

UNP

第一章 简介

1.1 概述

- 客户和服务器通常是用户进程，而TCP和IP协议通常是内核中协议栈的一部分

1.2 一个简单的时间获取客户程序

- 编译时 gcc daytimetccli.c -o daytimetccli -lunp
- autoconf 和automake
- Makefile文件
- Linux下使用automake、autoconf生成configure文件
- 创建TCP套接字 socket
- 指定服务器的IP地址和端口
 - bzero函数
 - htonl函数
 - 13 时间获取服务器的端口
 - inet_nton函数
- 建立与服务器的连接
 - connect
 - 通用套接字地址结构 struct sockaddr
- 读入并输出服务器的应答
 - read函数
 - TCP是一个没有记录边界的字节流协议
- 终止程序

1.3 协议无关性

- 修改部分代码即可
- 最好编写协议无关的程序

1.4 错误处理: 包裹函数

- 每个包裹函数完成实际的函数调用，检查返回值，并在发生错误时终止进程
- 约定包裹函数名是实际函数名的大写形式
- 只要一个Unix函数中有错误发生，全局变量errno就被设置为一个指明该错误类型的正值

1.5 一个简单的时间获取服务器程序

- 创建TCP套接字
- 把服务器众所周知端口绑到套接字

- bind
- 把套接字转换成监听套接字
 - listen
- 接受客户连接，发送应答
 - accept
 - 握手完毕后accept返回，返回值是一个已连接描述符
 - write函数把结果字符串写回给客户
- 终止连接
 - 调用close关闭与客户的连接，引发正常的TCP连接终止序列
 - 并发服务器：可使用Unix的fork函数实现，或线程技术
 - Unix守护进程：能在后台运行且不跟任何终端关联的进程

1.6 客户/服务器程序

- 时间获取客户程序的不同版本
- 时间获取服务器程序的不同版本
- 回射客户程序的不同版本
- 回射服务器程序的不同版本

1.7 OSI模型

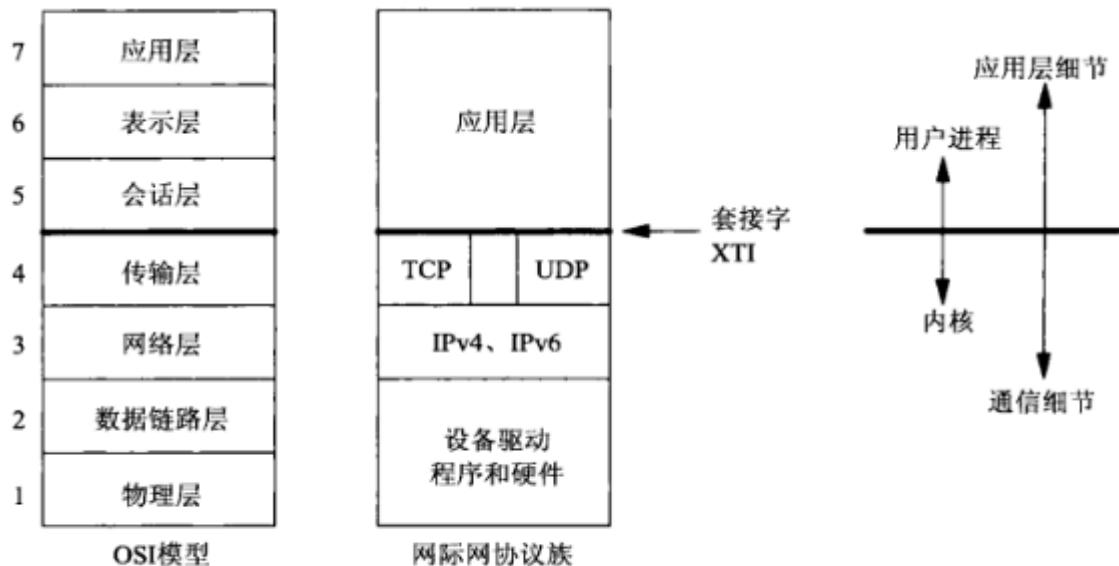


图1-14 OSI模型和网际协议族中的各层

- 套接字编程接口是从顶上三层(网际协议的应用层)进入传输层的接口
- 顶上三层通常构成用户进程，底下四层通常作为操作系统内核的一部分提供
- Linux这种流行的可免费获得的Unix实现并不适合归属自Berkeley系列，因为它的网络支持代码和套接字API都是从头开始开发的

1.9 测试用网络及主机

- netstat -i 提供网络接口的信息 -n 标志以输出数值地址

- 环回(loopback)接口称为lo
- 以太网接口称为eth0
- netstat -r 展示路由表
- ifconfig enp4s0 执行ifconfig可获得每个接口的详细信息
- 针对本地接口的广播地址执行ping 命令可获得本地网络中众多主机的IP地址 ping -b 广播地址

1.11 64位体系结构

- 在每个进程内部可以由此使用更长的编址长度，从而可以寻址很大的内存空间
- 套接字API对套接字地址结构的长度使用socklen_t数据类型

第二章 传输层: TCP UDP SCTP

- 绕过传输层直接使用IP协议称为原始套接字
- SCTP 流控制传输协议，一个可靠的传输协议

2.2 总图

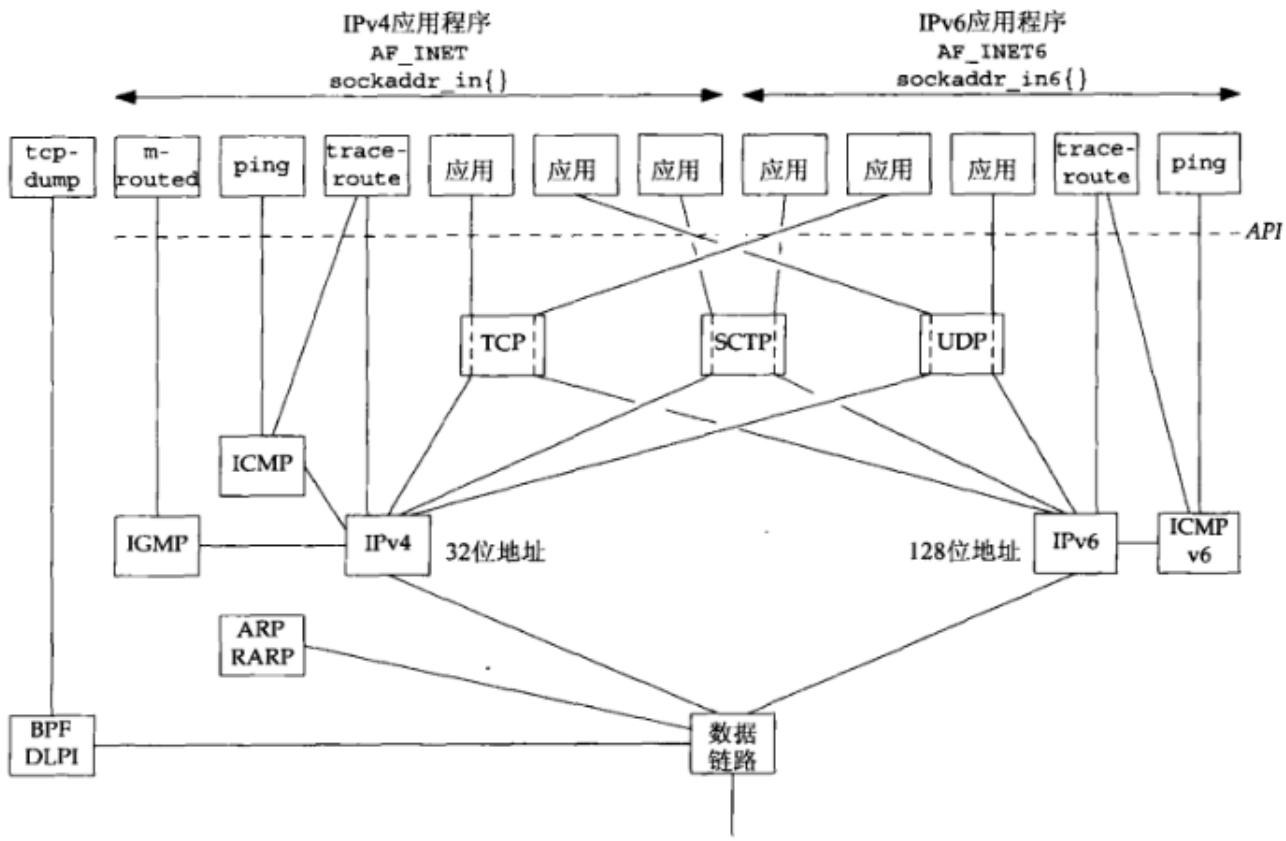


图2-1 TCP/IP协议概况

- tcpdump 使用数据链路提供者接口与数据链路进行通信
- traceroute程序使用两种套接字: IP套接字用于访问IP, ICMP套接字用于访问ICMP
- TCP套接字是一种流套接字,没有任何记录边界
- UDP套接字是一种数据报套接字

2.3 UDP

- 每个UDP数据报有一个长度，如果一个数据报正确到达目的地，那么该数据报的长度将随数据一道传递给接收端应用进程

2.4 TCP

2.5 SCTP

- SCTP在客户和服务器之间提供关联，并像TCP那样给应用提供可靠性、排序、流量控制以及全双工的数据传送，一个关联指代两个系统之间的一次通信，它可能因为SCTP支持多宿而涉及不止两个地址
- 面向消息的。提供各个记录的按序递送服务
- SCTP能够在所连接的端点之间提供多个流
- 多宿特性，使得单个SCTP端点能够支持多个IP地址

2.6 TCP连接的建立和终止

- 三路握手

(1) 服务器必须准备好接受外来的连接。这通常通过调用socket、bind和listen这3个函数来完成，我们称之为被动打开（passive open）。

(2) 客户通过调用connect发起主动打开（active open）。这导致客户TCP发送一个SYN（同步）分节，它告诉服务器客户将在（待建立的）连接中发送的数据的初始序列号。通常SYN分节不携带数据，其所在IP数据报只含有一个IP首部、一个TCP首部及可能有的TCP选项（我们稍后讲解）。

(3) 服务器必须确认（ACK）客户的SYN，同时自己也得发送一个SYN分节，它含有服务器将在同一连接中发送的数据的初始序列号。服务器在单个分节中发送SYN和对客户SYN的ACK（确认）。

(4) 客户必须确认服务器的SYN。

- 这种交换至少需要3个分组，因此称之为TCP的三路握手（three-way handshake）。图2-2展示了所交换的3个分节。

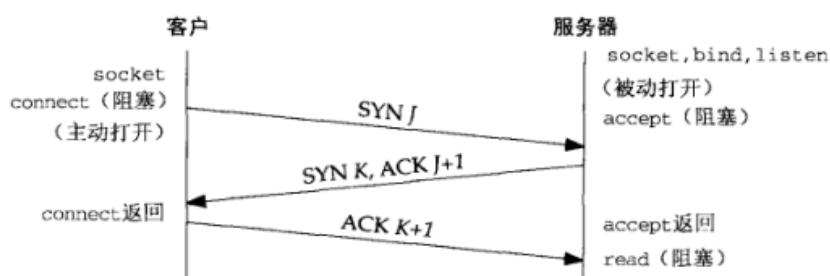


图2-2 TCP的三路握手

- TCP选项
 - MSS选项，最大分节大小
 - 窗口规模选项
 - 时间戳选项
- TCP连接终止
 - 建立一个连接需要三个分节，终止一个连接需要四个分节

TCP建立一个连接需3个分节，终止一个连接则需4个分节。

(1) 某个应用进程首先调用close，我们称该端执行主动关闭（active close）。该端的TCP于是发送一个FIN分节，表示数据发送完毕。

(2) 接收到这个FIN的对端执行被动关闭（passive close）。这个FIN由TCP确认。它的接收也作为一个文件结束符（end-of-file）传递给接收端应用进程（放在已排队等候该应用进程接收的任何其他数据之后），因为FIN的接收意味着接收端应用进程在相应连接上再无额外数据可接收。

(3) 一段时间后，接收到这个文件结束符的应用进程将调用close关闭它的套接字。这导致它的TCP也发送一个FIN。

(4) 接收这个最终FIN的原发送端TCP（即执行主动关闭的那一端）确认这个FIN。

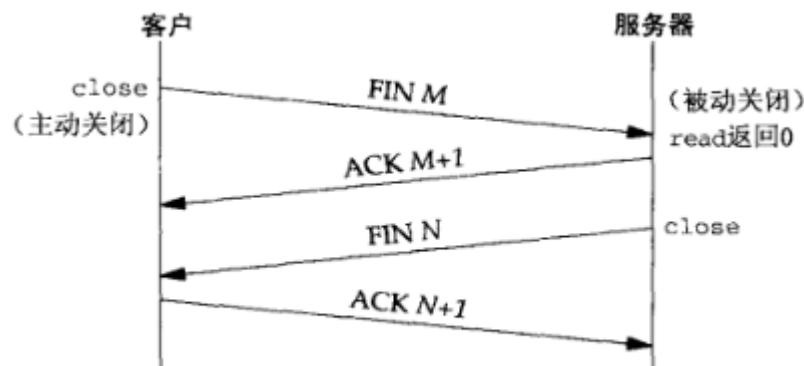


图2-3 TCP连接关闭时的分组交换

- TCP状态转换图

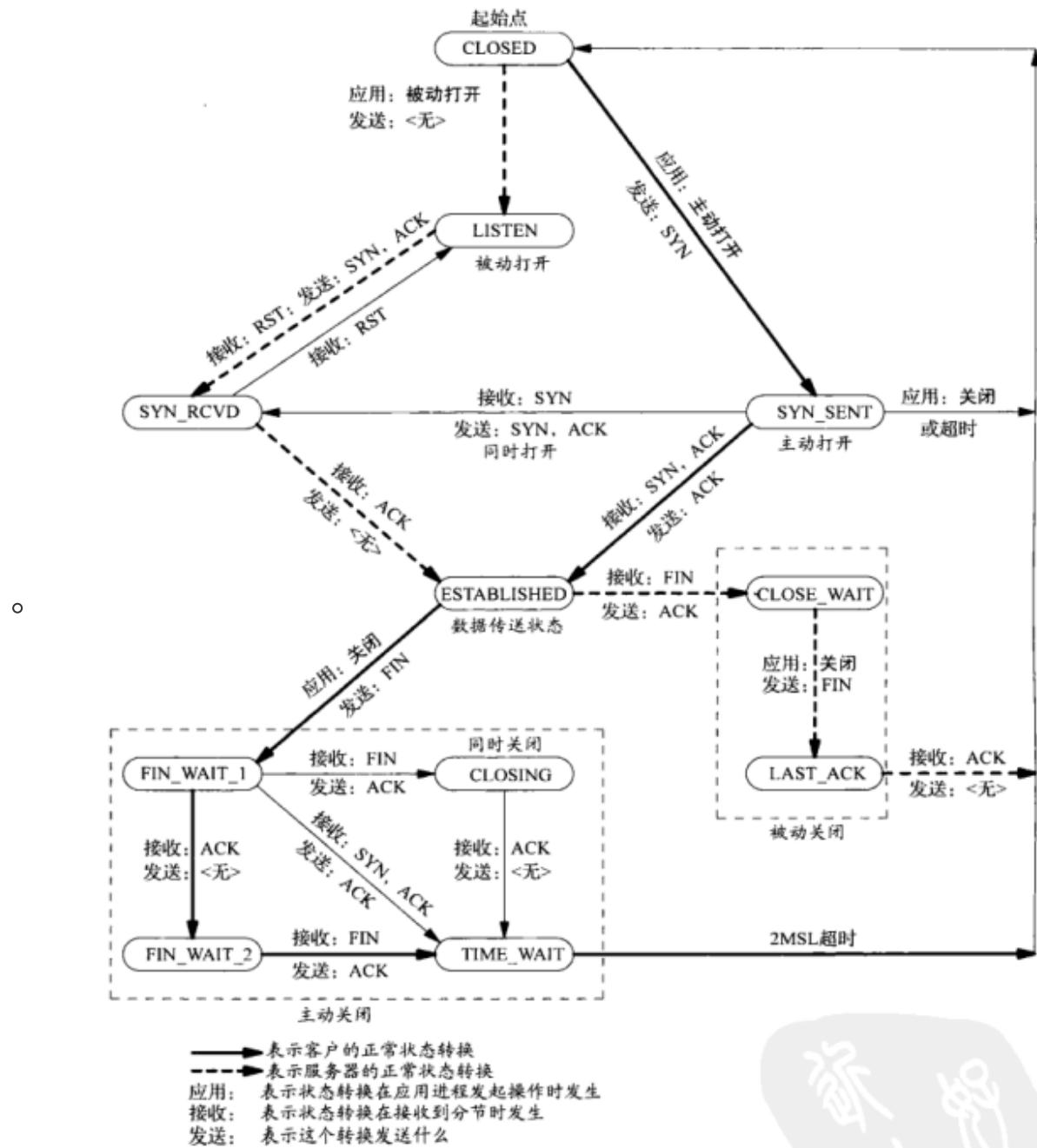


图2-4 TCP状态转换图

- 观察分组

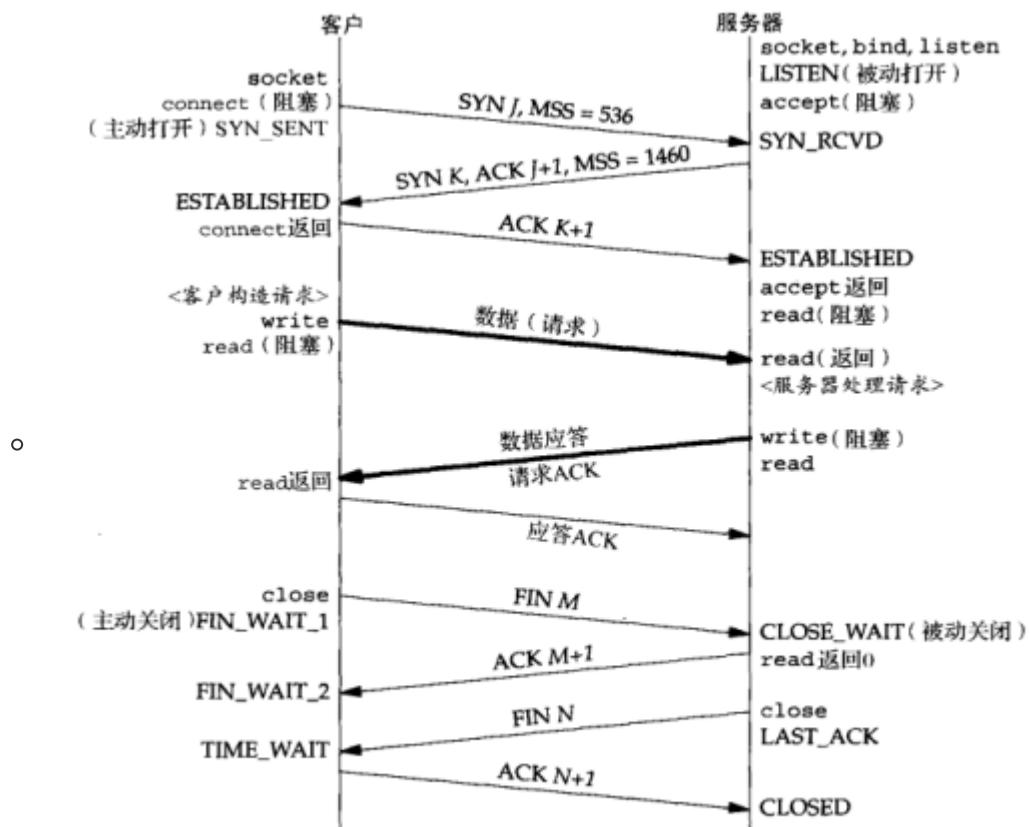


图2-5 TCP连接的分组交换

- UDP避免了TCP连接建立和终止所需的开销

2.7 TIME_WAIT状态

- 最长分节生命期MSL
- TIME_WAIT状态存在的两个理由
 - 可靠地实现TCP全双工连接的终止
 - 允许老的重复分节在网络中消逝

2.8 SCTP关联的建立和终止

- 四路握手

建立一个SCTP关联的时候会发生下述情形（类似于TCP）。

(1) 服务器必须准备好接受外来的关联。这通常通过调用socket、bind和listen这3个函数来完成，称为被动打开。

(2) 客户通过调用connect或者发送一个隐式打开该关联的消息进行主动打开。这使得客户SCTP发送一个INIT消息（初始化），该消息告诉服务器客户的IP地址清单、初始序列号、用于标识本关联中所有分组的起始标记、客户请求的外出流的数目以及客户能够支持的外来流的数目。

(3) 服务器以一个INIT ACK消息确认客户的INIT消息，其中含有服务器的IP地址清单、初始序列号、起始标记、服务器请求的外出流的数目、服务器能够支持的外来流的数目以及一个状态cookie。状态cookie包含服务器用于确信本关联有效所需的所有状态，它是数字化签名过的，以确保其有效性。

(4) 客户以一个COOKIE ECHO消息回射服务器的状态cookie。除COOKIE ECHO外，该消息可能在同一个分组中还捆绑了用户数据。

- (5) 服务器以一个COOKIE ACK消息确认客户回射的cookie是正确的，本关联于是建立。该消息也可能在同一个分组中还捆绑了用户数据。

以上交换过程至少需要4个分组，因此称之为SCTP的四路握手（four-way handshake）。图2-6展示了这4个分节。

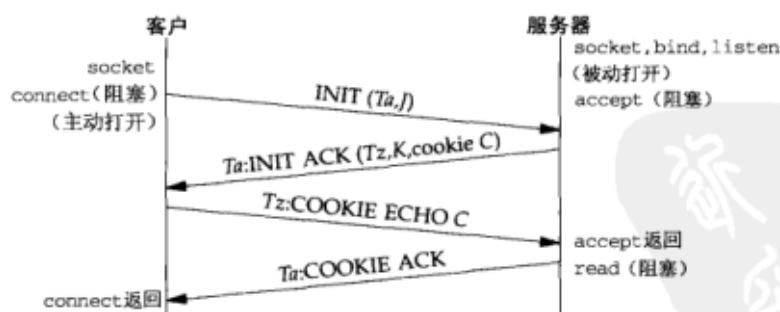


图2-6 SCTP的四路握手

- 四路握手结束后，两端各自选择一个主目的地址，当不存在网络故障时，主目的地址将用作数据要发送带的默认目的地

- 关联终止

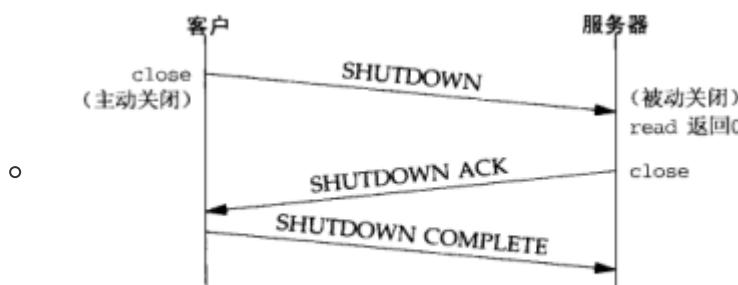


图2-7 SCTP关联关闭时的分组交换

- SCTP通过放置验证标记值就避免了TCP在TIME_WAIT状态保持整个连接的做法
- SCTP的状态转换图

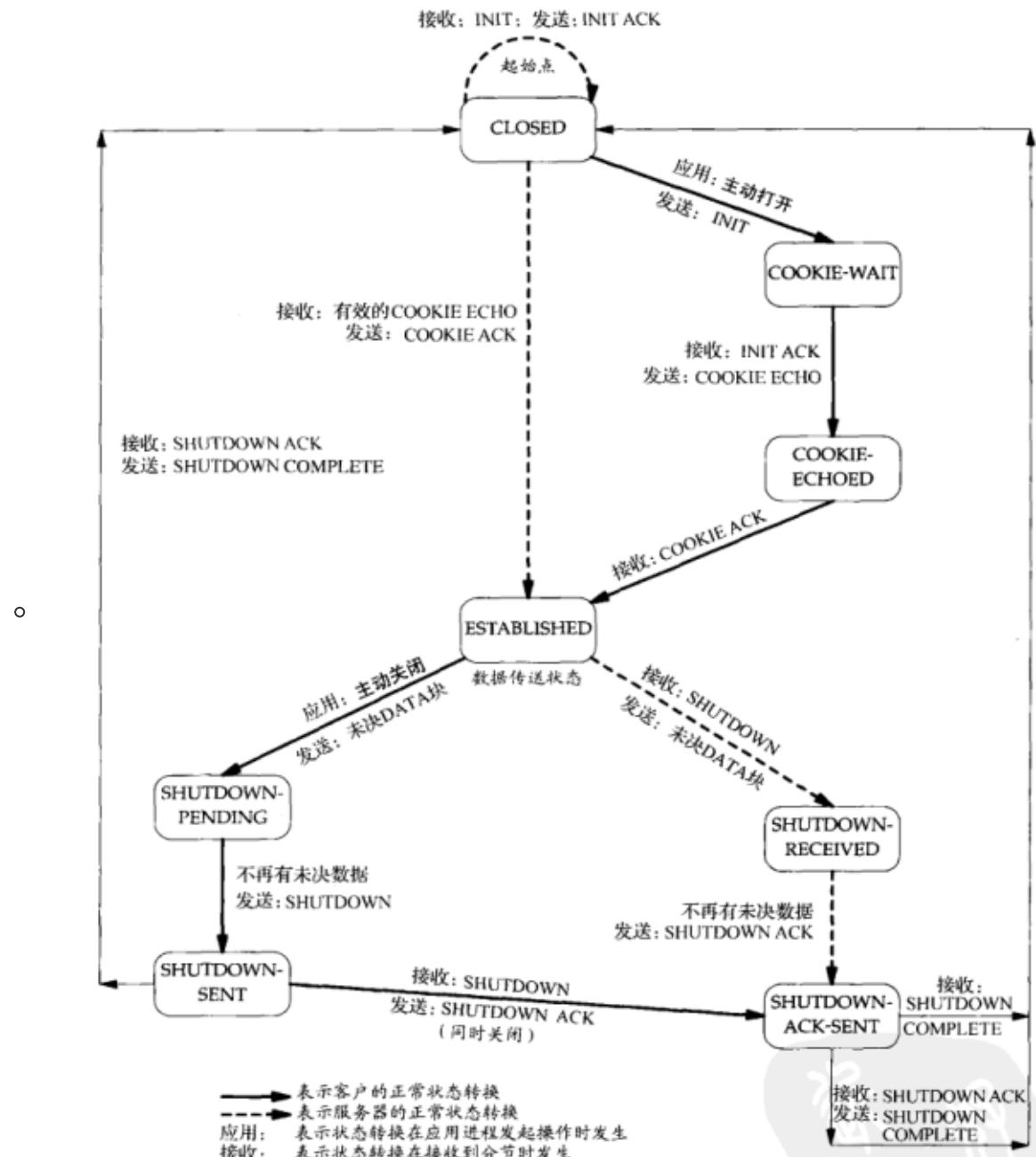


图2-8 SCTP状态转换图

- 观察分组

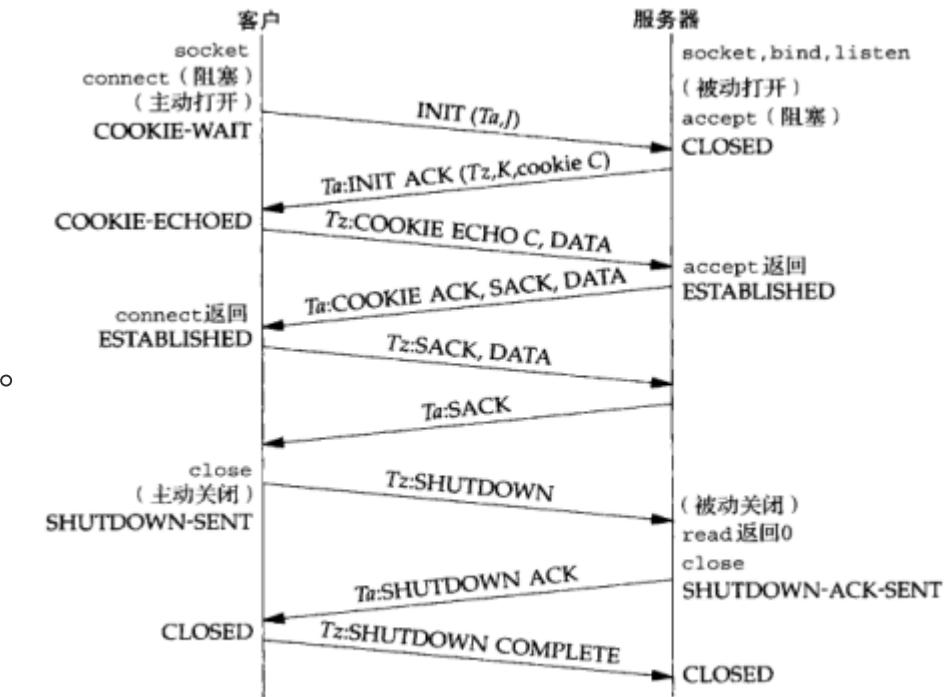


图2-9 SCTP关联中的分组交换

- SCTP选项

2.9 端口号

- 使用16位整数的端口号
- 套接字: IP地址和端口号
- 套接字对: 本地IP地址、本地TCP端口号、外地IP地址、外地TCP端口号

2.10 TCP端口号与并发服务器

- 并发服务器中主服务器循环通过派生一个子进程来处理每个新的连接
- 监听套接字
- TCP无法仅仅通过查看目的端口号来分离外来的分节到不同的端点，它必须查看套接字对的所有四个元素才能确定由哪个端点接收到达的分节

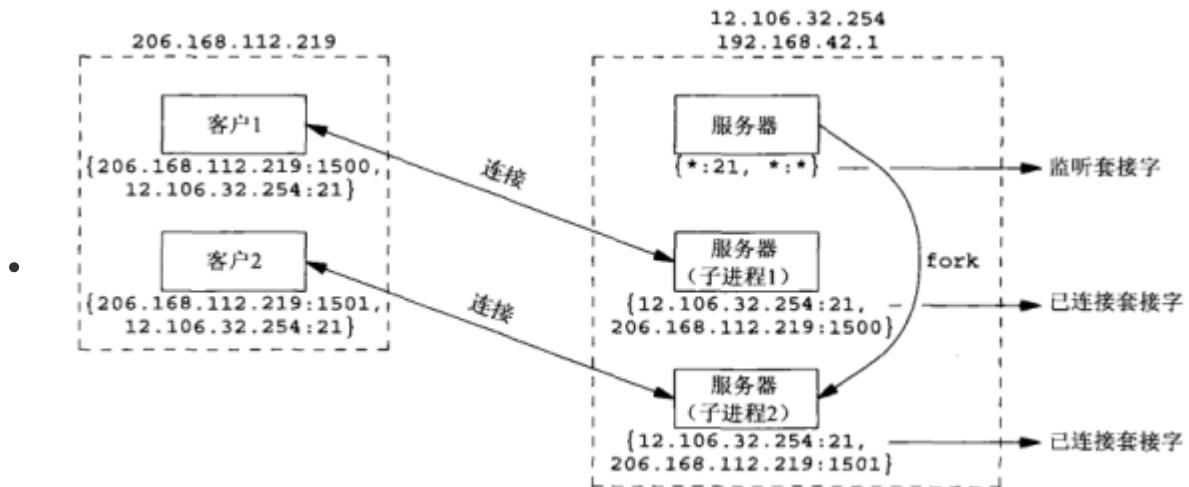


图2-14 第二个客户与同一个服务器的连接

2.11 缓冲区大小及其限制

- 当一个IP数据报从某个接口送出时，如果它的大小超过相应链路的MTU,将执行分片
- IPV4主机对其产生的数据报执行分片，IPV4路由器对其转发的数据报执行分片。IPV6只有主机对其产生的数据报执行分片
- 在两个主机之间的路径中最小的MTU称为路径MTU(最大传输单元)
- 最小重组缓冲区大小，IPv4和IPv6的任何实现都必须保证支持的最小数据报大小
- MSS最大分节大小
- TCP输出

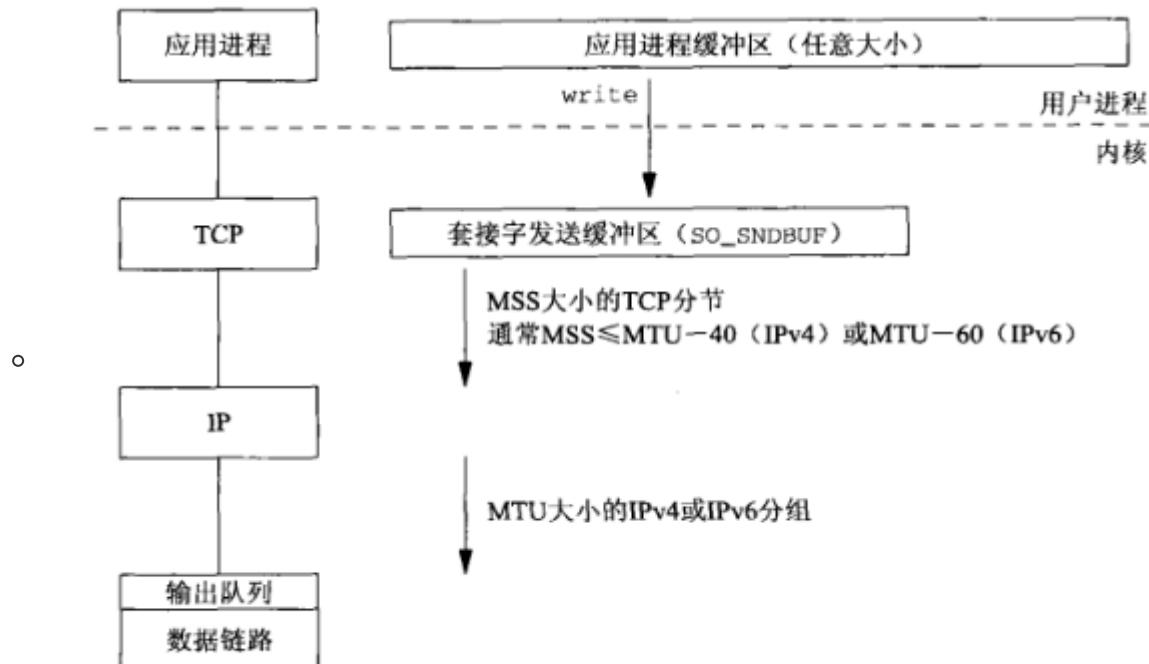


图2-15 应用进程写TCP套接字时涉及的步骤和缓冲区

- UDP输出

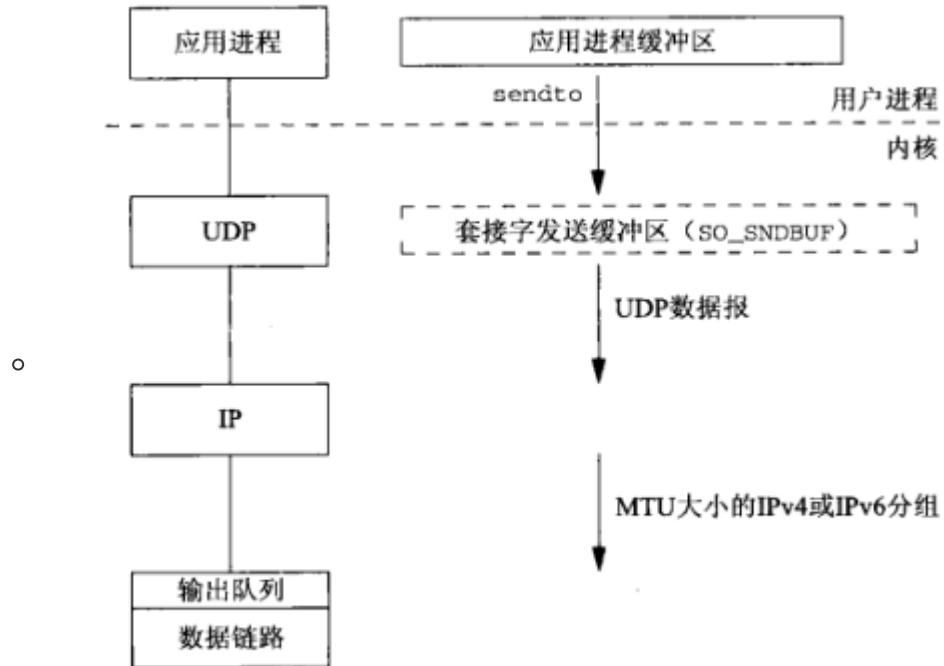


图2-16 应用进程写UDP套接字时涉及的步骤与缓冲区

- SCTP输出

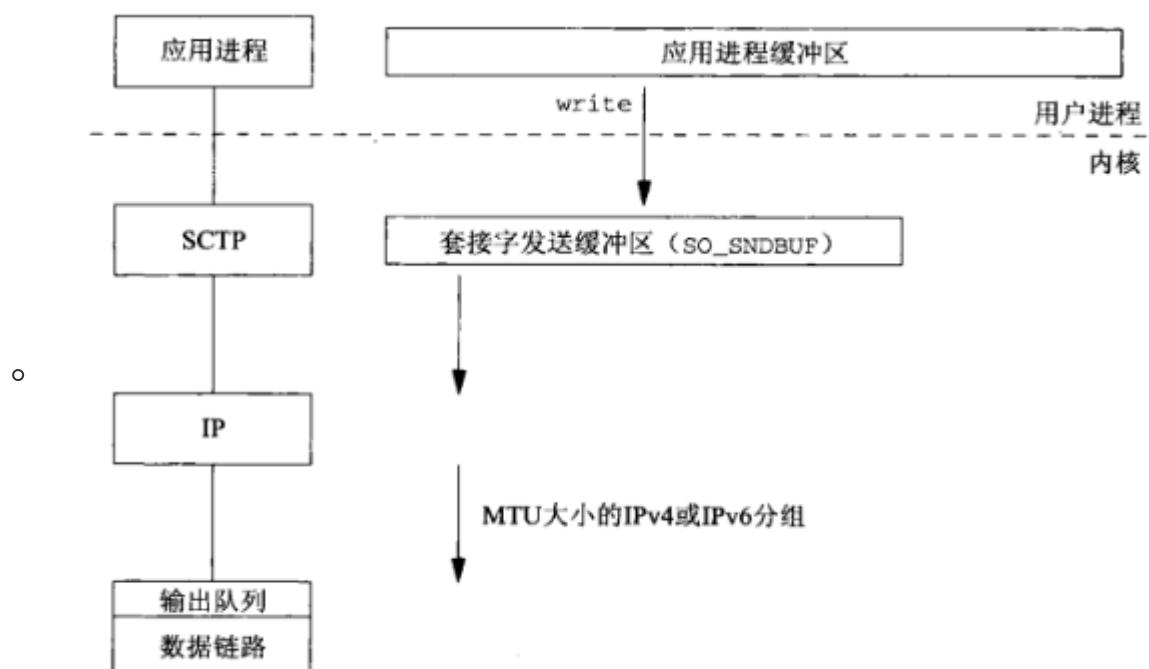


图2-17 应用进程写SCTP套接字时涉及的步骤和缓冲区

2.12 标准因特网服务

2.13 常见因特网应用的协议使用

因特网应用	IP	ICMP	UDP	TCP	SCTP
ping traceroute		• •	•		
OSPF (路由协议) RIP (路由协议) BGP (路由协议)	•		•	•	
BOOTP (引导协议) DHCP (引导协议) NTP (时间协议) TFTP (低级FTP) SNMP (网络管理)			• • • •		
SMTP (电子邮件) Telnet (远程登录) SSH (安全的远程登录) FTP (文件传送) HTTP (Web) NNTP (网络新闻) LPR (远程打印)				• • • • •	
DNS (域名系统) NFS (网络文件系统) Sun RPC (远程过程调用) DCE RPC (远程过程调用)			• • • •	• • • •	
IUA (IP之上的ISDN) M2UA/M3UA (SS7电话信令) H.248 (媒体网关控制) H.323 (IP电话) SIP (IP电话)				• • • • •	• • • • •

图2-19 各种常见因特网应用的协议使用情况

第三章 套接字编程简介

- 套接字地址结构
 - 从进程到内核和从内核到进程
- 地址转化函数在地址的文本表达和它们存放在套接字的地址结构中的二进制值之间进行转化，inet_addr和inet_ntoa函数 inet_pton和inet_ntop函数 它们与所转换的地址类型协议相关

3.2 套接字地址结构

- IPV4套接字地址结构
 - 以sockaddr_in命名，定义在<netinet/in.h>头文件中

```

struct in_addr {
    in_addr_t s_addr;
    /* 32-bit IPv4 address */
    /* network byte ordered */
};

struct sockaddr_in {
    uint8_t sin_len;           /* length of structure (16) */
    sa_family_t sin_family;   /* AF_INET */
    in_port_t sin_port;       /* 16-bit TCP or UDP port number */
    /* network byte ordered */
    struct in_addr sin_addr;  /* 32-bit IPv4 address */
    /* network byte ordered */
    char sin_zero[8];         /* unused */
};

```

图3-1 网际（IPv4）套接字地址结构：sockaddr_in

- 长度字段sin_len,不一定有，涉及路由套接字时才设置和检查它

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

图3-2 POSIX规范要求的数据类型

- IPv4地址和TCP或UDP端口号在套接字地址结构中总是以网络字节序来存储
- 套接字地址结构仅在给定主机上使用
- 通用套接字地址结构
 - 套接字地址结构以指针的方式传递给套接字函数
 - 对套接字函数的调用，必须要将指向特定于协议的套接字地址结构进行类型强制转换，变成指向某个通用套接字地址结构的指针。

```

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

```

图3-3 通用套接字地址结构：sockaddr

```

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

struct sockaddr_in serv;           /* IPv4 socket address structure */

/* fill in serv{} */

bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));

```

- 内核检查sa_family字段的值来确定这个结构的真实类型
- IPV6套接字地址结构

```

struct in6_addr {
    uint8_t s6_addr[16];           /* 128-bit IPv6 address */
    /* network byte ordered */
};

#define SIN6_LEN             /* required for compile-time tests */

struct sockaddr_in6 {
    uint8_t          sin6_len;        /* length of this struct (28) */
    sa_family_t      sin6_family;    /* AF_INET6 */
    in_port_t        sin6_port;      /* transport layer port# */
    /* network byte ordered */

    uint32_t         sin6_flowinfo;   /* flow information, undefined */
    struct in6_addr  sin6_addr;      /* IPv6 address */
    /* network byte ordered */

    uint32_t         sin6_scope_id;   /* set of interfaces for a scope */
};

```

图3-4 IPV6套接字地址结构: sockaddr_in6

- IPv4的地址族是AF_INET, IPV6的地址族是AF_INET6
- 结构中字段的先后顺序做过编排
- 新的通用套接字地址结构

- 新的struct sockaddr_storage足以容纳系统所支持的任何套接字地址结构

```

struct sockaddr_storage {
    uint8_t      ss_len;        /* length of this struct (implementation dependent) */
    sa_family_t  ss_family;    /* address family: AF_xxx value */
    /* implementation-dependent elements to provide:
     * a) alignment sufficient to fulfill the alignment requirements of
     * all socket address types that the system supports.
     * b) enough storage to hold any type of socket address that the
     * system supports.
    */
};

```

图3-5 存储套接字地址结构: sockaddr_storage

- 能满足对齐要求
- 除了ss_family和ss_len外，sockaddr_storage结构中的其他字段对用户来说是透明的，必须类型强制转换或复制到适合于ss_family字段给出地址类型的套接字地址结构中，才能访问其他字段
- 套接字地址结构的比较
 - 5种套接字地址结构: IPV4 IPV6 Unix域 数据链路 存储

