通过把返回的控制长度置为－1 来指示。数据由 getmsg 拷贝到调用者提供的缓冲区，其长度则作为本函数的返回值返回。
- 数据作为一个 M_DATA_IND 消息到达，这种情况下控制信息是一个 T_data_ind 结构。

```
struct T_data_ind {
  long PRIM_type; /* T_DATA_IND */
  long MORE flag; /* more data */
};
```

  如果该消息被返回，我们将忽略 MORE_flag 成员（对于像 TCP 这样的字节流协议，该成员永远不可能被设置），并返回由 getmsg 拷贝到调用者所提供的缓冲区中的数据的大小。
- 到达一个 T_ORDREL_IND 消息。该消息表示所有数据接收完毕，下一项是一个 FIN。

```
struct T_ordrel_ind {
  long PRIM_type; /* T_ORDREL_IND */
};
```

  这是我们在 28.9 节所描述的顺序释放。我们就返回 0，以向调用者指示已在连接上遇到文件结束符。
- 到达一个 T_DISCON_IND 消息。该消息意味着收到一个断连请求。我们在 28.10 节已经讨论过，在 TCP 中这意味着对于一个已存在连接收到一个 RST。在目前这个简单例子中，我们不处理这种情形，但在图 28.13 中我们确实处理过。

我们现在可以解释 28.12 节中不压入 tirdwr 模块到流中就调用 read 时看到的两种不同情形。第一种情形是产生“read error：Not a data message”错。当时提供者实际上作为一个 M_PROTO 消息（因为它既有控制又有数据）顺着流向上发送了一个 T_DATA_IND 消息，而 read 只能处理 M_DATA 消息，于是出错。

第二种情形是在服务器的预期应答都已收到并输出后再产生“read error：Bad message”错。当时那个系统上的提供者顺着流向上发送了两个消息，第一个是 M_DATA 消息，第二个是 T_ORDREL_IND 消息。结果 read 读取了第一个消息，但对第二个消息无法处理。

最后一个函数是图 33.12 的 tpi_close。

```
1 #include    "tpi_daytime.h"
2 void
3 tpi_close(int fd)
4 {
```

```
 5    struct T_ordrel_req    ordrel_req;
 6    struct strbuf    ctlbuf;
 7    ordrel_req.PRIM_type = T_ORDREL_REQ;
 8    ctlbuf.len = sizeof(struct T_ordrel_req);
 9    ctlbuf.buf = (char *) &ordrel_req;
10    Putmsg(fd, &ctlbuf, NULL, 0);
11    Close(fd);
12 }
```

图33.12 tpi_close函数:向对方发送一个顺序释放[streams/tpi_close.c]

向对方发送顺序释放

第7～10行 我们构造一个T_ordrel_req结构

```
struct T_ordrel_req {
  long PRIM_type;        /* T_ORDREL_REQ */
};
```

并作为一个M_PROTO消息用putmsg发送出去。这相应于XTI的t_sndrel函数。

本例子给我们展示了TPI的风味。应用进程顺着流向下向提供者发送消息(请求),提供者顺着流向上发送消息(应答)。一些消息交换是比较简单的请求应答情形(例如捆绑本地地址),而其他消息交换则需要耗费一定时间(例如建立一个连接),这种情形允许我们在等待应答时做些别的事。我们选择使用TPI编写TCP客户程序而不是服务器程序是为了简单;用它编写服务器程序并像30.7节所述的那样处理连接则要困难得多。

很明显,XTI到TPI的函数映射比较接近,而套接口到TPI的映射就不那么近。不过无论是XTI函数库还是套接口函数库都处理了大量的TPI细节,为我们编写应用程序省了很多事。

我们可以比较一下完成网络操作所需的系统调用次数。TPI相对于在内核实现的套接口,捆绑一个本地地址前者需两次系统调用,而后者只需一次(TCPv2第454页)。建立一个阻塞式描述字上的连接前者需三次系统调用,而后者只需一次(TCPv2第466页)。

33.7 小 结

XTI一般用流来实现。为了访问流子系统,需要使用4个新函数:getmsg、getpmsg、putmsg和putpmsg,当然已有的ioctl函数也用得很多。

TPI是从上层到传输层的SVR4流接口。XTI和套接口均使用TPI(图33.3)。作为一个展示TPI使用的基于消息接口的例子,我们直接使用TPI开发了时间/日期客户程序。

33.8 习 题

33.1 在图33.12中,我们调用putmsg顺着流向下发送一个顺序释放请求,然后立即关闭流。如果在关闭流时,该顺序释放请求被流子系统丢失,会发生什么情况?

第 34 章　XTI:其他函数

34.1　概　述

在以前几章中介绍的 XTI 函数包括:

- TCP 客户
- 主机名和服务名查找
- TCP 服务器
- UDP 客户和服务器
- 选项
- 通常的流实现

本章介绍剩下的 XTI 函数。

34.2　非阻塞 I/O

端点可以设成非阻塞方式。这是通过在调用 t_open 创建端点时指定 O_NONBLOCK 标志,或在创建端点之后调用 fcntl 函数(见图 7.10)来设置的。

当端点为非阻塞方式时,一些 XTI 函数的操作有所改变。

- t_connect 会立即返回 −1,t_errno 设成 TNODATA。在 TCP 上这个调用会启动三路握手,我们必须调用 t_rcvconnect(34.3 节)等待连接建立完成。
- 如果连接正在进展但还未完成,t_rcvconnect 返回 −1,t_errno 设为 TNODATA。
- 当没有已就绪的连接可供调用 t_accept 时,t_listen 立即返回 −1,t_errno 设为 TNODATA。
- 四个接收函数:t_rcv、t_rcvudata、t_rcvv 和 t_rcvvudata,在没有数据时返回 −1,t_errno 设成 TNODATA。如果有数据,即使比应用进程请求的少,也将这些数据返回。(上面提到的最后两个函数是新出现的;在 34.8 节中会对它们进行介绍。)
- 四个发送函数:t_snd、t_sndudata、t_sndv 和 t_sndvudata,在提供者不能接受数据时返回 −1,t_errno 设成 TFLOW。如果提供者可以接受一些数据,那么返回值可能比 t_snd 和 t_sndv 请求的数量要少。两个数据报函数写一个完整的数据报,否则将返回错误。(这四个函数中的后两个是新出现的,在 34.9 节中会对它们进行介绍。)

34.3　t_rcvconnect 函数

在前一节中提到在非阻塞方式下开始一个连接,然后调用 t_rcvconnect 等待连接完成。

```
#include <xti.h>
int t_rcvconnect(int fd,struct t_call *recvcall);
                                                    返回:成功为0,出错为-1
```

使用这个函数的典型步骤如下:

1. 用t_open创建一个端点并设置为非阻塞方式。
2. 用t_connect启动连接的建立。因为端点是在非阻塞方式下,所以这个函数立即返回-1,t_errno设为TNODATA。
3. 在其后的某个时间进程调用t_rcvconnect判断连接是否已完成。如果端点不再是处于非阻塞方式(在第二步的t_connect调用后进程已经关闭了非阻塞标志),t_rcvconnect将阻塞到连接建立为止。如果端点仍处于非阻塞方式,这个t_rcvconnect调用(a)如果连接已建立,则立即返回0,或(b)如果连接还没有建立,则返回-1,t_errno置成TNODATA。

注意在阻塞方式t_connect的(缺省)情况下,提供者在t_connect的第三个参数指向的t_call结构中返回信息。但如果是非阻塞方式的t_connect,这些信息在t_rcvconnect的第二个参数指向的t_call结构中返回。

除非应用进程在t_connect和t_rcvconnect(以上的第二步和第三步)之间将端点从非阻塞方式转换成阻塞方式,否则调用t_rcvconnect判断一个非阻塞的连接是否建立是对时间的浪费,因为应用进程必须以某种循环的形式调用t_rcvconnect,等待连接完成(或返回一个错误)。这被称为轮询(polling)。等待一个非阻塞连接建立完成的更好方法是调用select或poll(第6章),或使用信号驱动的I/O(34.11节)。

回想在15.4节的末尾对连接建立被中断的讨论。在XTI中,如果在一个阻塞方式的端点上的t_connect调用被中断,我们就调用t_rcvconnect等待连接建立完成。

34.4 t_getinfo函数

回想由t_open函数(28.2节)返回的t_info结构。下面的函数给调用者返回同样的信息。

```
#include <xti.h>
int t_getinfo(int fd,struct t_info *info);
                                                    返回:成功为0,出错为-1
```

举个例子,t_alloc调用这个函数以获取一个已打开的访问点的信息,从而得到需要的缓冲区的大小。

34.5 t_getstate函数

每个传输端点都有一个当前状态(current state)。下面的函数给调用者返回当前状态

(一个整数值)。

```
#include <xti.h>
int t_getstate(int fd);
                                    返回：成功时为当前状态，出错返回－1
```

当前状态的值是图 34.1 中的常值之一。图中最后三栏指出对不同的服务类型(图 28.3)哪些状态是有效的。

状态	描述	T_COTS	T_COTS_ORD	T_CLTS
T_DATAXFER	数据传送	•	•	
T_IDLE	已绑定，但空闲	•	•	•
T_INCON	外来的连接在被动端点上待处理	•	•	
T_INREL	外来顺序释放		•	
T_OUTCON	外出连接请求在主动端点上待处理	•	•	
T_OUTREL	外出顺序释放		•	
T_UNBND	未绑定	•	•	•
T_UNINIT	未初始化；开始和结束状态	•	•	•

图 34.1　XTI 端点可能的状态

关于在调用不同的 XTI 函数和发生不同的事件时，传输端点的状态是怎样改变的，可以通过画一个状态转换图来精确地展示。这个图也会给出在不同的状态下哪些函数是可以执行的。举例来说，在 T_UNINIT 状态下只允许调用 t_open 函数，新状态变成 T_UNBND。在 T_UNBND 状态下可以发生 4 种事件：

1. 一个成功返回的 t_close 把状态改变成 T_UNINIT。
2. 可以调用 t_optmgmt 但不会改变状态。(然而这个状态转换图中没能展示的是，对选项的处理可能根据状态而改变。举例来说，对 T_UDP_CHECKSUM 选项的处理在 T_UNBND 状态下和其他状态下是不同的。)
3. 一个成功返回的 t_bind 把状态改变成 T_IDLE。
4. 把一个连接传递到该端点是允许的(用 t_accept)，状态改变成 T_DATAXFER。

一旦通过了前两个状态，状态图变得十分复杂难于描绘，因此我们就不再进一步给出了。

34.6　t_sync 函数

过去 TLI 在 SVR3 中是以函数库的方式实现的。考虑一个如图 34.2 所示的调用 exec 并使用 TLI 的程序。左边的程序可能是一个监听的服务器，等待外来的连接，接收到连接后 exec 右边的程序处理客户的请求。(回忆一下，进程 ID 在 exec 后不变，但调用者的内存被新程序所替代，接着执行新程序中的 main 函数。)

这种情况下在 SVR3 中遇到的一个问题是，在进程中的 TLI 函数库和内核中的提供者都维护状态信息。在 exec 后，库中的所有状态信息都被丢弃，新程序中库的状态从头开始

(转换)。t_sync函数的目的是允许新程序(在图34.2的右边)将库的状态和内核中的提供者的状态同步。

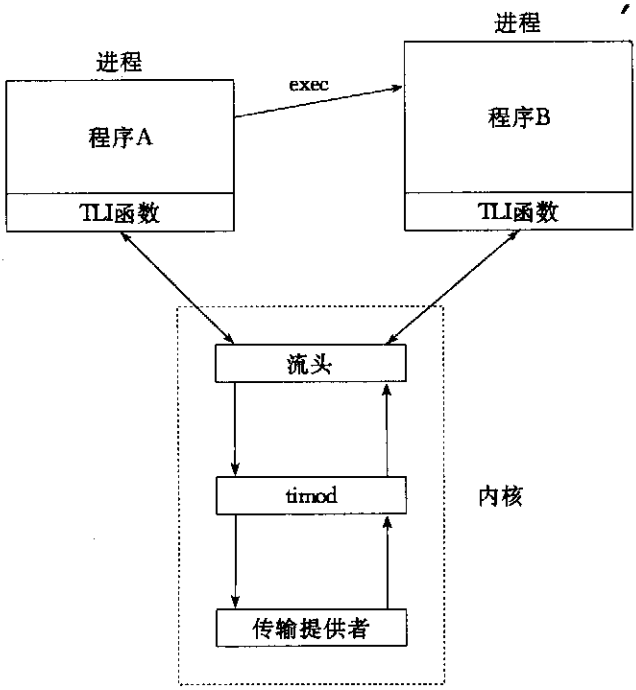


图34.2 进程调用exec时TLI的实现

然而在SVR4中,给如图34.2所示的情况调用t_sync的需要消失了。TLI库函数能自己检测是否需要执行同步。举例来说,如果库函数中有一个变量声明为:

```
static    int    synced;    /* initialized to 0 when program starts */
```

那么每个函数可以用类似于以下的语句来开头:

```
int
t_connect(fd, ... )
{
    if(synced == 0)
        t_sync(fd);    /* also sets synced = 1 */
    ...
}
```

虽然这样处理了进程调用exec的情况,但是仍有一种情况需要t_sync,尽管很少:当多个进程共享一个XTI端点时。在这种情况下,要求这些进程相互合作,而且每个进程在它认为需要时调用t_sync(当然,这依赖于应用程序的具体情况)。一个可能需要执行t_sync的例子是一对父子关系的进程,如果父进程调用t_listen,然后子进程调用t_accept;那么父进程需要调用t_sync更新在它库中的那一份端点状态的拷贝,因为这个状态在子进程调用t_accept后可能已改变了。

```
#include <xti.h>
int t_sync(int fd);
                                返回:如果成功返回当前状态,出错时为-1
```

这个函数成功时返回图 34.1 中的状态之一。

在套接口 API 中没有和这个函数类似的操作。

34.7　t_unbind 函数

t_unbind 函数取消 t_bind 函数的影响。

```
#include <xti.h>
int t_unbind(int fd);
                                返回:成功返回 0,出错时为-1
```

这个函数禁止由 fd 指定的传输端点。该端点于是不能再接受数据。可以调用 t_bind 为该端点捆绑另一个本地地址。

对于套接口来说,这样的操作只能在已连接 UDP 套接口上通过调用 bind 捆绑一个无效地址来完成。

34.8　t_rcvv 和 t_rcvvudata 函数

这两个函数扩展了 t_rcv 和 t_rcvudata 函数的功能,可对一组缓冲区(即一个缓冲区向量)进行操作,而不只是单个缓冲区。它们提供了分散读(scatter read)的能力。

这两个函数和下一节中介绍的两个函数是由 Posix.1g 引入的。

操作一组缓冲区的概念源于 readv 和 writev 函数以及 recvmsg 和 sendmsg 函数。

```
#include <xti.h>
int t_rcvv(int fd,struct t_iovec *iov,unsigned int iovcnt,int *flags);
int t_rcvvudata (int fd,struct t_unitdata *unitdata,
                 struct t_iovec *iov,unsigned int iovcnt,int *flags);
                                返回:如果成功返回读或写的字节数,出错返回-1
```

这两个函数的 iov 参数都是一个指向 t_iovec 结构数组的指针。

```
struct t_iovec {
  void        *iov_base;    /* starting address of buffer */
```

```
    size_t      iov_len;        /* length of buffer in bytes */
}
```

在这个数组中的项数由 iovcnt 参数指明。常值 T_IOV_MAX 给出了这个数组的项数的上限，它在<xti.h>头文件中定义，其值至少为 16。

这些新函数和以前对应的函数相比，可以看出以下几点：

- 缓冲区指针及其长度是 t_rcv 的中间两个参数。
- t_rcvv 函数使用的 t_iovec 结构数组中包含缓冲区指针和它们的长度，这个函数的中间两个参数指向这个结构数组并指明这个数组中的项数。
- t_rcvudata 使用的 t_unitdata 结构的 udata 成员中包含缓冲区指针及其长度。这个函数调用成功时返回 0，而且 t_unitdata 结构的 udata.len 成员中是收到的数据报的实际长度。
- t_rcvvudata 使用的 t_iovec 结构数组中包含缓冲区指针和它们的长度，这个函数的第三个参数是一个指向这个结构数组的指针，第四个参数是该数组的项数。第二个参数是一个指向 t_unitdata 结构的指针，addr 和 opt 成员仍被使用(用于获取发送者的协议地址和任何接收到的选项)，但 udata 成员被忽略。这个函数返回的是数据报中的字节数，而不是 0。

34.9 t_sndv 和 t_sndvudata 函数

这两个函数是对 t_snd 和 t_sndudata 的扩展，能处理一组缓冲区(即一个缓冲区向量)，而不只是单个缓冲区。它们是和前一节中两个函数对应的发送函数，提供集中写(gather write)的能力。

```
#include <xti.h>
int t_sndv(int fd,struct t_iovec *iov,unsigned int iovcnt,int flags);
                                        返回：如果成功返回读或写的字节数，出错返回-1
int t_sndvudata(int fd,struct t_unitdata *unitdata,
                struct t_iovec *iov,unsigned int iovcnt);
                                                    返回：成功为0，出错为-1
```

两个函数的 iov 参数都是一个指向 t_iovec 结构数组的指针，该结构的定义曾在前一节中给出过。iovcnt 参数指明数组中的项数。

输出缓冲区由 t_sndv 的中间两个参数指定，以代替 t_snd 的中间两个参数。对于数据报函数，t_sndudata 的输出缓冲区由 t_unitdata 结构的 udata 成员指定，而 t_sndvudata 是由 iov 向量指定。t_sndvudata 忽略 t_unitdata 结构中的 udata 成员。

34.10 t_rcvreldata 和 t_sndreldata 函数

如果用 t_sndrel(28.9 节)进行顺序释放，则不能跟顺序释放指示一起发送数据(该函数

唯一的参数是一个描述字),但如果用 t_snddis(28.10 节)发送断连是可以一起发送数据的(用 t_call 结构中的 udata 成员)。t_rcvrel 和 t_rcvdis 相比有着同样的限制。为了消除这个限制,XTI 增加了两个新函数,以在顺序释放时发送和接收数据。

```
#include <xti.h>
int t_sndreldata(int fd,const struct t_discon *discon);
int t_rcvreldata(int fd,struct t_discon *discon);
                                        两者均返回:成功为 0,出错为-1
```

这两个函数和 t_sndrel 及 t_rcvrel 的不同在于增加了第二个参数(一个指向 t_discon 结构的指针)。

这些函数只是在提供者支持随顺序释放一起发送数据时才有用,这由 t_info 结构(图 28.4)的 flag 成员中的 T_ORDRELDATA 标志来指明。如果支持,顺序释放能发送的数据量受限于 t_info 结构的 discon 成员的值。

TCP 不支持这种可选的功能。

34.11　信号驱动 I/O

信号驱动 I/O 不是由 XTI 而是由流系统提供的。所用信号的名字是 SIGPOLL,它并不会只因进程为它安装了信号处理程序而被递交。进程还必须发出 I_SETSIG 流 ioctl 请求来通知内核它想接收这个信号,请求本身则指定产生该信号的条件。这和前面在套接口 API 中介绍的为接收 SIGIO 和 SIGURG 信号要做的工作类似。

ioctl 的第三个参数是一个整数,用来指定 SIGPOLL 信号产生的条件。如果这个值为 0,进程将不再为这个流接收 SIGPOLL 信号。这个值也可由下面的这些常值的逻辑或来组成:

S_BANDURG	如果这个标志和 S_RDBAND 一起指定,当有一个优先级带大于 0 的数据可读时,将产生 SIGURG 信号代替 SIGPOLL 信号。
S_ERROR	流出错。
S_HANGUP	挂断消息到达流头。
S_HIPRI	有一个高优先级的消息可读。
S_INPUT	等价于 S_RDNORM | S_RDBAND,这意味着任意优先级带(包括 0)的消息都可读。
S_OUTPUT	就在流头下面的写队列上的普通消息(优先级带 0)不再受流控。
S_MSG	在流的读队列的前面有一个流信号消息。
S_RDNORM	有一个普通消息(优先级带 0)可读。
S_RDBAND	有一个优先级带大于 0 的消息可读。
S_WRNORM	等价于 S_OUTPUT。

S_WRBAND 优先级带大于0的消息在写队列上不再受流控。

S_BANDURG标志在用流实现的套接口API中也能使用。

对于S_HIPRI没有跟S_INPUT相对应的输出条件(即S_WRNORM和S_WRBAND)。这是因为putmsg和putpmsg在发送高优先级的消息时不会阻塞;这些消息不受流控。

流信号SIGPOLL既用于信号驱动I/O,又用于带外数据到来的通知。这对应于在套接口API中介绍的SIGIO和SIGURG两个信号。

SIGPOLL信号的缺省行为是终止进程,因此在使用这个信号时我们必须先建立信号处理程序,然后调用ioctl打开这个信号。

我们刚说的SIGPOLL的缺省行为即终止进程和SIGIO有矛盾。Posix.1g规定SIGIO的缺省行为是忽略。因为SVR4系统把这两个信号定义成同样的,这些系统将不得不将SIGPOLL的缺省行为改成忽略才算符合Posix.1g标准。

尽管SIGPOLL信号可以因很多条件而产生,不处理带外数据的典型应用程序只对S_RDNORM和S_WRNORM感兴趣。

34.12 带外数据

带外数据被XTI称为经加速数据(expedited data)。对这种特性的支持是由传输提供者和流系统提供的。在第33章中我们提到带外数据一般是作为优先级带1中的普通优先级数据实现的。普通数据(也就是非带外数据)则在优先级带0。

在第33章中我们还提到因为TCP的带外数据不是真正的经加速数据(从XTI的观点),所以它实际是在带0而不是带1实现的。

我们在第21章中关于TCP对带外数据的支持,以及TCP的紧急模式到带外数据的映射的所有讨论,对XTI都适用。

XTI的t_snd函数通过把flags参数设置为T_EXPEDITED来发送带外数据。t_rcv函数也会把这个标志返回给调用者。

在编写一个处理带外数据的应用程序时不能使用read和write函数(回想28.12节中的xti_rdwr函数)。我们必须使用t_snd和t_rcv。

因为TCP的带外数据对应的是优先级带0中的普通消息,为了在带外数据到来时收到SIGPOLL,需要在调用ioctl发I_SETSIG请求(34.11节)时设置S_RDNORM。因为这会使普通数据到来时也产生SIGPOLL信号,如果我们想要区分普通数据和带外数据,就必须在信号处理程序中调用t_look检查是T_DATA还是T_EXDATA(图28.9)。XTI在收到一个带有紧急指针的TCP分节时设置T_EXDATA事件,而且这个事件将保持到直到该紧急指针所指位置之前的所有数据(包括紧急数据本身)都收到为止。

SIGPOLL对应于套接口API的SIGURG信号。

如果用 poll 等待带外数据(6.10 节)，pollfd 结构的 events 成员必须设置成 POLLRDNORM，因为带外数据作为优先级带 0 中的普通数据呈现。我们将在图 34.4 和 C.5 中验证 TCP 的带外数据在 poll 时呈现为普通数据。还要注意的是 XTI 对带外数据的处置和套接口 API 不同，套接口 API 认为带外数据属于优先级带。

> 使用 poll 对应于调用 select 等待异常条件，但 poll 不能告诉我们到来的数据的类型，因此我们必须调用 t_look 和 t_rcv。
>
> 套接口 API 在缺省情况下从普通的数据流中移走接收到的带外数据字节，将它放到特别的单个字节的缓冲区中，应用程序用带 MSG_OOB 标志的 recv 来读它。这跟 XTI 对带外数据的处理毫无相似之处：TCP 的带外数据总是在线接收的，而这一点对于套接口来说需打开 SO_OOBINLINE 套接口选项。

现在看几个例子，看看信号驱动 I/O 和 poll 是怎样在 XTI 的带外数据上工作的。

使用 SIGPOLL 的例子

图 34.3 是一个使用 SIGPOLL 报告在 XTI 端点上何时有数据到来的程序。

```
 1 #include    "unpxti.h"
 2 #define NREAD     100
 3 int     listenfd, connfd;
 4 void    sig_poll(int);
 5 int
 6 main(int argc, char * * argv)
 7 {
 8     int     n, flags;
 9     char    buff[NREAD+1];     /* +1 for null at end */
10     if (argc == 2)
11         listenfd = Tcp_listen(NULL, argv[1], NULL);
12     else if (argc == 3)
13         listenfd = Tcp_listen(argv[1], argv[2], NULL);
14     else
15         err_quit("usage: tcprecv01 [ <host> ] <port#>");
16     connfd = Xti_accept(listenfd, NULL, NULL);
17     Signal(SIGPOLL, sig_poll);
18     Ioctl(connfd, I_SETSIG, S_RDNORM);
19     for ( ; ; ) {
20         flags = 0;
21         if ( (n = t_rcv(connfd, buff, NREAD, &flags)) < 0) {
22             if (t_errno == TLOOK) {
23                 if ( (n = T_look(connfd)) == T_ORDREL) {
24                     printf("received T_ORDREL\n");
25                     exit(0);
26                 } else
27                     err_quit("unexpected event after t_rcv: %d", n);
28             }
29             err_xti("t_rcv error");
```

```
30          }
31          buff[n] = 0;      /* null terminate */
32          printf("read %d bytes: %s, flags = %s\n",
33                  n, buff, Xti_flags_str(flags));
34      }
35  }
36  void
37  sig_poll(int signo)
38  {
39      printf("SIGPOLL received, event = %s\n", Xti_tlook_str(connfd));
40  }
```

图34.3 在XTI端点上用SIGPOLL接收普通和带外数据[xtioob/tcprecv01.c]

创建监听端点并等待连接

第10～16行 调用tcp_listen函数创建一个监听端点,然后用xti_accept函数等待并接受连接。

建立信号处理程序

第17～18行 调用signal为SIGPOLL建立一个信号处理程序,然后调用ioctl使得在这个端点上有普通数据到来时产生该信号。

循环,读数据

第19～34行 调用t_rcv接收数据,当对方关闭连接时处理顺序释放。输出收到的数据字节以及t_rcv返回的flags。我们的xti_flags_str函数返回一个描述对作为参数传入的标志的消息的指针。

信号处理程序

第36～40行 这个信号处理程序只是输出一个包含端点当前事件的消息。我们的xti_tlook_str函数调用t_look并返回一个描述端点当前事件的消息的指针。

启动这个程序,然后运行图21.3中的程序作为客户。下面是服务器的输出:

```
unixware % tcprecv01 9999
read 3 bytes: 123, flags = 0
SIGPOLL received,event = T_EXDATA
read 1 bytes: 4, flags = T_EXPEDITED
SIGPOLL received,event = T_DATA
read 2 bytes: 56, flags = 0
SIGPOLL received,event = T_EXDATA
read 1 bytes: 7, flags = T_EXPEDITED
SIGPOLL received,event = T_DATA
read 2 bytes: 89, flags = 0
SIGPOLL received,event = T_ORDREL
received T_ORDREL
```

收到的前三个字节是普通数据,但没有产生SIGPOLL。这是因时序问题引起的。回想图2.5,因为TCP的三路握手操作,客户的connect(如果是XTI则为t_connect)在服务器的accept(或t_accept)返回前二分之一个RTT时返回。这使得客户能抢先发出它的第一个分节,在这个例子中我们看到在信号处理程序建立之间有三个字节到来。

在这之后收到 SIGPOLL,事件为 T_EXDATA。t_rcv 返回的标志为 T_EXPEDITED。从其余的输出行中可以看到每收到一个 TCP 分节就产生一个 SIGPOLL,不过我们必须调用 t_look 查看发生的是什么事件。

当读完所有的数据并得到客户关闭连接的通知时,SIGPOLL 信号如期产生,事件为 T_ORDREL。

使用 poll 的例子

下一个例子如图 34.4 所示,使用 poll 函数。

```
1 #include    "unpxti.h"
2 #define NREAD     100
3 int      listenfd, connfd;
4 int
5 main(int argc, char * * argv)
6 {
7     int      n, flags;
8     char     buff[NREAD+1];          /* +1 for null at end */
9     struct pollfd      pollfd[1];
10    if (argc == 2)
11        listenfd = Tcp_listen(NULL, argv[1], NULL);
12    else if (argc == 3)
13        listenfd = Tcp_listen(argv[1], argv[2], NULL);
14    else
15        err_quit("usage: tcprecv03 [ <host> ] <port#>");
16    connfd = Xti_accept(listenfd, NULL, NULL);
17    pollfd[0].fd = connfd;
18    pollfd[0].events = POLLIN;
19    for ( ; ; ) {
20        Poll(pollfd, 1, INFTIM);
21        printf("revents = %x\n", pollfd[0].revents);
22        if (pollfd[0].revents & POLLIN) {
23            flags = 0;
24            if ( (n = t_rcv(connfd, buff, NREAD, &flags)) < 0) {
25                if (t_errno == TLOOK) {
26                    if ( (n = T_look(connfd)) == T_ORDREL) {
27                        printf("received T_ORDREL\n");
28                        exit(0);
29                    } else
30                        err_quit("unexpected event after t_rcv: %d", n);
31                }
32                err_xti("t_rcv error");
33            }
34            buff[n] = 0;      /* null terminate */
35            printf("read %d bytes: %s, flags = %s\n",
36                    n, buff, Xti_flags_str(flags));
37        }
38    }
39 }
```

图 34.4　在 XTI 端点上用 poll 接收普通和带外数据[xtioob/tcprecv03.c]

等待客户的连接

第10～16行创建监听端点并接收客户的连接。这和图34.3一样。

调用poll前的准备

第17～18行分配一个只有一个元素的pollfd数组并对它进行初始化,以期在普通或优先级数据到达我们的已连接端点时通知我们。

调用poll

第19～38行在一个无限循环中调用poll。当它返回时,我们输出pollfd结构的revents成员,以便查看到来的数据的类型。如果事件为POLLIN,我们就调用t_rcv读数据并输出这些数据和返回的标志。

运行这个程序,用图21.3中的程序作客户(和前一个例子的客户相同)。

```
unixware %tcprecv03 7777
revents = 1
read 3 bytes: 123, flags = 0
revents = 1
read 1 bytes: 4, flags = T_EXPEDITED
revents = 1
read 2 bytes: 56, flags = 0
revents = 1
read 1 bytes: 7, flags = T_EXPEDITED
revents = 1
read 2 bytes: 89, flags = 0
revents = 1
received T_ORDREL
```

每次poll返回时,事件的值是1,它在这个系统中为POLLIN。t_rcv通过返回的标志告诉我们返回的数据类型。

34.13 回馈传输提供者

多数XTI的实现都提供一个回馈传输提供者。在netconfig文件中三种类型的XTI回馈传输提供者的名字通常为ticlts、ticots和ticotsord(图28.3)。ti前缀代表"传输独立"。这三个名字也是用于t_open的/dev下的文件名。

在基于流的系统上Unix域套接口一般是用这三个提供者中的两个来实现的:ticlts用于SOCK_DGRAM套接口,ticots和ticotsord哪一个在netconfig文件中先出现,哪一个就用于SOCK_STREAM套接口。

要注意的是,当在XTI中直接使用这些提供者时,所用的地址被称为伸缩地址(flex address),它们只是一些一个或多个字节的任意字符串。这些地址不是以空字符结尾的;它们的长度由包含这个地址的netbuf结构中的len成员指明。但t_info结构的addr成员可能会返回成-1(T_INFINITE),导致t_alloc不给这个地址分配缓冲区。

> TLI和XTI的一个差别是t_alloc对T_INFINITE的处理。缺省时,TLI将分配一个1024字节的缓冲区,而XTI不会分配缓冲区。

34.14　小　结

在一个 XTI 端点上非阻塞 I/O 既可以通过在 t_open 调用时指定 O_NONBLOCK 打开，也可以在以后任何时候使用 ioctl 打开。当一个端点被设置为非阻塞方式时，XTI 的变化和套接口的变化类似，不过非阻塞方式 t_connect 例外。我们可以用 t_rcvconnect 函数等待它的完成。

XTI 定义了四个新的 I/O 函数以对一组缓冲区（即缓冲区向量）进行操作：t_rcvv、t_rcvvudata、t_sndv 和 t_sndvudata。

XTI 端点的信号驱动 I/O 是通过 ioctl 打开的，并且还要设置所有的产生信号的条件：某个 S_xxx 条件。带外数据是用设置了 T_EXPEDITED 标志的 t_snd 发送的。当接收带外数据时不需要做其他工作：t_rcv 会返回一个 T_EXPEDITED 标志。当带外数据到来时也可以产生一个信号。

