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。

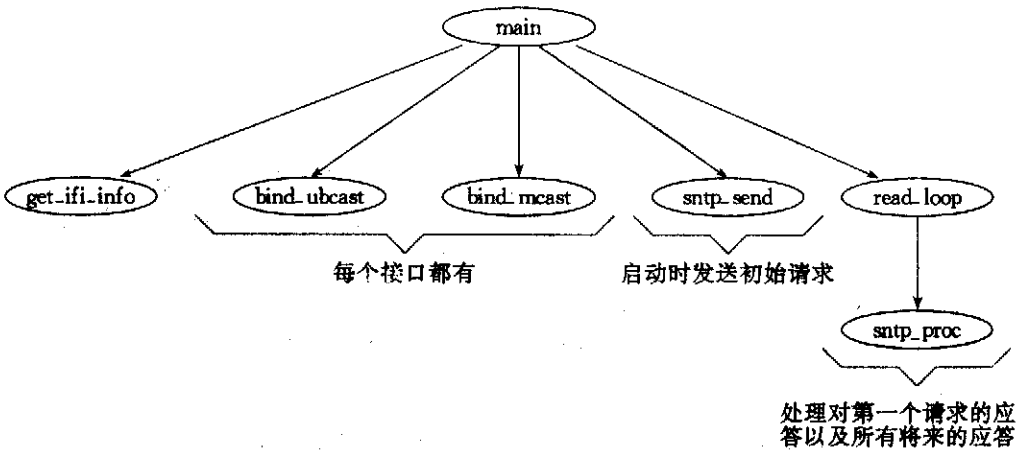


图19.20 SNTP客户程序各函数概览

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。

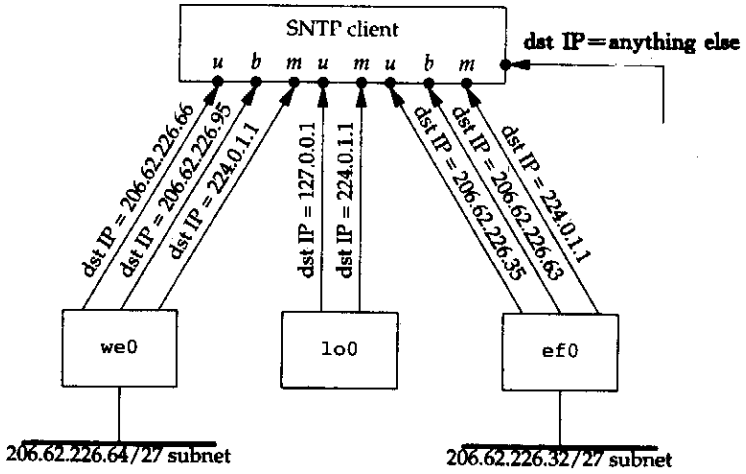


图 19.23 在主机 bsdi 上由 SNTP 客户创建的 9 个套接口

```
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。

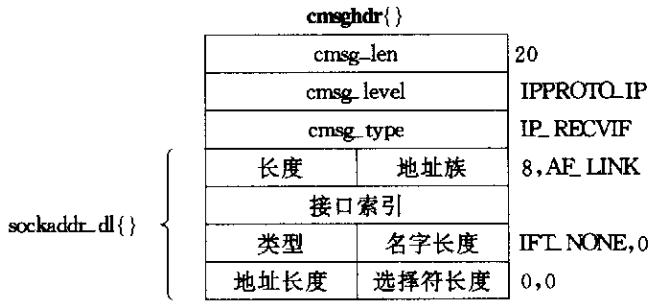


图 20.3 IP_RECVIF 返回的辅助数据对象

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。

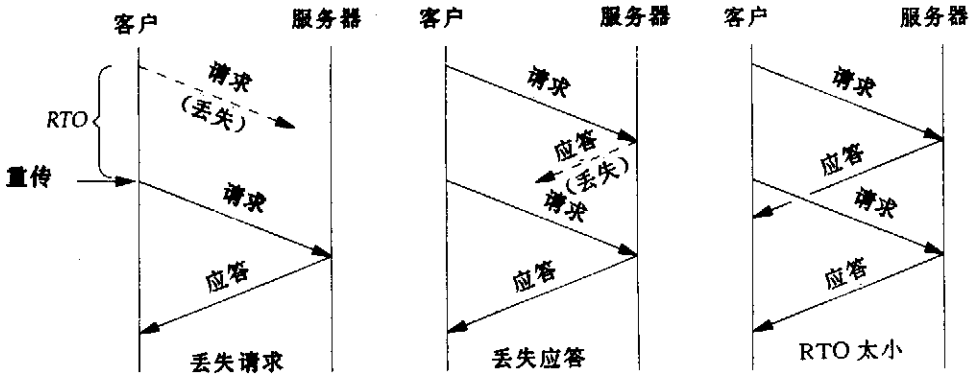


图 20.5　当重传定时器超时的三种情形

- 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。

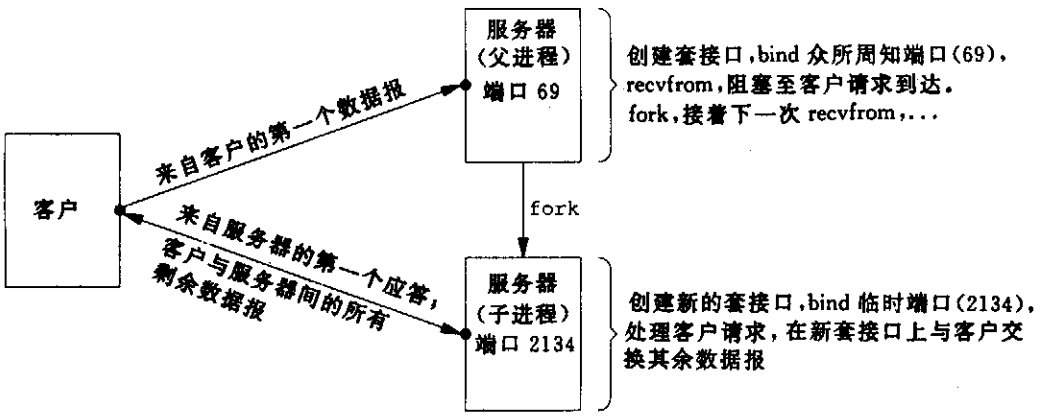


图 20.19　在独立运行的并发 UDP 服务器中所需的处理

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。

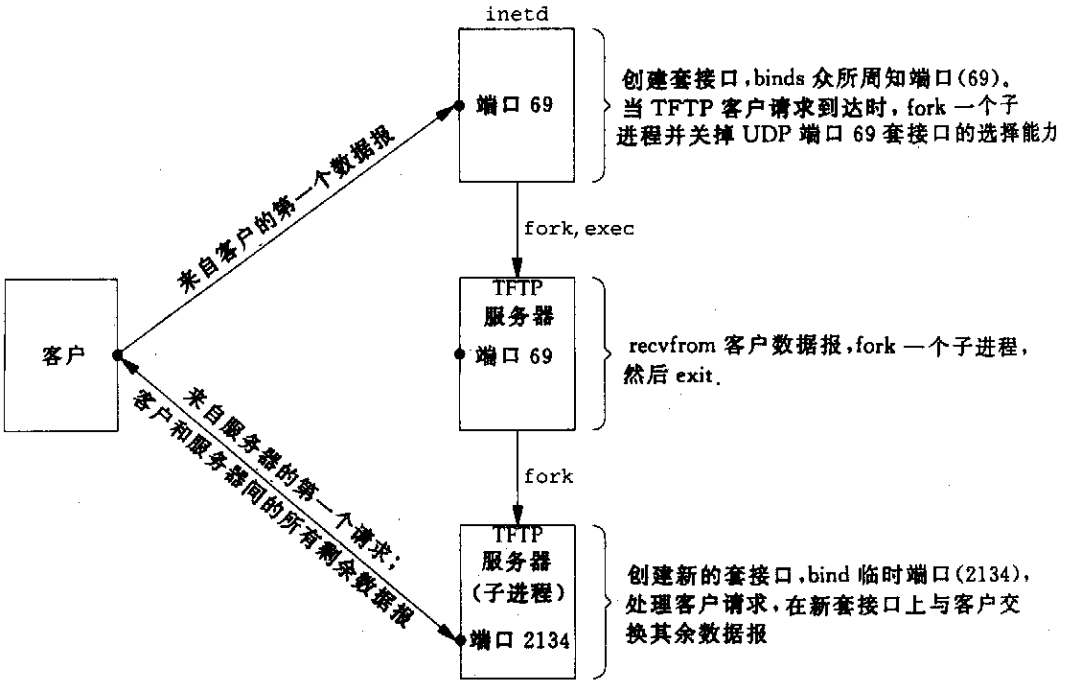


图 20.20 由 inetd 启动的 UDP 并发服务器

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 */
};
```

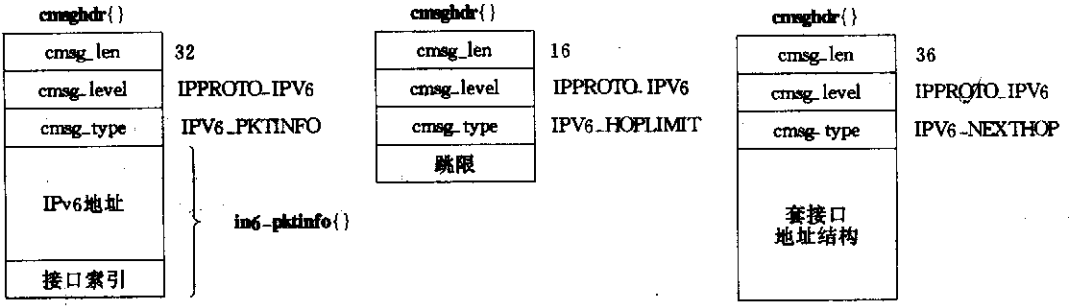


图 20.21　IPv6 分组信息的辅助数据

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。

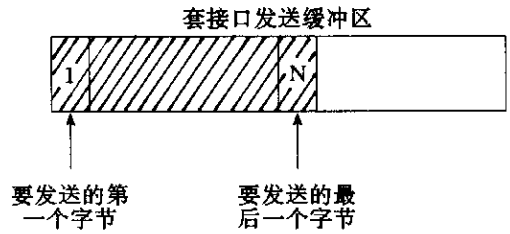


图21.1 包含要发送数据的套接口发送缓冲区

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"。

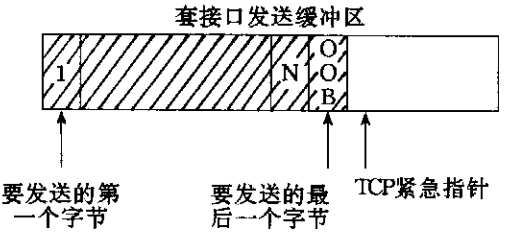


图 21.2　应用进程写入 1 字节带外数据后的套接口发送缓冲区

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。

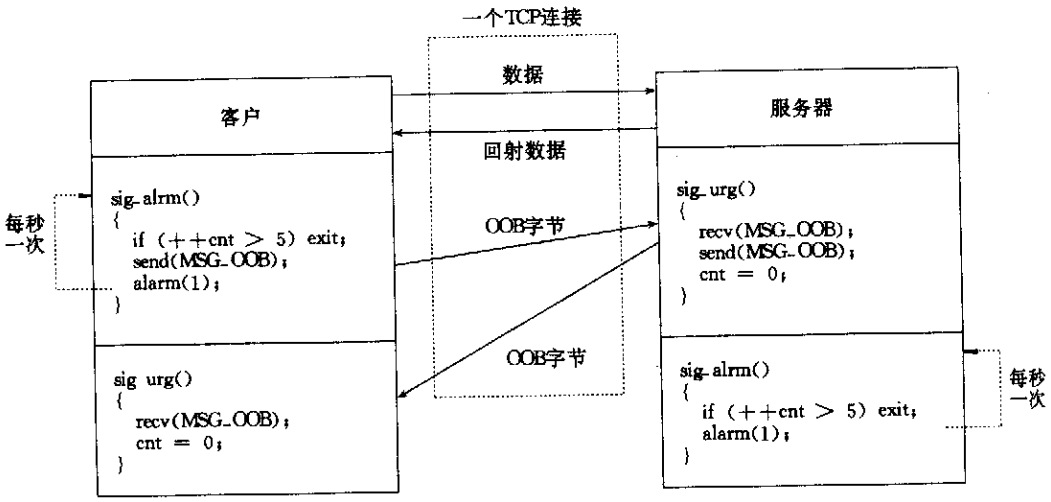


图 21.13 使用带外数据的客户-服务器心博函数

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。

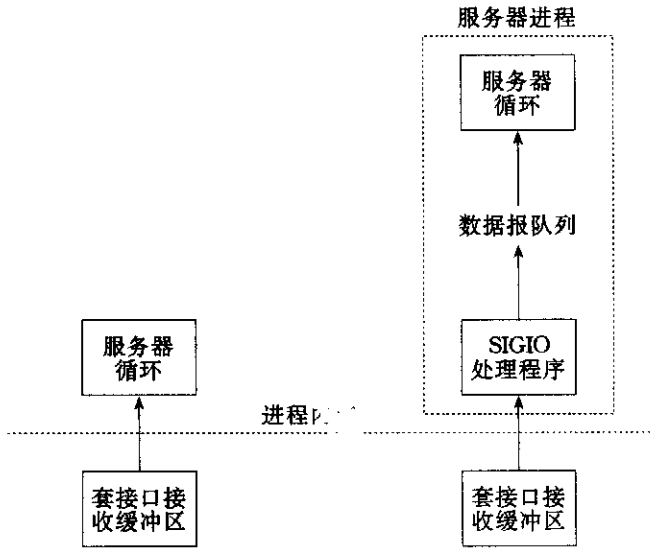


图 22.1　构建一个 UDP 服务器的两种方法

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]

