附录E

精选习题答案

第1章

1.3 在Solaris上我们得到

```
solaris % daytimetcpcli 127.0.0.1
socket error: Protocol not supported
```

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查找字符串Protocol not supported。

```
solaris % grep 'Protocol not supported' /usr/include/sys/errno.h
#define EPROTONOSUPPORT 120     /* Protocol not supported */
```

这就是由socket返回的errno值。我们然后查看手册页面：

```
solaris % man socket
```

大多数手册页面在将近结束处形如“Errors”的标题下给出这个错误的额外信息，不过有些简洁。

1.4 我们把第一个声明改成：

```
int     sockfd, n, counter = 0;
```

再作为while循环的第一个语句加上如下行：

```
counter++;
```

最后在结束之前加上如下行：

```
printf("counter = %d\n", counter);
```


所显示的值总是1。

1.5 我们声明一个名为i的int变量，再把write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。如果客户和服务器运行在同一个主机上，那么计数器值通常是1，意味着尽管服务器调用了26次write，所写出数据也仅由客户的一次read返回。然而如果客户运行在Solaris 2.5.1上而服务器运行在BSD/OS 3.0上，那么计数器值通常是2。如果监视以太网上的分组，我们发现第一个字符自成一个分组发送，剩余25个字符包含在下一个分组内发送。（我们在7.9节就Nagle算法的讨论解释了如此行为的原因。）相反，如果客户运行在BSD/OS 3.0上而服务器运行在Solaris 2.5.1上，那么计数器值是26。如果监视以太网上的分组，我们发现每个字符自成一个分组发送。

本例子的目的在于强调不同的TCP对数据做不同的处理，我们的应用程序必须做好作为字节流读入这些数据的准备，直到遇上数据流末尾。

第2章

2.1 访问http://www.iana.org/numbers.htm，找到名为“IP Version Number”的注册处，我们看到版本0是保留的，版本1～3未曾分配，版本5是网际网流协议（Internet Stream Protocol）。

2.2 所有RFC都可以通过电子邮件、匿名FTP或Web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。可以从取得当前RFC索引开始，它通常是文件rfc-index.txt（也可以取得它的HTML版本http://www.rfc-editor.org/rfc-index.html）。使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“Stream”一词，我们发现RFC 1819定义了网际网流协议的版本2。无论何时查找可能是由某个RFC涵盖的信息，别忘了先搜索RFC索引。

2.3 对于IPv4这个默认值产生576字节的IP数据报（其中IPv4首部占用20字节，TCP首部占用20字节，剩下536字节的TCP净荷），这是IPv4的最小重组缓冲区大小。

2.4 本例子中执行主动关闭操作的是服务器而不是客户。

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，但是为了避免分片，它不会超过外出接口（即以太网）的MTU。TCP净荷不能超过由对端宣告的MSS，但是净荷小于这个数量的TCP分节总是可以发送的。

2.6 Assigned Numbers网页（http://www.iana.org/numbers.htm）中的“Protocol Numbers”注册处给出OSPF的协议号为89。

2.7 选择性确认只是表明由选择性确认消息反映的序列号所涵盖的数据已被接收，而累积确认表明由累积确认消息中的序列号指示的所有以前的数据都已被接收。如果从发送缓冲区中基于选择性确认释放数据，那么系统只能释放确切被确认的数据，而不能释放之前或之后的任何数据。 914

第3章

3.1 C中函数不能改变按值传递的参数的值。要让被调用的函数修改由调用者传入的某个值，调用者必须传递指向这个待修改值的一个指针。

3.2 指针必须按所读或所写的字节数增长，但是C不允许void指针如此增长（因为C编译器不知道void指针指向的数据类型）。

第4章

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_打头的各个常值的定义。譬如说D类多播地址INADDR_MAX_LOCAL_GROUP的定义是0xe00000ff，其注释是“224.0.0.255”，它显然是按主机字节序定义的。

4.2 下面是在connect调用之后新添加的若干行：

```
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变量并声明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二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时引用计数从2递减为1，因此不会向客户发送FIN。以后当父进程调用close时引用计数递减为0，于是发送FIN。

4.4　accept返回EINVAL，因为它的第一个参数不是一个监听套接字描述符。

4.5　不调用bind的话，listen调用赋予监听套接字一个临时端口。

第5章

5.1　TIME_WAIT状态的持续时间应该在1分钟到4分钟之间，前提是MSL在30秒钟到2分钟之间。

5.2　把一个二进制文件作为客户的标准输入时我们的客户/服务器程序并不工作。假设前3个字节为二进制数1、二进制数0和一个换行符。图5-5中fgets调用最多读入MAXLINE-1个字符，除非碰到换行符或已到达文件尾而提前返回。在本例子中它将读入前3个字符，然后以一个空字节结束待返回的字符串。然而图5-5中strlen调用返回的是1，因为它只计到第一个空字节。客户于是只把第一个字节发送给服务器，导致服务器阻塞在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。这就是所谓的死锁（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。这里的问题是fgets以一个空字节表征所返回数据的结尾，因此它读入的数据不能含有任何空字节。

915

5.3　Telnet把输入行转换成NVT ASCII（TCPv1的26.4节），意味着以CR（回车符）后跟LF（换行符）的双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，我们仍然可以使用Telnet客户与我们的服务器通信，因为我们的服务器回射每个字符，包括每个换行符之前的回车符。

5.4　连接终止序列的最后两个分节并不发送。我们杀掉服务器子进程之后（在客户输入“another line”之前），客户向服务器发送数据导致服务器TCP响应以一个RST。这个RST使得连接中止，并防止连接的服务器端（执行主动关闭的那一端）经历TIME_WAIT状态。

5.5　没有什么变化，因为在服务器主机上新启动的服务器进程创建一个监听套接字就等待新的连接请求的到达。我们在步骤3发送的是需进入某个ESTABLISHED状态TCP连接的数据分节，而新启动的服务器在其监听套接字上绝看不到这些数据分节，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Solaris上运行它产生如下输出：