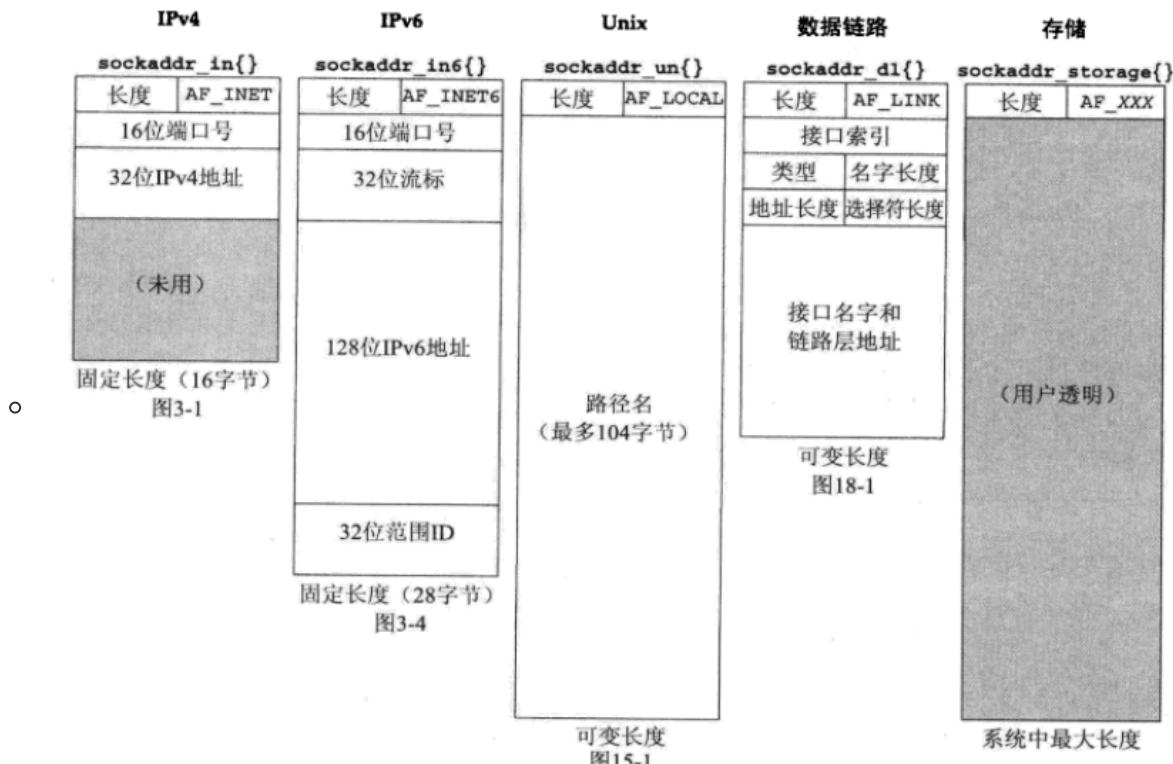


图3-6 不同套接字地址结构的比较

3.3 值-结果参数

- 从进程到内核传递套接字地址结构的函数有3个: bind connect sendto
 - connect(sockfd, (SA*) &serv, sizeof(serv));
- 从内核到进程传递套接字地址结构的函数有4个: accept recvfrom getsockname getpeername
 - 其中两个参数是指向某个套接字地址结构的指针和指向表示该结构大小的整数变量的指针

```

struct sockaddr_un cli; /* Unix domain */
socklen_t len;

len = sizeof(cli); /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */

```

把套接字地址结构大小这个参数从一个整数改为指向某个整数变量的指针，其原因在于：当函数被调用时，结构大小是一个值（value），它告诉内核该结构的大小，这样内核在写该结构时不至于越界；当函数返回时，结构大小又是一个结果（result），它告诉进程内核在该结构中究竟存储了多少信息。这种类型的参数称为值-结果（value-result）参数。图3-8展示了这个情形。

3.4 字节排序函数

- 小端字节序列和大端字节序列
-

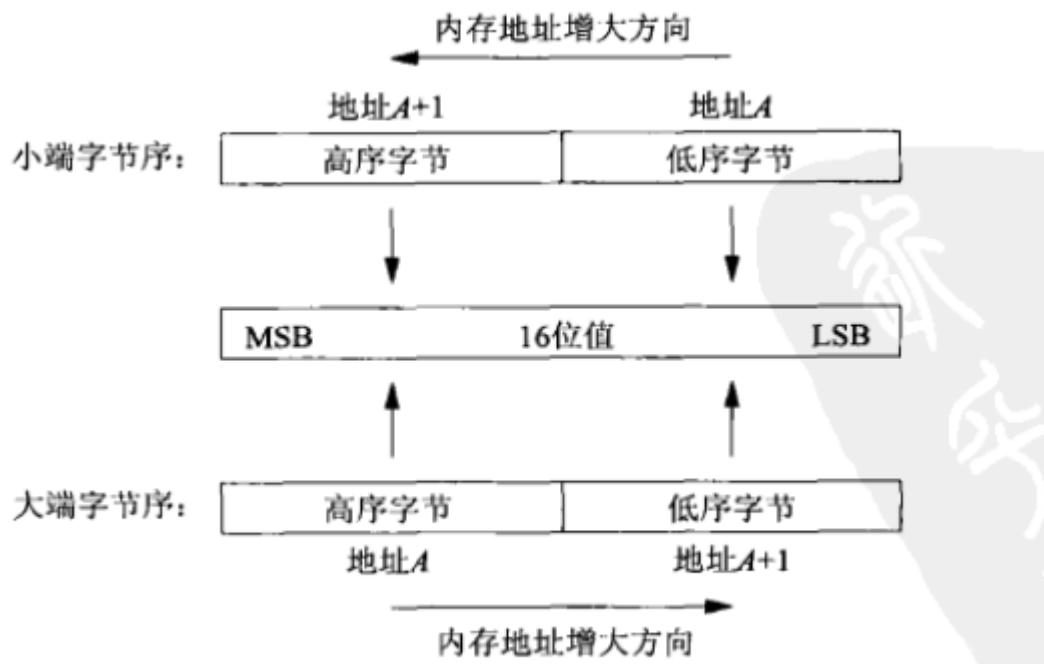


图3-9 16位整数的小端字节序和大端字节序

- x86_64-unknown-linux-gnu: little-endian 小端
- 网络协议必须指定一个网络字节序，网际协议首部字段使用大端字节序传送
- 具体实现可以按主机字节序存储套接字地址结构中的各个字段，等到需要在这些字段和协议首部相应字段之间移动时，再在主机字节序和网络字节序之间进行互转
 - 以下函数实现这两种字节序之间的转换
 - htons
 - htonl 返回网络字节序的值
 - ntohs
 - ntohl 返回主机字节序的值
 - 在大端系统里，这四个函数被定义为空宏

3.5 字节操纵函数

- 名字以b(表示字节)开头的第一组函数起源于4.2BSD，名字以mem(表示内存)开头的第二组函数起源于ANSI C标准
- bzero 把目标子串中指定数目的字节置为0,常使用该函数把一个套接字地址结构初始化为0
- bcopy 将指定数目的字节从源字节串移到目标字节串
- bcmp 比较
- memset 把目标字节串指定数目的字节置为 0
- memcpy 复制
- memcmp 比较

3.6 inet_aton inet_addr inet_ntoa函数

- 地址转换函数，在ASCII字符串与网络字节序的二进制之间转换网际地址
- inet_aton inet_addr inet_ntoa在点分十进制与它长度为32位的网络字节序二进制间转换IPV4地址
- inet_aton将C字符串转换成一个32位的网络字节序二进制
- inet_addr进行相同的转换，但不能处理255.255.255.255

- inet_ntoa函数将一个32位的网络字节序二进制IPV4地址转换成相应的点分十进制数，由该函数的返回值所指向的字符串驻留在静态内存中，该函数是不可重入的

3.7 inet_pton和inet_ntop函数

- 对于IPv4和IPV6都适用
- inet_pton将字符串转换成二进制
- inet_ntop 从数值格式转换成表达式
-

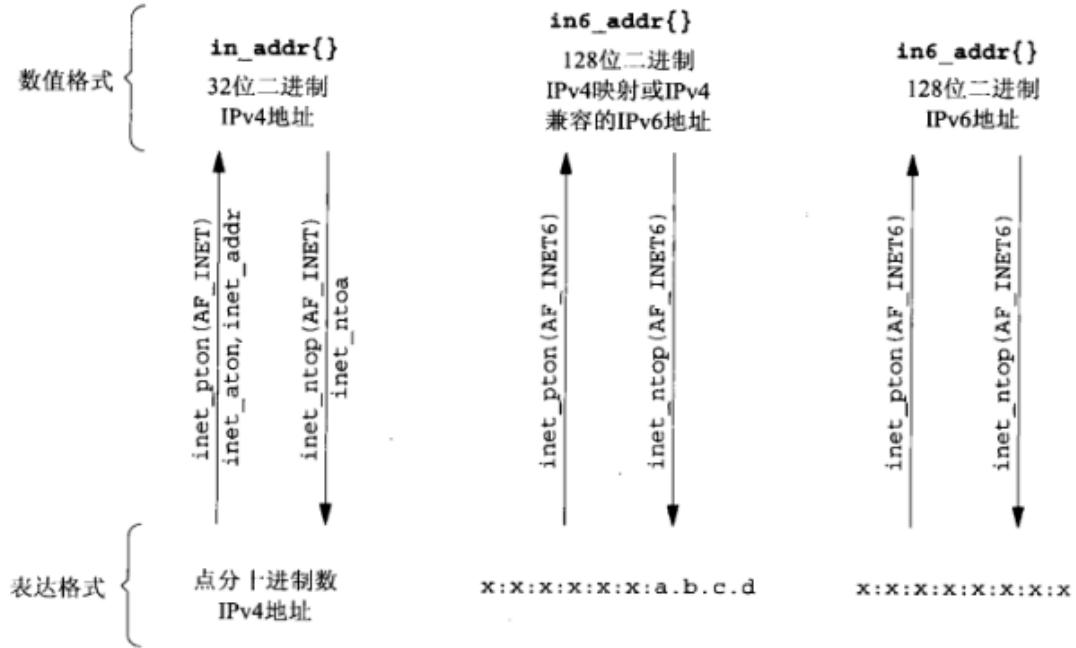


图3-11 地址转换函数小结

- snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks)); 写入到buff中，指定了缓冲区大小

3.8 sock_ntop和相关函数

- 自行编写的函数，以指向某个套接字地址结构的指针为参数，查看该结构的内部，然后调用适当的函数返回该地址的表达格式
- `char* sock_ntop(const struct sockaddr* sockaddr, socklen_t addrlen);`
- `sockaddr`指向一个长度为`addrlen`的套接字地址结构，用它自己的静态缓冲区来保存结果，而指向该缓冲区的一个指针就是它的返回值

```

5 char *
6 sock_ntop(const struct sockaddr *sa, socklen_t salen)
7 {
8     char    portstr[8];
9     static char str[128];           /* Unix domain is largest */
10    switch (sa->sa_family) {
11        case AF_INET: {
12            struct sockaddr_in *sin = (struct sockaddr_in *) sa;
13            if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str)) == NULL)
14                return(NULL);
15            if (ntohs(sin->sin_port) != 0) {
16                sprintf(portstr, sizeof(portstr), ":%d",
17                        ntohs(sin->sin_port));
18                strcat(str, portstr);
19            }
20        }
21    }

```

我们还为操作套接字地址结构定义了其他几个函数，它们将简化我们的代码在IPv4与IPv6之间的移植。

```

#include "ump.h"

int sock_bind_wild(int sockfd, int family);                                返回: 若成功则为0, 若出错则为-1
int sock_cmp_addr(const struct sockaddr *sockaddr1,
                   const struct sockaddr *sockaddr2, socklen_t addrlen);      返回: 若地址为同一协议族且相同则为0, 否则为非0
int sock_cmp_addr(const struct sockaddr *sockaddr1,
                   const struct sockaddr *sockaddr2, socklen_t addrlen);      返回: 若地址为同一协议族且端口相同则为0, 否则为非0
int sock_get_port(const struct sockaddr *sockaddr, socklen_t addrlen);      返回: 若为IPv4或IPv6地址则为非负端口号, 否则为-1
char *sock_ntop_host(const struct sockaddr *sockaddr, socklen_t addrlen);    返回: 若成功则为非空指针, 若出错则为NULL
void sock_set_addr(const struct sockaddr *sockaddr, socklen_t addrlen, void *ptr);
void sock_set_port(const struct sockaddr *sockaddr, socklen_t addrlen, int port);
void sock_set_wild(struct sockaddr *sockaddr, socklen_t addrlen);

```

- sock_bind_wild
- sock_cmp_addr
- sock_cmp_port
- sock_get_port
- sock_ntop_host
- sock_set_addr
- sock_set_port
- sock_set_wild

3.9 readn wirten readline函数

字节流套接字（例如TCP套接字）上的read和write函数所表现的行为不同于通常的文件I/O。字节流套接字上调用read或write输入或输出的字节数可能比请求的数量少，然而这不是出错的状态。这个现象的原因在于内核中用于套接字的缓冲区可能已达到了极限。此时所需的是调用者再次调用read或write函数，以输入或输出剩余的字节。有些版本的Unix在往一个管道中写多于4096字节的数据时也会表现出这样的行为。这个现象在read一个字节流套接字时很常见，但是在write一个字节流套接字时只能在该套接字为非阻塞的前提下才出现。尽管如此，为预防万一，不让实现返回一个不足的字节计数值，我们总是改为调用written函数来取代write函数。

我们提供的以下3个函数是每当我们读或写一个字节流套接字时总要使用的函数。

- 字节流套接字上调用read和write输入或输出的字节数可能比请求的数量少，原因在于内核中用于套接字的缓冲区已经到达了极限，此时需要再次调用read和write以输入或输出剩余的字节
- 提供的3个函数均返回读或写的字节数

- readn
- written
- readline

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

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

121

- ENTER错误，表示系统调用被一个捕获的信号中断
- stdio缓冲区的状态是不可见的
- 在readline.c中使用静态变量实现跨相继函数调用的状态信息维护，其结果是这些函数变得不可重入或非线程安全了

第四章 基本套接字编程

- 客户/服务器程序
- 并发服务器

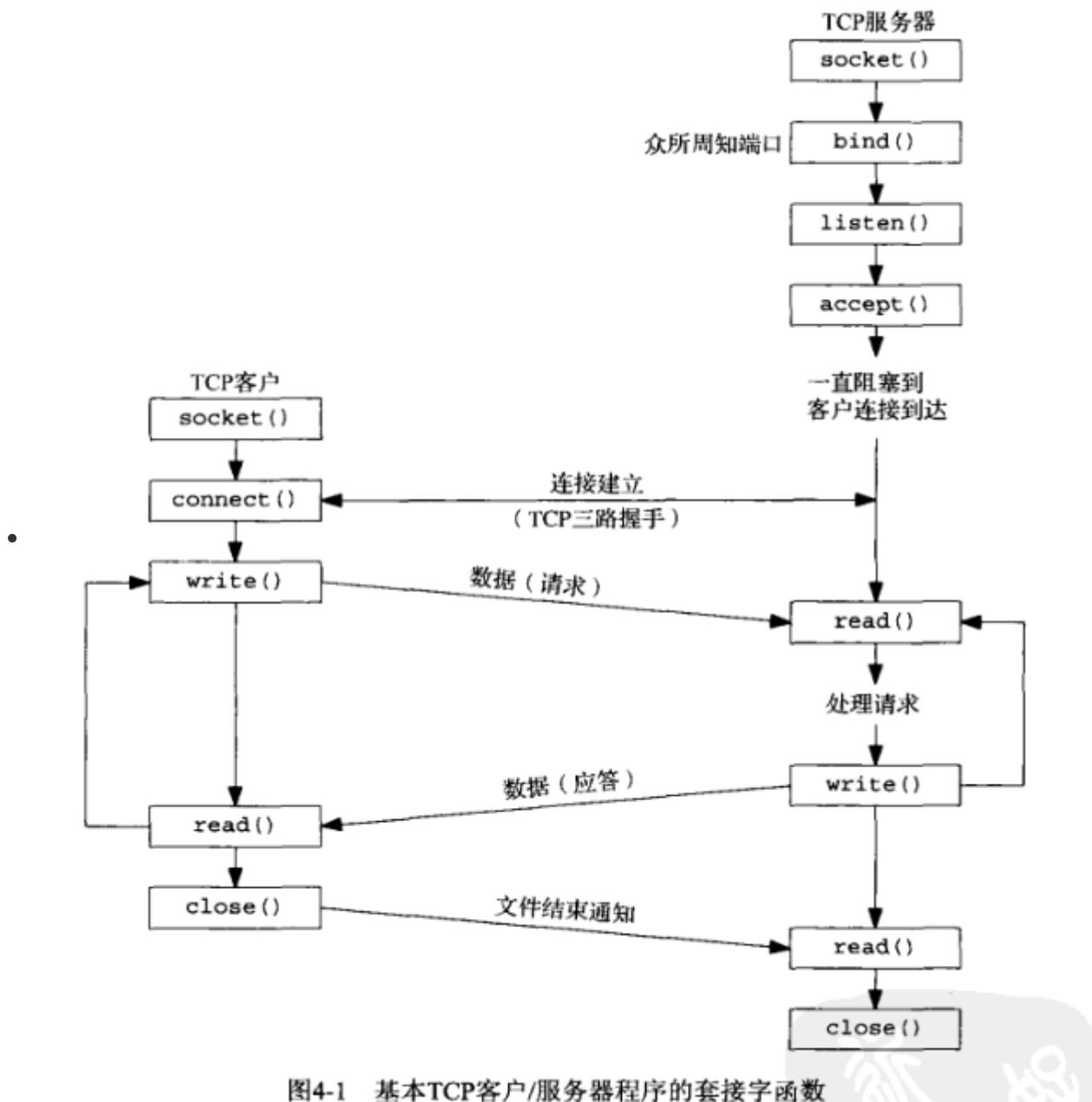


图4-1 基本TCP客户/服务器程序的套接字函数

4.2 socket函数

- int socket(int family,int type, int protocol)
- TCP是一个字节流协议，仅支持SOCK_STREAM套接字
- 路由套接字是内核中路由表的接口
- 密钥套接字是与内核中密钥表的接口
- 套接字描述符与文件描述符类似 sockfd
-

<i>family</i>	说 明
AF_INET	IPv4协议
AF_INET6	IPv6协议
AF_LOCAL	Unix域协议（见第15章）
AF_ROUTE	路由套接字（见第18章）
AF_KEY	密钥套接字（见第19章）

图4-2 socket函数的*family*常值

- type参数指明套接字类型

<i>type</i>	说 明
SOCK_STREAM	字节流套接字
SOCK_DGRAM	数据报套接字
SOCK_SEQPACKET	有序分组套接字
SOCK_RAW	原始套接字

图4-3 socket函数的*type*常值

<i>protocol</i>	说 明
IPPROTO_TCP	TCP传输协议
IPPROTO_UDP	UDP传输协议
IPPROTO_SCTP	SCTP传输协议

图4-4 socket函数AF_INET或AF_INET6的*protocol*常值

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP SCTP	TCP SCTP	是		
SOCK_DGRAM	UDP	UDP	是		
SOCK_SEQPACKET	SCTP	SCTP	是		
SOCK_RAW	IPv4	IPv6		是	是

图4-5 socket函数中*family*和*type*参数的组合

- AF_XXX和PF_XXX
 - AF_前缀表示地址族
 - PF_前缀表示协议族

4.3 connect函数

- TCP客户用connect函数建立与TCP服务器的连接
- int connect(int sockfd, const struct sockaddr* servaddr, socklen_t addrlen);
- 套接字地址结构必须含有服务器的IP地址和端口号
- 客户在调用函数connect前不必非得调用bind函数，内核会确定源IP地址，并选择一个临时端口作为源端口
- 如果是TCP套接字，调用connect函数将激发TCP的三路握手过程，而且仅在连接建立成功或出错时才返回
 - (1) 若TCP客户没有收到SYN分节的响应，则返回ETIMEDOUT错误。举例来说，调用connect函数时，4.4BSD内核发送一个SYN，若无响应则等待6s后再发送一个，若仍无响应则等待24s后再发送一个（TCPv2第828页）。若总共等了75s后仍未收到响应则返回本错误。
有些系统提供对超时值的管理性控制，见TCPv1的附录E。
 - (2) 若对客户的SYN的响应是RST（表示复位），则表明该服务器主机在我们指定的端口上没有进程在等待与之连接（例如服务器进程也许没在运行）。这是一种硬错误（hard error），客户一接收到RST就马上返回ECONNREFUSED错误。
 - RST是TCP在发生错误时发送的一种TCP分节。产生RST的三个条件是：目的地为某端口的SYN到达，然而该端口上没有正在监听的服务器（如前所述）；TCP想取消一个已有连接；TCP接收到一个根本不存在的连接上的分节。（TCPv1第246~250页有更详细的信息。）
 - (3) 若客户发出的SYN在中间的某个路由器上引发了一个“destination unreachable”（目的地不可达）ICMP错误，则认为是一种软错误（soft error）。客户主机内核保存该消息，并按第一种情况中所述的时间间隔继续发送SYN。若在某个规定的时间（4.4BSD规定75s）后仍未收到响应，则把保存的消息（即ICMP错误）作为EHOSTUNREACH或ENETUNREACH错误返回给进程。以下两种情形也是有可能的：一是按照本地系统的转发表，根本没有到达远程系统的路径；二是connect调用根本不等待就返回。
- 一个没有运行时间获取服务器程序的主机，服务器主机立即响应一个RST分节
 - connect error: Connection refused
- 指定本地子网上并不存在的一个IP地址，当客户主机发出ARP请求时，它将永远收不到ARP响应
 - connect error: Connection timed out
- 指定一个因特网中不可到达的IP地址，用tcpdump观察分组，发现路由器返回了主机不可达的ICMP错误
 - connect error: NO route to host
- 若connect失败则该套接字不再可用，必须关闭

4.4 bind 函数

- bind函数把一个本地协议地址赋予一个套接字
- 对于网际协议，协议地址是IPV4或IPV6的地址与TCP或UDP端口号的组合
- 对于TCP，调用bind函数可用指定一个端口号，或指定一个IP地址，也可以两者都指定，还可以都不指定
 - 如果一个TCP客户或服务器未曾调用bind绑定一个端口，则内核就要为相应的套接字选择一个临时端口

- 服务器在启动时捆绑它们的众所周知端口，我们在图1-9中已看到了。如果一个TCP客户或服务器未曾调用bind捆绑一个端口，当调用connect或listen时，内核就要为相应的套接字选择一个临时端口。让内核来选择临时端口对于TCP客户来说是正常的，除非应用需要一个预留端口（图2-10）；然而对于TCP服务器来说却极为罕见，因为服务器是通过它们的众所周知端口被大家认识的。

这个规则的例外是远程过程调用（Remote Procedure Call, RPC）服务器。它们通常就由内核为它们的监听套接字选择一个临时端口，而该端口随后通过RPC端口映射器进行注册。客户在connect这些服务器之前，必须与端口映射器联系以获取它们的临时端口。这种情况也适用于使用UDP的RPC服务器。

- 进程可以把一个特定的IP地址捆绑到它的套接字上，不过这个IP地址必须属于其所在主机的网络接口之一。对于TCP客户，这就为在该套接字上发送的IP数据报指派了源IP地址。对于TCP服务器，这就限定该套接字只接收那些目的地为这个IP地址的客户连接。
 - TCP客户通常不把IP地址捆绑到它的套接字上。当连接套接字时，内核将根据所用外出网络接口来选择源IP地址，而所用外出接口则取决于到达服务器所需的路径（TCPv2第737页）。如果TCP服务器没有把IP地址捆绑到它的套接字上，内核就把客户发送的SYN的目的IP地址作为服务器的源IP地址（TCPv2第943页）。

正如我们所说，调用bind可以指定IP地址或端口，可以两者都指定，也可以都不指定。图4-6汇总了如何根据预期的结果，设置sin_addr和sin_port或者sin6_addr和sin6_port的值。

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

图4-6 给bind函数指定要捆绑的IP地址和/或端口号产生的结果

- 对于IPV4，通配地址由常值INADDR_ANY来指定，其值一般为0，告知内核去选择IP地址
- 对于IPV6，系统预先分配in6addr_any变量并将其初始化为常值IN6ADDR_ANY_INIT，通配地址
- 为了得到内核所选择的临时端口号值，必须调用函数getsockname来返回协议地址
- 进程捆绑非通配IP地址到套接字的常见例子是为多个组织提供Web服务器的主机上
 - 把子网的所有IP地址都定义成单个网络接口的别名（使用ifconfig命令的alias选项）

替换上述方法的另一种技术是运行捆绑通配地址的单个服务器。当一个连接到达时，服务器调用getsockname函数获取来自客户的目的IP地址，它在我们的上述讨论中可以是198.69.10.128、198.69.10.129，等等。服务器然后根据这个客户连接所发往的IP地址来处理客户的请求。

捆绑非通配IP地址的好处是：把一个给定的目的IP地址解复用到一个给定的服务器进程是由内核（而不是服务器进程）完成的。
- 一个常见错误是EADDRINUSE（地址已使用）

4.5 listen函数

- 仅由TCP服务器调用
 - 当socket函数创建一个套接字时，它被假设为一个主动套接字，listen函数把一个未连接的套接字转换成一个被动套接字，指示内核应接受指向该套接字的连接请求
- int listen(int sockfd, int backlog); backlog指示了最大连接个数

- 内核为任何一个给定的监听套接字维护两个队列:

- 未完成连接队列
- 已完成连接队列

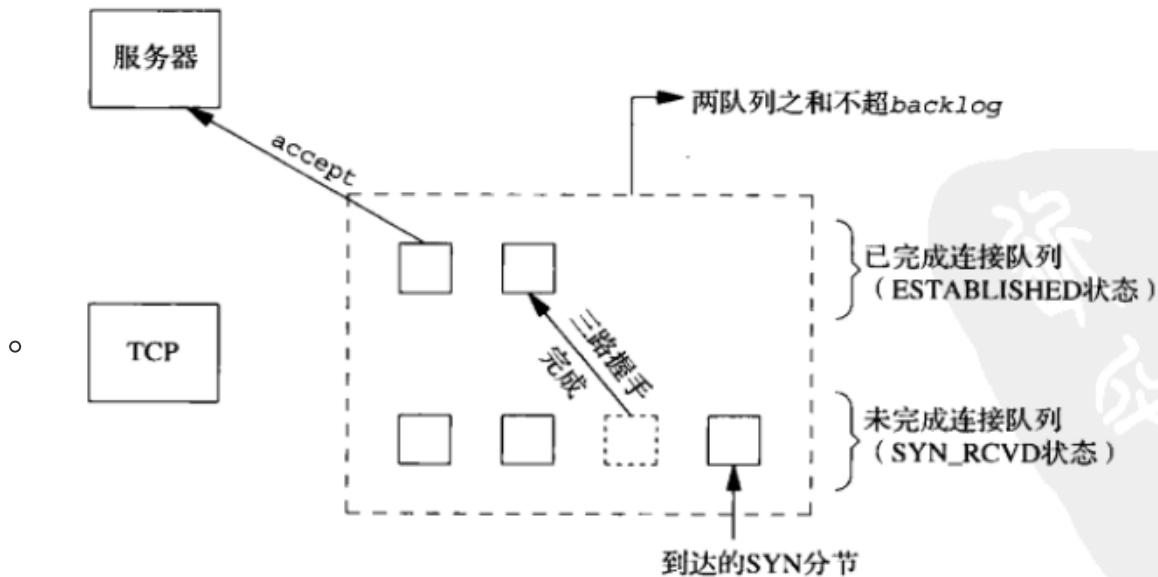


图4-7 TCP为监听套接字维护的两个队列

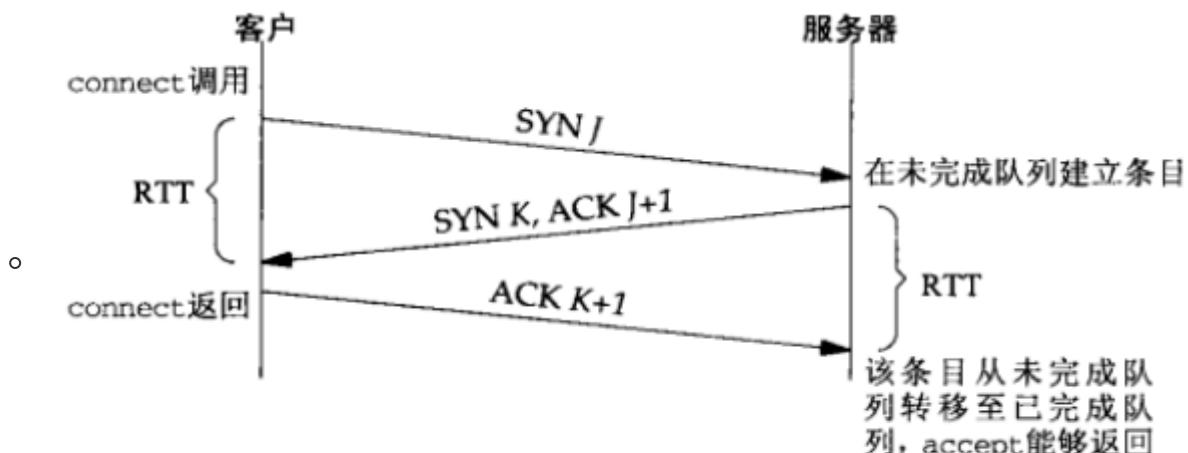


图4-8 TCP三路握手和监听套接字的两个队列

- 当进程调用accept时，已完成连接队列中的队头项将返回给进程，或者如果该队列为空，那么该进程将被投入睡眠，直到TCP在该队列中放入一项才唤醒它
- 不要把backlog定义为0
- 允许通过环境变量LISTENQ覆写由调用者指定的值
- 当一个客户SYN到达时，若这些队列是满的，TCP就忽略该分节
- 在三路握手完成之后，但在服务器调用accept之前到达的数据应由服务器TCP排队，最大数据量为相应已连接套接字的接受缓冲区大小

4.6 accept函数

- 当进程调用accept时，已完成连接队列中的队头项将返回给进程，或者如果该队列为空，那么该进程将被投入睡眠，直到TCP在该队列中放入一项才唤醒它
- `int accept(int sockfd, struct sockaddr*cliaddr,socklen_t *addrlen);`

- 参数cliaddr和addrlen用来返回已连接的对端进程(客户)的协议地址, addrlen是值-结果参数。调用前, *addrlen所引用的整数值置为由cliaddr所指的套接字地址结构的长度, 返回时, 该整数值为由内核存放在该套接字地址结构内的确切字节数
- 如果accept成功, 其返回值为由内核自动生成的一个全新描述符, 代表与所返回客户的TCP连接, 即已连接套接字描述符
- 一个服务器仅创建一个监听套接字, 在服务器的生命期内一直存在, 当服务器完成对某个给定客户服务时, 相应的已连接套接字就被关闭
- 可以把cliaddr和addrlen置为NULL
- 环回地址127.0.0.1

4.7 fork和exec函数

- 父进程中调用fork之前打开的所有描述符在fork返回后由子进程分享
- 父进程调用accept之后调用fork, 所接受的已连接套接字随后就在父进程与子进程之间共享, 通常情况下, 子进程接着读写这个已连接套接字, 父进程则关闭这个已连接套接字
- 进程调用exec之前打开的描述符通常跨exec继续保持打开, 本默认行为可以使用fcnt1设置FD_CLOEXEC描述符标志禁止掉, inetd服务器就利用了这个特性

4.8 并发服务器

```

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

```

图4-13 典型的并发服务器程序轮廓

- 每个文件或套接字都有一个引用计数, 引用计数在文件表中维护, 它是当前打开着的引用该文件或套接字的描述符的个数
- fork返回后, 这两个描述符就在父进程与子进程间共享(也就是被复制), 因此这两个套接字相关联的文件表项中各自的访问计数均为2

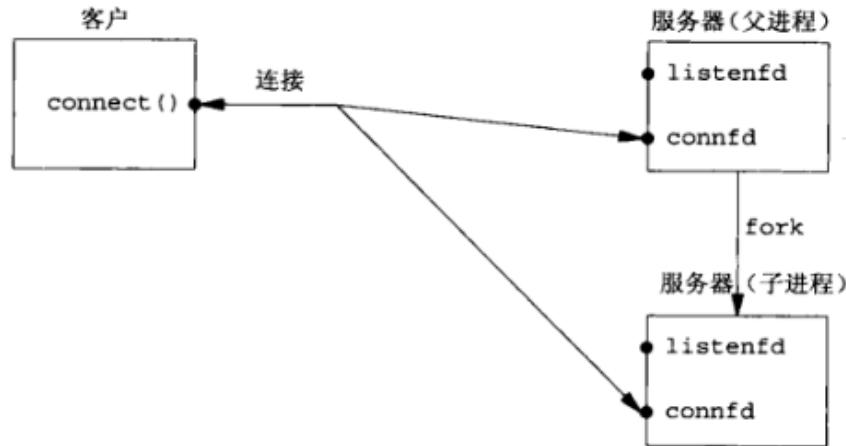


图4-16 fork返回后客户/服务器的状态

- 注意，此时listenfd和connfd这两个描述符都在父进程和子进程之间共享（被复制）。再下一步是由父进程关闭已连接套接字，由子进程关闭监听套接字，如图4-17所示。

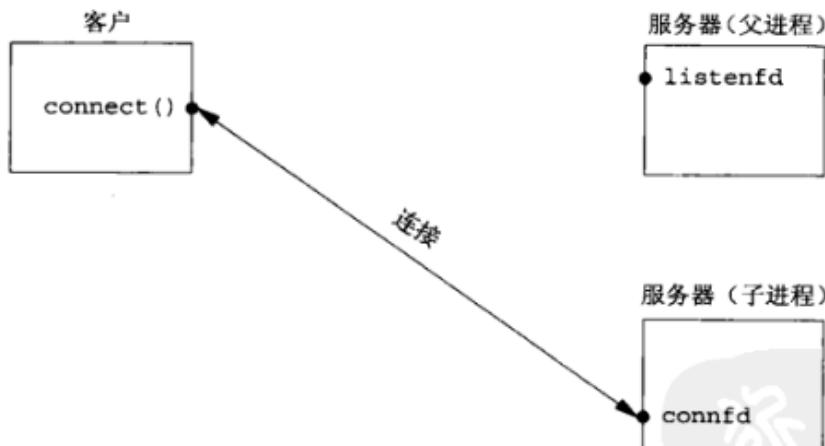


图4-17 父子进程关闭相应套接字后客户/服务器的状态

4.9 close函数

- 用来关闭套接字，终止TCP连接
close一个TCP套接字的默认行为是把该套接字标记成已关闭，然后立即返回到调用进程。
- 该套接字描述符不能再由调用进程使用，也就是说它不能再作为read或write的第一个参数。
然而TCP将尝试发送已排队等待发送到对端的任何数据，发送完毕后发生的是正常的TCP连接终止序列（2.6节）。

4.10 getsockname和getpeername函数

- 返回与某个套接字关联的本地协议地址 getsockname
- 返回与某个套接字关联的外地协议地址 getpeername
 - 在一个没有调用bind的TCP客户上，connect成功返回后，getsockname用于返回由内核赋予该连接的本地IP地址和本地端口号。
 - 在以端口号0调用bind（告知内核去选择本地端口号）后，getsockname用于返回由内核赋予的本地端口号。
- 已连接套接字描述符跨exec继续保持开放，调用getpeername,用于获取客户的IP地址和端口号

- 大多数TCP服务器是并发的，大多数UDP服务器是迭代的

第五章 TCP客户/服务器程序示例

- 回射服务器

5.2 TCP回射服务器程序: main函数

5.3 TCP回射服务器程序: str_echo函数

- 接收到客户的FIN将导致服务器子进程的read函数返回0

5.4 TCP回射客户程序: main函数

5.5 TCP回射客户程序: str_cli函数

5.6 正常启动

- netstat -a 监听套接字的状态, netstat用*表示一个为0的IP地址或为0的端口号
- ps -o pid,ppid,tty,stat,args,wchan 查看进程状态
- pts/6 表示伪终端号6
- STAT列为S, 表明进程在为等待某些资源而睡眠, 进程处于睡眠状态时WCHAN列出相应的条件
- Linux在进程阻塞于accept或connect时, 输出wait_for_connect, 在进程阻塞于套接字输入或输出时, 输出tcp_data_wait, 在进程阻塞于终端I/O时, 输出read_chan
- netstat -a | grep 9877

5.7 正常终止

- Ctrl+D表示EOF字符
- 僵死状态Z表示

5.8 POSIX信号处理

- 通过调用sigaction函数来设定一个信号的处置
 - 提供信号处理函数
 - 可以把某个信号的处置设定为SIG_IGN来忽略它
 - 可以把某个信号的处置设定为SIG_DFL来启用它的默认处置
- signal函数
 - 定义自己的signal函数, 只是调用POSIX的sigaction函数
 - 构造程序是指使用make工具把源程序和/或目标程序编译链接成可执行程序, 构造其中各个程序的makefile文件

```

1 #include    "unp.h"
2
3 Sigfunc *
4 signal(int signo, Sigfunc *func)
5 {
6     struct sigactionact, oact;
7
8     act.sa_handler = func;
9     sigemptyset(&act.sa_mask);
10    act.sa_flags = 0;
11    if (signo == SIGALRM) {
12 #ifdef SA_INTERRUPT
13        act.sa_flags |= SA_INTERRUPT; /* SunOS 4.x */
14 #endif
15    } else {
16 #ifdef SA_RESTART
17        act.sa_flags |= SA_RESTART; /* SVR4, 4.4BSD */
18 #endif
19    }
20    if (sigaction(signo, &act, &oact) < 0)
21        return(SIG_ERR);
22    return(oact.sa_handler);
23 }

```

图5-6 调用POSIX `sigaction`函数的`signal`函数

5.9 处理SIGCHLD信号

- 处理僵死进程
- 在信号处理函数中调用printf这样的标准I/O函数是不合适的
- 若信号是在父进程阻塞于慢系统调用(accept)时由父进程捕获的，内核就会使accept返回一个ENTER错误(被中断的系统调用)，而父进程不处理该错误，于是终止
- 处理被中断的系统调用
 - 慢系统调用: 可能永远阻塞的系统调用
 - 当阻塞于某个慢系统调用的一个进程捕获某个信号且相应信号处理函数返回时，该系统调用可能返回一个ENTER错误，有些内核自动重启某些被中断的系统调用
 - 自己重启被中断的系统调用

```

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

```

- 当connect被一个捕获的信号中断而且不自动重启时，我们就必须调用select来等待连接完成

5.10 wait和waitpid函数

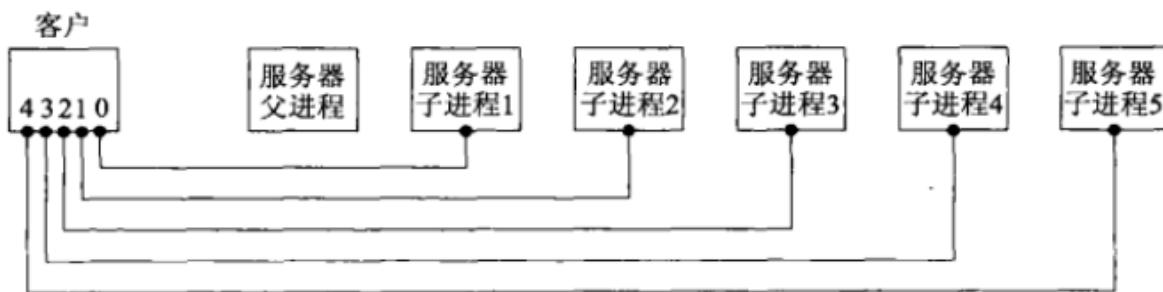


图5-8 与同一个并发服务器建立了5个连接的客户

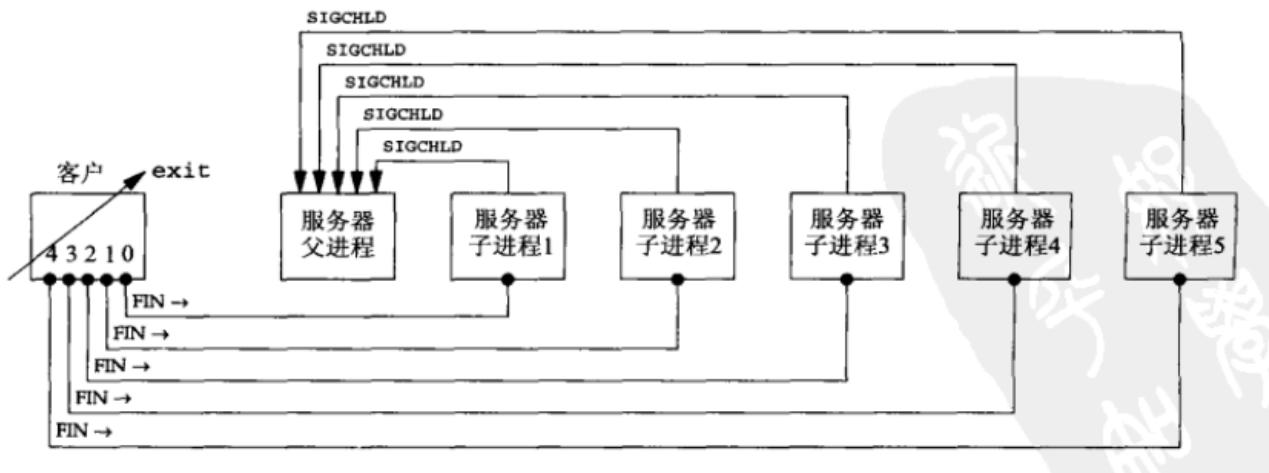


图5-10 客户终止，关闭5个连接，终止5个子进程

- 建立一个信号处理函数并在其中调用wait不足以防止出现僵死进程，由于Unix信号是不排队的。可以在一个循环内调用waitpid处理，以获取所有已终止子进程的状态，指定WNOHANG选项，告知waitpid在有尚未终止的子进程在运行时不要阻塞

```

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

```

- 网络编程遇到的三种情况
 - 当fork子进程时，必须捕获SIGCHLD信号；
 - 当捕获信号时，必须处理被中断的系统调用；
 - SIGCHLD的信号处理函数必须正确编写，应使用waitpid函数以免留下僵死进程。

5.11 accept返回前连接终止

- 可设置SO_LINGER套接字选项以产生RST

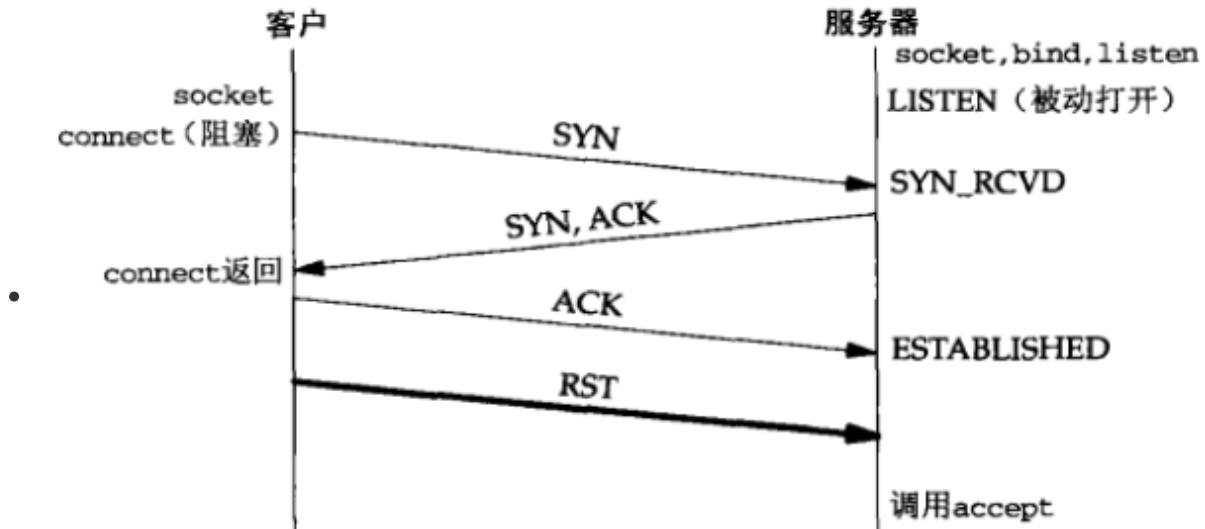


图5-13 ESTABLISHED状态的连接在调用accept之前收到RST

- POSIX指出返回一个错误给服务器进程，errno值是ECONNABORTED

5.12 服务器进程终止

- 客户TCP接收到FIN只是表示服务器进程已关闭了连接的服务器端，从而不向其中发送任何数据而已，FIN接收并没有告知客户TCP服务器进程已经终止
- 当服务器TCP接收到来自客户的数据时，既然先前打开的那个套接字的进程已经终止，于是响应一个RST
- 当FIN到达套接字时，客户正阻塞在fgets调用上，客户实际上在应对两个描述符—套接字和用户输入，它不能单纯阻塞在这两个源中某个特定源上，而是应该阻塞在其中任何一个源上，正是select和poll的编写目的之一