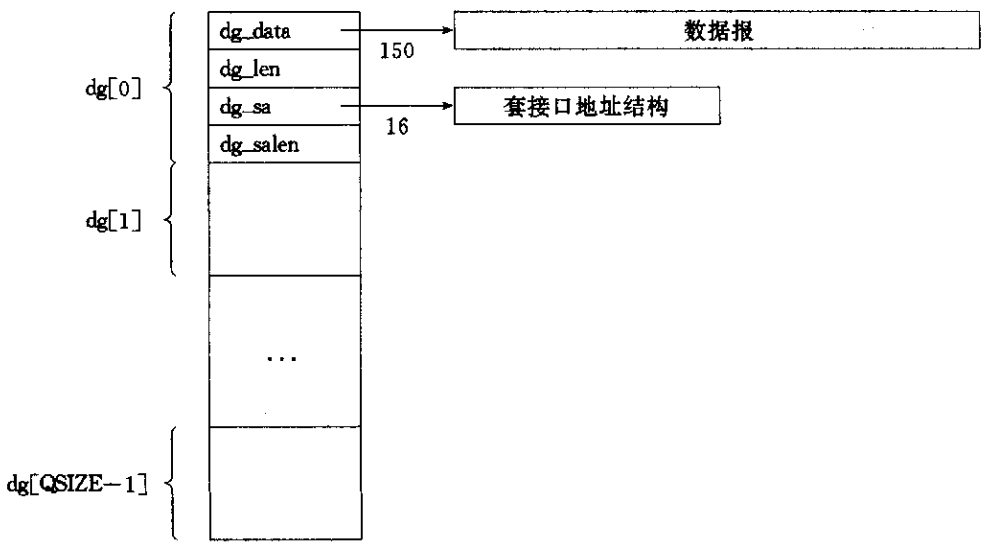


图22.3 用于存放接收到的数据报及其套接口地址结构的数据结构

```
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。

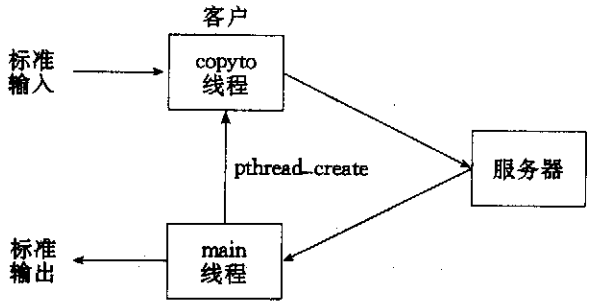


图 23.1　用线程重新编写 str_cli

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。

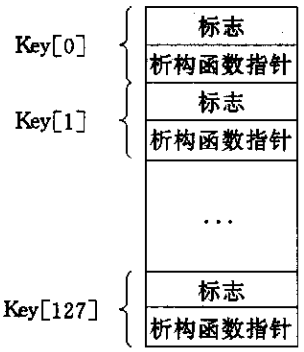


图 23.7 线程特定数据的可能实现

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。

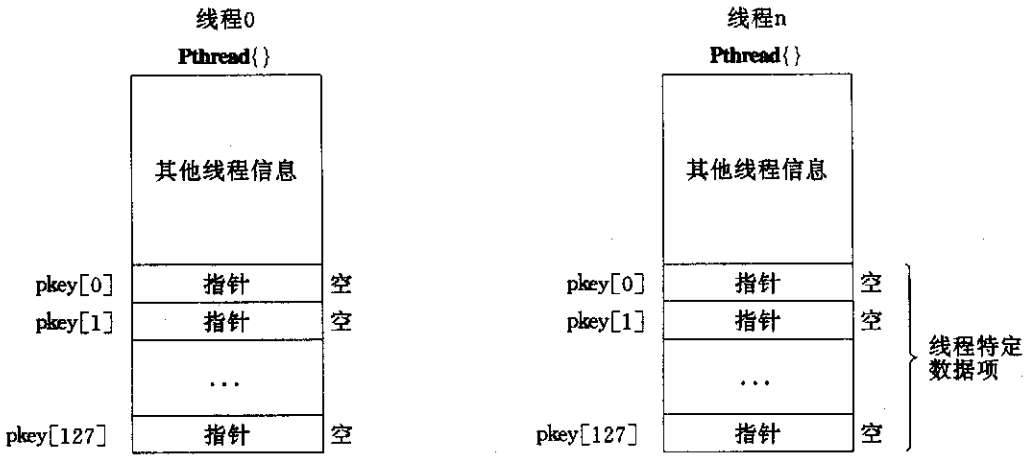


图 23.8　系统为每个线程维护的信息

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。

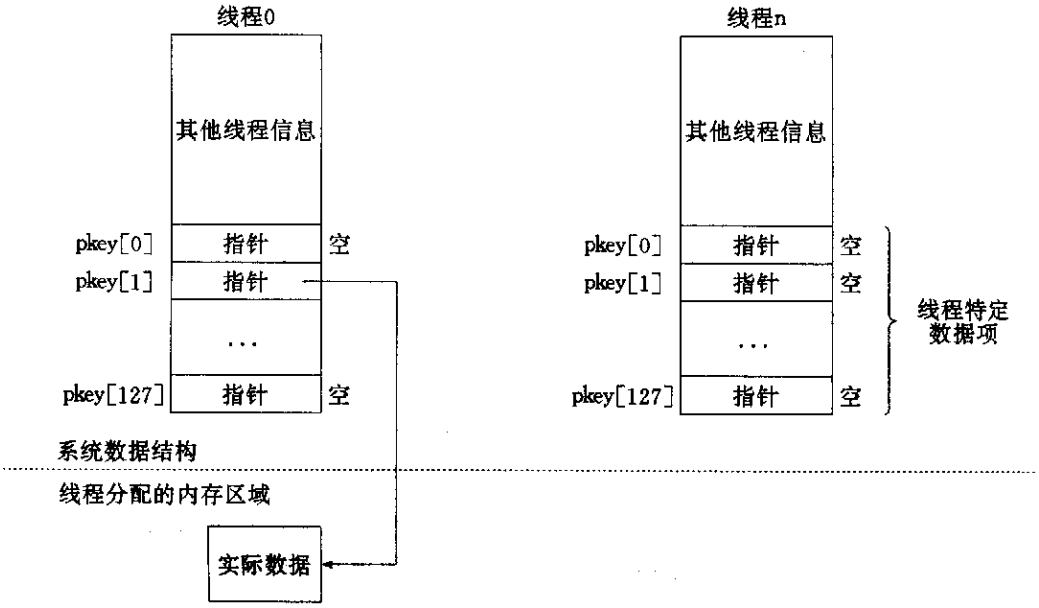


图 23.9 将分配的存储区域和线程特定数据指针关联

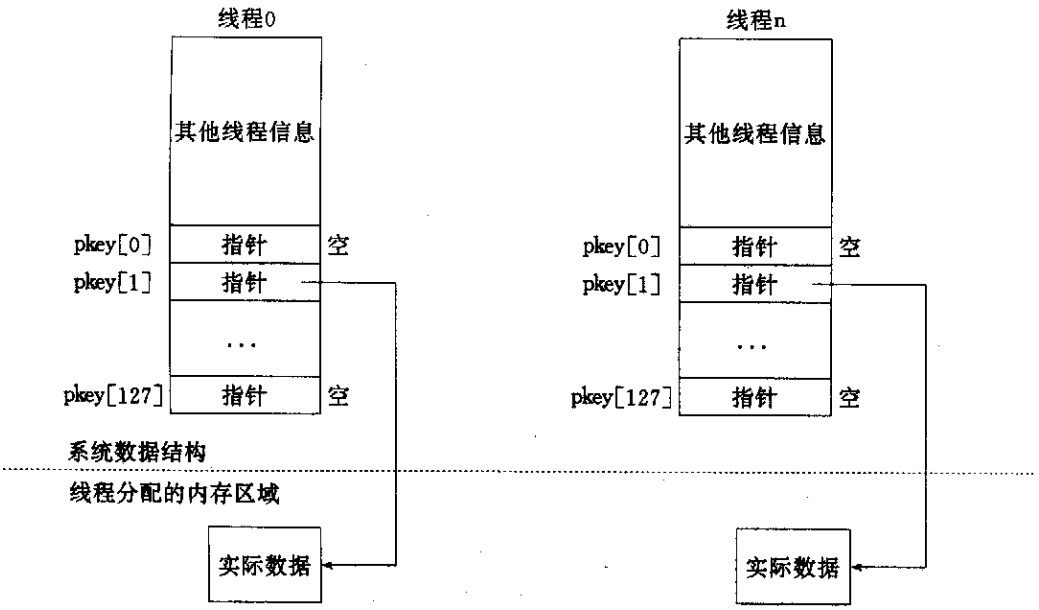


图 23.10 线程 n 初始化其线程特定数据后的数据结构

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。

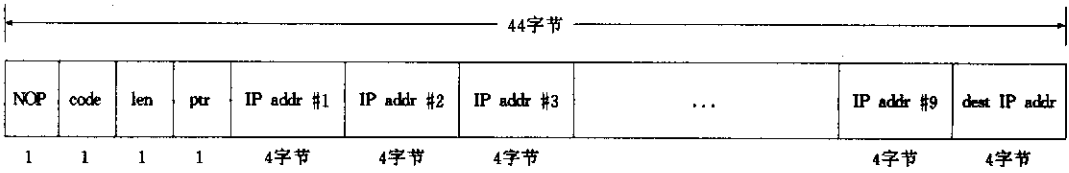


图 24.1 传递源路径给内核

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。

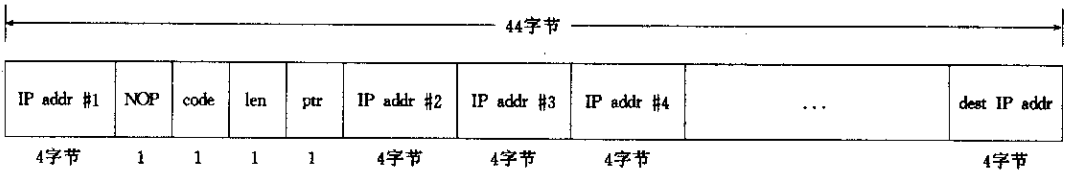


图 24.4　getsockopt 返回的源路径选项格式

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。

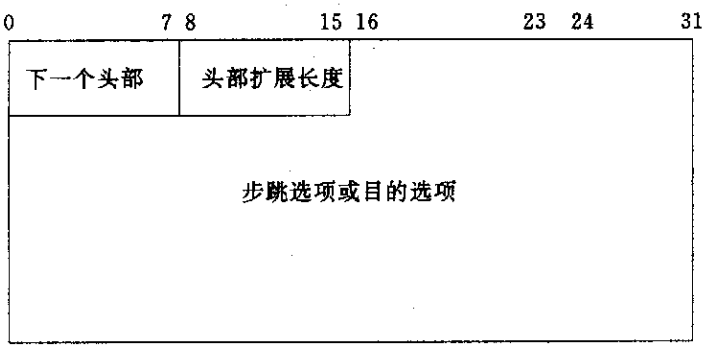


图24.7 步跳选项和目的选项的格式

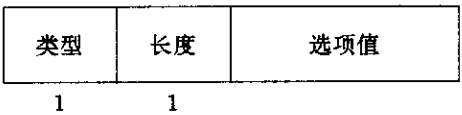


图24.8 步跳选项和目的选项的单个选项的格式

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。

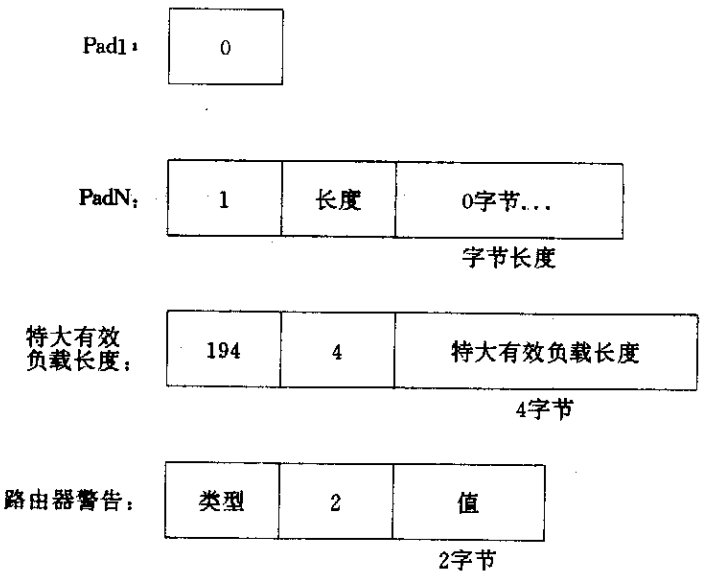


图 24.9　IPv6 步跳选项

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。

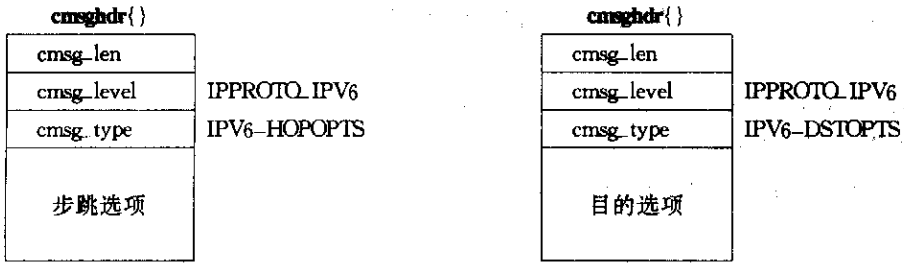


图 24.10 步跳选项和目的选项的辅助数据对象

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。

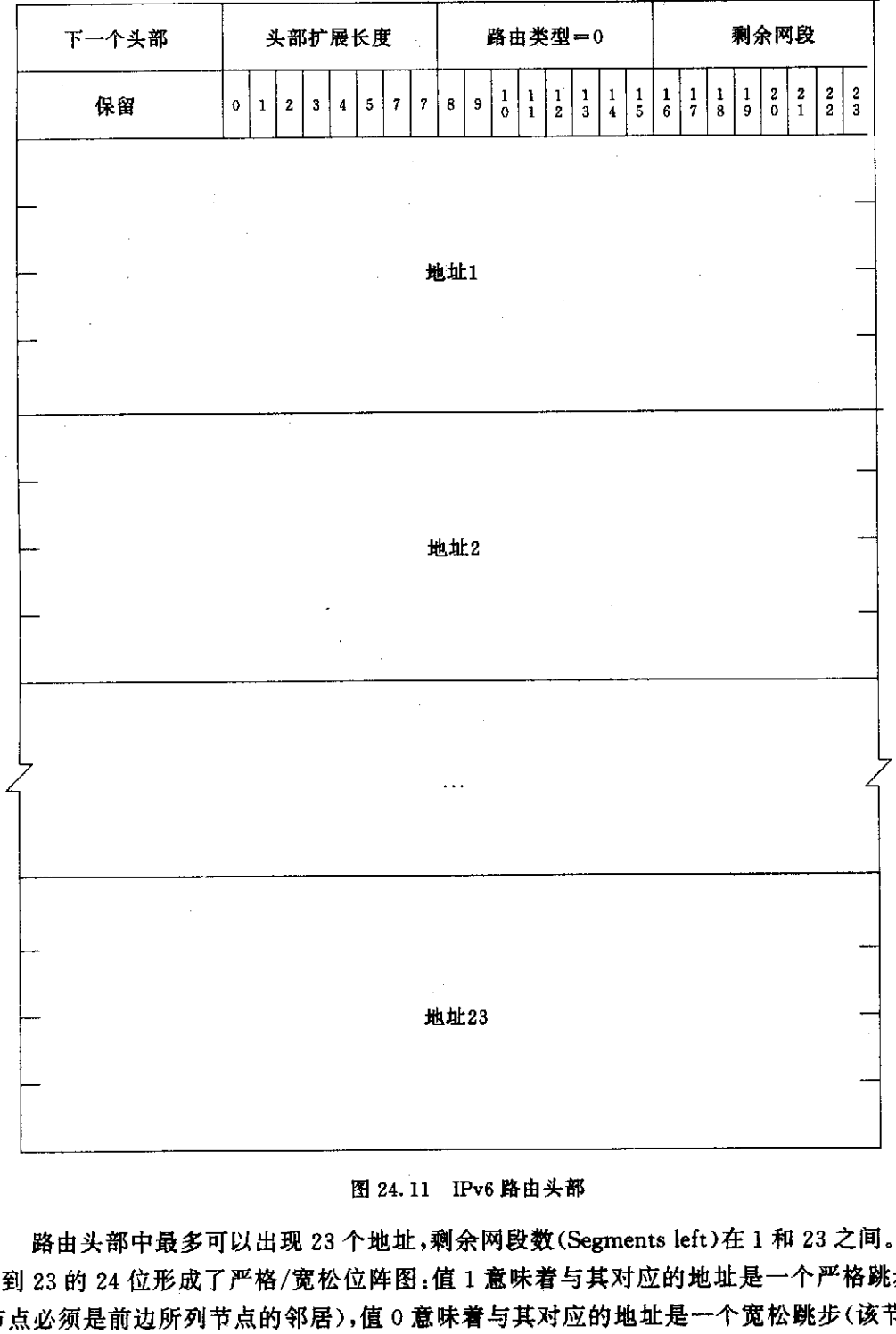


图 24.11 IPv6 路由头部

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。

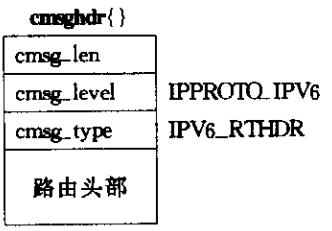


图 24.12　IPv6 路由头部的辅助数据对象

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：

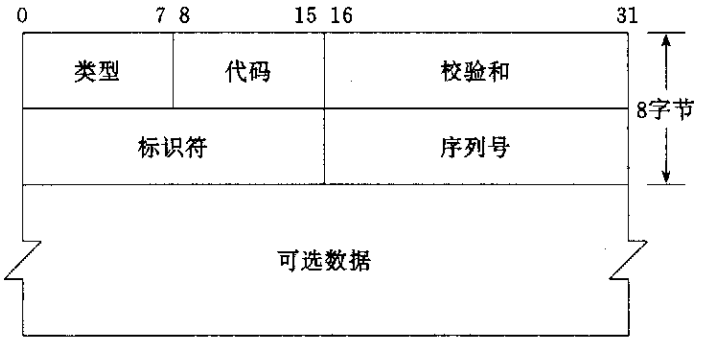


图 25.1 ICMPv4 及 ICMPv6 回射请求和回射应答消息的格式

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。

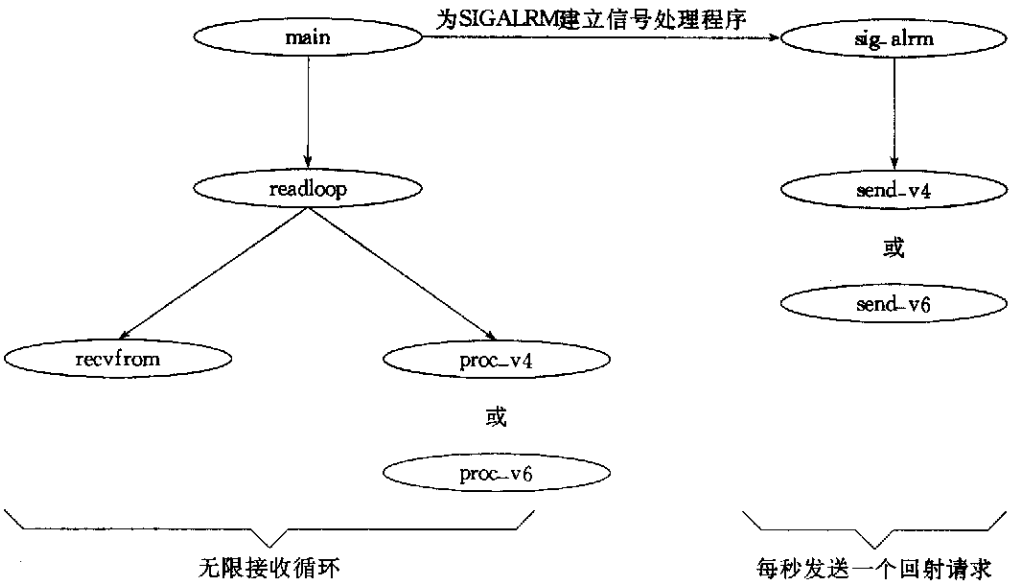


图 25.3 Ping 程序的函数概貌