```
solaris % tsigpipe 192.168.1.10
SIGPIPE received
write error: Broken pipe
```

第一个2秒钟的sleep用于让daytime服务器发送应答并关闭它的连接所在端。第一个write导致发送一个数据分节到服务器，服务器则响应以RST（因为daytime服务器已经完全关闭了它的套接字）。注意TCP允许我们继续写出到一个已收到FIN的套接字。第二个sleep让客户接收到服务器的RST，于是第二个write引发SIGPIPE信号。既然信号处理函数返回主控制流，write于是返回一个EPIPE的错误。

tcpcliserv/tsigpipe.c

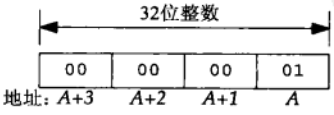
```
 1 #include      "unp.h"
 2 void
 3 sig_pipe(int signo)
 4 {
 5       printf("SIGPIPE received\n");
 6       return;
 7 }
 8 int
 9 main(int argc, char **argv)
10 {
11       int     sockfd;
12       struct sockaddr_in  servaddr;
13       if (argc != 2)
14           err_quit("usage: tcpcli <IPaddress>");
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 }
```

tcpcliserv/tsigpipe.c

图E-1 产生SIGPIPE

5.7 假设服务器主机支持弱端系统模型（weak end system model，在8.8节讲解），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IP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（本例子中它承载一个TCP分节）。我们可以如下测试这一点：在主机linux（图1-16）上运行服务器，然后在主机solaris上启动客户，不过给客户指定的是服务器主机的另一个IP地址（206.168.112.96）。连接建立之后如果在服务器主机上运行netstat，我们将看到该连接的本地IP地址是来自客户SYN的目的IP地址，而不是SYN到达数据链路的IP地址（这跟我们在4.4节提及的一样）。

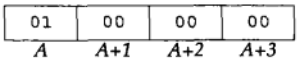
5.8 我们的客户运行在小端字节序的Intel系统上，那儿32位整数值1按图E-2所示格式存放。



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*的顺序通过套接字发送，然后以如图E-3所示的大端

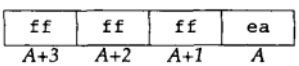
字节序格式存放。



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就是16777216。类似地，由客户发送的整数2将被服务器解释成0x02000000即33554432。这两个整数的和是50331648即0x03000000。服务器把这个大端字节序值发送给客户后，客户把它解释成整数3。

然而32位整数值–22在小端字节序系统上如图E-4表示，采用的是负数的二进制补码表示。



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-352321537。类似地，–77的小端字节序表示是0xffffffb3，但是在大端字节序服务器主机上却表示成0xb3ffffff即-1275068417。服务器上的这两个整数相加的结果是0x9efffe即-1627389954。这个大端字节序的值通过套接字发送给客户后以小端字节序解释的值是0xfefff9e即-16777314，它就是我们的例子所显示的值。

5.9　技术路线是正确的(把二进制值转换成网络字节序表示)，但是不能使用htonl和ntohl这两个函数。尽管这两个函数中的l曾经表意“long”(长整数)，它们却只是操作在32位整数上（3.4节）。64位系统上一个长整数可能占据64位，这两个函数就不能正确工作了。有人也许定义hton64和ntoh64这两个函数来解决本问题，但是它们在使用32位表示长整数的系统上又不能工作。

5.10　第一种情形下服务器将永远阻塞在图5-20的readn中，因为客户发送的是2个32位值，但是服务器等待的却是2个64位值。这两个主机之间对换客户和服务器将导致客户发送2个64位值，但是服务器只读入第一个64位，并把它解释成2个32位值。第二个64位值仍然在服务器的套接字接收缓冲区中。服务器往回写出1个32位值，但是客户却仍然永远阻塞在图5-19的readn中，等待读入1个64位值。

5.11　IP路由功能查看目的IP地址（服务器主机的IP地址），搜索路由表确定外出接口和下一跳（ICPv1第9章）。外出接口的主IP地址用作源IP地址，前提是该套接字尚未绑定某个本地IP地址。

918

第6章

6.1　这个整数数组包含在一个结构中，而C是允许结构跨等号赋值的。

6.2　如果select告诉我们某个套接字可写，该套接字的发送缓冲区就有8192字节的可用空间，但是当我们以8193字节的缓冲区长度对这个阻塞式套接字调用write时，write将会阻塞，等待最后1个字节的可用空间。对阻塞式套接字的读操作只要有数据总会返回一个不足计数（short count），然而对阻塞式套接字的写操作将一直阻塞到所有数据都能被内核接受为止。可见当使用select测试某个套接字的可写条件时，我们必须

把该套接字预先设置成非阻塞以避免阻塞。

6.3 如果两个描述符都可读，那么只执行第一个测试，它测试的是套接字描述符。不过这么做并没有导致客户程序不能工作，它只是降低了效率而已。这就是说，如果select返回值表明两个描述符均可读，那么第一个if语句为真，导致客户从套接字readline并fputs到标准输出。下一个if语句却被跳过（就因为我们在这个if关键词之前所冠的else关键词），不过select接着再次被调用，它马上发现标准输入可读，于是立即返回。这里的关键概念是清除“标准输入可读”条件的不是select的返回，而是从标准输入真正地读入。

6.4 使用getrlimit函数取得RLIMIT_NOFILE资源的当前值，然后调用setrlimit把当前软限制（rlim_cur）设置成硬限制（rlim_max）。举例来说，Solaris 2.5上描述符数目的软限制是64，但是任何进程都可以把它增长到默认的硬限制1024。

getrlimit和setrlimit不属于POSIX.1，但是在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 以参数SHUT_RDWR或SHUT_WR调用shutdown总是发送FIN，而close只在调用时描述符引用计数为1的条件下才发送FIN。

6.7 read返回一个错误，我们的Read包裹函数于是终止服务器。服务器不应该如此脆弱。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健壮。考虑客户和服务器之间丧失连接的情况，服务器某个响应的发送尝试最终发生超时，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中止。它应该登记错误，关闭出错套接字，并继续服务其他客户。对于像这样由单个进程来应付所有客户的服务器来说，简单中止的错误处理方式是难以接受的。然而如果服务器是由一个子进程来应付仅仅一个客户，那么中止某个子进程并不会影响父进程（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。

919

第7章

7.2 图E-5给出了本习题的解答之一。我们去掉了用于显示由服务器返回的数据串的那些语句，因为本例子用不着这些值。

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（尤如Solaris 2.5.1及更早版本）总是返回0值的套接字缓冲区大小，使得我们无法查看该值在连接前后有什么事发生。

至于MSS，在connect之前显示的值是实现的默认值（通常是536或512），在connect之后显示的值则取决于可能有的来自对端的MSS选项。举例来说，本地以太网上connect之后的值可能是1460。然而connect某个远程网络上的一个服务器主机之后显示的MSS值可能类似默认值，除非你的系统支持路径MTU发现功能。如果可能的话，在程序运行期间运行一个像tcpdump（C.5节）这样的工具，以查看来自对端的SYN分节中的真正MSS选项。

至于套接字接收缓冲区的大小，许多实现在连接建立之后把它向上舍入成MSS的倍数。查看连接建立之后套接字接收缓冲区大小的另一个方法是使用像tcpdump这样的工具监视分组，观察TCP的通告窗口（advertised window）。

sockopt/rcvbuf.c

```
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 }
```

sockopt/rcvbuf.c

图E-5　在连接建立前后显示套接字接收缓冲区大小和MSS值

7.3　分配一个名为ling的linger结构并如下初始化它：

```
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而不是正常的4分节交换终止连接。服务器子进程的readline调用返回ECONNRESET错误，所显示的消息如下：

```
readline error: Connection reset by peer
```

尽管执行主动关闭的是客户，它也不应该经历TIME_WAIT状态。

7.4　第一个客户调用setsockopt、bind和connect。如果第二个客户在第一个客户调用bind和connect之间调用bind，那么它将返回EADDRINUSE错误。然而一旦第一个客户已连接到对端，第二个客户的bind就正常工作，因为第一个客户的套接字当时处于已连接状态。处理这种竞争状态的唯一办法是让第二个客户在bind调用返回EADDRINUSE

错误的情况下再尝试调用bind多次，而不是一返回该错误就放弃。

7.5 我们在支持多播的一个主机（MacOS X 10.2.6）上运行这个程序[①]。

```
macosx % sock -s 9999 &                                  以通配地址启动第一个服务器
[1]         29297
macosx % sock -s 172.24.37.78 9999                       不使用-A尝试启动第二个服务器
can't bind local address: Address already in use
macosx % sock -s -A 172.24.37.78 9999 &                  使用-A再次尝试；成功
[2]         29699
macosx % sock -s -A 127.0.0.1 9999 &                     使用-A启动第三个服务器；成功
[3]         29700
macosx % netstat -na | grep 9999
tcp4       0   0   127.0.0.1.9999         *.*          LISTEN
tcp4       0   0   172.24.37.78.9999      *.*          LISTEN
tcp4       0   0   *.9999                 *.*          LISTEN
```

920 l 921

7.6 我们首先在支持多播但不支持SO_REUSEPORT选项的一个主机（Solaris 9）上尝试[②]。

---

① 本书第2版解答如下。我们在不支持多播的一个主机（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    127.0.0.1.9999       *.*          LISTEN
tcp    0    0    206.62.226.37.9999   *.*          LISTEN
tcp    0    0    *.9999                    *.*     LISTEN          ——译者注
```

② 本书第2版解答如下。我们首先在不支持多播的一个主机（UnixWare 2.1.2）上尝试。

```
unixware % sock -s -u -A 206.62.226.37 8888 &            第一个服务器启动
[4]    29707
unixware % sock -s -u -A 206.62.226.37 8888
can't bind local address: Address already in use         不能启动第二个服务器
```

我们给这两个运行实例都指定了SO_REUSEADDR选项，但是它不起作用。

我们接着在支持多播但不支持SO_REUSEPORT选项的一个主机（Solaris 2.6）上尝试。

```
solaris26 % sock -s -u 8888 &                            第一个服务器启动
[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启动第二个服务器；成功
solaris26 % netstat -na | grep 8888                      我们看到重复的捆绑
      *.8888             Idle
      *.8888             Idle
```

这个系统上第一个bind不必指定SO_REUSEADDR，但是第二个起必须指定。

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。我们首先给一前一后两个服务器尝试SO_REUSEADDR选项，但是它不起作用。

```
bsdi % sock -u -s -A 7777 &
[1]    17610
bsdi % sock -u -s -A 7777
can't bind local address: Address already in use
```

接着只给第二个服务器而不给第一个服务器尝试SO_REUSEPORT选项。这也不起作用，因为完全重复的捆绑要求共享同一捆绑的所有套接字都使用该选项。

```
bsdi % sock -u -s 8888 &
[1]    17612
bsdi % sock -u -s -T 8888
can't bind local address: Address already in use
```

最后给两个服务器都指定SO_REUSEPORT选项，这是起作用的。

```
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          *.*              ——译者注
```

```
solaris % sock -s -u 8888 &                          第一个服务器启动
[1]             24051
solaris % sock -s -u 8888
can't bind local address: Address already in use
solaris % sock -s -u -A 8888 &                       使用-A启动第二个服务器；成功
solaris % netstat -na | grep 8888                    我们看到重复的捆绑
      *.8888                                 Idle
      *.8888                                 Idle
```

这个系统上第一个bind不必指定SO_REUSEADDR，但是第二个起必须指定。

最后我们在既支持多播又支持SO_REUSEPORT选项的MacOS X 10.2.6上进行尝试。我们首先给一前一后两个服务器尝试SO_REUSEADDR选项，但是它不起作用。

```
macosx % sock -u -s -A 7777 &
[1]             17610
macosx % sock -u -s -A 7777
can't bind local address: Address already in use
```

接着只给第二个服务器而不给第一个服务器尝试SO_REUSEPORT选项。这也不起作用，因为完全重复的捆绑要求共享同一捆绑的所有套接字都使用该选项。

```
macosx % sock -u -s 8888 &
[1]         17612
macosx % sock -u -s -T 8888
can't bind local address: Address already in use
```

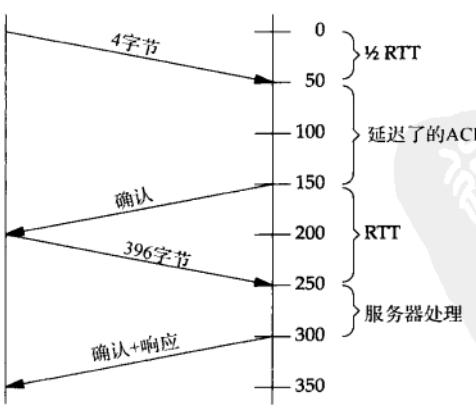
最后给两个服务器都指定SO_REUSEPORT选项，这是起作用的。