第5部分 附 录

附录A IPv4、IPv6、ICMPv4和ICMPv6

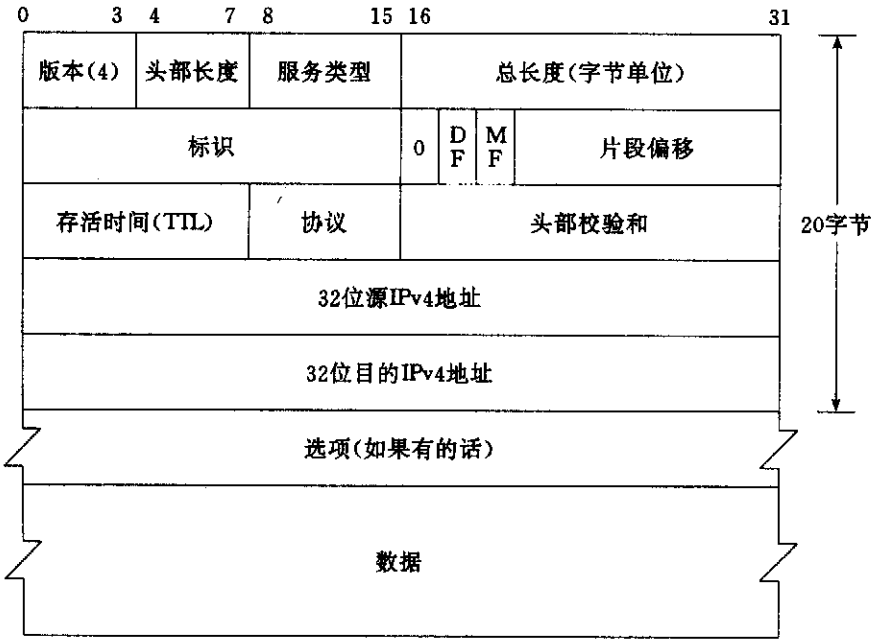
A.1 概 述

本附录给出IPv4、IPv6、ICMPv4及ICMPv6的概貌。这些材料所提供的额外背景知识对于理解第2章中有关TCP和UDP的讨论会有帮助。本书一些高级内容的章节也用上了IP和ICMP的某些特性：例如IP选项(第24章)以及Ping和Traceroute程序(第25章)。

A.2 IPv4头部

IP层提供无连接不可靠的数据报递送服务(RFC 791[Postel 1981a])。它尽力把IP数据报递送到指定的目的地，但并不保证是否到达。任何期望的可靠性必须由更高的层次提供。对TCP应用程序来说，这是由TCP本身完成的。对UDP应用程序来说，这得由应用程序本身完成，因为UPD不可靠，我们在20.5节给出了这样的一个例子。

IP层最重要的功能之一是路由(routing)。每个IP数据报都包含源地址和目的地址。图A.1给出了IPv6数据报头部的格式。



图A.1 IPv4头部格式

- 4 位的版本(version)值为 4。这是自 80 年代早期以来一直使用的 IP 版本。
- 头部长度(head length)是整个 IP 头部的长度,包括任何选项,以 32 位字为单位。这个仅有 4 位的字段能表示的最大值为 15,对应的最大 IP 头部长度为 60 字节。除头部的固定部分占据的 20 字节外,它最多允许 40 字节的选项。
- 8 位的服务类型(TOS)是由 3 位优先级字段(忽略)、指定服务类型的 4 个位及必须为 0 的 1 个不用位构成的。我们可以使用 IP_TOS 套接口选项来设置该字段(图 7.12)。
- 16 位的总长度(total length)是包括头部在内的以字节为单位的 IP 数据报总长度,数据报中数据长度就是这个字段减掉 4 乘以头部长度。本字段是必需的,因为有些数据链路需要把帧垫补成某个最小长度(例如以太网),而有效 IP 数据报的大小小于数据链路最小长度是有可能的。
- 16 位标识(identification)给每个 IP 数据报设置不同的值,用于分片和重组(2.9 节)。
- DF 位(不要分片)、MF 位(还有片段)和 13 位的片段偏移(fragment offset)也用于分片和重组。
- 8 位的存活时间(TTL)由数据报的发送者设置并由转发它的每个路由器减 1。当任何路由器把 TTL 的值减为 0 时数据报就丢弃。TTL 把任何 IP 数据报的生命期限定成最大 255 跳。它的常用缺省值为 64,但我们可以使用 IP_TTL 和 IP_MULTICAST_TTL 套接口选项(7.6 节)查询和修改这个缺省值。
- 8 位的协议(protocol)指定包含在 IP 数据报中的数据类型。它的典型值有 1(ICMPv4)、2(IGMPv4)、6(TCP)和 17(UDP)。这些值在 RFC 1700[Reynolds and Postel 1994]中指定。
- 16 位头部检验和(head checksum)只对 IP 头部(包括任何选项)计算。其算法是标准的因特网校验和算法,即简单的 16 位反码加法(16-bit ones-complement addition),具体参见图 25.14。
- 源 IPv4 地址(source IPv4 address)和目的 IPv4 地址(destination IPv4 address)都是 32 位字段。
- 选项(options)我们是在 24.2 节叙述的,在 24.3 节我们给出了 IPv4 源路径选项的例子。

A.3 IPv6 头部

图 A.2 给出了 IPv6 头部的格式(RFC 1883 [Deering and Hinden 1995])。

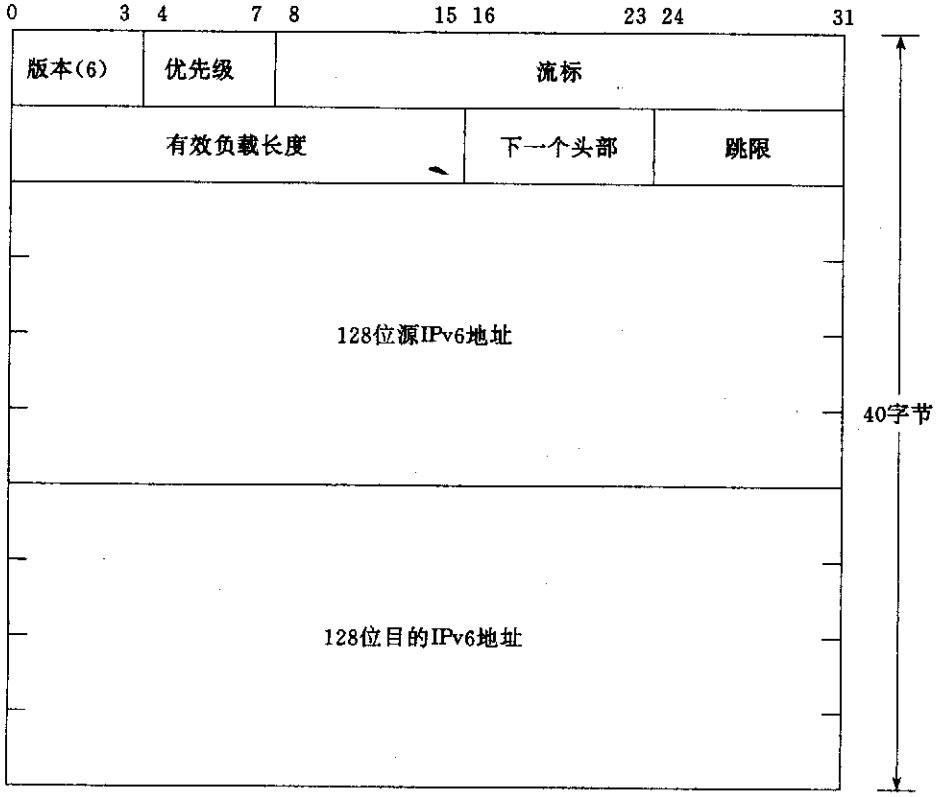


图 A.2 IPv6 头部格式

- 4 位的版本(version)值为 6。由于本字段占据头部第 1 个字节的前 4 位(与图 A.1 给出的 IPv4 版本类似),因此它允许支持这两个版本的接收方 IP 协议栈区分开它们。90 年代初期开发 IPv6 时, 在赋予 6 这个版本号之前, 该协议称为 IPng,代表"下一代 IP"。你可能还会碰到 IPng 的称谓。
- 4 位优先级(priority)是由发送者设定的。

  本字段的有用性仍是一个研究课题。RFC 1883[Deering and Hinden 1995]给这个字段定义了两组值:值 0 到 7 标识在拥塞时排队等待通过的数据单元(例如 TCP 数据),值 8 到 15 则标识即使拥塞也不排队等候的数据单元(例如以恒定速率发送的实时分组)。从 1997 年年中起,对本字段的提议是这 4 个位对接收者无关紧要,但发送者可通过设置最低位表示数据流是"交互式的"(也就是说延迟比吞吐量更为重要)。这 4 个位也可以由路由器因专用目的而改写。

- 24 位流标(flow label)可由应用程序为给定的套接口随意选择。(本字段的使用尚在试验之中。)所谓的流(flow)是从特定源头到特定目的地的分组序列,而且其源头期望中间的路由器对这些分组进行特殊处理。对于给定的流,一旦流标由其源头选定就不再改变。值为 0 的流标(缺省设置)标识并不属于某个流的分组。

  优先级和流标这两个字段合称为流信息(flow information)。它们都包含在 sockaddr_in6 套接口地址结构的 sin6_flowinfo 成员中(图 3.4)。

- 16 位有效负载长度(payload length)是 40 字节 IPv6 头部之后以字节为单位的所有内容的长度。值为 0 表示长度超过 16 位,从而包含在某个特大有效负载选项(图 24.9)中。这样的数据报称为特大报(jumbogram)。
- 8 位下一个头部(next header)类似于 IPv4 的协议字段。实际上,当上层基本上不变时,所用的值也相同,例如 6 代表 TCP,17 代表 UDP。从 ICMPv4 到 ICMPv6 的变动比较多,因此赋给后者的是新值 58。
- 8 位的跳限(hop limit)类似于 IPv4 的 TTL 字段。它的值由转发给定分组的每个路由器减 1,当某个路由器把它减成 0 时,该分组被丢弃。本字段的缺省值可使用 IPV6_UNICAST_HOPS 和 IPV6_MULTICAST_HOPS 套接口选项设置与获取(7.8 节)。IPV6_HOPLIMIT 套接口选项也可用来设置本字段或从接收到的数据报中取得其值。
- 源 IPv6 地址(source IPv6 address)和目的 IPv6 地址(destination IPv6 address)都是 128 位的字段。

IPv6 数据报在其 40 字节的 IPv6 头部之后可以跟多个头部。这就是称为"下一个头部"的字段不叫做"协议"字段的原因。IPv4 的数据报在其 IPv4 头部之后只能有一个协议头部。

IPv4 的早期规范要求路由器从 TTL 中减掉的值或者为 1,或者是存储相应分组的秒数,这得看哪个值比较大。因此采用"存活时间"的名称。然而实际使用中该字段总是减 1 处理。IPv6 要求它的跳限字段总是减 1,因而换了个不同于 IPv4 的名称。

从 IPv4 到 IPv6 的最显著变化自然是 IPv6 采用更大的地址字段。另一个变化是简化 IPv6 头部,因为头部越简单,路由器处理起来也更快。这两种头部之间的变化还有:

- IPv6 没有头部长度字段,因为头部不再有选项。固定的 40 字节 IPv6 头部之后可跟任意种类和数目的头部,但它们都有各自的长度字段。
- 如果头部本身 64 位对准的话,两个 IPv6 地址也在 64 位边界对准。这样可以加快在 64 位体系结构上的处理。
- IPv6 头部没有与分片相关的字段,因为用于这个目的的有一个独立的分片头部。之所以做出这样的设计决策是因为分片属于异常情况,而异常情况是不应该减慢正常处理的。
- IPv6 头部没有其自身的校验和。这是因为所有上层(TCP、UDP 和 ICMPv6)都有各自的校验和,其校验范围包括上层头部、上层数据以及 IPv6 头部的这些字段:IPv6 源地址、IPv6 目的地址、有效负载长度以及下一个头部。把校验和从 IPv6 头部省掉后,转发分组的路由器修改了跳数限制也不必重新计算头部校验和。加快路由器的转发速度再次成为关键出发点。

如果你是首次遭遇 IPv6 的话,我们再指出从 IPv4 到 IPv6 的重大变化:

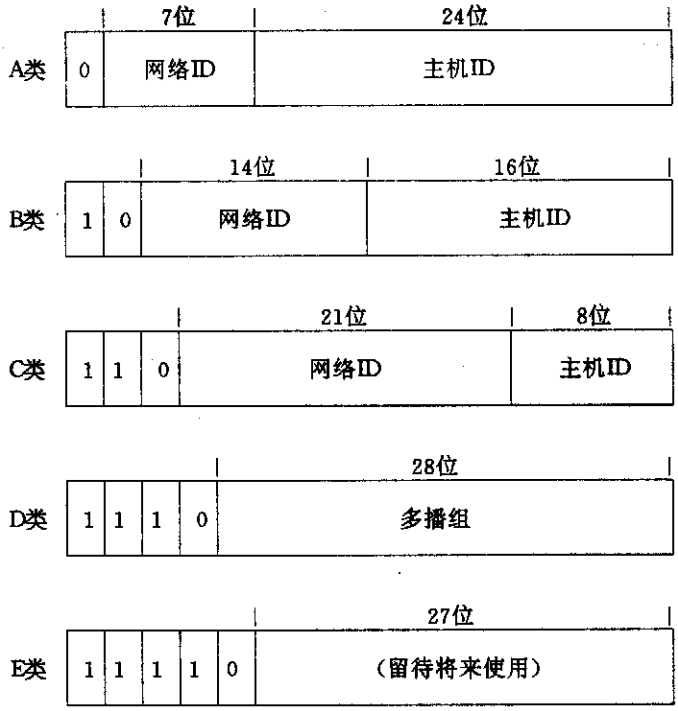
- IPv6 没有广播(第 18 章)。IPv4 中可选的多播(第 19 章)在 IPv6 中却是必需的。
- IPv6 路由器不给所转发的分组进行分片。IPv6 的分片只在起始主机上进行。

- IPv6要求支持认证和安全选项。
- IPv6要求支持路径MTU发现功能(2.9节)。从技术上说这种支持是可选的,像引导加载程序中的最小实现就可以省掉它,然而如果某个节点没有实现这个功能的话,它就不能发送大于IPv6最小链路MTU的数据报(576字节;2.9节)。

A.4 IPv4地址

32位的IPv4地址有如图A.3所示的5种格式。以前的做法是,一个组织赋予一个A类、B类或C类ID,之后它就可以随自己的意图任意处理该地址的主机ID。然而到了90年代中期随着无类地址的出现,情况发生了变化。

IPv4地址通常书写成以点号隔开的4个十进制数,这种书写称为点分十进制数记法(dotted-decimal notation),其中每个十进制数代表32位地址4个字节中的某一个。这4个十进制数中的第1个表明了地址类,图A.4给出了它们的关系。



图A.3 IPv4地址格式

类	范　围
A	**0**. 0. 0. 0 到 **127**. 255. 255. 255
B	**128**. 0. 0. 0 到 **191**. 255. 255. 255
C	**192**. 0. 0. 0 到 **223**. 255. 255. 255
D	**224**. 0. 0. 0 到 **239**. 255. 255. 255
E	**240**. 0. 0. 0 到 **247**. 255. 255. 255

图 A.4　5 类 IPv4 地址的范围

无类地址与 CIDR

IPv4 地址现在被认为是无类(classless)的。这就是说，我们可以忽略 A、B、C 三类地址之间的差别以及它们的网络 ID 和主机 ID 之间如图 A.3 所示的隐含界线。代之的是，无论何时把某个 IPv4 网络地址赋予一个组织，总是同时给出 32 位的网络地址和相应的 32 位掩码。掩码中值为 1 的位涵盖网络地址，值为 0 的位涵盖主机地址。既然掩码中值为 1 的位总是从最左位开始连续排列，因此掩码也可以使用前缀长度(prefix length)指定，它表示从左边开始连续的值为 1 的位数。例如，A 类地址的隐含掩码是 255. 0. 0. 0，其前缀长度为 8；B 类地址的隐含掩码是 255. 255. 0. 0，其前缀长度为 16；C 类地址的隐含掩码是 255. 255. 255. 0，其前缀长度为 24。

无类地址的优势是我们不再被限定在 A、B 和 C 三类地址的固定前缀长度 8、16 和 24 上。相反，任何地址可以赋予一个不同于缺省值的前缀长度。例如，因特网业务供应商(ISP)可以使用无类地址把 1 个 C 类地址赋给 4 个不同的客户，每个都使用值为 255. 255. 255. 192 的掩码，其前缀长度为 26。这些客户都有 6 位(而不是 8 位)可自行处置，他们可以在其上选择一个子网边界(如果需要的话)，然后赋予子网 ID 和主机 ID。

现在分配的所有因特网 IPv4 地址都是无类的，IPv6 也使用了同样的概念。IPv4 网络地址通常书写成点分十进制数，后跟一个斜杠，再跟前缀长度。图 1.16 给出了这样的例子。

使用无类地址要求无类路由，它通常称为 CIDR 即无类域间路由(Classless InterDomain Routing)(RFC 1519[Fuller et al. 1993])。使用 CIDR 的目的是减少因特网主干路由表的大小，延缓 IPv4 地址耗尽的速率。TCPv1 的 10.8 节涉及关于 CIDR 的更详细内容。

子网地址

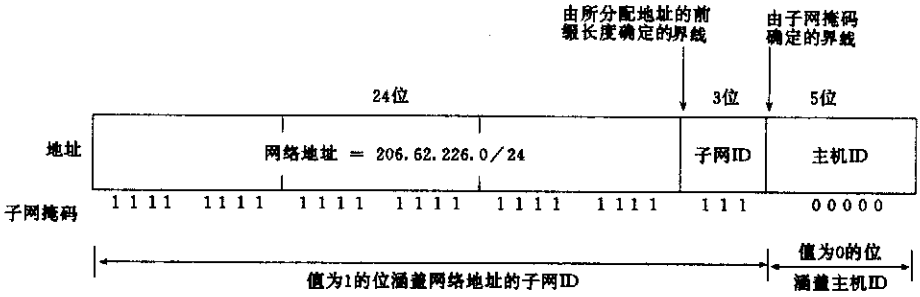
IPv4 地址通常划分成子网(RFC 950[Mogul and Postel 1985])。这么做增加了另外一级地址层次：

- 网络 ID(分配给网点)
- 子网 ID(由网点选择)
- 主机 ID(由网点选择)

网络 ID 和子网 ID 之间的界线由所分配网络地址的前缀长度确定。这个前缀长度通常是由相应组织的 ISP 赋予的。而子网 ID 和主机 ID 之间的界线是由网点选择的。同一子网上所有主机共享同一个子网掩码(subnet mask)，它指定子网 ID 和主机 ID 之间的界线。子

网掩码中值为1的位涵盖网络ID和子网ID，而值为0的位则涵盖主机ID。

例如让我们考虑图1.16中作者的子网。从ISP分配到的网络地址是206.62.226.0/24，它是一个完整的C类网络。作者然后把剩下的8位划分成3位子网ID和5位主机ID。图A.5表示出这样的划分。这些地址的子网掩码是0xffffffe0或255.255.255.224。



图A.5 24位网络地址加3位子网ID和5位主机ID

图1.16中顶部的子网其子网ID的3个位设置成001，我们把这个子网表示成206.62.226.32/27。其中"/27"的记号表明子网掩码由左边的27个位构成。整个网络地址(206.62.226.0/24)和各个子网地址(例如206.62.226.32/27)我们都使用这样的前缀记法。这个子网上各主机的地址在206.62.226.33和206.62.226.62范围内，而主机ID各位全是1的地址(206.62.226.63)则是该子网的广播地址(18.2节)。图1.16中另一个子网其子网ID的3个位设置成010，我们把它表示成206.62.226.64/27。

> 20世纪80年代中期IP地址子网划分刚开始时，不连续的子网掩码是允许的但不推荐使用(RFC 950[Mogul and Postel,1985])。但目前随着无类地址的使用，不连续子网掩码不再允许。IPv6也要求所有地址掩码从最左边的位开始保持连续。
>
> RFC950建议不要使用子网ID各位全为0或全为1的两个子网。现在有些软件支持这两种格式的子网ID。

作为图1.16中子网划分的另外一个例子，让我们考虑底部的子网。赋给noao.edu的网络地址为140.252.0.0/16，它是一个完整的B类网络。NOAO然后把剩下的16位划分成8位子网ID和8位主机ID，这是拥有B类地址组织的典型做法。我们在图A.6中表示了这样的划分。

子网掩码是0xffffff00或255.255.255.0。图中给出的子网其子网ID为1，我们把它表示成140.252.1.0/24。

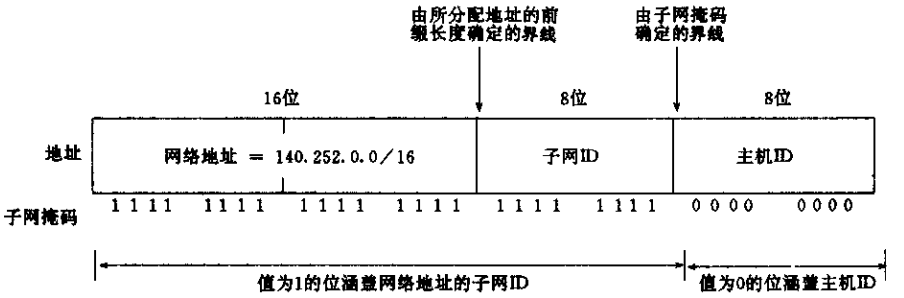


图 A.6 16 位网络地址加 8 位子网 ID 和 8 位主机 ID

回馈地址

按照约定地址 127.0.0.1 分配给回馈接口。任何发往这个 IP 地址的东西回馈成 IP 输入。我们在测试同一主机上的客户和服务器时经常使用这个地址。这个地址通常为人所知的名字是 INAPDR_LOOPBACK。

网络 127/8 上任何地址都可以分配给回馈接口，但 127.0.0.1 是最常用的。

未指定地址

由 32 个值均为 0 的位构成的地址是 IPv4 的未指定地址(unspecified address)。在 IPv4 分组中，它只能作为引导过程中的节点在获悉其 IP 地址前所发送分组中的源地址。在套接口 API 中，这个地址称为通配地址，其通常为人所知的名字是 INADDR_ANY。

多宿与地址别名

多宿主机(multihomed host)的传统定义是具有多个接口的主机：例如两个以太网或者一个以太网加一个点到点链路。每个接口必须有一个唯一的 IPv4 地址。在计算接口数以判定某个主机是否多宿时，回馈接口不算在内。

路由器按定义是多宿的，因为它要把到达某个接口的分组转发到另一个接口。然而多宿主机除非转发分组，否则并不是路由器。实际上多宿主机不应该仅为具有多个接口的原因而认定自己是路由器；除非它被配置成路由器(典型做法是由系统管理员打开某个配置选项)，否则它绝不能扮演这个角色。

然而多宿(multihoming)的说法现在已更为一般化，它包含两种不同的情况(RFC 1122 的 3.3.4 节[Braden 1989])。

1. 有多个接口的主机是多宿的，每个接口必须有各自的 IP 地址。这是传统的定义。
2. 较新的主机具备给指定的物理接口分配多个 IP 地址的能力，每个额外的 IP 地址称为第一个 IP 地址即主 IP 地址的一个别名(alias)或逻辑接口(logical interface)。通常作为别名的 IP 地址跟主地址共享同一个子网地址，只是主机 ID 不同。不过别名地址与主地址具有完全不同的网络地址或子网地址也是可能的。16.6 节给出了别名地址的一个例子。

因此多宿主机的定义是具有多个接口的主机，至于这些接口是物理的还是逻辑的则不必关心。

多宿也用在另外一个上下文中。有多个连接去往因特网的网络也称为多宿的。例如，有些网点有两个而不是一个去往因特网的连接，这样可提供备份能力。

A.5 IPv6 地址

IPv6 地址有 128 位，通常书写成 8 个 16 位十六进制数格式。跟 IPv4 不同，IPv6 本质上没有地址类；不过其 128 位地址的高序位隐含着地址类型([Hinden and Deering 1997])。图 A.7 给出了高序位不同值与所隐含地址类型的关系。

地址分配	格式前缀
保留	0000 0000
未分配	0000 0001
保留给 NSAP	0000 001
保留给 IPX	0000 010
未分配	0000 011
未分配	0000 1
未分配	0001
可聚集全局单播地址	001
未分配	010
未分配	011
未分配	100
未分配	101
未分配	110
未分配	1110
未分配	1111 0
未分配	1111 10
未分配	1111 110
未分配	1111 1110 0
链接局部单播地址	1111 1110 10
网点局部单播地址	1111 1110 11
多播地址	1111 1111

图 A.7 IPv6 地址中高序位的含义

这些高序位称为格式前缀(format prefix)。例如高序 3 位是 001 的地址称为可聚集全局单播地址(aggregatable global unicast address);高序 8 位是 11111111(0xff)的地址称为多播地址;高序 8 位是 00000000 的地址则是保留的。

可聚集全局单播地址

IPv6 地址中最常用的可能是可聚集全局单播地址,它们是以 001 这个 3 位前缀开头的。这些地址将取代 IPv4 中的 A、B、C 三类地址。

> IPv6 地址的初始规范即 RFC1884[Hinden and Deering 1995]称 3 位前缀是 010 的地址为基于供应商的单播地址(provider-based unicast addresses)。这些在 RFC 2073[Rekhter et al. 1997]中说明。然而在 1997 年 3 月召开的因特网工程任务攻坚组(IETF)会议上作出的决议是使用另外一种格式的单播地址继续探索。

基于聚集的单播地址格式在[Hinden,O' Dell,and Deering 1997]中定义,它从最左位开始往右包含如下各字段:

- 格式前缀(001)
- TLA ID(顶级聚集标识)
- NLA ID(次级聚集标识)
- SLA ID(网点级聚集标识;例如子网 ID)
- 接口 ID

图 A.8 给出了可聚集全局单播地址的例子。

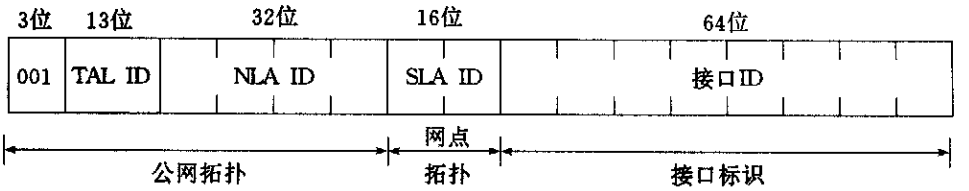


图 A.8 IPv6 可聚集全局单播地址

接口 ID 必须按 IEEE EUI-64 格式[IEEE 1997b]构造。这是大多数 LAN 接口卡所分配的 48 位 IEEE 802 MAC 地址的超集。这个标识应自动分配给接口,如果可能的话应基于其硬件 MAC 地址。构造基于 EUI-64 接口标识的细节在[Hinden and Deering 1997]的附录 A 中说明。

6bone 测试地址

6bone 是一个用于早期 IPv6 协议测试的虚拟网络(B.3 节)。书写本书时尽管已有在 6bone 上给可聚集全局单播地址使用某个特殊格式的计划,但还没有真正分配([Hinden, Fink,and Postel 1997])。相反,在 RFC 1897[Hinden and Postel 1996]中描叙的如图 A.9 的地址格式却在 6bone 的所有节点上使用。这些地址被认为是临时性的,将来分配可聚集全局单播地址时,使用这些地址的节点需要重新分配。

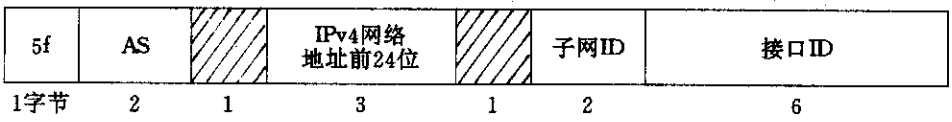


图 A.9 用于 6bone 的 IPv6 测试地址

6bone 测试地址的高序位字节是 0x5f。16 位的 AS 字段是分配给组织或她的 ISP 的自治系统号(autonomous system number)。它是用在 IPv4 中标识路由域的。下一个字段是节点当前 IPv4 地址的高序 24 位。子网 ID(subnet ID)是由组织选择的任意标识,接口 ID(interface ID)则通常是 48 位 IEEE 802 MAC 地址。

9.2 节中我们给图 1.16 中名为 solaris 的主机分配的 IPv6 地址为 5f1b:df00:ce3e:e200:0020:0800:2078:e3e3。其中 AS 是 7135(0x1bdf),IPv4 地址前 24 位是 206.62.226(0xce3ee2),子网掩码是 0x0020,低序 48 位则是该主机以太网卡的 MAC 地址。

IPv4 映射的 IPv6 地址

IPv4 映射的 IPv6 地址(IPv4-mapped IPv6 address)在因特网向 IPv6 的过渡阶段允许在同时支持 IPv4 和 IPv6 的主机上运行的 IPv6 应用进程与只支持 IPv4 的主机通信。这些地址是在某个 IPv6 应用进程查询某台主机的 IPv6 地址,而那台主机却只有 IPv4 地址时由 DNS 解析器(图 9.5)自动建立的。

我们从图 10.4 看到 IPv6 套接口上使用这种类型的地址将导致往目的 IPv4 主机发送 IPv4 数据报。这些地址并不保存在任何 DNS 数据文件中——它们是由解析器根据需要建立的。

图 A.10 给出了这些地址的格式。低序 32 位含有一个 IPv4 地址。

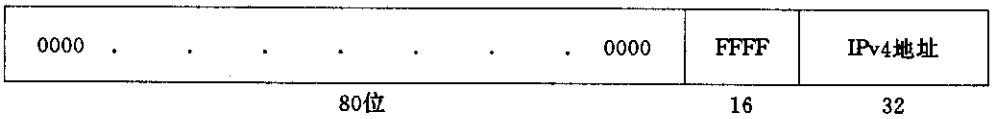


图 A.10 IPv4 映射的 IPv6 地址

书写 IPv6 地址时,连续的值为 0 的串可以简写成两个冒号。另外,嵌在其中的 IPv4 地址也可以用点分十进制数记法书写。例如,我们可以把 IPv4 映射的 IPv6 地址 0:0:0:0:0:FFFF:206.62.226.33 写成::FFFF:206.62.226.33。

IPv4 兼容的 IPv6 地址

IPv4 兼容的 IPv6 地址(IPv4-compatible IPv6 address)也用在从 IPv4 到 IPv6 的过渡阶段。一台同时支持 IPv4 和 IPv4 的主机没有邻近的 IPv6 路由器时,它的管理员应建立一个 DNS AAAA 记录,给出一个 IPv4 兼容的 IPv6 地址。有 IPv6 数据报要发送到这个地址的任何其他 IPv6 主机都将用一个 IPv4 头部来封装 IPv6 数据报,这种技术称为自动生成的隧道(automatic tunnel)。我们将在 B.3 节具体探讨隧道技术,并在图 B.2 中给出在 IPv4 头部封装 IPv6 数据报的例子。相反,6bone 上的每个隧道却是配置成的(configured)而不是自动生成的(例如是由管理员在某个启动文件中设置的)。有了 IPv4 兼容的 IPv6 地址后,只有地址需要手工配置(例如作为一个 AAAA 记录放到 DNS 的数据文件中),隧道则是自动生成的。

图 A.11 给出了 IPv4 兼容的 IPv6 地址的格式。

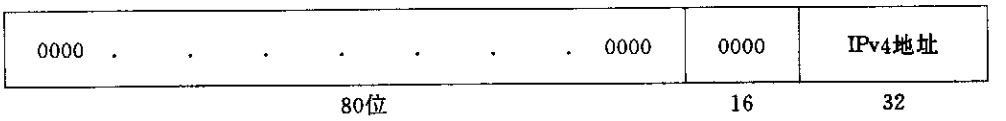


图 A.11　IPv4 兼容的 IPv6 地址

这种类型地址的例子之一是::206.62.226.33。

回馈地址

由 127 个值为 0 的位和单个值为 1 的位组成的 IPv6 地址(书写成::1)是 IPv6 的回馈地址。在套接口 API 中,它被称为 in6addr_loopback 或 IN6ADDR_LOOPBACK_INIT。

未指定地址

所有 128 位的值都为 0 的 IPv6 地址(书写成 0::0 或::)是 IPv6 未指定地址。在 IPv6 分组中,它只能作为引导过程中的节点在获悉其 IPv6 地址前所发送分组中的源地址。

在套接口 API 中,这个地址称为通配地址,指定它(例如把它 bind 到某个监听 TCP 套接口)表示相应套接口将接受目的地为本节点任意地址的客户连接。它被称为 in6addr_any 或 IN6ADDR_ANY_INIT。

链路局部地址

链路局部地址(link－local address)用在单个链路上,这种情况下数据报不会被转发。使用这种地址的例子包括引导时的自动地址配置以及邻居的发现(类似于 IPv4 的 ARP)。图 A.12 给出了这些地址的格式。

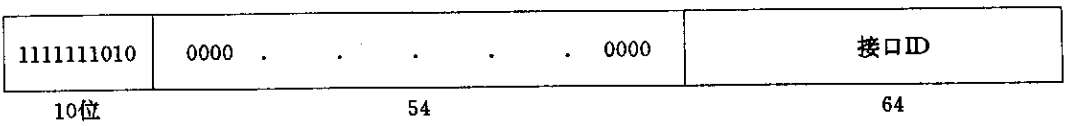


图 A.12　IPv6 链路局部地址

这些地址总是以 fe80 开头。IPv6 路由器绝不能把使用链路局部源或目的地址的数据报转发到另一个链路上。9.2 节我们给出了与名字 aix_611 相关联的链路局部地址。

网点局部地址

网点局部地址(site-local address)可用在一个网点范围内的编址,不需要额外的全局前缀。图 A.13 给出了这些地址的格式。

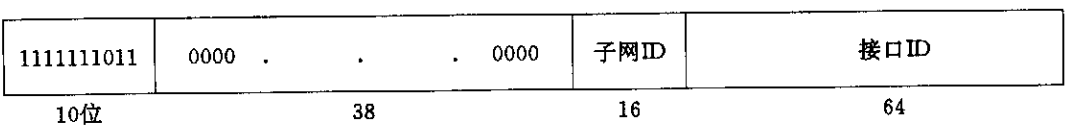


图 A.13　IPv6 网点局部地址

这些地址总是以 fec0 开头。IPv6 路由器绝不能把使用网点局部源或目的地址的数据报转发到网点之外。

A.6 ICMPv4 和 ICMPv6:网际控制消息协议

ICMP 是任何 IPv4 或 IPv6 实现都必需的有机组成部分。它通常用在 IP 节点(包括路由器和主机)之间互通出错消息,但应用程序也偶尔使用它们。例如 Ping 和 Traceroute 程序(第 25 章)就使用 ICMP。