```
 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]

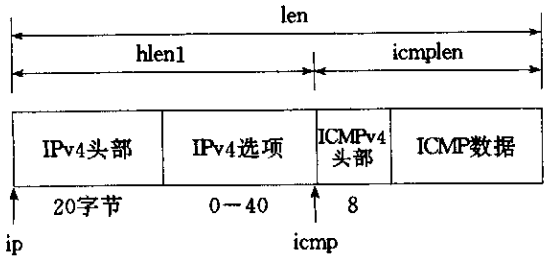


图 25.9 ICMPv4 应答结构：头部、指针及长度

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]

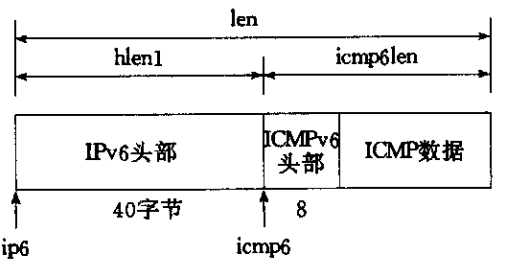


图 25.11　ICMPv6 应答结构：头部、指针及长度

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。

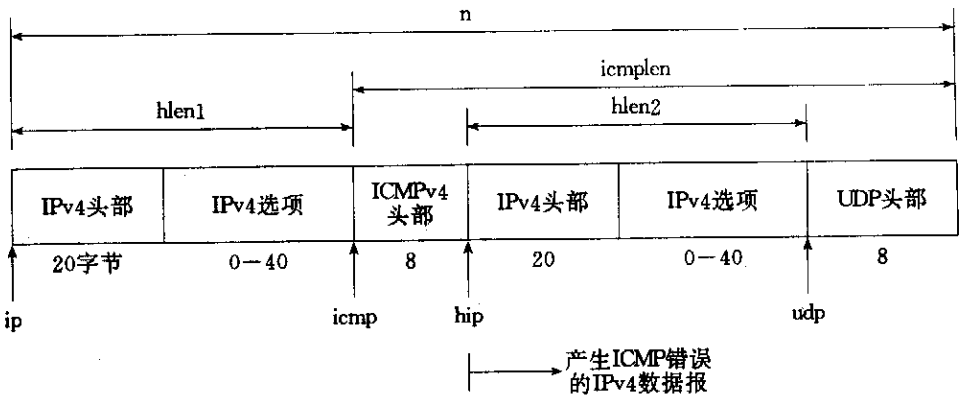


图 25.20　ICMPv4 出错消息的处理：头部、指针和长度

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。

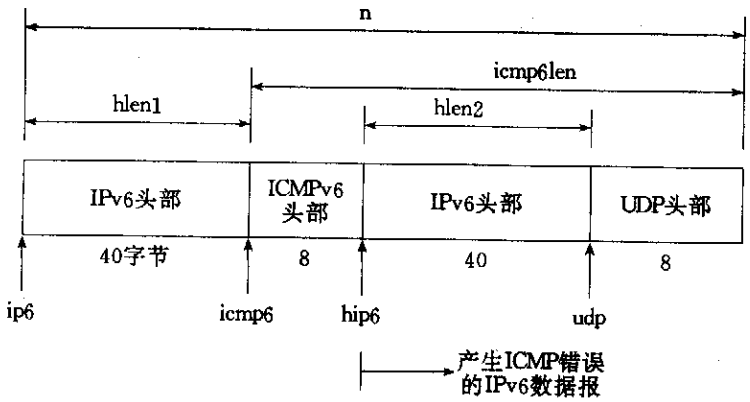


图25.21 ICMPv6出错消息的处理:头部、指针和长度

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。

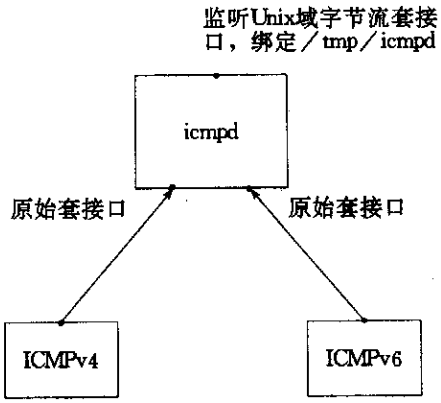


图 25.25 icmpd 守护进程，初始套接口建立

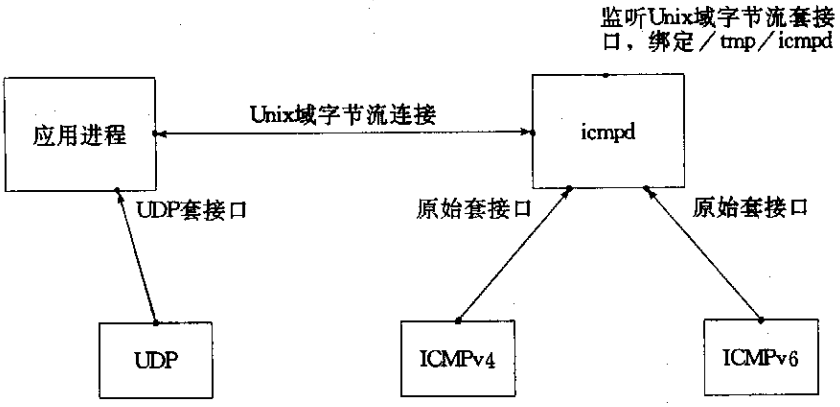


图 25.26 应用进程创建它的 UDP 套接口并与守护进程建立 Unix 域连接

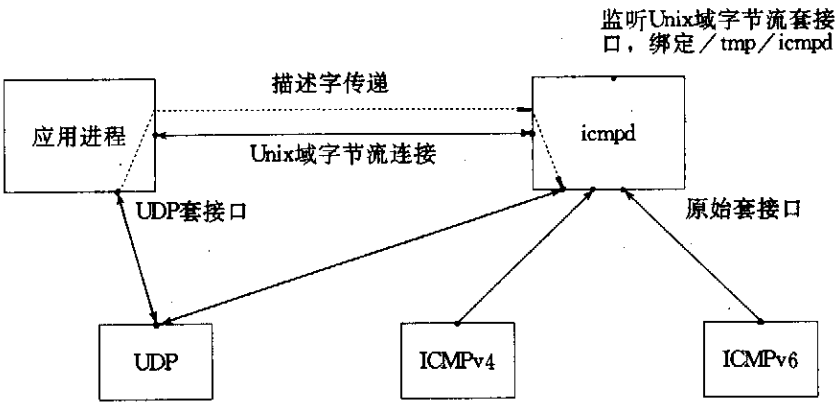


图 25.27 跨越 Unix 域连接，将 UDP 套接口描述字传递给 icmpd

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。