```
macosx % sock -u -s -T 9999 &
[1]         17614
macosx % sock -u -s -T 9999 &
[2]         17615
macosx % netstat -na | grep 9999
udp4       0      0      *.9999         *.*
udp4       0      0      *.9999         *.*
```

7.7　它不起任何作用，因为ping使用ICMP套接字，而SO_DEBUG套接字选项只影响TCP套接字。SO_DEBUG套接字选项的描述一直有些笼通，譬如说“这个选项开启相应协议层中的调试”，但是实现该选项的唯一协议层一直只是TCP。

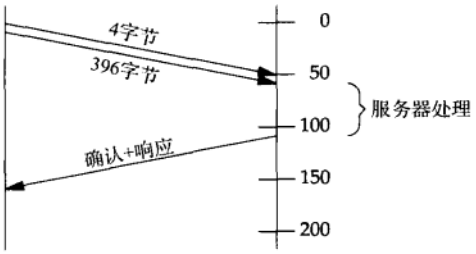
922

7.8　图E-6给出了时间线。



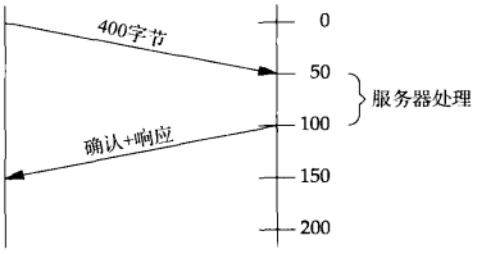
图E-6　Nagle算法与延滞ACK的交互情况

7.9　设置TCP_NODELAY套接字选项导致来自第二个write的数据被立即发送，即使该连接上还有一个尚未得到确认的小分组。这种情况如图E-7所示。本例子中总时间只稍微超过150 ms。



图E-7　通过设置TCP_NODELAY套接字选项避免Nagle算法

7.10　这种办法的优点是减少了分组的个数，如图E-8所示。



图E-8　使用writev代替设置TCP_NODELAY套接字选项

7.11　4.2.3.2节声称“延滞必须低于0.5秒钟，而且在完全大小分节流上至少每隔一个分节就应该有一个ACK”。源自Berkeley的实现延滞ACK最久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阻塞于accept调用中，图5-3中的子进程则大部分时间花在阻塞于read调用中，它是由readline调用的。保持存活选项对于监听套接字不起作用，因此父进程不受客户主机崩溃影响。子进程的read将返回ETIMEDOUT错误，它在跨连接的最后一次数据交换之后约2小时发生。

923

7.13　图5-5中的客户大部分时间花在阻塞于fgets调用中，fgets本身则阻塞在标准I/O函数库中对于标准输入某种类型的读操作中。当跨连接的最后一次数据交换之后约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没有诱发来自服务器的响应时，套接字的待处理错误被设置成ETIMEDOUT。然而客户阻塞在对于标准输入的fgets调用中，因此看不到这个错误，直到对于套接字执行读或写操作。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阻塞于select调用中，一旦待处理错误被设置成ETIMEDOUT（如上一题的解答所述），select就立即返回套接字的可读条件。

7.15　只交换2个而不是4个TCP分节。两个系统的定时器精确同步的可能性非常低；因此一端的保持存活定时器会比另一端略早一点超时。首先超时的那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

把保持存活定时器重置成2小时。

7.16　最初的套接字API并没有listen函数。相反，socket函数的第四个参数含有套接字选项，而SO_ACCEPTCON就是用来指定监听套接字的。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

8.1　是的。read返回4096字节的数据，recvfrom则返回2048字节（2个数据报中的第一个）。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字地址结构，clilen很可能太大。我们将在第15章看到这对于Unix域套接字地址结构是可以接受的，不过正确编写这个函数的方式是把由recvfrom返回的真正长度用作sendto的长度。

924

8.4　像这样运行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钟1次降低到每60秒钟1次以减少输出量。如果我们在主机aix上运行我们的UDP客户程序，所指定的服务器IP地址为192.168.42.1，同时运行ping程序，就会得到如下输出（注意即使指定-v选项，也并非所有ping客户程序都显示所接收到的ICMP错误）：

```
aix % ping -v -i 60 127.0.0.1
PING 127.0.0.1: (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=255 time=0 ms
36 bytes from 192.168.42.1: Destination Port Unreachable
Vr HL TOS  Len   ID Flg  off TTL Pro  cks      Src      Dst Data
 4  5  00 0022 0007   0 0000  1e  11 c770 192.168.42.2  192.168.42.1
UDP: from port 40645, to port 9877  (decimal)
```

8.5　监听TCP套接字也许有一个套接字接收缓冲区大小，但是它绝不会接受数据。大多数实现并不预先给套接字发送缓冲区或接收缓冲区分配内存空间。使用SO_SNDBUF和SO_RCVBUF套接字选项指定的套接字缓冲区大小仅仅是给套接字设定的上限。

8.6　我们在多宿主机freebsd上指定-u选项（使用UDP）和-l选项（指定本地IP地址和端口）运行sock程序。

```
freebsd % sock -u -l 12.106.32.254.4444 192.168.42.2 8888
hello
```

本地IP地址在图1-16中是主机freebsd的因特网侧接口，但是数据报要到达目的地则必须从另一个接口出去。[①]使用tcpdump监视网络表明源IP地址确实是由客户绑定的那个地址，而不是外出接口的地址。

```
14:28:29.614846 12.106.32.254.4444 > 192.168.42.2.8888: udp 6
14:28:29.615225 192.168.42.2 > 12.106.32.254: icmp: 192.168.42.2
                                                udp port 8888 unreachable
```

8.7　在客户程序中放一个printf调用会在每个数据报之间引入一个延迟，从而允许服务器接收更多的数据报。在服务器程序中放一个printf调用则会导致服务器丢失更多数据报。

8.8　IPv4数据报最大为65535字节，这由图A-1中16位的总长度字段限定。IPv4首部需要20

① “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ICMP端口不可达错误。——译者注

字节，UDP首部需要8字节，留给UDP用户数据最大65507字节。对于没有任何扩展首部的IPv6数据报而言（自然没有特大报支持，因为特大净荷长度是一个步跳选项，出现在步跳选项扩展首部中），扣除IPv6首部的净荷最大为65535字节（图A-2），再扣除8字节UDP首部，留给UDP用户数据最大65527字节。①

图E-9给出了dg_cli函数的新版本。如果没有预先设置发送缓冲区大小，源自Berkeley的内核就给sendto调用返回EMSGSIZE错误，因为套接字发送缓冲区的默认大小通常不足以暂存最大的UDP数据报（先做完习题7.1）。然而如果我们运行如图E-9所示客户 925
程序，先设置客户套接字缓冲区大小再发送和接收UDP数据报，那么服务器不返送任何数据报。我们可以运行tcpdump验证客户的数据报发送到了服务器上，但是如果在服务器程序中放一个printf调用，我们就发现它的recvfrom调用并没有返回这个数据报。问题出在服务器的UDP套接字接收缓冲区小于我们发送的数据报，因此该数据报被丢弃掉而不是被递送到套接字。在FreeBSD系统上我们可通过运行netstat -s命令，并查看接收这个大数据报前后“dropped due to full socket buffers”（因套接字缓冲区满而丢弃）计数器值的变化加以验证。最终的办法就是修改服务器程序，预先设置它的套接字发送缓冲区与接收缓冲区的大小。

udpcliserv/dgclibig.c