ICMPv4 和 ICMPv6 消息的前 32 位是相同的,如图 A.14 所示。RFC 792[Postel 1981b]叙述了 ICMPv4,RFC1885[Conta and Deering 1995]叙述了 ICMPv6。

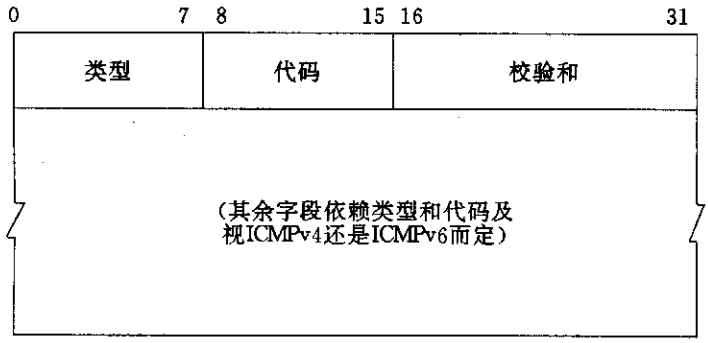


图 A.14 ICMPv4 和 ICMPv6 消息的格式

8 位类型(type)是 ICMPv4 或 ICMPv6 消息的类型,有些类型有 8 位的代码(code)提供额外信息。校验和(checksum)是标准的因特网检验和,不过校验范围 ICMPv4 和 ICMPv6 存在差异。

从网络编程角度看,我们需要知道哪些 ICMP 消息能返送给应用进程,哪些情况会导致出错以及错误如何返回应用进程。图 A.15 列出了所有的 ICMPv4 消息以及 4.4BSD 如何处理它们。最后一列指出给应用进程返送的出错消息所对应的 errno 变量的值。图 A.16 所列的是 ICMPv6 消息。

类型	代码	说明	处理者或 errno
0	0	回射应答	用户进程(Ping)
3		目的地不可达:	
	0	网络不可达	EHOSTUNREACH
	1	主机不可达	EHOSTUNREACH
	2	协议不可达	ECONNREFUSED
	3	端口不可达	ECONNREFUSED
	4	需要分片但 DF 位设置	EMSGSIZE
	5	源路径失败	EHOSTUNREACH
	6	目的网络不可知	EHOSTUNREACH
	7	目的主机不可知	EHOSTUNREACH
	8	源主机隔离(过时不用)	EHOSTUNREACH

（续）

类型	代码	说明	处理者或 errno
	9	目的网络由管理手段禁止访问	EHOSTUNREACH
	10	目的主机由管理手段禁止访问	EHOSTUNREACH
	11	因 TOS 网络不可达	EHOSTUNREACH
	12	因 TOS 主机不可达	EHOSTUNREACH
	13	通信由管理手段禁止	（忽略）
	14	主机优先级侵权	（忽略）
	15	在效果上优先级受损	（忽略）
4	0	源熄灭	TCP 由内核处理，UDP 忽略
5		重定向：	
	0	重定向网络	内核更新路由表
	1	重定向主机	内核更新路由表
	2	重定向服务类型和网络	内核更新路由表
	3	重定向服务类型和主机	内核更新路由表
8	0	回射请求	内核生成应答
9	0	路由器通告	用户进程
10	0	路由器征求	用户进程
11		超时：	
	0	传送中 TTL 等于 0	用户进程
	1	重组时 TTL 等于 0	用户进程
12		参数问题：	
	0	IP 头部坏（包罗一切的错误）	ENOPROTOOPT
	1	所需选项遗漏	ENOPROTOOPT
13	0	时间戳请求	内核生成应答
14	0	时间戳应答	用户进程
15	0	信息请求（过时不用）	（忽略）
16	0	信息应答（过时不用）	用户进程
17	0	地址掩码请求	内核生成应答
18	0	地址掩码应答	用户进程

图 A.15　4.4BSD 对 ICMP 消息类型的处理

类型	代码	说明	处理者或 errno
1		目的地不可达	
	0	没有到目的地的路径	EHOSTUNREACH
	1	由管理手段禁止（防火墙过滤器）	EHOSTUNREACH
	2	非邻居（不正确的严格源路径）	EHOSTUNREACH
	3	地址不可达（任何其他原因）	EHOSTDOWN
	4	端口不可达（UDP）	ECONNREFUSED
2	0	分组太长	内核进行 PMTU 发现
3		超时：	
	0	传送中超过跳限	用户进程
	1	片段重组超时	用户进程
4		参数问题：	
	0	错误的头部字段	ENOPROTOOPT

（续）

类型	代码	说明	处理者或 errno
	1	无法认出下一个头部	ENOPROTOOPT
	2	无法认出选项	ENOPROTOOPT
128	0	回射请求(Ping)	内核生成应答
129	0	回射应答(Ping)	用户进程(Ping)
130	0	组成员关系查询	用户进程
131	0	组成员关系汇报	用户进程
132	0	组成员关系缩减	用户进程
133	0	路由器征求	用户进程
134	0	路由器通告	用户进程
135	0	邻居征求	用户进程
136	0	邻居通告	用户进程
137	0	重定向	内核更新路由表

图 A.16 ICMPv6 消息

记号“用户进程”的含义是内核不处理这些消息，它们由打开原始套接口的用户进程处理。我们还得注意不同的实现对于特定的消息可能有不同的处理。例如，尽管 Unix 系统通常在一个用户进程中处理路由器征求与路由器通告，其他实现可能是在内核中处理这些消息。

ICMPv6 给错误性消息(类型 1～4)清除类型字段的最高位，给信息性消息(类型 128～137)则设置该位。

附录B 虚拟网络

B.1 概 述

当TCP中加进一个新特性(例如定义在RFC 1323中的长胖管道支持)时,相应支持只在使用TCP的主机上需要;路由器上是不需要改动的。像RFC 1323这样的变动是慢慢出现在TCP的主机实现中的,当建立新的TCP连接时,两端主机能彼此判定对方是否支持新特性。如果两者都支持就能用上这些特性。

这跟诸如80年代末的多播和90年代中的IPv6对IP层所做的改动是不一样的,因为这些新特性要求主机和所有路由器都变动。然而人们不愿等到所有系统都升级才开始使用新特性。为此目的人们使用隧道(tunnel)技术在已有的IPv4因特网上建立虚拟网络(virtual network)。

B.2 MBone

我们使用隧道建立的第一个虚拟网络例子是MBone,它是从1992年开始的[Eriksson 1994]。如果一个LAN上有2个或多个主机支持多播,多播应用进程就能运行在所有这些主机上并彼此通信。要把这样的LAN连接到另外一个同样有可多播主机的LAN上,每个LAN中得有一个主机相互间配置成一个隧道,如图B.1所示。图中我们用数字标出了如下的步骤。

1. 源主机MH1上的某个应用进程向一个D类地址发送一个多播数据报。
2. 我们的例子给出的是UDP数据报,因为大多数多播应用程序使用UDP。我们在第19章较具体地讨论了多播以及如何发送和接收多播数据报。
3. 该数据报由本LAN上所有可多播主机接收,包括MR2。我们注意MR2也用作多播路由器,运行着mrouted程序,由它进行多播路由。MR2是隧道的源端(tunnel source)。
4. MR2在这个数据报前添加另外一个IPv4头部,并把这个新头部的目的IPv4地址设置成隧道末端(tunnel endpoint)MR5的单播地址。这个单播地址是由MR2的管理员配置并由mrouted程序在启动时读入的。类似地,在隧道另一端的MR5上也配置了MR2的地址。新的IPv4头部的协议字段设置成4,它代表IPv4套IPv4(IPv4-in-IPv4)封装。本数据报发送给下一跳路由器UR3,而它已被明确标明是一台单播路由器。这就是说UR3不理解多播,这也是我们使用隧道的彻底原因。这个IPv4数据报的阴影部分除IPv4头部的TTL字段减1外,跟第1步发送时相比没有变化。
5. UR3查找最外层IPv4头部中的目的IPv4地址,然后把数据报转发给下一跳路由器UR4,它是另外一个单播路由器。

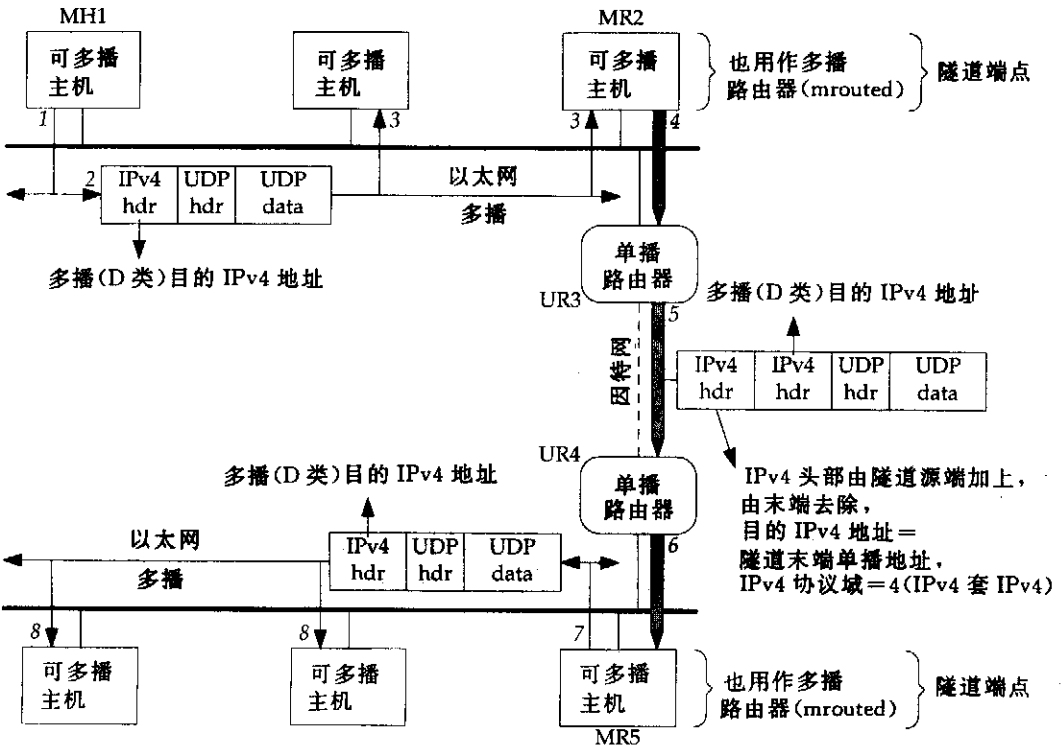


图 B.1 用于 MBone 的 IPv4 套 IPv4 封装

6. UR4 把数据报递送到它的目的地 MR5，它是隧道的末端。
7. MR5 接收数据报，发现其协议字段是 IPv4 套 IPv4 封装就去除第 1 个 IPv4 头部，然后把该数据报的剩余部分(也就是在顶部 LAN 上多播的数据报的拷贝)作为一个多播数据报输出到自己的 LAN 上。
8. 底部 LAN 上的所有可多播主机都接收到这个多播数据报。

最终结果是在顶部 LAN 发出的多播数据报也作为一个多播数据报传送到底部 LAN。即使跟这两个 LAN 分别相连的两个路由器以及它们之间的所有因特网路由器都没有多播能力也能做到这一点。

本例中我们指出了每个 LAN 上有一台主机通过运行 mrouted 程序提供多播路由功能。这是 MBone 一开始的情况。到 1996 年左右多播路由功能开始出现在大多数主要路由器厂商生产的路由器中。要是图 B.1 中的两个单播路由路 UR3 和 UR4 具有多播能力的话，我们就根本不需要运行 mrouted，UR3 和 UR4 就用作多播路由器。不过要是 UR3 和 UR4 之间还有无多播能力的其他路由器的话，隧道还是必需的。这时的隧道端点将是 MR3(UR3 的可多播替代物)和 MR4(UR4 的可多播替代物)，而不是 MR2 和 MR5。

在图 B.1 所示的情形中，每个多播分组在顶部和底部 LAN 上都出现两次：一次是作为多播分组，另一次是作为隧道内的单播分组呈现在运行 mrouted 的主机和下一跳单播路由器之间(例如 MR2 和 UR3 之间以及 UR4 和 MR5 之间)的传送之中。这个额外拷贝是引入隧道技术的代价。图 B.1 中的两个单播

路由器 UR3 和 UR4 改用可多播路由器(我们称它们为 MR3 和 MR4)取代的好处是免除了出现在 LAN 上的每个多播分组的额外拷贝。即使 MR3 和 MR4 仍必须在它们之间建立一条隧道,因为它们之间某些中间路由器(图中未表示)可能没有多播能力;这样的取代还是有优势的,毕竟它避免了每个 LAN 上的重复拷贝。

B.3 6bone

6bone 是基于类似于 MBone 的原因于 1996 年建立的一个虚拟网络,也就是说由支持 IPv6 的主机构成的孤岛上的用户希望使用虚拟网络彼此连接起来,而不必等到所有的中间路由器都变成支持 IPv6。图 3.2 给出了支持 IPv6 的两个 LAN 使用隧道跨越只支持 IPv4 的中间路由器彼此连接的例子。图中我们用数字标出了如下的步骤。

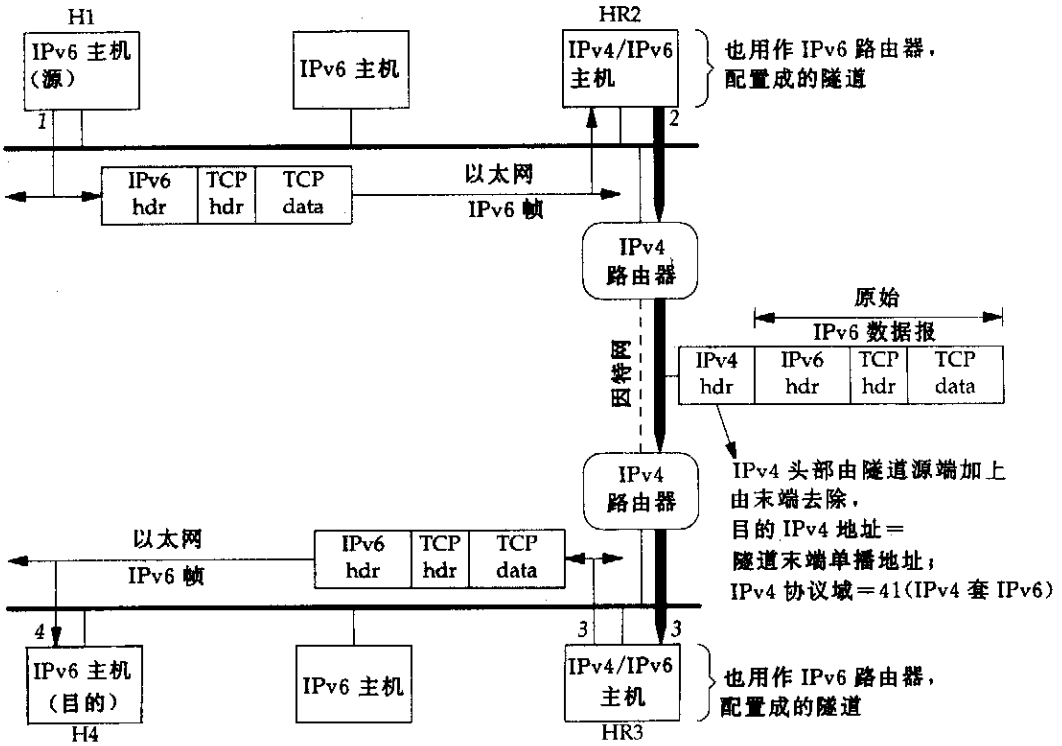


图 B.2 用于 6bone 的 IPv4 套 IPv6 封装

1. 顶部 LAN 上的主机 H1 发送一个 IPv6 数据报到底部 LAN 上的主机 H4,其中含有一个 TCP 分节。我们把这两台主机标成"IPv6 主机",但它们都可能还同时运行 IPv4。H1 上的 IPv6 路由表指定主机 HR2 为下一跳路由器,因此 IPv6 数据报发送给这台主机并由它转发。
2. 主机 HR2(隧道源端)上有一个配置成的到达主机 HR3 的隧道。这个隧道通过在 IPv4 数据报中封装 IPv6 数据报(称为"IPv4 套 IPv6 封装")以允许 IPv6 数据报在它

的两个端点之间跨越 IPv4 因特网发送。IPv4 协议字段的值为 41。我们注意到隧道两端的两台 IPv4/IPv6 主机 HR2 和 HR3 同进也用作 IPv6 路由器，因为它们把从一个接口接收到的 IPv6 数据报转发到另一个接口。配置成的隧道也算一个接口，尽管它只是一个虚拟的而非物理的接口。

3. 隧道末端的 HR3 接收到封装过的数据报后剥掉 IPv4 头部，然后把 IPv6 数据报发送到它的 LAN 上。
4. 目的主机 H4 接收到这个 IPv6 数据报。

这些虚拟网络的最终目的是随着时间的推移，中间环节的路由器获得所需的功能(就 MBone 来说是多播路由，就 6bone 来说是 IPv6 路由)后，它们就消失，我们叙述这两种虚拟网络是因为正文中有些例子使用了 MBone 和 6bone。

附录C 调试技术

本附录包含调试网络应用程序的建议和技术。没有一种技术能满足所有人的需要;相反,我们应熟悉各种各样的工具,然后在我们的环境中使用起作用的任何工具。

C.1 系统调用跟踪

许多版本的 Unix 提供系统调用跟踪工具。它通常能提供有价值的调试技术。

在这个级别上调试程序时,我们需要区分系统调用和函数。前者是指向内核中的表项,我们可以使用将在本节介绍的工具来跟踪它们。Posix 以及大多数其他标准使用函数这个术语来描述从用户看来象函数的东西,即使在某些实现上它们可能是系统调用也这样。例如在源自 Berkeley 的内核上 socket 是一个系统调用,但对应用编程人员来说看起来是个普通的 C 函数。然而在 SVR4 中我们马上会发现它是套接口函数库中的一个库函数,由它调用 putmsg 和 getmsg,而后两者才是真正的系统调用。

本节我们检查运行时间/日期客户程序过程中涉及的系统调用。我们在图 1.5 中给出了该程序的套接口版本,在图 28.13 中给出了它的 XTI 版本。

SVR4 基于流的套接字函数库

我们以 SVR4 中基于流的套接字实现开始讨论。SVR4 提供的 truss 程序能运行另外一个程序并跟踪所执行的系统调用。我们如下执行这个程序。

```
unixware % truss -o truss.out -v getmsg,putmsg,ioctl \
daytimetcpcli 140.252.1.54
```

(我们把这个长命令折成了两行。)其中-o 选项把输出定向到一个文件(这种技术通常有大量的输出),-v 选项则给指定的 3 个系统调用打开详尽跟踪功能。这将给出有关这些系统调用参数的额外信息。

输出的开头部分(约 40 行)给出使用映射到内存的 I/O 把动态函数库链接到程序的信息。这是由使用 open 和 mmap 这两个系统调用产生的。(第二个系统调用在 APUE 的 12.9 节中介绍,它与我们就网络 API 的讨论无关。)我们省略掉这些行。

接下来是打开/etc/netconfig 文件,再读入整个文件(806 字节)。

```
open("/etc/netconfig",O_RDONLY,0666)          = 3
...
read(3," t c p\t t p i_c o t s"...,8192)       = 806
read(3,0x0804A6B8,8192)                        = 0
...
close(3)                                       = 0
```

我们已经省略掉了与当前讨论无关的某些 ioctl 和 lseek 调用。调用 open 一行等号右侧的值了是返回的描述字。调用 read 时请求 8192 字节,但返回值却是 806(文件大小)。下一次

调用 read 返回的是 0(文件结束)。truss 还给出由 read 返回的前 12 字节(以 tcp 开头),它是文件第一行的开始部分。该文件然后关闭。我们可以猜想这个对 netconfig 文件的读操作是在首次调用 socket 启动套接口函数库时完成的。

接下来的系统调用是针对/dev/tcp 的 open,所返回的描述字还是 3。

```
open("/dev/tcp", O_RDWR, 027776333624)          = 3
ioctl(3, I_FIND,"sockmod")                      = 0
ioctl(3, I_PUSH,"sockmod")                      = 0
...
ioctl(3, I_STR,0x080467E4)                      = 0
        cmd=TI_BIND timout=-1  len=32  dp=0x0804ABA8
...
```

open 之后的 ioctl 检查模块 sockmod 是否已经在流上。返回值 0 表示它还没在流上,因此下一个 ioctl 执行 I_PUSH 把它推入流中。此后是针对这个流的多个 ioctl 和大量的信号处理(这里我们略掉了)。执行 I_STR 的 ioctl 发送一个内部流 ioctl 消息,其命令是 TI_BIND。长度 32 可能指的是一个 T_BIND_REQ,它在 T_bind_req 结构(我们在图 33.9 中讨论过)中由四个 4 字节的成员构成,其后跟一个 16 字节的 sockaddr_in 结构。这也许是我们在未绑定的套接口上调用 connect 时由套接口函数库执行的跟某个任意本地地址的捆绑。

> 就跟在我们的 tpi_bind 函数中一样,我们会在这儿预期看到一个 putmsg 后跟一个 getmsg。

接着我们看到第一次调用 putmsg,它只是用于控制信息,所用标志为 0。

```
putmsg(3, 0x08046958,0x00000000, 0)             = 0
      ctl:   maxlen=428    len=36    buf=0x0804ABA8
getmsg(3, 0x08046924,0x08046918,0x08046934)     = 0
      ctl:   maxlen=428    len=8     buf=0x0804ABA8
      dat:   maxlen=128    len=-1    buf=0x08046898
getmsg(3, 0x08046964, 0x08046958, 0x0804697C)   = 0
      ctl:   maxlen=428    len=56    buf=0x0804ABA8
      dat:   maxlen=128    len=-1    buf=0x080468D8
```

其中长度 36 可能指一个 T_CONN_REQ 请求(图 33.10):五个 4 字节的值后跟一个 16 字节的 sockaddr_in 结构。接下来的 getmsg 返回 8 字节的控制信息但没有数据,这也许是一个 T_OK_ACK 消息(图 33.10)。它后面的 getmsg 返回 56 字节的控制信息,也没有数据返回,这可能是一个 T_CONN_CON 消息(图 33.10):五个 4 字节成员,一个 16 字节 sockaddr_in 结构再加 20 字节的选项。我们只能猜想最后的 20 个字节是选项,因为 t_opthdr 结构(四个 4 字节成员,32.2 节)后跟 1 字节的 IP_TOS 选项(这是此刻我们可预期已经返回的来自图 32.1 的唯一端到端选项),再后跟 3 字节的垫补(图 32.3),恰好是 20 字节。

下一个系统调用是 getmsg,它返回 8 字节的控制信息和 26 字节的数据。这也许是一个 T_DATA_IND 消息。

```
getmsg(3, 0x08046AEC,0x08046AE0, 0x08046B14)    = 0
      ctl:   maxlen=428    len=8     buf=0x0804ABA8
      dat:   maxlen=4096   len=26    buf=0x08046BB0
write(1, "Fri Apr  4 0"..., 26)      = 26
```

```
getmsg(3, 0x08046AEC, 0x08046AE0, 0x08046B14)    = 0
      ctl:  maxlen=428    len=-1  buf=0x0804ABA8
      dat:  maxlen=4096   len=0   buf=0x08046BB0
  _exit(0)
```

我们的客户程序接着调用 write 把这个 26 字节的数据写到标准输出(描述字为 1)。下一次调用 getmsg 不返回控制信息,返回数据则为 0 个字节,它也许表明供应者已碰上文件尾。

此刻我们期待接收到一个 T_ORDREL_IND 消息。

SVR4 基于流的 XTI 函数库

我们下一个例子是 Solaris 2.6 上基于流的 XTI 函数库。我们预期其中的系统调用跟 33.6 节中的调用类似。

在输出中用 mmap 映射函数库之后,我们找到针对 TCP 传输提供者的 open 调用,它后面跟着一个对 timod 的检查,然后就是把这个模块推入流中。

```
...
open("/dev/tcp",O_RDWR)                    = 3
ioctl(3, I_FIND, "timod")                  = 0
ioctl(3, I_PUSH, "timod")                  = 0
```

接着我们找到多个 ioctl 调用并伴随信号处理(这些我们略掉了)。其中有一个 ioctl 调用看来是跟我们调用 t_bind 对应的,显然 XTI 函数库是通过调用这个针对 timod 的 ioctl 来完成捆绑的。(这跟我们前面在 UnixWare 上看到的基于流的套接口例子是类似的。)

第一次调用 putmsg 发送 36 字节的控制信息但没有数据。这也许是因我们调用 t_connect 而产生的 T_CONN_REQ 请求。

```
  ...
putmsg(3, 0xEFFFE7F4, 0x00000000, 0)              = 0
      ctl:  maxlen=912    len=36    buf=0x0002BAB8: "\0\0\0\0\0\0\010"...
getmsg(3, 0xEFFFE710, 0xEFFFE700, 0xEFFFE71C)     = 0
      ctl:  maxlen=912    len=8     buf=0x0002C1E8: "\0\0\013\0\0\0\0"
      dat:  maxlen=0      len=-1    buf=0x00000000
      flags: 0x0001
getmsg(3, 0xEFFFE7F4, 0xEFFFE770, 0xEFFFE77C)     = 0
      ctl:  maxlen=912    len=36    buf=0x0002BAB8: "\0\0\0\f\0\0\010"..
      dat:  maxlen=0      len=-1    buf=0x00000000
      flags: 0x0000
```

我们还看到 Solaris 使用 C 转义序列以十六进制格式输出缓冲区的前 8 个字节。对于刚才的 putmsg,我们看到 7 个字节的 0 后跟 1 个字节的 0x10(16)。但它实际上是两个 4 字节成员:第一个 4 字节是 T_CONN_REQ 请求(它的值为 0),第二个 4 字节是目的地址的长度(16)。

前面的输出中第一次调用 getmsg 返回一个 T_OK_ACK 消息,下一次调用返回一个 T_CONN_CON 消息。我们之所以能分辨出第一次返回消息的类型是因为 T_OK_ACK 的值是 19(0x13),并且后面所跟的 4 个字节指示所确认的原语(即 T_CONN_REQ,它的值我

们已经说过是0)。我们之所以能分辨出第二次返回消息的类型是因为T_CONN_CON的值为12(它用C转义序列输出就是换表符\f),后面所跟的是对方地址的长度(16,输出成0x10)。

Solaris输出的标志是由getmsg的最后一个参数所指向变量的值。我们参到第一次调用返回值为1(它表示MSG_HIPRI,这正是我们期望的,因为T_OK_ACK消息是一种M_PCPROTO消息),第二次调用返回值为0(这表示普通的消息,它也是所期望的)。我们还注意到Solaris的T_CONN_CON消息看来并不返回任何选项(长度只有36字节),而前面的UnixWare例子看来却返回了20字节的选项(长度共56字节)。

我们感兴趣的下一个系统调用是另外一次调用getmsg,它也许对应我们调用t_rcv。这次调用返回26字节的数据但没有控制信息,它也许是跟服务器的响应对应的M_DATA消息。

```
getmsg(3, 0xEFFFE7EC, 0xEFFFE7DC, 0xEFFE81C)      = 0
      ctl:    maxlen=912    len=-1    buf=0x0002BAB8
      dat:    maxlen=4096   len=26    buf=0xEFFFE8D0: "Fri Apr"..
      flags: 0x0000
write(1, "Fri Apr  4 1"..., 26)                    = 26
getmsg(3,0xEFFFE7EC, 0xEFFFE7DC, 0xEFFFE81C)       = 0
      ctl:    maxlen=912    len=4     buf=0x0002BAB8: "\0\0\017"
      dat:    maxlen=4096   len=-1    buf=0xEFFFE8D0
      flags:  0x0000
  ...
  _exit(0)
```

我们的客户进程接着调用write,因此下一次调用getmsg返回一个T_ORDREL_IND消息(4字节的控制信息,没有数据)。T_ORDREL_IND的值是23,输出成0x17。

BSD内核套接口

我们的下一个例子是BSD/OS,它是源自Berkeley的内核,其中所有的套接口函数都是系统调用。系统调用跟踪程序是ktrace。它把跟踪信息写到一个文件(其缺省名字为ktrace.out),我们可使用kdump输出。执行套接口客户程序的命令如下:

```
bsdi % ktrace daytimetcpcli 206.62.226.43
Fri Apr 4 17:24:30 1997
```

我们然后运行kdump把跟踪信息倾泻到标准输出。

```
13187 daytimetcpcli CALL    socket(0x2,0x1,0)
13187 daytimetcpcli RET     socket 3

13187 daytimetcpcli CALL    connect(0x3,0xefbfc9a0,0x10)
13187 daytimetcpcli RET     connect 0

13187 daytimetcpcli CALL    read(0x3,0xefbfc9b0,0x1000)
13187 daytimetcpcli GIO     fd 3 read 26 bytes
      "Fri Apr  4 17:24:30 1997\r\n"
13187 daytimetcpcli RET     read 26/0x1a

...
13187 daytimetcpcli CALL    write(0x1,0x9000,0x1a)
```

```
13187 daytimetcpcli GIO      fd 1 wrote 26 bytes
      "Fri Apr  4 17:24:30 1997\r\n"
13187 daytimetcpcli RET      write 26/0x1a
13187 daytimetcpcli CALL     read(0x3,0xefbfc9b0,0x100)
13187 daytimetcpcli GIO      fd 3 read 0 bytes
      ""
13187 daytimetcpcli RET      read 0
13187 daytimetcpcli CALL     exit (0)
```

其中 13187 是进程 ID。CALL 标明系统调用，RET 是返回值，GIO 则代表普通进程 I/O。我们看到调用 socket 和 connect 之后是返回 26 个字节的 read 调用。我们的客户进程把这些字节写到标准输出，下一次调用 read 时返回 0(文件结束)。

Solaris 2.6 内核套接口

Solaris 2.x 基于 SVR4，2.6 以前的所有版本如图 33.3 所示实现套接口。以这种方式实现套接口的所有 SVR4 系统存在的一个问题是难以提供跟源自 Berkeley 的内核套接口 100%的兼容。为提供额外的兼容性，solaris 2.6 修改了实现技术，使用 sockfs 文件系统实现了套接口。这样就提供了内核套接口，我们可以使用 truss 在我们的套接口客户程序上进行验证。

```
solaris26 % truss -v connect daytimetcpcli 198.69.10.4
Sat Apr  5 11:32:07 1997
```

经过通常的函数库动态链接后，我们看到的第一个系统调用是 so_socket，它是我们调用 socket 引发的系统调用。

```
so_socket(2, 2, 0, "", 1)                    = 3
connect(3, 0xEFFFE8C8, 16)                   = 0
        name = 198.69.10.4/13
read(3, "Sat Apr  5 1"..., 4096)             = 26
  ...
Sat Apr  5 11:32:07 1997
write(1, "Sat Apr  5 1"...,26)               = 26
read(3, 0xEFFFE8D8, 4096)                    =0
  ...
  _exit(0)
```

其中 so_socket 的前 3 个参数就是我们调用 socket 的 3 个参数。

我们接下来看到的系统调用是 connect，当 truss 以-v connect 标志执行时，它还输出由第 2 个参数所指向的套接口地址结构的内容(IP 地址和端口号)。我们用省略号省掉的只是一些处理标准输入和标准输出的系统调用。

这种新的实现方法的副作用之一是往内核增加了 18 个系统调用。

C.2 标准因特网服务

搞熟图 2.13 说明的标准因特网服务。我们为测试所提供的客户程序已经多次使用了 daytime 服务。discard 服务是我们可以发送数据的方便端口。echo 服务则类似于整本书都用

到的回射服务器。

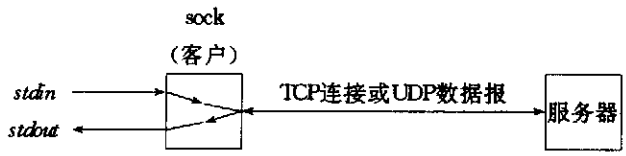
> 许多网点现在禁止穿越防火墙访问这些服务，这是因为1996年出现的一些拒绝服务型攻击用到了这些服务(习题12.3)。不过在你自己的网络内部，你是有希望使用这些服务的。

C.3 sock程序

作者编写的sock程序最早出现在TCPv1中，在那儿它经常用来产生特殊的个案条件，本书已使用tcpdump程序检查了其中的大多数条件。这个程序的便利之处在于能够生成如此之多的不同条件，从而免除了被迫编写特殊测试程序之苦。

我们在正文中没有给出这个程序的源代码(它是超过2000行的C程序)，但它是可以免费获取的(参见前言)。该程序运行在四种模式之一，每种模式既可使用TCP也可使用UDP。

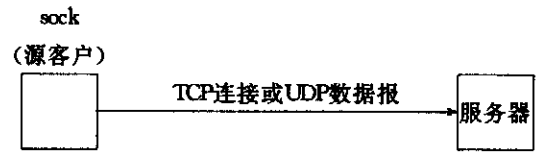
1. 标准输入，标准输出客户(图C.1)。



图C.1 sock客户，标准输入，标准输出

客户模式下从标准输入读入的任何东西都写到网络，从网络上收到的任何东西都写到标准输出。服务器的IP地址和端口必须指定，如果使用TCP该程序就执行一次主动打开操作。

2. 标准输入，标准输出服务器。这种模式类似于上一种模式，差别只是该程序把一个众所周知的端口捆绑到它的套接口上，如果使用TCP还执行一次被动打开操作。
3. 源客户(图C.2)。