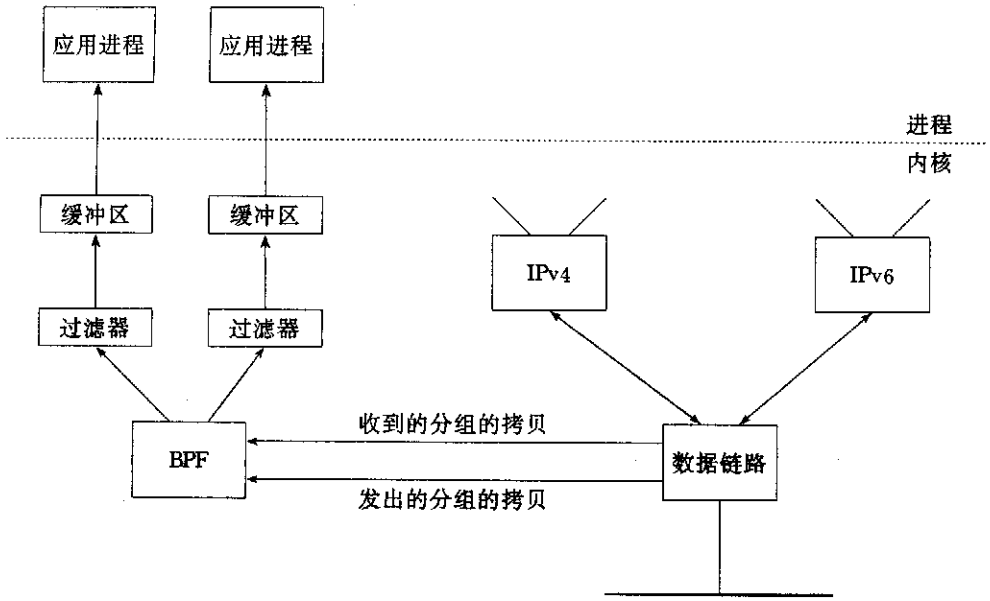


图 26.1 使用 BPF 截获分组

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。

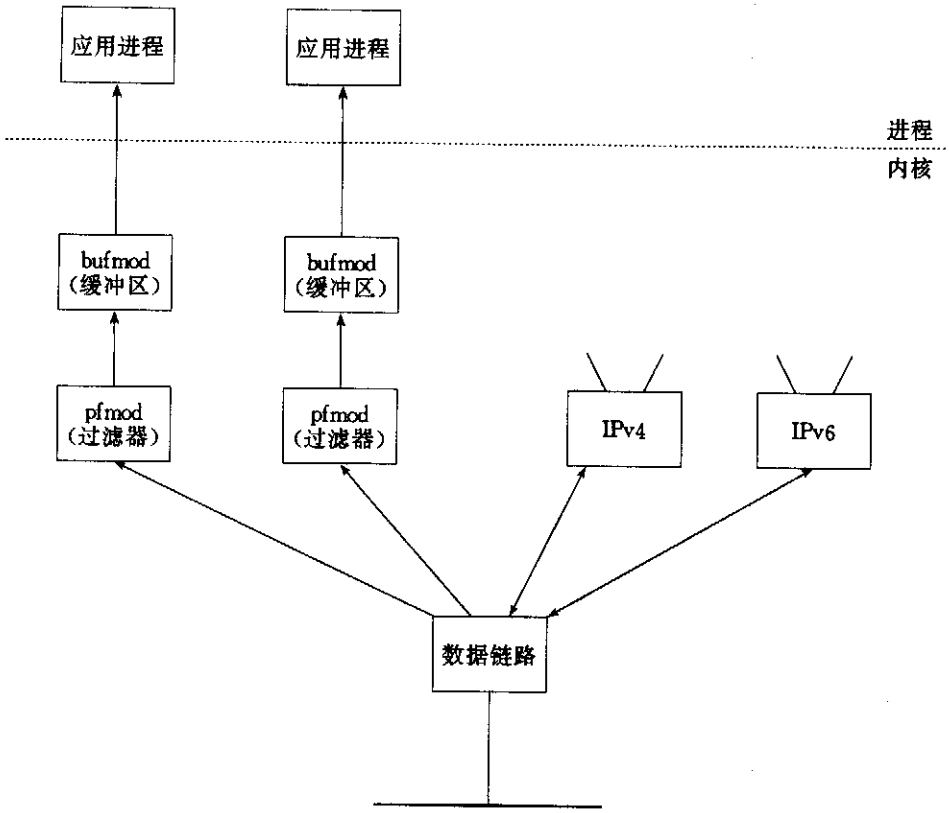


图 26.2 使用 DLPI、pfmod 和 bufmod 捕获分组

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。

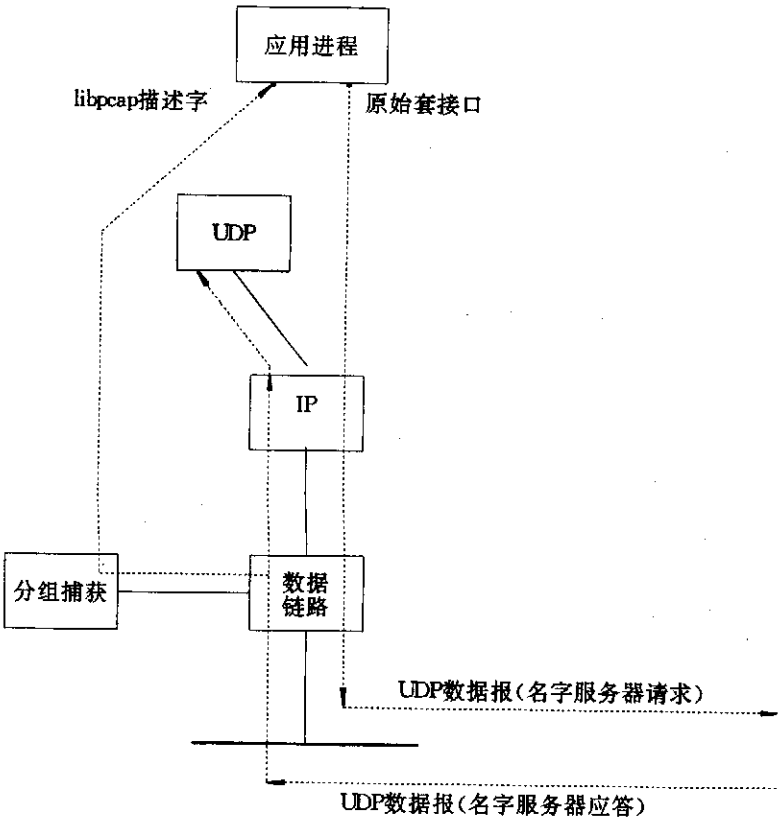


图 26.3 检查名字服务器是否计算 UDP 校验和的应用程序

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。

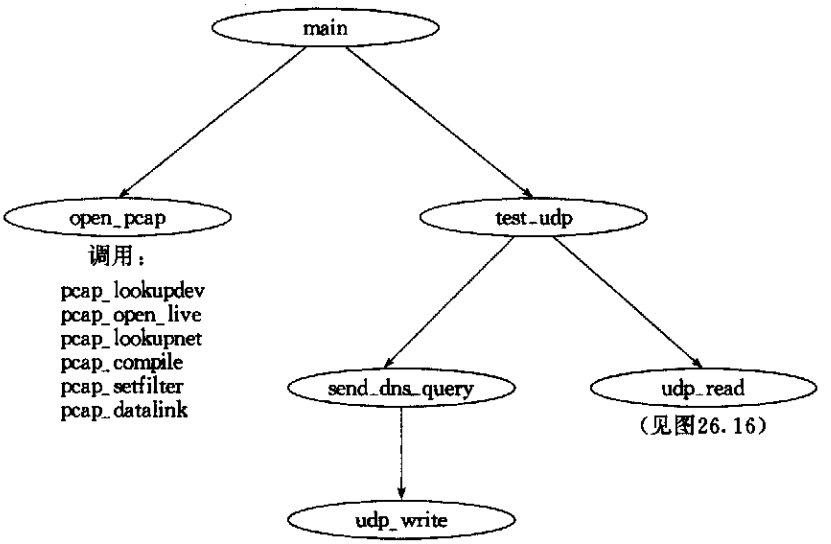


图 26.4　udpcksum 程序中的函数

```
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。

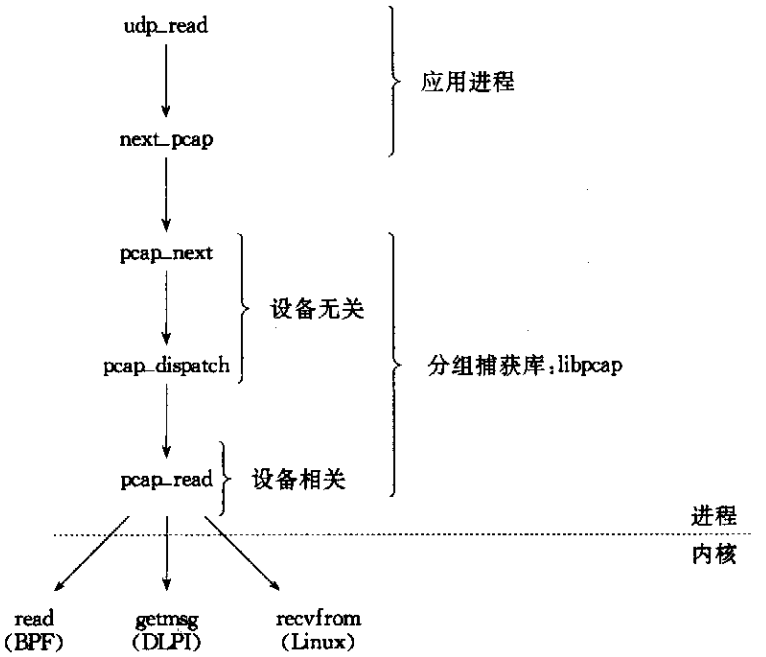


图 26.16　分组捕获库内与读取相关的函数调用

```
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。

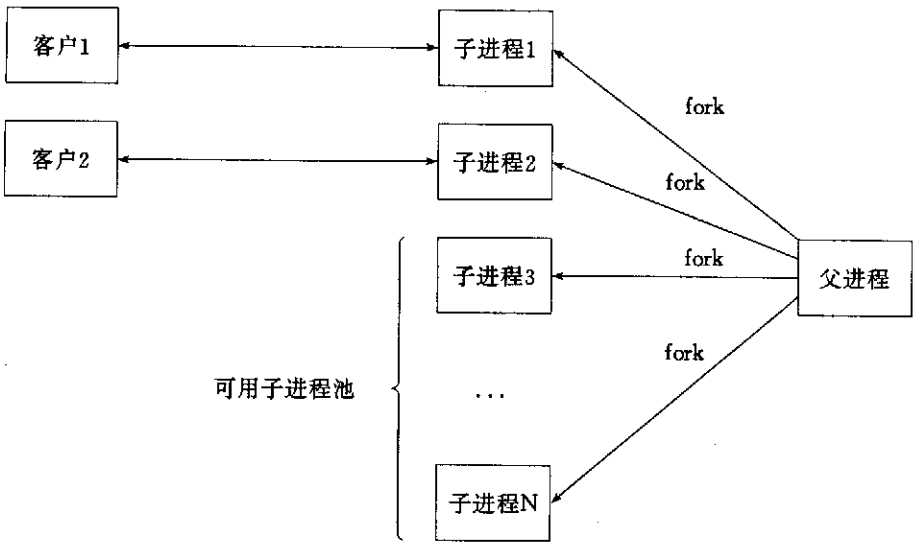


图 27.9 服务器预先派生子进程

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。

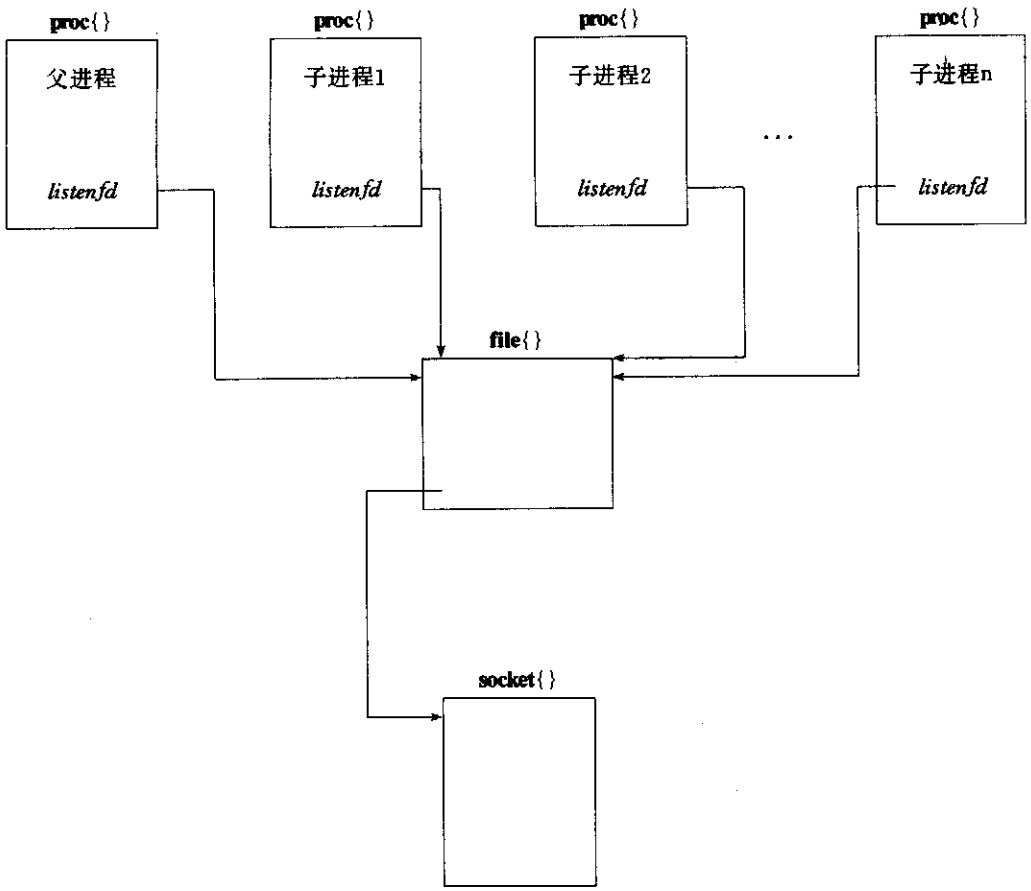


图 27.14 proc、file 和 socket 结构的关系

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]