```
 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 }
```

udpcliserv/dgclibig.c

图E-9 写出最大大小的UDP/IPv4数据报

大多数网络上65535字节的IP数据报需要分片。回顾2.11节，我们知道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

9.1 总地说来，整体上接受短期请求偶尔需要长期会话的应用系统可以利用sctp_peeloff。举例来说，某个类似传统UDP的SCTP应用系统中，服务器通常有如 926
短期事务处理那般响应客户的请求，不过偶尔被请求执行长期的音频数据传送。大多

① 本书第2版Stevens先生可能据于早期IPv6规范得出UDP/IPv6用户数据最大为65 487字节（65535-40-8），第3版新作者尽管一直在强调最新的IPv6规范，在UDP/IPv6用户数据的计算上却仍然使用第2版不合时的推导。译者对此做了修正。——译者注

数情况下服务器发送少数几个短小的UDP消息就行了；然而一旦音频请求到达，长期会话就被激活，以发送音频信息。这种情形下可以使用该函数把音频流剥离出来给专门的线程或进程处理。

9.2　因为SCTP不支持半关闭状态，造成当客户调用close时，关联终止序列把来自服务器的任何已排队但尚未处理的未决数据冲刷掉以终止关联，达到关闭关联目的。

9.3　对于一到一式套接字客户必须首先调用sctp_connect显式建立关联，但是该函数没有额外指定数据的参数，因此无法在四路握手的第三个分组中随COOKIE ECHO消息携带数据。对于一到多式套接字客户不比先建立关联再发送数据，而是可以调用sctp_sendto同时完成两者，也就是说这样发送的数据随COOKIE ECHO消息被IP数据报载送到对端，这样的关联是隐式建立的。

9.4　本端准备与之建立关联的对端在关联建立阶段能够发送回数据的唯一可能情形是它在关联建立之前就准备好了数据。当两端都使用一到多式套接字几乎同时发送数据隐式建立关联时就会发生这种情形。这种关联建立称为INIT冲突，详见［Stewart and Xie 2001］第4章。

9.5　某些情况下并非所有绑定的地址都可以传递到对端端点。具体地说，如果某个应用进程绑定的IP地址中既有公用的又有私用的，那么可能只有公用地址可以与对端共享。另一个例子是IPv6的链路局部地址不一定能够与对端共享。

第10章

10.1　如果sctp_sendmsg调用返回错误，那就不会发送任何消息，客户进程于是阻塞在sctp_recvmsg调用中，等待永远不会返送回来的响应消息。解决该问题的办法显然是检查这个函数调用的返回值，如果发现消息发送出错，那就报告错误而不再接收。
如果sctp_recvmsg调用返回错误，那就不会有响应消息达到，客户进程循环回去继续尝试发送消息，可能导致建立新的关联。避免这个问题的办法也是检查这个函数调用的返回值，根据情况或者报告错误并关闭套接字，从而让服务器也收到一个错误，或者若错误是暂时的则重新尝试sctp_recvmsg调用。

927　10.2　如果服务器在接收一个请求后退出，客户将被永远挂起，等着决不会到来的消息。客户检测这种情况的方法之一是开启关联事件。这样当服务器退出时客户将收到一个消息，告知客户该关联已经不复存在。客户可就此采取恢复手段，譬如说联系另外一个服务器。方法之二是客户启动一个定时器，过一段时间收不到响应消息就取消关联。

10.3　我们选择800字节是为了试图让每个SCTP块处于单个分组中。更好的方法也许是通过SCTP_MAXSEG套接字选项确定适合一个SCTP块的大小。

10.4　Nagle算法（由SCTP_NODELAY套接字选项控制，见7.10节）只会在选择较小的数据传送大小前提下导致问题。只要以迫使SCTP立即发送的大小写出数据就不会发生危害。然而如果给out_sz选择一个偏小的值，结果就会发生扭曲，暂缓发送某些数据以等待来自对端的SACK。因此如果使用较小的大小值，那么禁止Nagle算法（即开启SCTP_NODELAY套接字选项）可能比较可取。

10.5　如果应用进程在建立一个关联之后改动流的数目，那么这个关联的实际流数不会改变。这是因为流数的变更仅仅影响新的关联，而不影响现有关联。

10.6 一到多式套接字允许隐式设置关联。为了使用辅助数据更改某个关联的设置，我们首先需要使用sendmsg调用把这些数据提供给对端。因此请求更多的流要求使用辅助数据通过sendmsg进行隐式关联重新设置。

第11章

11.1 图E-10给出了调用gethostbyaddr的程序。

names/hostent2.c

```
 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 }
```

names/hostent2.c

图E-10 图11-3改成调用gethostbyaddr的结果

本程序针对只有一个IP地址的主机运行没有问题。如果针对拥有8个IP地址的一个主机运行图11-3中的程序，我们得到如下输出：

```
freebsd % hostent cnn.com
official hostname: cnn.com
        address: 64.236.16.20
        address: 64.236.16.52
        address: 64.236.16.84
        address: 64.236.16.116
        address: 64.236.24.4
        address: 64.236.24.12
        address: 64.236.24.20
        address: 64.236.24.28
```

但是如果我们针对同一个主机运行图E-10中的程序，那么它只输出其中一个IP地址：

```
freebsd % hostent2 cnn.com
official hostname: cnn.com
        address: 64.236.24.4
        name = www1.cnn.com
```

问题在于gethostbyname和gethostbyaddr这两个函数共享同一个hostent结构，就如11.18节开首部分所示。当我们的新程序在调用gethostbyname之后调用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7个IP地址。

928～929

11.2　如果你的系统不支持重入版本的gethostbyaddr（我们将在11.19节讲解），那么你必须在调用gethostbyaddr之前复制由gethostbyname返回的指针数组以及由该数组所指向的数据。

11.3　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）。

11.4　这是较新版本BIND的一个特性，不过POSIX没有规定这种处理方式，在可移植程序中不能依赖它。图E-11给出了图11-4中程序的修改后版本。对主机名字符串的测试顺序很重要。我们首先调用inet_pton，因为它是一个快速的全内存访问测试函数，用于判定主机名字符串是不是一个有效的点分十进制数IP地址。仅当这种测试失败时我们才调用gethostbyname，它往往牵涉某些网络资源，因而得花一段时间。

如果这个字符串是一个有效的点分十进制数IP地址，我们就自行构造指向这个IP地址的指针数组（addrs），它使得以后使用pptr的循环代码保持不变。

既然主机名字符串已被转换成套接字地址结构中的二进制数格式，我们于是进入使用pptr的循环。我们把图11-4中的memcpy调用改为memmove（两者功能相同，不过后者能够正确处理源目的内存区重叠的情形），这是因为如果主机名字符串是一个点分十进制数IP地址，那么调用memmove的源和目的内存区是相同的。

11.5　图E-12给出了这个程序。

我们使用由gethostbyname返回的h_addrtype值判定地址类型，并使用我们的sock_set_port和sock_set_addr这两个函数（见3.8节）在合适的套接字地址结构中设置端口和地址这两个字段。

本程序尽管能够工作，却存在两个局限。首先，我们必须处理所有差异，查看h_addrtype后再适当地设置sa和salen。更好的办法是由某个库函数不仅完成主机名

和服务名的查找，而且完成整个套接字地址结构的填写（例如11.6节的getaddrinfo）。其次，本程序只在支持IPv6的主机上能够编译。要在仅仅支持IPv4的主机上编译就得添加不少#ifdef伪代码，从而把代码高得复杂起来。

names/daytimetcpcli2.c

```
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", Sock_ntop((SA *) &servaddr, sizeof(servaddr)));

33        if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) == 0)
34            break;              /* success */
35        err_ret("connect error");
36        close(sockfd);
37    }
38    if (*pptr == NULL)
39        err_quit("unable to connect");

40    while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
41        recvline[n] = 0;        /* null terminate */
42        Fputs(recvline, stdout);
43    }
44    exit(0);
45 }
```

names/daytimetcpcli2.c

930

图E-11 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/daytimetcpcli3.c

```
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));
931
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 }
```

names/daytimetcpcli3.c

图E-12 图11-4中程序同时适用于IPv4和IPv6的修改版本

11.7 分配一个大缓冲区（比任何套接字地址结构都要大）并调用getsockname。它的第三个参数是一个值-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字地址结构的协议（例如IPv4和IPv6），对于能够返回可变长度套接字地址结构的协议（例如Unix域套接字，第15章）却不能保证正确工作。 932

11.8 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];
```

然后在accept返回之后改为调用getnameinfo以取代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);
```

既然这是服务器程序，我们指定NI_NUMERICHOST和NI_NUMERICSERV标志以避免查询DNS和查找/etc/services文件。

11.9 第二个服务器碰到的第一个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套接字选项。最容易的解决办法是制作udp_server函数的一个副本，把它命名为udp_server_reuseaddr，由它设置这个套接字选项，再从服务器程序调用这个新函数。

11.10 当客户输出“Trying 206.62.226.35...”时，gethostbyname已经返回了IP地址。客户在此之前的任何停顿是解析器用于查找主机名的时间。输出“Connected to bsdi.kohala.com”意味着connect已经返回。这两个输出行之间的任何停顿是connect用来建立连接的时间。

第12章

12.1 下面是相关的摘录片段（省掉了登录和列目录等内容）。主机freebsd上的FTP客户不论使用IPv4还是IPv6总是先尝试EPRT命令，若不工作则退回到PORT命令。

```
freebsd % ftp aix-4
Connected to aix-4.unpbook.com.
220 aix FTP server ...
...
230 Guest login ok, access restrictions apply.
ftp> debug
Debugging on (debug=1).
ftp> passive
Passive mode: off; fallback to active mode: off.
ftp> dir
---> EPRT |1|192.168.42.1|50484|
500 'EPRT |1|192.168.42.1|50484|': command not understood.
disabling epsv4 for this connection
---> PORT 192,168,42,1,197,52
200 PORT command successful.
---> LIST
150 Opening ASCII mode data connection for /bin/ls.
...
freebsd % ftp ftp.kame.net
Trying 2001:200:0:4819:203:47ff:fea5:3085...
Connected to orange.kame.com.
220 orange.kame.com FTP server ...
...
230 Guest login ok, access restrictions apply.
```

933

```
ftp> debug
Debugging on (debug=1).
ftp> passive
Passive mode: off; fallback to active mode: off.
ftp> dir
---> EPRT |2|3ffe:b80:3:9ad1::2|50480|
200 EPRT command successful.
---> LIST
150 Opening ASCII mode data connection for '/bin/ls'.
```

第13章

13.1　daemon_init中关闭所有描述符的close调用也将关闭由tcp_listen建立的监听TCP套接字。既然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出错消息都能写到某个终端。所有出错消息都应该使用syslog登记，即使诸如命令行参数无效之类的启动出错消息也不例外。

13.2　TCP版本的echo、discard和chargen服务器由inetd派生出来之后作为子进程运行，因为它们需要运行到客户终止连接为止。另外2个TCP服务器time和daytime并不需要inetd派生子进程，因为它们的服务极易实现（即取得当前时间和日期，把它格式化后写出，再关闭连接），于是由inetd直接处理。所有5个UDP服务的处理都不需要inetd派生子进程，因为每个服务对于引发它的任一客户数据报所作的响应只是最多产生一个数据报。因此这5个服务也由inetd直接处理。

13.3　这是一个众所周知的拒绝服务型攻击（[CERT 1996a]）。来自端口7的第一个数据报导致chargen服务器发送回一个数据报到端口7，它被回射成发送到chargen服务器的下一个数据报，这样一直循环下去。FreeBSD上实现的解决办法是拒绝源端口和目的端口都是内部服务的外来数据报。另一个常用的解决办法是在每个主机上通过inetd禁止这些内部服务，或者在一个组织机构接入因特网的路由器上这么做。

13.4　客户的IP地址和端口取自由accept填写的套接字地址结构。

inetd对UDP套接字无能为力的原因是读入数据报的recvfrom是由通过exec激活的真正服务器而不是inetd本身执行的。

inetd可以仅仅为了获取客户的IP地址和端口而指定MSG_PEEK标志（14.7节）窥读数据报，被窥读的数据报保持原地不动，留待真正的服务器读入。

934

第14章

14.1　如果未曾建立过信号处理函数，那么第一个signal调用将返回SIG_DFL，而重新设置信号处理函数的第二个signal调用只是把它设置回默认处置。

14.3　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);
}
```