图C.2 作为源客户的sock程序

该程序以某个指定的大小向网络执行固定数目的写操作。

4. 漏槽服务器(图C.3)。

图C.3 作为漏槽服务器的sock程序

该程序从网络执行固定数目的读操作。

这四个操作模式与以下四个命令相对应：

```
sock[options] hostname service
sock[options] -s [hostname]service
sock[options] -i hostname service
sock[options] -is [hostname]service
```

其中 hostname 是主机名或 IP 地址，service 是服务名或端口号。在两种服务器模式中除非指定了任选的 hostname，否则捆绑的是通配地址。

sock 程序约有 40 个命令行选项可以指定，我们不详细说明这些选项，不过第 7 章中叙述的套接口选项几乎都能设定。不给出任何参数执行本程序输出如下的选项总结：

```
-b n   bind n as client's local port number
-c     convert newline to CR/LF & vice versa
-f a.b.c.d.p   foreign IP address = a.b.c.d, foreign port# = p
-g a.b.c.d loose source route
-h     issue TCP half-close on standard input EOF
-i     "source" data to socket, "sink" data from socket (w/-s)
-j a.b.c.d join multicast group
-k     write or writev in chunks
-l a.b.c.d.p   client's local IP address = a.b.c.d, local port# = p
-n n   #buffers to write for "source" client (default 1024)
-o     do NOT connect UDP client
-p n   #ms to pause before each read or write (source/sink)
-q n   size of listen queue for TCP server (default 5)
-r n   #bytes per read() for "sink" server (default 1024)
-s     operate as server instead of client
-u     use UDP instead of TCP
-v     verbose
-w n   #bytes per write() for "source" client (default 1024)
-x n   #ms for SO_RCVTIMEO (receive timeout)
-y n   #ms for SO_SNDTIMEO (send timeout)
-A     SO_REUSEADDR option
-B     SO_BROADCAST option
-D     SO_DEBUG option
-E     IP_RECVDSTADDR option
-F     fork after connection accepted (TCP concurrent server)
-G a.b.c.d   strict source route
-H n   IP_TOS option (16=min del, 8=max thru, 4=max rel, 2=min cost)
-I     SIGIO signal
-J n   IP_TTL option
-K     SO_KEEPALIVE option
-L n   SO_LINGER option, n = linger time
-N     TCP_NODELAY option
-O n   #ms to pause after listen,but before first accept
-P n   #ms to pause before first read or write (source/sink)
-Q n   #ms to pause after receiving FIN, but before close
-R n   SO_RCVBUF option
-S n   SO_SNDBUF option
-T     SO_REUSEPORT option
```

```
-U n   enter urgent mode before write number n (source only)
-V     use writev() instead of write(); enables -k too
-W     ignore write errors for sink client
-X n   TCP_MAXSEG option (set MSS)
-Y     SO_DONTROUTE option
-Z     MSG_PEEK
-2     IP_ONESBCAST option (255.255.255.255) for broadcast
```

C.4 小测试程序

作者书写本书时一直使用的另外一个有用的调试技术就是编写简单的测试程序，检查某个给定的特性在精心构造的测试个案中是否起作用。这在测试一组库函数的包裹函数和一些简单错误处理函数时非常有用。这样一来减少了我们必须编写的代码量，同时又能满足测试错误的需求。

例子：XTI 带外数据，什么带？

作为这种技术的一个例子，它与系统调用跟综相结合将回答这样的问题："XTI 是如何在 TCP 中发送带外数据的？"我们首先建立跟服务器的连接，接着调用 t_snd 发送 1 个字节，发送时设置 T_EXPEDITED 标志。图 C.4 给出了我们的简单测试程序。

```
1 #include    "unpxti.h"
2 int
3 main(int argc, char **argv)
4 {
5     int         tfd;
6     if (argc != 3)
7         err_quit("usage: test01 <hostname/IPaddress> <service/port#>");
8     tfd = Tcp_connect(argv[1], argv[2]);
9     t_snd(tfd, "", 1, T_EXPEDITED);
10    exit(0);
11 }
```

图 C.4 检查 XTI 如何发送 TCP 带外数据的简单测试程序[debug/test01.c]

我们然后在 solaris 2.6 下使用 truss 运行该程序以跟踪系统调用。

```
solaris26 % truss -v putmsg,putpmsg test01 198.69.10.4 discard
```

输出中的最后几行是这样的：

```
putpmsg(3, 0xEFFFF7D4, 0xEFFFF7C0, 0, 0X0004)      = 0
        ctl:    maxlen=8    len=8    buf=0xEFFFF7CC: "\0\0\004\0\0\0\0"
        dat:    maxlen=1    len=1    buf=0x00015318: "\0"
```

它解答了我们的问题。putmsg 的第 3 个参数即优先级带号为 0。控制信息缓冲区中前 4 个字节值为 4 表示 T_EXDATA_REQ(经加速数据请求)，它说明 XTI 函数库把带外数据作为优先级带 0 中的消息发送给提供者。

例子:XTI 带外数据,poll 哪些事件?

有关 TCP 带外数据和 XTI 的另一个问题是:"当等待带外数据时我们应轮询图 6.23 中哪些可能的输入事件:POLLIN、POLLRDNORM、POLLRDBAND 还是 POLLPRI?"这次我们根据图 30.5 的简单 XTI 服务器程序构造出如图 C.5 的测试程序。

```
1 #include    "unpxti.h"
2 int
3 main(int argc, char **argv)
4 {
5     int       listenfd, connfd, n, flags;
6     char      buff[MAXLINE];
7     struct pollfd     fds[1];
8     if (argc == 2)
9         listenfd = Tcp_listen(NULL, argv[1], NULL);
10    else if (argc == 3)
11        listenfd = Tcp_listen(argv[1], argv[2], NULL);
12    else
13        err_quit("usage: daytimetcpsrv01 [ <host> ] <service or port>");
14    connfd = Xti_accept(listenfd, NULL, 0);
15    fds[0].fd = connfd;
16    fds[0].events = POLLIN | POLLRDNORM | POLLRDBAND | POLLPRI;
17    for ( ; ; ) {
18        n = poll(fds, 1, INFTIM);
19        printf("poll returned %d, revents = 0x%x\n", n, fds[0].revents);
20        n = T_rcv(connfd, buff, sizeof(buff), &flags);
21        printf("received %d bytes, flags = %d\n", n, flags);
22    }
23 }
```

图 C.5 检查如何轮询 TCP 带外数据的简单测试程序[debug/test03.c]

我们在输入事件中设置了所有 4 个条件,然后输出返回事件。接着我们调用 t_rcv 并输出所返回的字节数以及返回的标志。

我们需要给这个程序发送普通数据和带外数据,要这样做我们可以在另一台主机上作为客户运行我们的 sock 程序。

```
solaris % sock  -v -i -w 1  -n 3  -U 2  -p 4000  192.9.5.9  8888
connected on 206.62.226.33.34560 to 192.9.5.9.8888
TCP_MAXSEG = 1460
wrote 1 bytes
wrote 1 bytes of urgent data
wrote 1 bytes
wrote 1 bytes
```

其中-v 打开详尽输出标志,-i 使得程序写数据到网络(源客户模式),-w 1 表示一次写 1 个字节,-n 3 表示执行 3 次写操作,-U 2 表示在第 2 次写之前立即写 1 个字节的紧急数据,-p 4000 使得每次写之后停顿 4000 ms(4 秒)。我们先启动测试程序,然后启动 sock 程序,以下是我们的测试程序的输出:

```
solaris26 % test03 8888
poll returned 1, revents = 0x41
received 1 bytes, flags = 0
poll returned 1, revents = 0x41
received 1 bytes, flags = 2
poll returned 1, revents = 0x41
received 1 bytes, flags = 0
poll returned 1, revents = 0x41
received 1 bytes, flags = 0
poll returned 1, revents = 0x41
t_rcv error: An event requires attention
```

每次返回的事件都是 POLLIN 和 POLLRDNORM，它告诉我们 TCP 带外数据的到达并没由 poll 做特殊处理。(我们是在<sys/poll.h>头文件中检查返回事件 0x41 中的两个置 1 位的。编写这样的小测试程序时，以数字形式输出这些值并查看合适的头文件通常要容易些并更快些。)不过 t_rcv 返回带外字节时所返回的标志是 2(T_EXPEDITED)。

C.5 tcpdump 程序

像 tcpdump 这样的工具在网络编程中是非常有用的。这个程序从网络读入分组并输出有关它们的大量信息。它还有只输出与指定准则匹配的那些分组的能力。例如：

```
% tcpdump '(udp and port daytime) or icmp'
```

只输出源或目的端口为 13(daytime 服务)的 UDP 数据报或者 ICMP 分组。如下命令：

```
% tcpdump 'tcp and port 80 and tcp[13:1] & 2 != 0'
```

只输出源或目的端口为 80(HTTP 服务)并且设置了 SYN 标志的 TCP 分节。SYN 标志相应于从 TCP 头部开始处偏移为 13 的字节的值为 2。如下命令：

```
% tcpdump 'tcp and tcp[0:2] > 7000 and tcp[0:2] <= 7005'
```

只输出源端口在 7001 和 7005 之间的 TCP 分节。源端口在 TCP 头部中从偏移为 0 的字节开始，占据 2 个字节。

TCPv1 的附录 A 详细叙述了这个程序的运作细节。

本程序可从 ftp.ee.lbl.gov 获取，并能在许多不同版本的 Unix 上工作。它是由 Van Jacobson、Craig Leres 和 Steven McCanne 编写的。

有些厂家自己提供具有类似功能的程序，例如 Solaris 2.x 提供的是 snoop 程序。tcpdump 的优势在于它在许多版本的 Unix 上都能运行，而能够在异构环境中使用同一个工具而不是给每个环境分别使用一个工具这一点本身就是个大优点。

C.6 netstat 程序

纵贯全书我们已经多次使用了 netstat 程序。这个程序可用于多种目的。

- 输出网络端点状态。5.6 节中我们启动客户和服务器程序后就这样追踪了其端点的状态。
- 输出一台主机上各个接口所属的多播组。-ia 标志是这种输出的通常方式，Solaris 2.x 下则使用-g 标志。
- 使用-s 选项输出各个协议的统计信息。8.13 节中查看 UDP 缺乏流量控制能力时我们给出了这样的例子。
- 使用-r 选项输出路由表，使用-i 选项输出接口信息。我们在 1.9 节给出了这样的例子，在那儿我们用 netstat 发现了我们的网络拓扑。

netstat 还有其他的用法，而且大多数厂家都添加了他们自己的特性。检查一下系统中的手册页面你就会发现。

C.7 lsof 程序

名字 lsof 代表"列出打开的文件"。跟 tcpdump 一样，它也是一个可公开获取的方便调试的工具，并已被移植到许多版本的 Unix 中。

lsof 的通常用法之一是找出哪个进程在指定的 IP 地址或端口上打开了一个套接口。netstat 告诉我们哪些 IP 地址和端口在使用中以及 TCP 连接的状态，但并未标识出进程，lsof 则弥补了这个缺陷。例如，为找出哪个进程在提供 daytime 服务，我们执行如下命令：

```
solaris % lsof -i TCP:daytime
COMMAND  PID  USER  FD  TYPE     DEVICE  SIZE/OFF  INODE NAME
inetd    222  root  15u inet  0xf5a801f8       0t0      TCP *:daytime
```

它告诉我们提供该服务的命令(这个服务是由 inetd 服务器提供的)、它的进程 ID、属主、描述字(15，u 表示打开目的是读-写)、套接口类型、协议控制块地址、文件的大小或偏移(对套接口来说没有意义)、协议类型及名称①。

当启动一个服务器并把它捆绑到众所周知的端口时，如果得到该地址已在使用的出错消息，我们就可以使用 lsof 找出正在使用该端口的进程。

由于 lsof 只汇报已打开的文件，因此不跟某个打开的文件关联的网络端点它是无法汇报的，处于 TIME_WAIT 状态的 TCP 端点就这样。

ftp://vic.cc.purdue.edu/pub/tools/unix/lsof 是这个程序的 URL 地址。它是由 Vic Abell 编写的。

有些厂家提供自己的类似工具，例如 BSD/OS 提供的是 fstat 程序。lsof 的优势跟 tcpdump 一样仍然在于许多版本的 Unix 上它都能工作。

① 译者注：作者在文中说文件的大小或偏移对套接口没有意义，本程序(lsof)的作者却认为套接口的偏移可能会很有用。大多数 Unix 系统上偏移来自文件结构的 f_offset 成员。该结构成员在每次输入或输出文件中字节时都会增长。因此偏移量的变化对套接口来说意味着数据的传送在进行中。

附录D 杂凑的源代码

D.1 unp.h头文件

本书正文中几乎每个程序都包括了如图D.1所示的unp.h头文件。这个文件包括大多数网络程序都需要的所有标准系统头文件以及一些普通的系统头文件。它还定义了诸如MAXLINE等常值以及我们已在正文中定义过的函数(例如readline)及所用到的所有包裹函数的ANSI C函数原型。我们没有给出这些原型。

```
1 /* Our own header.   Tabs are set for 4 spaces, not 8 */
2 #ifndef __unp_h
3 #define __unp_h
4 #include    "../config.h"       /* configuration options for current OS */
5                                 /* "../config.h" is generated by configure */
6 /* If anything changes in the following list of #includes, must change
7   acsite.m4 also, for configure's tests. */
8 #include    <sys/types.h>       /* basic system data types */
9 #include    <sys/socket.h>      /* basic socket definitions */
10 #include   <sys/time.h>        /* timeval{} for select() */
11 #include   <time.h>            /* timespec{} for pselect() */
12 #include   <netinet/in.h>      /* sockaddr_in{} and other Internet defns */
13 #include   <arpa/inet.h>       /* inet(3) functions */
14 #include   <errno.h>
15 #include   <fcntl.h>           /* for nonblocking */
16 #include   <netdb.h>
17 #include   <signal.h>
18 #include   <stdio.h>
19 #include   <stdlib.h>
20 #include   <string.h>
21 #include   <sys/stat.h>        /* for S_xxx file mode constants */
22 #include   <sys/uio.h>         /* for iovec{} and readv/writev */
23 #include   <unistd.h>
24 #include   <sys/wait.h>
25 #include   <sys/un.h>          /* for Unix domain sockets */
26 #ifdef  HAVE_SYS_SELECT_H
27 #include   <sys/select.h>      /* for convenience */
28 #endif
29 #ifdef  HAVE_POLL_H
30 #include   <poll.h>            /* for convenience */
31 #endif
32 #ifdef  HAVE_STRINGS_H
33 #include   <strings.h>         /* for convenience */
```

```
#endif
/* Three headers are normally needed for socket/file ioctl's:
 * <sys/ioctl.h>, <sys/filio.h>, and <sys/sockio.h>.
 */
#ifdef  HAVE_SYS_IOCTL_H
#include    <sys/ioctl.h>
#endif
#ifdef  HAVE_SYS_FILIO_H
#include    <sys/filio.h>
#endif
#ifdef  HAVE_SYS_SOCKIO_H
#include    <sys/sockio.h>
#endif
#ifdef  HAVE_PTHREAD_H
#include    <pthread.h>
#endif
/* OSF/1 actually disables recv() and send() in <sys/socket.h> */
#ifdef __osf__
#undef  recv
#undef  send
#define recv(a,b,c,d)   recvfrom(a,b,c,d,0,0)
#define send(a,b,c,d)   sendto(a,b,c,d,0,0)
#endif
#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff  /* should have been in <netinet/in.h> */
#endif
#ifndef SHUT_RD                 /* these three Posix.1g names are quite new */
#define SHUT_RD     0           /* shutdown for reading */
#define SHUT_WR     1           /* shutdown for writing */
#define SHUT_RDWR   2           /* shutdown for reading and writing */
#endif
#ifndef INET_ADDRSTRLEN
#define INET_ADDRSTRLEN     16  /* "ddd.ddd.ddd.ddd\0"
                                   1234567890123456 */
#endif
/* Define following even if IPv6 not supported, so we can always allocate
   an adequately-sized buffer, without #ifdefs in the code. */
#ifndef INET6_ADDRSTRLEN
#define INET6_ADDRSTRLEN    46  /* max size of IPv6 address string:
              "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx" or
              "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:ddd.ddd.ddd.ddd\0"
               1234567890123456789012345678901234567890123456 */
#endif
/* Define bzero() as a macro if it's not in standard C library. */
#ifndef HAVE_BZERO
#define bzero(ptr,n)        memset(ptr, 0, n)
#endif
/* Older resolvers do not have gethostbyname2() */
```

```
82 #ifndef HAVE_GETHOSTBYNAME2
83 #define gethostbyname2(host,family)     gethostbyname((host))
84 #endif

85 /* The structure returned by recvfrom_flags() */
86 struct in_pktinfo {
87     struct in_addr ipi_addr;     /* dst IPv4 address */
88     int     ipi_ifindex;         /* received interface index */
89 };

90 /* We need the newer CMSG_LEN() and CMSG_SPACE() macros, but few
91    implementations support them today.  These two macros really need
92    an ALIGN() macro, but each implementation does this differently. */
93 #ifndef CMSG_LEN
94 #define CMSG_LEN(size)     (sizeof(struct cmsghdr) + (size))
95 #endif
96 #ifndef CMSG_SPACE
97 #define CMSG_SPACE(size)     (sizeof(struct cmsghdr) + (size))
98 #endif

99 /* Posix.1g requires the SUN_LEN() macro but not all implementations define
100    it (yet).  Note that this 4.4BSD macro works regardless whether there is
101    a length field or not. */
102 #ifndef SUN_LEN
103 #define SUN_LEN(su) \
104     (sizeof(*(su)) - sizeof((su)->sun_path) + strlen((su)->sun_path))
105 #endif

106 /* Posix.1g renames "Unix domain" as "local IPC".
107    But not all systems define AF_LOCAL and PF_LOCAL (yet). */
108 #ifndef AF_LOCAL
109 #define AF_LOCAL     AF_UNIX
110 #endif
111 #ifndef PF_LOCAL
112 #define PF_LOCAL     PF_UNIX
113 #endif

114 /* Posix.1g requires that an #include of <poll.h> define INFTIM, but many
115    systems still define it in <sys/stropts.h>.  We don't want to include
116    all the streams stuff if it's not needed, so we just define INFTIM here.
117    This is the standard value, but there's no guarantee it is -1. */
118 #ifndef INFTIM
119 #define INFTIM          (-1)     /* infinite poll timeout */
120 #ifdef  HAVE_POLL_H
121 #define INFTIM_UNPH             /* tell unpxti.h we defined it */
122 #endif
123 #endif

124 /* Following could be derived from SOMAXCONN in <sys/socket.h>, but many
125    kernels still #define it as 5, while actually supporting many more */
126 #define LISTENQ     1024       /* 2nd argument to listen() */

127 /* Miscellaneous constants */
128 #define MAXLINE     4096       /* max text line length */
129 #define MAXSOCKADDR  128       /* max socket address structure size */
130 #define BUFFSIZE     8192      /* buffer size for reads and writes */
```

```
131 /* Define some port number that can be used for client-servers */
132 #define SERV_PORT        9877   /* TCP and UDP client-servers */
133 #define SERV_PORT_STR    "9877" /* TCP and UDP client-servers */
134 #define UNIXSTR_PATH   "/tmp/unix.str"  /* Unix domain stream cli-serv */
135 #define UNIXDG_PATH    "/tmp/unix.dg"  /* Unix domain datagram cli-serv */

136 /* Following shortens all the type casts of pointer arguments */
137 #define SA   struct sockaddr

138 #define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
139                     /* default file access permissions for new files */
140 #define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
141                     /* default permissions for new directories */

142 typedef void Sigfunc (int);      /* for signal handlers */

143 #define min(a,b)    ((a) < (b) ? (a) : (b))
144 #define max(a,b)    ((a) > (b) ? (a) : (b))

145 #ifndef HAVE_ADDRINFO_STRUCT
146 #include    "../lib/addrinfo.h"
147 #endif

148 #ifndef HAVE_IF_NAMEINDEX_STRUCT
149 struct if_nameindex {
150     unsigned int if_index;     /* 1, 2, ... */
151     char    *if_name;          /* null terminated name: "le0", ... */
152 };
153 #endif

154 #ifndef HAVE_TIMESPEC_STRUCT
155 struct timespec {
156     time_t   tv_sec;           /* seconds */
157     long     tv_nsec;          /* and nanoseconds */
158 };
159 #endif
```

图 D.1 我们的 unp.h 头文件[lib/unp.h]

D.2 config.h 头文件

本书中使用了 GNU autoconf 工具以辅助所有源代码的移植。它可以从 ftp://prep.ai.mit.edu/pub/gnu 获取。这个工具生成一个名为 configure 的 shell 脚本，把软件下载到你的系统后你必须运行它。这个脚本确定由你的 Unix 系统提供的特性：套接口地址结构有长度成员吗？支持多播吗？支持数据链路套接口地址结构吗？等等，最终生成一个名为 config.h 的头文件。它是上节介绍的 unp.h 文件中包括的第一个头文件。图 D.2 给出了 BSD/OS 3.0 上生成的 config.h 文件。

其中从第 1 列开始以 #define 开头的行代表系统提供了的特性。注释掉并且含有 #undef 的行代表系统没有提供的特性。

```
1 /* config.h.  Generated automatically by configure.  */
2 /* Define the following if you have the corresponding header */
3 #define CPU_VENDOR_OS "i386-pc-bsdi3.0"
```

```
 4 /* #undef  HAVE_NETCONFIG_H */          /* <netconfig.h> */
 5 /* #undef  HAVE_NETDIR_H */             /* <netdir.h> */
 6 #define   HAVE_PTHREAD_H 1             /* <pthread.h> */
 7 #define   HAVE_STRINGS_H 1             /* <strings.h> */
 8 /* #undef  HAVE_XTI_INET_H */           /* <xti_inet.h> */
 9 #define   HAVE_SYS_FILIO_H 1           /* <sys/filio.h> */
10 #define     HAVE_SYS_IOCTL_H 1        /* <sys/ioctl.h> */
11 #define     HAVE_SYS_SELECT_H 1       /* <sys/select.h> */
12 #define     HAVE_SYS_SOCKIO_H 1       /* <sys/sockio.h> */
13 #define     HAVE_SYS_SYSCTL_H 1       /* <sys/sysctl.h> */
14 #define     HAVE_SYS_TIME_H 1         /* <sys/time.h> */

15 /* Define if we can include <time.h> with <sys/time.h> */
16 #define TIME_WITH_SYS_TIME 1

17 /* Define the following if the function is provided */
18 #define   HAVE_BZERO 1
19 #define   HAVE_GETHOSTBYNAME2 1
20 /* #undef  HAVE_PSELECT */
21 #define   HAVE_VSNPRINTF 1

22 /* Define the following if the function prototype is in a header */
23 /* #undef HAVE_GETADDRINFO_PROTO */     /* <netdb.h> */
24 /* #undef HAVE_GETNAMEINFO_PROTO */     /* <netdb.h> */
25 #define     HAVE_GETHOSTNAME_PROTO 1    /* <unistd.h> */
26 #define     HAVE_GETRUSAGE_PROTO 1      /* <sys/resource.h> */
27 #define     HAVE_HSTRERROR_PROTO 1      /* <netdb.h> */
28 /* #undef HAVE_IF_NAMETOINDEX_PROTO */  /* <net/if.h> */
29 #define HAVE_INET_ATON_PROTO 1          /* <arpa/inet.h> */
30 #define HAVE_INET_PTON_PROTO 1          /* <arpa/inet.h> */
31 /* #undef HAVE_ISFDTYPE_PROTO */        /* <sys/stat.h> */
32 /* #undef HAVE_PSELECT_PROTO */         /* <sys/select.h> */
33 #define HAVE_SNPRINTF_PROTO 1           /* <stdio.h> */
34 /* #undef HAVE_SOCKATMARK_PROTO */      /* <sys/socket.h> */

35 /* Define the following if the structure is defined. */
36 /* #undef HAVE_ADDRINFO_STRUCT */       /* <netdb.h> */
37 /* #undef HAVE_IF_NAMEINDEX_STRUCT */   /* <net/if.h> */
38 #define HAVE_SOCKADDR_DL_STRUCT 1       /* <net/if_dl.h> */
39 #define HAVE_TIMESPEC_STRUCT 1          /* <time.h> */

40 /* Define the following if feature is provided. */
41 #define HAVE_SOCKADDR_SA_LEN 1     /* sockaddr{} has sa_len member */
42 #define HAVE_MSGHDR_MSG_CONTROL 1 /* msghdr{} has msg_control member */

43 /* Names of XTI devices for TCP and UDP */
44 /* #undef HAVE_DEV_TCP */     /* most XTI have devices here */
45 /* #undef HAVE_DEV_XTI_TCP */     /* AIX has them here */
46 /* #undef HAVE_DEV_STREAMS_XTISO_TCP */  /* OSF 3.2 has them here */

47 /* Define the following to the appropriate datatype, if necessary */
48 /* #undef int8_t */              /* <sys/types.h> */
49 /* #undef int16_t */             /* <sys/types.h> */
50 /* #undef int32_t */             /* <sys/types.h> */
51 #define uint8_t unsigned char     /* <sys/types.h> */
52 #define uint16_t unsigned short   /* <sys/types.h> */
53 #define uint32_t unsigned int     /* <sys/types.h> */
```

```
54 /* #undef size_t */                /* <sys/types.h> */
55 /* #undef ssize_t */               /* <sys/types.h> */
56 /* socklen_t should be typedef'd as uint32_t, but configure defines it
57    to be an unsigned int, as it is needed early in the compile process,
58    sometimes before some implementations define uint32_t. */
59 #define socklen_t unsigned int              /* <sys/socket.h> */
60 #define sa_family_t SA_FAMILY_T             /* <sys/socket.h> */
61 #define SA_FAMILY_T uint8_t

62 #define t_scalar_t int32_t                  /* <xti.h> */
63 #define t_uscalar_t uint32_t                /* <xti.h> */

64 /* Define the following, if system suports the feature */
65 #define IPV4 1                  /* IPv4, uppercase V name */
66 #define IPv4 1                  /* IPv4, lowercase v name, just in case */
67 /* #undef IPV6 */               /* IPv6, uppercase V name */
68 /* #undef IPv6 */               /* IPv6, lowercase v name, just in case */
69 #define UNIXDOMAIN 1            /* Unix domain sockets */
70 #define UNIXdomain 1            /* Unix domain sockets */
71 #define MCAST 1                 /* multicasting support */
```

图 D.2 BSD/OS 3.0 上的 config.h 头文件[i368-pc-bsdi3.0/config.h]

D.3 unpxti.h 头文件

我们所有的 XTI 程序都包括一个名为 unpxti.h 的头文件，它如图 D.3 所示。跟我们在 D.1 节列出 unp.h 文件一样，我们在这儿也略掉了所有的函数原型。

```
1 #ifndef __unp_xti_h
2 #define __unp_xti_h

3 #include    "unp.h"

4 #include    <xti.h>
5 #ifdef HAVE_XTI_INET_H
6 #include    <xti_inet.h>
7 #endif
8 #ifdef HAVE_NETCONFIG_H
9 #include    <netconfig.h>
10 #endif
11 #ifdef HAVE_NETDIR_H
12 #include    <netdir.h>
13 #endif

14 #ifdef INFTIM_UNPH
15 #undef INFTIM           /* was not in <poll.h>, undef for <stropts.h> */
16 #endif

17 #include    <stropts.h>

18 /* Provide compatibility with the new names prepended with T_
19    in XNS Issue 5, which are not in Posix.1g. */

20 #ifndef T_INET_TCP
21 #define T_INET_TCP      INET_TCP
22 #endif
```

```
23 #ifndef T_INET_UDP
24 #define T_INET_UDP      INET_UDP
25 #endif
26 #ifndef T_INET_IP
27 #define T_INET_IP       INET_IP
28 #endif
29 #ifndef T_TCP_NODELAY
30 #define T_TCP_NODELAY    TCP_NODELAY
31 #endif
32 #ifndef T_TCP_MAXSEG
33 #define T_TCP_MAXSEG     TCP_MAXSEG
34 #endif
35 #ifndef T_TCP_KEEPALIVE
36 #define T_TCP_KEEPALIVE TCP_KEEPALIVE
37 #endif
38 #ifndef T_UDP_CHECKSUM
39 #define T_UDP_CHECKSUM   UDP_CHECKSUM
40 #endif
41 #ifndef T_IP_OPTIONS
42 #define T_IP_OPTIONS     IP_OPTIONS
43 #endif
44 #ifndef T_IP_TOS
45 #define T_IP_TOS         IP_TOS
46 #endif
47 #ifndef T_IP_TTL
48 #define T_IP_TTL         IP_TTL
49 #endif
50 #ifndef T_IP_REUSEADDR
51 #define T_IP_REUSEADDR   IP_REUSEADDR
52 #endif
53 #ifndef T_IP_DONTROUTE
54 #define T_IP_DONTROUTE   IP_DONTROUTE
55 #endif
56 #ifndef T_IP_BROADCAST
57 #define T_IP_BROADCAST   IP_BROADCAST
58 #endif
59 /* Define the appropriate devices for t_open(). */
60 #ifdef  HAVE_DEV_TCP
61 #define XTI_TCP      "/dev/tcp"
62 #define XTI_UDP      "/dev/udp"
63 #endif
64 #ifdef  HAVE_DEV_XTI_TCP
65 #define XTI_TCP      "/dev/xti/tcp"
66 #define XTI_UDP      "/dev/xti/udp"
67 #endif
68 #ifdef  HAVE_DEV_STREAMS_XTISO_TCP
69 #define XTI_TCP      "/dev/streams/xtiso/tcp+"   /* + for XPG4 */
70 #define XTI_UDP      "/dev/streams/xtiso/udp+"   /* + for XPG4 */
```

```
71 #endif
72          /* device to t_open() for t_accept(); set by tcp_listen() */
73 extern char xti_serv_dev[];
```

图 D.3 用于 XTI 程序的 unpxti.h 头文件[libxti/unpxti.h]

D.4 标准错误处理函数

我们定义了自己的一组错误处理函数，它们用在整本书中以处理错误情况。定义一组自己的错误处理函数的原因是我们可以用一行简单的 C 代码写出错误处理过程，就像如下所示：

```
if (error condition)
    err_sys(printf format with any number of arguments);
```

而不是如下那样用多个行：

```
if(error condition) {
    char   buff[200];
    snprintf(buff,sizeof(buff),printf format with any number of arguments);
    perror(buff);
    exit(1);
}
```

我们的错误处理函数使用来自 ANSI C 的可变长度参数表机制。具体细节参见[Kernighan and Ritchie 1988]的 7.3 节。

图 D.4 列出了各个错误处理函数之间的差异。如果全局整数 daemon_proc 为非零的话，错误消息按指定的级别传递给 syslog；否则错误消息输出到标准错误。

函数	strerror(errno)?	结束语句	syslog 级别
err_dump	是	abort();	LOG_ERR
err_msg	否	return;	LOG_INFO
err_quit	否	exit(1);	LOG_ERR
err_ret	是	return;	LOG_INFO
err_sys	是	exit(1);	LOG_ERR
err_xti	是	exit(1);	LOG_ERR
err_xti_ret	是	return;	LOG_INFO

图 D.4 标准错误处理函数汇总

图 D.5 给出了图 D.4 中的前 5 个函数。

```
1 #include    "unp.h"
2 #include    <stdarg.h>          /* ANSI C header file */
3 #include    <syslog.h>          /* for syslog() */
4 int         daemon_proc;        /* set nonzero by daemon_init() */
5 static void err_doit(int, int, const char *, va_list);
6 /* Nonfatal error related to a system call.
```

```
 7  * Print a message and return.  */
 8 void
 9 err_ret(const char *fmt, ...)
10 {
11     va_list ap;
12     va_start(ap, fmt);
13     err_doit(1, LOG_INFO, fmt, ap);
14     va_end(ap);
15     return;
16 }
17 /* Fatal error related to a system call.
18  * Print a message and terminate.  */
19 void
20 err_sys(const char *fmt, ...)
21 {
22     va_list ap;
23     va_start(ap, fmt);
24     err_doit(1, LOG_ERR, fmt, ap);
25     va_end(ap);
26     exit(1);
27 }
28 /* Fatal error related to a system call.
29  * Print a message, dump core, and terminate.  */
30 void
31 err_dump(const char *fmt, ...)
32 {
33     va_list ap;
34     va_start(ap, fmt);
35     err_doit(1, LOG_ERR, fmt, ap);
36     va_end(ap);
37     abort();            /* dump core and terminate */
38     exit(1);            /* shouldn't get here */
38 }
40 /* Nonfatal error unrelated to a system call.
41  * Print a message and return.  */
42 void
43 err_msg(const char *fmt, ...)
44 {
45     va_list         ap;
46     va_start(ap, fmt);
47     err_doit(0, LOG_INFO, fmt, ap);
48     va_end(ap);
49     return;
50 }
51 /* Fatal error unrelated to a system call.
52  * Print a message and terminate.  */
53 void
```

```
54 err_quit(const char *fmt, ...)
55 {
56     va_list ap;
57     va_start(ap, fmt);
58     err_doit(0, LOG_ERR, fmt, ap);
59     va_end(ap);
60     exit(1);
61 }
62 /* Print a message and return to caller.
63  * Caller specifies "errnoflag" and "level". */
64 static void
65 err_doit(int errnoflag, int level, const char *fmt, va_list ap)
66 {
67     int        errno_save, n;
68     char   buf[MAXLINE+1];
69     errno_save = errno;          /* value caller might want printed */
70 #ifdef     HAVE_VSNPRINTF
71     vsnprintf(buf, MAXLINE, fmt, ap);      /* this is safe */
72 #else
73     vsprintf(buf, fmt, ap);                /* this is not safe */
74 #endif
75     n = strlen(buf);
76     if (errnoflag)
77         snprintf(buf+n, MAXLINE-n, ": %s", strerror(errno_save));
78     strcat(buf, "\n");
79     if (daemon_proc) {
80         syslog(level, buf);
81     } else {
82         fflush(stdout);        /* in case stdout and stderr are the same */
83         fputs(buf, stderr);
84         fflush(stderr);
85     }
86     return;
87 }
```