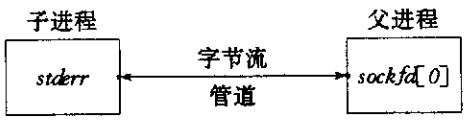


图27.23 父进程和子进程各关闭一端后的字节流管道

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。

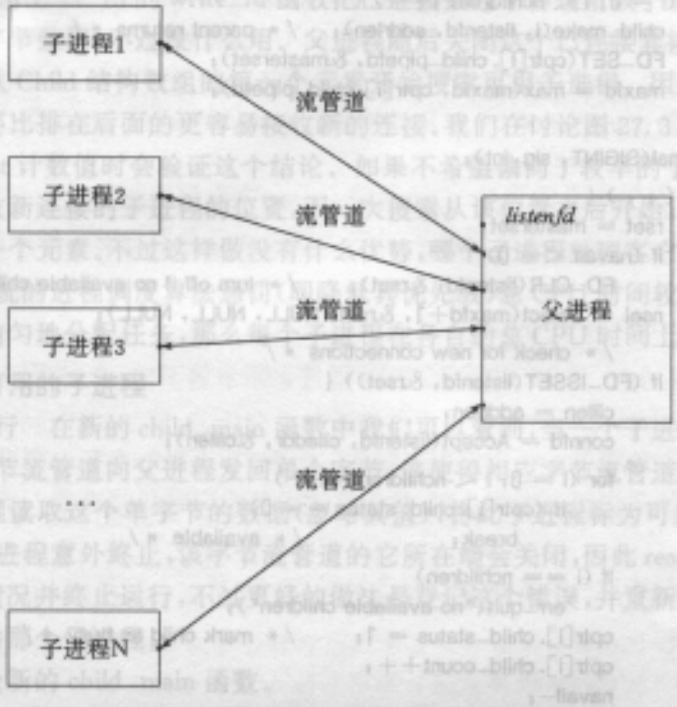


图 27.24　所有子进程都派生后的各个字节流管道

```
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。

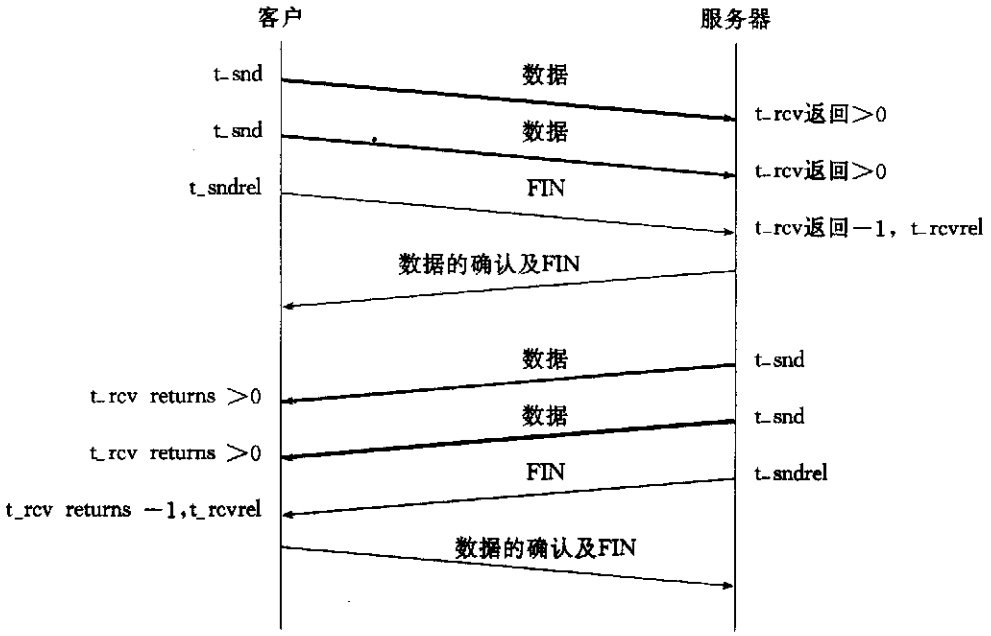


图 28.11 使用 XTI 的 TCP 半关闭

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。

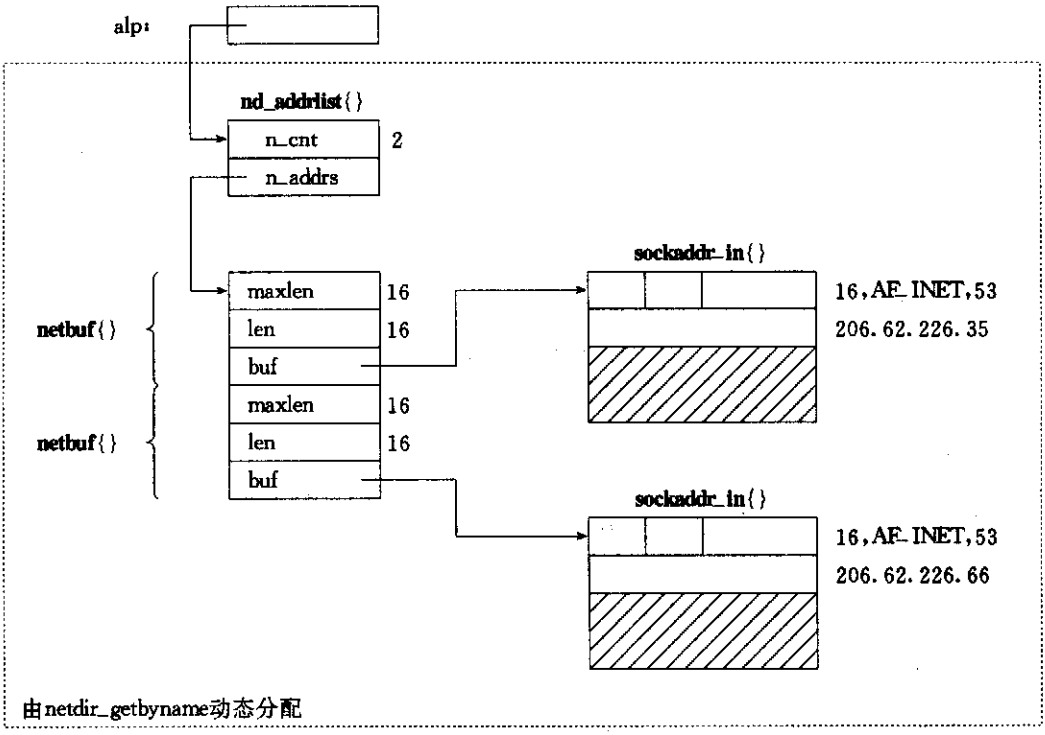


图 29.2 netdir_getbyname 返回的数据结构

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。

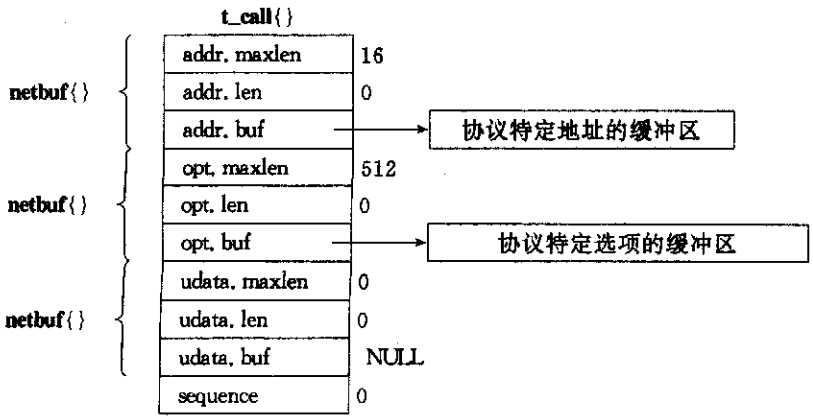


图 29.5 使用 t_alloc 分配结构和缓冲区

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。

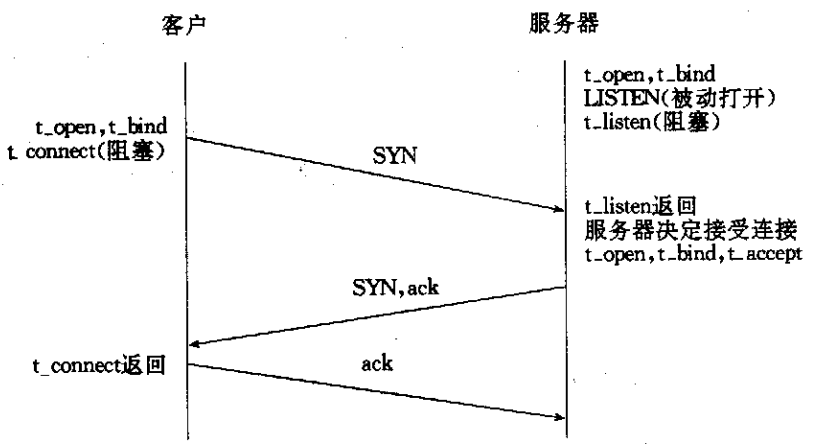


图 30.1　XTI 服务器接收连接请求的预期模型

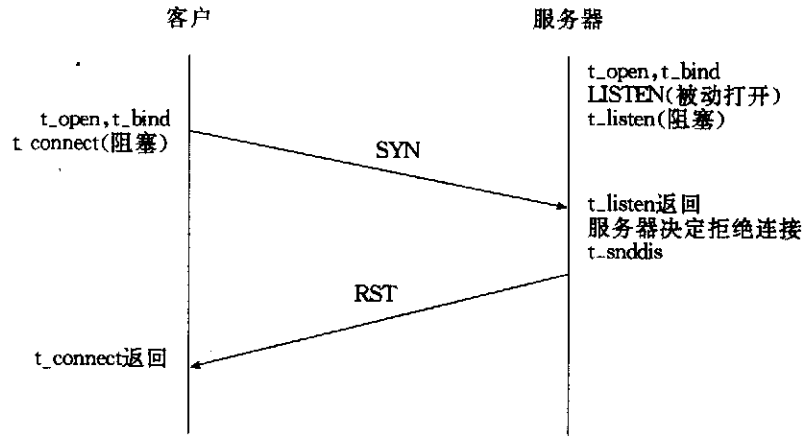


图30.2 XTI服务器拒绝连接请求的预期模型

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。

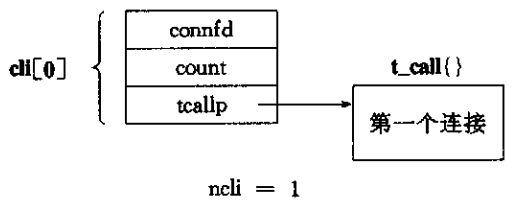


图 30.7　t_listen 返回头一个客户连接后的数据结构

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。

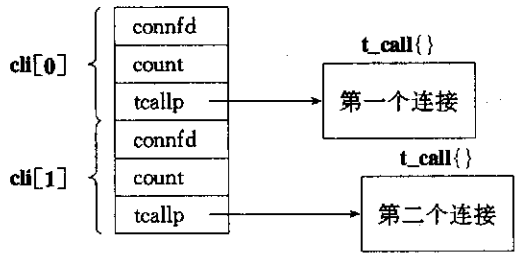


图30.8 t_listen返回第二个客户连接后的数据结构

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。

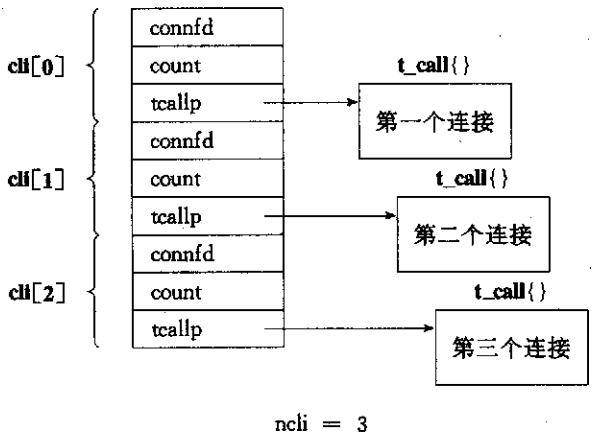


图30.9 t_listen返回第三个客户连接后的数据结构

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:

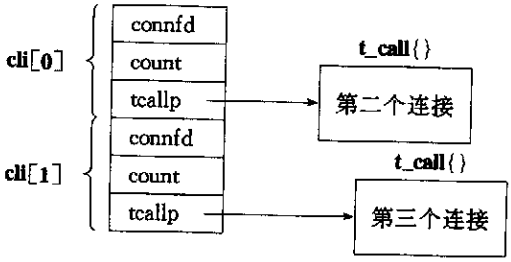


图 30.10 第一个连接夭折后的数据结构

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。

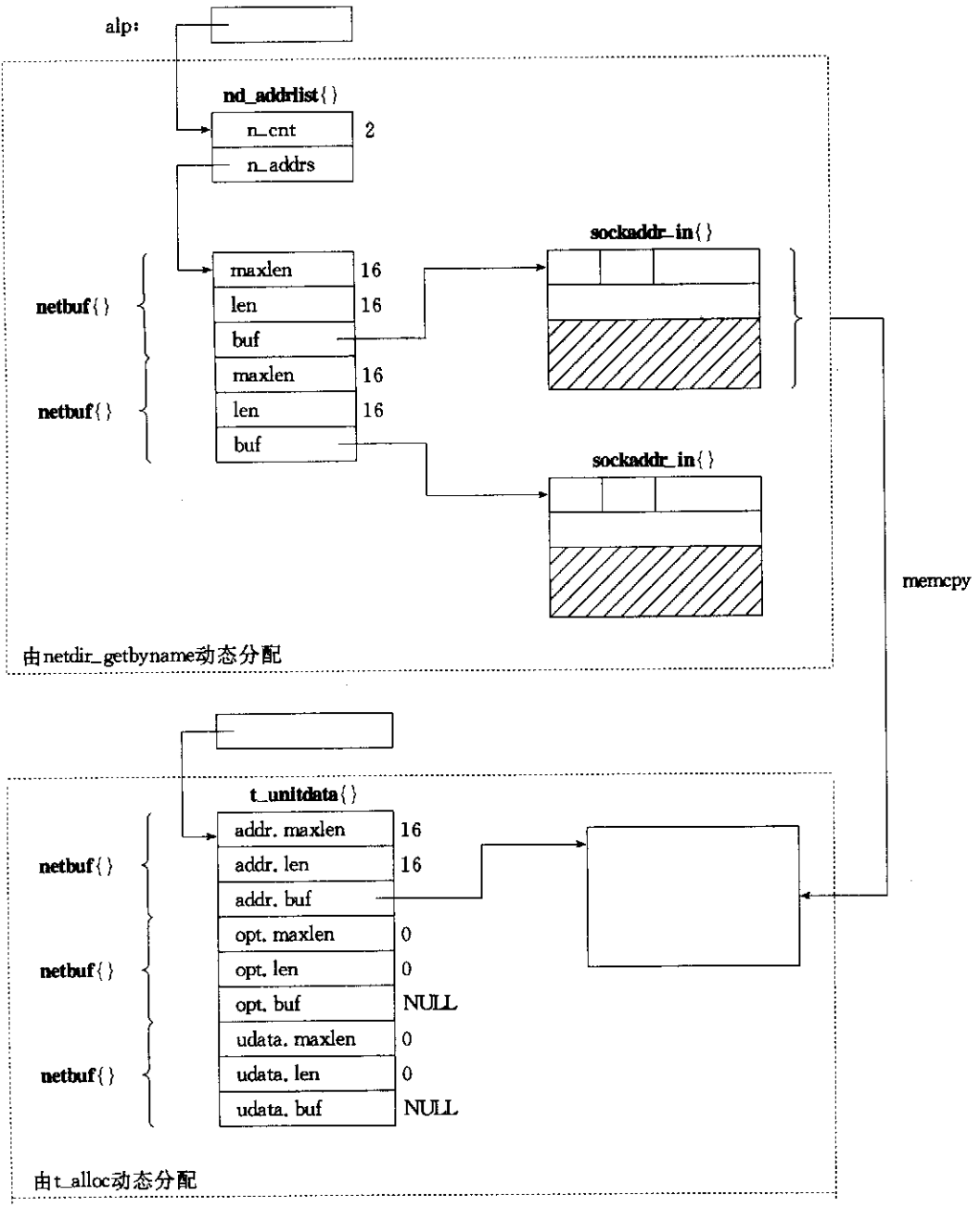


图 31.2 调用 udp_client 期间的数据结构

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。