14.4 数据仍然输出，因为掉出main函数末尾等同于从这个函数返回，而main函数又是由C启动例程如下调用的：

```
exit(main(argc, argv));
```

因此exit仍然被调用，标准I/O清扫例程也同样被调用。

第15章

15.1 unlink从文件系统中删除了路径名，此后客户调用connect就会失败。服务器的监听套接字不受影响，不过unlink之后没有客户能够成功connect到其上。

15.2 即使路径名仍然存在，客户也无法connect到服务器，这是因为connect成功要求当前有一个打开着的绑定了那个路径名的Unix域套接字（15.4节）。

15.3 当服务器通过调用sock_ntop显示客户的协议地址时，输出信息将是“datagram from (no pathname bound)”（数据报来自（无路径名绑定）），因为默认情况下客户的套接字上不绑定任何路径名。

解决办法之一是在udp_client和udp_connect中明确检查是否为一个Unix域套接字，若是则调用bind给它捆绑一个临时路径名。这么做把协议相关处理置于原本所属的库函数中，而不是置于我们的应用程序中。

15.4 尽管我们迫使服务器程序为它的26字节应答逐个字节调用write，客户程序中放置的sleep调用还是保证在调用read之前所有26个分节都接收到，使得单个read调用返回完整的应答。这个例子只是为了（再次）验证TCP是一个没有内在记录边界的字节流。 935

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。

服务器为每个send指定MSG_EOR标志之后逐个发送的每个字节都被认为是一个逻辑记录，客户每次调用read所返回的也将是1个字节。这里碰巧的是源自Berkeley的实现默认支持MSG_EOR标志。不过这一点没有写在正式文档中，在生产性代码中不应该使用。我们这儿使用它作为表现字节流协议和面向记录协议之差异的一个例子。从实现角度看，每个输出操作都进入一个内存缓冲区（mbuf），MSG_EOR标志由内核随mbuf从发送套接字转移到接收套接字的接收缓冲区维持在mbuf中。调用read时MSG_EOR标志仍然依附在每个mbuf上，因此通用内核read例程（它支持MSG_EOR标志，因为一些协议使用它）独自返回每个字节。如果我们改用recvmsg取代read，它将在每次返回一个字节时还在msg_flags成员中返回MSG_EOR标志。这个特性并不适用于TCP，因为发送端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首部中把这个标志传递给接收端TCP。（感谢Matt Thomas指出这个没有写在文档中的“特性”。）

15.5 图E-13给出了这个程序的实现。

debug/backlog.c

```
1 #include    "unp.h"
2 #define PORT         9999
3 #define ADDR         "127.0.0.1"
4 #define MAXBACKLOG   100
```

图E-13 确定不同的*backlog*值对应的真正已排队连接数

```
 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]);
```

图E-13　（续）

```
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  }
```

debug/backlog.c

图E-13 （续）

第16章

16.1 套接字描述符是在父子进程之间共享的，因此它的引用计数为2。要是父进程调用close，那么这只是把该引用计数由2减为1，而且既然它仍然大于0，FIN就不发送。这就是使用shutdown函数的另一个理由：即使描述符的引用计数仍然大于0，FIN也被强迫发送出去。

16.2 父进程将继续写出到已经接收FIN的套接字。它发送给服务器的第一个分节将引发RST响应。此后的那个write调用将导致内核像我们在5.12节讨论过的那样向父进程发送SIGPIPE信号。

16.3 当子进程调用getppid以向父进程发送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（也就是说它继承所有其父进程在子进程仍在运行时就终止的那些子进程）。子进程试图向init进程发送这个信号，但是没有足够的权限。然而如果这个客户程序有机会以超级用户特权运行，从而允许它向init发送信号，那么在发送该信号之前应该检测getppid的返回值。

16.4 如果去掉这两行，select就被调用。不过select调用将立即返回，因为连接建立之后套接字是可写的。这个测试加goto语句只是避免不必要地调用select。

16.5 如果服务器在accept调用返回之后立即发送数据，而当三路握手的第二个分节到达以在客户端完成连接的时候客户主机却比较忙（图2-5），那么来自服务器的数据可能在客户的connect调用返回之前到达。举例来说，SMTP服务器在未从中读之前就立即往

一个新建立的连接中写，以便给客户发送一个问候消息。

第17章

17.1　无关紧要，因为图17-2中union的前3个成员都是套接字地址结构。

第18章

18.1　sdl_nlen成员将是5，sdl_alen成员将是8。整个sockaddr_dl结构需要21个字节，在32位体系结构上则向上舍入成24个字节（TCPv2第89页）。

18.2　内核的响应绝不发送到这个套接字。SO_USELOOPBACK套接字选项确定内核是否把应答发送给发送进程，TCPv2第649～650页讨论了这一点。它的默认设置是开启，因为大多数进程需要这些应答。禁止该选项将防止内核把应答发送给发送进程。

第20章

20.1　如果你接收到许多应答，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。

20.2　FreeBSD上当信号处理函数往管道中写入一个空字节并返回之后，select返回EINTR。select再次被调用时返回管道的可读条件。

第21章

21.1　我们运行该程序得不到任何输出。为了防止进程偶尔收取并非期待的多播数据报，内核不把接收到的多播数据报递送给未曾在其上执行过任何多播操作（譬如加入某个组）的目的地套接字。这里发送的那个UDP数据报的目的地址是224.0.0.1，它是所有具备多播能力的节点都必须参加的所有主机组。该UDP数据报作为一个多播以太网帧发送，因而所有具备多播能力的节点都接收到它，因为它们都属于这个组。然而内核丢弃了接收到的数据报，因为捆绑了daytime端口的那个进程（通常就是inetd）未曾设置任何多播选项。[①]

① 本书第2版解答如下。我们运行该程序的输出如下：

```
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个给出响应的主机运行的操作系统有AIX、BSD/OS、Digital Unix和Linux。没有给出响应却具有多播能力的仅有节点是运行Solaris 2.5的主机和Cisco路由器。

这里发送的那个UDP数据报的目的地址是224.0.0.1，它是所有具备多播能力的节点都必须参加的所有主机组。该UDP数据报作为一个多播以太网帧发送，因而所有具备多播能力的节点都接收到它，因为它们都属于这个组。给出响应的主机都把接收到的数据报传递给UDP版本的daytime服务器（它通常是inetd的一部分），而不管其套接字是否已经加入所有主机组。然而Solaris的实现却要求目的地套接字必须加入所有主机组才能接收该数据报。本例子表明决不是设计来响应多播数据报的UDP程序也能接收到多播数据报。我们在第20章看到过这个daytime例子发生同样的事情：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。——译者注

939

21.2 图E-14给出了调用bind捆绑多播地址和端口0的main函数简单修改版本。

不幸的是，我们尝试运行本程序的3个系统（FreeBSD 4.8、MacOS X和Linux 2.4.7）都允许如此bind，随后发送的UDP数据报具有多播源IP地址。[①]

21.3 在支持多播的主机aix上这么做的输出如下：

```
aix % ping 224.0.0.1
PING 224.0.0.1: 56 data bytes
64 bytes from 192.168.42.2: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 192.168.42.1: icmp_seq=0 ttl=64 time=1 ms (DUP!)
^C
----224.0.0.1 PING Statistics----
1 packets transmitted, 1 packets received, +1 duplicates, 0% packet loss
round-trip min/avg/max = 0/0/0 ms
```

图1-16中右侧以太网上两个主机都给出响应。[②]

为了防护特定拒绝服务攻击，某些系统默认情况下对于广播或多播ping不给出响应。为了让主机freebsd给出响应，我们必须使用如下命令进行配置：

```
freebsd % sysctl net.inet.icmp.bmcastecho=1
```

940

21.5 值1073741824转换成浮点数并除以4294967296得到0.250。再乘以1000000得到250000，

---

① 本书第2版接着解答如下。不幸的是，我们尝试运行本程序的3个系统（BSD/OS、Digital Unix和Solaris 2.5）都允许如此bind，随后发送的UDP数据报具有多播源IP地址。给出响应的那5个系统（跟上一道习题一样）都在应答中对换源IP地址和目的IP地址，结果所有5个应答都是多播数据报！接收了这些应答的具有多播能力的客户主机倒没对它们过度反应，因为这些应答的目的端口就是最初捆绑多播地址时由客户主机的内核选定的临时端口，当时该端口没有绑定在任何套接字上。对于多播UDP数据报，ICMP不产生端口不可达消息。——译者注

② 本书第2版解答如下。在支持多播的主机solaris上这么做的输出如下：