5.13 SIGPIPE信号

- 当一个进程向某一个已收到RST的套接字执行写操作时，内核向该进程发送一个SIGPIPE信号，该信号的默认行为是终止进程，因此进程必须捕获它
- 写一个已接收了FIN的套接字不成问题，但是写一个已接收了RST的套接字则是一个错误

5.14 服务器主机崩溃

5.15 服务器主机崩溃后重启

- 当服务器主机崩溃后重启时，它的TCP丢失了崩溃前的所有连接信息，因此服务器TCP对于所收到的来自客户的数据分节响应一个RST
- 当客户TCP收到该RST时，客户正阻塞于readline调用，导致该调用返回ECONNRESET错误

5.16 服务器主机关机

Unix系统关机时，init进程通常先给所有进程发送SIGTERM信号（该信号可被捕获），等待一段固定的时间（往往在5到20秒之间），然后给所有仍在运行的进程发送SIGKILL信号（该信号不能被捕获）。这么做留给所有运行的进程一小段时间来清除和终止。如果我们不捕获SIGTERM信号并终止，我们的服务器将由SIGKILL信号终止。^①当服务器子进程终止时，它的所有打开着的描述符都被关闭，随后发生的步骤与5.12节中讨论过的一样。正如那一节所述，我们必须在客户中使用select或poll函数，使得服务器进程的终止一经发生，客户就能检测到。

5.17 TCP程序

5.18 数据格式

- 在客户和服务器之间传递文本串

```
1 #include "unp.h"
2 void
3 str_echo(int sockfd)
4 {
5     long    arg1, arg2;
6     ssize_t n;
7     char    line[MAXLINE];
8     for ( ; ; ) {
9         if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
10             return;           /* connection closed by other end */
11         if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
12             snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
13         else
14             snprintf(line, sizeof(line), "input error\n");
15         n = strlen(line);
16         Writen(sockfd, line, n);
17     }
18 }
```

tcpcliserv.c

图5-17 对两个数求和的str_echo函数

- sscanf函数
- snprintf函数

- 在客户与服务器之间传递二进制结构

- 注意两个主机的体系结构可能不同

(1) 不同的实现以不同的格式存储二进制数。最常见的格式便是3.4节讨论过的大端字节序与小端字节序。

(2) 不同的实现在存储相同的C数据类型上可能存在差异。举例来说，大多数32位Unix系统使用32位表示长整数，而64位系统却典型地使用64位来表示同样的数据类型（图1-17）。对于short、int或long等整数类型，它们各自的大小没有确定的保证。

(3) 不同的实现给结构打包的方式存在差异，取决于各种数据类型所用的位数以及机器的对齐限制。因此，穿越套接字传送二进制结构绝不是明智的。

- 解决这种数据格式问题有两个常用方法。

(1) 把所有的数值数据作为文本串来传递。这就是图5-17的做法。当然这里假设客户和服务器主机具有相同的字符集。

(2) 显式定义所支持数据类型的二进制格式（位数、大端或小端字节序），并以这样的格式在客户与服务器之间传递所有数据。远程过程调用（Remote Procedure Call, RPC）软件包通常使用这种技术。RFC 1832[Srinivasan 1995]阐述了Sun RPC软件包所用的外部数据表示（External Data Representation, XDR）标准。

第六章 I/O复用: select和poll函数

- 进程需要一种预先告知内核的能力，使得内核一旦发现进程指定的一个或多个I/O条件就绪，它就通知进程，这个能力称为I/O复用，是由select和poll函数支持的
- I/O复用的场合
 - 当客户使用多个描述符(如交互输入和网络套接字)
 - 一个TCP服务器既要处理监听套接字，又要处理已连接套接字
 - 一个服务器既要处理TCP，又要处理UDP
 - 一个服务器要处理多个协议或多个服务

6.2 I/O模型

- Unix下可用的5种I/O模型
 - 阻塞式I/O
 - 非阻塞式I/O
 - I/O复用(select和poll)
 - 信号驱动式I/O(SIGIO)
 - 异步I/O
- 一个输入操作
 - 等待数据准备好(从网络中到达)
 - 从内核向进程复制
- 阻塞式I/O模型
 - 默认情况下，所有套接字都是阻塞的

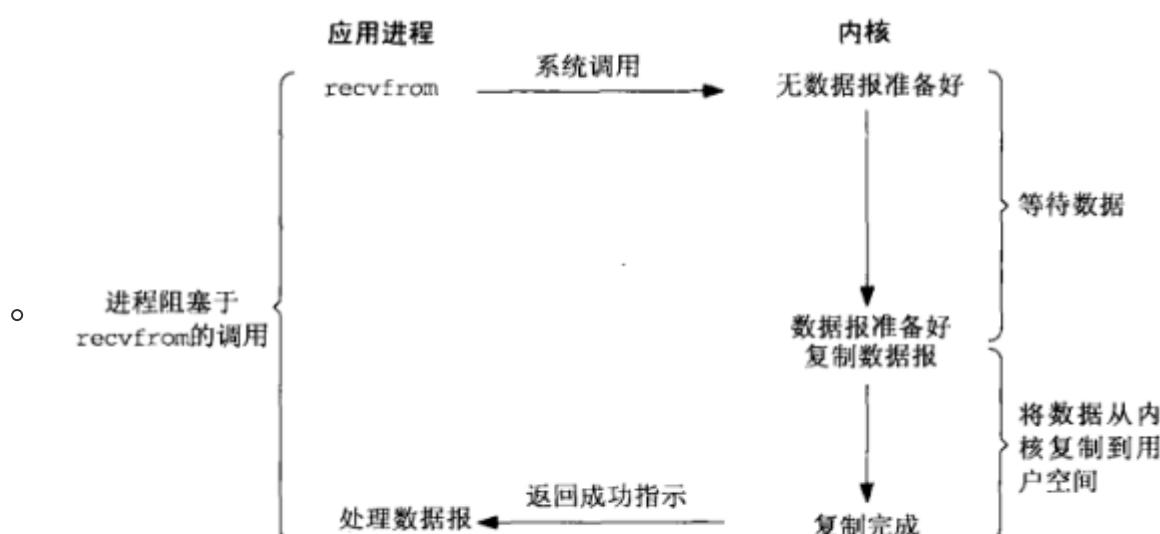


图6-1 阻塞式I/O模型

- 进程调用recvfrom, 其系统调用直到数据报到达且被复制到应用进程的缓冲区中或者发生错误才返回，进程在从调用recvfrom开始到它返回的整段时间内是被阻塞的，成功返回后，进程开始处理数据报
- 非阻塞式I/O模型
 - 进程把一个套接字设置成非阻塞是在通知内核：当所请求的I/O操作非得把本进程投入睡眠才能完成时，不要把本进程投入睡眠，而是返回一个错误

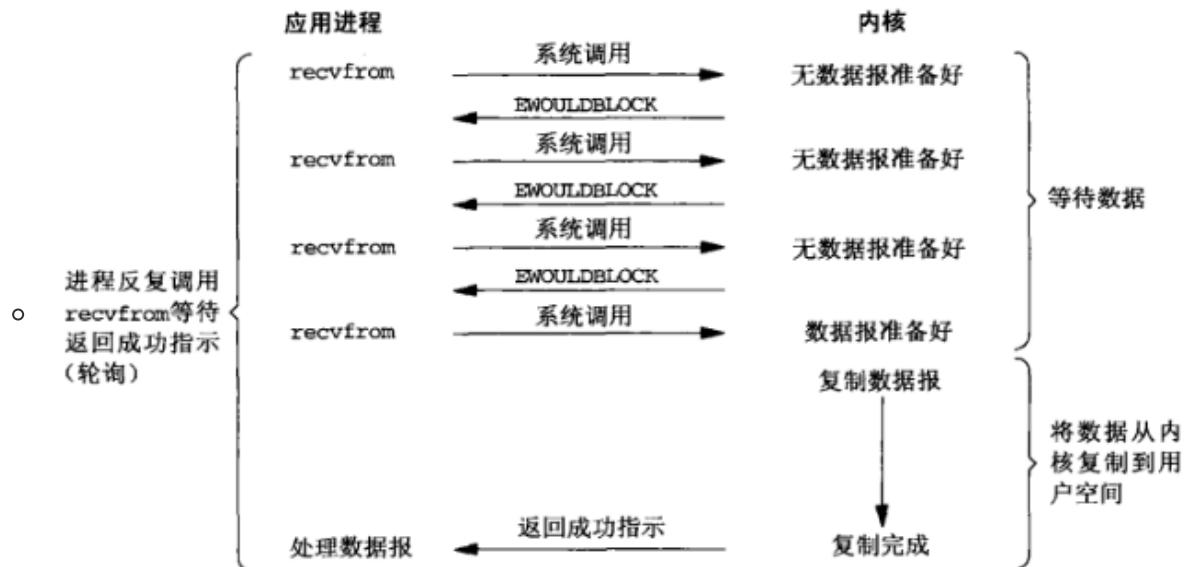


图6-2 非阻塞式I/O模型

- I/O复用模型

- 调用select或poll，阻塞在这两个系统调用中的某一个之上，而不是阻塞在真正的I/O系统调用上

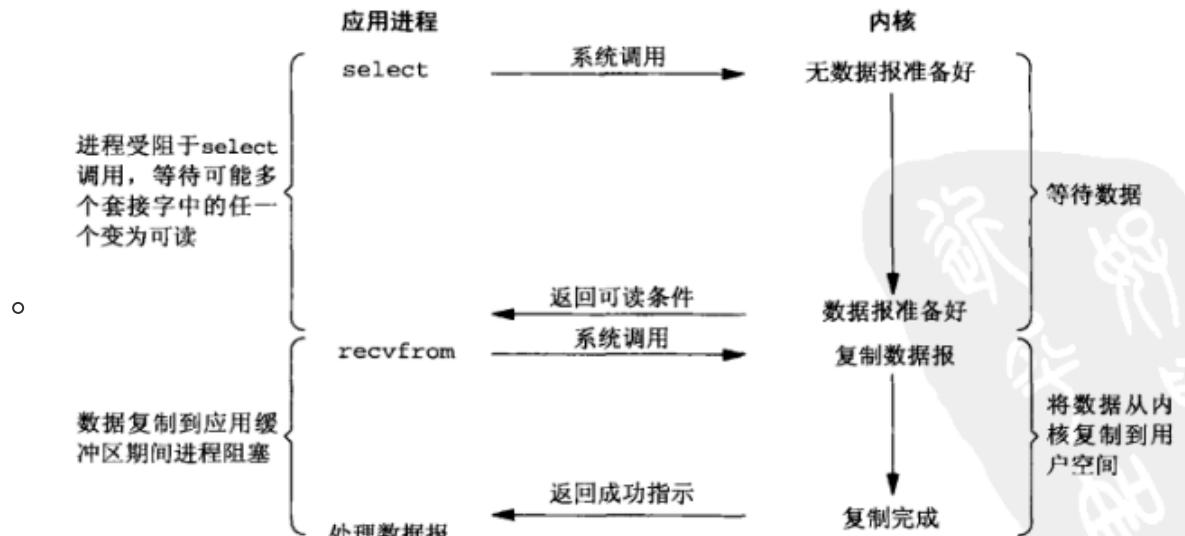


图6-3 I/O复用模型

- 使用select的优势在于我们可以等待多个描述符就绪
- 在多线程中使用阻塞式I/O，这样每个线程都可以自由地调用诸如recvfrom之类的阻塞式I/O系统调用
- 信号驱动式I/O模型

- 让内核在描述符就绪时发送SIGIO信号通知我们

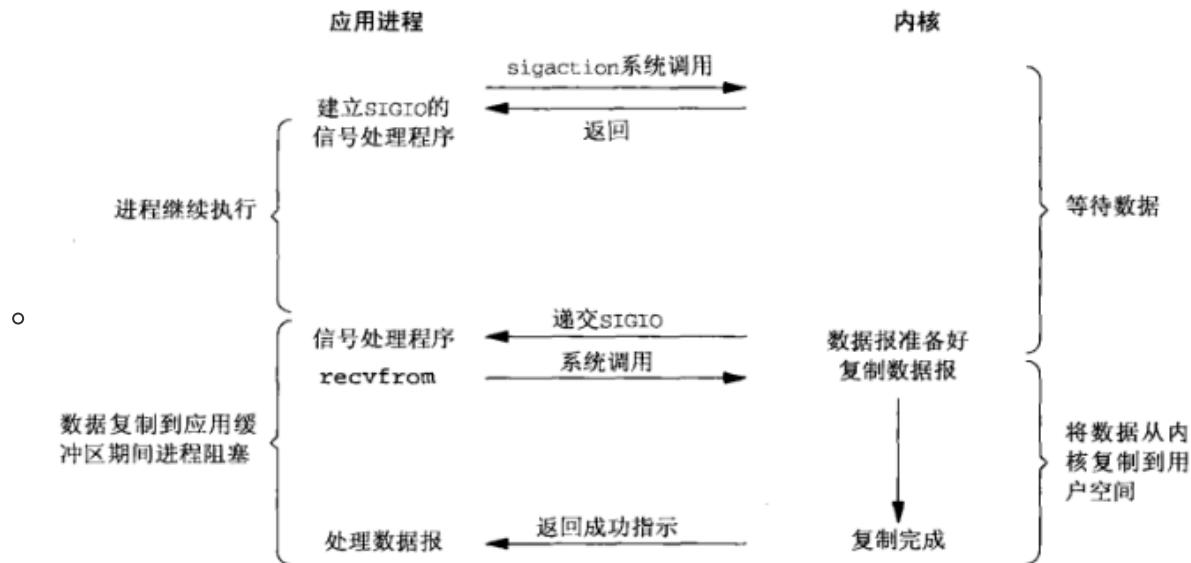


图6-4 信号驱动式I/O模型

- 等待数据报期间进程不被阻塞
- 异步I/O模型
 - 告知内核启动某个操作，并让内核在整个操作完成后通知我们

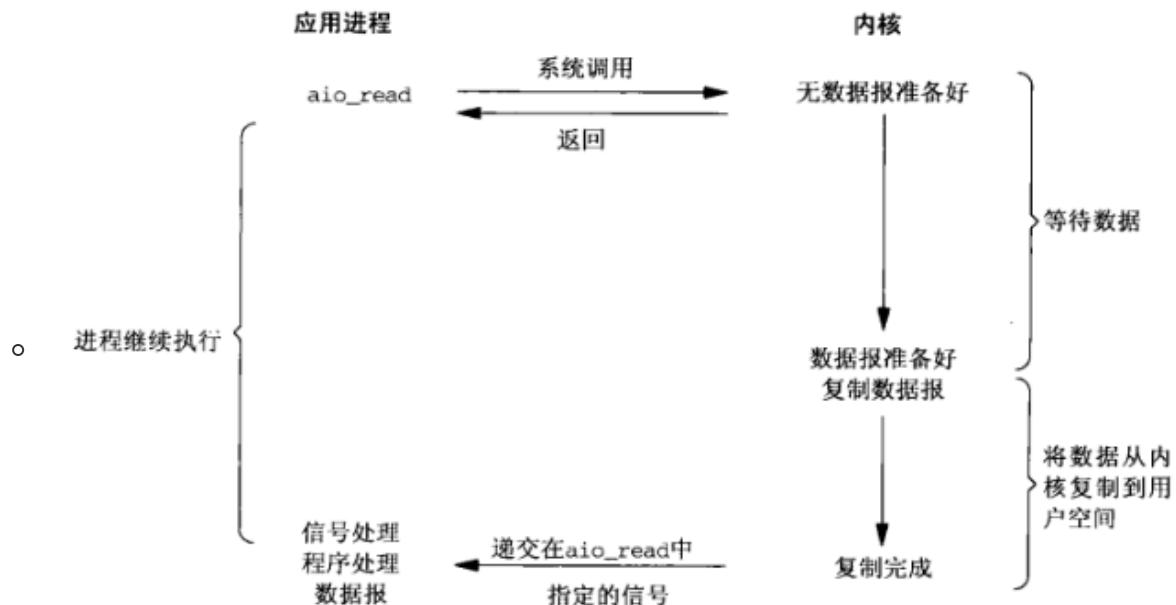


图6-5 异步I/O模型

- 调用aio_read函数，给内核传递描述符，缓冲区指针，大小，文件偏移，该系统调用立即返回，而且在等待I/O完成期间，进程不被阻塞
- 各种I/O模型的比较

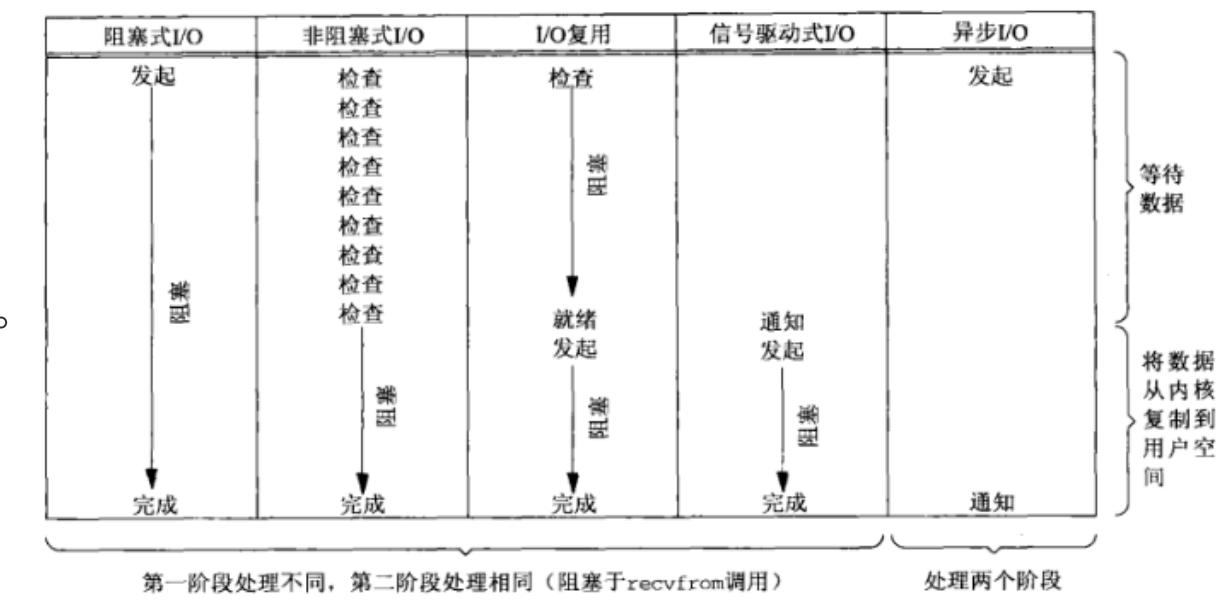


图6-6 5种I/O模型的比较

- 同步I/O和异步I/O
 - 同步I/O 导致请求进程阻塞，直到I/O操作完成， 前四种都为同步I/O
 - 异步I/O不导致请求进程阻塞

6.3 select函数

- 该函数允许进程指示内核等待多个事件中的任何一个发生，并且只有在一个或多个事件发生或经历一段指定的时间后才唤醒它
- 任何描述符都可以使用select来测试
- int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval * timeout)
- timeout参数

这个参数有以下三种可能。

- (1) 永远等待下去：仅在有一个描述符准备好I/O时才返回。为此，我们把该参数设置为空指针。
- (2) 等待一段固定时间：在有一个描述符准备好I/O时返回，但是不超过由该参数所指向的timeval结构中指定的秒数和微秒数。
- (3) 根本不等待：检查描述符后立即返回，这称为轮询（polling）。为此，该参数必须指向一个timeval结构，而且其中的定时器值（由该结构指定的秒数和微秒数）必须为0。

前两种情形的等待通常会被进程在等待期间捕获的信号中断，并从信号处理函数返回。

- 中间的三个参数readset、writeset和exceptset指定我们要让内核测试读、写和异常条件的描述符。目前支持的异常条件只有两个：
- (1) 某个套接字的带外数据的到达，我们将在第24章中详细讲述这个异常条件；
- (2) 某个已置为分组模式的伪终端存在可从其主端读取的控制状态信息，本书不讨论伪终端。

```

• void FD_ZERO(fd_set *fdset);           /* clear all bits in fdset */
void FD_SET(int fd, fd_set *fdset);       /* turn on the bit for fd in fdset */
void FD_CLR(int fd, fd_set *fdset);        /* turn off the bit for fd in fdset */
int FD_ISSET(int fd, fd_set *fdset);       /* is the bit for fd on in fdset ? */

```

- 分配一个fd_set数据类型的描述符集，并用这些宏设置或测试该集合中的每一位

- 举个例子，以下代码用于定义一个fd_set类型的变量，然后打开描述符1、4和5的对应位：

```
fd_set rset;
FD_ZERO(&rset);           /* initialize the set: all bits off */
FD_SET(1, &rset);         /* turn on bit for fd 1 */
FD_SET(4, &rset);         /* turn on bit for fd 4 */
FD_SET(5, &rset);         /* turn on bit for fd 5 */
```

- 这三个参数都是值-结果参数，该函数返回时，结果将指示哪些描述符已就绪，使用FD_ISSET宏来测试fd_set数据类型中的描述符，任何与未就绪描述符对应的位返回时均清成0
- 描述符就绪条件
 - 一个套接字准备读
 - a) 该套接字接收缓冲区中的数据字节数大于等于套接字接收缓冲区低水位标记的当前大小。对这样的套接字执行读操作不会阻塞并将返回一个大于0的值（也就是返回准备好读入的数据）。我们可以使用SO_RCVLOWAT套接字选项设置该套接字的低水位标记。对于TCP和UDP套接字而言，其默认值为1。
 - b) 该连接的读半部关闭（也就是接收了FIN的TCP连接）。对这样的套接字的读操作将不阻塞并返回0（也就是返回EOF）。
 - c) 该套接字是一个监听套接字且已完成的连接数不为0。对这样的套接字的accept通常不会阻塞，不过我们将在15.6节讲解accept可能阻塞的一种时序条件。
 - d) 其上有一个套接字错误待处理。对这样的套接字的读操作将不阻塞并返回-1（也就是返回一个错误），同时把errno设置成确切的错误条件。这些待处理错误（pending error）也可以通过指定SO_ERROR套接字选项调用getsockopt获取并清除。
 - 一个套接字准备写
 - a) 该套接字发送缓冲区中的可用空间字节数大于等于套接字发送缓冲区低水位标记的当前大小，并且或者该套接字已连接，或者该套接字不需要连接（如UDP套接字）。这意味着如果我们把这样的套接字设置成非阻塞（第16章），写操作将不阻塞并返回一个正值（例如由传输层接受的字节数）。我们可以使用SO_SNDLOWAT套接字选项来设置该套接字的低水位标记。对于TCP和UDP套接字而言，其默认值通常为2048。
 - b) 该连接的写半部关闭。对这样的套接字的写操作将产生SIGPIPE信号（5.12节）。
 - c) 使用非阻塞式connect的套接字已建立连接，或者connect已经以失败告终。
 - d) 其上有一个套接字错误待处理。对这样的套接字的写操作将不阻塞并返回-1（也就是返回一个错误），同时把errno设置成确切的错误条件。这些待处理的错误也可以通过指定SO_ERROR套接字选项调用getsockopt获取并清除。
 - 如果一个套接字存在带外数据或者仍处于带外标记，那么它有异常条件待处理
 - 当某个套接字上发生错误时，它将由select标记为既可读又可写
 - 接收低水位标记和发送低水位标记：允许应用进程控制在select返回可读或可写条件之前有多少数据可读或有多大空间可用于写
 -

条件	可读吗？	可写吗？	异常吗？
有数据可读	•		
关闭连接的读一半	•		
给监听套接口准备好新连接	•		
有可用于写的空间		•	
关闭连接的写一半		•	
待处理错误	•	•	
TCP带外数据			•

图6-7 select返回某个套接字就绪的条件小结

- select的最大描述符数
 - 增大描述符集大小的唯一方法是先增大FD_SETSIZE的值，再重新编译内核，不重新编译内核而改变其值是不够的

6.4 str_cli函数

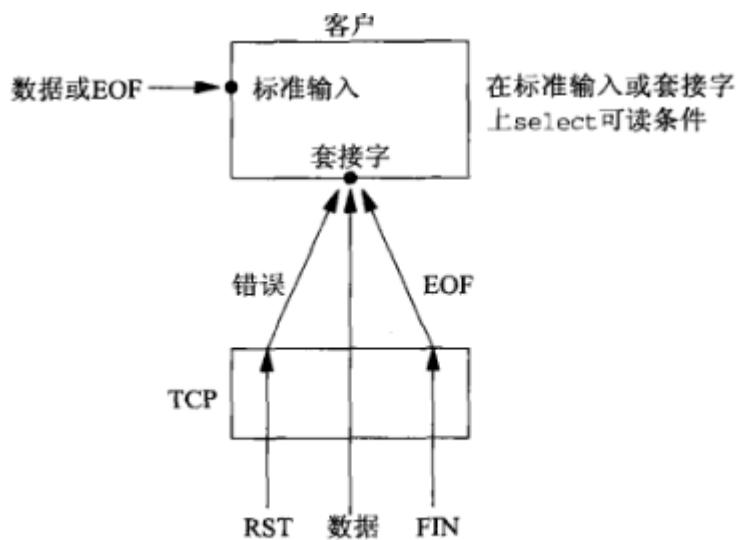


图6-8 str_cli函数中由select处理的各种条件

客户的套接字上的三个条件处理如下。

- (1) 如果对端TCP发送数据，那么该套接字变为可读，并且read返回一个大于0的值（即读入数据的字节数）。
- (2) 如果对端TCP发送一个FIN（对端进程终止），那么该套接字变为可读，并且read返回0（EOF）。
- (3) 如果对端TCP发送一个RST（对端主机崩溃并重新启动），那么该套接字变为可读，并且read返回-1，而errno中含有确切的错误码。

```

1 #include "unp.h"
2
3 void
4 str_cli(FILE *fp, int sockfd)
5 {
6     int maxfdp1;
7     fd_set rset;
8     char sendline[MAXLINE], recvline[MAXLINE];
9
10    FD_ZERO(&rset);
11    for ( ; ; ) {
12        FD_SET(fileno(fp), &rset);
13        FD_SET(sockfd, &rset);
14        maxfdp1 = max(fileno(fp), sockfd) + 1;
15        Select(maxfdp1, &rset, NULL, NULL, NULL);
16
17        if (FD_ISSET(sockfd, &rset)) { /* socket is readable */
18            if (Readline(sockfd, recvline, MAXLINE) == 0)
19                err_quit("str_cli: server terminated prematurely");
20            Fputs(recvline, stdout);
21
22            if (FD_ISSET(fileno(fp), &rset)) { /* input is readable */
23                if (Fgets(sendline, MAXLINE, fp) == NULL)
24                    return; /* all done */
25                Writen(sockfd, sendline, strlen(sendline));
26            }
27        }
28    }
29 }

```

select/strcli

图6-9 使用select的str_cli函数的实现（在图6-13中改进）

6.5 批量输入

- 在批量方式下，标准输入中的EOF并不意味着我们同时也完成了从套接字的读入，可能仍有请求在去往服务器的路上，或者仍有应答在返回客户的路上
- 需要一种关闭TCP连接其中一半的方法，给服务器发送一个FIN，告诉它我们已经完成了数据发送，但是仍然保持套接字描述符打开以便读取，由shutdown函数完成
- 为了提升性能而引入缓冲机制增加了网络应用程序的复杂性
 - fgets读取输入，使可用的文本输入行被读入到stdio所用的缓冲区中，然而fgets只返回一行，其余的仍在stdio缓冲区中
 - 混合使用select和stdio容易出错

一般地说，为提升性能而引入缓冲机制增加了网络应用程序的复杂性，图6-9所示的代码就遭受这种复杂性之害。考虑有多个来自标准输入的文本输入行可用的情况。select将使第20行代码用fgets读取输入，这又转而使已可用的文本输入行被读入到stdio所用的缓冲区中。然而fgets只返回其中第一行，其余输入行仍在stdio缓冲区中。第22行代码把fgets返回的单个输入行写给服务器，随后select再次被调用以等待新的工作，而不管stdio缓冲区中还有额外的输入行待消费。究其原因在于select不知道stdio使用了缓冲区——它只是从read系统调用的角度指出是否有数据可读，而不是从fgets之类调用的角度考虑。基于上述原因，混合使用stdio和select被认为是非常容易犯错误的，在这样做时必须极其小心。

同样的问题存在于图6-9的readline调用中。这回select不可见的数据不是隐藏在stdio缓冲区中，而是隐藏在readline自己的缓冲区中。回顾3.9节我们提供的一个可以看到readline缓冲区的函数，因此可能的解决办法之一是修改我们的代码，在调用select之前使用那个函数，以查看是否存在已经读入而尚未消费的数据。然而为了处理readline缓冲区中既可能有不完整的输入行（意味着我们需要继续读入），也可能有一个或多个完整的输入行（这些行我们可以直接消费）这两种情况而引入的复杂性会迅速增长到难以控制的地步。

6.6 shutdown函数

(1) close把描述符的引用计数减1，仅在该计数变为0时才关闭套接字。我们已在4.8节讨论过这一点。使用shutdown可以不管引用计数就激发TCP的正常连接终止序列（图2-5中由FIN开始的4个分节）。

(2) close终止读和写两个方向的数据传送。既然TCP连接是全双工的，有时候我们需要告知对端我们已经完成了数据发送，即使对端仍有数据要发送给我们。这就是我们在前一节中遇到的str_cli函数在批量输入时的情况。图6-12展示了这样的情况下典型的函数调用。

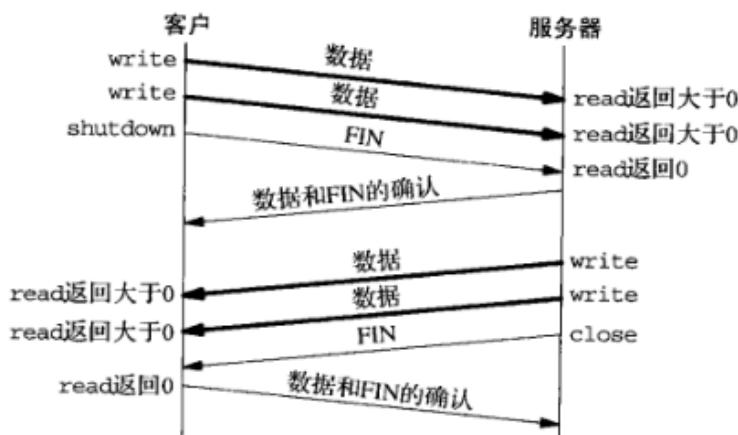


图6-12 调用shutdown关闭一半TCP连接

- int shutdown(int sockfd, int howto)
 - howto的值
 - SHUT_RD 关闭连接的读这一半
 - SHUT_WR 关闭连接的写这一半
 - SHUT_RDWR 连接的读半部和写半部都关闭

6.7 str_cli函数

```

#include     "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int         maxfdp1, stdineof;
    fd_set     rset;
    char       buf[MAXLINE];
    int        n;

```

```

stdineof = 0;
FD_ZERO(&rset);
for ( ; ; ) {
    if (stdineof == 0)
        FD_SET(fileno(fp), &rset);
    FD_SET(sockfd, &rset);
    maxfdp1 = max(fileno(fp), sockfd) + 1;
    Select(maxfdp1, &rset, NULL, NULL, NULL);

    if (FD_ISSET(sockfd, &rset)) { /* socket is readable */
        if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
            if (stdineof == 1)
                return; /* normal termination */
            else
                err_quit("str_cli: server terminated prematurely");
        }

        Write(fileno(stdout), buf, n);
    }

    if (FD_ISSET(fileno(fp), &rset)) { /* input is readable */
        if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
            stdineof = 1;
            Shutdown(sockfd, SHUT_WR); /* send FIN */
            FD_CLR(fileno(fp), &rset);
            continue;
        }

        Writen(sockfd, buf, n);
    }
}
}

```

6.8 TCP回射服务器程序

- 重新成使用select来处理任意个客户的单进程程序
- 接收到FIN 或 EOF, read返回的值是0

```

/* include fig01 */
#include     "unp.h"

int
main(int argc, char **argv)
{
    int                 i, maxi, maxfd, listenfd, connfd, sockfd;
    int                 nready, client[FD_SETSIZE];
    ssize_t              n;
    fd_set               rset, allset;
    char                buf[MAXLINE];
    socklen_t            clilen;

```

```

struct sockaddr_in cliaddr, servaddr;

listenfd = Socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

Listen(listenfd, LISTENQ);

maxfd = listenfd;           /* initialize */
maxi = -1;                  /* index into client[] array */
for (i = 0; i < FD_SETSIZE; i++)
    client[i] = -1;          /* -1 indicates available entry */
FD_ZERO(&allset);
FD_SET(listenfd, &allset);
/* end fig01 */

/* include fig02 */
for ( ; ; ) {
    rset = allset;           /* structure assignment */
    nready = Select(maxfd+1, &rset, NULL, NULL, NULL);

    if (FD_ISSET(listenfd, &rset)) {      /* new client connection */
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
#ifndef NOTDEF
        printf("new client: %s, port %d\n",
               Inet_ntop(AF_INET, &cliaddr.sin_addr, 4, NULL),
               ntohs(cliaddr.sin_port));
#endif
    }
    for (i = 0; i < FD_SETSIZE; i++)
        if (client[i] < 0) {
            client[i] = connfd; /* save descriptor */
            break;
        }
    if (i == FD_SETSIZE)
        err_quit("too many clients");

    FD_SET(connfd, &allset); /* add new descriptor to set */
    if (connfd > maxfd)
        maxfd = connfd;        /* for select */
    if (i > maxi)
        maxi = i;              /* max index in client[] array */

    if (--nready <= 0)
        continue;             /* no more readable descriptors */
}

```

```

        for (i = 0; i <= maxi; i++) { /* check all clients for data */
            if ((sockfd = client[i]) < 0)
                continue;
            if (FD_ISSET(sockfd, &rset)) {
                if ((n = Read(sockfd, buf, MAXLINE)) == 0) {
                    /* connection closed by client */
                    Close(sockfd);
                    FD_CLR(sockfd, &allset);
                    client[i] = -1;
                } else
                    Writen(sockfd, buf, n);

                if (--nready <= 0)
                    break; /* no more readable descriptors */
            }
        }
    }
}

/* end fig02 */

```

- 当一个服务器在处理多个客户时，它不能阻塞于只与单个客户相关的函数调用，否则可能导致服务器被挂起，拒绝为所有其他客户提供服务，这就是拒绝服务型攻击。可能的解决办法
 - 使用非阻塞式I/O
 - 让每个客户由单独的控制线程提供服务
 - 对I/O操作设置一个超时

6.9 pselect函数

- 使用timespec结构
- 增加了第六个参数：一个指向信号掩码的指针，该参数允许程序先禁止递交某些信号，再测试由这些当前被禁止信号的信号处理函数设置的全局变量，然后调用pselect，重新设置信号掩码

```

sigset(SIG_BLOCK, &newmask, &oldmask, &zeromask);
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* block SIGINT */
if (intr_flag)
    handle_intr(); /* handle the signal */
if ((nready = pselect(..., &zeromask)) < 0) {
    if (errno == EINTR) {
        if (intr_flag)
            handle_intr();
    }
    ...
}

```

在测试intr_flag变量之前，我们阻塞SIGINT。当pselect被调用时，它先以空集（即zeromask）替代进程的信号掩码，再检查描述符，并可能进入睡眠。然而当pselect函数返回时，进程的信号掩码又被重置为调用pselect之前的值（即SIGINT被阻塞）。

6.10 poll函数

- poll函数提供的功能与select类似，不过在处理流设备时，它能够提供额外的信息
- int poll(struct pollfd* fdarray, unsigned long nfds, int timeout);

```

struct pollfd {
    int      fd;          /* descriptor to check */
    short    events;       /* events of interest on fd */
    short    revents;      /* events that occurred on fd */
};


```

常 值	作为events的输入吗?	作为revents的结果吗?	说 明
POLLIN	•	•	普通或优先级带数据可读
POLLRDNORM	•	•	普通数据可读
POLLRDBAND	•	•	优先级带数据可读
POLLPRI	•	•	高优先级数据可读
POLLOUT	•	•	普通数据可写
POLLWRNORM	•	•	普通数据可写
POLLWRBAND	•	•	优先级带数据可写
POLLERR		•	发生错误
POLLHUP		•	发生挂起
POLLNVAL		•	描述符不是一个打开的文件

图6-23 poll函数的输入events和返回revents

- poll识别三类数据: 普通、优先级带 高优先级 出自于基于流的实现

就TCP和UDP套接字而言,以下条件引起poll返回特定的revent。不幸的是,POSIX在其poll的定义中留了许多空洞(也就是说有多种方法可返回相同的条件)。

- 所有正规TCP数据和所有UDP数据都被认为是普通数据。
 - TCP的带外数据(第24章)被认为是优先级带数据。
 - 当TCP连接的读半部关闭时(譬如收到了一个来自对端的FIN),也被认为是普通数据,随后的读操作将返回0。
 - TCP连接存在错误既可认为是普通数据,也可认为是错误(POLLERR)。无论哪种情况,随后的读操作将返回-1,并把errno设置成合适的值。这可用于处理诸如接收到RST或发生超时等条件。
 - 在监听套接字上有新的连接可用既可认为是普通数据,也可认为是优先级数据。大多数实现视之为普通数据。
 - 非阻塞式connect的完成被认为是使相应套接字可写。
- 结构数组中元素的个数是由nfds参数指定。

6.11 TCP回射服务器程序

```

/* include fig01 */
#include    "unp.h"
#include    <limits.h>      /* for OPEN_MAX */

int
main(int argc, char **argv)
{
    int                  i, maxi, listenfd, connfd, sockfd;
    int                  nready;
    ssize_t               n;
    char                buf[MAXLINE];
    socklen_t             clilen;

```

```

struct pollfd      client[OPEN_MAX];
struct sockaddr_in cliaddr, servaddr;

listenfd = Socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(SERV_PORT);

Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

Listen(listenfd, LISTENQ);

client[0].fd = listenfd;
client[0].events = POLLRDNORM;
for (i = 1; i < OPEN_MAX; i++)
    client[i].fd = -1;           /* -1 indicates available entry */
maxi = 0;                      /* max index into client[] array */
/* end fig01 */

/* include fig02 */
for ( ; ; ) {
    nready = Poll(client, maxi+1, INFTIM);

    if (client[0].revents & POLLRDNORM) { /* new client connection */
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
#ifndef NOTDEF
        printf("new client: %s\n", Sock_ntop((SA *) &cliaddr, clilen));
#endif

        for (i = 1; i < OPEN_MAX; i++)
            if (client[i].fd < 0) {
                client[i].fd = connfd; /* save descriptor */
                break;
            }
        if (i == OPEN_MAX)
            err_quit("too many clients");

        client[i].events = POLLRDNORM;
        if (i > maxi)
            maxi = i;             /* max index in client[] array */

        if (--nready <= 0)
            continue;             /* no more readable descriptors */
    }

    for (i = 1; i <= maxi; i++) { /* check all clients for data */
        if ( (sockfd = client[i].fd) < 0)
            continue;
        if (client[i].revents & (POLLRDNORM | POLLERR)) {
            if ( (n = read(sockfd, buf, MAXLINE)) < 0) {

```

```

        if (errno == ECONNRESET) {
            /* connection reset by client */
#ifndef NOTDEF
                printf("client[%d] aborted connection\n", i);
#endif
            Close(sockfd);
            client[i].fd = -1;
        } else
            err_sys("read error");
    } else if (n == 0) {
        /* connection closed by client */
#ifndef NOTDEF
        printf("client[%d] closed connection\n", i);
#endif
        Close(sockfd);
        client[i].fd = -1;
    } else
        Writen(sockfd, buf, n);

    if (--nready <= 0)
        break;           /* no more readable descriptors */
}
}
}
}

/* end fig02 */

```

第七章 套接字选项

- 获取和设置影响套接字的选项
 - getsockopt和setsockopt函数
 - fnct1函数
 - icot1函数

7.2 getsockopt和setsockopt函数

- 仅用于套接字
- 套接字选项分为两大基本类型
 - 一是启用或禁止某个特性的二元选项(标志选项)
 - 二是取得并返回我们可以设置或检查的特定值的选项(值选项)
-

<i>level</i> (级别)	<i>optname</i> (选项名)	<i>get</i>	<i>set</i>	说 明	标 志	数据类型
SOL_SOCKET	SO_BROADCAST SO_DEBUG SO_DONTROUTE SO_ERROR SO_KEEPALIVE SO_LINGER SO_OOBINLINE SO_RCVBUF SO_SNDBUF SO_RCVLOWAT SO SNDLOWAT SO_RCVTIMEO SO_SNDTIMEO SO_REUSEADDR SO_REUSEPORT SO_TYPE SO_USELOOPBACK	• • • • • • • • • • • • • • • • • • • •	• • • • • • • • • • • • • • • • • • •	允许发送广播数据报 开启调试跟踪 绕过外出路由表查询 获取待处理错误并清除 周期性测试连接是否存活 若有数据待发送则延迟关闭 让接收到的带外数据继续在线留存 接收缓冲区大小 发送缓冲区大小 接收缓冲区低水位标记 发送缓冲区低水位标记 接收超时 发送超时 允许重用本地地址 允许重用本地端口 取得套接字类型 路由套接字取得所发送数据的副本	• • • • • linger() • • • • • • • timeval() timeval() • • • • •	int int int int int linger() int int int int int int timeval() timeval() int int int int
IPPROTO_IP	IP_HDRINCL IP_OPTIONS IP_RECVDSTADDR IP_RECVIF IP_TOS IP_TTL IP_MULTICAST_IF IP_MULTICAST_TTL IP_MULTICAST_LOOP IP_ADD_MEMBERSHIP IP_DROP_MEMBERSHIP IP_BLOCK_SOURCE IP_UNBLOCK_SOURCE IP_ADD_SOURCE_MEMBERSHIP IP_DROP_SOURCE_MEMBERSHIP	• • • • • • • • • • • • • • • • • • •	• • • • • • • • • • • • • • • • • •	随数据包含的IP首部 IP首部选项 返回目的IP地址 返回接收接口索引 服务类型和优先权 存活时间 指定外出接口 指定外出TTL 指定是否环回 加入多播组 离开多播组 阻塞多播源 开通多播源 加入源特定多播组 离开源特定多播组	• (见正文) • • • • in_addr() u_char u_char ip_mreq() ip_mreq() ip_mreq_source() ip_mreq_source() ip_mreq_source() ip_mreq_source()	int (见正文) int int int int
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	指定待传递的ICMPv6消息类型	icmp6_filter()	
IPPROTO_IPV6	IPV6_CHECKSUM IPV6_DONTFRAG IPV6_NEXTHOP IPV6_PATHMTU IPV6_RECVDSTOPTS IPV6_RECVHOPLIMIT IPV6_RECVHOPOPTS IPV6_RECVPATHMTU IPV6_RECVPKTINFO IPV6_RECVRTHDR IPV6_RECVTCLASS IPV6_UNICAST_HOPS IPV6_USE_MIN_MTU IPV6_V6ONLY IPV6_XXX	• • • • • • • • • • • • • • • • • •	• • • • • • • • • • • • • • • • • •	用于原始套接字的校验和字段偏移 丢弃大的分组而非将其分片 指定下一跳地址 获取当前路径MTU 接收目的地选项 接收单播跳限 接收步跳选项 接收路径MTU 接收分组信息 接收源路径 接收流通类别 默认单播跳限 使用最小MTU 禁止v4兼容 粘附性辅助数据	int int sockaddr_in6() ip6_mtuinfo() int int int int int int int int int int (见正文)	int int sockaddr_in6() ip6_mtuinfo() int int int int int int int int int int (见正文)
IPPROTO_IP或 IPPROTO_IPV6	MCAST_JOIN_GROUP MCAST_LEAVE_GROUP MCAST_BLOCK_SOURCE MCAST_UNBLOCK_SOURCE MCAST_JOIN_SOURCE_GROUP MCAST_LEAVE_SOURCE_GROUP		•	指定外出接口 指定外出跳限 指定是否环回 加入多播组 离开多播组	group_req() group_source_req() group_source_req() group_source_req() group_source_req() group_source_req()	group_req() group_source_req() group_source_req() group_source_req() group_source_req() group_source_req()