图 D.5　我们的标准错误处理函数[lib/error.c]

附录E 部分习题解答

第1章习题答案

1.3 在AIX下我们得到

```
aix % daytimetcpcli 206.62.226.33
socket error: Addr family not supported by protocol
```

要找出有关这个错误的详细信息，我们首先在<sys/errno.h>头文件中使用grep查找串Addr。

```
aix % grep Addr /usr/include/sys/errno.h
#define EAFNOSUPPORT 66 /* Address family not supported by protocol family */
#define EADDRINUSE    67 /* Address already in use */
```

第一个就是由socket返回的errno值。我们然后查看手册页面：

```
aix % man socket
```

大多数手册页面在近结束处形如"Errors"的标题下给出额外的信息，不过有些简洁。

1.4 我们把第一个声明改成：

```
int sockfd,n,counter = 0;
```

再加上语句：

```
counter++;
```

作为while循环的第一条语句。最后在结束前加上语句：

```
printf("counter = %d \n", counter);
```

所输出的值总是1。

1.5 我们声明一个名为i的int变量，再把对write的调用改为：

```
for (i = 0; i < strlen(buff); i++);
    Write(connfd,&buff[i],1);
```

其结果随客户主机和服务器主机而定。如果客户和服务器在同一台主机上，计数器值通常就是1，它意味着即使服务器调用了26次write，数据也仅由一次read返回。但是如果客户运行在Solaris 2.5.1上而服务器运行在BSD/OS 3.0上，计数器值通常就是2。要是检查以太网上的分组的话，我们发现第1个字符自成一个分组发送，剩下的25个字符则包含在下一个分组内。(7.9节中我们就Nagle算法的讨论解释了这种行为的原因。)如果客户运行在BSD/OS 3.0上而服务器运行在Solaris 2.5.1上，计数器值就是26。要是检查分组的话，我们发现每个字符自成一个分组发送。

本例子的目的是说明不同的 TCP 对数据做不同的处理，我们的应用程序必须准备好以字节流读入数据，直至到达数据流尾为止。

第 2 章习题答案

2.1　所有 RFC 都可以通过电子邮件、匿名 FTP 或 WWW 免费获取。起始点之一是 http://www.ietf.org。目录 ftp://ftp.isi.edu/in-notes 是一个存放 RFC 的位置。你可以从取得当前 RFC 索引开始，它通常是文件 rfc-index.txt。查看一下标题为“Assigned Numbers(已分配号)”的 RFC 1340 表项，我们注意到它已被 RFC 1700 取代而过时了。尽管 RFC 1700 现在还在编写，但等你读它时它可能也过时了。从这些过时的 RFC 往前走就能找到当前的(也就是说编号最高的)“Assigned Numbers”RFC。

这个 RFC 中标题为“Version Numbers(版本号)”一节标识了各种 IP 版本号。版本 0 是保留的，版本 1～3 没有分配，版本 5 则是网际流协议(Internet Stream protocol)。

2.2　如果用某种形式的编辑器搜索 RFC 索引(参见上一个习题的解答)查找术语“Stream”，我们会发现 RFC 1819 定义了网际流协议的第 2 版。无论什么时候想查找可能由某个 RFC 涵盖的信息，别忘了搜索 RFC 索引。

2.3　使用 IPv4 这样生成的是 576 字节的 IP 数据报(其中 IPv4 头部占用 20 字节，TCP 头部也占用 20 字节)，这是 IPv4 最小的重组缓冲区大小。

2.4　本例中是服务器而不是客户执行主动关闭操作。

2.5　令牌环网上的主机不能发送超过 1460 字节的数据，因为它接收到的 MSS 是 1460。以太网上的主机可以发送最多 4096 字节的数据，但为避免分片，它不会超过外出接口(即以太网)的 MTU。TCP 不能超过由另外一端声称的 MSS，但低于这个数量总是可以发送的。

2.6　Assigned Numbers RFC 中“Protocol Numbers(协议号)”一节给出 OSPF 的协议号为 89。

第 3 章习题答案

3.1　C 中函数不能改变以值传递的参数的值。要让被调用的函数修改由调用者传入的值，调用者必须传递指向待修改值的指针。

3.2　指针必须按所读或写的字节数增加，但 C 并不允许 void 指针的增加(因为编译器并不知道所指向的数据类型)。

第 4 章习题答案

4.1　看一下除 INADDR_ANY(它的各位全为 0)和 INADDR_NONE(它的各位全为 1)外以 INADDR_开头的各个常值的定义。例如 D 类多播地址 INADDR_MAX_LOCAL_GROUP 定义成 0xe00000ff，其注释为“224.0.0.255”，它显然是按主机字节序定义的。

4.2 下面是在调用 connect 后新添加的若干行：

```
len = sizeof(cliaddr);
Getsockname(sockfd, (SA *) &cliaddr, &len);
printf("local addr: %s\n",
    Sock_ntop((SA *) &cliaddr,len));
```

这要求声明 len 为 socklen_t 变量，cliaddr 为 struct sockaddr_in 变量。注意 getsockname 的值-结果参数(len)必须在调用之前初始化成由第 2 个参数所指向变量的大小。涉及值-结果参数的最常见编程错误就是忘记了这样的初始化。

4.3 子进程调用 close 时访问计数从 2 减为 1，因此不会向客户发送 FIN。以后当父进程调用 close 时，访问计数减为 0，于是发送 FIN。

4.4 accept 返回 EINVAL，因为它的第 1 个参数不是一个监听套接口描述字。

4.5 不调用 bind 的话，调用 listen 分配给监听套接口的是一个临时端口。

第5章习题答案

5.1 TIME_WAIT 状态的持续时间应该在 1 分钟到 4 分钟之间，因此 MSL 在 30 秒到 2 分钟之间。

5.2 我们的客户-服务器程序在使用二进制文件做标准输入时并不工作。假定该文件的前 3 个字节为二进制 1、二进制 0 和一个换行符。图 5.5 中对 fgets 的调用最多读入 MAXLINE-1 个字符，它也可能在碰到换行符或到达文件尾时结束。本例中它将读入前 3 个字符，然后以一个空字节结束串。不过我们在图 5.5 中调用 strlen 时返回的是 1，因为它只计到第 1 个空字节。因此第 1 个字节发送给服务器后，服务器阻塞在 readline 调用上，等待一个换行符。客户也阻塞在等待服务器的应答上。这种情况称为死锁(deadlock)；两个进程都阻塞在等待因对方原因而永远不会到达的事件上。这儿的问题是 fgets 用空字节来表示所返回数据的结尾，因此它读入的数据不能含有任何空字节。

5.3 Telnet 把输入行转换成 NVT ASCII(TCPv1 的 26.4 节)，它意味着以 CR(回车符)后跟 LF(换行符)的 2 字节序列终止每一行。而我们的客户程序只加一个换行符。尽管如此，我们还是可以使用 Telnet 客户程序跟我们的服务器通信的，因为我们的服务器把每个字符都反射回来，它包括每个换行符之前的 CR。

5.4 没有，连接终止序列的最后两个分节没有发送。我们杀掉服务器子进程后(输入“another line”处)，客户向服务器发送数据导致服务器的 TCP 响应一个 RST。这个 RST 使连接夭折并防止连接的服务器端(执行主动关闭一端)经历 TIME_WAIT 状态。

5.5 什么都不变化，因为在服务器主机上新启动的服务器进程创建的是监听套接口，它等待新的连接请求的到达。我们在第 3 步发送的是去往一个 ESTABLISHED 状态 TCP 连接的数据分节。而新启动的具有监听套接口的服务器绝不会看到这些数据分节，因此服务器主机的 TCP 对它们的响应仍然是 RST。

5.6　图 E.1 给出了这个程序。在 AIX 上运行它产生如下输出：

```
aix % tsigpipe 206.62.226.34
SIGPIPE received
write error: Broken pipe
```

其中第 1 个 2 秒的 sleep 调用是让 daytime 服务器发送应答并关闭其所在连接端。第 1 个 write 调用发送数据分节到服务器，而它的响应则是 RST(因为 daytime 服务器已经完全关闭了它的套接口)。注意 TCP 允许我们继续写已接收到 FIN 的套接口。第 2 个 sleep 是让客户接收到服务器的 RST，于是第 2 个 write 引发 SIGPIPE 信号。从信号处理程序返回后，write 返回一个 EPIPE 的错误。

```
 1 #include    "unp.h"
 2 void
 3 sig_pipe(int signo)
 4 {
 5     printf("SIGPIPE received\n");
 6     return;
 7 }
 8 int
 9 main(int argc, char * * argv)
10 {
11     int    sockfd;
12     struct sockaddr_in    servaddr;
13     if (argc ! = 2)
14         err_quit("usage: tcpcli <IPaddress>");
15     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
16     bzero(&servaddr, sizeof(servaddr));
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_port = htons(13);          /* daytime server */
19     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
20     Signal(SIGPIPE, sig_pipe);
21     Connect(sockfd, (SA * ) &servaddr, sizeof(servaddr));
22     sleep(2);
23     Write(sockfd, "hello", 5);
24     sleep(2);
25     Write(sockfd, "world", 5);
26     exit(0);
27 }
```

图 E.1　生成 SIGPIPE[tcpcliserv/tsigpipe.c]

5.7　如果服务器主机支持弱端系统模型(weak end system model，我们在 8.8 节叙述过)，那么一切正常。也就是说即使目的 IP 地址是右端数据链路的地址，服务器主机也会接受到达左端数据链路的外来 IP 数据报(这种情况下它含有一个 TCP 分节)。我们可以这样测试：在主机 bsdi(图 1.16)上运行服务器程序，在主机 solaris 上运行客户程序，但给它指定的是服务器主机的另外一个 IP 地址。连接建立后，如果我们在服务器主机上运行 netstat，我们将看到该连接的本地 IP 地址是取自

客户 SYN 的目的 IP 地址，而不是 SYN 到达的数据链路的 IP 地址（这跟我们在 4.4 节提及的一样）。

5.8 我们的客户运行在小端字节序的 Intel 系统上，那儿 32 位整数值 1 按图 E.2 格式存放。

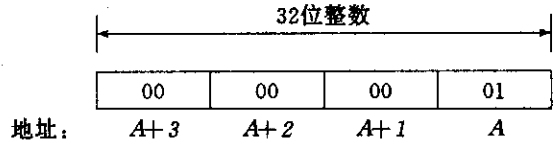


图 E.2 32 位整数值 1 的小端字节序格式表示

这 4 个字节按 A、A+1、A+2 和 A+3 的顺序通过套接口发送，然后以如图 E.3 所示的大端字节序格式存放。

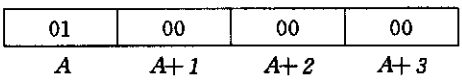


图 E.3 来自图 E.2 的 32 位整数以大端字节序格式表示

值 0x01000000 解释成 16,777,216。类似地，由客户发送的整数 2 将被解释成服务器上的 0x02000000 即 33,554,432。这两个整数的和是 50,331,648 即 0x03000000。当服务器上的这个大端字节序值发送给客户时，它被解释成整数 3。

但 32 位整数值-22 在小端字节序系统上是如图 E.4 表示的，采用的是负数的 2 补码表示法。

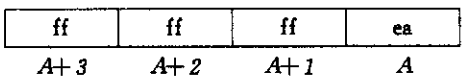


图 E.4 32 位整数值-22 的小端字节序格式表示

它在大端字节序服务器主机上被解释成 0xeaffffff 即－352,321,537。类似地，－77 的小端字节序表示是 0xffffffb3，但它在大端字节序服务器主机上表示成 0xb3ffffff 即－1,275,068,417。服务器上的两个整数相加的结果是 0x9effffe 即－1,627,389,954。这个大端字节序的值通过套接口发送给客户后以小端字节序解释的值是 0xfeffff9e 即－16,777,314，它就是我们的例子所输出的值。

5.9 技术是正确的（把二进制值转换成网络字节顺序），但不能使用 htonl 和 ntohl 这两个函数。即使这些函数中的 l 曾表示“long”，它们也只能在 32 位整数上操作（3.4 节）。64 位系统上 long 可能占据 64 位，这时这两个函数将不能正确工作。有人也许定义 hton64 和 ntoh64 这两个函数来解决该问题，但它们在使用 32 位代表 long 的系统上又不能工作。

5.10 第一种情况下服务器永远阻塞在图 5.20 的 readn 上，因为客户发送 2 个 32 位值，但服务器等待的是 2 个 64 位值。这两台主机之间对换客户和服务器导致客

户发送 2 个 64 位值，但服务器只读入第 1 个 64 位，并把解释成 2 个 32 位值。第 2 个 64 位值仍然在服务器的套接口接收缓冲区上。服务器往回写 1 个 32 位值，于是客户永远阻塞在图 5.19 的 readn 上，等待读入 2 个 64 位值。

5.11 IP 路由函数查看目的 IP 地址(服务器的 IP 地址)，搜索路由表确定外出接口和下一跳主机(ICPv1 第 9 章)。外出接口的主 IP 地址用作源 IP 地址，前提是套接口尚未捆绑在某个本地 IP 地址上。

第 6 章习题答案

6.1 这个整数数组包含在一个结构中，而 C 是允许结构跨越等号赋值的。

6.2 如果 select 告诉我们套接口可写，该套接口的发送缓冲区就有 8192 字节的空间，但是我们以 8193 字节的缓冲区长度对这个阻塞式套接口调用 write 时，write 会阻塞，给最后 1 个字节等待空间。对阻塞式套接口的读操作只要有数据总会返回一个短计数值，相反，对它的写操作除所有数据都被内核接受外会一直阻塞。因此使用 select 来测试是否可写时，我们必须把套接口设成非阻塞式以避免阻塞。

6.3 如果两个描述字都可读，那么只执行第 1 个测试即对套接口描述字的测试。不过这样并没有导致客户不能工作；它只是降低了效率而已。这就是说，如果 select 的返回值说明两个描述字都可读，那么第 1 个 if 语句为真，导致客户从套接口 readline 并 fputs 到标准输出。下一个 if 语句跳过(因为我们添加的 else 语句的缘故)，但 select 接着再次调用并马上发现标准输入是可读的，从而立即返回。这里的关键概念是消除“标准输入可读”条件的不是 select 的返回，而是从这个描述字真正地读。

6.4 使用 getrlimit 函数取得 RLIMIT_NOFILE 资源的值，然后调用 setrlimit 把当前软限制(rlim_cur)设定成硬限制(rlim_max)。例如在 Solaris 2.5 下软限制是 64，但任何进程都可以把它增加到缺省的硬限制 1024。

getrlimit 和 setrlimit 不属于 Posix.1，但在 Unix 98 中却是必需的。

6.5 服务器应用进程持续向客户发送数据，客户 TCP 确认后扔掉它们。

6.6 shutdown 总是发送 FIN，而 close 只在描述字访问计数为 1 的条件下才发送 FIN。

6.7 readline 返回一个错误，我们的 Readline 包裹函数则终止服务器。服务器必须更为健壮。注意我们在图 6.26 中处理了这种情况，尽管即便这样的代码还是不够。考虑客户和服务器间的连接丢失以及服务器的某个响应超时的情况。所返回的错误可能是 ETIMEDOUT。

通常服务器不应该因为这样的原因而夭折。它应该记录错误，关闭套接口，然后继续服务其他客户。对于象这样由单个进程来应付所有客户的服务器来说，简单夭折的错误处理类型是难以接受的。但是如果服务器是由一个子进程来应付仅仅一个客户，那么某个子进程的夭折并不会影响父进程(我们假定它处理所有的新连接并派生子进程)和服务其他客户的任何其他子进程。

第7章习题答案

7.2 图E.5给出了本习题的一种解答。我们去掉了输出由服务器返回的数据串的语句，因为那个值并不是必需的。

```
1 #include    "unp.h"
2 #include    <netinet/tcp.h>        /* for TCP_MAXSEG */
3 int
4 main(int argc, char **argv)
5 {
6     int    sockfd, rcvbuf, mss;
7     socklen_t    len;
8     struct sockaddr_in    servaddr;
9     if (argc != 2)
10        err_quit("usage: rcvbuf <IPaddress>");
11    sockfd = Socket(AF_INET, SOCK_STREAM, 0);
12    len = sizeof(rcvbuf);
13    Getsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &rcvbuf, &len);
14    len = sizeof(mss);
15    Getsockopt(sockfd, IPPROTO_TCP, TCP_MAXSEG, &mss, &len);
16    printf("defaults: SO_RCVBUF = %d, MSS = %d\n", rcvbuf, mss);
17    bzero(&servaddr, sizeof(servaddr));
18    servaddr.sin_family = AF_INET;
19    servaddr.sin_port = htons(13);          /* daytime server */
20    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
21    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
22    len = sizeof(rcvbuf);
23    Getsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &rcvbuf, &len);
24    len = sizeof(mss);
25    Getsockopt(sockfd, IPPROTO_TCP, TCP_MAXSEG, &mss, &len);
26    printf("after connect: SO_RCVBUF = %d, MSS = %d\n", rcvbuf, mss);
27    exit(0);
28 }
```

图E.5 在连接建立前后输出套接口接收缓冲区的大小以及MSS[sockopt/rcvbuf.c]

首先声明这个程序没有"唯一正确"的输出，其结果随系统而变化。有些系统(例如Solaris 2.5.1和更早版本)返回的套接口缓冲区大小总是0，使得我们无法跨连接检查发生了什么事。

至于MSS，它在connect之前输出的是实现缺省值(通常是536或512)，在connect之后输出的则取决于来自对方的可能有的MSS选项。例如在本地以太网上connect后的值可能是1460。然而如果connect的服务器在某个远程网络上，那么MSS可能类似于缺省值，除非你的系统支持路径MTU发现功能。如果可能的话，在程序运行时运行一个像tcpdump(c.5节)的工具，查看来自对方的SYN分节上的真正MSS选项。

至于套接口接收缓冲区的大小，许多实现在连接建立后把它向上舍入成MSS的

倍数。连接建立后检查套接口接收缓冲区大小的另一种方法是使用像 tcpdump 的工具，查看 TCP 的通告窗口(advertised window)。

7.3 分配一个名为 ling 的 linger 结构并把它初始化成：

```
str_cli(stdin,sockfd);

ling.l_onoff = 1;
ling.l_linger = 0;
Setsockopt(sockfd,SOL_SOCKET,SO_LINGER,&ling,sizeof(ling));

exit(0);
```

这应该使得客户 TCP 以一个 RST 而不是通常的 4 分节交换终止连接。服务器子进程调用 readline 返回 ECONNRESET 错误，所输出的消息是：

```
readline error: Connection reset by peer
```

客户即使主动关闭也不应该经历 TIME_WAIT 状态。

7.4 第一个客户调用 setsockopt、bind 和 connect。如果它在调用 bind 和 connect 之间第二个客户调用了 bind，那么 TCP 会返回 EADDRINUSE 的错误给第二个客户。但是一旦第一个客户连接到对方，第二个客户的 bind 就能工作，因为第一个客户的套接口已经连接上。处理这种竞争的唯一办法是让第二个客户在碰到 EADDRINUSE 的错误后继续 bind 调用的尝试，而不是一碰到错误就马上放弃。(Posix.1g 标准中指出了这种竞争状态。)

7.5 我们在不支持多播的一台主机(UnixWare 2.1.2)上运行这个程序。

```
unixware % sock -s 9999 &                          以通配地址启动第一个服务器
[1]     29697
unixware % sock -s 206.62.226.37 9999              不使用-A 尝试启动第二个服务器
can't bind local address: Address already in use
unixware % sock -s -A 206.62.226.37 9999&          使用-A 再次尝试;成功
[2]     29699
unixware % sock -s -A 127.0.0.1 9999 &             使用-A 启动第三个服务器;成功
[3]     29700
unixware % netstat -na | grep 9999
tcp        0      0  127.0.0.1.999          *.*          LISTEN
tcp        0      0  206.62.226.37.9999     *.*          LISTEN
tcp        0      0  *.9999                 *.*          LISTEN
```

7.6 我们首先在不支持多播的一台主机(UnixWare 2.1.2)上尝试。

```
unixware % sock -s -u -A 206.62.226.37 8888 &      第一个启动
[4]     29707
unixware % sock -s -u -A 206.62.226.37 8888
can't bind local address: Address already in use   不能启动第二个
```

我们给这两个尝试都指定了 SO_REUSEADDR 选项，但它不起作用。

我们现在在支持多播但不支持 SO_REUSEPORT 选项的一台主机(Solaris 2.6)上尝试。

```
solaris26 % sock -s -u 8888 &                      第一个启动
[1]     1135
solaris26 % sock -s -u 8888
```

```
can't bind local address: Address already in use
solaris26 % sock -s -u -A 8888 &          使用-A尝试第二个;成功
solaris26 % netstat -na | grep 8888       而且我们看到重复的捆绑
      *.8888                 Idle
      *.8888                 Idle
```

这样的系统上第一个 bind 不必指定 SO_REUSEADDR，但第二个 bind 则必须。最后我们在既支持多播又支持 SO_REUSEPORT 选项的 BSD/OS 3.0 上进行尝试。我们先给两个服务器尝试 SO_REUSEADDR 选项，但它不起作用。

```
bsdi % sock -u -s -A 7777 &
[1]    17610
bsdi % sock -u -s -A 7777
can't bind local address: Address already in use
```

接下来我们只给第二个服务器尝试 SO_REUSEPORT 选项。这也不起作用，因为完全重复的捆绑要求共享捆绑的所有套接口都使用该选项。

```
bsdi % sock -u -s 8888 &
[1]    17612
bsdi % sock -u -s -T 8888
can't bind local address: Address already in use
```

最后我们给两个服务器都指定 SO_REUSEPORT 选项，它是起作用的。

```
bsdi % sock -u -s -T 9999 &
[1]    17614
bsdi % sock -u -s -T 9999 &
[2]    17615
bsdi % netstat -na | grep 9999
udp        0      0    *.9999                 *.*
udp        0      0    *.9999                 *.*
```

7.7 它不起任何作用，因为 Ping 使用 ICMP 套接口，而 SO_DEBUG 套接口选项只影响 TCP 套接口。SO_DEBUG 这个套接口选项的说明总是有些笼通，例如“这个选项打开各个协议层的调试”，但实现该选项的唯一协议层只是 TCP。

7.8 图 E.6 给出了时间线。

7.9 设置 TCP_NODELAY 套接口选项导致来自第 2 个 write 的数据被立即发送，即使连接的承受能力比较小也这样。这种情况如图 E.7 所示。本例子中的总时间只稍微超过 150ms。

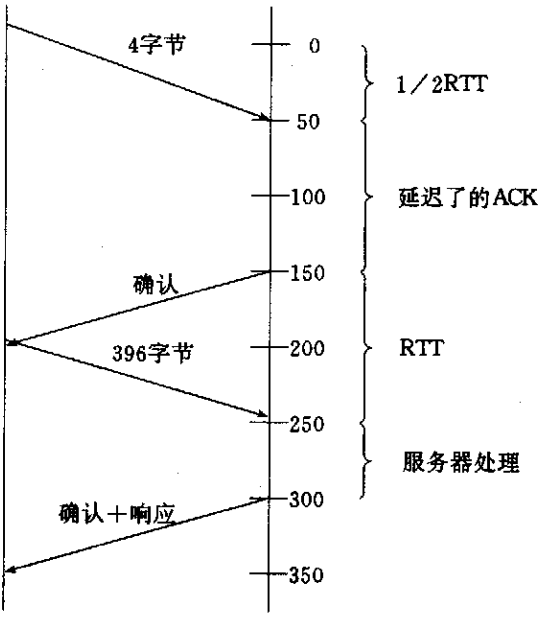


图 E.6　带延迟了的 ACK 的 Nagle 算法交互情况

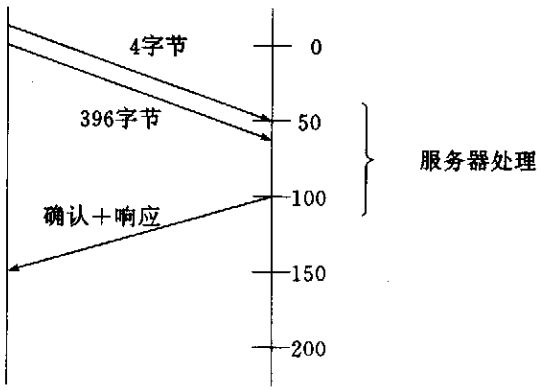


图 E.7　通过设置 TCP_NODELAY 套接口选项避免 Nagle 算法

7.10　这种办法的优点是减少了分组的个数，它如图 E.8 所示。

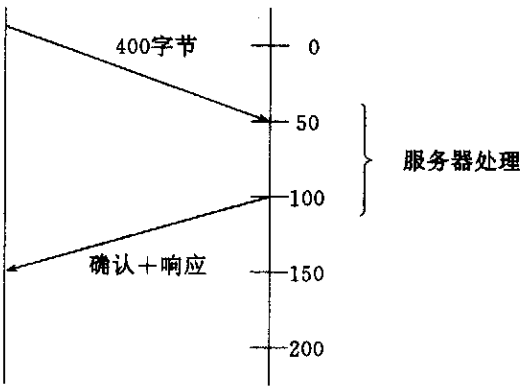


图 E.8　使用 writev 代替设置 TCP_NODELAY 套接口选项

7.11 4.2.3.2节声称"延迟必须低于0.5秒,在完全大小分节传送的流上每隔一个分节应该至少有一个ACK。"源自Berkeley的实现ACK的延迟最多为200ms(TCPv2第821页)。

7.12 图5.2中的服务器父进程大部分时间花在调用accept的阻塞上,图5.3中的子进程则大部分时间花在调用read的阻塞上,它是由readline调用的。保持存活选项对监听套接口不起作用,因此要是客户主机崩溃的话父进程不受影响。子进程的read将返回ETIMEDOUT错误,它在跨越连接的最后一次数据交换后约2小时发生。

7.13 图5.5中的客户大部分时间花在调用fgets的阻塞上,fgets本身则阻塞的在标准I/O例程库中对标准输入的某种类型读操作上。当跨越连接的最后一次数据交换后约2小时保持存活定时器超时并且所有保持存活侦探分组都探测不到来自服务器的响应时,套接口的待处理错误设置成ETIMEDOUT。然而客户阻塞在对标准输入的fgets调用上,因此在对套接口执行读或写操作前,它是看不到这个错误的。这就是我们在第6章把图5.5改成使用select的原因之一。

7.14 客户大部分时间花在调用select的阻塞上,一旦待处理错误设置成ETIMEDOUT(如上题的解答所述),select会立即返回套接口可读的信息。

7.15 只交换2个而不是4个。两个系统的定时器精确同步的可能性非常低;因此一端的定时器会比另一端略早一点超时。首先超时一端发送保持存活侦探分组,导致另一端确认这个分组。然而保持存活侦探分组的接收导致时钟略慢的主机把定时器重置成2小时。

7.16 最初的套接口API并没有listen函数。相反,socket函数的第4个参数含有套接口选项,而SO_ACCEPTCON就是用来指定监听套接口的。加了listen函数后,这个选项还是保留着,不过现在只是由内核来设置(TCPv2第456页)。

第8章习题答案

8.1 是的。read返回4096字节的数据,recvfrom则返回2048字节(2个数据报中的第1个)。不管应用请求多大,recvfrom决不会返回多于1个数据报的数据。

8.2 如果协议使用可变长度套接口地址结构,clilen很可能太大。在第14章我们将看到这对于Unix域的套接口地址结构是合适的,不过正确的编写函数方法应该是使用由recvfrom返回的真正长度作为sendto的长度。

8.4 象这样运行ping是查看由运行ping的主机接收到的ICMP消息的简易方法。我们把分组发送频率由通常的每秒1次降低到每60秒1次,目的是减少输出量。如果我们在主机solaris上运行我们的UDP客户程序,所指定的服务器IP地址为206.62.226.42,同时还运行ping程序,我们就会取得如下的输出:

```
solaris % ping -v -I 60 127.0.0.1
PING 127.0.0.1: 56 data bytes
64 bytes from localhost (127.0.0.1): icmp_seq=0. time=2. ms
ICMP Port Unreachable from gateway alpha.kohala.com (206.62.226.42)
  for udp from solaris.kohala.com (206.62.226.33)
            to alpha.kohala.com (206.62.226.42) port 9877
```

8.5　它也许有一个套接口接收缓冲区大小，但它绝不会接受数据。大多数实现并不预先给套接口发送缓冲区或接收缓冲区分配内存。用 SO_SNDBUF 和 SO_RCVBUF 套接口选项指定的套接口缓冲区大小仅仅是给套接口设定的上限。

8.6　我们在多宿主机 bsdi 上运行 sock 程序，同时指定-u 选项(使用 UDP)和-l 选项(指定本地 IP 地址和端口)。

```
bsdi % sock -u -l 206.62.226.66.4444 206.62.226.42 8888
hello
recv error: Connection refused
```

本地 IP 地址在图 1.16 中是在主机 bsdi 下面的以太网上，但数据报要到达目的地必须从上面的以太网出去。“Connection refused(连接被拒)”错误的返回是因为 sock 程序调用 connect，导致服务器主机返回端口不可达的 ICMP 消息。使用 tcpdump 监视网络表明源 IP 地址是由客户捆绑的那个地址，而不是外出接口的地址。

```
14:39:46.211130 206.62.226.66.4444 > 206.62.226.42.8888: udp 6
14:39:46.211656 206.62.226.42 > 206.62.226.66:icmp: 206.62.226.42
                                    udp port 8888 unreachable
```

8.7　在客户程序中加一个 printf 语句会在每个数据报间引入一个延迟，从而允许服务器接收更多的数据报。在服务器程序中加入一个 printf 语句则会导致服务器丢失更多的数据报。

8.8　最大的 IPv4 数据报是 65535 字节，这是由图 A.1 中 16 位的总长度字段限定的。IP 头部需要 20 字节，UDP 头部需要 8 字节，留给用户数据的最大 65507 字节。对 IPv6 来说，在没有特大报支持的情况下，IPv6 头部的大小是 40 字节，留给用户数据的最大 65487 字节。

图 E.9 给出了 dg_cli 的新版本。如果你忘记设置发送缓冲区大小的话，源自 Berkeley 的内核给 sendto 返回 EMSGSIZE 错误，因为套接口发送缓冲区通常不满足最大的 UDP 数据报之需(先做完习题 7.1)。然而如果我们如图 E.9 所示设置客户的套接口缓冲区大小并运行它的话，服务器就什么都不返回。我们可以运行 tcpdump 验证客户的数据报发送到了服务器上，但是如果在服务器中增加一个 printf 语句的话，我们发现它的 recvfrom 调用并没有返回这个数据报。问题出在服务器的套接口接收缓冲区小于我们发送的数据报，因此它被丢弃掉而不是递送到套接口。在 BSD/OS 系统上我们可通过运行 netstat -s 命令，查看接收这个大数据报前后“dropped due to full socket buffers(因套接口缓冲区满而丢弃)”计数器值的变化来验证。最终的办法就是修改服务器，重新设置它的套接口发送缓冲区与接收缓冲区的大小。

```
1 #include    "unp.h"
2 #undef    MAXLINE
3 #define    MAXLINE 65507
4 void
5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
```

```
6 {
7     int             size;
8     char        sendline[MAXLINE], recvline[MAXLINE + 1];
9     ssize_t         n;

10    size = 70000;
11    Setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &size, sizeof(size));
12    Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));

13    Sendto(sockfd, sendline, MAXLINE, 0, pservaddr, servlen);

14    n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

15    printf("received %d bytes\n", n);
16 }
```

图 E.9 发送最大大小的 UDP/IPv4 数据报[udpcliserv/dgclibig.c]

大多数网络上 65535 字节的 IP 数据报需要分片。不过 2.9 节中我们说过 IP 层必须支持的重组缓冲区大小只有 576 字节,因此你可能会碰到接收不了本习题发送的最大大小数据报的主机。另外源自 Berkeley 的许多实现(包括 4.4BSD-Lite 2)有一个正负号缺陷(bug),它导致 UDP 不能接受大于 32767 字节的数据报(TCPv2 第 770 页第 95 行)。

第 9 章习题答案

9.1 图 E.10 给出了调用 gethostbyaddr 的程序。

```
1 #include    "unp.h"

2 int
3 main(int argc, char **argv)
4 {
5     char    *ptr, **pptr;
6     char    str[INET6_ADDRSTRLEN];
7     struct hostent  *hptr;

8     while (--argc > 0) {
9         ptr = *++argv;
10        if ( (hptr = gethostbyname(ptr)) == NULL) {
11            err_msg("gethostbyname error for host: %s: %s",
12                    ptr, hstrerror(h_errno));
13            continue;
14        }
15        printf("official hostname: %s\n", hptr->h_name);

16        for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
17            printf("     alias: %s\n", *pptr);

18        switch (hptr->h_addrtype) {
19        case AF_INET:
20 #ifdef   AF_INET6
21        case AF_INET6:
22 #endif
23            pptr = hptr->h_addr_list;
24            for ( ; *pptr != NULL; pptr++) {
25                printf("\taddress: %s\n",
26                    inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
```

```
27                  if ( (hptr = gethostbyaddr( *pptr, hptr->h_length,
28                                              hptr->h_addrtype)) == NULL)
29                      printf("\t(gethostbyaddr failed)\n");
30                  else if (hptr->h_name != NULL)
31                      printf("\tname = %s\n", hptr->h_name);
32                  else
33                      printf("\t(no hostname returned by gethostbyaddr)\n");
34              }
35              break;
36          default:
37              err_ret("unknown address type");
38              break;
39          }
40      }
41      exit(0);
42  }
```