```
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
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
```

对于所有主机组，本书第2版图1-16中顶部以太网上除没有多播能力的unixware外所有主机都给出响应（当然包括发送主机本身）。对于所有路由器组，我们期待bsdi给出响应，因为它是所在子网上的多播路由器，有一个通往MBone（B.2节）的隧道，且运行着mrouted。路由器gw也给出响应，不过它并不扮演多播路由器角色。——译者注

它以微秒为单位就是1/4秒。

最大的整数小数部分是4294967295，它除以4294967296得到0.99999999976716935634。再乘以1000000并截成整数得到999999，它就是最大的微秒数。

mcast/udpcli06.c

```
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 }
```

mcast/udpcli06.c

图E-14　捆绑多播地址的UDP客户程序main函数

第22章

22.1　我们已经知道sock_ntop使用自己的静态缓冲区存放结果。如果我们在同一个printf中作为参数调用它两次，第二次调用就会覆写第一次调用的结果。

22.2　是的，如果应答中包含0个字节的用户数据的话（也就是仅有一个hdr结构）。

22.3　由于select并不修改指定其时间限制的timeval结构，因此你必须记下第一个分组的发送时刻（它已由rtt_ts以微秒为单位返回）。当select返回套接字的可读条件时，记下当前时刻；如果需要再次调用recvmsg，那就给select计算新的超时值。

22.4　常用的技巧就如我们在22.6节所做的那样给每个接口地址创建一个套接字，然后就从请求到达的那个套接字发送相应的应答。

22.5　既不给出主机名参数也不设置AI_PASSIVE标志就调用getaddrinfo会导致它假定获取本地主机地址0::1（IPv6）或127.0.0.1（IPv4）的信息。回顾一下，我们知道在主机支持IPv6的前提下，getaddrinfo先于IPv4套接字地址结构返回IPv6套接字地址结构。如果主机同时支持这两个协议，那么udp_client中的socket调用尝试将以地址族为AF_INET6的首次尝试成功告终。

图E-15是这个程序的协议无关版本。

advio/udpserv04.c

```
1 #include    "unpifi.h"
2 void    mydg_echo(int, SA *, socklen_t);
3 int
```

图E-15　22.6节中程序的协议无关版本

```
 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     }
```

941

图E-15 （续）

```
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      }
942 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", getpid(), Sock_ntop(cli, len));
81          printf(", to %s\n", Sock_ntop(myaddr, salen));
82          Sendto(sockfd, mesg, n, 0, cli, len);
83      }
84  }
```

advio/udpserv04.c

图E-15　（续）

第24章

24.1　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。

24.2　图E-16给出了使用poll的版本。

oob/tcprecv03p.c

```
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)
```

图E-16　以poll代替select的图24-6中程序的修改版本