图7-1 套接字层和IP层的套接字选项汇总

<i>level</i> (级别)	<i>optname</i> (选项名)	<i>get</i>	<i>set</i>	说 明	标志	数据类型
IPPROTO_TCP	TCP_MAXSEG	•	•	TCP最大分节大小	•	int
	TCP_NODELAY	•	•	禁止Nagle算法		int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	适配层指示	•	sctp_setadaption()
	SCTP_ASSOCINFO	†	•	检查并设置关联信息		sctp_assocparams()
	SCTP_AUTOCLOSE	•	•	自动关闭操作		int
	SCTP_DEFAULT_SEND_PARAM	•	•	默认发送参数		sctp_sndrcvinfo()
	SCTP_DISABLE_FRAGMENTS	•	•	SCTP分片		int
	SCTP_EVENTS	•	•	感兴趣事件的通知		sctp_event_subscribe()
	SCTP_GET_PEER_ADDR_INFO	†	•	获取对端地址状态		sctp_paddrinfo()
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	映射的v4地址		int
	SCTP_INITMSG	•	•	默认的INIT参数		sctp_initmsg()
	SCTP_MAXBURST	•	•	最大猝发大小		int
	SCTP_MAXSEG	•	•	最大分片大小		int
	SCTP_NODELAY	•	•	禁止Nagle算法		int
	SCTP_PEER_ADDR_PARAMS	†	•	对端地址参数		sctp_paddrparams()
	SCTP_PRIMARY_ADDR	†	•	主目的地址		sctp_setprim()
	SCTP_RTOINFO	†	•	RTO信息		sctp_rtoinfo()
	SCTP_SFT_PEER_PRIMARY_ADDR	†	•	对端的主目的地址		sctp_setpeerprim()
	SCTP_STATUS	†	•	获取关联状态		sctp_status()

图7-2 传输层的套接字选项汇总

7.3 检查选项是否受支持并获取默认值

- 创建一个用于测试选项的套接字，测试套接字层，TCP层和IPv4层套接字选项所用的是一个IPv4的TCP套接字
- 测试IPv6层套接字选项所用的是一个IPv6的TCP套接字，测试SCTP层套接字选项所用的上一个IPv4的SCTP套接字

7.4 套接字状态

下面的套接字选项是由TCP已连接套接字从监听套接字继承来的 (TCPv2第462~463页)：

- SO_DEBUG、SO_DONTROUTE、SO_KEEPALIVE、SO_LINGER、SO_OOBINLINE、SO_RCVBUF、SO_RCVLOWAT、SO_SNDBUF、SO SNDLOWAT、TCP_MAXSEG和TCP_NODELAY。这对TCP是很重要的，因为accept一直要到TCP层完成三路握手后才会给服务器返回已连接套接字。如果想在三路握手完成时确保这些套接字选项中的某一个是给已连接套接字设置的，那么我们必须先给监听套接字设置该选项。

7.5 通用套接字选项

- 这些选项是协议无关的
- SO_BROADCAST
 - 本选项开启或禁止进程发送广播消息
- SO_DEBUG
 - 仅由TCP支持，当给一个TCP套接字开启本选项时，内核将为TCP在该套接字发送和接收的所有分组保留详细跟踪信息，保存在内核的某个环形缓冲区汇中
- SO_DONTROUTE
 - 规定外出的分组将绕过底层协议的正常路由机制
- SO_ERROR
 - 当一个套接字发生错误时，so_error的变量，待处理错误，当进程调用read且没有数据返回时，如果so_error为非0值，那么read返回-1且errno被置为so_error的值
 - 一个套接字上出现的待处理错误一旦返回给用户进程，它的so_error就得复位为0

- SO_KEEPALIVE

- 保持存活选项

给一个TCP套接字设置保持存活(keep-alive)选项后,如果2小时内该套接字的任一方向上都没有数据交换,TCP就自动给对端发送一个保持存活探测分节(keep-alive probe)。这是一个对端必须响应的TCP分节,它会导致以下三种情况之一。

- (1) 对端以期望的ACK响应。应用进程得不到通知(因为一切正常)。在又经过仍无动静的2小时后,TCP将发出另一个探测分节。

(2) 对端以RST响应,它告知本端TCP:对端已崩溃且已重新启动。该套接字的待处理错误被置为ECONNRESET,套接字本身则被关闭。

(3) 对端对保持存活探测分节没有任何响应。源自Berkeley的TCP将另外发送8个探测分节,两两相隔75秒,试图得到一个响应。TCP在发出第一个探测分节后11分15秒内若没有得到任何响应则放弃。

- 一般由服务器使用,服务器花大部分时间阻塞在等待穿越TCP连接的输入上
- 保持存活选项将检测出半开连接并终止它们
- 应用进程自己也可以实现超时
- 检测各种TCP条件的方法

情形	对端进程崩溃	对端主机崩溃	对端主机不可达
本端TCP正主动发送数据	对端TCP发送一个FIN,这通过使用select判断可读条件立即能检测出来。如果本端TCP发送另外一个分节,对端TCP就以RST响应。如果在本端TCP收到RST之后应用进程仍试图写套接字,我们的套接字实现就给该进程发送一个SIGPIPE信号	本端TCP将超时,且套接字的待处理错误被设置为ETIMEDOUT	本端TCP将超时,且套接字的待处理错误被设置为EHOSTUNREACH
本端TCP正主动接收数据	对端TCP将发送一个FIN,我们将把它作为一个(可能是过早的)EOF读入	我们将停止接收数据	我们将停止接收数据
连接空闲,保持存活选项已设置	对端TCP发送一个FIN,这通过使用select判断可读条件立即能检测出来	在毫无动静2小时后,发送9个保持存活探测分节,然后套接字的待处理错误被设置为ETIMEDOUT	在毫无动静2小时后,发送9个保持存活探测分节,然后套接字的待处理错误被设置为EHOSTUNREACH
连接空闲,保持存活选项未设置	对端TCP发送一个FIN,这通过使用select判断可读条件立即能检测出来	(无)	(无)

图7-6 检测各种TCP条件的方法

- SO_LINGER

- 指定close函数对面向连接的协议如何操作,默认操作是close立即返回,但是如果有数据残留在套接字发送缓冲区,系统将试着把这些数据发送给对端
- struct linger{ int l_onoff; int l_linger};
- 应用进程检查close的返回值是非常重要的,因为如果在数据发送完并被确认前延迟时间到,close将返回EWOULDBLOCK错误,套接字发送缓冲区中的任何残留数据都被丢弃

(1) 如果l_onoff为0, 那么关闭本选项。l_linger的值被忽略, 先前讨论的TCP默认设置生效, 即close立即返回。

(2) 如果l_onoff为非0值且l_linger为0, 那么当close某个连接时TCP将中止该连接 (TCPv2第1019~1020页)。这就是说TCP将丢弃保留在套接字发送缓冲区中的任何数据, 并发送一个RST给对端, 而没有通常的四分组连接终止序列 (2.6节)。我们将在图16-21中给出这样的一个例子。这么一来避免了TCP的TIME_WAIT状态, 然而存在以下可能性: 在2MSL秒内创建该连接的另一个化身, 导致来自刚被终止的连接上的旧的重复分节被不正确地递送到新的化身上 (2.7节)。

这种情形下SCTP也通过发送一个ABORT块给对端而中止性地关闭关联 ([Stewart and Xie 2001] 9.2节)。

偶尔张贴在USENET上的消息提倡使用本特性, 其目的是为了避免TIME_WAIT状态, 并且即使在跟某个服务器的众所周知端口的连接仍在使用的情况下也能重启其监听服务器。这么做万万不可, 它可能导致数据被破坏, 详情见RFC 1337 [Braden 1992a]。作为替代, 总是在服务器程序中调用bind前使用SO_REUSEADDR套接字选项, 我们马上会讲述到。TIME_WAIT状态是我们的朋友, 它是有助于我们的(也就是说, 它让旧的重复分节在网络中超时消失)。不要试图避免这个状态, 而是应该弄清楚它 (2.7节)。

个别环境下使用本特性执行中止性的关闭是合理的。例子之一是因试图向某个停滞的终端端口递送数据而可能永远滞留在CLOSE_WAIT状态的一个RS-232终端服务器, 要是它得到一个RST以丢弃待处理的数据, 它会适当地复位那个停滞的终端端口。

- (3) 如果l_onoff为非0值且l_linger也为非0值, 那么当套接字关闭时内核将拖延一段时

160 第7章 套接字选项

间。这就是说如果在套接字发送缓冲区中仍残留有数据, 那么进程将被投入睡眠, 直到(a)所有数据都已发送完且均被对方确认或(b)延滞时间到。如果套接字被设置为非阻塞型 (第16章), 那么它将不等待close完成, 即使延滞时间为非0也是如此。当使用SO_LINGER选项的这个特性时, 应用进程检查close的返回值是非常重要的, 因为如果在数据发送完并被确认前延滞时间到的话, close将返回EWOULDBLOCK错误, 且套接字发送缓冲区中的任何残留数据都被丢弃。

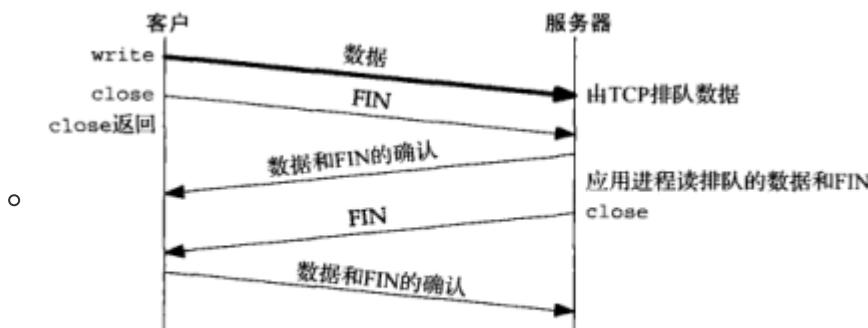


图7-7 close的默认操作: 立即返回

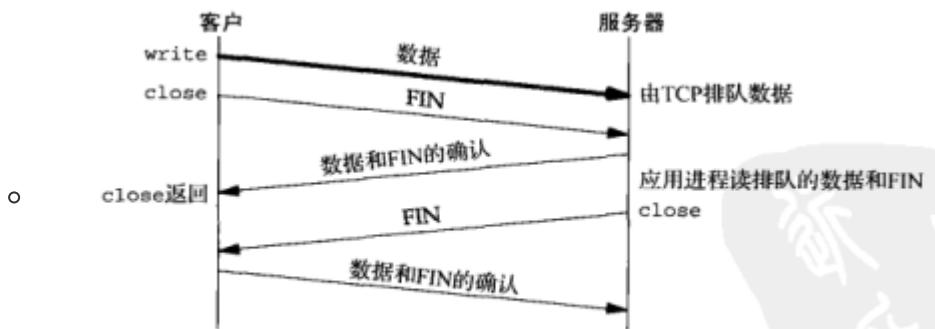


图7-8 设置SO_LINGER套接字选项且linger为正值时的close

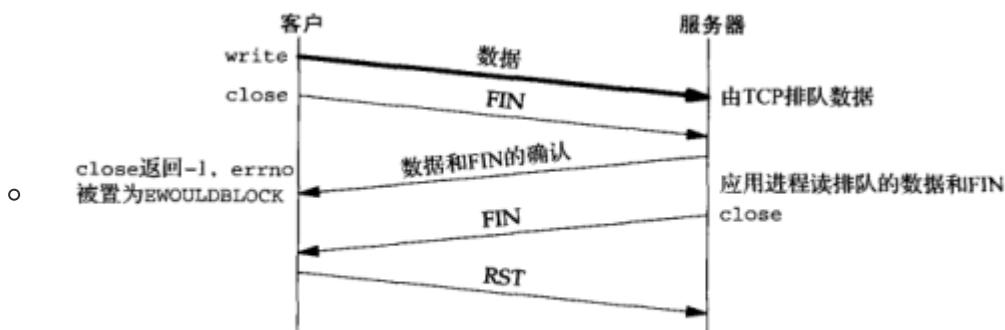


图7-9 设置SO_LINGER套接字选项且linger为偏小正值时的close

- 设置SO_LINGER套接字选项后，close的成功返回只是告诉我们先前发送的数据和FIN已由对端TCP确认，而不能告诉我们对端应用进程是否已读取数据
- 让客户知道服务器已读取数据的一个方法是改为调用shutdown，而不是调用close，并等待对端连接的当地端

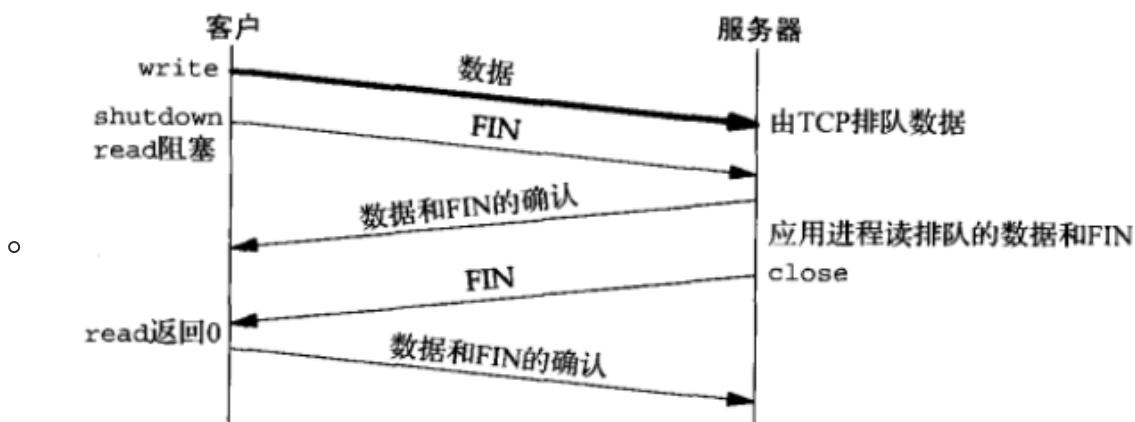


图7-10 用shutdown来获知对方已接收数据

- 比较本图与图7-7及图7-8我们看到，当关闭连接的本地端（客户端）时，根据所调用的函数（close或shutdown）以及是否设置了SO_LINGER套接字选项，可在以下3个不同的时机返回。
- (1) close立即返回，根本不等待（默认状况，图7-7）。
 - (2) close一直拖延到接收了对于客户端FIN的ACK才返回（图7-8）。
 - (3) 后跟一个read调用的shutdown一直等到接收了对端的FIN才返回（图7-10）。
 - 应用级ACK

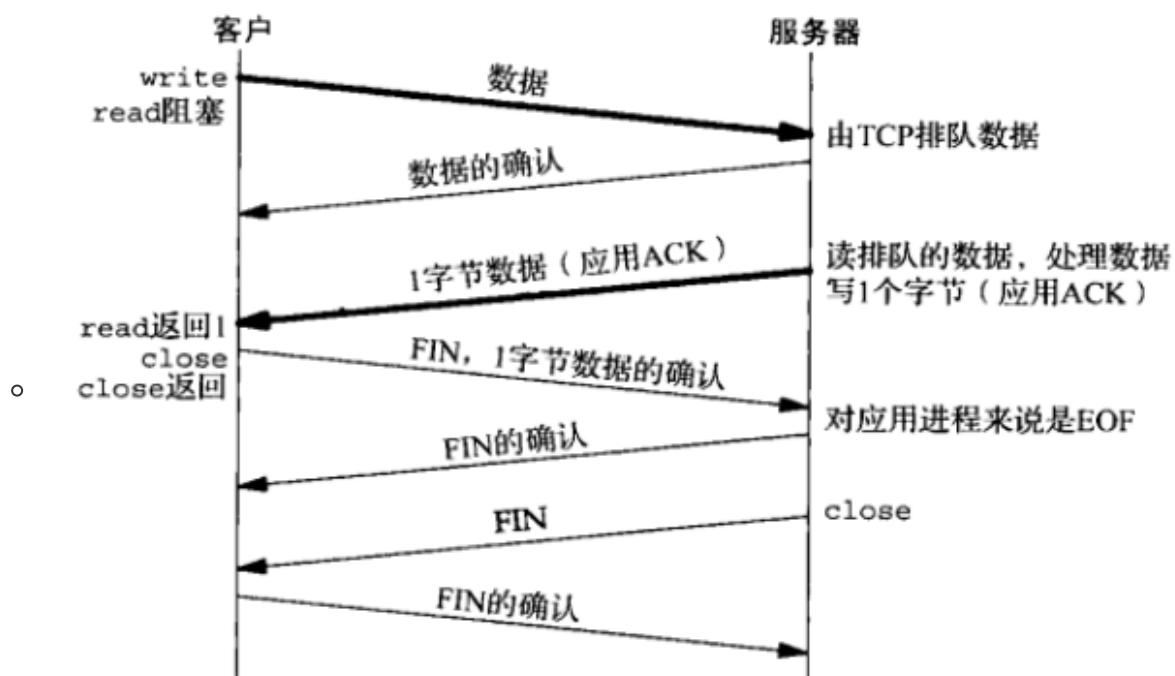


图7-11 应用ACK

函 数	说 明
shutdown, SHUT_RD	在套接字上不能再发出接收请求；进程仍可往套接字发送数据；套接字接收缓冲区中所有数据被丢弃；再接收到的任何数据由TCP丢弃（习题6.5）；对套接字发送缓冲区没有任何影响。
shutdown, SHUT_WR	在套接字上不能再发出发送请求；进程仍可从套接字接收数据；套接字发送缓冲区中的内容被发送到对端，后跟正常的TCP连接终止序列（即发送FIN）；对套接字接收缓冲区无任何影响。
close, l_onoff = 0 (默认情况)	在套接字上不能再发出发送或接收请求；套接字发送缓冲区中的内容被发送到对端。如果描述符引用计数变为0：在发送完发送缓冲区中的数据后，跟以正常的TCP连接终止序列（即发送FIN）；套接字接收缓冲区中内容被丢弃。
close, l_onoff = 1 l_linger = 0	在套接字上不能再发出发送或接收请求。如果描述符引用计数变为0：RST被发送到对端；连接的状态被置为CLOSED（没有TIME_WAIT状态）；套接字发送缓冲区和套接字接收缓冲区中的数据被丢弃。
close, l_onoff = 1 l_linger != 0	在套接字上不能再发出发送或接收请求；套接字发送缓冲区中的数据被发送到对端。如果描述符引用计数变为0：在发送完发送缓冲区中的数据后，跟以正常的TCP连接终止序列（即发送FIN）；套接字接收缓冲区中数据被丢弃；如果在连接变为CLOSED状态前延滞时间到，那么close返回EWOULDBLOCK错误。

图7-12 shutdown和SO_LINGER各种情况的总结

- SO_OOBINLINE
 - 带外数据将被留在正常的输入队列中
 - SO_RCVBUF和SO_SNDBUF
 - TCP套接字接收缓冲区不可能溢出，因为不允许对端发出超过本端所通告窗口大小的数据
 - 对UDP,当接收到的数据报装不进套接字接收缓冲区时，该数据报被丢弃
 - TCP的窗口规模选项是在SYN分节得到的，SO_RCVBUF必须在调用connect之前设置，对于服务器，必须在调用listen之前设置
 - 至少是MSS值的4倍
 - TCP必须为每个分节保留一个副本，直到接收到来自服务器的相应ACK
 - SO_RCVLOWAT和SO_SNDBUF

- 接收低水位标记
 - 发送低水位标记
 - 由select函数使用
 - UDP套接字的发送缓冲区中可用空间的字节数不变，UDP不为应用进程传递给它的数据保留副本
- SO_RCVTIMEO和SO_SNDFTIMEO
 - 给套接字的接收和发送设置一个超时值
- SO_REUSEADDR和SO_REUSEPORT
 - 四个不同的功能
- SO_TYPE
 - 返回套接字的类型
- SO_USELOOPBACK
 - 仅用于路由域的套接字
 - 开启时，相应套接字将接收在其上发送的任何数据报的一个副本

7.6 IPV4套接字选项

- IP_HDRINCL
 - 如果本选项是给一个原始IP套接字设置的，我们必须为所有在该原始套接字上发送的数据报构造自己的IP首部
- IP_OPTIONS
 - 允许我们在IPV4首部中设置IP选项
- IP_RECVDSTADDR
 - 所收到的UDP数据报的目的地址由recvmsg函数作为辅助函数返回
- IP_RECVIF
 - 所收到UDP数据报的接收接口索引由recvmsg函数作为辅助数据返回
- IP_TOS
 - 允许为TCP UDP或SCTP套接字设置IP首部中的服务类型字段
- IP_TTL
 - 使用本选项设置或获取系统用在从某个给定套接字发送的单播分组上的默认TTL值

7.7 ICMPv6套接字选项

- ICMP6_FILTER
 - 获取或设置一个imcp6_filter结构，指出256个可能的ICMPv6消息类型中哪些将经由某个原始套接字传递给所在进程

7.8 IPv6套接字选项

- IPV6_CHECKSUM
 - 用于原始套接字的校验和字段偏移
 - 内核总是给ICMPv6原始套接字计算并存储校验和
- IPV6_DONTFRAG
 - 开启本选项将禁止为UDP套接字或原始套接字自动插入分组首部，外出分组中大小超过发送接口MTU的那些分组将被丢弃
- IPv6_NEXTHOP

- 将外出数据报的下一跳地址指定为一个套接字地址结构
- IPv6_PATHMTU
 - 获取本选项时，返回的是由路径MTU发现功能确定的当前MTU
- IPv6_RECVDSTOPTS
 - 任何接收到的IPv6目的地选项都将由recvmsg作为辅助函数返回
- IPV6_RECVHOPLIMIT
- IPV6_RECVHOPOPTS
- IPV6_RECVPATHMTU
- IPV6_RECVPKTINFO
- IPV6_RECVRTHDR
- IPv6_RECVTCLASS
- IPV6_UNICAST_HOPS
 - 本选项类似于IPv4的IP_TTL套接字选项
- IPv6_USE_MIN_MTU
- IPv6_V6ONLY
 - 在一个AF_INET6套接字上开启本选项将限制它只执行IPv6通信
- IPv6_XXX

7.9 TCP套接字选项

- TCP_MAXSEG
 - 允许我们获取或设置TCP连接的最大分节大小(MSS)
- TCP_NODELAY
 - 开启本选项将禁止TCP的Nagle算法，默认是启动的
 - Nagle 算法的目的在于防止一个连接在任何时刻有多个小分组(小于MSS的分组)待确认
 - Nagle算法常与ACK延滞算法(等待一段时间，期待ACK可由发送到对端的数据捎带)联合使用

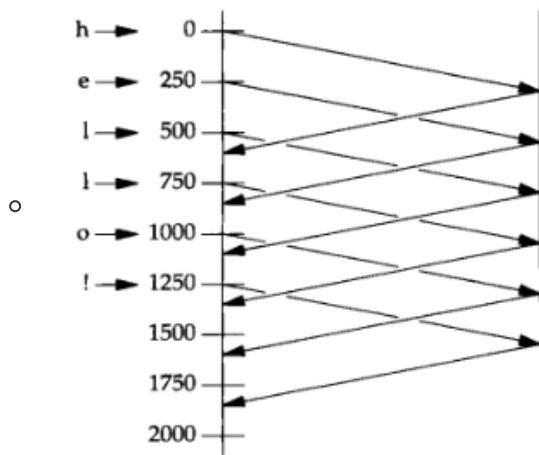


图7-14 禁止Nagle算法时由服务器回显的六个字符

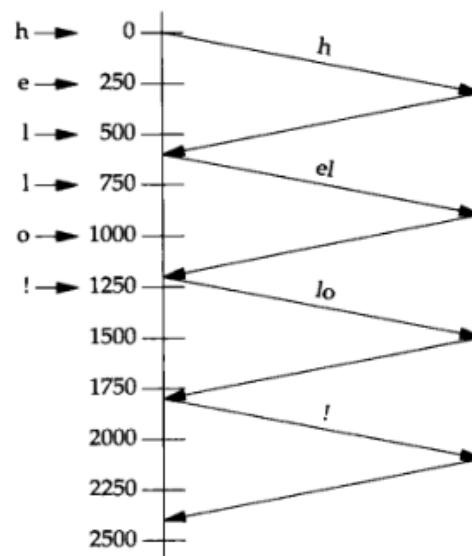


图7-15 开启Nagle算法时由服务器回显的六个字符

7.10 SCTP套接字选项

- SCTP_ADAPTION_LAYER
 - 适配层指示
 - 允许调用者获取或设置将由本端提供给对端的适配层指示
- SCTP_ASSOCINFO

本套接字选项可用于以下三个目的：(a) 获取关于某个现有关联的信息，(b) 改变某个已有关联的参数，(c) 为未来的关联设置默认信息。在获取关于某个现有关联的信息时，应该使用sctp_opt_info函数而不是getsockopt函数。作为本选项的输入的是sctp_assocparams结构。

```
struct sctp_assocparams {
    sctp_assoc_t sasoc_assoc_id;
    u_int16_t sasoc_asocmaxrxt;
    u_int16_t sasoc_number_peer_destinations;
    u_int32_t sasoc_peer_rwnd;
    u_int32_t sasoc_local_rwnd;
    u_int32_t sasoc_cookie_life;
};
```

这些字段的含义如下所述。

- • sasoc_assoc_id存放待访问关联的标识（即关联ID）。如果在调用setsockopt时置0本字段，那么sasoc_asocmaxrxt和sasoc_cookie_life字段代表将作为默认信息设置在相应套接字上的值。如果在调用getsockopt时提供关联ID，返回的就是特定于该关联的信息，否则如果置0本字段，返回的就是默认的端点设置信息。
- • sasoc_asocmaxrxt存放的是某个关联在已发送数据没有得到确认的情况下尝试重传的最大次数。达到这个次数后SCTP放弃重传，报告用户对端不可用，然后关闭该关联。
- • sasoc_number_peer_destinations存放对端目的地址数。它不能设置，只能获取。
- • sasoc_peer_rwnd存放对端的当前接收窗口。该值表示还能发送给对端的数据字节总数。本字段是动态的，本地端点发送数据时其值减小，外地应用进程读取已经收到的数据时其值增大。它不能设置，只能获取。
- • sasoc_local_rwnd存放本地SCTP协议栈当前通告对端的接收窗口。本字段也是动态的，并受SO_SNDBUF套接字选项影响。它不能设置，只能获取。
- • sasoc_cookie_life存放送给对端的状态cookie以毫秒为单位的有效期。为了防护重放
- SCTP_AUTOCLOSE
 - 获取或设置一个SCTP端点的自动关闭时间
- SCTP_DEFAULT_SEND_PARAM
 - 希望发送大量消息且所有消息具有相同发送参数的应用进程可以使用本选项设置默认参数
 - struct sctp_sndrcvinfo{...}
- SCTP_DISABLE_FRAGMENTS
 - 禁止消息分块
- SCTP_EVENTS
 - 允许调用者获取、开启或禁止各种SCTP通知
 - SCTP通知是由SCTP协议栈发送给应用进程的消息
 - struct sctp_event_subscribe{..}

字 段	说 明
sctp_data_io_event	开启/禁止每次recvmsg调用返回sctp_sndrcvinfo。
sctp_association_event	开启/禁止关联建立事件通知。
sctp_address_event	开启/禁止地址事件通知。
sctp_send_failure_event	开启/禁止消息发送故障事件通知。
sctp_peer_error_event	开启/禁止对端协议出错事件通知。
sctp_shutdown_event	开启/禁止关联终止事件通知。
sctp_partial_delivery_event	开启/禁止部分递送API事件通知。
sctp_adaption_layer_event	开启/禁止适配层事件通知。

图7-17 sctp_event_subscribe结构的各个字段

- SCTP_GET_PEER_ADDR_INFO
 - 获取某个给定对端地址的相关信息
 - struct sctp_paddrinfo{..}
- SCTP_I_WANT_MAPPED_V4_ADDR
 - 用于为AF_INET6类型的套接字开启或禁止IPv4映射地址
- SCTP_INITMSG
 - 用户获取或设置某个SCTP套接字在发送INIT消息时所用的默认初始参数
 - struct sctp_initmsg{..}
- SCTP_MAXBURST
 - 最大猝发大小
- SCTP_MAXSEG
 - 最大片段大小
- SCTP_NODELAY
 - 禁止SCTP的Nagle算法
- SCTP_PEER_ADDR_PARAMS
 - 获取或设置某个关联的对端地址的各种参数
 - sctp_paddrparams结构
- SCTP_PRIMARY_ADDR
 - 获取或设置本地端点所用的主目的地址
 - sctp_setprim
- SCTP_RTOINFO
 - 获取或设置各种RTO消息
 - sctp_rtoinfo结构
- SCTP_SET_PEER_PRIMARY_ADDR
 - 请求对端把所指定的本地地址作为它的主目的地址
 - sctp_setpeerprim结构
- SCTP_STATUS
 - 用于获取某个关联的状态
 - sctp_status结构

7.11 fcnt1函数

操作	fcntl	ioctl	路由套接字	POSIX
设置套接字为非阻塞式I/O型	F_SETFL, O_NONBLOCK	FIONBIO		fcntl
设置套接字为信号驱动式I/O型	F_SETFL, O_ASYNC	FIOASYNC		fcntl
设置套接字属主	F_SETOWN	SIOCSPGRP或FIOSETOWN		fcntl
获取套接字属主	F_GETOWN	SIOCGPGRP或FIOGETOWN		fcntl
获取套接字接收缓冲区中的字节数		FIONREAD		
测试套接字是否处于带外标志		SIOCATMARK		socketmark
获取接口列表		SIOCGIFCONF	sysctl	
接口操作		SIOC[GS]IFxxx		
ARP高速缓存操作		SIOCxARP	RTM_xxx	
路由表操作		SIOCxxRT	RTM_xxx	

图7-20 fcntl、 ioctl和路由套接字操作小结

- 设置某个文件状态标志的唯一正确的方法是: 先取得当前标志, 与新标志逻辑或后再设置标志
 - 非阻塞式I/O。通过使用F_SETFL命令设置O_NONBLOCK文件状态标志, 我们可以把一个套接字设置为非阻塞型。我们将在第16章中讲述非阻塞式I/O。
 - 信号驱动式I/O。通过使用F_SETFL命令设置O_ASYNC文件状态标志, 我们可以把一个套接字设置成一旦其状态发生变化, 内核就产生一个SIGIO信号。我们将在第25章中讨论这一点。
 - F_SETOWN命令允许我们指定用于接收SIGIO和SIGURG信号的套接字属主(进程ID或进程组ID)。其中SIGIO信号是套接字被设置为信号驱动式I/O型后产生的(第25章), SIGURG信号是在新的带外数据到达套接字时产生的(第24章)。F_GETOWN命令返回套接字的当前属主。

第八章 基本UDP套接字编程

8.1 概述

- 使用UDP编写的一些常见应用程序: DNS NFS SNMP

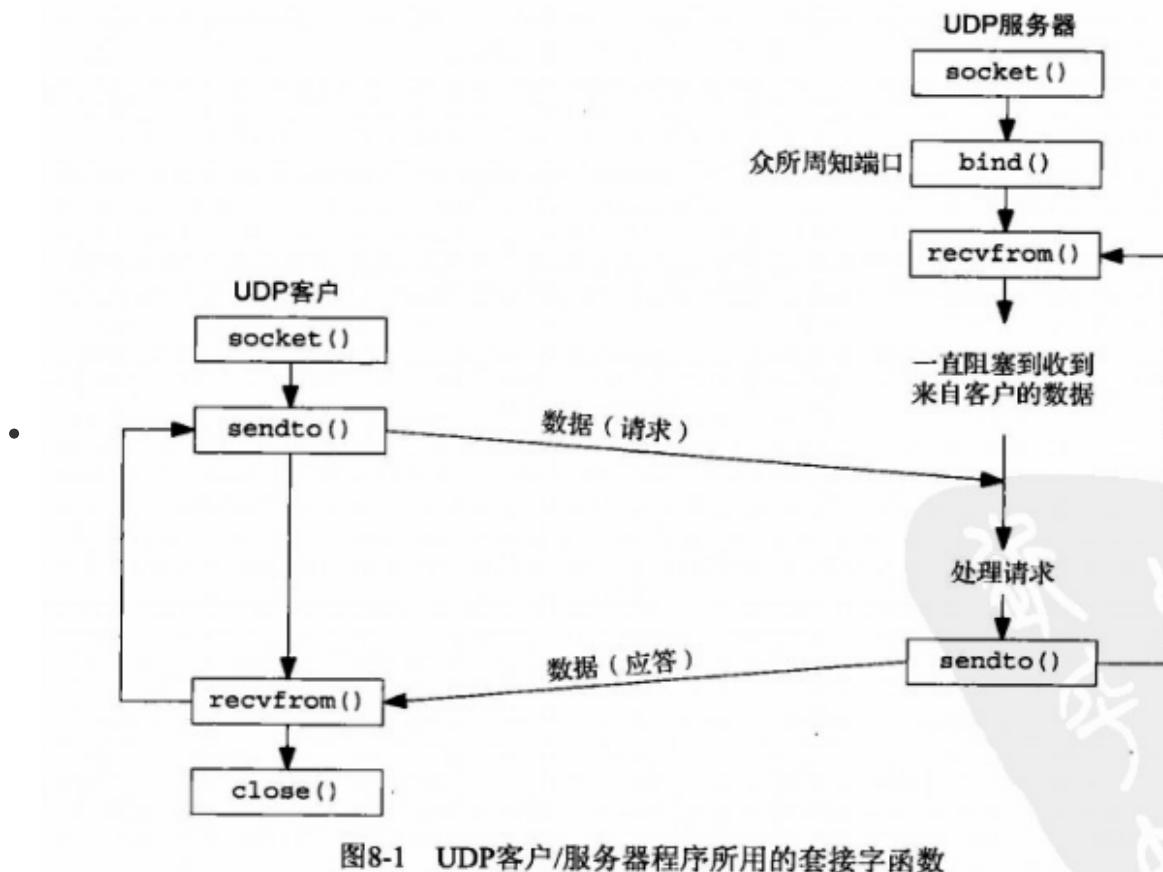


图8-1 UDP客户/服务器程序所用的套接字函数

8.2 recvfrom和sendto函数

- 类似于标准的read和write
- 把读写数据的长度作为函数返回值，recvfrom返回0值是可接受的

8.3 UDP回射服务器程序: main函数

8.4 UDP回射服务器程序: dg_echo函数

```

1 #include    "unp.h"
2
3 void
4 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
5 {
6     int      n;
7     socklen_t len;
8     char    mesg[MAXLINE];
9
10    for ( ; ; ) {
11        len = clilen;
12        n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
13        Sendto(sockfd, mesg, n, 0, pcliaddr, len);
14    }
}

```

lib/dg_echo.c

图8-4 dg_echo函数：在数据报套接字上回射文本行

- 没有对fork的调用，是一个迭代服务器，单个服务器进程就得处理所有客户
- 大多数TCP服务器是并发的，大多数UDP服务器是迭代的
- 每个UDP套接字都有一个接收缓冲区，到达该套接字的每个数据报都进入这个套接字接收缓冲区

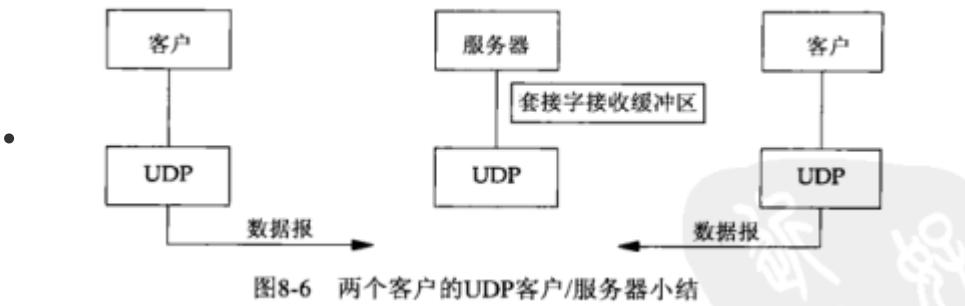


图8-6 两个客户的UDP客户/服务器小结

8.5 UDP回射客户程序: main函数

8.6 UDP回射客户程序: dg_cli函数

```

1 #include      "unp.h"
2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int      n;
6     char    sendline[MAXLINE], recvline[MAXLINE + 1];
7     while (Fgets(sendline, MAXLINE, fp) != NULL) {
8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
9         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
10        recvline[n] = 0; /* null terminate */
11        Fputs(recvline, stdout);
12    }
13 }
```

lib/dg_cli.c

图8-8 dg_cli函数: 客户处理循环

- 对于一个UDP套接字，如果其进程首次调用sendto时没有绑定一个本地端口，那么内核在此时为它选择一个临时端口

8.7 数据报的丢失

- 设定超时
- 数据报没有到达服务器，服务器的应答没有回到客户

8.8 验证接收到的响应

```

1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int      n;
6     char    sendline[MAXLINE], recvline[MAXLINE + 1];
7     socklen_t len;
8     struct sockaddr *preply_addr;
9
10    preply_addr = Malloc(servlen);
11
12    while (Fgets(sendline, MAXLINE, fp) != NULL) {
13        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
14        len = servlen;
15        n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
16        if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
17            printf("reply from %s (ignored)\n", Sock_ntop(prepay_addr, len));
18            continue;
19        }
20    }
21 }

```

—udpcliserv/dgcliaddr.c

图8-9 验证返回的套接字地址的dg_cli函数版本

- 保留来自数据报所发往服务器的应答，而忽略任何其他数据报
- 大多数IP实现接受目的地址为本主机任一IP地址的数据报，而不管数据报到达的接口
- 服务器主机可能是多宿的，会产生很多问题

8.9 服务器进程未运行

- 对一个UDP套接字，由它引发的异步错误(sendto发送成功，ICMP不可达错误)却并不返回给它，除非它已连接
- recvfrom可以返回的信息仅有errno值，它没有办法返回出错数据报的目的IP地址和目的UDP端口号，仅在进程已将其UDP套接字连接到恰恰一个对端时，这些异步错误才返回给进程

8.10 UDP程序例子小结

- 客户的临时端口是在第一次调用sendto时一次性选定，不能改变，然而客户的IP地址却可以随客户发送的每个UDP数据报而变动

来自客户的IP数据报	TCP服务器	UDP服务器
源IP地址	accept	recvfrom
源端口号	accept	recvfrom
目的IP地址	getsockname	recvmsg
目的端口号	getsockname	getsockname

图8-13 服务器可从到达的IP数据报中获取的信息

8.11 UDP的connect函数

- 给UDP套接字调用connect，没有三路握手过程，内核只是检查是否存在立即可知的错误(如一个显然不可达的目的地)，记录对端的IP地址和端口号，然后立即返回到调用进程
- 已连接套接字
 - 不使用sendto,改用write和send

- 不必使用recvfrom,而改用read recv或recvmsg.限制一个已连接UDP套接字能且只能与一个对端交换数据报
- 由已连接UDP套接字引发的异步错误会返回给它们所在的进程

套接字类型	write或send	不指定目的地的sendto	指定目的地的sendto
TCP套接字	可以	可以	EISCONN
UDP套接字, 已连接	可以	可以	EISCONN
UDP套接字, 未连接	EDESTADDRREQ	EDESTADDRREQ	可以

图8-14 TCP和UDP套接字: 可指定目的地协议地址吗?