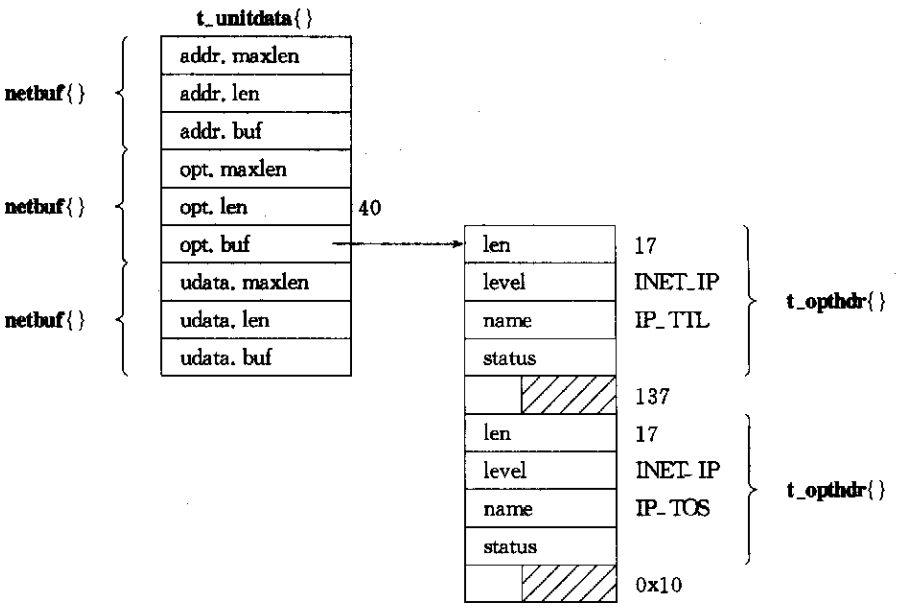


图 32.3　由一个 netbuf 结构指向的两个选项的例子

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。

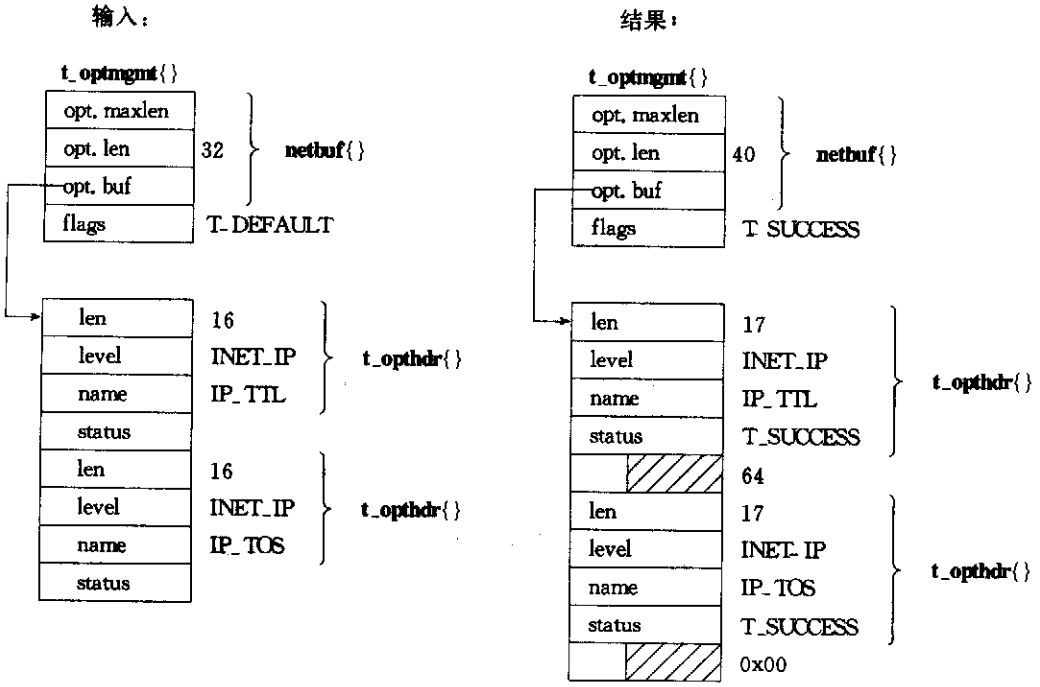


图32.4 从t_optmgmt请求两个选项的缺省值

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)。

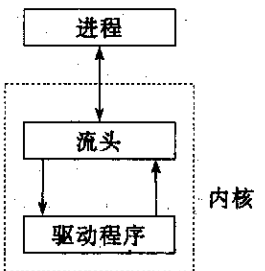


图 33.1　进程和驱动程序之间的一个流

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。

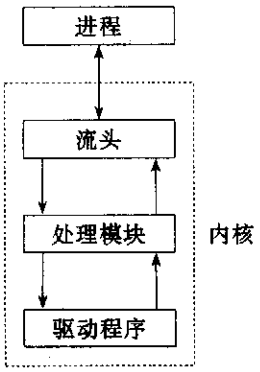


图 33.2 一个有处理模块的流

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。

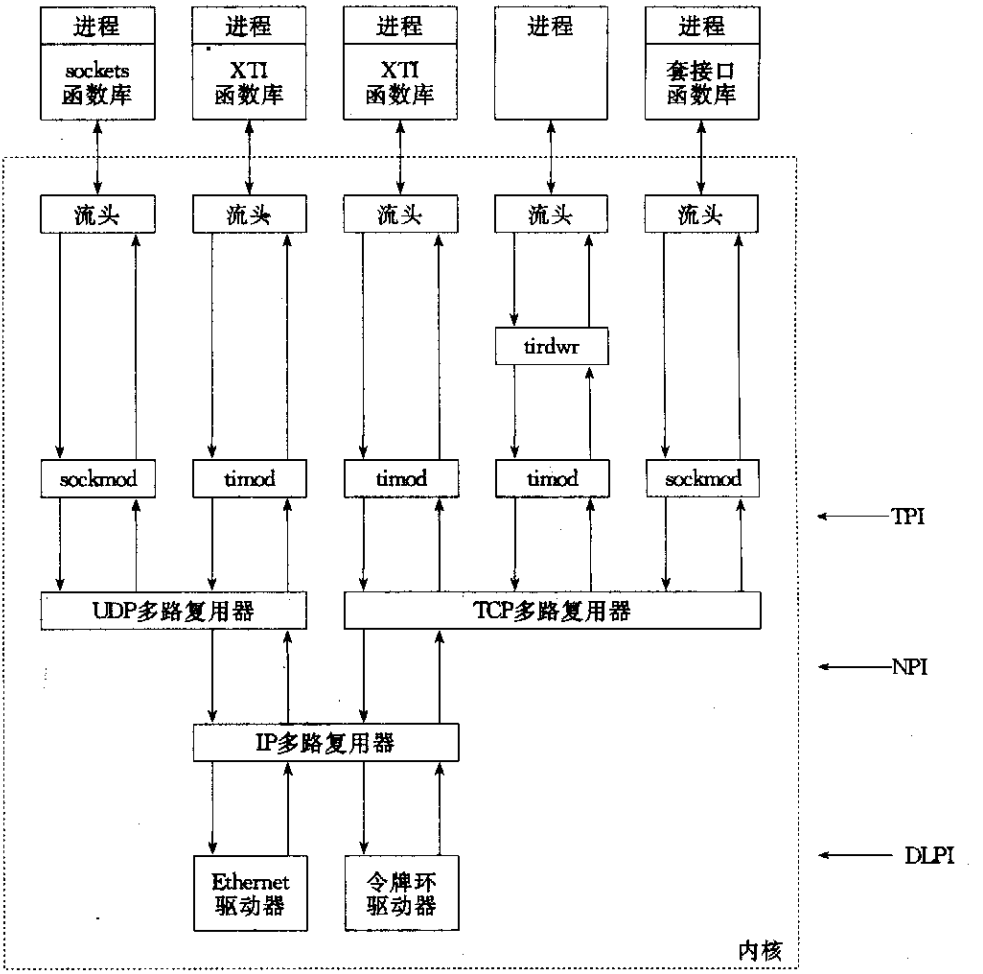


图 33.3　使用流的 TCP/IP 的简化实现

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。

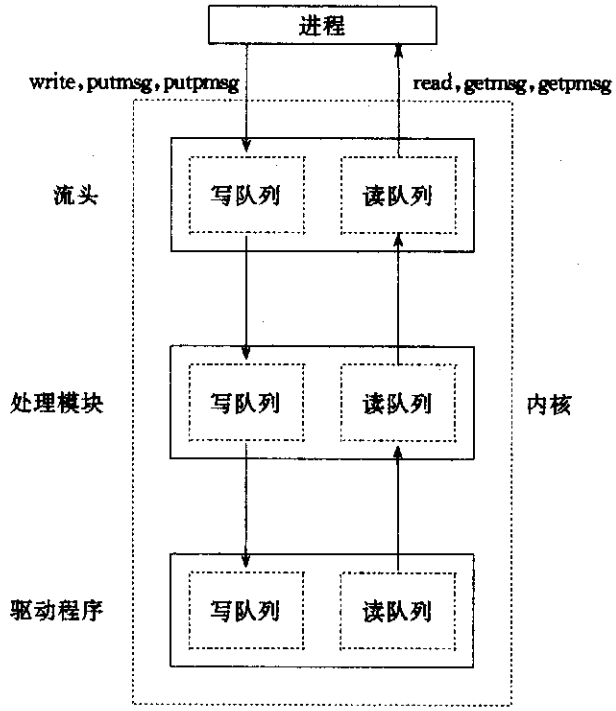


图 33.4 流中每个部件至少有一对队列

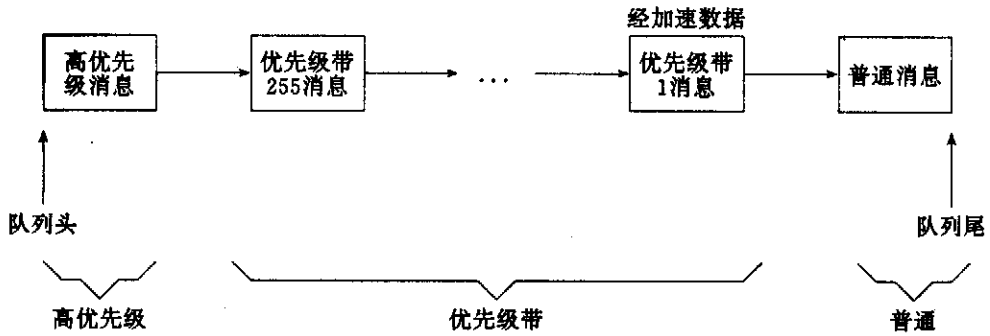


图 33.5 一个队列中的流消息基于优先级的顺序

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。

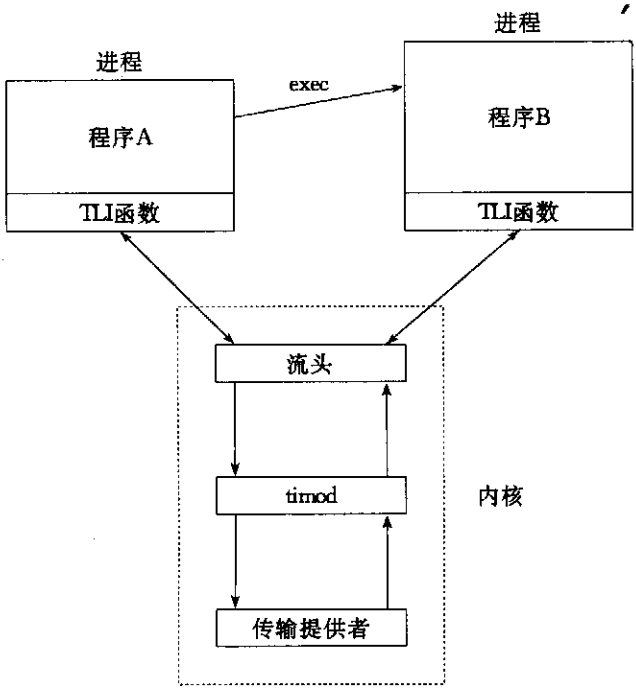


图34.2 进程调用exec时TLI的实现

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

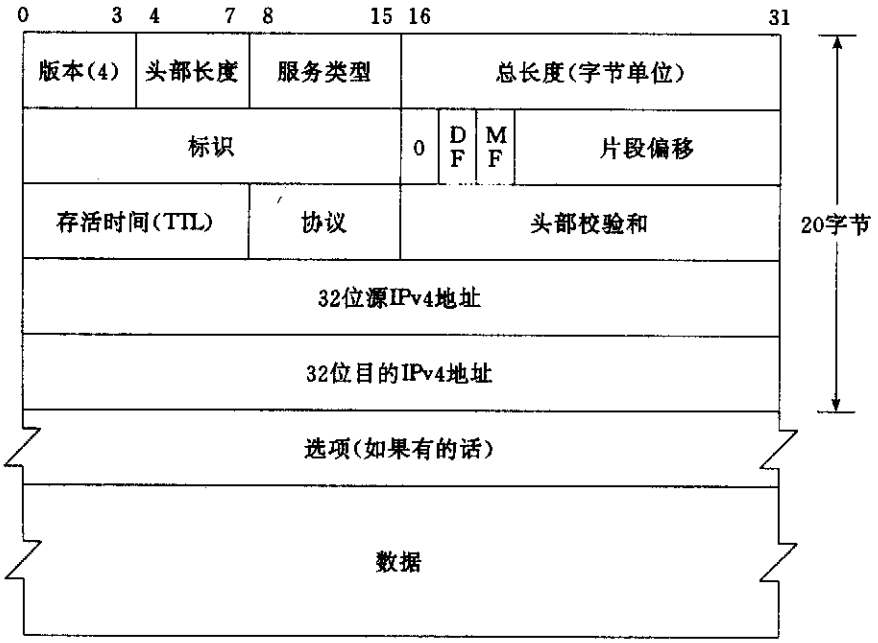
A.1 概 述

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])。

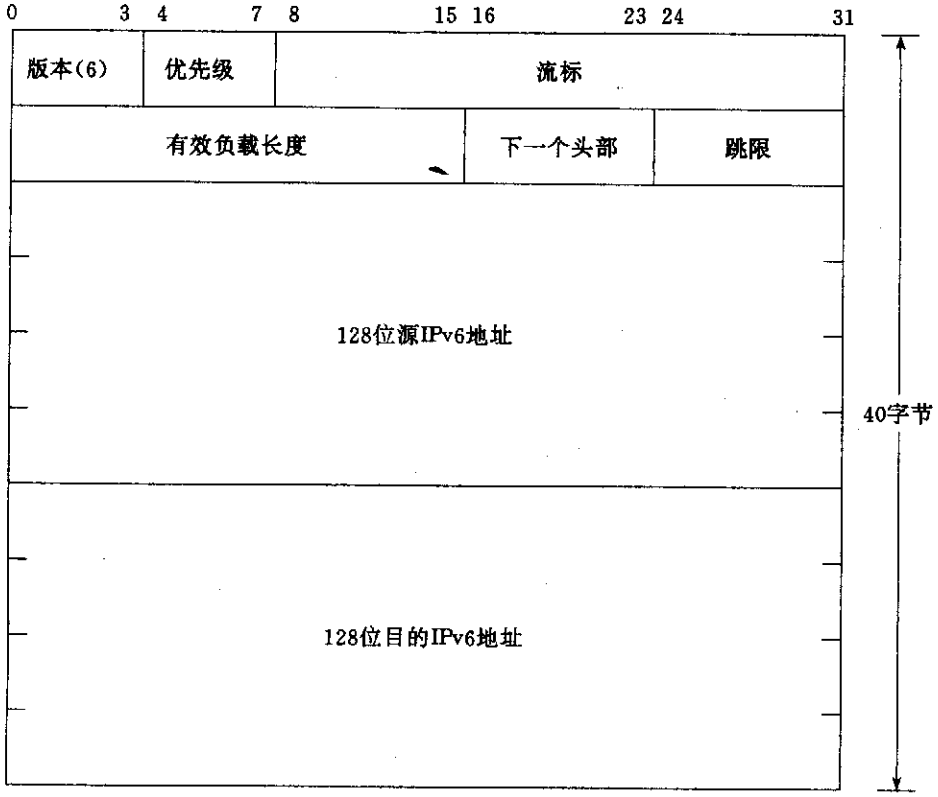


图 A.2 IPv6 头部格式

- 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:

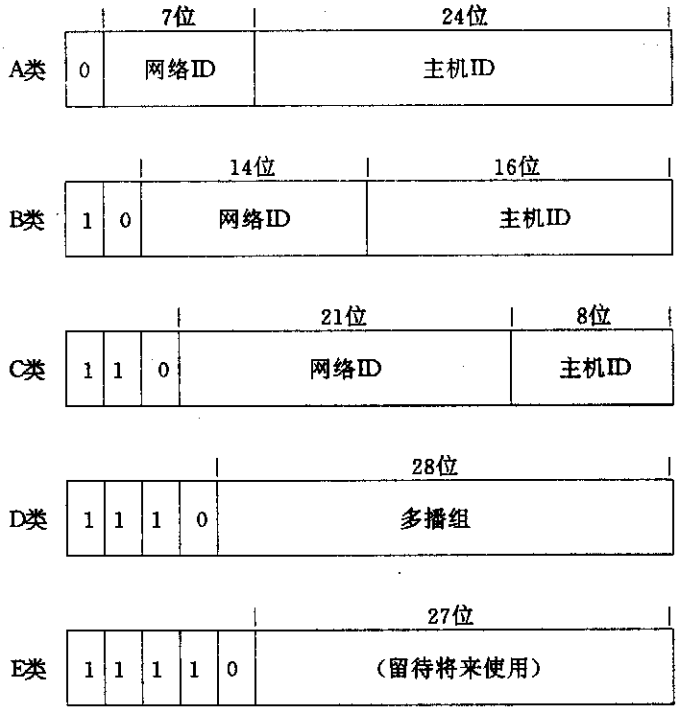
- IPv6 没有广播(第 18 章)。IPv4 中可选的多播(第 19 章)在 IPv6 中却是必需的。
- IPv6 路由器不给所转发的分组进行分片。IPv6 的分片只在起始主机上进行。

- 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

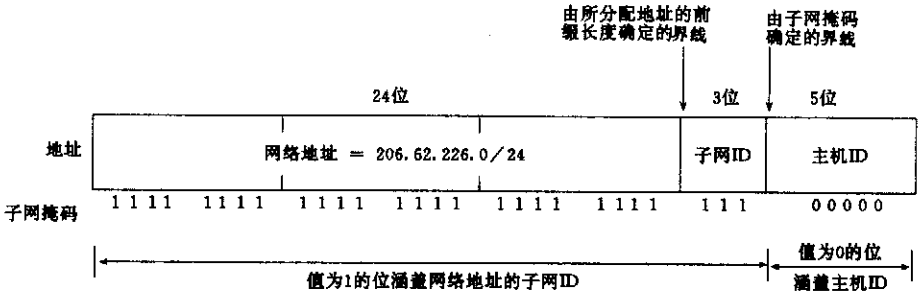
IPv4 地址通常划分成子网(RFC 950[Mogul and Postel 1985])。这么做增加了另外一级地址层次：