图 E.10　图 9.4 改成调用 gethostbyaddr 的结果[names/hostent2.c]

本程序在只有一个 IP 地址的主机上运行得很好。如果我们在有 4 个 IP 地址的主机上运行图 9.4 中的程序，我们得到如下的输出：

```
solaris % hostent gemini.tuc.noao.edu
official hostname: gemini.tuc.noao.edu
        address: 140.252.8.54
        address: 140.252.4.54
        address: 140.252.3.54
        address: 140.252.1.11
```

但是如果我们在同一台主机上运行图 E.10 中的程序，那么它只输出第 1 个 IP 地址：

```
solaris % hostent2 gemini.tuc.noao.edu
official hostname: gemini.tuc.noao.edu
        address: 140.252.8.54
        name = gemini.tuc.noao.edu
```

问题在于 gethostbyname 和 gethostbyaddr 这两个函数共享同一个 hostent 结构（参见 11.14 节开头部分）。当我们的新程序调用 gethostbyaddr 时，它重用了这个结构以及由它指向的存储区（即 h_addr_list 指针数组及由该数组所指向的数据），结果冲掉了由 gethostbyname 返回的其余 3 个 IP 地址。

9.2　如果你的系统不支持重入版本的 gethostbyaddr（我们将在 11.15 节叙述），那你必须在调用 gethostbyaddr 前拷贝由 gethostbyname 返回的指针数组以及由该数组所指向的数据。

9.3　my_addrs 函数如图 E.11 所示，main 函数则如图 E.12 所示。

```
1 #include    "unp.h"
2 #include    <sys/param.h>

3 char **
4 my_addrs(int *addrtype)
5 {
6     struct hostent    *hptr;
```

```
 7    char            myname[MAXHOSTNAMELEN];
 8    if (gethostname(myname, sizeof(myname)) < 0)
 9        return(NULL);
10    if ( (hptr = gethostbyname(myname)) == NULL)
11        return(NULL);
12    *addrtype = hptr->h_addrtype;
13    return(hptr->h_addr_list);
14 }
```

图 E.11 图 9.7 函数的调用 gethostname 版本[names/myaddrs1.c]

```
 1 #include    "unp.h"
 2 char    **my_addrs(int *);
 3 int
 4 main(int argc, char **argv)
 5 {
 6    int         addrtype;
 7    char    **pptr, buf[INET6_ADDRSTRLEN];
 8    if ( (pptr = my_addrs(&addrtype)) == NULL)
 9        err_quit("my_addrs error");
10    for ( ; *pptr != NULL; pptr++)
11        printf("\taddress: %s\n",
12                inet_ntop(addrtype, *pptr, buf, sizeof(buf)));
13    exit(0);
14 }
```

图 E.12 图 9.7 和 E.11 函数的测试程序[names/prmyaddrs1.c]

9.4 如果 gethostbyname 返回的 hostent 结构给出一个或多个 IPv6 地址,h_length 将会是 16。它将导致 sockaddr_in 结构的溢出,溢出部分将写到内存中该结构后的任意存储区上。这种解析器选项只能在程序准备好处理 IPv6 地址时设置,而我们的程序却不是这样。本例子还指出了 memcpy 的长度参数应该是目标的大小(本例中是 sizeof(struct in_addr)),而不是源的大小(即 hp->h_length),即使两者相同也应该这样。

9.5 chargen 服务器一直向客户发送数据,直到客户关闭连接为止(也就是说你中断客户为止)。

9.6 正如图 9.3 所提及的那样,这是较新版本 BIND 的一个特性。图 E.13 给出了程序的修改后版本。对主机名的测试顺序很重要。我们首先调用 inet_pton,因为它是一个快速的局限于内存的测试函数,用于判定主机名串是不是一个有效的点分十进制数 IP 地址。这种测试失败时我们才调用 gethostbyname,它一般涉及某些网络资源,也得花一段时间。

如果这个串是一个有效的点分十进制数 IP 地址,我们就构造指向这个 IP 地址的指针数组(addrs),它使得使用 pptr 的循环其代码保持不变。

地址转换成套接口地址结构中的二进制格式后,我们把图 9.8 中的 memcpy 调用改成了 memmove,这是因为输入点分十进制数 IP 地址时,这个调用的源和目标存储区是相同的。习题 30.3 讨论 memcpy 和 memmove 对重叠存储区的处理。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int           sockfd, n;
6     char          recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     struct in_addr **pptr, *addrs[2];
9     struct hostent *hp;
10    struct servent *sp;
11    if (argc != 3)
12        err_quit("usage: daytimetcpcli2 <hostname> <service>");
13    bzero(&servaddr, sizeof(servaddr));
14    servaddr.sin_family = AF_INET;
15    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) == 1) {
16        addrs[0] = &servaddr.sin_addr;
17        addrs[1] = NULL;
18        pptr = &addrs[0];
19    } else if ( (hp = gethostbyname(argv[1])) != NULL) {
20        pptr = (struct in_addr **) hp->h_addr_list;
21    } else
22        err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno));
23    if ( (n = atoi(argv[2])) > 0)
24        servaddr.sin_port = htons(n);
25    else if ( (sp = getservbyname(argv[2], "tcp")) != NULL)
26        servaddr.sin_port = sp->s_port;
27    else
28        err_quit("getservbyname error for %s", argv[2]);
29    for ( ; *pptr != NULL; pptr++) {
30        sockfd = Socket(AF_INET, SOCK_STREAM, 0);
31        memmove(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
32        printf("trying %s\n",
33                Sock_ntop((SA *) &servaddr, sizeof(servaddr)));
34        if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) == 0)
35            break;          /* success */
36        err_ret("connect error");
37        close(sockfd);
38    }
39    if (*pptr == NULL)
40        err_quit("unable to connect");
41    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
42        recvline[n] = 0;   /* null terminate */
43        Fputs(recvline, stdout);
44    }
45    exit(0);
46 }
```

图 E.13　允许点分十进制数 IP 地址或主机名,端口号或服务名的版本[names/daytimetcpcli2.c]

9.7　图 E.14 给出了这个程序。

```
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5     int           sockfd, n;
6     char          recvline[MAXLINE + 1];
7     struct sockaddr_in servaddr;
8     struct sockaddr_in6 servaddr6;
9     struct sockaddr *sa;
10    socklen_t salen;
11    struct in_addr **pptr;
12    struct hostent *hp;
13    struct servent *sp;
14    if (argc != 3)
15        err_quit("usage: daytimetcpcli3 <hostname> <service>");
16    if ( (hp = gethostbyname(argv[1])) == NULL)
17        err_quit("hostname error for %s: %s", argv[1], hstrerror(h_errno));
18    if ( (sp = getservbyname(argv[2], "tcp")) == NULL)
19        err_quit("getservbyname error for %s", argv[2]);
20    pptr = (struct in_addr **) hp->h_addr_list;
21    for ( ; *pptr != NULL; pptr++) {
22        sockfd = Socket(hp->h_addrtype, SOCK_STREAM, 0);
23        if (hp->h_addrtype == AF_INET) {
24            sa = (SA *) &servaddr;
25            salen = sizeof(servaddr);
26        } else if (hp->h_addrtype == AF_INET6) {
27            sa = (SA *) &servaddr6;
28            salen = sizeof(servaddr6);
29        } else
30            err_quit("unknown addrtype %d", hp->h_addrtype);
31        bzero(sa, salen);
32        sa->sa_family = hp->h_addrtype;
33        sock_set_port(sa, salen, sp->s_port);
34        sock_set_addr(sa, salen, *pptr);
35        printf("trying %s\n", Sock_ntop(sa, salen));
36        if (connect(sockfd, sa, salen) == 0)
37            break;          /* success */
38        err_ret("connect error");
39        close(sockfd);
40    }
41    if (*pptr == NULL)
42        err_quit("unable to connect");
43    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
44        recvline[n] = 0;   /* null terminate */
45        Fputs(recvline, stdout);
46    }
47    exit(0);
48 }
```

图 E.14 图 9.8 程序同时适用于 IPv4 和 IPv6 的修改版本[names/daytimetcpcli3.c]

我们使用由 gethostbyname 返回的 h_addrtype 值判定地址类型，另外使用 sock_set_port 和 sock_set_addr 这两个函数(3.8 节)设定合适的套接口地址结构中的这两个成员。

尽管本程序能够工作，它还是有两个局限。首先，我们必须处理所有的差异，查看 h_addrtype 后再设置适当的 sa 和 salen。更好的办法是由某个库函数不仅完成主机名和服务名的查看，而且完成整个套接口地址结构的填写(例如 11.2 节的 getaddrinfo)。其次，本程序只能在支持 IPv6 的主机上编译。要在只支持 IPv4 的主机上编译必须加不少 #ifdef 伪代码，结果变得复杂起来。

我们在第 11 章讨论协议无关性概念，那儿我们会看到更好的办法。

第 10 章习题答案

10.1　下面是相关的片段(省掉了登录和列目录等内容)。

```
solaris % ftp bsdi
Connected to bsdi.kohala.com.
220 bsdi.kohala.com FTP server ...
...
230 Guest login ok,access restrictions apply.
ftp> debug
Debugging on (debug=1).
ftp> dir
---> PORT 206,62,226,33,129,145
200 PORT command successful.
---> LIST
150 Opening ASCII mode data connection for /bin/ls.
...

solaris % ftp sunos5
Connected to sunos5.kohala.com.
220 sunos5.kohala.com FTP server ...
...
230 Guest login ok, access restrictions apply.
ftp> debug
Debugging on (debug=1)
ftp> dir
---> LPRT 6,16,95,27,223,0,206,62,226,0,0,32,8,0,32,120,227,227,2,129,148
200 LPRT command successful.
---> LIST
150 ASCII data connection for /bin/ls (5f1b:df00:ce3e:e200:20:800:2078:e3e3,3172)
(0 bytes).
```

第 11 章习题答案

11.1　分配一个大缓冲区(比任何套接口地址结构都要大)并调用 getsockname。它的第 3 个参数是一个值-结果参数，由它返回真正的协议地址大小。不过这种方法只适合具有固定长度套接口地址结构的协议(例如 IPv4 和 IPv6)，对于返回可变长度套接口地址结构的协议(例如 Unix 域的套接口，第 14 章)，却不能保证正确工作。

11.2　我们首先分配存放主机名和服务名的数组：

```
char host[NI_MAXHOST],serv[NI_MAXSERV];
```

在 accept 返回之后我们调用 getnameinfo 取代 sock_ntop:

```
if (getnameinfo (cliaddr,len,host,NI_MAXHOST,serv,NI_MAXSERV,
                 NI_NUMERICHOST | NI_NUMERICSERV) == 0)
    printf("connection from %s. %s\n", host, serv);
```

由于这是服务器,我们指定 NI_NUMERICHOST 和 NI_NUMERICSERV 标志以避免查询 DNS 和查看/etc/services 文件。

11.3 第二个服务器碰到的问题是无法捆绑与第一个服务器相同的端口,这是因为没有设置 SO_REUSEADDR 套接口选项。最容易的解决办法是拷贝 udp_server 函数,把它的名字换成 udp_server_reuseaddr,由它设置本套接口选项,再从服务器程序调用这个新函数。

11.4 如果该变量是全局变量,getaddrinfo 就不会线程安全。

11.5 是的。ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers 给出的服务名 cl/1 和 914c/g 都含有斜杠。

11.6 当客户输出"Trying 206.62.226.35..."时,gethostbyname 已经返回了 IP 地址。客户在此之前的停顿是解析器用于查询主机名的时间。输出"Connected to bsdi.kohala.com"意味着 connect 已经返回。这两个输出行之间的停顿是 connect 用来建立连接的时间。

第 12 章习题答案

12.1 daemon_init 中关闭所有描述字的 close 调用也将关闭由 tcp_listen 建立的监听 TCP 套接口。由于作为守护进程编写的程序可能是从某个系统启动命令脚本执行的,因此我们不应该假设任何错误消息都能写到终端上。即使象无效的命令行参数这样的启动出错消息也应该使用 syslog 登记。

12.2 TCP 版本的 echo、discard 和 chargen 服务器由 inetd 派生出来后都作为子进程运行,这是因为它们都要运行到相应的客户终止连接为止。另外 2 个 TCP 服务器 time 和 daytime 并不需要派生,因为它们的服务实现起来非常简单(取得当前时间和日期,把它格式化后写出,再关闭连接),因此它们是由 inetd 直接处理的。所有 5 个 UDP 服务的处理都不需要 fork,因为它们只就引发它们的客户数据报生成并回应最多一个数据报。因此它们也是由 inetd 直接处理的。

12.3 这是个众所周知的拒绝服务型攻击([CERT 1996a])。来自端口 7 的第 1 个数据报导致 chargen 服务器发回一个数据报到端口 7,它被回射成发往 chargen 服务器的下一个数据报,这样一直循环下去。BSD/OS 上实现的解决办法是:如果进入的数据报其源端口和目的端口都属于内部服务,那么它们将被拒绝。另外一种常用的解决办法是:或者在每台主机上通过 inetd 禁止这些内部服务,或者在一个组织去往因特网的路由器上这么做。

12.4 客户的 IP 地址和端口是从由 accept 填写的套接口地址结构获取的。
inetd 对 UDP 套接口无能为力的原因是读入数据报的 recvfrom 是由调用 exec 启动的真正服务器而不是 inetd 执行的。

指定 MSG_PEEK 标志(13.7 节)后 inetd 也能读数据报,它只是获取客户的 IP 地址和端口,整个数据报则纹丝不动地由真正的服务器来读。

第 13 章习题答案

13.1　如果以前未曾设置处理程序的话,第一次调用 signal 将返回 SIG_DFL,第二次调用 signal 设回原来的处理程序时,所设回的就是 SIGALRM 的缺省处理程序。

13.3　下面是修改后的 for 循环。

```
for ( ; ; ) {
    if ( (n = Recv(sockfd, recvline, MAXLINE, MSG_PEEK) ) == 0)
        break;          /* server closed connection */

    ioctl(sockfd, FIONREAD, &npend);
    printf("%d bytes from PEEK, %d bytes pending\n", n, npend);

    n = Read(sockfd, recvline, MAXLINE);
    recvline[n] = 0;            /* null terminate */
    Fputs(recvline, stdout);
}
```

13.4　数据仍然输出,因为掉出 main 函数的末尾跟从这个函数返回是一样的,而 main 函数又是由 C 启动例程如下调用的:

```
exit(main(argc, argv));
```

因此还是调用了 exit,从而也调用了标准 I/O 的清扫例程。

第 14 章习题答案

14.1　unlink 从文件系统中删除了路径名,此后客户调用 connect 就会失败。服务器的监听套接口不受影响,但在 unlink 之后没有客户能够成功 connect。

14.2　即使路径名仍然存在,客户也无法 connect 到服务器,这是因为 connect 成功要求当前已有一个打开的并且捆绑在那个路径名上的 Unix 域套接口(14.4 节)。

14.3　当服务器调用 sock_ntop 输出客户的协议地址时,输出信息为"datagram from (no pathname bound)(数据报来自(无路径名绑定))",这是因为缺省情况下没有路径名捆绑在客户的套接口上。

解决办法之一是在 udp_client 和 udp_connect 中明确检查 Unix 域套接口,然后调用 bind 给它捆绑一个临时路径名。这样就把协议相关处理置于原本所属的库函数中而不是我们的应用程序中。

14.4　即使我们给服务器程序的 26 字节应答施行单个字节的 write 调用,客户程序中增加的 sleep 调用还是保证在调用 read 之前所有 26 个分节都接收到,这使得 read 调用返回完整的服务器应答。这个情况只是(再次)证实了 TCP 是没有内在记录标记的字节流。

要使用 Unix 域协议,我们以 2 个命令行参数/local(或/unix)和/tmp/daytime(或你想使用的任何其他临时路径名)启动客户和服务器。情况没有变化:每次运行客户程序由 read 返回的都是 26 个字节。

服务器给每个 send 指定 MSG_EOR 标志后，每个字节都被认为是一个逻辑记录，从而每次调用 read 所返回的也是 1 个字节。碰巧源自 Berkeley 的实现缺省支持 MSG_EOR 标志。不过这一点没有写到正式文档中，在成品代码中也不应该使用。我们这儿用它来作为区别字节流协议和面向记录协议的例子。从实现角度看，每个输出操作都要进入一个 mbuf(内存缓冲区)，MSG_EOR 标志就是随 mbuf 从发送套接口到达接收套接口的接收缓冲区，由内核根据 mbuf 维持的。调用 read 时 MSG_EOR 标志仍然依附在每个 mbuf 上，因此普通的内核 read 例程(它因一些协议使用 MSG_EOR 标志而支持它)逐次返回各个字节。如果我们用 recvmsg 取代 read，它在每次返回一个字节的同时也把 MSG_EOR 标志返回到 msg_flags 成员中。这个特性并不适用于 TCP，因为发送方 TCP 从来不看所发送 mbuf 中的 MSG_EOR 标志，而且即使它看了，它也无法在 TCP 头部中把这个标志传递给接收方 TCP。(感谢 Matt Thomas 指出这个没写入文档中的“特性”。)

14.5 图 E.15 给出了这个程序的实现。我们在图 E.21 中再给出它的 XTI 版本。

```
1 #include    "unp.h"
2 #define     PORT            9999
3 #define     ADDR            "127.0.0.1"
4 #define     MAXBACKLOG      100
5               /* globals */
6 struct sockaddr_in     serv;
7 pid_t pid;   /* of child */
8 int pipefd[2];
9 #define pfd   pipefd[1]   /* parent's end */
10 #define cfd   pipefd[0]   /* child's end */
11              /* function prototypes */
12 void   do_parent(void);
13 void   do_child(void);
14 int
15 main(int argc, char * * argv)
16 {
17     if (argc != 1)
18         err_quit("usage: backlog");
19     Socketpair(AF_UNIX, SOCK_STREAM, 0, pipefd);
20     bzero(&serv, sizeof(serv));
21     serv.sin_family = AF_INET;
22     serv.sin_port = htons(PORT);
23     Inet_pton(AF_INET, ADDR, &serv.sin_addr);
24     if ( (pid = Fork()) == 0)
25         do_child();
26     else
27         do_parent();
28     exit(0);
29 }
30 void
31 parent_alrm(int signo)
```

```
32 {
33     return;          /* just interrupt blocked connect() */
34 }
35 void
36 do_parent(void)
37 {
38     int        backlog, j, k, junk, fd[MAXBACKLOG + 1];
39     Close(cfd);
40     Signal(SIGALRM, parent_alrm);
41     for (backlog = 0; backlog <= 14; backlog++) {
42         printf("backlog = %d: ", backlog);
43         Write(pfd, &backlog, sizeof(int));  /* tell child value */
44         Read(pfd, &junk, sizeof(int));          /* wait for child */
45         for (j = 1; j <= MAXBACKLOG; j++) {
46             fd[j] = Socket(AF_INET, SOCK_STREAM, 0);
47             alarm(2);
48             if (connect(fd[j], (SA *) &serv, sizeof(serv)) < 0) {
49                 if (errno != EINTR)
50                     err_sys("connect error, j = %d", j);
51                 printf("timeout, %d connections completed\n", j-1);
52                 for (k = 1; k <= j; k++)
53                     Close(fd[k]);
54                 break;      /* next value of backlog */
55             }
56             alarm(0);
57         }
58         if (j > MAXBACKLOG)
59             printf("%d connections? \n", MAXBACKLOG);
60     }
61     backlog = -1;          /* tell child we're all done */
62     Write(pfd, &backlog, sizeof(int));
63 }
64 void
65 do_child(void)
66 {
67     int        listenfd, backlog, junk;
68     const int   on = 1;
69     Close(pfd);
70     Read(cfd, &backlog, sizeof(int));   /* wait for parent */
71     while (backlog >= 0) {
72         listenfd = Socket(AF_INET, SOCK_STREAM, 0);
73         Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
74         Bind(listenfd, (SA *) &serv, sizeof(serv));
75         Listen(listenfd, backlog);          /* start the listen */
76         Write(cfd, &junk, sizeof(int));      /* tell parent */
77         Read(cfd, &backlog, sizeof(int));   /* just wait for parent */
78         Close(listenfd);   /* closes all queued connections too */
79     }
80 }
```

图 E.15 确定不同的 backlog 值对应的真正已排队连接数[debug/backlog.c]

第15章习题答案

15.1 描述字是在父子进程间共享的，因此它的访问计数为2。如果父进程调用close，那它只是把访问计数由2减为1，既然它仍然大于0，FIN也就不发送。这就是使用shutdown函数的另外一个原因：即使描述字的访问计数仍大于0，FIN还是强制发出。

15.2 父进程继续写往已经接收到FIN的套接口。它发送给服务器的第1个分节将引发RST响应。随后的下一个write调用将导致父进程接收到SIGPIPE信号，这跟我们在5.1节讨论过的一样。

15.3 当子进程调用getppid给父进程发送SIGTERM信号时，所返回的进程ID将是1即init进程，它是所有孤儿进程的继父。子进程试图向init进程发送信号，但没有足够的权限。要是这个客户程序有机会以超级用户特权运行，从而允许它给init发送信号的话，我们应在发送信号前检查getppid的返回值。

15.4 如果去掉这两行，select就被调用。不过select会立即返回，因为连接建立后套接口是可写的。这个测试加goto语句只是避免不必要地调用select。

15.5 如果服务器在accept调用返回后立即发送数据，而且客户在三路握手中的第2个分组到达时正在忙着完成客户端的连接(图2.5)，数据在connect调用返回前到达的情况就可能发生。例如SMTP服务器在建立新的连接后立即写，给客户发送一个问候消息。

第16章习题答案

16.1 不要紧，因为图16.2中union的前3个成员都是套接口地址结构。

第17章习题答案

17.1 sdl_nlen成员将是5，sdl_alen成员将是8。这需要21字节，在32位体系结构上则向上舍入成24个字节(TCPv2第89页)。

17.2 内核的响应决不发送到这个套接口。SO_USELOOPBACK这个套接口选项确定内核是否把应答发送给请求进程，TCPv2第649～650页讨论了这一点。既然大多数进程需要这些应答，这个选项缺省是打开的。关闭它应答就发送不到请求进程。

第18章习题答案

18.1 如果你得到许多应答，它们每次到达的先后顺序不应该都一样。不过发送主机本身的应答通常是第一个，因为其数据报的来往并不出现在真正的网络上。

18.2 1472等于以太网MTU(1500)减掉20字节的IPv4头部以及8字节的UDP头部。

18.3 BSD/OS中由信号处理程序写字节到管道并返回后，select返回EINTR。select再次调用时返回管道可读的信息。

第 19 章习题答案

19.1　我们运行这个程序的输出如下：

```
solaris % udpcli05 224.0.0.1
hi
from 206.62.226.34: Thu Jun 19 17:28:32 1997
from 206.62.226.43: Thu Jun 19 17:28:32 1997
from 206.62.226.42: Thu Jun 19 17:28:32 1997
from 206.62.226.40: Thu Jun 19 17:28:32 1997
from 206.62.226.35: Thu Jun 19 17:28:32 1997
```

5 个有响应的主机运行的操作系统有 AIX、BSD/OS、Digital Unix 和 Linux。没有响应却有多播能力的仅有节点是运行 Solaris 2.5 的主机和 Cisco 路由器。

这儿 UDP 数据报的目的地址是 224.0.0.1，它是所有主机组，所有具备多播能力的节点都必须参加。这个 UDP 数据报是作为一个多播以太网帧发送的，所有能多播的节点都接收到它，因为它们都属于这个组。有响应的主机都把接收到的数据报传递给了 UDP daytime 服务器(它通常是 inetd 的一部分)，而不管那个套接口是否已加入所有主机组。然而 Solaris 的实现却要求接收数据报的目的套接口必须加入这个组。

本例子表明决不是设计来响应多播数据报的 UDP 程序也能接收多播数据报。我们在第 18 章看到了这个 daytime 例子的同样情况：决不是设计来响应广播数据报的 UDP 程序也能接收广播数据报。

19.2　图 E.16 给出了调用 bind 捆绑多播地址和端口 0 的 main 函数修改版本。

```
1 #include    "unp.h"
2 int
3 main(int argc, char * * argv)
4 {
5     int         sockfd;
6     socklen_t salen;
7     struct sockaddr * cli, * serv;
8     if (argc ! = 2)
9         err_quit("usage: udpcli06 <IPaddress>");
10    sockfd = Udp_client(argv[1], "daytime", (void * * ) &serv, &salen);
11    cli = Malloc(salen);
12    memcpy(cli, serv, salen);          /* copy socket address struct */
13    sock_set_port(cli, salen, 0);      /* and set port to 0 */
14    Bind(sockfd, cli, salen);
15    dg_cli(stdin, sockfd, serv, salen);
16    exit(0);
17 }
```

图 E.16　捆绑多播地址的 UDP 客户程序 main 函数[mcast/udpcli06.c]

不幸的是，我们尝试的 3 个系统即 BSD/OS、Digital Unix 和 Solaris 2.5 都允许 bind，随后发送的 UDP 数据报具有多播源 IP 地址。有响应的那 5 个系统(跟上一个习题相同)都在应答中对换了源和目的 IP 地址，结果所有 5 个应答都是多

播的！接收到这些应答的有多播能力的客户主机倒没事，因为这些应答的目的端口就是最初捆绑多播地址时由(惹事的)客户所在主机的内核选定的临时端口，因此没有任何套接口绑定在其上。对于多播UDP数据报，ICMP不生成端口不可达消息。

19.3 从有多播能力的主机solaris这么做时，我们得到

```
solaris % ping 224.0.0.1
PING 224.0.0.1: 56 data bytes
64 bytes from solaris.kohala.com (206.62.226.33): icmp_seq=0. time=4. ms
64 bytes from linux.kohala.com(206.62.226.40): icmp_seq=0. time=9. ms
64 bytes from aix.kohala.com (206.62.226.43): icmp_seq=0. time=11. ms
64 bytes from bsdi.kohala.com(206.62.226.35): icmp_seq=0. time=13. ms
64 bytes from alpha.kohala.com (206.62.226.42): icmp_seq=0. time=15. ms
64 bytes from sunos5.kohala.com(206.62.226.36): icmp_seq=0. time=17. ms
64 bytes from bsdi2.kohala.com (206.62.226.34): icmp_seq=0. time=54. ms
64 bytes from gw.kohala.com (206.62.226.62): icmp_seq=0. time=75. ms
^?
----224.0.0.1 PING Statistics----
1 packets transmitted, 8 packets received, 8.00 times amplification
round-trip (ms) min/avg/max = 4/24/75
```

来自图1.16中顶部以太网的所有主机除没有多播能力的unixware外都有响应(当然发送者本身也有响应)。

19.4 既然内核没有多播能力，它把目的地址作为普通的IP地址来处理。它在普通的IP路由表中查看224.0.0.1，把它跟缺省路径匹配，而缺省路径指向的下一跳路由器是gw(图1.16)。一个单播ICMP回射请求于是发往这个路由器，其目的IP地址是224.0.0.1，硬件地址则是该路由器的以太网接口(也就是说硬件地址并不是多播地址)。该路由器接受所接收到的分组，因为它的目的硬件地址是路由器的接口之一，目的IP地址则是它所属的一个多播组。

19.5 如果我们在主机solaris上这么做，我们会得到：

```
solaris % ping 224.0.0.2
PING 224.0.0.2: 56 data bytes
64 bytes from bsdi.kohala.com (206.62.226.35): icmp_seq=0. time=3. ms
64 bytes from gw.kohala.com (206.62.226.62): icmp_seq=0. time=24. ms
^?
----224.0.0.2 PING Statistics ----
1 packets transimitted, 2 packets received, 2.00 times amplification
round-trip (ms) min/avg/max = 3/13/24
```

我们期待bsdi响应，因为它是本子网的多播路由器，有一个通往MBone(B.2节)的隧道，运行着mrouted。路由器gw也有响应，但它并不作为多播路由器工作。

19.7 从图19.17我们看到NTP时间戳是自1990年1月1日以来的秒数，一年有31,536,000秒(365×24×60×60)，因此图中两个值约是96.7年(忽略闰年)，这是明智的。另外这个声明的版本号比会话ID大，而会话ID我们预期是在首次声明会话时分配的，因此这同样是明智的。

19.8 值1,073,741,824转换成浮点数并除以4,294,967,296后得到0.250。再乘以

1,000,000 后得到 250,000,它以微秒为单位就是¼秒。

最大的整分数是 4,294,967,295,它除以 4,294,967,296 后得到 0.99999999976716935634。再乘以 1,000,000 并截成整数后得到 999,999,它就是最大的微秒数。

19.9 在 7.5 节我们讨论这两上套接口选项时,我们注意到如果捆绑的 IP 地址是一个多播地址,SO_REUSEADDR 就被认为跟 SO_REUSEPORT 等价。这样一来简化了代码的可移植能力,因为要不是这样的话我们将不得不如下编写:

```
#ifdef SO_REUSEPORT
    Setsockopt(fd, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on));
#else
    Setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
#endif
```

19.10 为把多播数据报递送给某个套接口,有些系统要求它必须已加入多播数据报的目的多播组。这些系统上仅仅捆绑端口是不够的。Solaris 2.5 就是以这种方式运作的。相反,源自 Berkeley 的实现只要求套接口的端口与多播数据报的目的端口匹配,至于是否已加入目的多播组则无关紧要。回忆一下图 19.22 中我们调用 bind_ubcast 函数把套接口间接捆绑到通配地址,因此它并没有加入目的多播组中。

19.11 会接收到 12 个额外数据报:图 19.31 中发出 3 个多播数据报后,每个都被递送到 4 个匹配的套接口(3 个多播套接口加上通配套接口)。

第 20 章习题答案

20.1 我们已知道 sock_ntop 使用自己的静态缓冲区来存放结果。如果我们在同一个 printf 中作为参数调用它两次的话,第 2 次调用会改写掉第 1 次调用的结果。

20.2 是的,如果应答中包含的用户数据是 0 个字节(也就是说仅有一个 hdr 结构)。

20.3 由于 select 并不修改指定其时间限制的 timeval 结构,因此你必须记下第 1 个分组的发送时刻(它已由 rtt_ts 以微秒为单位返回)。当 select 返回套接口可读的信息时,记下当前时刻;当再次调用 recvmsg 时,给 select 计算新的超时值。

20.4 常用的技术是给每个接口地址建立一个套接口,就象我们在 19.11 和 20.6 节所做的那样,应答就从跟请求到达相同的套接口上发出。

20.5 既不给出主机名参数也不设置 AI_PASSIVE 标志就调用 getaddrinfo 会导致它假定成本地主机(localhost):0::1(IPv6)和 127.0.0.1(IPv4)。回忆一下 getaddrinfo 先于 IPv4 套接口地址结构返回 IPv6 套接口地址结构,前提是主机支持 IPv6。如果这两种协议都支持,udp_client 中使用 AF_INET6 地址族调用 socket 将会成功。

图 E.17 是本程序的协议无关版本。

```
1 #include    "unpifi.h"
2 void  mydg_echo(int, SA *, socklen_t);
3 int
```

```
4 main(int argc, char * * argv)
5 {
6     int        sockfd, family, port;
7     const int on = 1;
8     pid_t pid;
9     socklen_t salen;
10    struct sockaddr * sa, * wild;
11    struct ifi_info * ifi, * ifihead;
12    if (argc == 2)
13        sockfd = Udp_client(NULL, argv[1], (void * * ) &sa, &salen);
14    else if (argc == 3)
15        sockfd = Udp_client(argv[1], argv[2], (void * * ) &sa, &salen);
16    else
17        err_quit("usage: udpserv04 [ <host> ] <service or port>");
18    family = sa->sa_family;
19    port = sock_get_port(sa, salen);
20    Close(sockfd);          /* we just want family, port, salen */
21    for (ifihead = ifi = Get_ifi_info(family, 1);
22         ifi != NULL; ifi = ifi->ifi_next) {
23            /* bind unicast address */
24        sockfd = Socket(family, SOCK_DGRAM, 0);
25        Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
26        sock_set_port(ifi->ifi_addr, salen, port);
27        Bind(sockfd, ifi->ifi_addr, salen);
28        printf("bound %s\n", Sock_ntop(ifi->ifi_addr, salen));
29        if ( (pid = Fork()) == 0) {        /* child */
30          mydg_echo(sockfd, ifi->ifi_addr, salen);
31          exit(0);          /* never executed */
32        }
33        if (ifi->ifi_flags & IFF_BROADCAST) {
34                /* try to bind broadcast address */
35          sockfd = Socket(family, SOCK_DGRAM, 0);
36          Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
37          sock_set_port(ifi->ifi_brdaddr, salen, port);
38          if (bind(sockfd, ifi->ifi_brdaddr, salen) < 0) {
39              if (errno == EADDRINUSE) {
40                  printf("EADDRINUSE: %s\n",
41                         Sock_ntop(ifi->ifi_brdaddr, salen));
42                  Close(sockfd);
43                  continue;
44              } else
45                  err_sys("bind error for %s",
46                          Sock_ntop(ifi->ifi_brdaddr, salen));
47          }
48          printf("bound %s\n", Sock_ntop(ifi->ifi_brdaddr, salen));
49          if ( (pid = Fork()) == 0) {        /* child */
50            mydg_echo(sockfd, ifi->ifi_brdaddr, salen);
51            exit(0);          /* never executed */
52          }
53        }
54    }
```