```
 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  }
```

oob/tcprecv03p.c

943

图E-16 （续）

第 25 章

25.1 这样的改动引入了一个错误。问题在于nqueue是在处理数组元素dg[iget]之前递减的，导致信号处理函数有可能把新的数据报从套接字读入到这个数组元素。

第 26 章

26.1 使用fork的例子将会使用101个描述符，其中1个是监听套接字描述符，其余100个是已连接套接字描述符。不过101个进程（1个父进程，100个子进程）的每一个只打开着一个描述符（忽略任何其他描述符，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个描述符，每个线程（包括主线程）处理其中一个。

26.2 TCP连接终止序列的最后2个分节（服务器的FIN和客户对于该FIN的ACK）将不会交换。这使得连接的客户端一直处于FIN_WAIT_2状态（图2-4）。源自Berkeley的实现在客户端保持这种状态超过11分钟时就会超时断连（TCPv2第825～827页）。服务器还可能（最终）耗尽描述符。

26.3　这个消息应该在主线程已从套接字读入EOF而另一个线程却还在运行时显示。这么做的一个简单方法是声明名为done且初始化为0的另一个外部变量。线程copyto在返回
944 之前把该变量设置成1。主线程检查该变量，如果其值为0就显示这个出错消息。既然设置该变量的线程只有一个，因而没有任何同步的必要。

第27章

27.1　没有变化，所有系统都是邻居，因此严格的源路径等同于宽松的源路径。

27.2　我们会在缓冲区的末尾放一个EOL（值为0的单个字节）。

27.3　ping创建的是一个原始套接字（第28章），因此能够获取使用recvfrom读入的每个数据报的完整IP首部，包括任何IP选项在内。

27.4　因为rlogind是由inetd激活的（13.5节），而描述符0正是通达客户的套接字。

27.5　问题在于setsockopt的第五个参数以指向长度的指针取代长度本身。这个缺陷可能是在开始使用ANSI C原型时修正的。

这个缺陷结果是无害的，因为正如我们所提，禁止IP_OPTIONS套接字选项既可以指定一个空指针作为第四个参数，也可以使用值为0的第五个（长度）参数（TCPv2第269页）。

第28章

28.1　IPv6首部中的版本字段和下一个首部字段是无法得到的。净荷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，但是如果需要特大净荷选项，那么真正的选项本身应用进程是得不到的。分片首部应用进程也得不到。

28.2　最终客户的套接字接收缓冲区会被填满，导致作为服务器的icmpd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icmpd在任何套接字上都停止处理新的数据。最容易的解决办法是让icmpd把它跟客户的Unix域连接的本地端设置成非阻塞式。icmpd然后必须改为调用write以取代它的包裹函数Write，并仅仅忽略EWOULDBLOCK错误。

28.3　源自Berkeley的内核默认允许在原始套接字上的广播（TCPv2第1057页）。SO_BROADCAST套接字选项只有UDP套接字才需指定。

28.4　我们的程序既不检查多播地址，也不设置IP_MULTICAST_IF套接字选项，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套接字选项，因此它默认成1，这是合理的。 945

第29章

29.1　这个标志表示跳转缓冲区已由sigsetjmp设置（图29-10）。尽管这个标志看似多余，但是在信号处理函数建立之后和调用sigsetjmp之前，SIGALRM信号被递交的机会还是存在的。即使程序本身不会导致产生该信号，它也可能以其他方式产生，譬如使用kill命令。

第30章

30.1　父进程保持监听套接字打开着是为以后需要fork额外的子进程而做准备（这是对于现行代码的一种改进）。

30.2 是的，数据报套接字能够取代字节流套接字用于传递描述符。在使用数据报套接字情况下，当某个子进程过早终止时，父进程在流管道的拥有端接收不到EOF，不过父进程可以使用SIGCHLD信号达到这个目的。这种能够使用SIGCHLD的情形与28.7节中的icmpd守护进程情形相比的一个差别是：后者的客户和服务器之间不存在父子关系，因此流管道上的EOF是服务器检测某个客户已消失的唯一办法。

第 31 章

31.1 我们假定流关闭时协议的默认处理就是顺序释放，这对TCP来说是正确的。 946

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。这里不给出RFC的URL。

标记为“Internet Draft（因特网草案）”的条目是因特网工程任务攻坚组（Internet Engineering Task Force，IETF）正在进展中的著作。这些草案在出版后6个月就完成使命。因此它们或者在本书付印之后就有新版本出现，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。这里给出每个因特网草案的URL中的文件名部分，因为文件名含有版本号。

要是能够找到本参考文献所引用的论文或报告的电子文件，其URL就同时列出。需留意的是这些URL可能随时间而变动，因此读者应该经常访问本书的主页（http://www.unpbook.com/），检查本书的最新勘误表。http://citeseer.nj.nec.com/cs是一个相当棒的论文在线数据库。通过这个数据库输入一篇论文或报告的标题不仅可以找到引用它的其他论文，还能给出已知的在线版本。

Albitz, P. and Liu, C. 2001, *DNS and Bind, Fourth Edition*. O' Reilly & Associates, Sebastopol, CA.

Allman, M., Floyd, S., and Partridge, C. 2002. “Increasing TCP' s Initial Window,” RFC 3390.

Allman, M., Ostermann, S., and Metz, C.W. 1998. “FTP Extensions for IPv6 and NATs,” RFC 2428.

Allman, M., Paxson, V., and Stevens, W.R. 1999. “TCP Congestion Control,” RFC 2581.

Almquist, P. 1992. “Type of Service in the Internet Protocol Suite,” RFC 1349 (obsoleted by RFC 2474).

如何使用IPv4首部的服务类型字段。已被RFC 2474［Nicholset al. 1998］和RFC 3168［Ramakrishnan, Floyd, and Black 2001］淘汰。

Baker, F. 1995. “Requirements for IP Version 4 Routers,” RFC 1812.

Borman, D.A. 1997a. “Re: Frequency of RST Terminated Connections,” end2end-interest mailing list (http://www.unpbook.com/borman.97jan30.txt).

Borman, D.A. 1997b. “Re: SYN/RST cookies,” tcp-impl mailing list (http://www.unpbook.com/borman.97jun06.txt).

Borman, D.A. Deering, S.E., and Hinden,R. 1999. “IPv6 Jumbograms,” RFC 2675.

Braden, R.T., 1989. “Requirements for Internet Hosts-Communication Layers,” RFC 1122.

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. 1992. “TIME-WAIT Assassination Hazards in TCP”, RFC 1337.

Braden, R.T. Borman, D.A., and Partridge, C. 1988. “Computing the Internet Checksum,” RFC 1071.

Bradner, S. 1996. “The Internet Standards Process-Revision3,” RFC 2026.

Bush, R. 2001. “Delegation of IP6.ARPA,” RFC 3152.

Butenhof, D.R. 1997. *Programming with POSIX Threads*. Addison-Wesley, Reading, MA.

Cain, B., Deering, S.E., Kouvelas, I., Fenner, B., and Thyagarajan, A. 2002. "Internet Group Managenent Protocol, Bersion 3," RFC 3376.

Carpenter, B. and Moore, K. 2001. "Connection of IPv6 Domains via IPv4 Clouds," RFC 3056.

CERT. 1996a. "UDP Port Denial-of-Service Attack," Advisory CA-96.01, Computer Emergency Response Team, Pittsburgh, PA.

CERT. 1996b. "TCP SYN Flooding and IP Spoofing Attacks," Advisory CA-96.21, Computer Emergency Response Team, Pittsburgh, PA.

Cheswick, W.R., Bellovin, S.M. and Rubin, A.D. 2003. *Firewalls and Internet Security: Repelling the Wily Hacker, Second Edition*. Addison-Wesley, Reading, MA.

Conta, A., and Deering, S.E. 1998. "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," RFC 2463.

Conta, A., and Deering, S.E. 2001. "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," draft-ietf-ipngwg-icmp-v3-02.txt (Internet Draft).

这是对［Conta and Deering 1998］的修订版本，期望最终取代它。

Crawford, M. 1999a. "Transmission of IPv6 Packets over Ethernet Networks," RFC 2464.

Crawford, M. 1998b. "Transmission of IPv6 Packets over FDDI Networks," RFC 2467.

Crawford, M., Narten, T., and Thomas, S. 1998. "Transmission of IPv6 Packets over Token Ring Networks," RFC 2470.

Deering, S.E. 1989. "Host Extensions for IP Multicasting," RFC 1112.

Deering, S.E. and Hinden, R. 1998. "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460.

Dewar, R.B.K., and Smosna, M. 1990. Microprocessors: A Programmers View. McGraw-Hill, NY.

Draves, R. 2003. "Default Address Selection for Internet Protocol version 6 (IPv6)," RFC 3484.

Eriksson, H. 1994. "MBONE: The Multicast Backbone," *Communications of the ACM*, vol.37, no.8, pp.54-60.

Fenner, W.C. 1997. Private Communication.

Fink, R. and Hinden, R. 2003. "6bone (IPv6 Testing Address Allocation) Phaseout," draft-fink-6bone-phaseout-04.txt (Internet Draft).

Fuller, V., Li, T., Yu, J.Y., and Varadhan, K. 1993. "Classless Inter-Domain Routing (CIDR):an Address Assignment and Aggregation Strategy," RFC 1519.

Garfinkel, S.L., Schwartz, A., and Spafford, E.H. 2003. *Practical UNIX & Internet Security, 3rd Edition*. O'Reilly & Associates, Sebastopol, CA.

Gettys, J. and Nielsen, H.F. 1998. *SMUX Protocol Specification* (http://www.w3.org/TR/WD- mux).

Gierth, A. 1996. *Private Communication*.

Gilligan, R.E. and Nordmark, E. 2000. "Transition Mechanisms for IPv6 Hosts & Routers," RFC 2893.

Gilligan, R.E., Thomson, S., Bound, J., McCann, J., and Stevens, W.R. 2003. "Basic Socket Interface Extensions for IPv6," RFC 3493.

Gilligan, R.E., Thomson, S., Bound, J., and Stevens, W.R. 1997. "Basic Socket Interface Extensions for IPv6," RFC 2133(obsoleted by RFC 2553).

Gilligan, R.E., Thomson, S., Bound, J., and Stevens, W.R. 1999. "Basic Socket Interface Extensions for

IPv6," RFC 2553(obsoleted by RFC 3493).

Haberman, B. 2002. "Allocation Guidelines for IPv6 Multicast Addresses," RFC 3307.

Haberman, B. and Thaler, D. 2002. "Unicast-Prefix-based IPv6 Multicast Addresses," RFC 3306.

Handley, M. and Jacobson, V. 1998. "SDP: Session Description Protocol," RFC 2327.

Handley, M., Perkins, C., and Whelan, E. 2000. "Session Announcement Protocol," RFC 2974.

Harkins, D. and Carrel, D. 1998. "The Internet Key Exchange (IKE)," RFC 2409.

Hinden, R. and Deering, S.E. 2003. "Internet Protocol Version 6 (IPv6) Addressing Architecture," RFC 3513.

Hinden, R., Deering, S.E., and Nordmark, E. 2003. "IP6 Global Unicast Address Format," RFC 3587.

Hinden, R., Fink, R., and Postel, J.B. 1998. "IPv6 Testing Address Allocation," RFC 2471.

Holbrook, H. and Cheriton, D. 1999. "IP multicast channels: EXPRESS support for large-scale single-source applications," *Computer Communication Review*, vol.29, no.4, pp.65-78.

Huitema, C. 2001. "An Anycast Prefix for 6to4 Relay Routers," RFC 3068.

IANA. 2003. *Protocol/Number Assignments Directory* (http://www.iana.org/numbers.htm).

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, NJ.

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. 1997. *Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority*. Institute of Electrical and Electronics Engineers, Piscataway, NJ.(http://standards.ieee.org/regauth/oui/tutorials/EUI64.html).

Jacobson, V. 1988. "Congestion Avoidance and Control," *Computer Communication Review*, vol.18, no.4, pp.314-329(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., Braden, R.T., and Borman, D.A. 1992. "TCP Extensions for High Performance," RFC 1323.

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 (obsoleted by RFC 1323).

Josey, A., ed. 1997. *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification*. Prentice Hall, Upper Saddle River, NJ.

Josey, A., ed. 2002. *The Single UNIX Specification-The Authorized Guide to Version 3*. The Open Group, Berkshire, UK.

Joy, W.N. 1994. *Private Communication*.

Karn, P., and Partridge, C. 1991. "Improving Round-Trip Time Estimates in Reliable Transport Protocols," *ACM Transactions on Computer Systems*, vol.9, no.4, pp.364-373.