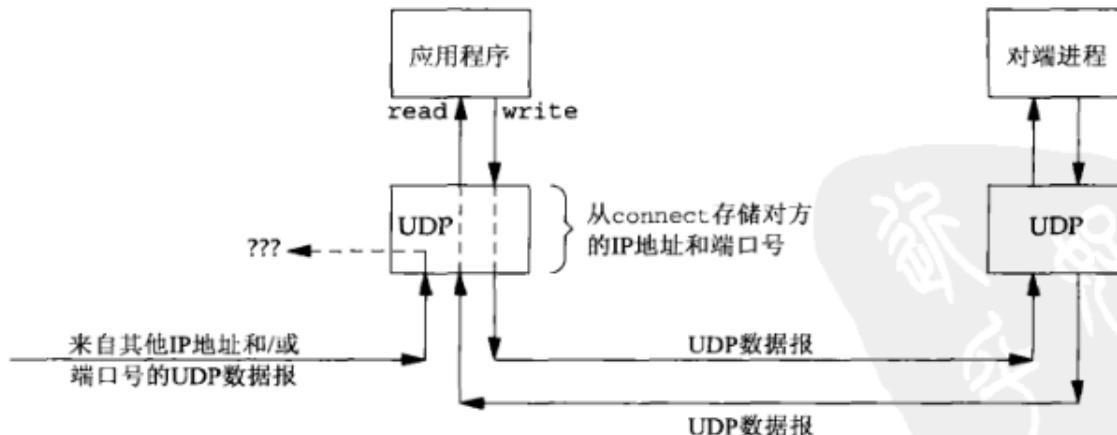


图8-15 已连接UDP套接字

应用进程首先调用connect指定对端的IP地址和端口号, 然后使用read和write与对端进

- UDP客户进程或服务器进程只在使用自己的UDP套接字与确定的唯一对端进行通信时, 才可调用connect
- 给一个UDP套接字多次调用connect
 - 指定新的IP地址和端口号
 - 断开套接字
- 性能
 - 显式连接套接字效率更高
 - 在一个未连接的UDP套接字上给两个数据报调用sendto函数
 - 连接套接字
 - 输出第一个数据报
 - 断开套接字连接
 - 连接套接字
 - 输出第二个数据报
 - 断开套接字连接

8.12 dg_cli函数

```

1 #include    "unp.h"
2 void
3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int      n;
6     char    sendline[MAXLINE], recvline[MAXLINE + 1];
7     Connect(sockfd, (SA *) pservaddr, servlen);
8     while (Fgets(sendline, MAXLINE, fp) != NULL) {
9         Write(sockfd, sendline, strlen(sendline));
10        n = Read(sockfd, recvline, MAXLINE);
11        recvline[n] = 0;          /* null terminate */
12        Fputs(recvline, stdout);
13    }
14 }
```

udpcliserv/dgcliconnect.c

图8-17 调用connect的dg_cli函数

8.13 UDP缺乏流量控制

- netstat -s 输出的统计数据将表明丢失了多少数据报
- 因套接字缓冲区满而丢弃数据报
- UDP套接字接收缓冲区

8.14 UDP中的外出接口的确定

- 已连接套接字还可用来确定用于某个特定目的地的外出接口
- connect到一个指定的IP地址后调用getsockname得到本地IP地址和端口号并显示输出
- 在UDP套接字上调用connect并不给对端主机发送任何信息，只是保存对端的IP地址和端口号，同时也给该套接字指定一个临时端口

8.15 使用select函数的TCP和UDP回射服务器程序

- 单个使用select来复用TCP和UDP套接字的服务器程序
- TCP端口是独立于UDP端口的

```

/* include udpsrvselect01 */
#include    "unp.h"

int
main(int argc, char **argv)
{
    int                  listenfd, connfd, udpfd, nready, maxfdp1;
    char                mesg[MAXLINE];
    pid_t               childpid;
    fd_set              rset;
    ssize_t              n;
    socklen_t            len;
    const int            on = 1;
    struct sockaddr_in   cliaddr, servaddr;
    void                sig_chld(int);
```

```

/* 4create listening TCP socket */
listenfd = Socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(SERV_PORT);

Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

Listen(listenfd, LISTENQ);

/* 4create UDP socket */
udpfd = Socket(AF_INET, SOCK_DGRAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(SERV_PORT);

Bind(udpfd, (SA *) &servaddr, sizeof(servaddr));
/* end udpserveselect01 */

/* include udpserveselect02 */
Signal(SIGCHLD, sig_chld); /* must call waitpid() */

FD_ZERO(&rset);
maxfdp1 = max(listenfd, udpfd) + 1;
for ( ; ; ) {
    FD_SET(listenfd, &rset);
    FD_SET(udpfd, &rset);
    if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
        if (errno == EINTR)
            continue; /* back to for() */
        else
            err_sys("select error");
    }

    if (FD_ISSET(listenfd, &rset)) {
        len = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &len);

        if ( (childpid = Fork()) == 0) { /* child process */
            Close(listenfd); /* close listening socket */
            str_echo(connfd); /* process the request */
            exit(0);
        }
        Close(connfd); /* parent closes connected socket */
    }

    if (FD_ISSET(udpfd, &rset)) {
        len = sizeof(cliaddr);

```

```

n = Recvfrom(udpfd, mesg, MAXLINE, 0, (SA *) &cliaddr, &len);

Sendto(udpfd, mesg, n, 0, (SA *) &cliaddr, len);
}

}

/* end udpservelelect02 */

```

第9章 基本SCTP套接字编程

9.2 接口模型

- SCTP套接字分为一到一套接字和一到多套接字
- 一到一形式
 - 开发一到一形式的目的是将现有TCP应用程序移植到SCTP上
 - 任何TCP套接字选项必须转换成等效的SCTP套接字选项
 - SCTP保存消息边界

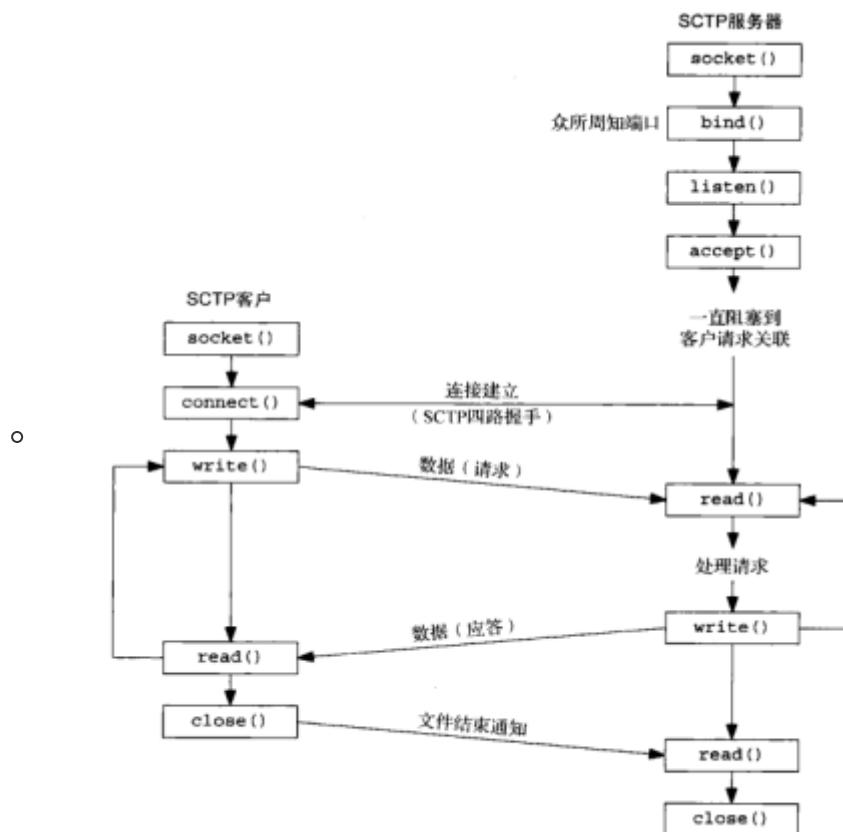


图9-1 SCTP一到一形式的套接字函数

- 一到多形式
 - 单个套接字描述符将代表多个关联，用于标识单个关联的是一个关联标识，类型为sctp_assoc_t值
 - 当一个客户关闭其关联时，其服务器也自动关闭同一个关联
 - 可用于致使在四路握手的第三个或第四个分组中捎带用户数据的唯一办法是使用一到多形式
 - 对于一个与它还没有关联存在的IP地址，任何以它为目的地的sendto sendmsg或sctp_sendmsg将导致对主动打开的尝试
 - 必须使用sendto sendmsg或sctp_sendmsg这三个分组发送函数
 - 调用分组发送函数时，所用的目的地址是由系统在关联建立阶段选定的主目的地址

- 关联事件可能被启用

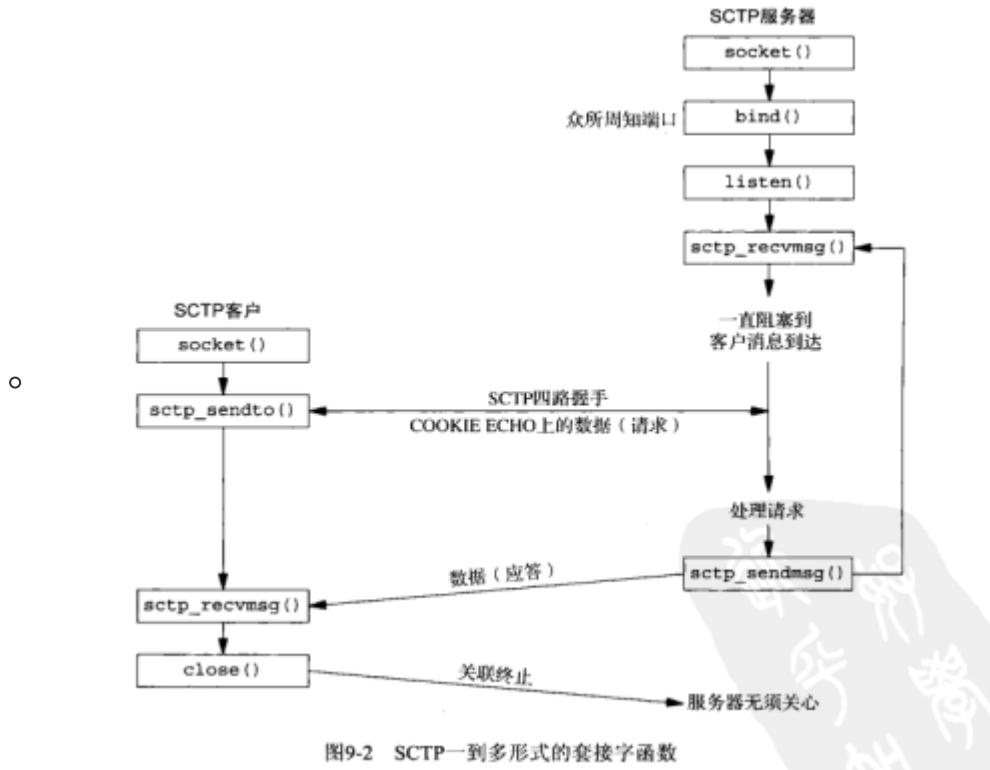


图9-2 SCTP一到多形式的套接字函数

9.3 sctp_bindx函数

- 捆绑与所在主机系统相关IP地址的一个子集
- 如果在一个监听套接字上执行sctp_bindx，那么将来产生的关联将使用新的地址配置，已经存在的关联则不受影响

9.4 sctp_connectx函数

- 用于连接到一个多宿对端主机

9.5 sctp_getpaddrs函数

- 可以知道对端的所有地址

9.6 sctp_freepaddrs

- 释放由sctp_getpaddrs函数分配的资源

9.7 sctp_getladdrs

- 用于获取属于某个关联的本地地址

9.8 sctp_freeladdrs

- 释放sctp_getladdrs函数分配的资源

9.9 sctp_sendmsg

- 通过使用伴随辅助数据的sendmsg函数，应用进程能够控制SCTP的各种特性

- 许多SCTP实现提供了一个辅助函数库调用sctp_sendmsg

9.10 sctp_recvmsg

9.11 sctp_opt_info函数

- 为无法为SCTP使用getsockopt函数的那些实现提供的

9.12 sctp_peeloff

- 从一个一到多式套接字中抽取一个关联，构成一个一到一式套接字，调用结束时将返回一个新的套接字描述符

9.13 shutdown函数

- SCTP允许一个端点调用shutdown,之后这个端点可以重用原套接字连接到一个新的对端
- 当相互通信的两个SCTP端点中任何一个发起关联终止序列时，这两个端点都得把已排队的任何数据发送掉，然后关闭关联



图9-5 调用shutdown关闭一个SCTP关联

9.14 通知

- SCTP为应用程序提供了多种可用的通知，SCTP用户可用经由这些通知追踪相关关联的状态
- recvmsg或sctp_recvmsg,如果所返回的数据是一个事件通知，那么这两个函数返回的msg_flags参数将含有MSG_NOTIFICATION标志
- SCTP_ASSOC_CHANGE
- SCTP_PEER_ADDR_CHANGE
- SCTP_REMOTE_ERROR
- SCTP_SEND_FAILED
- SCTP_SHUTDOWN_EVENT
- SCTP_ADAPTION_INDICATION
- SCTP_PARITAL_DELIVERY_EVENT

第10章 SCTP客户/服务器程序例子

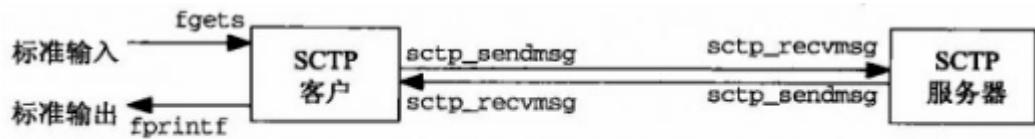


图10-1 简单的SCTP流分回射客户/服务器

10.2 SCTP一到多式流分回射服务器程序: main函数

```
#include "unp.h"

int
main(int argc, char **argv)
{
    int sock_fd, msg_flags;
    char readbuf[BUFFSIZE];
    struct sockaddr_in servaddr, cliaddr;
    struct sctp_sndrcvinfo sri;
    struct sctp_event_subscribe evnts;
    int stream_increment=1;
    socklen_t len;
    size_t rd_sz;

    if (argc == 2)
        stream_increment = atoi(argv[1]);
    sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(sock_fd, (SA *) &servaddr, sizeof(servaddr));

    bzero(&evnts, sizeof(evnts));
    evnts.sctp_data_io_event = 1;
    Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS,
               &evnts, sizeof(evnts));

    Listen(sock_fd, LISTENQ);
    for ( ; ; ) {
        len = sizeof(struct sockaddr_in);
        rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
                             (SA *)&cliaddr, &len,
                             &sri, &msg_flags);
        if(stream_increment) {
            sri.sinfo_stream++;
            if(sri.sinfo_stream >= sctp_get_no_streams(sock_fd, (SA *)&cliaddr, len))
                sri.sinfo_stream = 0;
        }
        Sctp_sendmsg(sock_fd, readbuf, rd_sz,
                    (SA *)&cliaddr, len,
```

```

        sri.sinfo_ppid,
        sri.sinfo_flags,
        sri.sinfo_stream,
        0, 0);
}
}

```

10.3 SCTP一到多式流分回射客户程序:main函数

```

#include    "unp.h"

int
main(int argc, char **argv)
{
    int sock_fd;
    struct sockaddr_in servaddr;
    struct sctp_event_subscribe evnts;
    int echo_to_all=0;

    if(argc < 2)
        err_quit("Missing host argument - use '%s host [echo]\n",
            argv[0]);
    if(argc > 2) {
        printf("Echoing messages to all streams\n");
        echo_to_all = 1;
    }
    sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    bzero(&evnts, sizeof(evnts));
    evnts.sctp_data_io_event = 1;
    Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS,
               &evnts, sizeof(evnts));
    if(echo_to_all == 0)
        sctpstr_cli(stdin, sock_fd, (SA *)&servaddr, sizeof(servaddr));
    else
        sctpstr_cli_echoall(stdin, sock_fd, (SA *)&servaddr, sizeof(servaddr));
    close(sock_fd);
    return(0);
}

```

10.4 SCTP流分回射客户程序:sctpstr_cli函数

```

#include    "unp.h"

void
sctpstr_cli(FILE *fp, int sock_fd, struct sockaddr *to, socklen_t tolen)

```

```

{
    struct sockaddr_in peeraddr;
    struct sctp_sndrcvinfo sri;
    char sendline[MAXLINE], recvline[MAXLINE];
    socklen_t len;
    int out_sz,rd_sz;
    int msg_flags;

    bzero(&sri,sizeof(sri));
    while (fgets(sendline, MAXLINE, fp) != NULL) {
        if(sendline[0] != '[') {
            printf("Error, line must be of the form '[streamnum]text'\n");
            continue;
        }
        sri.sinfo_stream = strtol(&sendline[1],NULL,0);
        out_sz = strlen(sendline);
        Sctp_sendmsg(sock_fd, sendline, out_sz,
                     to, tolen,
                     0, 0,
                     sri.sinfo_stream,
                     0, 0);

        len = sizeof(peeraddr);
        rd_sz = Sctp_recvmsg(sock_fd, recvline, sizeof(recvline),
                             (SA *)&peeraddr, &len,
                             &sri,&msg_flags);
        printf("From str:%d seq:%d (assoc:0x%lx):",
               sri.sinfo_stream,sri.sinfo_ssn,
               (u_int)sri.sinfo_assoc_id);
        printf("%.*s",rd_sz,recvline);
    }
}

```

10.5 探究头端阻塞

- 头端阻塞发生在一个TCP分节丢失，导致其后续分节不按序到达接收端的时候
- SCTP的多流特性能够尽可能减少头端阻塞
- SCTP流可以说是一个既能避免头端阻塞又能在相关的消息之间保持顺序的有效机制

```

21      for (i = 0 ; i < SERV_MAX_SCTP_STRM; i++) {
22          snprintf(sendline + strsz, sizeof(sendline) - strsz,
23                  ".msg.%d 1", i);
24          Sctp_sendmsg(sock_fd, sendline, sizeof(sendline),
25                         to, tolen, 0, 0, i, 0, 0);
26          snprintf(sendline + strsz, sizeof(sendline) - strsz,
27                  ".msg.%d 2", i);
28          Sctp_sendmsg(sock_fd, sendline, sizeof(sendline),
29                         to, tolen, 0, 0, i, 0, 0);
30      }
31      for (i = 0; i < SERV_MAX_SCTP_STRM * 2; i++) {
32          len = sizeof(peeraddr);

```

sctp/sctp_strcliecho2.c

图10-9 sctpcstr_cli函数改动部分

读回消息并显示

31 这儿的代码只需略加改动：把客户期待收回的来自回射服务器的消息数目翻倍。

10.5.2 运行改动过的代码

我们像先前那样执行服务器程序和改动过的客户程序，得到的来自客户的输出如下。

```

freebsd4% sctpclient01 10.1.4.1 echo
Echoing messages to all streams
Hello
From str:0 seq:0 (assoc:0xc99e15a0):Hello.msg.0 1
From str:0 seq:1 (assoc:0xc99e15a0):Hello.msg.0 2
From str:1 seq:0 (assoc:0xc99e15a0):Hello.msg.1 1
From str:4 seq:0 (assoc:0xc99e15a0):Hello.msg.4 1
From str:5 seq:0 (assoc:0xc99e15a0):Hello.msg.5 1
From str:7 seq:0 (assoc:0xc99e15a0):Hello.msg.7 1
From str:8 seq:0 (assoc:0xc99e15a0):Hello.msg.8 1
From str:9 seq:0 (assoc:0xc99e15a0):Hello.msg.9 1
From str:3 seq:0 (assoc:0xc99e15a0):Hello.msg.3 1
From str:3 seq:1 (assoc:0xc99e15a0):Hello.msg.3 2
From str:1 seq:1 (assoc:0xc99e15a0):Hello.msg.1 2
From str:5 seq:1 (assoc:0xc99e15a0):Hello.msg.5 2
From str:2 seq:0 (assoc:0xc99e15a0):Hello.msg.2 1
From str:6 seq:0 (assoc:0xc99e15a0):Hello.msg.6 1
From str:6 seq:1 (assoc:0xc99e15a0):Hello.msg.6 2
From str:2 seq:1 (assoc:0xc99e15a0):Hello.msg.2 2
From str:7 seq:1 (assoc:0xc99e15a0):Hello.msg.7 2
From str:8 seq:1 (assoc:0xc99e15a0):Hello.msg.8 2
From str:9 seq:1 (assoc:0xc99e15a0):Hello.msg.9 2
From str:4 seq:1 (assoc:0xc99e15a0):Hello.msg.4 2
^D
freebsd4%

```

98

- 消息存在丢失现象，只有在同一个流内的信息才因此延缓，其他流中的消息不受影响

10.6 控制流的数目

```

14     if (argc == 2)
15         stream_increment = atoi(argv[1]);
16     sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
17     bzero(&initm,sizeof(initm));
18     initm.sinit_num_ostreams = SERV_MORE_STRMS_SCTP;
19     Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_INITMSG, &initm, sizeof(initm));

```

sctp/sctpserv02.c

图10-10 服务器程序请求更多流的改动部分

10.7 控制终结

- 如果服务器希望在发送完一个应答消息后终止一个关联，那么可以在与该消息对应的sctp_sndrcvinfo结构的sinfo_flags字段中设置MSG_EOF标志，迫使所发送消息被客户确认后，相应关联也被终止
- 另一个办法是把MSG_ABORT标志应用于sinfo_flags字段，该标志将以ABORT块迫使立即终止关联，类似与TCP的RST分节，能够无延迟地终止任何关联，尚未发送的任何数据都被丢弃

```
sctp/sctpserv03.c
```

```

25     for ( ; ; ) {
26         len = sizeof(struct sockaddr_in);
27         rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
28                               (SA *)&cliaddr, &len, &sri, &msg_flags);
29         if (stream_increment) {
30             sri.sinfo_stream++;
31             if (sri.sinfo_stream >=
32                 sctp_get_no_streams(sock_fd, (SA *)&cliaddr, len))
33                 sri.sinfo_stream = 0;
34         }
35         Sctp_sendmsg(sock_fd, readbuf, rd_sz,
36                       (SA *)&cliaddr, len,
37                       sri.sinfo_ppid,
38                       (sri.sinfo_flags | MSG_EOF), sri.sinfo_stream, 0, 0);
39     }

```

```
sctp/sctpserv03.c
```

图10-11 服务器程序应答同时终止关联的改动部分

```
sctp/scptclient02.c
```

```

25     if (echo_to_all == 0)
26         sctpstr_cli(stdin, sock_fd, (SA *)&servaddr, sizeof(servaddr));
27     else
28         sctpstr_cli_echoall(stdin, sock_fd, (SA *)&servaddr,
29                             sizeof(servaddr));
30     strcpy(byemsg, "goodbye");
31     Sctp_sendmsg(sock_fd, byemsg, strlen(byemsg),
32                   (SA *)&servaddr, sizeof(servaddr), 0, MSG_ABORT, 0, 0, 0);
33     Close(sock_fd);

```

```
sctp/scptclient02.c
```

图10-12 客户程序预先中止关联的改动部分

第11章 名字与地址转换

11.2 域名系统

- 资源记录
 - DNS中的条目称为资源记录
 - A记录把一个主机名映射成一个32位的IPV4地址
 - AAAA记录把一个主机名映射成一个128位的IPV6地址
 - PTR记录把IP地址映射成主机名
 - MX记录把一个主机指定作为给定主机的邮件交换器
 - CNAME 为常用的服务指派CANME记录
- 解析器和名字服务器

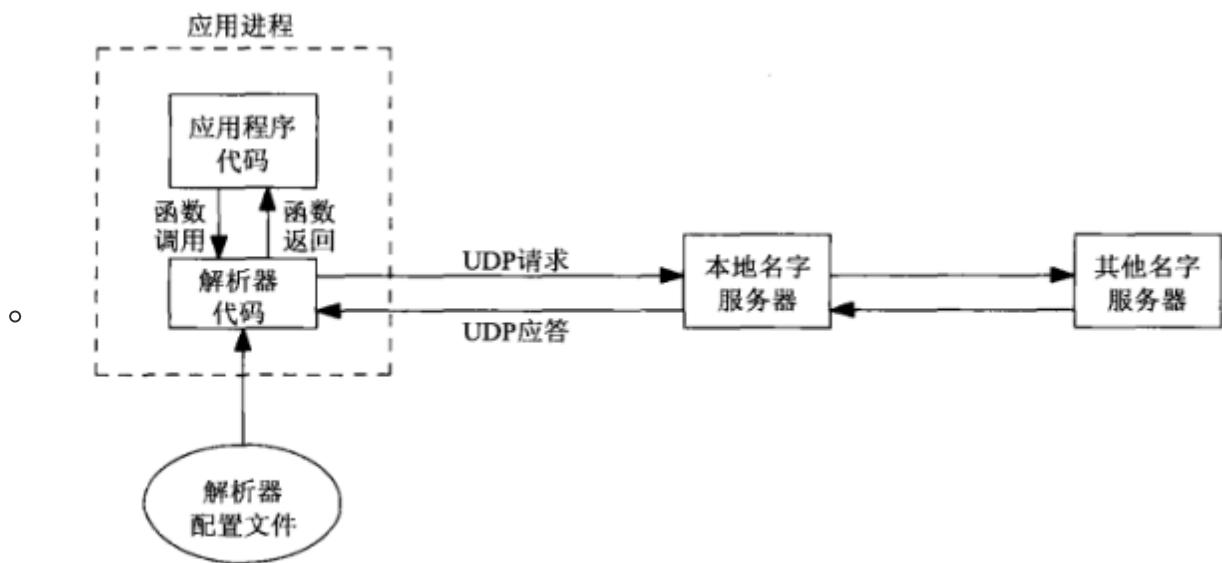


图11-1 客户、解析器和名字服务器的典型关系

- 文件/etc/resolv.conf通常包含本地名字服务器主机的IP地址

- DNS替代方法

- 静态主机文件 /etc/hosts
- 网络信息系统

11.3 gethostbyname

- 返回一个指向hostent结构的指针，该结构包含所查找主机的所有IPv4地址

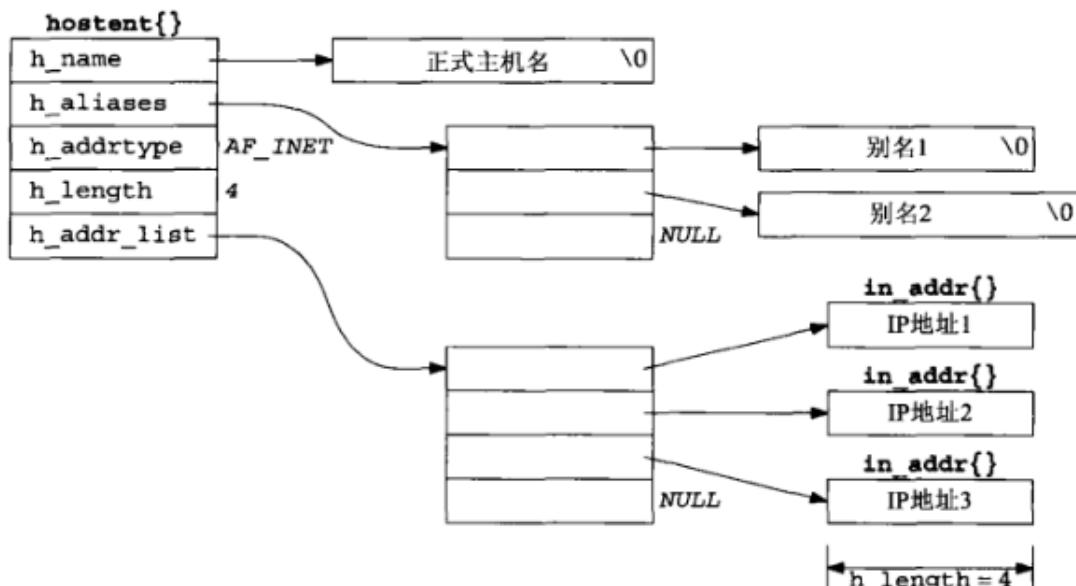


图11-2 hostent结构和它所包含的信息

```
#include    "unp.h"

int
main(int argc, char **argv)
{
    char          *ptr, **pptr;
    char          str[INET_ADDRSTRLEN];
```

```

    struct hostent *hptr;

    while (--argc > 0) {
        ptr = *++argv;
        if ( (hptr = gethostbyname(ptr)) == NULL) {
            err_msg("gethostbyname error for host: %s: %s",
                    ptr, hstrerror(h_errno));
            continue;
        }
        printf("official hostname: %s\n", hptr->h_name);

        for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
            printf("\talias: %s\n", *pptr);

        switch (hptr->h_addrtype) {
        case AF_INET:
            pptr = hptr->h_addr_list;
            for ( ; *pptr != NULL; pptr++)
                printf("\taddress: %s\n",
                       Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
            break;

        default:
            err_ret("unknown address type");
            break;
        }
    }
    exit(0);
}

```

- hstrerror以某个h_errno值作为唯一的参数，返回一个const char*指针

11.4 gethostbyaddr

- 由一个二进制IP地址找到相应的主机名

11.5 getservbyname和getservbyport

- cat /etc/services 从名字到端口号的映射关系
- 根据给定名字查找相应服务 getservbyname
- getservbyport根据给定端口号和可选协议查找相应服务

11.6 getaddrinfo函数

- 能够处理名字到地址以及服务到端口这两种转换，返回一个sockaddr结构
- struct addrinfo结构
- 在addrinfo结构中返回的信息可现成用于socket调用，随后用于适合客户的connect或sendto调用，或者是适合客户的bind调用

11.7 gai_strerror函数

11.8 freeaddrinfo函数

- 由getaddrinfo返回的所有存储空间都是动态获取的，这些存储空间通过调用freeaddrinfo返还给系统
- 浅复制：只复制这个结构而不复制由它转而指向的其他结构
- 深复制：既复制这个结构又复制它指向的所有其他结构

11.9 getaddrinfo函数：IPV6

调用者指定的主机名	调用者指定的地址族	主机名字符串包含	结 果	行 为
非空主机名字符串；主动或被动	AF_UNSPEC	主机名	以sockaddr_in6{}返回所有AAAA记录，以sockaddr_in{}返回所有A记录	AAAA记录搜索加上A记录搜索
		十六进制数串	一个sockaddr_in6{}	inet_nton(AF_INET6)
		点分十进制数串	一个sockaddr_in{}	inet_nton(AF_INET)
	AF_INET6	主机名	以sockaddr_in6{}返回所有AAAA记录	AAAA记录搜索
			在 ai_flags 含 AI_V4MAPPED 前提下：若存在AAAA记录则以sockaddr_in6{}返回所有AAAA记录；否则以sockaddr_in6{}作为IPv4映射的IPv6地址返回所有A记录	AAAAA记录搜索，若无结果则A记录搜索
		主机名	在 ai_flags 含 AI_V4MAPPED 和 AI_ALL前提下：以sockaddr_in6{}返回所有AAAA记录，并且以sockaddr_in6{}作为 IPv4 映射的IPv6地址返回所有A记录	AAAA记录搜索加上A记录搜索
			十六进制数串	一个sockaddr_in6{}
			点分十进制数串	作为主机名查找
	AF_INET	主机名	以sockaddr_in{}返回所有A记录	A记录搜索
		十六进制数串	作为主机名查找	
		点分十进制数串	一个sockaddr_in{}	inet_nton(AF_INET)
空主机名字字符串；被动	AF_UNSPEC	隐含0::0 隐含0.0.0.0	一个sockaddr_in6{}和一个sockaddr_in{}	inet_nton(AF_INET6) inet_nton(AF_INET)
	AF_INET6	隐含0::0	一个sockaddr_in6{}	inet_nton(AF_INET6)
	AF_INET	隐含0.0.0.0	一个sockaddr_in{}	inet_nton(AF_INET)
空主机名字字符串；主动	AF_UNSPEC	隐含0::1 隐含127.0.0.1	一个sockaddr_in6{}和一个sockaddr_in{}	inet_nton(AF_INET6) inet_nton(AF_INET)
	AF_INET6	隐含0::1	一个sockaddr_in6{}	inet_nton(AF_INET6)
	AF_INET	隐含127.0.0.1	一个sockaddr_in{}	inet_nton(AF_INET)

图11-8 getaddrinfo函数及其行为和结果汇总

11.10 getaddrinfo函数：例子

- 双栈主机上的IPv6客户或服务器既能与IPv6对端通信，也能与IPv4对端通信

11.11 host_serv函数

11.12 tcp_connect函数

- 创建一个TCP套接字并连接到一个服务器

```
/* include tcp_connect */
#include "unp.h"
```

```

int
tcp_connect(const char *host, const char *serv)
{
    int                 sockfd, n;
    struct addrinfo hints, *res, *ressave;

    bzero(&hints, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
        err_quit("tcp_connect error for %s, %s: %s",
                 host, serv, gai_strerror(n));
    ressave = res;

    do {
        sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
        if (sockfd < 0)
            continue; /* ignore this one */

        if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
            break; /* success */

        Close(sockfd); /* ignore this one */
    } while ((res = res->ai_next) != NULL);

    if (res == NULL) /* errno set from final connect() */
        err_sys("tcp_connect error for %s, %s", host, serv);

    freeaddrinfo(ressave);

    return(sockfd);
}
/* end tcp_connect */

/*
 * We place the wrapper function here, not in wraplib.c, because some
 * XTI programs need to include wraplib.c, and it also defines
 * a Tcp_connect() function.
 */
int
Tcp_connect(const char *host, const char *serv)
{
    return(tcp_connect(host, serv));
}

```

```

#include    "unp.h"

int
main(int argc, char **argv)
{
    int                 sockfd, n;

```

```

char          recvline[MAXLINE + 1];
socklen_t     len;
struct sockaddr_storage ss;

if (argc != 3)
    err_quit("usage: daytimetcpccli <hostname/IPaddress> <service/port#>");

sockfd = Tcp_connect(argv[1], argv[2]);

len = sizeof(ss);
Getpeername(sockfd, (SA *)&ss, &len);
printf("connected to %s\n", Sock_ntop_host((SA *)&ss, len));

while ( (n = Read(sockfd, recvline, MAXLINE)) > 0) {
    recvline[n] = 0; /* null terminate */
    Fputs(recvline, stdout);
}
exit(0);
}

```

11.13 tcp_listen函数

- 创建一个TCP套接字，给它捆绑服务器众所周知端口，并允许外来的连接请求

```

/* include tcp_listen */
#include    "unp.h"

int
tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
{
    int            listenfd, n;
    const int      on = 1;
    struct addrinfo hints, *res, *ressave;

    bzero(&hints, sizeof(struct addrinfo));
    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
        err_quit("tcp_listen error for %s, %s: %s",
                 host, serv, gai_strerror(n));
    ressave = res;

    do {
        listenfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
        if (listenfd < 0)
            continue; /* error, try next one */

        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
        if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
            break; /* success */
    }
}
```

```

        Close(listenfd);      /* bind error, close and try next one */
    } while ( (res = res->ai_next) != NULL);

    if (res == NULL)      /* errno from final socket() or bind() */
        err_sys("tcp_listen error for %s, %s", host, serv);

    Listen(listenfd, LISTENQ);

    if (addrlenp)
        *addrlenp = res->ai_addrlen;      /* return size of protocol address */

    freeaddrinfo(ressave);

    return(listenfd);
}
/* end tcp_listen */

/*
 * We place the wrapper function here, not in wrplib.c, because some
 * XTI programs need to include wrplib.c, and it also defines
 * a Tcp_listen() function.
 */
int
Tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
{
    return(tcp_listen(host, serv, addrlenp));
}

```

- ```

#include "unp.h"
#include <time.h>

int
main(int argc, char **argv)
{
 int listenfd, connfd;
 socklen_t len;
 char buff[MAXLINE];
 time_t ticks;
 struct sockaddr_storage cliaddr;

 if (argc != 2)
 err_quit("usage: daytimetcpsrv1 <service or port#>");

 listenfd = Tcp_listen(NULL, argv[1], NULL);

 for (; ;) {
 len = sizeof(cliaddr);
 connfd = Accept(listenfd, (SA *)&cliaddr, &len);
 printf("connection from %s\n", Sock_ntop((SA *)&cliaddr, len));

 ticks = time(NULL);
 snprintf(buff, sizeof(buff), "%s.%d", ctime(&ticks));

```

```
 write(connfd, buff, strlen(buff));
```

```
 Close(connfd);
```

```
}
```

```
}
```

- 运行在双栈主机上的IPv6服务器既能处理Ipv4客户，也能处理Ipv6客户，Ipv4客户主机的地址作为Ipv4映射的IPv6地址传递给Ipv6服务器

## 11.14 udp\_client函数

- 创建一个未连接的UDP套接字

```
/* include udp_client */
#include "unp.h"

int
udp_client(const char *host, const char *serv, SA **saptr, socklen_t *lenp)
{
 int sockfd, n;
 struct addrinfo hints, *res, *ressave;

 bzero(&hints, sizeof(struct addrinfo));
 hints.ai_family = AF_UNSPEC;
 hints.ai_socktype = SOCK_DGRAM;

 if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
 err_quit("udp_client error for %s, %s: %s",
 host, serv, gai_strerror(n));
 ressave = res;

 do {
 sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
 if (sockfd >= 0)
 break; /* success */
 } while ((res = res->ai_next) != NULL);

 if (res == NULL) /* errno set from final socket() */
 err_sys("udp_client error for %s, %s", host, serv);

 *saptr = Malloc(res->ai_addrlen);
 memcpy(*saptr, res->ai_addr, res->ai_addrlen);
 *lenp = res->ai_addrlen;

 freeaddrinfo(ressave);

 return(sockfd);
}
/* end udp_client */

int
Udp_client(const char *host, const char *serv, SA **saptr, socklen_t *lenptr)
{
```

```

 return(udp_client(host, serv, saptr, lenptr));
}

• #include "unp.h"

int
main(int argc, char **argv)
{
 int sockfd, n;
 char recvline[MAXLINE + 1];
 socklen_t salen;
 struct sockaddr *sa;

 if (argc != 3)
 err_quit("usage: daytimeudpccli1 <hostname/IPaddress> <service/port#>");

 sockfd = Udp_client(argv[1], argv[2], (void **) &sa, &salen);

 printf("sending to %s\n", Sock_ntop_host(sa, salen));

 Sendto(sockfd, "", 1, 0, sa, salen); /* send 1-byte datagram */

 n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
 recvline[n] = '\0'; /* null terminate */
 Fputs(recvline, stdout);

 exit(0);
}

```

## 11.15 udp\_connect函数

- 创建一个已连接UDP套接字

```

• /* include udp_connect */
#include "unp.h"

int
udp_connect(const char *host, const char *serv)
{
 int sockfd, n;
 struct addrinfo hints, *res, *ressave;

 bzero(&hints, sizeof(struct addrinfo));
 hints.ai_family = AF_UNSPEC;
 hints.ai_socktype = SOCK_DGRAM;

 if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
 err_quit("udp_connect error for %s, %s: %s",
 host, serv, gai_strerror(n));
 ressave = res;

 do {

```

```

 sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
 if (sockfd < 0)
 continue; /* ignore this one */

 if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
 break; /* success */

 Close(sockfd); /* ignore this one */
 } while ((res = res->ai_next) != NULL);

 if (res == NULL) /* errno set from final connect() */
 err_sys("udp_connect error for %s, %s", host, serv);

 freeaddrinfo(ressave);

 return(sockfd);
}
/* end udp_connect */

int
Udp_connect(const char *host, const char *serv)
{
 int n;

 if ((n = udp_connect(host, serv)) < 0) {
 err_quit("udp_connect error for %s, %s: %s",
 host, serv, gai_strerror(-n));
 }
 return(n);
}

```

## 11.16 udp\_server函数

```

/* include udp_server */
#include "unp.h"

int
udp_server(const char *host, const char *serv, socklen_t *addrlenp)
{
 int sockfd, n;
 struct addrinfo hints, *res, *ressave;

 bzero(&hints, sizeof(struct addrinfo));
 hints.ai_flags = AI_PASSIVE;
 hints.ai_family = AF_UNSPEC;
 hints.ai_socktype = SOCK_DGRAM;

 if ((n = getaddrinfo(host, serv, &hints, &res)) != 0)
 err_quit("udp_server error for %s, %s: %s",
 host, serv, gai_strerror(n));
 ressave = res;

```

```

do {
 sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
 if (sockfd < 0)
 continue; /* error - try next one */

 if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0)
 break; /* success */

 Close(sockfd); /* bind error - close and try next one */
} while ((res = res->ai_next) != NULL);

if (res == NULL) /* errno from final socket() or bind() */
 err_sys("udp_server error for %s, %s", host, serv);

if (addrlenp)
 addrlenp = res->ai_addrlen; / return size of protocol address */

freeaddrinfo(ressave);

return(sockfd);
}
/* end udp_server */

int
Udp_server(const char *host, const char *serv, socklen_t *addrlenp)
{
 return(udp_server(host, serv, addrlenp));
}

```

## 11.17 getnameinfo函数

- 以一个套接字地址为参数，返回描述其中的主机的一个字符串和描述其中的服务的另一个字符串
- sock\_ntop和getnameinfo的差别在于，前者不涉及DNS,只返回IP地址和端口号的一个可显示版本，后者通常尝试获取主机和服务的名字

| 常 值             | 说 明              |
|-----------------|------------------|
| NI_DGRAM        | 数据报服务            |
| NI_NAMEREQD     | 若不能从地址解析出名字则返回错误 |
| NI_NOFQDN       | 只返回FQDN的主机名部分    |
| NI_NUMERICHOST  | 以数串格式返回主机字符串     |
| NI_NUMERICSCOPE | 以数串格式返回范围标识字符串   |
| NI_NUMERICSERV  | 以数串格式返回服务字符串     |

图11-20 getnameinfo的标志值

## 11.18 可重入函数

- 进程的主控制流被暂停以执行信号处理函数
- gethostbyname、gethostaddr、getservbyname getservbyport不是可重入的,因为它们返回指向同一个静态结构的指针
- 支持线程的一些实现同时提供这4个函数的可重入版本，以\_r结尾

- 支持线程的另一些实现，使用线程特定数据提供这些函数的可重入版本
- inet\_nton inet\_ntop是可重入的
- 许多版本的标准I/O函数库是不可重入的

## 11.19 gethostbyname\_r和gethostbyaddr\_r函数

- 把不可重入函数填写并返回静态结构的做法改为由调用者分配再由可重入函数填写结构
- 由可重入函数调用malloc以动态分配内存空间

## 11.20 作废的IPv6地址解析函数

- RES\_USE\_INET6常值
- gethostbyname2
- getipnodebyname

## 11.21 其他网络相关信息

所有四类信息都可以存放在一个文件中，每类信息各定义有三个访问函数：

- (1) 函数getXXXent读出文件中的下一个表项，必要的话首先打开文件；
- (2) 函数setXXXent打开（如果尚未打开的话）并回绕文件；
- (3) 函数endXXXent关闭文件。

每类信息都定义了各自的结构，包括hostent、netent、protoent和servent。这些定义通过包含头文件<netdb.h>提供。

除了用于顺序处理文件的get、set和end这三个函数外，每类信息还提供一些键值查找(keyed lookup)函数。这些函数顺序遍历整个文件（通过调用getXXXent函数读出每一行），但是不把每一行都返回给调用者，而是寻找与某个参数匹配的一个表项。这些键值查找函数具有形如getXXXbyYYY的名字。举例来说，针对主机信息的两个键值查找函数是gethostbyname(查找匹配某个主机名的表项)和gethostbyaddr(查找匹配某个IP地址的表项)。图11-21汇总了这些信息。

| 信息 | 数据文件           | 结构       | 键值查找函数                           |
|----|----------------|----------|----------------------------------|
| 主机 | /etc/hosts     | hostent  | gethostbyaddr, gethostbyname     |
| 网络 | /etc/networks  | netent   | getnetbyaddr, getnetbyname       |
| 协议 | /etc/protocols | protoent | getprotobynumber, getprotobyname |
| 服务 | /etc/services  | servent  | getservbyname, getservbyport     |

图11-21 四类网络相关信息

在使用DNS的前提下如何应用这些函数呢？首先，只有主机和网络信息可通过DNS获取，协议和服务信息总是从相应的文件中读取。我们早先在本章中（随图11-1）提到过，不同的实现有不同的方法供系统管理员指定是使用DNS还是使用文件来查找主机和网络信息。

- 只有主机和网络信息可通过DNS获取，协议和服务信息总是从相应的文件中读取
- man resolver 直接调用解析器函数的手册

## 第12章 IPv4与IPv6的互操作性

### 12.2 IPv4客户与IPv6服务器

-

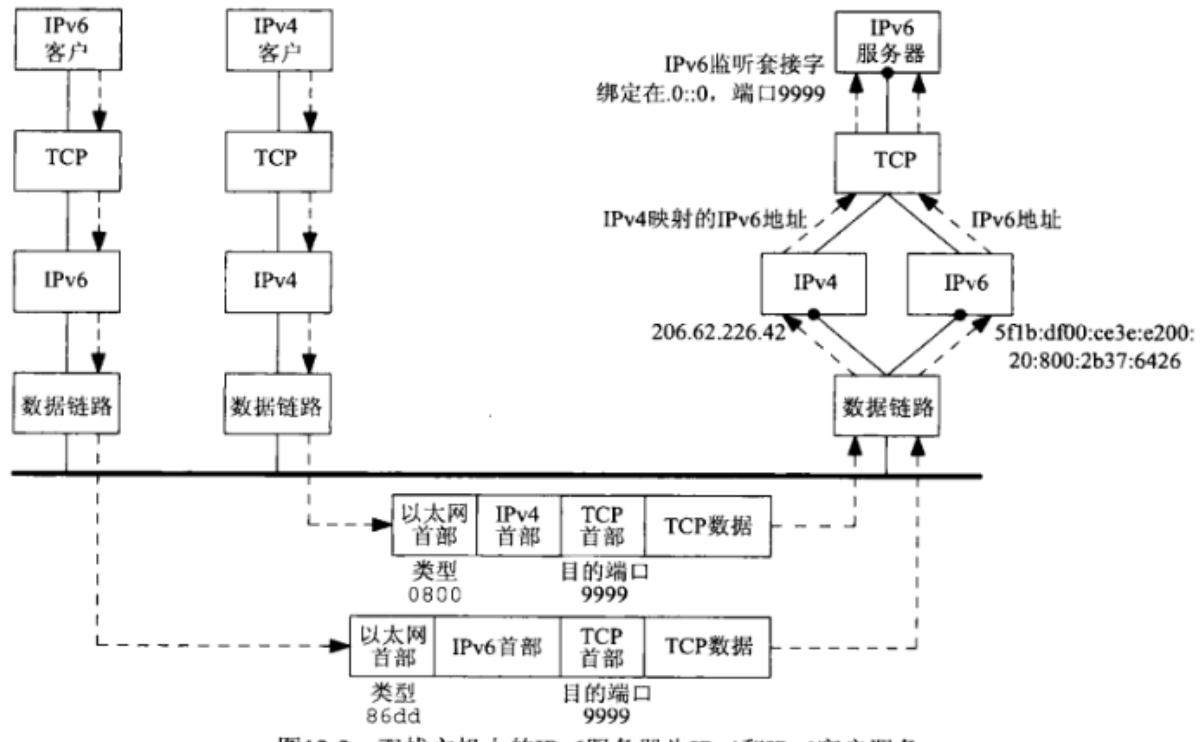


图12-2 双栈主机上的IPv6服务器为IPv4和IPv6客户服务

- 双栈主机

(1) IPv4监听套接字只能接受来自IPv4客户的外来连接。

(2) 如果服务器有一个绑定了通配地址的IPv6监听套接字，而且该套接字未设置IPV6\_V6ONLY套接字选项（7.8节），那么该套接字既能接受来自IPv4客户的外来连接，又能接受来自IPv6客户的外来连接。对于来自IPv4客户的连接而言，其服务器端的本地地址将是与某个本地IPv4地址对应的IPv4映射的IPv6地址。