```
55          /* bind wildcard address */
56      sockfd = Socket(family, SOCK_DGRAM, 0);
57      Setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
58      wild = Malloc(salen);
59      memcpy(wild, sa, salen);      /* copy family and port */
60      sock_set_wild(wild, salen);
61      Bind(sockfd, wild, salen);
62      printf("bound %s\n", Sock_ntop(wild, salen));
63      if ( (pid = Fork()) == 0) {          /* child */
64          mydg_echo(sockfd, wild, salen);
65          exit(0);          /* never executed */
66      }
67      exit(0);
68  }
69  void
70  mydg_echo(int sockfd, SA *myaddr, socklen_t salen)
71  {
72      int             n;
73      char        mesg[MAXLINE];
74      socklen_t   len;
75      struct sockaddr *cli;
76      cli = Malloc(salen);
77      for ( ; ; ) {
78          len = salen;
79          n = Recvfrom(sockfd, mesg, MAXLINE, 0, cli, &len);
80          printf("child %d, datagram from %s",
81                  getpid(), Sock_ntop(cli, len));
82          printf(", to %s\n", Sock_ntop(myaddr, salen));
83          Sendto(sockfd, mesg, n, 0, cli, len);
84      }
85  }
```

图 E.17　20.6 节程序的协议无关版本[advio/udpserv04.c]

第 21 章习题答案

21.1　是的。第一个例子中的 2 个字节是随单个紧急指针发送的，该指针指向的是 b 后面的字节。第二个例子(两个函数调用)中首先发送的是 a 以及指向它之后字节的紧急指针，接着以另外一个 TCP 分节发送的是 b 和指向它之后字节的另外一个紧急指针。

21.2　图 E.18 给出了使用 poll 的版本。

```
1  #include    "unp.h"
2  int
3  main(int argc, char **argv)
4  {
5      int    listenfd, connfd, n, justreadoob = 0;
6      char   buff[100];
7      struct pollfd      pollfd[1];
8      if (argc == 2)
```

```
 9          listenfd = Tcp_listen(NULL, argv[1], NULL);
10      else if (argc == 3)
11          listenfd = Tcp_listen(argv[1], argv[2], NULL);
12      else
13          err_quit("usage: tcprecv03p [ <host> ] <port#>");
14      connfd = Accept(listenfd, NULL, NULL);
15      pollfd[0].fd = connfd;
16      pollfd[0].events = POLLRDNORM;
17      for ( ; ; ) {
18          if (justreadoob == 0)
19              pollfd[0].events |= POLLRDBAND;
20          Poll(pollfd, 1, INFTIM);
21          if (pollfd[0].revents & POLLRDBAND) {
22              n = Recv(connfd, buff, sizeof(buff)-1, MSG_OOB);
23              buff[n] = 0;            /* null terminate */
24              printf("read %d OOB byte: %s\n", n, buff);
25              justreadoob = 1;
26              pollfd[0].events &= ~POLLRDBAND;      /* turn bit off */
27          }
28          if (pollfd[0].revents & POLLRDNORM) {
29              if ( (n = Read(connfd, buff, sizeof(buff)-1)) == 0) {
30                  printf("received EOF\n");
31                  exit(0);
32              }
33              buff[n] = 0;      /* null terminate */
34              printf("read %d bytes: %s\n", n, buff);
35              justreadoob = 0;
36          }
37      }
38  }
```

图 E.18 以 poll 代替 select 的图 21.6 程序修改版本[oob/tcprecv03p.c]

第 22 章习题答案

22.1 这样的改动引入了一个错误。问题在于 nqueue 是在数组项 dg[iget]处理前减 1 的,因此信号处理程序读入的新数据报可能会写在这个数组元素上。

第 23 章习题答案

23.1 使用 fork 的例子将会使用 101 个描述字,其中有 1 个是监听套接口描述字,其余 100 个是已连接套接口描述字。不过 101 个进程(1 个父进程,100 个子进程)的每一个只有一个描述字是打开的(忽略任何其他描述字,例如服务器不是守护进程时的标准输入)。然而线程化的服务器是单个进程中有 101 个描述字,每个线程(包括主线程)处理其中一个。

23.2 TCP连接终止的最后 2 个分节(服务器的 FIN 和客户对这个 FIN 的 ACK)将不会交换。这使得连接的客户端处于 FIN_WAIT_2 状态(图 2.4)。源自 Berkeley 的实现只要客户端保持这种状态超过 11 分钟就会超时断连(TCPv2 第 825～827页)。服务器(最终)也会耗尽描述字。

23.3 这个消息应该在主线程从套接口读入文件结束符而另外一个线程却还在运行时输出。这样做的一个简单方法是:声明另外一个名为 done 的外部变量并把它初始化为 0。线程 copyto 在返回前再把它设置成 1。主线程检查这个变量,如果为 0 就输出出错消息。由于只有一个线程设置这个变量,因而没有任何同步的必要。

第 24 章习题答案

24.1 服务器上没有变化。首先,路由器采用普通的弱端系统模型,因此即使目的地址是另外的接口,它也能接受来自以太网的外来数据报。其次,使用 IPv4 源路径时,转发主机将以外出接口的地址替换源路径表中它的地址。不论分组发送到哪个地址,外出接口总是以太网接口(206.62.226.62)。

24.2 没有变化。所有系统都是邻居,因此严格的源路径跟宽松的源路径是等同的。

24.3 我们会在缓冲区的末尾放一个 EOL(值为 0 的单个字节)。

24.4 由于 ping 创建的是原始套接口(第 25 章),因此它能接收到用 recvfrom 读入的每个数据报的完整 IP 头部,任何 IP 选项也包括在内。

24.5 因为 rlogind 是由 inetd 激活的(12.5 节)。

24.6 问题在于 setsockopt 的第 5 个参数以指向长度的指针代替了长度本身。这个缺陷可能是在开始使用 ANSI C 原型时修正的。

这个缺陷结果却是无害的,因为正如我们所提到过的那样,IP_OPTIONS 套接口选项的关闭既可指定一个空指针作为第 4 个参数,也可使用值为 0 的第 5 个参数即长度(TCPv2 第 269 页)。

第 25 章习题答案

25.1 IPv6 头部中的版本字段和下一个头部字段是无法得到的。有效负载长度字段或者作为某个输出函数的一个参数,或者作为来自某个输入函数的返回值总是可得到,但是当需要特大有效负载选项时,真正的选项本身应用进程是得不到的。片段的头部应用进程也得不到。

25.2 最终客户的套接口接收缓冲区会填满,它导致服务器守护进程的 write 调用阻塞。我们不希望发生这种情况,因为它使得守护进程在任何一个套接口上都停止处理新的数据。最容易的解决办法是让守护进程把它跟客户的 Unix 域连接的本地端设置成非阻塞。守护进程还必须调用 write 以取代包裹函数 Write,同时忽略 EWOULDBLOCK 错误。

25.3 源自 Berkeley 的内核缺省允许在原始套接口上的广播(TCPv2 第 1057 页)。SO_BROADCAST 套接口选项只有 UDP 套接口才需指定。

25.4 我们的程序既不检查多播地址,也不设置 IP_MULTICAST_IF 套接口选项,因此内核可能通过搜索 224.0.0.1 的路由表项选定外出接口。我们也不设置 IP_MULTICAST_TTL 套接口选项,因此它缺省成 1,这是合理的。

第 26 章习题答案

26.1 这个标志表示跳转缓冲区已由 sigsetjmp 设置(图 26.10)。它看来可能多余,但在

信号处理程序建立后和 sgsetjmp 调用前，信号被递交的机会还是有的。即使程序本身不会导致信号的产生，它也可能以其他方式产生(例如使用 kill 命令)。

第 27 章习题答案

27.1 父进程保持监听套接口打开是为以后 fork 额外的子进程做准备(它是对现行代码的改进)。

27.2 是的，数据报套接口也能代替流套接口用于描述字的传递。使用数据报套接口时，即使某个子进程过早终止了，父进程也不会在流管道的本地端接收到文件结束符，不过父进程可以使用 SIGCHLD 达到这个目的。大家应意识到这里能使用 SIGCHLD 的情形跟 25.7 节中的 icmpd 守护进程是有差别的：后者情形下，客户和服务器间并不存在父子关系，因此流管道上的文件结束符是服务器检测某个客户已消失的唯一办法。

第 28 章习题答案

28.1 通常没有用，因为把 qlen 成员设置成非零值的唯一应用程序类型是面向连接的服务器程序(例如 TCP)。但是服务器通常捆绑在各自的众所周知端口上，而不是由系统选择一个临时端口。例外情况之一是 RPC(远程过程调用)服务器，它捆绑在临时端口上，然后使用 RPC 端口映射器(port mapper)登记该端口。

28.2 接收到响应 SYN 的 ICMP 目的地不可达消息并不是致命的错误。TCP 应重发若干次 SYN，或等到它的重传定时器超时为止。

28.3 给已经接收到 RST 的套接口执行 write 会产生 SIGPIPE 信号。如果进程不处理这个信号，缺省处理就是终止进程。如果进程忽略这个信号，write 将返回 EPIPE 错误。

第 29 章习题答案

29.1 getnetconfig 函数是线程安全的。setnetconfig 动态分配用于存放 netconfig 结构以及由它所指向各数组的内存空间。这部分内存空间最后由 endnetconfig 释放。注意在调用 endnetconfig 后不要访问由 getnetconfig 返回的 netconfig 结构，因为那部分内存空间已经释放了。

29.2 图 E.19 给出了这个程序。

```
1 #include    "unpxti.h"
2 int
3 main(int argc, char **argv)
4 {
5     int    fd;
6     struct t_call    *tcall;
7     fd = T_open(XTI_TCP, O_RDWR, NULL);
8     tcall = T_alloc(fd, T_CALL, T_ALL);
9     printf("first t_alloc OK\n");
10    tcall = T_alloc(fd, T_CALL, T_ADDR | T_OPT | T_UDATA);
```

```
11      printf("second t_alloc OK\n");
12      exit(0);
13  }
```

图 E.19 比较 T_ALL 与指定每个 netbuf 结构[debug/test06.c]

执行这个程序的输出如下：

```
alpha % test06
first t_alloc OK
t_alloc error;system error: invalid argument
```

TCP 端点不支持连接请求中携带用户数据(图 28.1 中 connect 行的值为－2)。当指定 T_ALL 时，t_alloc 会跳过不受支持的结构，因此我们看到图 29.5 中 udata.len 被置成 0，udata.buf 则被置成空指针。然而，如果我们显式地逐项指定所有 3 个 netbuf 结构作为 t_alloc 的第 3 个参数以取代 T_ALL，那么只要有一个得不到支持就会返回错误。这也是坚持使用 T_ALL 的另外一个原因。

29.3 该函数在只给出结构指针的情况下无法辨别其类型。如果 t_free 只释放这个结构，其类型参数是不需要的。然而既然它要进入该结构并释放其内部的 netbuf 结构指向的任何缓冲区，它就必须知道结构的类型。

29.4 我们不能保证结构内部各成员的顺序。

第 30 章习题答案

30.1 这种技术不再适用，因为整数通常以 32 位存储，而指针却需要 64 位(图 1.17)。

30.2 Posix.1 的 PATH_MAX 常值不包括结尾的空字节。

30.3 图 E.20 中我们画了 4 字节的数组和其中要移动的 2 字节存储区，它表明源和目标是重叠的。

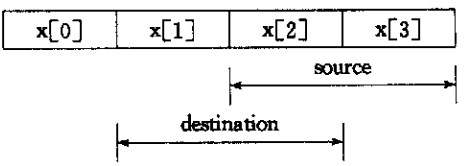


图 E.20 拷贝操作的源和目标重叠

如果拷贝是从源的开头到目标的开头进行的，用 C 语言书写的片段如下：

```
while (ngytes--)
    *dst++ = *src++;
```

那么所执行的赋值语句为：

```
x[1] = x[2];
x[2] = x[3];
```

它是正确的。然而如果拷贝在另一个方向上进行，用 C 语言书写的片段如下：

```
src += nbytes;
dst += nbytes;
while (nbytes--)
```

```
*--dst = *--src;
```

那么所执行的赋值语句却是：

```
x[2] = x[3];
x[1] = x[2];
```

它是错误的(为什么?)。memcpy 的拷贝顺序是没有保证的；因此当源和目标存储区有重叠时必须使用 memmove。memmove 通过以"正确"方向上的拷贝来处理重叠，这得取决于源和目标的关系。

30.4 图 E.21 给出了两个函数 do_parent 和 do_child。在此之前跟图 E.15 相比唯一的变化是用 #include unpxti.h 代替了 #inclnde unp.h。

```
35 void
36 do_parent(void)
37 {
38     int     qlen, j, k, junk, fd[MAXBACKLOG + 1];
39     struct t_call tcall;

40     Close(cfd);
41     Signal(SIGALRM, parent_alrm);

42     for (qlen = 0; qlen <= 14; qlen++) {
43         printf("qlen = %d: ", qlen);
44         Write(pfd, &qlen, sizeof(int));      /* tell child value */
45         Read(pfd, &junk, sizeof(int));       /* wait for child */

46         for (j = 0; j <= MAXBACKLOG; j++) {
47             fd[j] = T_open(XTI_TCP, O_RDWR, NULL);
48             T_bind(fd[j], NULL, NULL);

49             tcall.addr.maxlen = sizeof(serv);
50             tcall.addr.len = sizeof(serv);
51             tcall.addr.buf = &serv;
52             tcall.opt.len = 0;
53             tcall.udata.len = 0;

54             alarm(2);
55             if (t_connect(fd[j], &tcall, NULL) < 0) {
56                 if (errno != EINTR)
57                     err_xti("t_connect error, j = %d", j);
58                 printf("timeout, %d connections completed\n", j-1);
59                 for (k = 1; k < j; k++)
60                     T_close(fd[k]);
61                 break;      /* next value of qlen */
62             }
63             alarm(0);
64         }
65         if (j > MAXBACKLOG)
66             printf("%d connections? \n", MAXBACKLOG);
67     }
68     qlen = -1;          /* tell child we're all done */
69     Write(pfd, &qlen, sizeof(int));
70 }