- 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。

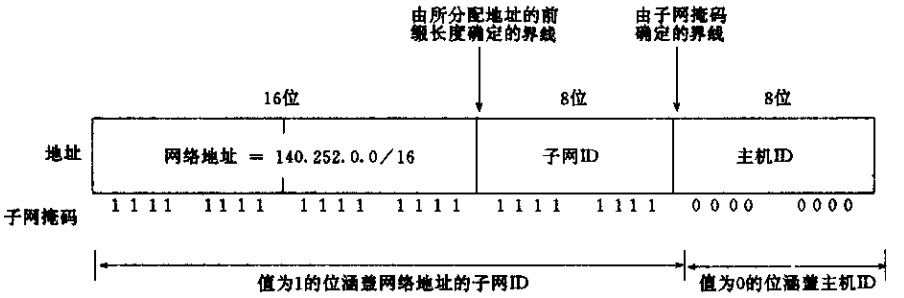


图 A.6 16 位网络地址加 8 位子网 ID 和 8 位主机 ID

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。

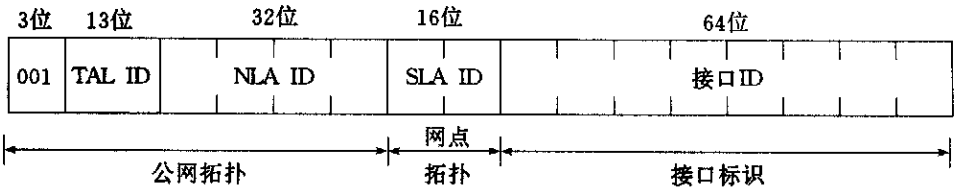


图 A.8 IPv6 可聚集全局单播地址

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。

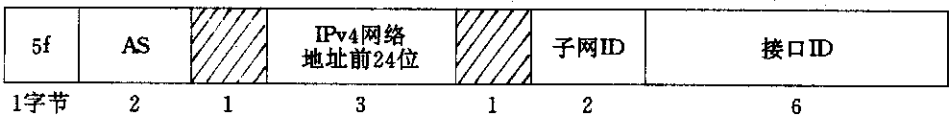


图 A.9 用于 6bone 的 IPv6 测试地址

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。

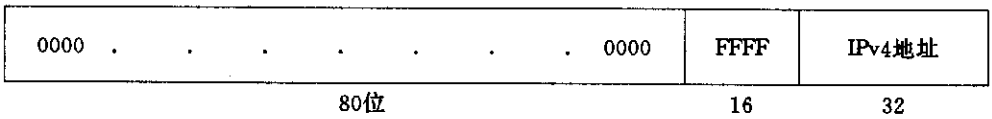


图 A.10 IPv4 映射的 IPv6 地址

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。

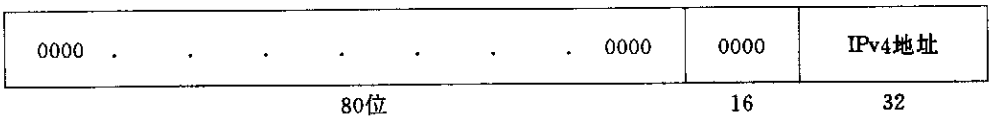


图 A.11　IPv4 兼容的 IPv6 地址

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。

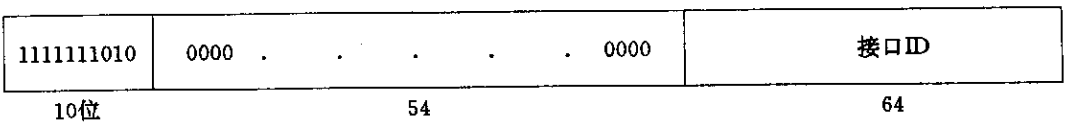


图 A.12　IPv6 链路局部地址

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。

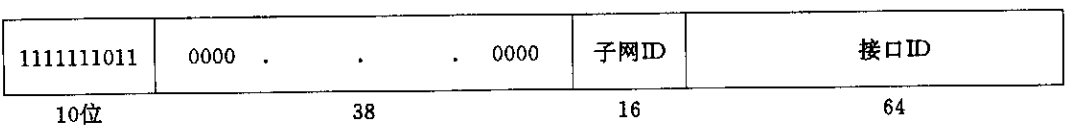


图 A.13　IPv6 网点局部地址

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。

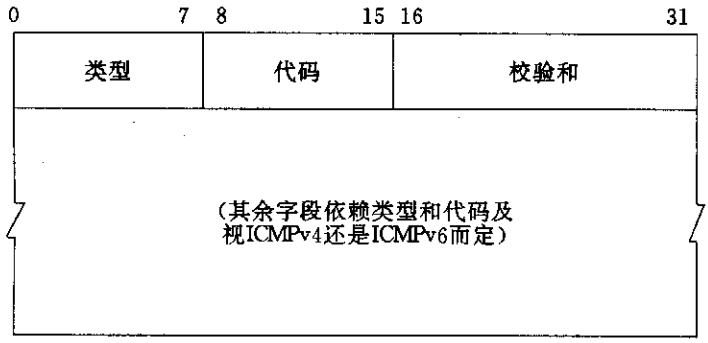


图 A.14 ICMPv4 和 ICMPv6 消息的格式

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。

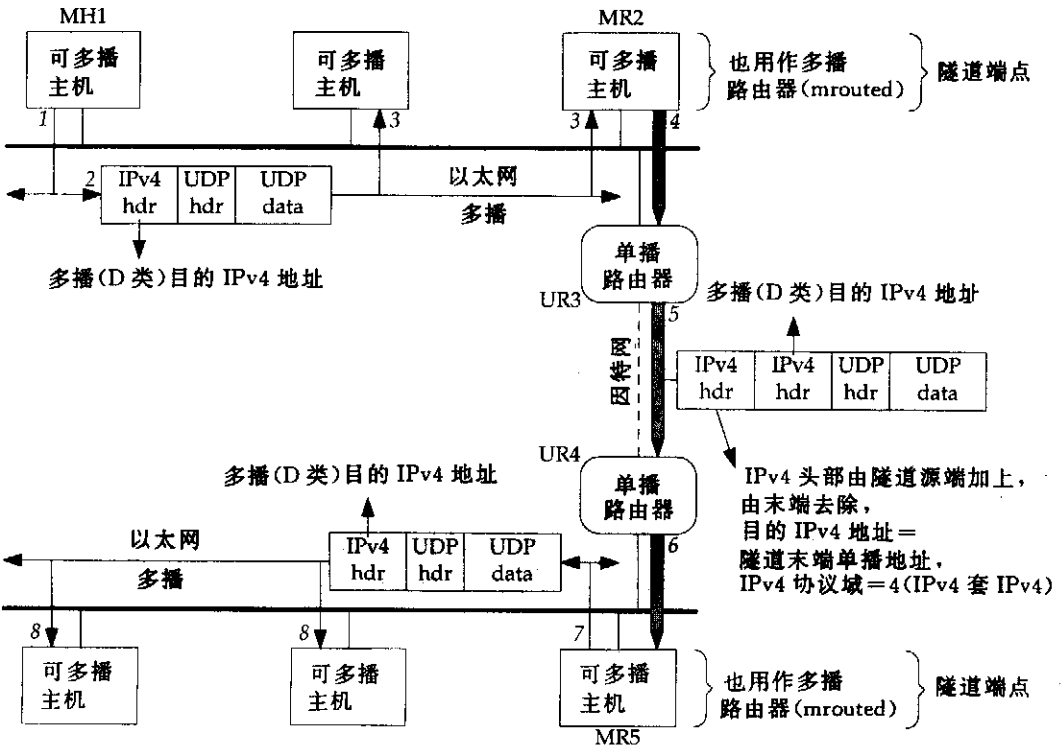


图 B.1 用于 MBone 的 IPv4 套 IPv4 封装

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。

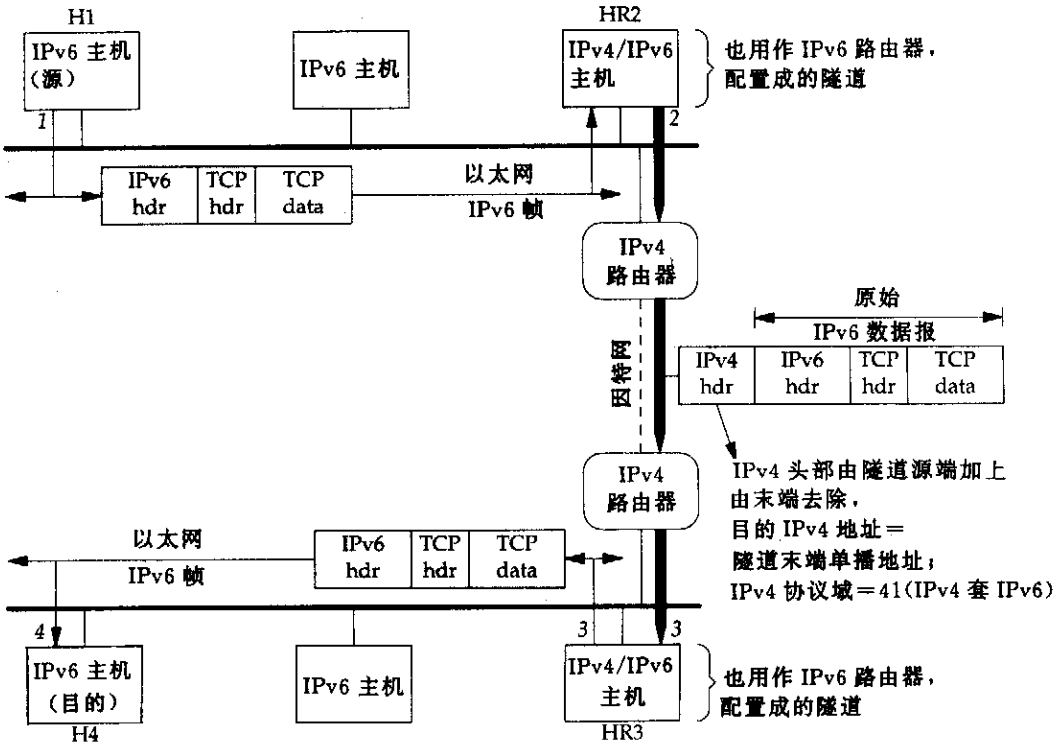


图 B.2 用于 6bone 的 IPv4 套 IPv6 封装

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。

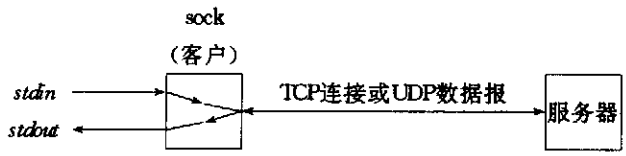
> 许多网点现在禁止穿越防火墙访问这些服务，这是因为1996年出现的一些拒绝服务型攻击用到了这些服务(习题12.3)。不过在你自己的网络内部，你是有希望使用这些服务的。