(3) 如果服务器有一个IPv6监听套接字，而且绑定在其上的是除IPv4映射的IPv6地址之外的某个非通配IPv6地址，或者绑定在其上的是通配地址，不过还设置了IPV6\_V6ONLY套接字选项（7.8节），那么该套接字只能接受来自IPv6客户的外来连接。

### 12.3 IPv6客户与IPv4服务器

- 如果一个IPv6的TCP客户指定一个IPv4地址映射的IPv6地址以调用connect,或者一个IPv6的UDP客户指定一个IPv4映射的IPv6地址以调用sendto,那么内核检测到这个映射地址后改为发送一个IPv4数据报而不是IPv6数据报

|                  | IPv4服务器IPv4单栈主机(纯A) | IPv6服务器IPv6单栈主机(纯AAAA) | IPv4服务器双栈主机(A和AAAA) | IPv6服务器双栈主机(A和AAAA) |
|------------------|---------------------|------------------------|---------------------|---------------------|
| IPv4客户, IPv4单栈主机 | IPv4                | (无)                    | IPv4                | IPv4                |
| IPv6客户, IPv6单栈主机 | (无)                 | IPv6                   | (无)                 | IPv6                |
| IPv4客户, 双栈主机     | IPv4                | (无)                    | IPv4                | IPv4                |
| IPv6客户, 双栈主机     | IPv4                | IPv6                   | (无*)                | IPv6                |

图12-5 IPv4和IPv6客户与服务器互操作性总结

图标为“IPv4”或“IPv6”的栏目表示相应组合有效，并指出了实际使用的协议；标为“(no)”的栏目表示相应组合无效。最后一行第三列标了星号，因为该栏目的互操作性取决于客户选择的地址。如果选择AAAA记录从而发送IPv6数据报，那就不能工作。然而如果选择A记录，

### 12.4 IPv6地址测试宏

```

#include <netinet/in.h>

int IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *aptr);
int IN6_IS_ADDR_LOOPBACK(const struct in6_addr *aptr);
int IN6_IS_ADDR_MULTICAST(const struct in6_addr *aptr);
int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4MAPPED(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4COMPAT(const struct in6_addr *aptr);

int IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr *aptr);

```

均返回：若IPv6地址归属指定类型则为非0，否则为0

## 12.5 源代码可移植性

## 第13章 守护进程和inetd超级服务器

- 守护进程是在后台运行且不与任何控制终端关联的进程
- 守护进程的启动方法

(1) 在系统启动阶段，许多守护进程由系统初始化脚本启动。这些脚本通常位于/etc目录或以/etc/rc开头的某个目录中，它们的具体位置和内容却是实现相关的。由这些脚本启动的守护进程一开始时拥有超级用户特权。

有若干个网络服务器通常从这些脚本启动：inetd超级服务器（见下一条）、Web服务器、邮件服务器（经常是sendmail）。我们将在13.2节讲解的syslogd守护进程通常也由某个系统初始化脚本启动。

- (2) 许多网络服务器由将在本章靠后介绍的inetd超级服务器启动。inetd自身由上一条中的某个脚本启动。inetd监听网络请求（Telnet、FTP等），每当有一个请求到达时，启动相应的实际服务器（Telnet服务器、FTP服务器等）。

(3) cron守护进程按照规则定期执行一些程序，而由它启动执行的程序同样作为守护进程运行。cron自身由第1条启动方法中的某个脚本启动。

(4) at命令用于指定将来某个时刻的程序执行。这些程序的执行时刻到来时，通常由cron守护进程启动执行它们，因此这些程序同样作为守护进程运行。

(5) 守护进程还可以从用户终端或在前台或在后台启动。这么做往往是为了测试守护程序或重启因某种原因而终止了的某个守护进程。

## 13.2 syslogd守护进程

- syslogd守护进程通常由某个系统初始化脚本启动
- syslogd在启动时执行以下步骤
  - 读取配置文件，/etc/syslog.cnf
  - 创建一个UNix域套接字，捆绑路径名
  - 创建一个UDP套接字，捆绑端口514
  - 打开路径名/dev/klog
  - 调用select等待以上三个描述符

## 13.3 syslog函数

- 当syslog被应用进程首次调用时，它创建一个Unix域套接字，然后调用connect连接到由syslogd守护进程创建的Unix域数据报套接字的众所周知的路径名

## 13.4 daemon\_init函数

- 把一个普通进程转化为守护进程

```

#include "unp.h"
#include <syslog.h>

#define MAXFD 64

extern int daemon_proc; /* defined in error.c */

int
daemon_init(const char *pname, int facility)
{
 int i;
 pid_t pid;

 if ((pid = Fork()) < 0)
 return (-1);
 else if (pid)
 _exit(0); /* parent terminates */

 /* child 1 continues... */

 if (setsid() < 0) /* become session leader */
 return (-1);

 Signal(SIGHUP, SIG_IGN);
 if ((pid = Fork()) < 0)
 return (-1);
 else if (pid)
 _exit(0); /* child 1 terminates */

 /* child 2 continues... */

 daemon_proc = 1; /* for err_XXX() functions */

 chdir("/"); /* change working directory */

 /* close off file descriptors */
 for (i = 0; i < MAXFD; i++)
 close(i);

 /* redirect stdin, stdout, and stderr to /dev/null */
 open("/dev/null", O_RDONLY);
 open("/dev/null", O_RDWR);
 open("/dev/null", O_RDWR);

 openlog(pname, LOG_PID, facility);

```

```

 return (0); /* success */
}

```

## 13.5 inetd守护进程

- 通过由inetd处理普通守护进程的大部分启动细节以简化守护进程的编写
- 单个进程(inetd)就能为多个服务等待外来的客户请求，以此取代每个服务一个进程的做法
- inetd进程处理配置文件/etc/inetd.conf的配置文件指定本超级服务器处理哪些服务以及当一个服务到达时该怎么做

| 字 段                             | 说 明                             |
|---------------------------------|---------------------------------|
| <i>service-name</i>             | 必须在/etc/services文件中定义           |
| <i>socket-type</i>              | stream (对于TCP) 或dgram (对于UDP)   |
| <i>protocol</i>                 | 必须在/etc/protocols文件中定义: tcp或udp |
| <i>wait-flag</i>                | 对于TCP一般为nowait, 对于UDP一般为wait    |
| <i>login-name</i>               | 来自/etc/passwd的用户名, 一般为root      |
| <i>server-program</i>           | 调用exec指定的完整路径名                  |
| <i>server-program-arguments</i> | 调用exec指定的命令行参数                  |

图13-6 inetd.conf文件中的字段

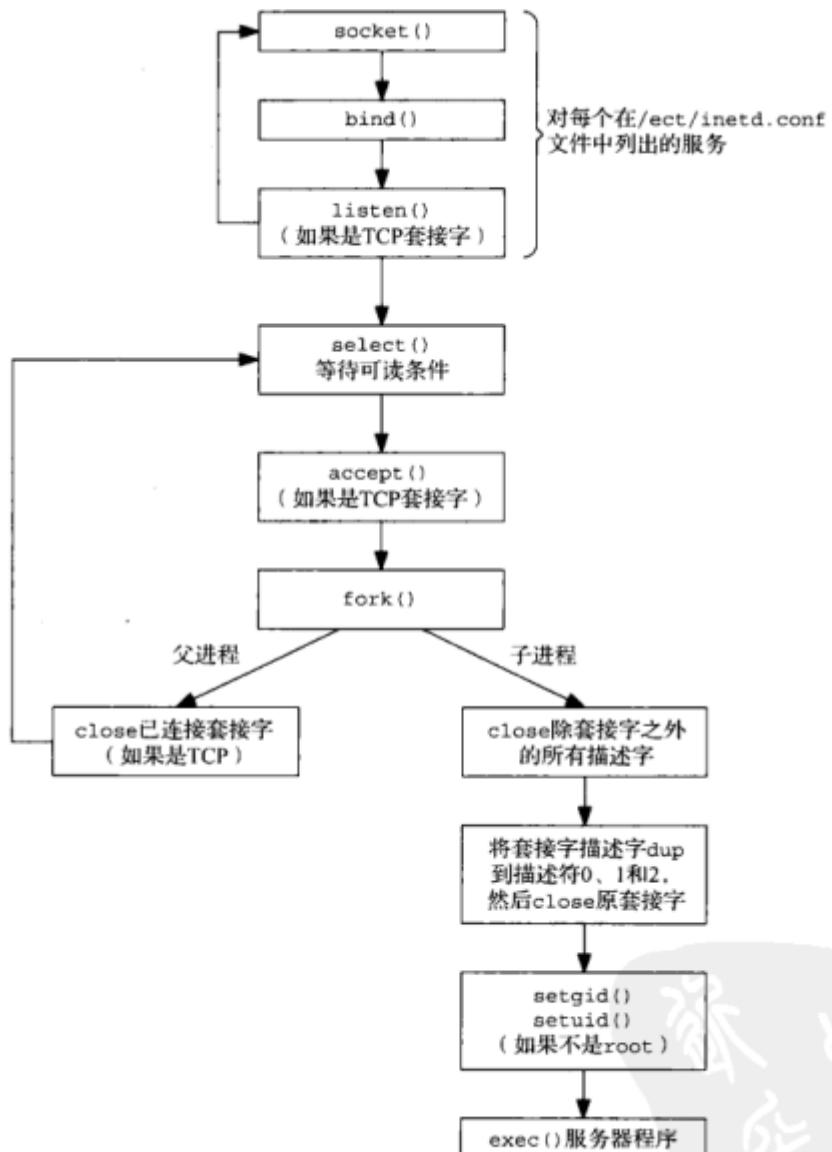


图13-7 inetd的工作流程

- Linux中的xinetd

## 13.6 daemon\_inetd函数

### 第14章 高级I/O函数

#### 14.2 套接字超时

- 使用SIGALRM为connect设置超时

```
o /* include connect_timeo */
#include "unp.h"

static void connect_alarm(int);

int
connect_timeo(int sockfd, const SA *saptr, socklen_t salen, int nsec)
{
 Sigfunc *sigfunc;
 int n;

 sigfunc = Signal(SIGALRM, connect_alarm);
 if (alarm(nsec) != 0)
 err_msg("connect_timeo: alarm was already set");

 if ((n = connect(sockfd, saptr, salen)) < 0) {
 close(sockfd);
 if (errno == EINTR)
 errno = ETIMEDOUT;
 }
 alarm(0); /* turn off the alarm */
 Signal(SIGALRM, sigfunc); /* restore previous signal handler */

 return(n);
}

static void
connect_alarm(int signo)
{
 return; /* just interrupt the connect() */
}
/* end connect_timeo */

void
Connect_timeo(int fd, const SA *sa, socklen_t salen, int sec)
{
 if (connect_timeo(fd, sa, salen, sec) < 0)
 err_sys("connect_timeo error");
}
```

- 使用了系统调用的可中断能力

- 在多线程化程序中正确使用信号非常困难
- 使用SIGALRM为recvfrom设置超时

```

◦ #include "unp.h"

static void sig_alarm(int);

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
 int n;
 char sendline[MAXLINE], recvline[MAXLINE + 1];

 Signal(SIGALRM, sig_alarm);

 while (Fgets(sendline, MAXLINE, fp) != NULL) {

 Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 alarm(5);
 if ((n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL)) < 0) {
 if (errno == EINTR)
 fprintf(stderr, "socket timeout\n");
 else
 err_sys("recvfrom error");
 } else {
 alarm(0);
 recvline[n] = 0; /* null terminate */
 Fputs(recvline, stdout);
 }
 }

 static void
 sig_alarm(int signo)
 {
 return; /* just interrupt the recvfrom() */
 }
}

```

- 使用select为recvfrom设置超时

```

◦ /* include readable_timeo */
#include "unp.h"

int
readable_timeo(int fd, int sec)
{
 fd_set rset;
 struct timeval tv;

 FD_ZERO(&rset);
 FD_SET(fd, &rset);
}

```

```

 tv.tv_sec = sec;
 tv.tv_usec = 0;

 return(select(fd+1, &rset, NULL, NULL, &tv));
 /* > 0 if descriptor is readable */
 }
/* end readable_timeo */

int
Readable_timeo(int fd, int sec)
{
 int n;

 if ((n = readable_timeo(fd, sec)) < 0)
 err_sys("readable_timeo error");
 return(n);
}

#include "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
 int n;
 char sendline[MAXLINE], recvline[MAXLINE + 1];

 while (Fgets(sendline, MAXLINE, fp) != NULL) {

 Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 if (Readable_timeo(sockfd, 5) == 0) {
 fprintf(stderr, "socket timeout\n");
 } else {
 n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
 recvline[n] = 0; /* null terminate */
 Fputs(recvline, stdout);
 }
 }
}

```

- 使用SO\_RCVTIMEO套接字选项为recvfrom设置超时
  - 本选项一旦设置到某个描述符上，其超时设置将应用于该描述符上的所有读操作
  - SO\_SNDTIMEO选项仅仅应用于写操作
  - 都不能为connect设置超时

```

#include "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
 int n;

```

```

char sendline[MAXLINE], recvline[MAXLINE + 1];
struct timeval tv;

tv.tv_sec = 5;
tv.tv_usec = 0;
Setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));

while (Fgets(sendline, MAXLINE, fp) != NULL) {

 Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
 if (n < 0) {
 if (errno == EWOULDBLOCK) {
 fprintf(stderr, "socket timeout\n");
 continue;
 } else
 err_sys("recvfrom error");
 }

 recvline[n] = 0; /* null terminate */
 Fputs(recvline, stdout);
}
}

```

## 14.3 recv和send函数

- 类似于标准的read和write函数，不过需要一个额外的参数
- flags参数
  - MSG\_DONTROUTE
  - MSG\_DONTWAIT,在无需打开相应套接字的非阻塞标志的前提下，把单个I/O操作临时指定为非阻塞
  - MSG\_OOB 发送或接收带外数据，TCP连接上只有一个字节可以作为带外数据发送
  - MSG\_PEEK
  - MSG\_WAITALL 告知内核不要在尚未读入请求数目的字节之前让一个读操作返回

## 14.4 readv和writev函数

- 类似于read和write,不过readv和writev允许单个系统调用读入或写出自一个或多个缓冲区
- writev是一个原子操作，对于一个基于记录的协议(如UDP),一次writev调用只产生单个UDP数据报

## 14.5 recvmsg和sendmsg函数

- 最通用的I/O函数，可以把所有read readv recv和recvfrom调用替换成recvmsg,各种输出函数调用也可以替换成sendmsg
- 这两个函数把大部分参数封装到一个msghdr结构中
-

```

struct msghdr {
 void *msg_name; /* protocol address */
 socklen_t msg_namelen; /* size of protocol address */
 struct iovec *msg iov; /* scatter/gather array */
 int msg iovlen; /* # elements in msg iov */
 void *msg_control; /* ancillary data (cmsghdr struct) */
 socklen_t msg_controllen; /* length of ancillary data */
 int msg_flags; /* flags returned by recvmsg() */
};


```

- msg\_name和msg\_namelen用于套接字未连接的场合
- msg iov和msg iovlen指定输入或输出缓冲区数组
- msg\_control和msg\_controllen指定可选的辅助数据的位置和大小
- 只有recvfrom使用msg\_flags

| 标志               | 由内核检查send、<br>sendto或sendmsg函数<br>的flags参数 | 由内核检查recv、<br>recvfrom或recvmsg函<br>数的flags参数 | 由内核通过recvmsg函<br>数的msg_flags结构参<br>数成员返回 |
|------------------|--------------------------------------------|----------------------------------------------|------------------------------------------|
| MSG_DONTROUTE    | •                                          |                                              |                                          |
| MSG_DONTWAIT     | •                                          | •                                            |                                          |
| MSG_PEEK         |                                            | •                                            |                                          |
| MSG_WAITALL      |                                            | •                                            |                                          |
| MSG_EOR          | •                                          | •                                            | •                                        |
| MSG_OOB          | •                                          |                                              | •                                        |
| MSG_BCAST        |                                            |                                              | •                                        |
| MSG_MCAST        |                                            |                                              | •                                        |
| MSG_TRUNC        |                                            |                                              | •                                        |
| MSG_CTRUNC       |                                            |                                              | •                                        |
| MSG_NOTIFICATION |                                            |                                              |                                          |

图14-7 各种I/O函数输入和输出标志的总结

这些标志中，内核只检查而不返回前4个标志，既检查又返回接下来的2个标志，不检查而只返回后4个标志。recvmsg返回的7个标志解释如下。

MSG\_BCAST

本标志随BSD/OS引入，相对较新。它的返回条件是本数据报作为链路层广播收取或者其目的IP地址是一个广播地址。与IP\_RECV DSTADDR套接字选项相比，本标志是用于判定一个UDP数据报是否发往某个广播地址的更好方法。

MSG\_MCAST

本标志随BSD/OS引入，相对较新。它的返回条件是本数据报作为链路层多播收取。

MSG\_TRUNC

本标志的返回条件是本数据报被截断，也就是说，内核预备返回的数据超过进程事先分配的空间（所有iov\_len成员之和）。我们将在22.3节详细讨论本问题。

MSG\_CTRUNC

本标志的返回条件是本数据报的辅助数据被截断，也就是说，内核预备返回的辅助数据超过进程事先分配的空间(msg\_controllen)。

MSG\_EOR

本标志的返回条件是返回数据结束一个逻辑记录。TCP不使用本标志，因为它是一个字节流协议。

MSG\_OOB

本标志绝不为TCP带外数据返回。它用于其他协议族（例如OSI协议族）。

MSG\_NOTIFICATION

本标志由SCTP接收者返回，指示读入的消息是一个事件通知，而不是数据消息。

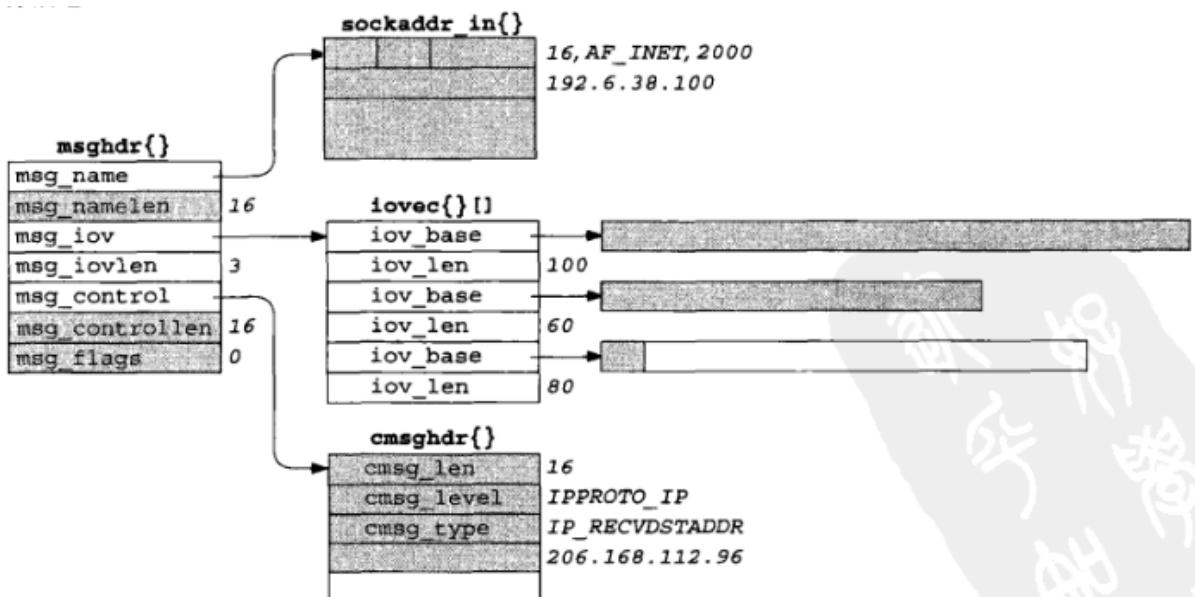


图14-9 recvmsg返回时对图14-8的更新

- 5组I/O函数的比较

| 函 数              | 任 何<br>描 述 符 | 仅 套 接 字<br>描 述 符 | 单 个 读 / 写<br>缓 缸 区 | 分 散 / 集 中<br>读 / 写 | 可 选 标 志 | 可 选 对 端<br>地 址 | 可 选 控 制<br>信 息 |
|------------------|--------------|------------------|--------------------|--------------------|---------|----------------|----------------|
| read, write      | ●            |                  | ●                  |                    |         |                |                |
| readv, writev    | ●            |                  |                    | ●                  |         |                |                |
| recv, send       |              | ●                | ●                  |                    | ●       |                |                |
| recvfrom, sendto |              | ●                | ●                  |                    | ●       | ●              |                |
| recvmsg, sendmsg |              | ●                |                    | ●                  | ●       | ●              | ●              |

图14-10 5组I/O函数的比较

## 14.6 辅助数据

- 控制信息

| 协 议   | cmsg_level   | cmsg_type                                                                                                  | 说 明                                                                                    |
|-------|--------------|------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| IPv4  | IPPROTO_IP   | IP_RECVSTADDR<br>IP_RECVIF                                                                                 | 随 UDP 数据报接收目的地址<br>随 UDP 数据报接收接口索引                                                     |
| IPv6  | IPPROTO_IPV6 | IPV6_DSTOPTS<br>IPV6_HOPLIMIT<br>IPV6_HOPOPTS<br>IPV6_NEXTHOP<br>IPV6_PKTINFO<br>IPV6_RTHDR<br>IPV6_TCLASS | 指定/接收目的地选项<br>指定/接收跳限<br>指定/接收步跳选项<br>指定下一跳地址<br>指定/接收分组信息<br>指定/接收路由首部<br>指定/接收分组流通类别 |
| Unix域 | SOL_SOCKET   | SCM_RIGHTS<br>SCM_CREDS                                                                                    | 发送/接收描述符<br>发送/接收用户凭证                                                                  |

图14-11 辅助数据用途的总结

```

• struct cmsghdr{
 socklen_t cmsg_len;
 int cmsg_level;
 int cmsg_level;
 /*unsigned char cmsg_data[] */
};
```

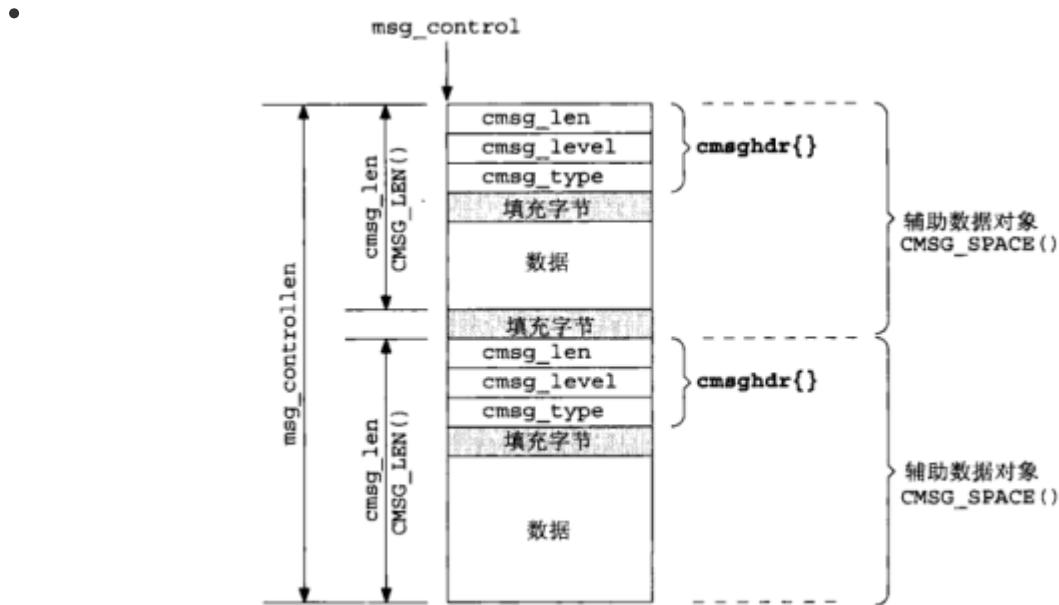


图14-12 包含两个辅助数据对象的辅助数据

- 由`recvmsg`返回的辅助数据可含有任意数目的辅助数据对象，为了对应用程序屏蔽可能出现的填充字节，定义了5个宏
- `CMSG_FIRSTHDR`
- `CMSG_NXTHDR`
- `CMSG_DATA`
- `CMSG_LEN`
- `CMSG_SPACE`

## 14.7 排队的数据量

- 使用非阻塞式I/O
- 使用`MSG_PEEK`标志，结合`MSG_DONTWAIT`使用
- 使用`ioctl`的`FIONREAD`命令，该命令的第三个参数，内核通过该参数返回的值就是套接字接收队列的当前字节数

## 14.8 套接字和标准I/O

- Unix I/O, 内核系统调用，围绕描述符工作
- 标准I/O函数库可用于套接字，注意以下几点
  - `fdopen`, `fileno`
  - 为一个给定套接字打开两个标准I/O流：一个用于读，一个用于写

```
#include <unp.h>

void
str_echo(int sockfd)
{
 char line[MAXLINE];
 FILE *fpin, *fout;
 fpin = Fdopen(sockfd, "r");
 fout = Fdopen(sockfd, "w");
 while (fgets(line, MAXLINE, fpin) != NULL)
 if (fputs(line, fout) == EOF)
 error("Fputs error");
}
```

```

fpout = Fdopen(sockfd, "w");

while (Fgets(line, MAXLINE, fpin) != NULL)
 Fputs(line, fpout);
}

```

- 服务器的标准I/O流被标准I/O函数库完全缓冲，意味着该函数库把回射行复制到输出流的标准I/O缓冲区，但是不把该缓冲区中的内容写到描述符，因为该缓冲区未满
- 标准I/O函数库执行以下缓冲
  - 完全缓冲：以下情况才发生I/O
  - 行缓冲
  - 不缓冲
- 尽量避免在套接字上使用标准I/O函数库

## 14.9 高级轮询技术

- /dev/poll接口

```

◦ #include "unp.h"
#include <sys/devpoll.h>

void
str_cli(FILE *fp, int sockfd)
{
 int stdineof;
 char buf[MAXLINE];
 int n;
 int wfd;
 struct pollfd pollfd[2];
 struct devpoll dopoll;
 int i;
 int result;

 wfd = Open("/dev/poll", O_RDWR, 0);

 pollfd[0].fd = fileno(fp);
 pollfd[0].events = POLLIN;
 pollfd[0].revents = 0;

 pollfd[1].fd = sockfd;
 pollfd[1].events = POLLIN;
 pollfd[1].revents = 0;

 Write(wfd, pollfd, sizeof(struct pollfd) * 2);

 stdineof = 0;
 for (; ;) {
 /* block until /dev/poll says something is ready */
 dopoll.dp_timeout = -1;
 dopoll.dp_nfds = 2;
 dopoll.dp_fds = pollfd;
 result = Iioctl(wfd, DP_POLL, &dopoll);
 }
}

```

```

 /* loop through ready file descriptors */
 for (i = 0; i < result; i++) {
 if (dopoll.dp_fds[i].fd == sockfd) {
 /* socket is readable */
 if ((n = Read(sockfd, buf, MAXLINE)) == 0) {
 if (stdineof == 1)
 return; /* normal
termination */
 else
 err_quit("str_cli: server terminated
prematurely");
 }
 Write(fdno(stdout), buf, n);
 } else {
 /* input is readable */
 if ((n = Read(fdno(fp), buf, MAXLINE)) == 0) {
 stdineof = 1;
 Shutdown(sockfd, SHUT_WR); /* send FIN
*/
 continue;
 }
 Writen(sockfd, buf, n);
 }
 }
 }
}

```

- kqueue接口

- 本接口允许进程向内核注册描述所关注kqueue事件的事件过滤器
- kqueue
- kevent
- EV\_SET
- ```

#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>

int kqueue(void);
int kevent(int kq, const struct kevent *changelist, int nchanges,
          struct kevent *eventlist, int nevents,
          const struct timespec *timeout);
void EV_SET(struct kevent *kev, uintptr_t ident, short filter,
           u_short flags, u_int fflags, intptr_t data, void *udata);

```
-

```

struct kevent {
    uintptr_t ident;      /* identifier (e.g., file descriptor) */
    short filter;         /* filter type (e.g., EVFILT_READ) */
    u_short flags;        /* action flags (e.g., EV_ADD) */
    u_int fflags;         /* filter-specific flags */
    uintptr_t data;        /* filter-specific data */
    void *udata;          /* opaque user data */
};


```

其中`flags`成员在调用时指定过滤器更改行为，在返回时额外给出条件，如图14-16所示。

<code>flags</code>	说 明	更 改	返 回
EV_ADD	增设事件：自动启用，除非同时指定EV_DISABLE	•	
EV_CLEAR	用户获取后复位事件状态	•	
EV_DELETE	删除事件	•	
EV_DISABLE	禁用事件但不删除	•	
EV_ENABLE	重新启用先前禁用的事件	•	
EV_ONESHOT	触发一次后删除事件	•	
EV_EOF	发生EOF条件		•
EV_ERROR	发生错误：errno值在data成员中		•

图14-16 kevent结构的`flags`成员

<code>filter</code>	说 明
EVFILT_AIO	异步I/O事件（6.2节）
EVFILT_PROC	进程exit、fork或exec事件
EVFILT_READ	描述符可读，类似select
EVFILT_SIGNAL	收到信号
EVFILT_TIMER	周期性或一次性的定时器
EVFILT_VNODE	文件修改和删除事件
EVFILT_WRITE	描述符可写，类似select

图14-17 kevent结构的`filter`成员

```

○ #include      "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int                 kq, i, n, nev, stdineof = 0, isfile;
    char                buf[MAXLINE];
    struct kevent   kev[2];
    struct timespec ts;
    struct stat       st;

    isfile = ((fstat(fileno(fp), &st) == 0) &&
              (st.st_mode & S_IFMT) == S_IFREG);

    EV_SET(&kev[0], fileno(fp), EVFILT_READ, EV_ADD, 0, 0, NULL);
    EV_SET(&kev[1], sockfd, EVFILT_READ, EV_ADD, 0, 0, NULL);

    kq = Kqueue();
    ts.tv_sec = ts.tv_nsec = 0;
    Kevent(kq, kev, 2, NULL, 0, &ts);

    for ( ; ; ) {
        nev = Kevent(kq, NULL, 0, kev, 2, NULL);

```

```

        for (i = 0; i < nev; i++) {
            if (kev[i].ident == sockfd) { /* socket is readable */
                if ((n = Read(sockfd, buf, MAXLINE)) == 0) {
                    if (stdineof == 1)
                        return; /* normal
termination */
                    else
                        err_quit("str_cli: server terminated
prematurely");
                }

                Write(fileno(stdout), buf, n);
            }

            if (kev[i].ident == fileno(fp)) { /* input is readable */
                n = Read(fileno(fp), buf, MAXLINE);
                if (n > 0)
                    Writen(sockfd, buf, n);

                if (n == 0 || (isfile && n == kev[i].data)) {
                    stdineof = 1;
                    Shutdown(sockfd, SHUT_WR); /* send FIN
*/
                    kev[i].flags = EV_DELETE;
                    Kevent(kq, &kev[i], 1, NULL, 0, &ts); /* remove kevent */
                    continue;
                }
            }
        }
    }
}

```

14.10 T/TCP: 事物目的TCP

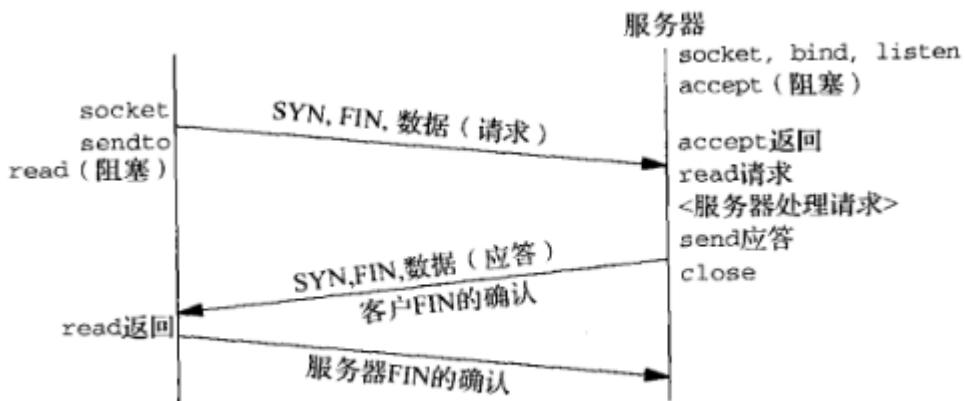


图14-19 最小T/TCP事务的时间线

第15章 Unix域协议

- 在单个主机上执行客户/服务器通信的一种方法

- 提供两类套接字: 字节流套接字 数据报套接字
- Unix域中用于标识客户和服务器的协议地址是普通文件系统中的路径名

15.2 Unix域套接字地址结构

- ```
struct sockaddr_un{
 sa_family sun_family;
 char sun_path[104];
};
```
- POSIX把AF\_UNIX变为AF\_LOCAL
 

```
#include "unp.h"

int
main(int argc, char **argv)
{
 int sockfd;
 socklen_t len;
 struct sockaddr_un addr1, addr2;

 if (argc != 2)
 err_quit("usage: unixbind <pathname>");

 sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);

 unlink(argv[1]); /* OK if this fails */

 bzero(&addr1, sizeof(addr1));
 addr1.sun_family = AF_LOCAL;
 strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path)-1);
 Bind(sockfd, (SA *) &addr1, SUN_LEN(&addr1));

 len = sizeof(addr2);
 Getsockname(sockfd, (SA *) &addr2, &len);
 printf("bound name = %s, returned len = %d\n", addr2.sun_path, len);

 exit(0);
}
```

- 路径名的文件类型为显示为s的套接字

## 15.3 socketpair函数

- 创建两个随后连接起来的套接字，仅适用于Unix域套接字
- 全双工管道

## 15.4 套接字函数

- 由bind创建的路径名默认访问权限应为0777
- 与Unix域套接字关联的路径名应该是一个绝对路径名

- Unix域字节流套接字类似于TCP套接字，为进程提供一个无记录边界的字节流接口
- 在一个未绑定的Unix域套接字上发送数据报不会自动给这个套接字捆绑一个路径名

## 15.5 Unix域字节流客户/服务器程序

```

• #include "unp.h"

int
main(int argc, char **argv)
{
 int listenfd, connfd;
 pid_t childpid;
 socklen_t clilen;
 struct sockaddr_un cliaddr, servaddr;
 void sig_chld(int);

 listenfd = Socket(AF_LOCAL, SOCK_STREAM, 0);

 unlink(UNIXSTR_PATH);
 bzero(&servaddr, sizeof(servaddr));
 servaddr.sun_family = AF_LOCAL;
 strcpy(servaddr.sun_path, UNIXSTR_PATH);

 Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

 Listen(listenfd, LISTENQ);

 Signal(SIGCHLD, sig_chld);

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

 if ((childpid = Fork()) == 0) { /* child process */
 Close(listenfd); /* close listening socket */
 str_echo(connfd); /* process request */
 exit(0);
 }
 Close(connfd); /* parent closes connected socket */
 }
}

```

```

• #include "unp.h"

int
main(int argc, char **argv)
{

```

```

int sockfd;
struct sockaddr_un servaddr;

sockfd = Socket(AF_LOCAL, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
strcpy(servaddr.sun_path, UNIXSTR_PATH);

Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

str_cli(stdin, sockfd); /* do it all */

exit(0);
}

```

## 15.6 Unix域数据报客户/服务器程序

- ```

#include "unp.h"

int
main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_un servaddr, cliaddr;

    sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

    unlink(UNIXDG_PATH);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_LOCAL;
    strcpy(servaddr.sun_path, UNIXDG_PATH);

    Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

    dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
}

```
- ```

#include "unp.h"

int
main(int argc, char **argv)
{
 int sockfd;
 struct sockaddr_un cliaddr, servaddr;

 sockfd = Socket(AF_LOCAL, SOCK_DGRAM, 0);

 bzero(&cliaddr, sizeof(cliaddr)); /* bind an address for us */
 cliaddr.sun_family = AF_LOCAL;
 strcpy(cliaddr.sun_path, tmpnam(NULL));

```

```

 Bind(sockfd, (SA *) &cliaddr, sizeof(cliaddr));

 bzero(&servaddr, sizeof(servaddr)); /* fill in server's address */
 servaddr.sun_family = AF_LOCAL;
 strcpy(servaddr.sun_path, UNIXDG_PATH);

 dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

 exit(0);
}

```

## 15.7 描述符传递

- 首先在这两个进程之间创建一个Unix域套接字，然后使用sendmsg跨这个套接字发送一个特殊消息
- 步骤如下
  - 创建一个字节流的或数据报的Unix域套接字
  - 发送进程通过调用返回描述符的任一Unix函数打开一个描述符
  - 发送进程创建一个msghdr结构，其中含有待传递的描述符，发送一个描述符会使该描述符的引用计数加1
  - 接收进程调用recvmsg在来自步骤1的Unix域套接字上接收这个描述符
- 通过执行另一个程序来打开文件的优势在于。另一个程序可以是一个setuid到root的程序，能够打开我们通常没有打开权限的文件
- ``c++ #include "unp.h"`

```
int my_open(const char *pathname, int mode) { int fd, sockfd[2], status; pid_t childpid; char c,
argsockfd[10], argmode[10];
```

```

Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);

if ((childpid = Fork()) == 0) { /* child process */
 Close(sockfd[0]);
 snprintf(argsockfd, sizeof(argsockfd), "%d", sockfd[1]);
 snprintf(argmode, sizeof(argmode), "%d", mode);
 execl("./openfile", "openfile", argsockfd, pathname, argmode,
 (char *) NULL);
 err_sys("execl error");
}

/* parent process - wait for the child to terminate */
Close(sockfd[1]); /* close the end we don't use */

Waitpid(childpid, &status, 0);
if (WIFEXITED(status) == 0)
 err_quit("child did not terminate");
if ((status = WEXITSTATUS(status)) == 0)
 Read_fd(sockfd[0], &c, 1, &fd);
else {
 errno = status; /* set errno value from child's status */
 fd = -1;
}

```

```
 Close(sockfd[0]);
 return(fd);
}
```

- /\* include read\_fd \*/  
#include "unp.h"  
  
ssize\_t  
read\_fd(int fd, void \*ptr, size\_t nbytes, int \*recvfd)  
{  
 struct msghdr msg;  
 struct iovec iov[1];  
 ssize\_t n;  
  
#ifdef HAVE\_MSGHDR\_MSG\_CONTROL  
 union {  
 struct cmsghdr cm;  
 char control[CMSG\_SPACE(sizeof(int))];  
 } control\_un;  
 struct cmsghdr \*cmptr;  
  
 msg.msg\_control = control\_un.control;  
 msg.msg\_controllen = sizeof(control\_un.control);  
#else  
 int newfd;  
  
 msg.msg\_accrights = (caddr\_t) &newfd;  
 msg.msg\_accrightslen = sizeof(int);  
#endif  
  
 msg.msg\_name = NULL;  
 msg.msg\_namelen = 0;  
  
 iov[0].iov\_base = ptr;  
 iov[0].iov\_len = nbytes;  
 msg.msg iov = iov;  
 msg.msg iovlen = 1;  
  
 if ( (n = recvmsg(fd, &msg, 0)) <= 0)  
 return(n);  
  
#ifdef HAVE\_MSGHDR\_MSG\_CONTROL  
 if ( (cmptr = CMSG\_FIRSTHDR(&msg)) != NULL &&  
 cmptr->cmsg\_len == CMSG\_LEN(sizeof(int))) {  
 if (cmptr->cmsg\_level != SOL\_SOCKET)  
 err\_quit("control level != SOL\_SOCKET");  
 if (cmptr->cmsg\_type != SCM\_RIGHTS)  
 err\_quit("control type != SCM\_RIGHTS");  
 \*recvfd = \*((int \*) CMSG\_DATA(cmptr));  
 }  
#endif

```

 } else
 recvfd = -1; / descriptor was not passed */
#endif
/* *INDENT-OFF* */
if (msg.msg_accrightslen == sizeof(int))
 *recvfd = newfd;
else
 recvfd = -1; / descriptor was not passed */
/* *INDENT-ON* */
#endif

 return(n);
}
/* end read_fd */

ssize_t
Read_fd(int fd, void *ptr, size_t nbytes, int *recvfd)
{
 ssize_t n;

 if ((n = read_fd(fd, ptr, nbytes, recvfd)) < 0)
 err_sys("read_fd error");

 return(n);
}

```

- /\* include write\_fd \*/

```

#include "unp.h"

ssize_t
write_fd(int fd, void *ptr, size_t nbytes, int sendfd)
{
 struct msghdr msg;
 struct iovec iov[1];

#ifndef HAVE_MSGHDR_MSG_CONTROL
 union {
 struct cmsghdr cm;
 char control[CMSG_SPACE(sizeof(int))];
 } control_un;
 struct cmsghdr *cmptr;

```

```

 msg.msg_control = control_un.control;
 msg.msg_controllen = sizeof(control_un.control);

```

```

 cmptr = CMSG_FIRSTHDR(&msg);
 cmptr->cmsg_len = CMSG_LEN(sizeof(int));
 cmptr->cmsg_level = SOL_SOCKET;
 cmptr->cmsg_type = SCM_RIGHTS;
 *((int *) CMSG_DATA(cmptr)) = sendfd;

```

```

#else
 msg.msg_accrights = (caddr_t) &sendfd;
 msg.msg_accrightslen = sizeof(int);

```

```

#endif

 msg.msg_name = NULL;
 msg.msg_namelen = 0;

 iov[0].iov_base = ptr;
 iov[0].iov_len = nbytes;
 msg.msg iov = iov;
 msg.msg iovlen = 1;

 return(sendmsg(fd, &msg, 0));
}

/* end write_fd */

ssize_t
Write_fd(int fd, void *ptr, size_t nbytes, int sendfd)
{
 ssize_t n;

 if ((n = write_fd(fd, ptr, nbytes, sendfd)) < 0)
 err_sys("write_fd error");

 return(n);
}

```

- #include "unp.h"

```

int
main(int argc, char **argv)
{
 int fd;

 if (argc != 4)
 err_quit("openfile <sockfd#> <filename> <mode>");

 if ((fd = open(argv[2], atoi(argv[3]))) < 0)
 exit((errno > 0) ? errno : 255);

 if (write_fd(atoi(argv[1]), "", 1, fd) < 0)
 exit((errno > 0) ? errno : 255);

 exit(0);
}

```

## 15.8 接收发送者的凭证

- 使用cmscred结构传递凭证
- 凭证发送进程必须提供其结构，其内容却是由内核填写的，发送进程无法伪造

- #include "unp.h"

```
#define CONTROL_LEN (sizeof(struct cmsghdr) + sizeof(struct cmscred))
```

```

ssize_t
read_cred(int fd, void *ptr, size_t nbytes, struct cmsgcred *cmmsgcredptr)
{
 struct msghdr msg;
 struct iovec iov[1];
 char control[CONTROL_LEN];
 int n;

 msg.msg_name = NULL;
 msg.msg_namelen = 0;
 iov[0].iov_base = ptr;
 iov[0].iov_len = nbytes;
 msg.msg iov = iov;
 msg.msg iovlen = 1;
 msg.msg control = control;
 msg.msg controllen = sizeof(control);
 msg.msg flags = 0;

 if ((n = recvmsg(fd, &msg, 0)) < 0)
 return(n);

 cmmsgcredptr->cmcred_ngroups = 0; /* indicates no credentials returned */
 if (cmmsgcredptr && msg.msg controllen > 0) {
 struct cmsghdr *cmptr = (struct cmsghdr *) control;

 if (cmptr->cmsg_len < CONTROL_LEN)
 err_quit("control length = %d", cmptr->cmsg_len);
 if (cmptr->cmsg_level != SOL_SOCKET)
 err_quit("control level != SOL_SOCKET");
 if (cmptr->cmsg_type != SCM_CREDS)
 err_quit("control type != SCM_CREDS");
 memcpy(cmmsgcredptr, CMSG_DATA(cmptr), sizeof(struct cmmsgcred));
 }

 return(n);
}