71 void
72 do_child(void)
73 {
```

```
74    int       listenfd, qlen, junk;
75    struct t_bind     tbind, tbindret;
76    Close(pipefd[1]);
77    Read(cfd, &qlen, sizeof(int));      /* wait for parent */
78    while (qlen >= 0) {
79        listenfd = T_open(XTI_TCP, O_RDWR, NULL);
80        tbind.addr.maxlen = sizeof(serv);
81        tbind.addr.len = sizeof(serv);
82        tbind.addr.buf = &serv;
83        tbind.qlen = qlen;
84        tbindret.addr.maxlen = 0;
85        tbindret.addr.len = 0;
86        T_bind(listenfd, &tbind, &tbindret);
87        printf("returned qlen = %d, ", tbindret.qlen);
88        fflush(stdout);
89        Write(cfd, &junk, sizeof(int));      /* tell parent */
90        Read(cfd, &qlen, sizeof(int));      /* just wait for parent */
91        T_close(listenfd);      /* closes all queued connections too */
92    }
93 }
```

图 E.21 确定不同的 qlen 值对应的真正已排队连接数[debug/qlen.c]

第 33 章习题答案

33.1 我们假定流关闭时协议的缺省处理就是顺序释放，这对 TCP 来说是正确的。

附录F 参考文献

所有RFC都可以通过电子邮件、匿名FTP或WWW免费获取，起始点之一是http://www.ietf.org。目录ftp://ftp.isi.edu/in-notes是一个存放RFC的位置。

标记为"Internet Draft(因特网草案)"的条目是来自IETF(因特网工程任务攻坚组)的进展中的著作。这些草案在出版后六个月就完成使命。因此它们或者在本书付印后就有新版本出现，或者已经作为RFC出版。它们跟RFC一样，也可以免费从因特网上获取。访问因特网草案的起始点之一也是http://www.ietf.org。

要是能找到本参考文献所引用的论文或报告的电子文档的话，我们就给出它们的URL。需留意的是URL可能随时间而变动，因此读者应经常访问作者在http://www.kohala.com/~rstevens的WWW主页，检查本书的最新勘误表。

Albitz, P., and Liu, C. 1997, DNS and BIND, second Edition. O'Reilly & Associates, Sebastopol, Calif.

Almquist, P. 1992. "Type of Service in the Internet Protocol Suite," RFC 1349, 28 pages (July).

如何使用IPv4头部的服务类型字段。

Atkinson, R. J. 1995a. "IP Authentication Header," RFC 1826, 13 pages(Aug.),

Atkinson, R. J. 1995b. "IP Encapsulating Security Payload (ESP)," RFC 1827, 12 pages(Aug.).

Baker, F., ed. 1995. "Requirements for IP Version 4 Routers," RFC 1812, 175 pages (June).

Borman, D. A. 1997a. "Re: Frequency of RST Terminated Connections," January 30, 1997, end2end-interest mailing list.

http://www.kohala.com/~rstevens/borman.97jan30.txt

Borman, D. A. 1997b. "TCP and UDP over IPv6 Jumbograms," RFC 2147, 3 pages (May).

Borman, D. A. 1997c. "Re: SYN/RST cookies," June 6, 1997, tcp-impl mailing list.

http://www.kohala.com/~rstevens/borman.97jan06.txt

Braden, R. T., ed. 1989. "Requirements for Internet Hosts—Communication Layers," RFC 1122, 116 pages(Oct.).

主机要求RFC(Host Requirements RFC)的前半部分。这部分的内容包括链路层、IPv4、ICMPv4、IGMPv4、ARP、TCP及UDP。

Braden, R. T. 1992a. "TIME-WAIT Assassination Hazards in TCP", RFC 1337, 11 pages(May).

Braden, R. T. 1992b. "Extending TCP for Transactions—Concepts," RFC 1379, 38 pages(Nov.).

Braden, R. T. 1993. "TCP Extensions for High Performance: AN Update," Internet Draft, 10 pages(June).

http://www.kohala.com/~rstevens/tcplw-extensions.txt

这是对 RFC 1323[Jacobson, Braden, and Borman 1992]的更新，RFC 1323 从未作为 RFC 出版过，但对它的更新应该会出版。

Braden, R. T. 1994. "T/TCP—TCP Extensions for Transactions, Functional Specification," RFC 1644, 38 pages (July).

Braden, R. T. Borman, D. A., and Partridge, C. 1988. "Computing the Internet Checksum," RFC 1071, 24 pages (Sept).

Braden, S. 1996. "The Internet Standards Process—Revision3," RFC 2026, 36 pages (Oct.).

Butenhof, D. R. 1997. Programming with POSIX Threads. Addison-Wesley, Reading, Mass.

CERT. 1996a. "UDP Port Denial-of-Service Attack," Advisory CA-96.01, Computer Emergency Response Team, Pittsburgh, Pa. (Feb.).

ftp://info.cert.org/pub/cert_advisories/CA-96.01.UDP_service_denial

CERT. 1996b. "TCP SYN Flooding and IP Spoofing Attacks," Advisory CA-96.21, Computer Emergency Response Team, Pittsburgh, Pa. (Sept.).

ftp://info.cert.org/pub/cert_advisories/CA-96.21.tcp_syn_flooding

Cheswick, W. R., and Bellovin, S. M. 1994. Firewalls and Internet Security: Repelling the Wily Hacker. Addison-Wesley, Reading, Mass.

Comer, D. E., and Lin, J. C. 1994. "TCP Buffering and Performance Over an ATM Network," Purdue Technical Report CSD-TR 94-026, Purdue University, West Lafayette, Ind. (Mar.).

ftp://gwen.cs.purdue.edu/pub/lin/TCP.atm.ps.Z

Conta, A., and Deering, S. E. 1995. "Internet Control Message Protocol(ICMPv6) for the Internet Protocol Version 6(IPv6) Specification," RFC 1885, 20 pages(Dec.).

Crawford, M. 1996a. "A Method for the Transmission of IPv6 Packets over Ethernet Networks," RFC 1972, 4 pages(Aug.).

Crawford, M. 1996b. "A Method for the Transmission of IPv6 Packets over FDDI Networks," RFC 2019, 6 pages(Oct.).

Deering, S. E. 1989. "Host Extensions for IP Multicasting," RFC 1112, 17 pages

(Aug.).

Deering, S. E. and Hinden, R. 1995. "Internet Protocol, Version 6 (IPv6) Specification," RFC 1883, 37 pages (Dec.).

Dewar, R. B. K., and Smosna, M. 1990. Microprocessors: A Programmer's View. McGraw-Hill, New York.

Eriksson, H. 1994. "MBONE: The Multicast Backbone," Communications of the ACM, vol. 37, no. 8, pp. 54-60 (Aug.).

Fenner, W. C. 1997. Private Communication.

Fuller, V., Li, T., Yu, J. Y., and Varadhan, K. 1993. "Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy," RFC 1519, 24 pages (Sept.).

Garfinkel, S. L., and Spafford, E. H. 1996. Practical UNIX and Internet Security, Second Edition. O'Reilly & Associates, Sebastopol, Calif.

Gierth, A. 1996. Private Communication.

Gilligan, R. E., Thomson, S., Bound, J., and Stevens, W. R. 1997. "Basic Socket Interface Extensions for IPv6," RFC 2133, 32 pages (Apr.).

Handley, M. 1996. "SAP: Session Announcement Protocol," Internet Draft, 14 pages (Nov.).

draft-ietf-mmusic-sap-00.txt

Handley, M., and Jacobson, V. 1997. "SDP: Session Description Protocol," Internet Draft (Mar.).

draft-ietf-mmusic-sdp-03.txt

Hinden, R., and Deering, S. E. 1995. "IP Version 6 Addressing Architecture," RFC 1884, 18 pages (Dec.).

Hinden, R., and Deering, S. E. 1997. "IP Version 6 Addressing Architecture," Internet Draft, 25 pages (July).

draft-ietf-ipngwg-addr-arch-v2-02.txt

本文档在成为 RFC 之后应替换 RPC 1884[Hinden and Deering 1995]。

Hinden, R., Fink, R., and Postel, J. B. 1997. "IPv6 Testing Address Allocation," Internet Draft, 4 pages (July).

draft-ietf-ipngwg-testv2-addralloc-01.txt

本文档在成为 RFC 之后应替换 RPC 1897[Hinden and Postel 1996]。

Hinden, R., O'Dell, M., and Deering, S. E. 1997. "An IPv6 Aggregatable Global Unicast Address Format," Internet Draft, 9 pages (July).

draft-ietf-ipngwg-unicast-aggr-02.txt

本文档在成为RFC之后应替换RPC 2073[Rekhter et al. 1997]。

Hinden, R., and Postel, J. B. 1996, "IPv6 Testing Address Allocation," RFC 1997, 4 pages(Jan.).

IEEE. 1996. "Information Technology—Portable Operationg System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]," IEEE Std 1003.1, 1996 Edition, Institute of Electrical and Electronics Enginerrs, Piscataway, N.J. (July).

这个版本的Posix.1含有1990基本API、1003.1b实时扩展(1993)、1003.1c pthreads(1995)以及1003-li技术性更正(1995)。它同时也是国际标准ISO/IEC 9945-1: 1996(E)。IEEE正式标准和草案标准的订购信息可从http://www.ieee.org获取。

IEEE. 1997a. "Infromation Technology—Portable Operationg System Interface(POSIX)—Part xx: Protocol Independent Interfaces(PII)," P1003.1g/D6.6, Institute of Electrical and Electronics Enginerrs, Piscataway, N.J. (Mar.).

这应该是Posix.1g的最终草案,不幸的是IEEE并没有将它放到因特网上在线发布。

IEEE. 1997b. Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority. Institute of Electrical and Electronics Engineers, Piscataway, N.J.

http://standards.ieee.org/db/oui/tutorials/EUI64.html

Jacobson, V. 1988. "Congestion Avoidance and Control," Computer communication Review, vol. 18, no. 4, pp. 314-329(Aug).

ftp://ftp.ee.lbl.gov/papers/congavoid.ps.z

描述TCP的慢启动和拥塞避免算法的经典论文。

Jacobson, V. 1994. "Re: half baked anycastoffidea...," June 27, 1994, end2end-interest mailing list.

http://www.kohala.com/~rstevens/vanj.94jun27.txt

Jacobson, V., Braden, R. T., and Borman, D. A. 1992. "TCP Extensions for High Performance," RFC 1323, 37 pages(May).

描述窗口规模选项、时间戳选项、PAWS算法以及添加它们的理由。[Braden 1993]是对它的更新。

Jacobson, V., Braden, R. T., and Zhang, L. 1990. "TCP Extensions for High-Speed Paths," RFC 1185, 21 pages(Oct.).

Josey, A., ed. 1997. Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification. Prentice Hall, Upper Saddle River, N.J.

Joy, W.N. 1994. Private Communication.

Karn, P., and Partridge, C. 1987. "Improving Round-Trip Time Estimates in Reliable Transport Protocols,"Computer Communication Review, vol. 17, no. 5, pp. 2—7(Aug).

ftp://sics.se/users/craig/karn-partridge.ps

Katz, D. 1993. "Transmission of IP and ARP over FDDI Network," RFC 1390, 11 pages(Jan.).

Katz, D. 1997. "IP Router Alert Option," RFC 2113, 4 pages(Feb.).

Katz, D., Atkinson, R. J., Partridge, C., and Jackson, A. 1997. "IPv6 Router Alert Option," Internet Draft, 5 Pages(June).

draft-ietf-ipngwg-ipv6-router-alert-02.txt

Kent, S. T. 1991. "U.S. Department of Defense Security Options for the Internet Protocol," RFC 1108, 17 pages(Nov.).

Kent, S. T., and Atkinson, R. J. 1997a. "IP Authentication Header," Internet Draft, 22 Pages(July).

draft-ietf-ipsec-auth-header-01.txt

Kent, S. T., and Atkinson, R. J. 1997b. "IP Encapsulating Security Payload(ESP)," Internet Draft, 19 pages(July).

draft-ietf-ipsec-esp-v2-00.txt

Kernighan, B. W., and Pike, R. 1984. The UNIX Programming Environment. Prentice Hall, Englewood Cliffs, N. J.

Kernighan, B. W., and Ritchie, D. M. 1988. The C Programming Language, Second Edition. Prentice Hall, Englewood Cliffs, N. J.

Korn, D. G., and Vo, K. P. 1991. "SFIO: Safe/Fast String/File IO", Proceedings of the 1991 Summer USENIX Conference, pp. 235-255, Nashville, Tenn.

对标准I/O函数库替代方法的描述。其源代码可从http://www.research.att.com/sw/tools/reuse获取。

Lanciani, D. 1996. "Re: sockets: AF_INET vs. PF_INET," Message-ID: <3561@news.IPSWITCH.COM>, Usenet, comp.protocols.tcp-ip Newsgroup(Apr.).

http://www.kohala.com/~rstevens/lanciani.96apr10.txt

Maslen, T. M. 1997. "Re: gethostbyXXXX() and Threads," Message-ID <maslen.862463530@shellx>, Usenet, comp.programming.threads Newsgroup(May).

http://www.kohala.com/~rstevens/maslen.97may01.txt

Maufer, T., and Semeria, C. 1997. "Introduction to IP Multicast Routing," Internet Draft(Mar.).

draft-ietf-mboned-intro-multicast-02.txt

McCann, J., Deering, S. E., and Mogul, J. C. 1996. "Path MTU Discovery for IP version 6," RFC 1981, 15 Pages(Aug.).

McCanne, S., and Jacobson, V. 1993. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," proceedings of the 1993 Winter USENIX Conference, pp. 259-269, San Diego. Calif.

ftp://ftp.ee.lbl.gov/papers/bpf-usenix93.ps.z

McDonald, D. L. 1997. "A Simple IP Security API Extension to BSD Sockets," Internet Draft, 12 pages(Mar.).

draft-mcdonald-simple-ipsec-api-01.txt

McDonald, D. L., Phan, B. G., and Atkinson, R. J. 1996. "A Socket-Based Key Management API(and surrounding infrastructure)," Proceedings of the INET' 96 Conference, PP. 53-63(June), Montreal, Quebec.

http://www.cs.hut.fi/ssh/crypto/pf-key.ps

McDonald, D. L., Metz, C. W., and Phan, B. G. 1997. "PF_KEY Key Management API, Version 2," Internet Draft, 67 pages(July).

draft-mcdonald-pf-key-v2-03.txt

McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley, Reading, Mass.

Meyer, D. 1997. "Administratively Scoped IP Multicast," Internet Draft, 7 pages (June).

draft-ietf-mboned-admin-ip-space-03.txt

Mills, D. L. 1992. "Network Time Protocol(Version 3): Specification, Implementation, and Analysis," RFC 1305, 113 pages(Mar.).

Mills, D. L. 1996. "Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI," RFC 2030, 18 pages(Oct.).

Mogul, J. C., and Deering, S. E. 1990. "Path MTU Discovery," RFC 1191, 19 pages (Apr.).

Mogul, J. C., and Postel, J. B. 1985. "Internet Standard Subnetting Procedure," RFC 950, 18 pages (Aug.).

Nemeth, E. 1997. Private Communication.

Open Group, The. 1997. CAE Specification, Networking Services(XNS). Issue 5. The Open Group, Reading, Berkshire, U. K.

这是 Unix 98 中套接口和 XTI API 的规范。本手册还在其附录中描述了 XTI 在 NetBIOS、OSI 协议、SNA 及 Netware IPX 和 SPX 协议中的使用。另有三个附录介绍套接口和 XTI API 在 ATM 中的使用。

Partridge, C., Mendez, T., and Milliken, W. 1993. "Host Anycasting Service," RFC 1546, 9 pages(Nov.).

Partridge, C., and Pink, S. 1993. "A Faster UDP," IEEE|ACM Transactions on Networking, vol. 1, no. 4, pp. 429-440(Aug.).

Paxson, V. 1996. "End-to-End Routing Behavior in the Internet," Computer Communication Review, vol. 26, no. 4, pp. 25-38(Oct.).

ftp://ftp.ee.lbl.gov/papers/routing.SIGCOMM.ps.z

Piscitello, D. M. 1994. "FTP Operation Over Big Address Records (FOOBAR)," RFC 1639, 5 pages(June).

Plauger, P. J. 1992. The Standard C Library. Prentice Hall, Englewood Cliffs, N. J.

Postel, J. B. 1980. "User Datagram Protocol," RFC 768, 3 pages(Aug.).

Postel, J. B. ed. 1981a. "Internet Protocol," RFC 791, 45 pages(Sept.).

Postel, J. B. 1981b. "Internet Control Message Protocol," RC 792, 21 pages(Sept.).

Postel, J. B. ed. 1981c. "Transmission Control Protocol," RFC 793, 85 pages(Sept.).

Pusateri, T. 1993. "IP Multicast Over Token-Ring Local Area Networks," RFC 1469, 4 pages (June).

Rago. S. A. 1993, UNIX System V Network Programming. Addison-Wesley, Reading, Mass.

Rekhter, Y., Lothberg, P., Hinden, R., Deering, S. E., and Postel, J. B. 1997, "An IPv6 ProviderBased Unicast Address Format," RFC 2073, 7 pages(Jan.).

Reynolds, J. K., and Postel, J. B. 1994. "Assigned Numbers," RFC 1700, 230 pages (Oct.).

对RFC中所包含信息的改动可能在对整个RFC更新前发生。本RFC中所有的信息表都取自目录ftp://ftp.isi.edu/in-notes/iana/assignments,而这些文件是随相关信息的变动而更新的。本RFC给出了该目录下文件的URL,要取得最新信息就应该访问这些文件。

Ritchie, D. M. 1984. "A Stream Input-Output System," AT&T Bell Laboratories Technical Journal, vol. 63, no, 8, pp. 1897-1910(Oct.).

Salus, P. H. 1994. A Quarter Century of Unix. Addison-Wesley, Reading, Mass.

Salus, P. H. 1995. Casting the Net: From ARPANET to Internet and Beyond. Addison-Wesley, Reading, Mass.

Schimmel, C. 1994. UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers. Addison-Wesley, Reading, Mass.

Srinivasan, R. 1995. "XDR: External Data Representation Standard," RFC 1832, 24 pages(Aug.).

Stevens, W. R. 1992. Advanced Programming in the UNIX Environment, Addison-Wesley, Reading, Mass.

Unix 编程的所有细节。本书称之为 APUE。

Stevens, W. R. 1994. TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley, Reading, Mass.

对网际协议的完整介绍。本书称之为 TCPv1。

Stevens, W. R. 1996. TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols. Addison-Wesley, Reading, Mass.

本书称之为 TCPv3。

Stevens, W. R., and Thomas, M. 1997. "Advanced Sockets API for IPv6," Internet Draft(July).

draft-stevens-advanced-api-04.txt

Tanenbaum, A. S. 1987. Operating Systems Design and Implementation. Prentice Hall, Englewood Cliffs, N. J.

Thomas, S. 1997, "Transmission of IPv6 Packets over Token Ring Networks," Internet Draft, 10 pages (June).

draft-ietf-ipngwg-trans-tokenring-00.txt

Thomson, S., and Huitema, C. 1995. "DNS Extensions to Support IP version 6," RFC 1886, 5 pages(Dec.).

Torek, C. 1994. "Re: Delay in re-using TCP/IP port," Message-ID <199501010028.QAA16863@elf.bsdi.com>, Usenet, comp.unix.wizards Newsgroup(Dec.).

http://www.kohala.com/~rstevens/torek.94dec31.txt

Unix International. 1991. "Data Link Provider Interface Specification," Revision 2.0.0, Unix International, Parsippany, N. J. (Aug.).

http://www.kohala.com/~rstevens/dlpi.2.0.0.ps

本规范的较新版本可从 Open Group 的网页 http://www.rdg.opengroup.org/pubs/catalog/web.htm 在线获取。

Unix International. 1992a. "Network Provider Interface Specification," Revision 2.0.0, Unix International, Parsippany, N. J. (Aug.).

http://www.kohala.com/~rstevens/npi.2.0.0.ps

Unix International. 1992b. "Transport Provider Interface Specification," Revision 1.5, Unix International, Parsippany, N. J. (Dec.).

http://www.kohala.com/~rstevens/tpi.1.5.ps

本规范的较新版本可从 Open Group 的网页 http://www.rdg.opengroup.org/pubs/catalog/web.htm 在线获取。

http://www.rdg.opengroup.org/pubs/catalog/web.htm.

Vixie, P. A. 1996. Private Communication.

Wright, G. R., and Stevens, W. R. 1995. TCP/IP Illustrated, Volume 2: The Implementation. AddisonWesley, Reading, Mass.

网际协议在4.4BSD-Lite操作系统上的实现。本书称之为TCPv2。

附录 G　函数和宏定义索引表

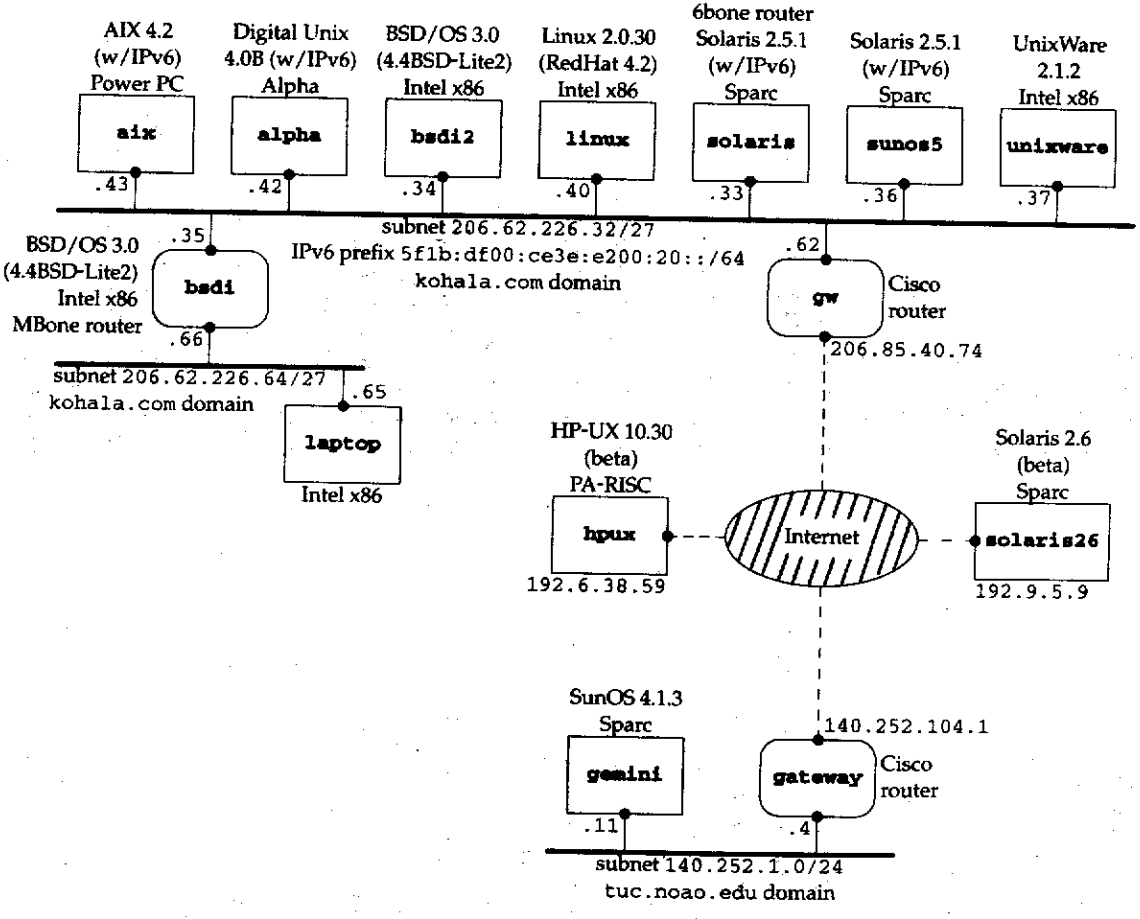
accept 83
bcmp 57
bcopy 57
bind 75
bzero 57
close 89
closelog 286
CMSG_xxx 311
connect 73
connect_nonb 349
connect_timeo 298
daemon_inetd 294
daemon_init 287
dg_send_recv 468
endnetconfig 669
endnetpath 670
err_doit 791
err_dump 791
err_msg 791
err_quit 791
err_ret 791
err_sys 791
execxxx 85
fcntl 175
fork 85
freeaddrinfo 237
free_ifi_info 392
gai_strerror 236
getaddrinfo 232
gethostbyaddr 214
gethostbyaddr_r 259
gethostbyname 207
gethostbyname2 212
gethostbyname_r 259
gethostname 216
get_ifi_info 367,392
getmsg 730
getnameinfo 255
getnetconfig 669
getnetpath 670
getpeername 90
getpmsg 730
getservbyname 216
getservbyport 216
getsockname 90
getsockopt 151
gf_time 344
heartbeat_cli 495
heartbeat_serv 499
host_serv 241
htonl 56
htons 56
ICMP6_FILTER_xxx 561
if_freenameindex 395
if_indextoname 395
if_nameindex 395
if_nametoindex 395
IN6_IS_ADDR_xxx 228
in_cksum 671
inet6_option_xxx 551
inet6_rthdr_xxx 553
inet_addr 58
inet_aton 58
inet_ntoa 58
inet_ntop 59

inet_pton 59
ioctl 731,363
isfdtype 67
listen 77
mcast_get_if 427
mcast_get_loop 427
mcast_get_ttl 427
mcast_join 427
mcast_leave 427
mcast_set_if 427
mcast_set_loop 427
mcast_set_ttl 427
memcmp 58
memcpy 58
memset 58
my_addrs 215
netdir_getbyaddr 672
netdir_getbyname 671
ntohl 56
htohs 56
openlog 286
poll 144
pselect 143
pthread_cond_broadcast 535
pthread_cond_signal 533
pthread_cond_timedwait 535
pthread_cond_wait 533
pthread_create 511
pthread_detach 512
pthread_exit 513
pthread_getspecific 524
pthread_join 512
pthread_key_create 523
pthread_mutex_lock 531
pthread_mutex_unlock 531
pthread_once 523
pthread_self 512
pthread_setspecific 524
putmsg 730
putpmsg 730
readable_timeo 301
read_fd 329
readline 64
readn 64
readv 304
recv 302
recvfrom 180
recvmsg 305
rtt_init 470
rtt_minmax 470
rtt_newpack 471
rtt_start 471
rtt_stop 472
rtt_timeout 472
rtt_ts 471
select 126
send 302
sendmsg 305
sendto 180
setnetconfig 669
setnetpath 670
setsockopt 151
shutdown 135
signal 100
sockatmark 488
sock_bind_wild 63
sock_cmp_addr 63
sock_cmp_port 63
socket 70
socketpair 321
sockfd_to_family 91
sock_get_port 63
sock_ntop 62
sock_ntop_host 63
sock_set_addr 63

sock_set_port 63
sock_set_wild 63
sysctl 388
syslog 388
t_accept 685
t_alloc 673
t_bind 656
t_connect 658
tcp_connect 676
tcp_listen 683
t_error 654
t_free 673
t_getinfo 742
t_getprotaddr 675
t_getstate 742
t_listen 682
t_look 660
t_open 651
t_optmgmt 717
t_rcv 658
t_rcvconnect 741
t_rcvdis 662
t_rcvrel 661
t_rcvreldata 746
t_rcvudata 700
t_rcvuderr 704
t_rcvv 745
t_rcvvudata 745
t_snd 658
t_snddis 662
t_sndrel 661
t_sndreldata 746
t_sndudata 700
t_sndv 746
t_sndvudata 746
t_strerror 654
t_sync 743
t_unbind 745
tv_sub 567
udp_client 700
udp_connect 252
udp_server 706
uname 215
wait 105
waitpid 105
write_fd 331
writen 64
writev 304
xti_accept 686
xti_getopt 721
xti_ntop 675
xti_rdwr 666
xti_setopt 723

附录H 结构定义索引表

addrinfo	232	servert	216
cmsghdr	309	sockaddr	50
fcred	332	sockaddr_dl	380
hostent	207	sockaddr_in	50
if_nameindex	395	sockaddr_in6	51
in6_pktinfo	479	sockaddr_un	319
in_addr	47	strbuf	729
in_pktinfo	455	t_bind	656
iovec	304	t_call	658
ip_mreq	424	t_discon	663
ipoption	543	timespec	143
ipv6_mreq	424	timeval	127
linger	159	t_info	651
msghdr	305	t_iovec	745
nd_addrlist	671	t_kpalive	716
nd_hostserv	671	t_linger	715
nd_hostservlist	672	t_opthdr	713
netbuf	655	t_optmgmt	719
netconfig	670	t_uderr	704
pcap_pkthdr	616	t_unitdata	700
pollfd	144	utsname	215



Hosts and networks used for most examples in the text.

附录I 中英文对照词汇表[①]

6bone(IPv6 backbone)	6bone(IPv6 主干)
abortive release,XTI	夭折性释放
absolute name,DNS	绝对名字
absolute requirement,XTI options	绝对要求
absolute time	绝对时间
acknowledgment flag,TCP header	确认标志
active [close, open]	主动[打开,关闭]
active socket	主动套接口
address request	地址请求
ARP(Address Resolution Protocol)	ARP(地址解析协议)
ARP cache operations, ioctl functions	ARP 高速缓存操作
administratively scoped multicast address	管理上划分范围的多播地址
aggregatable global unicast address	可聚集全局单播地址
alias address	别名地址
alignment	对齐,对准
all-hosts multicast group	所有主机多播组
all-nodes multicast group	所有节点多播组
all-routers multicast group	所有路由器多播组
ancillary data [object]	辅助数据[对象]
anycast | anycasting	任播
API(Application programming interface)	API(应用程序编程接口)
application	应用(泛指) 应用程序(特指程序本身,静态) 应用进程(特指运行中的程序,动态)
application protocol	应用协议
argument passing,thread	参数传递
AS(autonomous system)	AS(自治系统)
asynchronous [error, events, I/O]	异步[错误,事件,I/O]
ATM(Asynchronous Transfer Mode)	ATM(异步传输模式)
automatic tunnel	自动生成的隧道

① 注：英文术语一栏中逗号后内容为本术语出现处或定义处。

AVP(audio/video profile)	AVP(音频/视频定制文件)
bandwidth-delay product	带宽-延迟积
batch input	批量输入
Berkeley-derived implementation	源自 Berkeley 的实现
BGP(Border Gateway Protocol)	BGP(边界网关协议)
backlog	backlog{不译,因为无明确定义}
big-endian byte order	大端字节序
BIND(Berkeley Internet Name Domain)	BIND(Berkekey 因特网名字域系统)
bind | binding	捆绑
blocking I/O model	阻塞式 I/O 模型
BOOTP(Bootstrap Protocol)	BOOTP(引导协议)
bound	绑定
Bourne shell	Bourne shell, B shell, bsh 同义
BPF(BSD Packet Filter)	BPF(BSD 分组过滤器)
broadcast address	广播地址
broadcast [flooding, storm]	广播[泛滥,风暴]
BSD(Berkeley Software Distribution)	BSD(Berkeley 软件发布)
buffer [size]	缓冲区[大小]
byte manipulation functions	字节操纵函数
byte order	字节序
byte-stream protocol	字节流协议
C standard	C 标准
CDE(Common Desktop Environment)	CDE(公共桌面环境)
CERT (Computer Energency Response Team)	CERT(计算机紧急事件反应小组)
checksum	校验和
CIDR(classless interdomain routing)	CIDR(无类域间路由)
classless address	无类地址
client	客户{特指运行中的客户程序,动态} 客户程序{特指程序代码本身,静态} 客户主机{特指客户进程所在主机,罕见}
client-server	客户-服务器
clock resolution	时钟分辨率
clock time	流逝时间
CNAME(canonical name record),DNS	CNAME(规范名字记录)
code field,ICMP	代码域,代码字段
code(v.) | coding	编码
communications endpoint,XTI	通信端点
communications provider,XTI	通信提供者

completed connection [queue]	已完成连接[队列]
completely duplicate binding	完全重复捆绑
concurrent programming	并发编程
concurrent server	并发服务器(程序)
condition variable	条件变量
constant	常数,常量,常值
configured tunnel	配置成的隧道
congestion avoidance	拥塞避免
connect indication	连接指示
connected TCP sockets	已连接 TCP 套接口
connected UDP sockets	已连接 UDP 套接口
connection [abort, establishment, termination], TCP	连接[夭折,建立,终止]
connectionless	无连接(的)
connection-oriented	面向连接(的)
continent-local multicast scope	大洲局部多播范围,局部于大洲的多播范围
control information	控制信息
conventions	约定
copy-on-write	写时拷贝
CR(carriage return)	CR(回车)
crashing and rebooting of server host	服务器主机崩溃并重启
crashing of server host	服务器主机崩溃
credentials	凭证
creeping featurism	卑躬屈膝的特征主义
CSRG(Computer Systems Research Group)	计算机系统研究组
daemon	守护进程
data formats	数据格式
datagram service	数据报服务
datagram socket	数据报套接口
datagram truncation, UDP	数据报截断
datalink socket address structure	数据链路套接口地址结构
datatype	数据类型
DCE(Distributed Computing Environment)	DCE(分布式计算环境)
deadlock	死锁
debugging techniques	调试技术,排错技术
deep copy	深拷贝
delayed ACK	延滞 ACK
delta time	相差时间

denial-of-service attack	拒绝服务型攻击
descriptor	描述字
design alternatives, client-server	供选择的设计方法,其他设计方法
destination	目的(地)
destination [address, options]	目的[地址,选项]
destination unreachable	目的地不可达(错误)
destructor function	析构函数
detached thread	脱离的线程,已脱离线程
device	设备
DF(don't fragment flag), IP header	DF(不分片标志)
DHCP(Dynamic Host Configuration Protocol)	DHCP(动态主机配制协议)
diskless node	无盘节点
DLPI(Data Link Provider Interface)	DLI(数据链路提供者接口)
DNI(Detailed Network Interface)	DNI(详尽网络接口)
DNS(Domain Name System)	DNS(域名系统)
domain	域
dotted-decimal notation	点分十进制数记法
double buffering	双缓冲
drive	驱动器
driver, streams	驱动程序
dual-stack host	双(重)协议栈主机
duplicate	拷贝/重复分组(n.),复制/拷贝(v.)
dynamic port	动态端口
echo program	回射程序
echo reply	回射应答
echo request	回射请求
endpoint state, XTI	端点状态
end-to-end, XTI options	端到端
environment variable	环境变量
EOL(end of option list)	EOL(选项列表结束选项)
ephemeral port	临时端口
Epoch	纪元
error	错,错误
error message	出错消息
ESP(Encapsulating security payload)	ESP(封装安全有效负载)
Ethernet	以太网
ETSDU (expedited transport service data unit)	ETSDU(经加速的传输服务数据单元)

EUI(Extended unique identifier)	EUI(经扩展的唯一标识符)
examples road map,client-server	例子导读图
expedited data	(经)加速数据
exponential backoff	指数回退
extension headers	扩展头部
FAQ(frequently asked question)	FAQ(常问问题)
FDDI(Fiber Distributed Data Interface)	FDDI(光纤分布式数据接口)
FIFO(first in first out)	FIFO(先进先出)
file table	文件表
filter(v.) | filtering	过滤
FIN(finish flag,TCP header)	FIN(完成标志,TCP 分节之一)
firewall	防火墙
flex address	伸缩地址
flooding	泛滥
flow control	流控,流量控制
flow information	流信息
flow label field,IPv6	流标域,流标字段
format prefix	格式前缀
formats	格式
FQDN(fully qualified domain name)	FQDN(全限定域名)
fragment	片段
fragment offset field,IPv4	片段偏移域,片段偏移字段
fragment(v.) | fragmentation	分片
frame type	帧类型
FTP(File Transfer Protocol)	FTP(文件传送协议)
fudge factor	模糊因子
full-duplex	全双工
fully buffered standard I/O system	全缓冲的标准 I/O 系统
function	函数
gather write	集中写
generic socket address structure	通用套接口地址结构
GIF(graphics interchange format)	GIF(图形交换格式)
global multicast scope	全球多播范围
group ID	组 ID
hacker	黑客
half-close	半关闭
half-open connection	半打开连接
hard error	硬错误
header	头文件(.h 文件)

	头部(各种协议数据单元的头部)
header checksum,IPv4	头部校验和
header extension length	头部扩展长度
header length field	头部长度域,头部长度字段
heartbeat functions,client-server	心博函数
high-priority,streams message	高优先级
HIPPI (High-Performance Parallel Interface)	HIPPI(高性能并行接口)
hop count,routing	跳数
hop limit	跳限
hop-by-hop options	步跳选项
host byte order	主机字节序
HTML(Hypertext Markup Language)	HTML(超文本标记语言)
HTTP(Hypertext Transfer Protocol)	HTTP(超文本传送协议)
IANA (Internet Assigned Numbers Authority)	IANA(因特网已分配数值权威机构)
ICMP(Internet Control Message Protocol)	ICMP(网际控制消息协议)
ICMP error	ICMP 错误(等于 ICMP 出错消息)
ICMPv6 (Internet Control Message Protocol version 4)	ICMPv6(网际控制消息协议版本 4)
ICMPv4 (Internet Control Message Protocol version 6)	ICMPv6(网际控制消息协议版本 6)
identificaion field,IPv4	标识域,标识字段
IEC (International Electrotechnical Commission)	IEC(国际电工委员会)
IEEE (Institute of Electrical and Electronics Engineers)	IEEE(电气与电子工程师学会)
IETF(Internet Engineering Task Force)	IETF(因特网工程任务攻坚组)
IGMP (Internet Group Management Protocol)	IGMP(网际组管理协议)
ILP32,programming model	ILP32
imperfect filtering	非完备过滤
in-band data	带内数据
incarnation	化身
incomplete connection [queue]	未完成连接[队列]
initial thread	初始线程
interface address	接口地址
interface configuration	接口配置
interface ID	接口 ID

interface index	接口索引
internet	网际(网){作为修饰词} 互联网{作为名词}
Internet	因特网{专有名词}
Internet Draft	因特网草案
interoperability	互操作性
interrupts,software	中断
I/O multiplexing,model	I/O(多路)复用
IP(Internet Protocol)	IP(网际协议)
IPC(interprocess communication)	IPC(进程间通信)
IPng(Internet Protocol next generation)	IPng(下一代网际协议)
IP spoofing	IP 欺骗
IPv4(Internet Protocol version 4)	IPv4(网际协议版本 4)
IPv4-compatible IPv6 address	IPv4 兼容的 IPv6 地址
IPv4-mapped IPv6 address	IPv4 映射的 IPv6 地址
IPX(Internetwork Packet Exchange)	IPX(网络间分组交换)
IRS(Information Retrieval Service)	IRS(信息检索服务)
ISO(International Organization for Standardization)	ISO(国际标准化组织)
ISP(Internet service provider)	ISP(因特网业务供应商)
iterative,server	迭代
ITU (International Telecommunication Union)	ITU(国际电信联合会)
joinable thread	可联接线程{反义:已脱离线程}
jumbo payload length	特大有效负载长度
jumbogram	特大报
keepalive option	保持存活选项
Korn Shell	Korn Shell,K Shell,ksh 同义
LAN(local area network)	LAN(局域网)
latency,scheduling	延迟
lazy accept	惰性接受
LF(linefeed)	LF(换行)
library	函数库,例程库,库
LIFO(last in first out)	LIFO(后进先出)
lightweight process	轻权进程
line buffered standard I/O stream	按行缓冲的标准 I/O 流
link-local address	链路局部地址
link-local [multicast group, multicast scope]	链路局部[多播组,多播范围]
listening socket	监听套接口

little-endian byte order	小端字节序
local, XTI options	本地(的)
local host IP address	本地主机 IP 地址
logical interface	逻辑接口
logical loopback	逻辑回馈
login name	登录名
long-fat pipe	长胖管道
loopback [address, interface]	回馈[地址,接口]
loopback transport provider, XTI	回馈传输提供者
lost datagrams, UDP	丢失的数据报
lost duplicate(=wandering duplicate)	迷途的重复分组
LP64, programming model	LP64
LSRR(loose source and record route)	LSRR(宽松的源与记录路径)
MAC(medium access control)	MAC(媒体访问控制)
macro	宏
main thread	主线程
MBone(multicast backbone)	MBone(多播主干)
member	成员{指结构的成员}
memory leak	内存空间遗漏
message-based interface	基于消息的接口
MF(more fragments flag, IP header)	MF(还有片段标志)
MIB(management information base)	MIB(管理信息库)
minimum link MTU	最小链路 MTU
minimum reassembly buffer size	最小重组缓冲区大小
modules, streams	模块
MSL(maximum segment lifetime)	MSL(最长分节生命期)
MSS(maximum segment size), TCP	MSS(最大分节大小)
MSS option, TCP	MSS 选项
MTU(maximum transmission unit)	MTU(最大传输单元)
multicast [address, group address, group ID]	多播[地址,组地址,组 ID]
multicast routing protocol	多播路由协议
multicast scope	多播范围
multihomed	多宿(的)
multiplexor, streams	多路复用器
mutex	互斥锁
MX(mail exchange record), DNS	MX(邮件交换记录)
Nagle algorithm	Nagle 算法
name server	名字服务器

neighbor discovery	邻居发现
network byte order	网络字节序
network services library	网络服务函数库
network topology	网络拓扑
next header field, IPv6	下一个头部域,下一个头部字段
NFS(Network File System)	NFS(网络文件系统)
nibble	4位组〔意译〕
NIS(Network Information System)	NIS(网络信息系统)
NIT(network interface tap)	NIT(网络信息龙头)
NLA(next-level aggregation identifier)	NLA(次级聚集标识)
NNTP(Network News Transfer Protocol)	NNTP(网络新闻传送协议)
NOAO (National Optical Astronomy Observatories)	NOAO(国家光学天文台)
node-local multicase scope	节点局部多播范围,局部于节点的多播范围
nonblocking I/O	非阻塞式I/O
nonlocal goto	非本地跳转
NOP(no operation)	NOP(无操作)
normal, streams message	普通
NPI(Network Provider Interface)	NPI(网络提供者接口)
NTP(Network Time Protocol)	NTP(网络时间协议)
NVT(network virtual terminal)	NVT(网络虚拟终端)
octet	八位组
operation	操作
operator	操作符,运算符
orderly release, XTI	顺序释放
organization-local multicast scope	组织局部多播范围,局部于组织的多播范围
OSF(Open Software Foundation)	OSF(开放软件基金会)
OSI(open system interconnection)	开放系统互连
OSPF (open shortest path first, routing protocol)	OSPF(开放的最短路径优先)
out-of-band data, TCP|XTI	带外数据
owner, socket	属主
oxymoron	矛盾修饰法
packet information	分组信息
packet too big, ICMP	分组太大(错)
parallel programming	并行编程
parameter problem, ICMP	参数问题(错误)
passive [close, open]	被动[关闭,打开]
passive socket	被动套接口

path MTU [discovery]	路径 MTU[发现]
payload length field,IPv6	有效负载长度域,有效负载长度字段
PCM(pulse code modulation)	PCM(脉冲代码调制)
pending error	待处理错误
perfect filtering	完备过滤
persistent connction	持续连接
physical loopback	物理回馈
piggybacking | piggyback(v.)	捎带
polling	轮询
port mapper,RPC	端口映射器
port numbers	端口号
port stealing	端口盗用
port unreachable,ICMP	端口不可达(错)
POSIX(Portable Operating System Interface)	POSIX(可移植操作系统接口)
PPP(Point-to-Point Protocol)	PPP(点到点协议)
prefix length	前缀长度
preforked server,TCP	预先派生子进程的服务器
prethreaded server,TCP	预先创建线程的服务器
priority band,streams message	优先级带
private port	私用端口
process [group ID, group leader, ID]	进程[组 ID,组长,ID]
program	程序
programming model	编程模型
promiscuous mode	混杂模式
protocol dependence	协议相关性
protocol field	协议域,协议字段
protocol independence	协议无关性
provider-based unicast address	基于提供者的单播地址
pseudoheader	伪头部
PSH(push flag, TCP header)	PSH(PUSH 标志,TCP 分节之一)
PTR(pointer record,DNS)	PTR(指针记录)
qualifier	限定词
queue length	队列长度
queued data	已排队数据
queue(v.) | queueing	排队
race condition	竞争状态
RARP(Reverse Address Resolution Protocol)	RARP(反向地址解析协议)

raw socket [input, output]	原始套接口[输入,输出]
reassembly buffer size	重组缓冲区大小
receive timeout,BPF	接收超时
record boundaries	记录边界
record route	记录路径
redirect,ICMP	重定向
reentrant	可重入(的)
reference count,descriptor	访问计数
region-local multicast scope	地区局部多播范围,局部于地区的多播范围
registered port	经注册端口
reliable datagram service	可靠数据报服务
reserved port	保留端口
resolver	解析器
resource discovery	资源发现
retransmission [time out]	重传[超时]
RFC(Request for Comments)	RFC(请求注解)
RIP (Routing Information Protocol, routing protocol)	RIP(路由信息协议)
road map,client-server examples	导航图
round robin,DNS	顺序循环
router advertisement,ICMP	路由器通告
router solicitation,ICMP	路由器征求
route(n.)	路径
route(v.) | routing	路由
routing header	路由头部
routing protocol	路由协议
routing socket	路由套接口
routing table	路由表
routing type	路由类型
RPC(remote procedure call)	RPC(远程过程调用)
RR(resource record,DNS)	RR(资源记录)
RST(reset flag,TCP header)	RST(复位标志,TCP 分节之一)
RTO(retransmission time out)	RTO(重传超时)
RTP(Real-time Transport Protocol)	RTP(实时传输协议)
RTT(round-trip time)	往返时间
sanity check	理智检查
SAP(Session Announcement Protocol)	SAP(会话声明协议)
scatter read	分散读
scheduling latency	调度延迟

SDP(Session Description Protocol)	SDP(会话描述协议)
segment,TCP	分节
sequence number,UDP	序列号
server	服务器〔特指运行中的服务器程序,动态〕 服务器程序〔特指程序代码本身,静态〕 服务器主机〔特指服务器进程所在主机,罕见〕
service	服务
session announcement,MBone	会话声明
session leader	会话头进程
set-user-ID	SUID〔本概念难以简洁译出,按字面译往往更难理解。既然文件属性中有 SUID 位,译成 SUID 更好些。其含义是可执行文件执行时以文件属主即 user 的 ID 作为相应进程的有效用户 ID,而不是用执行者自己的 ID〕
shallow copy	浅拷贝
shutdown of server host	服务器主机关机
signal action	信号行为
signal blocking	信号阻塞
signal catching	信号俘获,信号捕获
signal delivery	信号递交
signal disposition	信号处置
signal generation	信号生成
signal handler	信号处理程序
signal mask	信号掩码
signal queueing	信号排队
signal-driven I/O,model	信号驱动 I/O
simple name,DNS	简名〔相对于 FQDN 即全限定域名〕
simultaneous [close, open]	同时[关闭,打开]
simultaneous connections	同时的连接
site	网点〔范围可大可小〕 站点〔单台 FTP 或 WWW 服务器主机〕
site-local address	网点局部地址
site-local multicast scope	网点局部多播范围,局部于网点的多播范围
SLA(site-level aggregation identifier)	SLA(网点级聚集标识)
SLIP(Serial Line Internet Protocol)	SLIP(串行线网际协议)
slow start	慢启动
SMTP(Simple Mail Transfer Protocol)	SMTP(简单的邮件传送协议)
SNA(System Network Architecture)	SNA(系统网络体系结构)

SNMP (Simple Network Management Protocol)	SNMP(简单的网络管理协议)
socket	套接口
socket address structure	套接口地址结构
socket descriptor (=sockfd)	套接口描述字(等于套接字)
socket options	套接口选项
socket owner	套接口属主
socket pair	套接口对
socket receive buffer, UDP	套接口接收缓冲区
sockets	套接口(编程),套接口 API〔专有名词〕
sockets and standard I/O	套接口与标准 I/O
sockets and XTI interoperability	套接口与 XTI 的互操作性
soft error	软错误
software interrupt	软件中断
source address	源地址
source code [portability]	源代码[可移植性]
source quench, ICMP	源熄灭
source routing	源路由
SPT(server processing time)	SPT(服务器处理时间)
SPX(Sequenced Packet Exchange)	SPX(有序的分组交换)
SSRR(strict source and record route)	SSRR(严格的源与记录路径)
standard Internet services	标准因特网服务
standard I/O [stream]	标准 I/O[流]
state transition diagram	状态转换图
sticky options, IPv6	粘附的选项
stream pipe	流管道
stream socket	字节流套接口
streams driver	流驱动程序
streams head	流头
streams module	流模块
streams queue	流队列
strong end system model	强端系统模型
subnet [address, ID, mask]	子网[地址,ID,掩码]
superuser	超级用户
SVR3(System V Release 3)	SVR3(System V 版本 3)
SVR3(System V Release 4)	SVR4(System V 版本 4)
SYN (synchronize sequence numbers flag, TCP header)	SYN(同步序列号标志,TCP 分节之一)
synchronous, I/O	同步

system call	系统调用
system time	系统时间
TCP(Transmission Control Protocol)	TCP(传输控制协议)
TCP/IP big picture	TCP/IP 总图
Telnet(remote rerminal protocol)	Telnet(远程终端协议)
termination of server process	服务器进程终止
test networks and hosts	测试(用)网络和主机
test programs	测试(用)程序
TFTP(Trivial File Transfer Protocol)	TFTP(简化文件传送协议)
thread attributes	线程属性
thread ID	线程 ID
thread-safe	线程安全
thread-specific data	线程特定数据
three-way handshake,TCP	三路握手
thundering herd	惊群
time exceeded,ICMP	超时
timeout	超时
timestamp option,TCP	时间戳选项
timestamp request,ICMP	时间戳请求
TLA(top-level aggregation identifier)	TLA(顶级聚集标识)
TLI(Transport Layer Interface)	TLI(传输层接口)
TLV(type/length/value)	TLV(类型/长度/值)
token ring	令牌环
TOS(type of service)	TOS(服务类型)
total length field,IPv4	总长度域,总长度字段
TPI(Transport Provider Interface)	TPI(传输提供者接口)
transaction time	事务处理时间
transient multicast group	临时多播组
transport address,XTI	传输地址
transport endpoint,TLI	传输端点
transport provider,TLI	传输提供者
TSDU(transport service data unit)	TSDU(传输服务数据单元)
T/TCP(TCP for Transaction)	事务 TCP
TTL(time-to-live)	TTL(存活时间)
tunnel	隧道
tunnel source	隧道源端
tunnel endpoint	隧道末端
two-phase commit protocol	两阶段提交协议
type field,ICMP	类型域,类型字段

type filtering, ICMPv6	类型过滤
typo	排印或打字错误
typographical conventions	排版约定
UDP(User Datagram Protocol)	UDP(用户数据报协议)
UDP socket	UDP 套接口
unbuffered standard I/O stream	不缓冲的标准 I/O 流
unconnected UDP socket	未(经)连接的 UDP 套接口
unicast [address]	单播[地址]
universal address, XTI	通用地址
Unix domain socket	Unix 域套接口
Unix domain socket address structure	Unix 域套接口地址结构
Unix standard services	Unix 标准服务
Unix standards	Unix 标准
Unix versions and portability	Unix 版本与可移植性
unspecified address	未指定地址
URG (urgent pointer flag , TCP header)	URG(紧急指针标志, TCP 分节之一)
urgent mode	紧急模式
urgent [offset, pointer]	紧急[(位置)偏移, 指针]
URI(uniform resource identifier)	URI(统一资源标识符)
user ID	用户 ID
UTC(Coordinated Universal Time)	UTC(国际标准时间)〔意译〕
UUCP(Unix-to-Unix Copy)	UUCP(Unix 到 Unix 拷贝)
value-result argument	值-结果参数
variable	变量
version number field, IP	版本号域, 版本号字段
virtual network	虚拟网络
waffle	胡说八道, 胡言乱语
WAN(wide area network)	WAN(广域网)
wandering duplicate	漫游的重复分组
weak end system model	弱端系统模型
well-known address	众所周知的地址
well-known multicast group	众所周知的多播组
well-known port	众所周知的端口
wildcard address	通配地址
window scale option, TCP	窗口规模选项
wrapper function	包裹函数
WWW(World Wide Web)	WWW(万维网)
XDR(external data representation)	XDR(外部数据表示)

XNS(Xerox Network System)	XNS(Xerox 网络系统)
XNS(X/Open Networking Services)	XNS(X/Open 网络服务)
XPG(X/Open Portablity Guide)	XPG(X/Open 可移植性指南)
XTI(X/Open Transport Interface)	XTI(X/Open 传输接口)
zombie	僵尸