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。

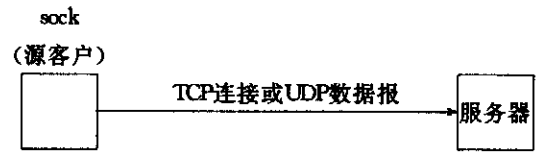
1. 标准输入，标准输出客户(图C.1)。



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。

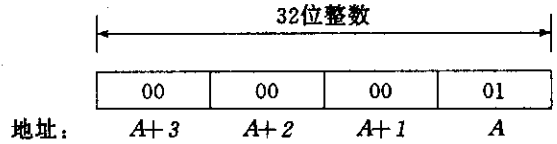


图 E.2 32 位整数值 1 的小端字节序格式表示

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。

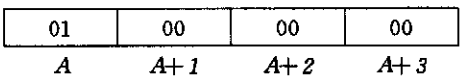


图 E.3 来自图 E.2 的 32 位整数以大端字节序格式表示

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。

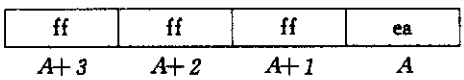


图 E.4 32 位整数值-22 的小端字节序格式表示

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。

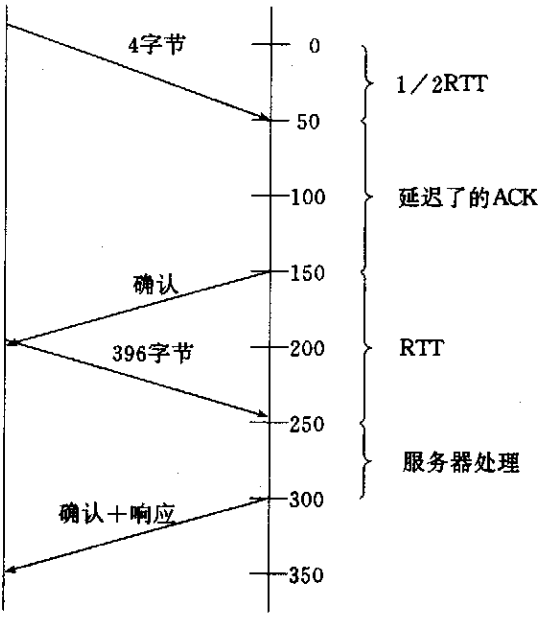


图 E.6　带延迟了的 ACK 的 Nagle 算法交互情况

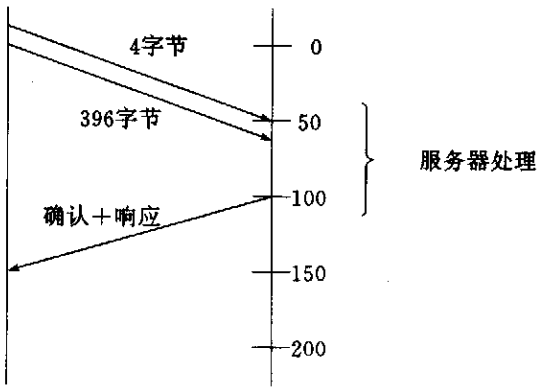


图 E.7　通过设置 TCP_NODELAY 套接口选项避免 Nagle 算法

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。

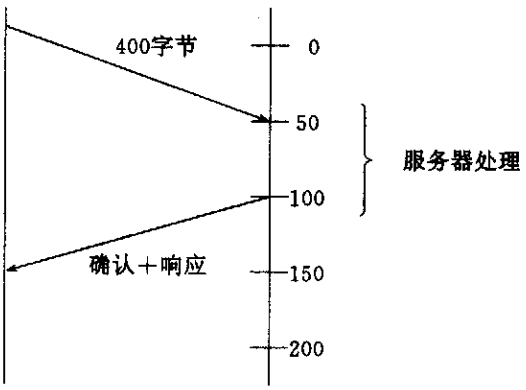


图 E.8　使用 writev 代替设置 TCP_NODELAY 套接口选项

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。

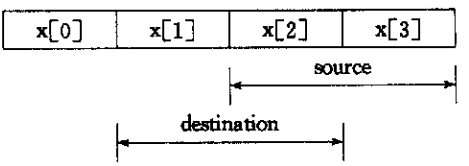


图 E.20 拷贝操作的源和目标重叠

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

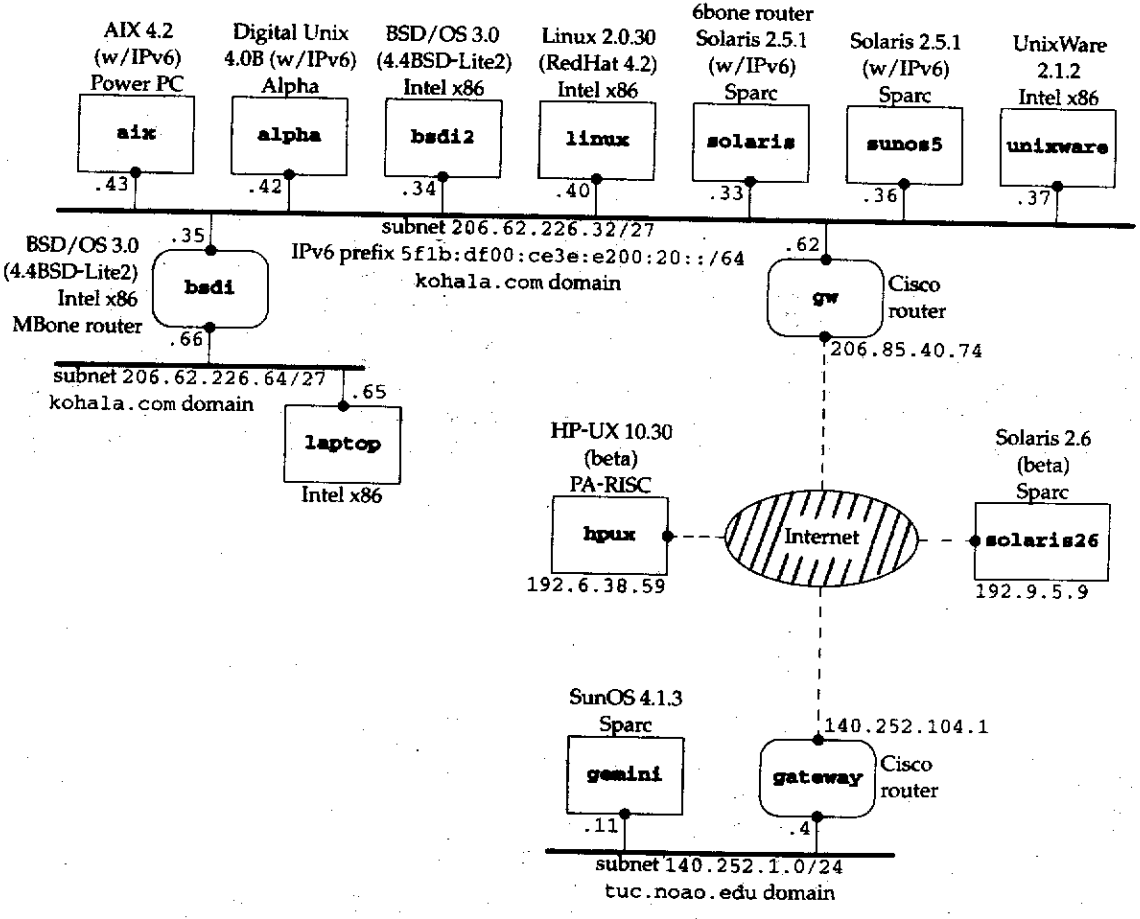
accept 83
bcmp 57
bcopy 57
bind 75
bzero 57
close 89
closelog 286
CMSG_xxx 311
connect 73
connect_nonb 349
connect_timeo 298
daemon_inetd 294
daemon_init 287
dg_send_recv 468
endnetconfig 669
endnetpath 670
err_doit 791
err_dump 791
err_msg 791
err_quit 791
err_ret 791
err_sys 791
execxxx 85
fcntl 175
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