```

- ```
#include      "unp.h"

ssize_t read_cred(int, void *, size_t, struct cmmsgcred *);

void
str_echo(int sockfd)
{
    ssize_t          n;
    int              i;
    char             buf[MAXLINE];
    struct cmmsgcred cred;

again:
    while ( (n = read_cred(sockfd, buf, MAXLINE, &cred)) > 0) {
        if (cred.cmcred_ngroups == 0) {
```

```

        printf("(no credentials returned)\n");
    } else {
        printf("PID of sender = %d\n", cred.cmcred_pid);
        printf("real user ID = %d\n", cred.cmcred_uid);
        printf("real group ID = %d\n", cred.cmcred_gid);
        printf("effective user ID = %d\n", cred.cmcred_euid);
        printf("%d groups:", cred.cmcred_ngroups - 1);
        for (i = 1; i < cred.cmcred_ngroups; i++)
            printf(" %d", cred.cmcred_groups[i]);
        printf("\n");
    }
    written(sockfd, buf, n);
}

if (n < 0 && errno == EINTR)
    goto again;
else if (n < 0)
    err_sys("str_echo: read error");
}

```

第16章 非阻塞式I/O

- 可能阻塞的套接字调用
 - 输入操作: read readv recv recvfrom recvmsg
 - 输出操作: write writev send sendto sendmsg, 内核将从应用进程的缓冲区到该套接字的发送缓冲区复制数据
 - 接受外来连接, 即accept函数
 - 发起外出连接, 即用于TCP的connect函数
- 对于不能被满足的非阻塞式I/O操作, 返回EAGAIN错误或EWOULDBLOCK错误

16.2 非阻塞读和写: str_cli函数

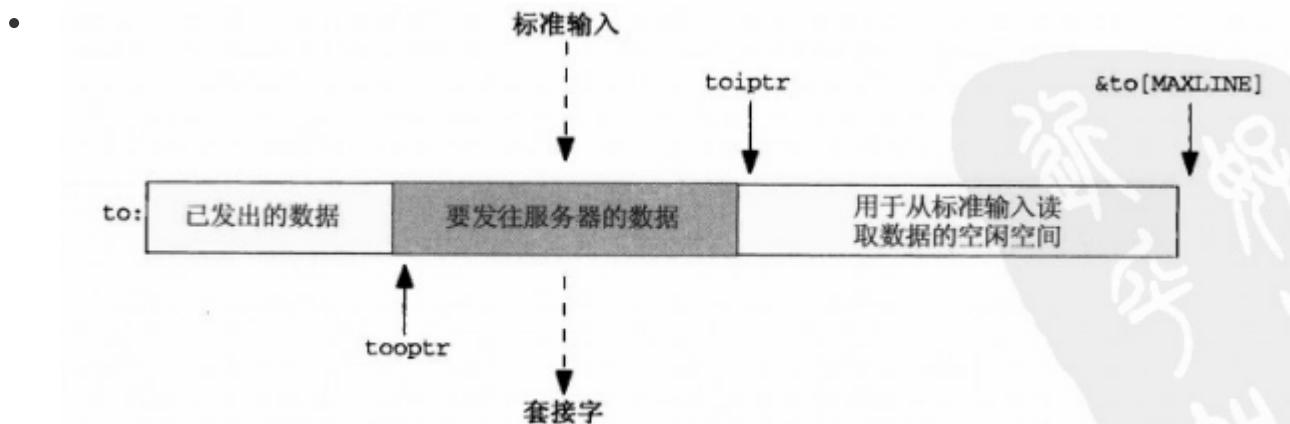


图16-1 容纳从标准输入到套接字的数据的缓冲区

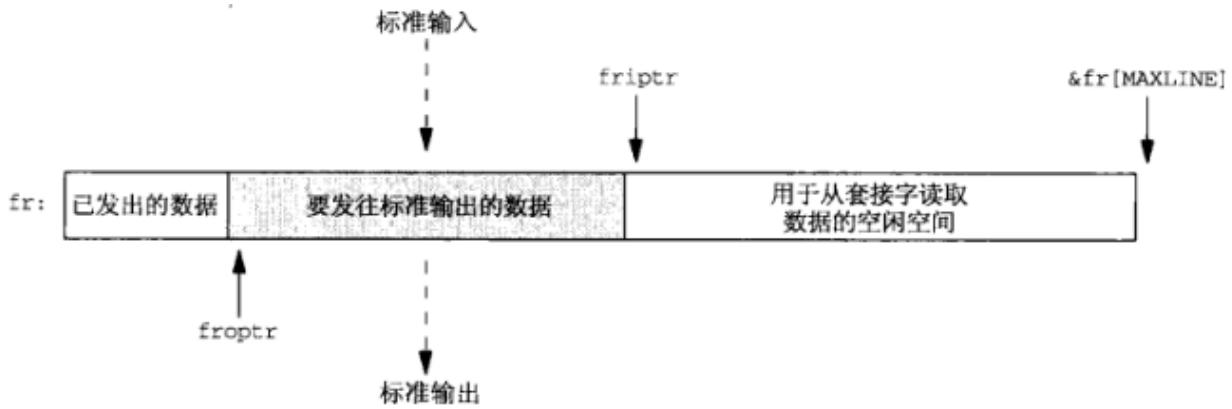


图16-2 容纳从套接字到标准输出的数据的缓冲区

```

• /* include nonb1 */
#include      "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int                 maxfdp1, val, stdineof;
    ssize_t             n, nwritten;
    fd_set              rset, wset;
    char                to[MAXLINE], fr[MAXLINE];
    char                *toiptr, *tooptr, *friptr, *froptr;

    val = Fcntl(sockfd, F_GETFL, 0);
    Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDIN_FILENO, F_GETFL, 0);
    Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
    Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);

    toiptr = tooptr = to; /* initialize buffer pointers */
    friptr = froptr = fr;
    stdineof = 0;

    maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
    for ( ; ; ) {
        FD_ZERO(&rset);
        FD_ZERO(&wset);
        if (stdineof == 0 && toiptr < &to[MAXLINE])
            FD_SET(STDIN_FILENO, &rset); /* read from stdin */
        if (friptr < &fr[MAXLINE])
            FD_SET(sockfd, &rset);           /* read from socket */
        if (tooptr != toiptr)
            FD_SET(sockfd, &wset);         /* data to write to
socket */
        if (froptr != friptr)
            FD_SET(STDOUT_FILENO, &wset); /* data to write to stdout */

        Select(maxfdp1, &rset, &wset, NULL, NULL);
    }
}

```

```

/* end nonb1 */
/* include nonb2 */
    if (FD_ISSET(STDIN_FILENO, &rset)) {
        if ( (n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr)) < 0)
{
            if (errno != EWOULDBLOCK)
                err_sys("read error on stdin");

        } else if (n == 0) {
#endif VOL2
                fprintf(stderr, "%s: EOF on stdin\n", gf_time());
#endif
                stdineof = 1;           /* all done with stdin */
                if (toiptr == toiptr)
                    Shutdown(sockfd, SHUT_WR);/* send FIN */

        } else {
#endif VOL2
                fprintf(stderr, "%s: read %d bytes from stdin\n",
gf_time(), n);
#endif
                toiptr += n;           /* # just read */
                FD_SET(sockfd, &wset); /* try and write to socket below
*/
}
}

if (FD_ISSET(sockfd, &rset)) {
    if ( (n = read(sockfd, friptr, &fr[MAXLINE] - friptr)) < 0) {
        if (errno != EWOULDBLOCK)
            err_sys("read error on socket");

    } else if (n == 0) {
#endif VOL2
            fprintf(stderr, "%s: EOF on socket\n", gf_time());
#endif
            if (stdineof)
                return;           /* normal termination */
            else
                err_quit("str_cli: server terminated
prematurely");

    } else {
#endif VOL2
        fprintf(stderr, "%s: read %d bytes from socket\n",
gf_time(), n);
#endif
        friptr += n;           /* # just read */
        FD_SET(STDOUT_FILENO, &wset); /* try and write below */
    }
}

/* end nonb2 */
/* include nonb3 */

```

```

        if (FD_ISSET(STDOUT_FILENO, &wset) && ( (n = friptr - froptr) > 0)) {
            if ( (nwritten = write(STDOUT_FILENO, froptr, n)) < 0) {
                if (errno != EWOULDBLOCK)
                    err_sys("write error to stdout");

        } else {
#endif VOL2
                fprintf(stderr, "%s: wrote %d bytes to stdout\n",
                        gf_time(), nwritten);
#endif
                froptr += nwritten;           /* # just written */
                if (froptr == friptr)
                    froptr = friptr = fr;   /* back to beginning of
buffer */
            }
        }

        if (FD_ISSET(sockfd, &wset) && ( (n = toiptr - tooptr) > 0)) {
            if ( (nwritten = write(sockfd, tooptr, n)) < 0) {
                if (errno != EWOULDBLOCK)
                    err_sys("write error to socket");

        } else {
#endif VOL2
                fprintf(stderr, "%s: wrote %d bytes to socket\n",
                        gf_time(), nwritten);
#endif
                tooptr += nwritten;           /* # just written */
                if (tooptr == toiptr) {
                    toiptr = tooptr = to;   /* back to beginning of
buffer */
                }
                if (stdineof)
                    Shutdown(sockfd, SHUT_WR);    /* send
FIN */
            }
        }
    }
}
/* end nonb3 */

```

- ```

#include "unp.h"
#include <time.h>

char *
gf_time(void)
{
 struct timeval tv;
 time_t t;
 static char str[30];
 char *ptr;

 if (gettimeofday(&tv, NULL) < 0)

```

```

 err_sys("gettimeofday error");

 t = tv.tv_sec; /* POSIX says tv.tv_sec is time_t; some BSDs don't agree. */
 ptr = ctime(&t);
 strcpy(str, &ptr[11]);
 /* Fri Sep 13 00:00:00 1986\n\0 */
 /* 0123456789012345678901234 5 */
 sprintf(str+8, sizeof(str)-8, ".%06ld", tv.tv_usec);

 return(str);
}

```

- tcpdump -w tcpd tcp and port 7 指定捕获只去往或来自端口7的TCP分节，程序输出到在名为tcpd的文件名中
- str\_cli的较简单版本
- str\_cli执行时间
  - 停等版本
  - select加阻塞式I/O版本
  - 非阻塞式I/O版本
  - fork版本
  - 线程化版本

## 16.3 非阻塞connect

- 当在一个非阻塞的TCP套接字上调用connect时，connect将立即返回一个EINPROGRESS错误，不过已经发起的TCP三路握手继续进行，接着使用select检测这个连接或成功或失败的已建立条件
- 尽管套接字是非阻塞的，如果连接到的服务器在同一个主机上，那么当我们调用connect时，连接通常立刻建立
- 当连接成功建立时，描述符变为可写。当连接建立遇到错误时，描述符变为既可读又可写

## 16.4 非阻塞connect: 时间获取服务器

```

• #include "unp.h"

int
connect_nonb(int sockfd, const SA *saptr, socklen_t salen, int nsec)
{
 int flags, n, error;
 socklen_t len;
 fd_set rset, wset;
 struct timeval tval;

 flags = Fcntl(sockfd, F_GETFL, 0);
 Fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);

 error = 0;
 if ((n = connect(sockfd, saptr, salen)) < 0)
 if (errno != EINPROGRESS)
 return(-1);

 /* Do whatever we want while the connect is taking place. */

```

```

 if (n == 0)
 goto done; /* connect completed immediately */

 FD_ZERO(&rset);
 FD_SET(sockfd, &rset);
 wset = rset;
 tval.tv_sec = nsec;
 tval.tv_usec = 0;

 if ((n = Select(sockfd+1, &rset, &wset, NULL,
 nsec ? &tval : NULL)) == 0) {
 close(sockfd); /* timeout */
 errno = ETIMEDOUT;
 return(-1);
 }

 if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
 len = sizeof(error);
 if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0)
 return(-1); /* Solaris pending error */
 } else
 err_quit("select error: sockfd not set");

done:
 Fcntl(sockfd, F_SETFL, flags); /* restore file status flags */

 if (error) {
 close(sockfd); /* just in case */
 errno = error;
 return(-1);
 }
 return(0);
}

```

- 非阻塞connect是网络编程中最不易移植的部分。避免移植性问题的一个较简单技术是为每个连接创建一个处理线程
- 被中断的connect
  - 如果被中断的connect调用不由内核自动重启，那么它将返回EINTR，不能再次调用connect等待未完成的连接继续完成，这样做将导致返回EADDRINUSE错误。
  - 这种情形下只能调用select，像对于非阻塞connect所做的那样，连接建立成功时select返回套接字可写条件，连接建立失败时select返回套接字既可读又可写条件

## 16.5 非阻塞connect:Web客户端程序

- 同时处理多个非阻塞connect
- 一个流行的Web应用程序即Netscape浏览器使用的特性之一
- 同时打开多个TCP连接以减少从单个服务器取得多个文件所需时钟时间的Web客户端程序，但考虑到TCP的拥塞避免机制，它是对网络不利的

## 16.6 非阻塞accept

- 客户如图16-21所示建立一个连接并随后中止它。
- select向服务器进程返回可读条件，不过服务器要过一小段时间才调用accept。
- 在服务器从select返回到调用accept期间，服务器TCP收到来自客户的RST。
- 这个已完成的连接被服务器TCP驱除出队列，我们假设队列中没有其他已完成的连接。
- 服务器调用accept，但是由于没有任何已完成的连接，服务器于是阻塞。

服务器会一直阻塞在accept调用上，直到其他某个客户建立一个连接为止。但是在此期间，就以图6-22给出的服务器程序为例，服务器单纯阻塞在accept调用上，无法处理任何其他已就绪的描述符。

本问题和6.8节讲述的拒绝服务攻击多少有些类似，不过对于这个新的缺陷，一旦另有客户建立一个连接，服务器就会脱出阻塞中的accept。

本问题的解决办法如下。

(1) 当使用select获悉某个监听套接字上何时有已完成连接准备好被accept时，总是把这个监听套接字设置为非阻塞。

(2) 在后续的accept调用中忽略以下错误：EWOULDBLOCK（源自Berkeley的实现，客户中止连接时）、ECONNABORTED（POSIX实现，客户中止连接时）、EPROTO（SVR4实现，客户中止连接时）和EINTR（如果有信号被捕获）。

- 当使用select获悉某个监听套接字上何时有已完成连接准备好被accept时，总是把这个监听套接字设置为非阻塞
- 在后续的accept调用中忽略以下错误: EWOULDBLOCK 等

## 第17章 ioctl操作

---

- ioctl函数传统上一直作为那些不适合归入其他精细定义类别的特性的系统接口
- 网络程序通常在程序启动执行后使用ioctl获取所在主机全部网络接口的信息，包括: 接口地址、是否支持广播、是否支持多播，等

### 17.2 ioctl函数

- 影响由 fd参数引用的一个打开的文件
- request可分为6类
  - 套接字操作
  - 文件操作
  - 接口操作
  - ARP高速缓存操作
  - 路由表操作
  - 流系统
-

图17-1列出了网络相关ioctl请求的request参数以及arg地址必须指向的数据类型。以下各节详细讲解这些请求。

| 类别  | request        | 说 明              | 数据类型           |
|-----|----------------|------------------|----------------|
| 套接字 | SIOCATMARK     | 是否位于带外标记         | int            |
|     | SIOCSPGRP      | 设置套接字的进程ID或进程组ID | int            |
|     | SIOCGPGRP      | 获取套接字的进程ID或进程组ID | int            |
| 文件  | FIONBIO        | 设置/清除非阻塞式I/O标志   | int            |
|     | FIOASYNC       | 设置/清除信号驱动异步I/O标志 | int            |
|     | FIONREAD       | 获取接收缓冲区中的字节数     | int            |
|     | FIOSETOWN      | 设置文件的进程ID或进程组ID  | int            |
|     | FIOGETOWN      | 获取文件的进程ID或进程组ID  | int            |
| 接口  | SIOCGIFCONF    | 获取所有接口的列表        | struct ifconf  |
|     | SIOCSIFADDR    | 设置接口地址           | struct ifreq   |
|     | SIOCGIFADDR    | 获取接口地址           | struct ifreq   |
|     | SIOCSIFFLAGS   | 设置接口标志           | struct ifreq   |
|     | SIOCGIFFLAGS   | 获取接口标志           | struct ifreq   |
|     | SIOCSIFDSTADDR | 设置点到点地址          | struct ifreq   |
|     | SIOCGIFDSTADDR | 获取点到点地址          | struct ifreq   |
|     | SIOCGIFBRDADDR | 获取广播地址           | struct ifreq   |
|     | SIOCSIFBRDADDR | 设置广播地址           | struct ifreq   |
|     | SIOCGIFNETMASK | 获取子网掩码           | struct ifreq   |
|     | SIOCSIFNETMASK | 设置子网掩码           | struct ifreq   |
|     | SIOCGIFMETRIC  | 获取接口的测度          | struct ifreq   |
|     | SIOCSIFMETRIC  | 设置接口的测度          | struct ifreq   |
|     | SIOCGIFMTU     | 获取接口MTU          | struct ifreq   |
|     | SIOCxxx        | (还有很多：取决于实现)     | struct ifreq   |
| ARP | SIOCSARP       | 创建/修改ARP表项       | struct arpreq  |
|     | SIOCGARP       | 获取ARP表项          | struct arpreq  |
|     | SIOCDAWRP      | 删除ARP表项          | struct arpreq  |
| 路由  | SIOCADDRT      | 增加路径             | struct rtentry |
|     | SIOCDELRT      | 删除路径             | struct rtentry |
| 流   | I_xxx          | (参见31.5节)        |                |

图17-1 网络相关ioctl请求的总结

## 17.3 套接字操作

- SIOCATMARK 如果本套接字的读指针当前位于带外标记，则通过第三个参数返回一个非0值
- SIOCGPGRP 返回本套接字的进程ID或进程组ID
- SIOCSPGRP 把本套接字的进程ID或进程组ID设置为...

## 17.4 文件操作

- FIONBIO 可清除或设置本套接字的非阻塞式I/O标志
- FIOASYNC 可清除或设置针对本套接字的信号驱动异步I/O标志
- FIONREAD 返回本套接字接收缓冲区中的字节数
- FIOSETOWN
- FIOGETOWN

## 17.5 接口配置

- 从内核获取配置在系统中的所有接口，由SIOCGIFCONF请求完成，使用ifconf结构，ifconf又使用ifreq结构

```

struct ifconf {
 lint ifc_len; /* size of buffer, value-result */
 union {
 caddr_t ifcu_buf; /* input from user -> kernel */
 struct ifreq *ifcu_req; /* return from kernel -> user */
 } ifc_ifcu;
};

#define ifc_buf ifc_ifcu.ifcu_buf /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req /* array of structures returned */

#define IFNAMSIZ 16

struct ifreq {
 char ifr_name[IFNAMSIZ]; /* interface name, e.g., "le0" */
 union {
 struct sockaddr ifru_addr;
 struct sockaddr ifru_dstaddr;
 struct sockaddr ifru_broadaddr;
 short ifru_flags;
 int ifru_metric;
 caddr_t ifru_data;
 } ifr_ifru;
};

#define ifr_addr ifr_ifru.ifru_addr /* address */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr /* other end of p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags ifr_ifru.ifru_flags /* flags */
#define ifr_metric ifr_ifru.ifru_metric /* metric */
#define ifr_data ifr_ifru.ifru_data /* for use by interface */

```

---

<net/if.h>

## 17.6 get\_ifi\_info函数

- 返回一个结构链表，其中每个结构对应一个当前处于在工状态的接口

```

/* Our own header for the programs that need interface configuration info.
 Include this file, instead of "unp.h". */

#ifndef __unp_ifi_h
#define __unp_ifi_h

#include "unp.h"
#include <net/if.h>

#define IFI_NAME 16 /* same as IFNAMSIZ in <net/if.h> */
#define IFI_HADDR 8 /* allow for 64-bit EUI-64 in future */

struct ifi_info {
 char ifi_name[IFI_NAME]; /* interface name, null-terminated */
 short ifi_index; /* interface index */
 short ifi_mtu; /* interface MTU */
 u_char ifi_haddr[IFI_HADDR]; /* hardware address */
 u_short ifi_hlen; /* # bytes in hardware address: 0, 6, 8 */
/*
 short ifi_flags; /* IFF_xxx constants from <net/if.h> */
 short ifi_myflags; /* our own IFI_xxx flags */
 struct sockaddr *ifi_addr; /* primary address */
 struct sockaddr *ifi_brdaddr; /* broadcast address */
 struct sockaddr *ifi_dstaddr; /* destination address */

```

```

 struct ifi_info *ifi_next; /* next of these structures */
};

#define IFI_ALIAS 1 /* ifi_addr is an alias */

/* function prototypes */
struct ifi_info *get_ifi_info(int, int);
struct ifi_info *Get_ifi_info(int, int);
void free_ifi_info(struct ifi_info *);

#endif /* __unp_ifi_h */

```

- `#include "unpifi.h"`

```

int
main(int argc, char **argv)
{
 struct ifi_info *ifi, *ifihead;
 struct sockaddr *sa;
 u_char *ptr;
 int i, family, doaliases;

 if (argc != 3)
 err_quit("usage: prifinfo <inet4|inet6> <doaliases>");

 if (strcmp(argv[1], "inet4") == 0)
 family = AF_INET;
#ifndef IPv6
 else if (strcmp(argv[1], "inet6") == 0)
 family = AF_INET6;
#endif
 else
 err_quit("invalid <address-family>");
 doaliases = atoi(argv[2]);

 for (ifihead = ifi = Get_ifi_info(family, doaliases);
 ifi != NULL; ifi = ifi->ifi_next) {
 printf("%s: ", ifi->ifi_name);
 if (ifi->ifi_index != 0)
 printf("(%d) ", ifi->ifi_index);
 printf("<");

/* *INDENT-OFF*/
 if (ifi->ifi_flags & IFF_UP) printf("UP ");
 if (ifi->ifi_flags & IFF_BROADCAST) printf("BCAST ");
 if (ifi->ifi_flags & IFF_MULTICAST) printf("MCAST ");
 if (ifi->ifi_flags & IFF_LOOPBACK) printf("LOOP ");
 if (ifi->ifi_flags & IFF_POINTOPOINT) printf("P2P ");
 printf(">\n");
/* *INDENT-ON*/
 if ((i = ifi->ifi_hlen) > 0) {
 ptr = ifi->ifi_haddr;
 do {

```

```

 printf("%s%x", (i == ifi->ifi_hlen) ? " " : ":" ,
*ptr++);
 } while (--i > 0);
 printf("\n");
 }
 if (ifi->ifi_mtu != 0)
 printf(" MTU: %d\n", ifi->ifi_mtu);

 if ((sa = ifi->ifi_addr) != NULL)
 printf(" IP addr: %s\n",
 Sock_ntop_host(sa, sizeof(*sa)));
 if ((sa = ifi->ifi_brdaddr) != NULL)
 printf(" broadcast addr: %s\n",
 Sock_ntop_host(sa, sizeof(*sa)));
 if ((sa = ifi->ifi_dstaddr) != NULL)
 printf(" destination addr: %s\n",
 Sock_ntop_host(sa, sizeof(*sa)));
 }
 free_ifi_info(ifihead);
 exit(0);
}

```

- SIOCGIFCONF存在的问题: 在缓冲区的大小不足以存放结果时, 一些实现不返回错误, 而是截断结果并返回成功

```

• /* include get_ifi_info1 */
#include "unpifi.h"

struct ifi_info *
get_ifi_info(int family, int doaliases)
{
 struct ifi_info *ifi, *ifihead, **ifipnext;
 int sockfd, len, lastlen, flags, myflags, idx
= 0, hlen = 0;
 char *ptr, *buf, lastname[IFNAMSIZ], *cptr, *haddr,
*sdlname;
 struct ifconf ifc;
 struct ifreq *ifr, ifrcopy;
 struct sockaddr_in *sinptr;
 struct sockaddr_in6 *sin6ptr;

 sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

 lastlen = 0;
 len = 100 * sizeof(struct ifreq); /* initial buffer size guess */
 for (; ;) {
 buf = Malloc(len);
 ifc.ifc_len = len;
 ifc.ifc_buf = buf;
 if (ioctl(sockfd, SIOCGIFCONF, &ifc) < 0) {
 if (errno != EINVAL || lastlen != 0)
 err_sys("ioctl error");
 } else {

```

```

 if (ifc.ifc_len == lastlen)
 break; /* success, len has not changed */
 lastlen = ifc.ifc_len;
 }
 len += 10 * sizeof(struct ifreq); /* increment */
 free(buf);
}
ifihead = NULL;
ifipnext = &ifihead;
lastname[0] = 0;
sdlname = NULL;
/* end get_ifi_info1 */

/* include get_ifi_info2 */
for (ptr = buf; ptr < buf + ifc.ifc_len;) {
 ifr = (struct ifreq *) ptr;

#ifndef HAVE_SOCKADDR_SA_LEN
 len = max(sizeof(struct sockaddr), ifr->ifr_addr.sa_len);
#else
 switch (ifr->ifr_addr.sa_family) {
#endif
#ifdef IPV6
 case AF_INET6:
 len = sizeof(struct sockaddr_in6);
 break;
#endif
#ifndef HAVE_SOCKADDR_SA_LEN
 case AF_INET:
 default:
 len = sizeof(struct sockaddr);
 break;
 }
#endif /* HAVE_SOCKADDR_SA_LEN */
 ptr += sizeof(ifr->ifr_name) + len; /* for next one in buffer */

#ifndef HAVE_SOCKADDR_DL_STRUCT
 /* assumes that AF_LINK precedes AF_INET or AF_INET6 */
 if (ifr->ifr_addr.sa_family == AF_LINK) {
 struct sockaddr_dl *sdl = (struct sockaddr_dl *)&ifr->ifr_addr;
 sdlname = ifr->ifr_name;
 idx = sdl->sdl_index;
 haddr = sdl->sdl_data + sdl->sdl_nlen;
 hlen = sdl->sdl_alen;
 }
#endif

 if (ifr->ifr_addr.sa_family != family)
 continue; /* ignore if not desired address family */

 myflags = 0;
 if ((cptr = strchr(ifr->ifr_name, ':')) != NULL)
 cptr = 0; / replace colon with null */
 if (strncmp(lastname, ifr->ifr_name, IFNAMSIZ) == 0) {
 if (doaliases == 0)

```

```

 continue; /* already processed this interface */
myflags = IFI_ALIAS;
}
memcpy(lastname, ifr->ifr_name, IFNAMSIZ);

ifrcopy = *ifr;
Ioctl(sockfd, SIOCGIFFLAGS, &ifrcopy);
flags = ifrcopy.ifr_flags;
if ((flags & IFF_UP) == 0)
 continue; /* ignore if interface not up */
/* end get_ifi_info2 */

/* include get_ifi_info3 */
ifi = Calloc(1, sizeof(struct ifi_info));
ifipnext = ifi; / prev points to this new one */
ifipnext = &ifi->ifi_next; /* pointer to next one goes here */

ifi->ifi_flags = flags; /* IFF_xxx values */
ifi->ifi_myflags = myflags; /* IFI_xxx values */
#if defined(SIOCGIFMTU) && defined(HAVE_STRUCT_IFREQ_IFR_MTU)
 Ioctl(sockfd, SIOCGIFMTU, &ifrcopy);
 ifi->ifi_mtu = ifrcopy.ifr_mtu;
#else
 ifi->ifi_mtu = 0;
#endif
memcpy(ifi->ifi_name, ifr->ifr_name, IFI_NAME);
ifi->ifi_name[IFI_NAME-1] = '\0';
/* If the sockaddr_dl is from a different interface, ignore it */
if (sdlname == NULL || strcmp(sdlname, ifr->ifr_name) != 0)
 idx = hlen = 0;
ifi->ifi_index = idx;
ifi->ifi_hlen = hlen;
if (ifi->ifi_hlen > IFI_HADDR)
 ifi->ifi_hlen = IFI_HADDR;
if (hlen)
 memcpy(ifi->ifi_haddr, haddr, ifi->ifi_hlen);
/* end get_ifi_info3 */
/* include get_ifi_info4 */
switch (ifr->ifr_addr.sa_family) {
case AF_INET:
 sinptr = (struct sockaddr_in *) &ifr->ifr_addr;
 ifi->ifi_addr = Calloc(1, sizeof(struct sockaddr_in));
 memcpy(ifi->ifi_addr, sinptr, sizeof(struct sockaddr_in));

#endifdef SIOCGIFBRDADDR
 if (flags & IFF_BROADCAST) {
 Ioctl(sockfd, SIOCGIFBRDADDR, &ifrcopy);
 sinptr = (struct sockaddr_in *) &ifrcopy.ifr_broadaddr;
 ifi->ifi_brdaddr = Calloc(1, sizeof(struct sockaddr_in));
 memcpy(ifi->ifi_brdaddr, sinptr, sizeof(struct
sockaddr_in));
 }
#endif

```

```

#define SIOCGIFDSTADDR
 if (flags & IFF_POINTOPOINT) {
 Ioctl(sockfd, SIOCGIFDSTADDR, &ifrcopy);
 sinptr = (struct sockaddr_in *) &ifrcopy.ifr_dstaddr;
 ifi->ifi_dstaddr = Calloc(1, sizeof(struct sockaddr_in));
 memcpy(ifi->ifi_dstaddr, sinptr, sizeof(struct
sockaddr_in));
 }
#endif
 break;

 case AF_INET6:
 sin6ptr = (struct sockaddr_in6 *) &ifr->ifr_addr;
 ifi->ifi_addr = Calloc(1, sizeof(struct sockaddr_in6));
 memcpy(ifi->ifi_addr, sin6ptr, sizeof(struct sockaddr_in6));

#define SIOCGIFDSTADDR
 if (flags & IFF_POINTOPOINT) {
 Ioctl(sockfd, SIOCGIFDSTADDR, &ifrcopy);
 sin6ptr = (struct sockaddr_in6 *) &ifrcopy.ifr_dstaddr;
 ifi->ifi_dstaddr = Calloc(1, sizeof(struct
sockaddr_in6));
 memcpy(ifi->ifi_dstaddr, sin6ptr, sizeof(struct
sockaddr_in6));
 }
#endif
 break;

 default:
 break;
 }
}
free(buf);
return(ifihead); /* pointer to first structure in linked list */
}
/* end get_ifi_info4 */

/* include free_ifi_info */
void
free_ifi_info(struct ifi_info *ifihead)
{
 struct ifi_info *ifi, *ifinext;

 for (ifi = ifihead; ifi != NULL; ifi = ifinext) {
 if (ifi->ifi_addr != NULL)
 free(ifi->ifi_addr);
 if (ifi->ifi_brdaddr != NULL)
 free(ifi->ifi_brdaddr);
 if (ifi->ifi_dstaddr != NULL)
 free(ifi->ifi_dstaddr);
 ifinext = ifi->ifi_next; /* can't fetch ifi_next after free() */
 free(ifi); /* the ifi_info{} itself

```

```

*/
}

}

/* end free_ifi_info */

struct ifi_info *
Get_ifi_info(int family, int doaliases)
{
 struct ifi_info *ifi;

 if ((ifi = get_ifi_info(family, doaliases)) == NULL)
 err_quit("get_ifi_info error");
 return(ifi);
}

```

## 17.7 接口操作

- 接口的获取版本(SIOCGXXX)通常由netstat程序发出
- 接口的设置版本(SIOCSXXX)通常由ipconfig程序发出

|                  |                                                                                                                                                |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| SIOCGIFADDR      | 在ifr_addr成员中返回单播地址。                                                                                                                            |
| SIOCSIFADDR      | 用ifr_addr成员设置接口地址。这个接口的初始化函数也被调用。                                                                                                              |
| SIOCGIFFLAGS     | 在ifr_flags成员中返回接口标志。这些标志的名字格式为IFF_XXX，在<net/if.h>头文件中定义。举例来说，这些标志指示接口是否处于在工状态 (IFF_UP)，是否为一个点到点接口 (IFF_POINTOPOINT)，是否支持广播 (IFF_BROADCAST)，等等。 |
| SIOCSIFFLAGS     | 用ifr_flags成员设置接口标志。                                                                                                                            |
| SIOCGIFDSTADDR   | 在ifr_dstaddr成员中返回点到点地址。                                                                                                                        |
| SIOCSIFDSTADDR   | 用ifr_dstaddr成员设置点到点地址。                                                                                                                         |
| • SIOCGIFBRDADDR | 在ifr_broadaddr成员中返回广播地址。应用进程必须首先获取接口标志，然后发出正确的请求：对于广播接口为SIOCGIFBRDADDR，对于点到点接口为SIOCGIFDSTADDR。                                                 |
| SIOCSIFBRDADDR   | 用ifr_broadaddr成员设置广播地址。                                                                                                                        |
| SIOCGIFNETMASK   | 在ifr_addr成员中返回子网掩码。                                                                                                                            |
| SIOCSIFNETMASK   | 用ifr_addr成员设置子网掩码。                                                                                                                             |
| SIOCGIFMETRIC    | 用ifr_metric成员返回接口测度。接口测度由内核为每个接口维护，不过使用它的是路由守护进程routed。接口测度被routed加到跳数上（使得某个接口更不被看好）。                                                          |
| SIOCSIFMETRIC    | 用ifr_metric成员设置接口的路由测度。                                                                                                                        |

## 17.8 ARP高速缓存操作

- arpreq结构
-

```

struct arpreq {
 struct sockaddr arp_pa; /* protocol address */
 struct sockaddr arp_ha; /* hardware address */
 int arp_flags; /* flags */
};

#define ATF_INUSE 0x01 /* entry in use */
#define ATF_COM 0x02 /* completed entry (hardware addr valid) */
#define ATF_PERM 0x04 /* permanent entry */
#define ATF_PUBL 0x08 /* published entry (respond for other host) */

```

<net/if\_arp.h>

图17-12 ARP高速缓存类ioctl请求所用的arpreq结构

- SIOCSARP
- SIOCDARP
- SIOCGARP
- ioctl没有办法列出所有ARP缓存
- 大多数arp -a程序通过读取内核的内存 /dev/kmem获取ARP高速缓存

```

• #include "unpifi.h"
#include <net/if_arp.h>

int
main(int argc, char **argv)
{
 int sockfd;
 struct ifi_info *ifi;
 unsigned char *ptr;
 struct arpreq arpreq;
 struct sockaddr_in *sin;

 sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
 for (ifi = get_ifi_info(AF_INET, 0); ifi != NULL; ifi = ifi->ifi_next) {
 printf("%s: ", Sock_ntop(ifi->ifi_addr, sizeof(struct sockaddr_in)));

 sin = (struct sockaddr_in *) &arpreq.arp_pa;
 memcpy(sin, ifi->ifi_addr, sizeof(struct sockaddr_in));

 if (ioctl(sockfd, SIOCGARP, &arpreq) < 0) {
 err_ret("ioctl SIOCGARP");
 continue;
 }

 ptr = &arpreq.arp_ha.sa_data[0];
 printf("%x:%x:%x:%x:%x:%x\n", *ptr, *(ptr+1),
 *(ptr+2), *(ptr+3), *(ptr+4), *(ptr+5));
 }
 exit(0);
}

```

## 17.9 路由表操作

- rtentry结构

- 通常由route程序发出
- SIOCADDRT
- SIOCDELRT
- netstat -r 通过读取内核的内存/dev/kmem获得整个路由表

## 第18章 路由套接字

- 通过创建AF\_ROUTE域对访问内核中路由子系统的接口做了清理，在路由域中支持的唯一一种套接字是原始套接字
- 路由套接字支持3种操作
  - 进程可以通过写出到路由套接字而往内核发送消息
  - 进程可以通过从路由套接字接入而自内核接收消息
  - 进程可以使用sysctl函数倾斜出路由表或列出所有已配置的接口

### 18.2 数据链路套接字地址结构

```
struct sockaddr_dl {
 uint8_t sdl_len;
 sa_family_t sdl_family; /* AF_LINK */
 uint16_t sdl_index; /* system assigned index, if > 0 */
 uint8_t sdl_type; /* IFT_ETHER, ect. from <net/if_types.h> */
 uint8_t sdl_nlen; /* name length, starting in sdl_data[0] */
 •
 uint8_t sdl_alen; /* link-layer address length */
 uint8_t sdl_slen; /* link-layer selector length */
 char sdl_data[12]; /* minimum work area, can be larger;
 contains i/f name and link-layer address */
};
```

图18-1 数据链路套接字地址结构

### 18.3 读和写

- 创建一个路由套接字后，进程可以通过写到该套接字向内核发送命令，通过读自该套接字从内核接收信息

| 消息类型           | 去往内核? | 来自内核? | 说 明         | 结构类型              |
|----------------|-------|-------|-------------|-------------------|
| RTM_ADD        | •     | •     | 增加路径        | rt_msghdr         |
| RTM_CHANGE     | •     | •     | 改动网关、测度或标志  | rt_msghdr         |
| RTM_DELADDR    |       | •     | 地址正被剥离接口    | ifa_msghdr        |
| RTM_DELETE     | •     | •     | 删除路径        | rt_msghdr         |
| RTM_DELMADDR   |       | •     | 多播地址正被剥离接口  | ifma_msghdr       |
| RTM_GRT        | •     | •     | 报告测度及其他路径信息 | rt_msghdr         |
| RTM_IFANNOUNCE |       | •     | 接口正被增至或剥离系统 | if_announcemsghdr |
| RTM_IFINFO     |       | •     | 接口正在开工、停工等  | if_msghdr         |
| RTM_LOCK       | •     | •     | 锁住给定的测度     | rt_msghdr         |

图18-2 通过路由套接字交换的消息类型

- 通过路由套接字交换的结构有5个类型: rt\_msghdr if\_msghdr ifa\_msghdr ifma\_msghdr if\_announcemsghdr
-

```

struct rt_msghdr { /* from <net/route.h> */
 u_short rtm_msrlen; /* to skip over non-understood messages */
 u_char rtm_version; /* future binary compatibility */
 u_char rtm_type; /* message type */
 u_short rtm_index; /* index for associated ifp */
 int rtm_flags; /* flags, incl. kern & message, e.g., DONE */
 int rtm_addrs; /* bitmask identifying sockaddrs in msg */
 pid_t rtm_pid; /* identify sender */
 int rtm_seq; /* for sender to identify action */
 int rtm_errno; /* why failed */
 int rtm_use; /* from rtentry */
 u_long rtm_inits; /* which metrics we are initializing */
 struct rt_metrics rtm_rmx; /* metrics themselves */
};

struct if_msghdr { /* from <net/if.h> */
 u_short ifm_msrlen; /* to skip over non-understood messages */
 u_char ifm_version; /* future binary compatibility */
 u_char ifm_type; /* message type */
 int ifm_addrs; /* like rtm_addrs */
 int ifm_flags; /* value of if_flags */
 u_short ifm_index; /* index for associated ifp */
 struct if_data ifm_data; /* statistics and other data about if */
};

struct ifa_msghdr { /* from <net/if.h> */
 u_short ifam_msrlen; /* to skip over non-understood messages */
 u_char ifam_version; /* future binary compatibility */
 u_char ifam_type; /* message type */
 int ifam_addrs; /* like rtm_addrs */
 int ifam_flags; /* value of ifa_flags */
 u_short ifam_index; /* index for associated ifp */
 int ifam_metric; /* value of ifa_metric */
};

struct ifma_msghdr { /* from <net/if.h> */
 u_short ifmam_msrlen; /* to skip over non-understood messages */
 u_char ifmam_version; /* future binary compatibility */
 u_char ifmam_type; /* message type */
 int ifmam_addrs; /* like rtm_addrs */
 int ifmam_flags; /* value of ifa_flags */
 u_short ifmam_index; /* index for associated ifp */
};

struct if_announce_msghdr { /* from <net/if.h> */
 u_short ifan_msrlen; /* to skip over non-understood messages */
 u_char ifan_version; /* future binary compatibility */
 u_char ifan_type; /* message type */
 u_short ifan_index; /* index for associated ifp */
 char ifan_name[IFNAMSIZ]; /* if name, e.g. "en0" */
 u_short ifan_what; /* what type of announcement */
};

```

图18-3 路由消息返回的三种结构

| 数位掩码        |      | 数组下标         |    | 套接字地址结构包含  |
|-------------|------|--------------|----|------------|
| 常值          | 数值   | 常值           | 数值 |            |
| RTA_DST     | 0x01 | RTAX_DST     | 0  | 目的地址       |
| RTA_GATEWAY | 0x02 | RTAX_GATEWAY | 1  | 网关地址       |
| RTA_NETMASK | 0x04 | RTAX_NETMASK | 2  | 网络掩码       |
| RTA_GENMASK | 0x08 | RTAX_GENMASK | 3  | 克隆掩码       |
| RTA_IFP     | 0x10 | RTAX_IFP     | 4  | 接口名字       |
| RTA_IFA     | 0x20 | RTAX_IFA     | 5  | 接口地址       |
| RTA_AUTHOR  | 0x40 | RTAX_AUTHOR  | 6  | 重定向原创者     |
| RTA_BRD     | 0x80 | RTAX_BRD     | 7  | 广播或点到点目的地址 |
|             |      | RTAX_MAX     | 8  | 最大元素数目     |

图18-4 在路由消息中用于指称套接字地址结构的常值

- 默认路径在路由表中的目的IP地址为0.0.0.0 掩码为0.0.0.0

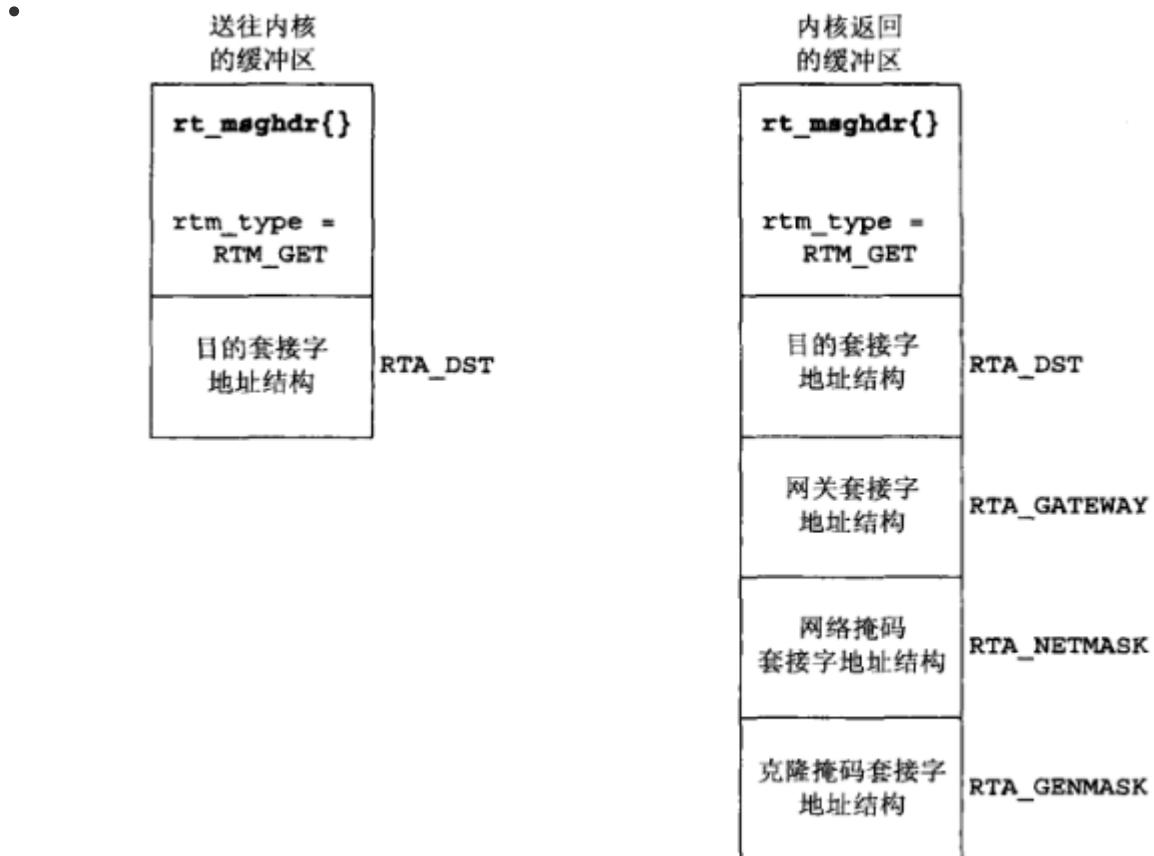


图18-5 RTM\_GET命令通过路由套接字与内核交换的数据

- ESRCH错误意味着内核找不到这个路径，EXIST错误意味着与这个路径一致的路由表项已经存在

```

/* include getrt1 */
#include "unproute.h"

#define BUflen (sizeof(struct rt_msghdr) + 512)
 /* sizeof(struct sockaddr_in6) * 8 = 192 */
#define SEQ 9999

int
main(int argc, char **argv)
{
 int sockfd;
 char *buf;
 pid_t pid;
 ssize_t n;
 struct rt_msghdr *rtm;
 struct sockaddr *sa, *rti_info[RTAX_MAX];
 struct sockaddr_in *sin;

 if (argc != 2)
 err_quit("usage: getrt <IPaddress>");

 sockfd = Socket(AF_ROUTE, SOCK_RAW, 0); /* need superuser privileges */

```

```

buf = Calloc(1, BUFSIZE); /* and initialized to 0 */

rtm = (struct rt_msghdr *) buf;
rtm->rtm_msflen = sizeof(struct rt_msghdr) + sizeof(struct sockaddr_in);
rtm->rtm_version = RTM_VERSION;
rtm->rtm_type = RTM_GET;
rtm->rtm_addrs = RTA_DST;
rtm->rtm_pid = pid = getpid();
rtm->rtm_seq = SEQ;

sin = (struct sockaddr_in *) (rtm + 1);
sin->sin_len = sizeof(struct sockaddr_in);
sin->sin_family = AF_INET;
Inet_pton(AF_INET, argv[1], &sin->sin_addr);

Write(sockfd, rtm, rtm->rtm_msflen);

do {
 n = Read(sockfd, rtm, BUFSIZE);
} while (rtm->rtm_type != RTM_GET || rtm->rtm_seq != SEQ ||
 rtm->rtm_pid != pid);
/* end getrt1 */

/* include getrt2 */
rtm = (struct rt_msghdr *) buf;
sa = (struct sockaddr *) (rtm + 1);
get_rtaddrs(rtm->rtm_addrs, sa, rti_info);
if ((sa = rti_info[RTAX_DST]) != NULL)
 printf("dest: %s\n", Sock_ntop_host(sa, sa->sa_len));

if ((sa = rti_info[RTAX_GATEWAY]) != NULL)
 printf("gateway: %s\n", Sock_ntop_host(sa, sa->sa_len));

if ((sa = rti_info[RTAX_NETMASK]) != NULL)
 printf("netmask: %s\n", Sock_masktop(sa, sa->sa_len));

if ((sa = rti_info[RTAX_GENMASK]) != NULL)
 printf("genmask: %s\n", Sock_masktop(sa, sa->sa_len));

exit(0);
}
/* end getrt2 */

```

- #include "unproute.h"

```

/*
 * Round up 'a' to next multiple of 'size', which must be a power of 2
 */
#define ROUNDUP(a, size) (((a) & ((size)-1)) ? (1 + ((a) | ((size)-1))) : (a))

/*
 * Step to next socket address structure;
 * if sa_len is 0, assume it is sizeof(u_long).