Katz, D. 1993. "Transmission of IP and ARP over FDDI Network," RFC 1390.

Katz, D. 1997. "IP Router Alert Option," RFC 2113.

Kent, S.T. 1991. "U.S. Department of Defense Security Options for the Internet Protocol," RFC 1108.

Kent, S.T. 2003a. "IP Authentication Header," draft-ietf-ipsec-rfc2402bis-04.txt (Internet Draft).

Kent, S.T. 2003b. "IP Encapsulating Security Payload (ESP)," draft-ietf-ipsec-esp-v3-06.txt (Internet Draft).

Kent, S.T. and Atkinson, R.J. 1998a. "Security Architecture for the Internet Protocol," RFC 2401.

Kent, S.T. and Atkinson, R.J. 1998b. "IP Authentication Header," RFC 2402.

本书写至此处时IETF IPsec工作组正在更新本RFC(见［Kent 2003a］)。

Kent, S.T. and Atkinson, R.J. 1998c. "IP Encapsulating Security Payload (ESP)," RFC 2406.

本书写至此处时IETF IPsec工作组正在更新本RFC(见［Kent 2003b］)。

Kernighan, B.W. and Pike, R. 1984. *The UNIX Programming Environment*. Prentice Hall, Englewood Cliffs, NJ.

Kernighan, B.W. and Ritchie, D.M. 1988. *The C Programming Language, Second Edition*. Prentice Hall, Englewood Cliffs, NJ.

Lanciani, D. 1996. "Re: sockets: AF_INET vs. PF_INET," Message-ID: <3561@news. IPSWITCH.COM>, USENET comp.protocols.tcp-ip Newsgroup (http://www.unpbook.com/lanciani.96apr10.txt).

Maslen, T.M. 1997. "Re: gethostbyXXXX() and Threads," Message-ID: <maslen.862463530@ shellx>, USENET comp.programming.threads Newsgroup (http://www.unpbook.com/maslen.97may01.txt).

McCann, J., Deering, S.E., and Mogul, J.C. 1996. "Path MTU Discovery for IP version 6," RFC 1981.

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*, San Diego. CA, pp.259-269.

McDonald, D.L., Metz, C.W., and Phan, B.G. 1998. "PF_KEY Key Management API, Version 2," RFC 2367.

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, MA.

Meyer, D. 1998. "Administratively Scoped IP Multicast," RFC 2365.

Mills, D.L. 1992. "Network Time Protocol(Version 3) Specification, Implementation," RFC 1305.

Mills, D.L. 1996. "Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI," RFC 2030.

Mogul, J.C., and Deering, S.E. 1990. "Path MTU Discovery," RFC 1191.

Mogul, J.C., and Postel, J.B. 1985. "Internet Standard Subnetting Procedure," RFC 950.

Narten, T. and Draves, R. 2001. "Privacy Extensions for Stateless Address Autoconfiguration in IPv6," RFC 3041.

Nemeth, E. 1997. *Private Communication*.

Nichols, K., Blake, S., Baker, F., and Black, D. 1998. "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," RFC 2474.

Nordmark, E. 2000. "Stateless IP/ICMP Translation Algorithm (SITT)," RFC 2765.

Ong, L., Rytina, I. Garcia, M., Schwarzbauer, H., Coene, L., Lin, H., Juhasz, I., Holdrege, M. and Sharp, C. 1999. "Framework Architecture for Signaling Transport," RFC 2719.

Ong, L., and Yoakum, J. 2002. "An Introduction to the Stream Control Transmission Protocol (SCTP)," RFC 3286.

The Open Group. 1997. *CAE Specification, Networking Services(XNS), Issue 5*. The Open Group, Berkshire, UK.

这是Unix 98中套接字和XTI这两个API的规范，现已被The Single Unix Specification, Version 3超越。本手册还在其附录中描述了XTI在NetBIOS、OSI协议族、SNA及Netware的IPX和SPX协议中的使用。另有三个附录介绍套接字和XTI这两个API在ATM中的使用。

Partridge, C. and Jackson, A. 1999. "IPv6 Router Alert Option," RFC 2711.

Partridge, C., Mendez, T., and Milliken, W. 1993. "Host Anycasting Service," RFC 1546.

Partridge, C. and Pink, S. 1993. "A Faster UDP," *IEEE/ACM Transactions on Networking*, vol.1, no.4, pp.429-440.

Paxson, V. 1996. "End-to-End Routing Behavior in the Internet," *Computer Communication Review*, vol.26, no.4, pp. 25-38 (ftp://ftp.ee.lbl.gov/papers/routing.SIGCOMM.ps.Z).

Paxson, V. and Allman, M. 2000. "Computing TCP' s Retransmission Timer," RFC 2988.

Plauger, P.J. 1992. *The Standard C Library*. Prentice Hall, Englewood Cliffs, NJ.

Postel, J.B. 1980. "User Datagram Protocol," RFC 768.

Postel, J.B. 1981a. "Internet Protocol," RFC 791.

Postel, J.B. 1981b. "Internet Control Message Protocol," RC 792.

Postel, J.B. 1981c. "Transmission Control Protocol," RFC 793.

Pusateri, T. 1993. "IP Multicast Over Token-Ring Local Area Networks," RFC 1469.

Rago.S.A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, MA.

Rajahalme, J., Conta, A., Carpenter, B., and Deering, S.E. 2003. "IPv6 Flow Label Specification," draft-ietf-ipv6-flow-label-07.txt (Internet Draft).

Ramakrishnan, K., Floyd, S., and Black, D. 2001. "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168.

Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G. J., and Lear, E. 1996. "Address Allocation for Private Intermets," RFC 1918.

Reynolds, J.K. 2002. "Assigned Numbers: RFC 1700 is Replaced by an On-line Database," RFC 3232.

本RFC指的数据库就是［IANA 2003］。

Reynolds, J.K., and Postel, J.B. 1994. "Assigned Numbers," RFC 1700 (obsoleted by RFC 3232)。

本RFC是"Assigned Numbers"系列RFC中的最后一个。由于其信息经常变化，因此整个目录都被放在因特网上。可参阅[Reynolds 2002]以获取更多解释，或是参考[IANA 2003]来了解数据库本身。

Ritchie, D.M. 1984. "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, vol.63, no, 8, pp.1897-1910.

Salus, P.H. 1994. *A Quarter Century of Unix*. Addison-Wesley, Reading, MA.

Salus, P.H. 1995. *Casting the Net: From ARPANET to Internet and Beyond*. Addison-Wesley, Reading, MA.

Schimmel, C. 1994. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley, Reading, MA.

Spero, S. 1996. *Session Control Protocol (SCP)* (http://www.w3.org/Protocols/HTTP-NG/http-ng-scp.html).

Srinivasan, R. 1995. "XDR: External Data Representation Standard," RFC 1832.

Stevens, W.R. 1992. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA.

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, MA.

对网际网协议的完整介绍。本书称之为TCPv1。

Stevens, W.R. 1996. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NTTP, and the UNIX Domain Protocols*. Addison-Wesley, Reading, MA.

本书称之为TCPv3。

Stevens, W.R. and Thomas, M. 1998. "Advanced Sockets API for IPv6," RFC 2292 (obsoleted by RFC 3542).

Stevens, W.R., Thomas, M., Nordmark, E., and Jinmei, T. 2003. "Advanced Sockets Application Program Interface (API) for IPv6," RFC 3542.

Stewart, R.R., Bestler, C., Jim, J., Ganguly, S., Shah, H., and Kashyap, V. 2003a. "Stream Control Transmission Protocol (SCTP) Remote Direct Memory Access (RDMA) Direct Data Placement (DDP) Adaptation," draft-stewart-rddp-sctp-02.txt (Internet Draft).

Stewart, R.R., Ramalho, M., Xie, Q., Tuexen, M., Rytina, I., Belinchon, M., and Conrad, P. 2003b. "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration," draft-ietf-tsvwg-addip-sctp-07.txt (Internet Draft).

Stewart, R.R. and Xie, Q. 2001. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison-Wesley, Reading, MA.

Stewart, R.R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and Paxson, V. 2000. "Stream Control Transmission Protocol," RFC 2960.

Stone, J., Stewart, R.R., and Otis, D. 2002. "Stream Control Transmission Protocol (SCTP) Checksum Change," RFC 3309.

Tanenbaum, A.S. 1987. *Operating Systems Design and Implementation*. Prentice Hall, Englewood Cliffs, NJ.

Thomson, S. and Huitema, C. 1995. "DNS Extensions to Support IP version 6," RFC 1886.

Torek, C. 1994. "Re: Delay in re-using TCP/IP port," Message-ID: <199501010028. QAA16863@elf.bsdi.com>, USENET comp.unix.wizards Newsgroup (http://www.unpbook.com/ torek.94dec31.txt).

Touch, J. 1997. "TCP Control Block Interdependence," RFC2140.

Unix International. 1991. *Data Link Provider Interface Specification,* Unix International, Parsippany, NJ *Revision 2.0.0*. (http://www.unpbook.com/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*. Unix International, Parsippany, NJ , *Revision 2.0.0* (http://www.unpbook.com/npi.2.0.0.ps).

Unix International. 1992b. *Transport Provider Interface Specification*. Unix International, Parsippany, NJ , *Revision 1.5* (http://www.unpbook.com/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。

Vixie, P.A. 1996. *Private Communication*.

Wright, G.R. and Stevens, W.R. 1995. *TCP/IP Illustrated, Volume 2: The Implementation*. AddisonWesley, Reading, MA.

网际网协议在4.4BSD-Lite操作系统上的实现。本书称之为TCPv2。

索　引

网络编程是一个密布首字母缩写词的领域。我们不提供一个单独的词汇表（其中大多数条目将是首字母缩写词），不过本索引也可用作本书所用所有首字母缩写词的词汇表。可以首字母缩写的词条其主条目编排在缩写词之下。举例来说，所有对Internet Control Message Protocol（网际网控制消息协议）的引用出现在ICMP之下，在完整词条“Internet Control Message Protocol”之下的条目只是引用回ICMP之下的主条目。

每个C函数的“definition of（定义）”条目给出该函数带方框的函数原型即基本描述的所在页。每个结构的“definition of（定义）”条目给出该结构的基本定义的所在页。那些在本书中有源代码实现的函数还有“source code（源代码）”条目。

索引中的页码为英文原书页码，与书中页边标注的页码一致。

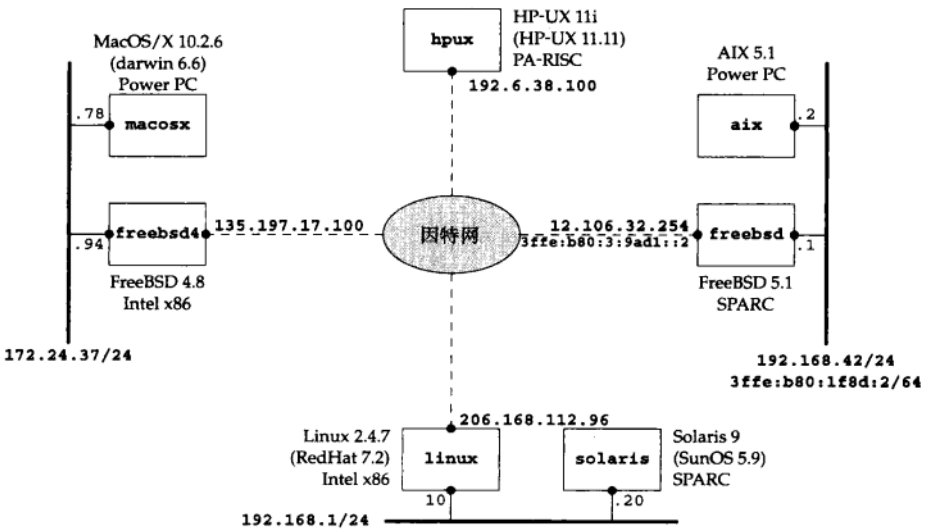
函数和宏定义索引表

（下面所列页码均指页边栏中标注的页码，加粗的页码指示源代码实现所在之处）

（续）

结构定义索引表

（下面所列页码均指页边栏中标注的页码）



用于正文中大多数示例的主机和网络

读者积分赠书卡

手机号码：_____（此为会员编号）

姓　　名：_____　性别：□男 □女　出生年月：____年__月

通信地址：_____

邮政编码：_____

电子邮件：_____

您购买的图书是：22840 /《UNIX网络编程　卷1：套接字联网API（第3版）》（129.00元）

您获得的会员积分是：12.9 分

欢迎参加“**有奖DEBUG**”活动。提交本书勘误，每确认一处即可获赠积分5分。详情见图灵网站。

请沿虚线剪下此页，寄回图灵公司，即可成为图灵读者俱乐部的一员（复印无效）。**积分累计，可获赠书**（赠书清单见图灵网站）。

邮政编码：100107

通信地址：北京市朝阳区北苑路 13 号院 1 号楼 C603

北京图灵文化发展有限公司　图灵读者俱乐部

[General Information]
书名=UNIX网络编程　卷1　套接字联网API
作者=（美）W.RichardStevens著
页数=809
出版社=人民邮电出版社
出版日期=2010.06
SS号=12586890
DX号=000006907086
URL=http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000006907086&d=055D0A2638ECBDEFF3A46D127866DC5D