```

```

/*
#define NEXT_SA(ap) ap = (SA *) \
 ((caddr_t) ap + (ap->sa_len ? ROUNDUP(ap->sa_len, sizeof (u_long)) : \
 sizeof(u_long)))
 sizeof(u_long)))

void
get_rtaddrs(int addrs, SA *sa, SA **rti_info)
{
 int i;

 for (i = 0; i < RTAX_MAX; i++) {
 if (addrs & (1 << i)) {
 rti_info[i] = sa;
 NEXT_SA(sa);
 } else
 rti_info[i] = NULL;
 }
}

```

- #include "unproute.h"

```

const char *
sock_masktop(SA *sa, socklen_t salen)
{
 static char str[INET6_ADDRSTRLEN];
 unsigned char *ptr = &sa->sa_data[2];

 if (sa->sa_len == 0)
 return("0.0.0.0");
 else if (sa->sa_len == 5)
 sprintf(str, sizeof(str), "%d.0.0.0", *ptr);
 else if (sa->sa_len == 6)
 sprintf(str, sizeof(str), "%d.%d.0.0", *ptr, *(ptr+1));
 else if (sa->sa_len == 7)
 sprintf(str, sizeof(str), "%d.%d.%d.0", *ptr, *(ptr+1), *(ptr+2));
 else if (sa->sa_len == 8)
 sprintf(str, sizeof(str), "%d.%d.%d.%d",
 *ptr, *(ptr+1), *(ptr+2), *(ptr+3));
 else
 sprintf(str, sizeof(str), "(unknown mask, len = %d, family = %d)",
 sa->sa_len, sa->sa_family);
 return(str);
}

```

## 18.4 sysctl操作

- 使用sysctl检查路由表和接口列表的进程不限用户权限
-

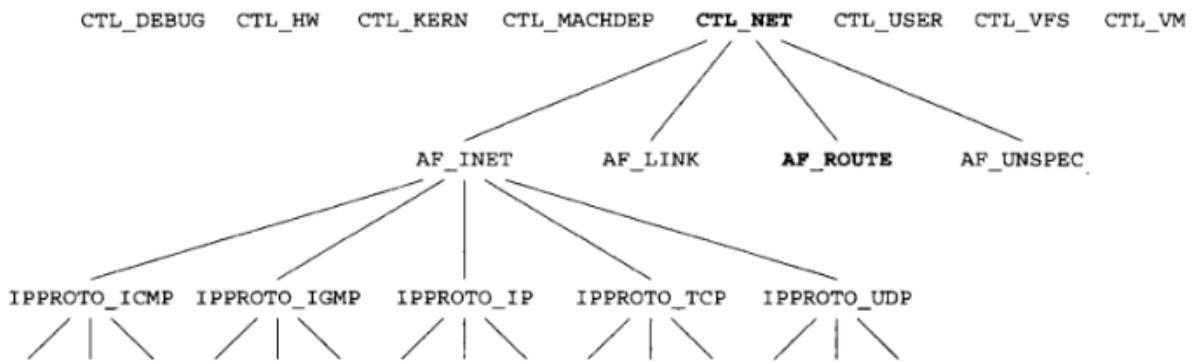


图18-11 sysctl名字的分层排列

- sysctl的手册页面详细叙述了可使用该函数获取的各种系统信息，有文件系统、虚拟内存、内核限制、硬件等各方面信息
- 当name数组的第二个元素为AF\_ROUTE时，第三个元素（协议号）总是为0（因为AF\_ROUTE族不像譬如说AF\_INET族那样其中有协议），第四个元素是一个地址族，第五和第六级指定做什么。图18-12对此做了汇总。

| name[] | 返回IPv4路由表   | 返回IPv4 ARP高速缓存 | 返回IPv6路由表   | 返回接口清单        |
|--------|-------------|----------------|-------------|---------------|
| 0      | CTL_NET     | CTL_NET        | CTL_NET     | CTL_NET       |
| 1      | AF_ROUTE    | AF_ROUTE       | AF_ROUTE    | AF_ROUTE      |
| 2      | 0           | 0              | 0           | 0             |
| 3      | AF_INET     | AF_INET        | AF_INET6    | 0             |
| 4      | NET_RT_DUMP | NET_RT_FLAGS   | NET_RT_DUMP | NET_RT_IFLIST |
| 5      | 0           | RTF_LLINFO     | 0           | 0             |

图18-12 sysctl在AF\_ROUTE域返回的信息

- 路由域支持3种操作，由name[4]指定，这三种操作返回的信息通过sysctl调用中的oldp指针返回
  - NET\_RTDUMP 返回由name[3]指定的地址族的路由表
  - NET\_RT\_FLAGS返回由name[3]指定的地址族的路由表，但是仅限于那些所带标志与由name[5]指定的标志相匹配的路由表项
  - NET\_RT\_IFLIST返回所有已配置接口的信息....

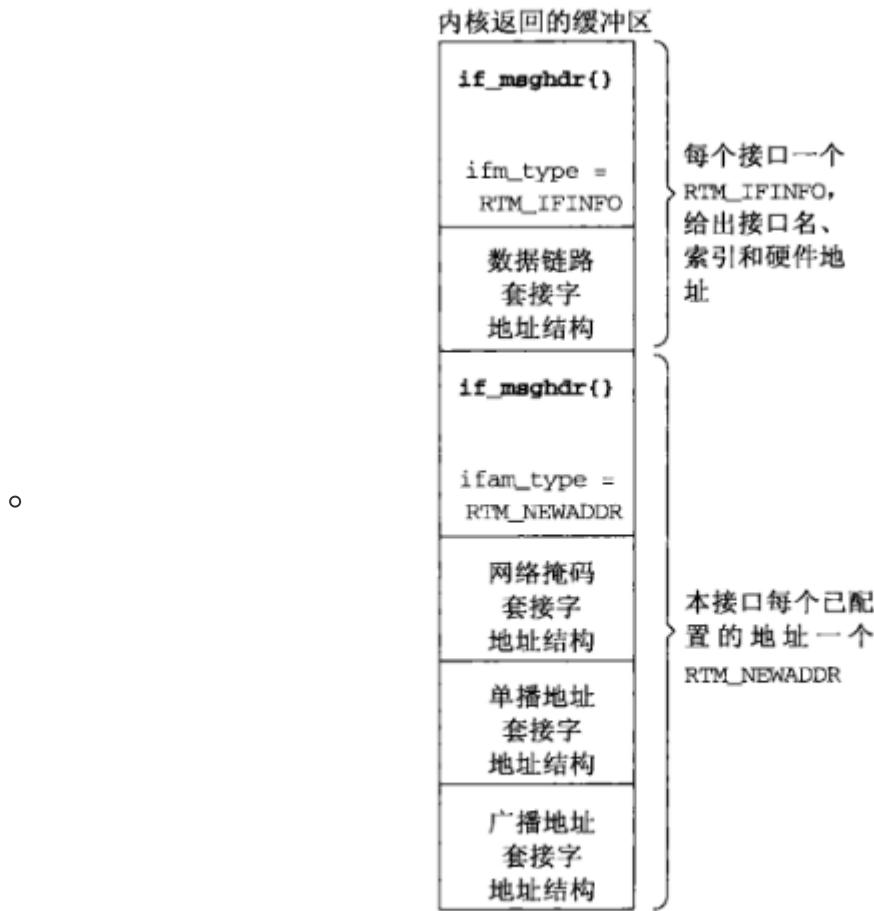


图18-13 由sysctl的CTL\_NET/AF\_ROUTE/NET\_RT\_IFLIST命令返回的信息

- 检查UDP校验和是否开启

```

o #include "unproute.h"
include <netinet/udp.h>
include <netinet/ip_var.h>
include <netinet/udp_var.h> /* for UDPCTL_XXX constants */

int
main(int argc, char **argv)
{
 int mib[4], val;
 size_t len;

 mib[0] = CTL_NET;
 mib[1] = AF_INET;
 mib[2] = IPPROTO_UDP;
 mib[3] = UDPCTL_CHECKSUM;

 len = sizeof(val);
 Sysctl(mib, 4, &val, &len, NULL, 0);
 printf("udp checksum flag: %d\n", val);

 exit(0);
}

```

## 18.5 get\_ifi\_info函数

- 使用sysctl实现的版本，取代ioctl SIOCGIFCONF实现的版本
- 调用sysctl返回接口列表

```
o /* include net_rt_iflist */
#include "unproute.h"

char *
net_rt_iflist(int family, int flags, size_t *lenp)
{
 int mib[6];
 char *buf;

 mib[0] = CTL_NET;
 mib[1] = AF_ROUTE;
 mib[2] = 0;
 mib[3] = family; /* only addresses of this family */
 mib[4] = NET_RT_IFLIST;
 mib[5] = flags; /* interface index or 0 */
 if (sysctl(mib, 6, NULL, lenp, NULL, 0) < 0)
 return(NULL);

 if ((buf = malloc(*lenp)) == NULL)
 return(NULL);
 if (sysctl(mib, 6, buf, lenp, NULL, 0) < 0) {
 free(buf);
 return(NULL);
 }

 return(buf);
}
/* end net_rt_iflist */

char *
Net_rt_iflist(int family, int flags, size_t *lenp)
{
 char *ptr;

 if ((ptr = net_rt_iflist(family, flags, lenp)) == NULL)
 err_sys("net_rt_iflist error");
 return(ptr);
}
```

- #include "unpifi.h"
 #include "unproute.h"

 /\* include get\_ifi\_info1 \*/
 struct ifi\_info \*
 get\_ifi\_info(int family, int doaliases)
 {
 int flags;

```

char *buf, *next, *lim;
size_t len;
struct if_msghdr *ifm;
struct ifa_msghdr *ifam;
struct sockaddr *sa, *rti_info[RTAX_MAX];
struct sockaddr_dl *sdl;
struct ifi_info *ifi, *ifisave, *ifihead, **ifipnext;

buf = Net_rt_iflist(family, 0, &len);

ifihead = NULL;
ifipnext = &ifihead;

lim = buf + len;
for (next = buf; next < lim; next += ifm->ifm_msrlen) {
 ifm = (struct if_msghdr *) next;
 if (ifm->ifm_type == RTM_IFINFO) {
 if (((flags = ifm->ifm_flags) & IFF_UP) == 0)
 continue; /* ignore if interface not up */

 sa = (struct sockaddr *) (ifm + 1);
 get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
 if ((sa = rti_info[RTAX_IFP]) != NULL) {
 ifi = Calloc(1, sizeof(struct ifi_info));
 ifipnext = ifi; / prev points to
this new one */
 ifipnext = &ifi->ifi_next; /* ptr to next one goes
here */

 ifi->ifi_flags = flags;
 if (sa->sa_family == AF_LINK) {
 sdl = (struct sockaddr_dl *) sa;
 ifi->ifi_index = sdl->sdl_index;
 if (sdl->sdl_nlen > 0)
 sprintf(ifi->ifi_name, IFI_NAME, "%*s",
sdl->sdl_nlen, &sdl-
>sdl_data[0]);
 else
 sprintf(ifi->ifi_name, IFI_NAME, "index
%d",
sdl->sdl_index);

 if ((ifi->ifi_hlen = sdl->sdl_alen) > 0)
 memcpy(ifi->ifi_haddr, LLADDR(sdl),
min(IFI_HADDR, sdl-
>sdl_alen));
 }
 }
/* end get_ifi_info1 */

/* include get_ifi_info3 */
 } else if (ifm->ifm_type == RTM_NEWADDR) {
 if (ifi->ifi_addr) { /* already have an IP addr for i/f */

```

```

 if (doaliases == 0)
 continue;

 /* we have a new IP addr for existing interface
 */
 ifisave = ifi;
 ifi = Calloc(1, sizeof(struct ifi_info));
 ifipnext = ifi; / prev points to
this new one */
 ifipnext = &ifi->ifi_next; /* ptr to next one goes
here */
 ifi->ifi_flags = ifisave->ifi_flags;
 ifi->ifi_index = ifisave->ifi_index;
 ifi->ifi_hlen = ifisave->ifi_hlen;
 memcpy(ifi->ifi_name, ifisave->ifi_name, IFI_NAME);
 memcpy(ifi->ifi_haddr, ifisave->ifi_haddr, IFI_HADDR);
 }

 ifam = (struct ifa_msghdr *) next;
 sa = (struct sockaddr *) (ifam + 1);
 get_rtaddrs(ifam->ifam_addrs, sa, rti_info);

 if ((sa = rti_info[RTAX_IFA]) != NULL) {
 ifi->ifi_addr = Calloc(1, sa->sa_len);
 memcpy(ifi->ifi_addr, sa, sa->sa_len);
 }

 if ((flags & IFF_BROADCAST) &&
 (sa = rti_info[RTAX_BRD]) != NULL) {
 ifi->ifi_brdaddr = Calloc(1, sa->sa_len);
 memcpy(ifi->ifi_brdaddr, sa, sa->sa_len);
 }

 if ((flags & IFF_POINTOPOINT) &&
 (sa = rti_info[RTAX_BRD]) != NULL) {
 ifi->ifi_dstaddr = Calloc(1, sa->sa_len);
 memcpy(ifi->ifi_dstaddr, sa, sa->sa_len);
 }

} else
 err_quit("unexpected message type %d", ifm->ifm_type);
}

/* "ifihead" points to the first structure in the linked list */
return(ifihead); /* ptr to first structure in linked list */
}
/* end get_ifi_info3 */

void
free_ifi_info(struct ifi_info *ifihead)
{
 struct ifi_info *ifi, *ifinext;

 for (ifi = ifihead; ifi != NULL; ifi = ifinext) {

```

```

 if (ifi->ifi_addr != NULL)
 free(ifi->ifi_addr);
 if (ifi->ifi_brdaddr != NULL)
 free(ifi->ifi_brdaddr);
 if (ifi->ifi_dstaddr != NULL)
 free(ifi->ifi_dstaddr);
 ifinext = ifi->ifi_next; /* can't fetch ifi_next after
free() */
 free(ifi); /* the ifi_info{} itself
*/
}
}

struct ifi_info *
Get_ifi_info(int family, int doaliases)
{
 struct ifi_info *ifi;

 if ((ifi = get_ifi_info(family, doaliases)) == NULL)
 err_quit("get_ifi_info error");
 return(ifi);
}

```

## 18.6 接口名字和索引函数

- 每个接口都有一个唯一的名字和唯一的正值索引
- if\_namrtoindex 返回名字为ifname的接口的索引
- if\_indextoname返回索引为ifindex的接口的名字
- if\_nameindex返回一个指向if\_nameindex结构数组的指针，if\_freetenameindex函数归还给系统
- if\_nametoindex函数

```

o /* include if_nametoindex */
#include "unpifi.h"
#include "unproute.h"

unsigned int
if_nametoindex(const char *name)
{
 unsigned int idx, namelen;
 char *buf, *next, *lim;
 size_t len;
 struct if_msghdr *ifm;
 struct sockaddr *sa, *rti_info[RTAX_MAX];
 struct sockaddr_dl *sdl;

 if ((buf = net_rt_iflist(0, 0, &len)) == NULL)
 return(0);

 namelen = strlen(name);
 lim = buf + len;
 for (next = buf; next < lim; next += ifm->ifm_msglen) {

```

```

 ifm = (struct if_msghdr *) next;
 if (ifm->ifm_type == RTM_IFINFO) {
 sa = (struct sockaddr *) (ifm + 1);
 get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
 if ((sa = rti_info[RTAX_IFP]) != NULL) {
 if (sa->sa_family == AF_LINK) {
 sdl = (struct sockaddr_dl *) sa;
 if (sdl->sdl_nlen == namelen &&
strncmp(&sdl->sdl_data[0], name, sdl->sdl_nlen) == 0) {
 idx = sdl->sdl_index; /* save
before free() */
 free(buf);
 return(idx);
 }
 }
 }
 }
 free(buf);
 return(0); /* no match for name */
 }
/* end if_nametoindex */

unsigned int
If_nametoindex(const char *name)
{
 int idx;

 if ((idx = if_nametoindex(name)) == 0)
 err_quit("if_nametoindex error for %s", name);
 return(idx);
}

```

- if\_indextoname函数

```

○ /* include if_indextoname */
#include "unpifi.h"
#include "unproute.h"

char *
if_indextoname(unsigned int idx, char *name)
{
 char *buf, *next, *lim;
 size_t len;
 struct if_msghdr *ifm;
 struct sockaddr *sa, *rti_info[RTAX_MAX];
 struct sockaddr_dl *sdl;

 if ((buf = net_rt_iflist(0, idx, &len)) == NULL)
 return(NULL);

 lim = buf + len;

```

```

 for (next = buf; next < lim; next += ifm->ifm_msrlen) {
 ifm = (struct if_msghdr *) next;
 if (ifm->ifm_type == RTM_IFINFO) {
 sa = (struct sockaddr *) (ifm + 1);
 get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
 if ((sa = rti_info[RTAX_IFP]) != NULL) {
 if (sa->sa_family == AF_LINK) {
 sdl = (struct sockaddr_dl *) sa;
 if (sdl->sdl_index == idx) {
 int slen = min(IFNAMSIZ - 1, sdl-
>sdl_nlen);
 strncpy(name, sdl->sdl_data, slen);
 name[slen] = 0; /* null terminate */
 free(buf);
 return(name);
 }
 }
 }
 }
 free(buf);
 return(NULL); /* no match for index */
 }
 /* end if_indextoname */

 char *
If_indextoname(unsigned int idx, char *name)
{
 char *ptr;

 if ((ptr = if_indextoname(idx, name)) == NULL)
 err_quit("if_indextoname error for %d", idx);
 return(ptr);
}

```

- if\_nameindex函数

```

 /* include if_nameindex */
#include "unpifi.h"
#include "unproute.h"

struct if_nameindex *
if_nameindex(void)
{
 char *buf, *next, *lim;
 size_t len;
 struct if_msghdr *ifm;
 struct sockaddr *sa, *rti_info[RTAX_MAX];
 struct sockaddr_dl *sdl;
 struct if_nameindex *result, *ifptr;
 char *namptr;

```

```

 if ((buf = net_rt_iflist(0, 0, &len)) == NULL)
 return(NULL);

 if ((result = malloc(len)) == NULL) /* overestimate */
 return(NULL);
 ifptr = result;
 namptr = (char *) result + len; /* names start at end of buffer */

 lim = buf + len;
 for (next = buf; next < lim; next += ifm->ifm_msrlen) {
 ifm = (struct if_msghdr *) next;
 if (ifm->ifm_type == RTM_IFINFO) {
 sa = (struct sockaddr *) (ifm + 1);
 get_rtaddrs(ifm->ifm_addrs, sa, rti_info);
 if ((sa = rti_info[RTAX_IFP]) != NULL) {
 if (sa->sa_family == AF_LINK) {
 sdl = (struct sockaddr_dl *) sa;
 namptr -= sdl->sdl_nlen + 1;
 strncpy(namptr, &sdl->sdl_data[0], sdl-
>sdl_nlen);
 namptr[sdl->sdl_nlen] = 0; /* null
terminate */
 ifptr->if_name = namptr;
 ifptr->if_index = sdl->sdl_index;
 ifptr++;
 }
 }
 }
 }
 ifptr->if_name = NULL; /* mark end of array of structs */
 ifptr->if_index = 0;
 free(buf);
 return(result); /* caller must free() this when done */
 }
/* end if_nameindex */

/* include if_freenameindex */
void
if_freenameindex(struct if_nameindex *ptr)
{
 free(ptr);
}
/* end if_freenameindex */

struct if_nameindex *
If_nameindex(void)
{
 struct if_nameindex *ifptr;

 if ((ifptr = if_nameindex()) == NULL)
 err_quit("if_nameindex error");
}

```

```
 return(ifptr);
 }
```

## 第19章 密钥管理套接字

- 通用密钥管理API, 可用于IPsec和其他网络安全服务, 该API也创建了一个新的协议族即PF\_KEY域, 唯一支持的一种套接字是原始套接字, 仅限超级用户有打开的权利
- IPsec基于安全关联SA为分组提供安全服务
- 存放在一个系统中的所有SA构成的集合称为安全关联数据库SADB
- 安全策略数据库SPDB描述分组流通的需求
- 密钥管理套接字支持3种类型的操作
  - 通过写出到密钥管理套接字, 进程可以往内核以及打开着密钥管理套接字的所有其他进程发送消息
  - 通过请求从密钥管理套接字读入, 进程可以自内核(或其他进程)接收消息
  - 进程可以往内核发送一个倾泄请求消息, 得到SADB

### 19.2 读和写

- 密钥管理消息首部, sadb\_msg

```
struct sadb_msg{
 u_int8_t sadb_msg_version;
 u_int8_t sadb_msg_type;
 u_int8_t sadb_msg_errno;
 u_int8_t sadb_msg_satype;
 u_int16_t sadb_msg_len;
 u_int16_t sadb_msg_reserved;
 u_int32_t sadb_msg_seq;
 u_int32_t sadb_msg_pid;
}
```

- 每个sadb\_msg首部将后跟零个或多个扩展
-

| 消息类型          | 去往内核? | 来自内核? | 说    明              |
|---------------|-------|-------|---------------------|
| SADB_ACQUIRE  | •     | •     | 请求创建一个SADB表项        |
| SADB_ADD      | •     | •     | 增加一个完整的SADB表项       |
| SADB_DELETE   | •     | •     | 删除一个SADB表项          |
| SADB_DUMP     | •     | •     | 倾泻出SADB (调试用)       |
| SADB_EXPIRE   |       | •     | 通知某个SADB表项已经期满      |
| SADB_FLUSH    | •     | •     | 冲刷整个SADB            |
| SADB_GET      | •     | •     | 获取一个SADB表项          |
| SADB_GETSPI   | •     | •     | 分配一个用于创建SADB表项的SPI  |
| SADB_REGISTER | •     |       | 注册成SADB_ACQUIRE的应答者 |
| SADB_UPDATE   | •     | •     | 更改一个不完备的SADB表项      |

图19-2 通过密钥管理套接字交换的消息类型

| 扩展首部类型                    | 说    明   | 结    构        |
|---------------------------|----------|---------------|
| SADB_EXT_ADDRESS_DST      | SA目的地址   | sadb_address  |
| SADB_EXT_ADDRESS_PROXY    | SA代理地址   | sadb_address  |
| SADB_EXT_ADDRESS_SRC      | SA源地址    | sadb_address  |
| SADB_EXT_IDENTITY_DST     | 目的身份     | sadb_ident    |
| SADB_EXT_IDENTITY_SRC     | 源身份      | sadb_ident    |
| SADB_EXT_KEY_AUTH         | 认证密钥     | sadb_key      |
| SADB_EXT_KEY_ENCRYPT      | 加密密钥     | sadb_key      |
| SADB_EXT_LIFETIME_CURRENT | SA当前生命期  | sadb_lifetime |
| SADB_EXT_LIFETIME_HARD    | SA生命期硬限制 | sadb_lifetime |
| SADB_EXT_LIFETIME_SOFT    | SA生命期软限制 | sadb_lifetime |

图19-3 PF\_KEY扩展类型

### 19.3 倾泄安全关联数据库

- SADB\_DUMP消息倾泻出当前的SADB,不需要任何扩展,单纯是16字节的sadb\_msg首部
- SA类型

```
#include "unp.h"
#include <net/pfkeyv2.h>

/* include sadb_dump */
void
sadb_dump(int type)
{
 int s;
 char buf[4096];
 struct sadb_msg msg;
 int goteof;

 s = Socket(PF_KEY, SOCK_RAW, PF_KEY_V2);

 /* Build and write SADB_DUMP request */
 bzero(&msg, sizeof(msg));
 msg.sadb_msg_version = PF_KEY_V2;
 msg.sadb_msg_type = SADB_DUMP;
 msg.sadb_msg_satype = type;
 msg.sadb_msg_len = sizeof(msg) / 8;
 msg.sadb_msg_pid = getpid();
 printf("Sending dump message:\n");
 print_sadb_msg(&msg, sizeof(msg));
 Write(s, &msg, sizeof(msg));

 printf("\nMessages returned:\n");
}
```

```

/* Read and print SADB_DUMP replies until done */
goteof = 0;
while (goteof == 0) {
 int msglen;
 struct sadb_msg *msgp;

 msglen = Read(s, &buf, sizeof(buf));
 msgp = (struct sadb_msg *)&buf;
 print_sadb_msg(msgp, msglen);
 if (msgp->sadb_msg_seq == 0)
 goteof = 1;
}
close(s);
}

int
main(int argc, char **argv)
{
 int satype = SADB_SATYPE_UNSPEC;
 int c;

 opterr = 0; /* don't want getopt() writing to stderr */
 while ((c = getopt(argc, argv, "t:")) != -1) {
 switch (c) {
 case 't':
 if ((satype = getsatypename(optarg)) == -1)
 err_quit("invalid -t option %s", optarg);
 break;

 default:
 err_quit("unrecognized option: %c", c);
 }
 }

 sadb_dump(satype);
}
/* end sadb_dump */

```

## 19.4 创建静态安全关联

- SADB\_ADD消息

---

```

struct sadb_sa {
 u_int16_t sadb_sa_len; /* length of extension / 8 */
 u_int16_t sadb_sa_exttype; /* SADB_EXT_SA */
 u_int32_t sadb_sa_spi; /* Security Parameters Index (SPI) */
 u_int8_t sadb_sa_replay; /* replay window size, or zero */
 u_int8_t sadb_sa_state; /* SA state, see Figure 19.7 */
 u_int8_t sadb_sa_auth; /* authentication algorithm, see Figure 19.8 */
 u_int8_t sadb_sa_encrypt; /* encryption algorithm, see Figure 19.8 */
 u_int32_t sadb_sa_flags; /* bitmask of flags */
};

```

---

图19-6 SA扩展

- 安全参数索引|SPI结合目的地址和所用协议唯一标识一个SA
-

| SA状态                | 说 明     | 可用否? |
|---------------------|---------|------|
| SADB_SASTATE_LARVAL | 被创建过程中  | 否    |
| SADB_SASTATE_MATURE | 完全形成    | 是    |
| SADB_SASTATE_DYING  | 软生命周期结束 | 是    |
| SADB_SASTATE_DEAD   | 硬生命周期结束 | 否    |

图19-7 SA的可能状态

| 算 法                | 说 明           | 参 考      |
|--------------------|---------------|----------|
| SADB_AALG_NONE     | 不认证           |          |
| SADB_AALG_MD5HMAC  | HMAC-MD5-96   | RFC 2403 |
| SADB_AALG_SHA1HMAC | HMAC-SHA-1-96 | RFC 2404 |
| SADB_EALG_NONE     | 不加密           |          |
| SADB_EALG_DESCBC   | DES-CBC       | RFC 2405 |
| SADB_EALG_3DESCBC  | 3DES-CBC      | RFC 1851 |
| SADB_EALG_NULL     | NULL          | RFC 2410 |

图19-8 认证和加密算法

```

struct sadb_address {
 u_int16_t sadb_address_len; /* length of extension + address / 8 */
 u_int16_t sadb_address_exttype; /* SADB_EXT_ADDRESS_{SRC,DST,PROXY} */
 u_int8_t sadb_address_proto; /* IP protocol, or 0 for all */
 u_int8_t sadb_address_prefixlen; /* # significant bits in address */
 u_int16_t sadb_address_reserved; /* reserved for extension */

};

/* followed by appropriate sockaddr */

```

图19-9 地址扩展

- sadb\_address结构后跟合适地址族的sockaddr结构
- 密钥本身则紧跟在sadb\_key结构之后

```

struct sadb_key {
 u_int16_t sadb_key_len; /* length of extension + key / 8 */
 u_int16_t sadb_key_exttype; /* SADB_EXT_KEY_{AUTH,ENCRYPT} */
 u_int16_t sadb_key_bits; /* # bits in key */
 u_int16_t sadb_key_reserved; /* reserved for extension */

};

/* followed by key data */

```

图19-10 密钥扩展

- 应答消息被发送到所有PF\_KEY套接字，然而不同的套接字可能属于不同的保护域，密钥数据不应该跨越保护域

```

struct sadb_lifetime {
 u_int16_t sadb_lifetime_len; /* length of extension / 8 */
 u_int16_t sadb_lifetime_exttype; /* SADB_EXT_LIFETIME_{SOFT,HARD,CURRENT} */
 u_int32_t sadb_lifetime_allocations; /* # connections, endpoints, or flows */
 u_int64_t sadb_lifetime_bytes; /* # bytes */
 u_int64_t sadb_lifetime_addtime; /* time of creation, or time from
 creation to expiration */
 u_int64_t sadb_lifetime_usetime; /* time first used, or time from
 first use to expiration */

};

```

图19-12 生命期扩展

生命周期扩展共有3个类型。SADB\_LIFETIME\_SOFT和SADB\_LIFETIME\_HARD这两个扩展分别指定一个SA的软生命期和硬生命期。当软生命期结束时，内核发送一个SADB\_EXPIRE消息；当硬生命期结束后，该SA不能再用。用于指出相应SA当前生命期的SADB\_LIFETIME\_CURRENT扩展在以下响应消息中返回：SADB\_DUMP、SADB\_EXPIRE和SADB\_GET。

```

#include "unp.h"
#include <net/pfkeyv2.h>

```

```

int
salen(struct sockaddr *sa)
{
#endif HAVE_SOCKADDR_SA_LEN
 return sa->sa_len;
#else
 switch (sa->sa_family) {
 case AF_INET:
 return sizeof(struct sockaddr_in);
#endif IPV6
 case AF_INET6:
 return sizeof(struct sockaddr_in6);
#endif
 default:
 return 0; /* XXX */
 }
#endif
}

int
prefix_all(struct sockaddr *sa)
{
 switch (sa->sa_family) {
 case AF_INET:
 return 32;
#endif IPV6
 case AF_INET6:
 return 128;
#endif
 default:
 return 0; /* XXX */
 }
}

/* include sadb_add */
void
sadb_add(struct sockaddr *src, struct sockaddr *dst, int type, int alg,
 int spi, int keybits, unsigned char *keydata)
{
 int s;
 char buf[4096], *p; /* XXX */
 struct sadb_msg *msg;
 struct sadb_sa *saext;
 struct sadb_address *addrext;
 struct sadb_key *keyext;
 int len;
 int mypid;

 s = Socket(PF_KEY, SOCK_RAW, PF_KEY_V2);

 mypid = getpid();

```

```

/* Build and write SADB_ADD request */
bzero(&buf, sizeof(buf));
p = buf;
msg = (struct sadb_msg *)p;
msg->sadb_msg_version = PF_KEY_V2;
msg->sadb_msg_type = SADB_ADD;
msg->sadb_msg_satype = type;
msg->sadb_msg_pid = getpid();
len = sizeof(*msg);
p += sizeof(*msg);

saext = (struct sadb_sa *)p;
saext->sadb_sa_len = sizeof(*saext) / 8;
saext->sadb_sa_exttype = SADB_EXT_SA;
saext->sadb_sa_spi = htonl.spi;
saext->sadb_sa_replay = 0; /* no replay protection with static keys */
saext->sadb_sa_state = SADB_SASTATE_MATURE;
saext->sadb_sa_auth = alg;
saext->sadb_sa_encrypt = SADB_EALG_NONE;
saext->sadb_sa_flags = 0;
len += saext->sadb_sa_len * 8;
p += saext->sadb_sa_len * 8;

addrext = (struct sadb_address *)p;
addrext->sadb_address_len = (sizeof(*addrext) + salen(src) + 7) / 8;
addrext->sadb_address_exttype = SADB_EXT_ADDRESS_SRC;
addrext->sadb_address_proto = 0; /* any protocol */
addrext->sadb_address_prefixlen = prefix_all(src);
addrext->sadb_address_reserved = 0;
memcpy(addrext + 1, src, salen(src));
len += addrext->sadb_address_len * 8;
p += addrext->sadb_address_len * 8;

addrext = (struct sadb_address *)p;
addrext->sadb_address_len = (sizeof(*addrext) + salen(dst) + 7) / 8;
addrext->sadb_address_exttype = SADB_EXT_ADDRESS_DST;
addrext->sadb_address_proto = 0; /* any protocol */
addrext->sadb_address_prefixlen = prefix_all(dst);
addrext->sadb_address_reserved = 0;
memcpy(addrext + 1, dst, salen(dst));
len += addrext->sadb_address_len * 8;
p += addrext->sadb_address_len * 8;

keyext = (struct sadb_key *)p;
/* "+7" handles alignment requirements */
keyext->sadb_key_len = (sizeof(*keyext) + (keybits / 8) + 7) / 8;
keyext->sadb_key_exttype = SADB_EXT_KEY_AUTH;
keyext->sadb_key_bits = keybits;
keyext->sadb_key_reserved = 0;
memcpy(keyext + 1, keydata, keybits / 8);
len += keyext->sadb_key_len * 8;
p += keyext->sadb_key_len * 8;

```

```

msg->sadb_msg_len = len / 8;
printf("Sending add message:\n");
print_sadb_msg(buf, len);
write(s, buf, len);

printf("\nReply returned:\n");
/* Read and print SADB_ADD reply, discarding any others */
for (;;) {
 int msglen;
 struct sadb_msg *msgp;

 msglen = Read(s, &buf, sizeof(buf));
 msgp = (struct sadb_msg *)&buf;
 if (msgp->sadb_msg_pid == mypid &&
 msgp->sadb_msg_type == SADB_ADD) {
 print_sadb_msg(msgp, msglen);
 break;
 }
}
close(s);
}

/* end sadb_add */

int
main(int argc, char **argv)
{
 struct addrinfo hints, *src, *dst;
 unsigned char *p, *keydata, *kp;
 char *ep;
 int ret, len, i;
 int satype, alg, keybits;

 bzero(&hints, sizeof(hints));
 if ((ret = getaddrinfo(argv[1], NULL, &hints, &src)) != 0) {
 err_quit("%s: %s\n", argv[1], gai_strerror(ret));
 }
 if ((ret = getaddrinfo(argv[2], NULL, &hints, &dst)) != 0) {
 err_quit("%s: %s\n", argv[2], gai_strerror(ret));
 }
 if (src->ai_family != dst->ai_family) {
 err_quit("%s and %s not same addr family\n", argv[1], argv[2]);
 }
 satype = SADB_SATYPE_AH;
 if ((alg = getsaalgbyname(satype, argv[3])) < 0) {
 err_quit("Unknown SA type / algorithm pair ah/%s\n", argv[3]);
 }
 keybits = strtoul(argv[4], &ep, 0);
 if (ep == argv[4] || *ep != '\0' || (keybits % 8) != 0) {
 err_quit("Invalid number of bits %s\n", argv[4]);
 }
 p = argv[5];
 if (p[0] == '0' && (p[1] == 'x' || p[1] == 'X'))
 p += 2;
}

```

```

len = strlen(p);
kp = keydata = malloc(keybits / 8);
for (i = 0; i < keybits; i += 8) {
 int c;

 if (len < 2) {
 err_quit("%s: not enough bytes (expected %d)\n", argv[5], keybits
/ 8);
 }
 if (sscanf(p, "%2x", &c) != 1) {
 err_quit("%s contains invalid hex digit\n", argv[5]);
 }
 *kp++ = c;
 p += 2;
 len -= 2;
}
if (len > 0) {
 err_quit("%s: too many bytes (expected %d)\n", argv[5], keybits / 8);
}
sadb_add(src->ai_addr, dst->ai_addr, satype, alg, 0x9876, keybits, keydata);
}

```

## 19.5 动态维护安全关联

- 周期性地重新产生密钥有助于进一步提高安全性
- 密钥管理守护进程预先使用SADB\_REGISTER请求消息向内核注册自身，在相应的SADB\_REGISTER应答消息中，内核提供一个受支持算法扩展，指出哪些加密和/或认证机制及密钥长度得到支持
- 

---

```

struct sadb_supported {
 u_int16_t sadb_supported_len; /* length of extension + algorithms / 8 */
 u_int16_t sadb_supported_exttype; /* SADB_EXT_SUPPORTED_(AUTH,ENCRYPT) */
 u_int32_t sadb_supported_reserved; /* reserved for future expansion */
};

/* followed by algorithm list */

struct sadb_alg {
 u_int8_t sadb_alg_id; /* algorithm ID from Figure 19.8 */
 u_int8_t sadb_alg_ivlen; /* IV length, or zero */
 u_int16_t sadb_alg_minbits; /* minimum key length */
 u_int16_t sadb_alg_maxbits; /* maximum key length */
 u_int16_t sadb_alg_reserved; /* reserved for future expansion */
};

```

---

图19-13 受支持算法扩展

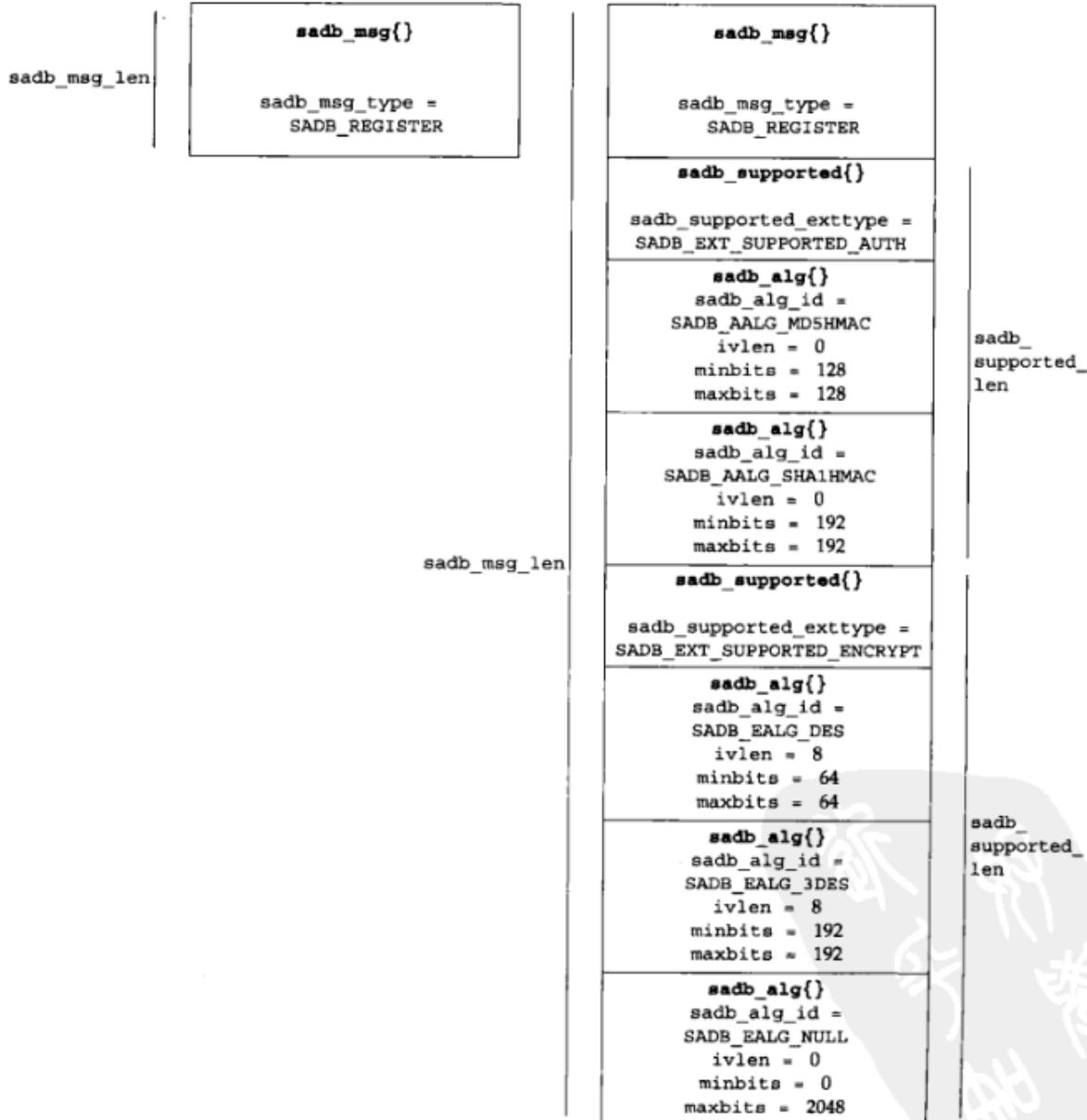


图19-14 内核为SADB\_REGISTER请求返回的应答

- 当内核需与某个目的地址通信时，如果根据策略该单向分组流必须经由一个SA而内核却并没有一个SA可用，内核就向注册了所需SA类型的密钥管理套接字发送一个SADB\_ACQUIRE消息，其中含有一个描述内核所提议算法及密钥长度的提议扩展。该提议可能综合了系统支持的配置与限制该单向分组流的预配置策略。提议内容是一个由算法、密钥长度和生命周期构成的按照优先顺序排列的列表。当一个密钥管理守护进程收到一个SADB\_ACQUIRE消息之后，它执行必要的操作以选择一个符合内核之提议的密钥，再把该密钥安装到内核中。它使用SADB\_GETSPI消息请求内核从一个期望的范围内选择一个SPI。内核对于该SADB\_GETSPI消息的响应包括建立一个处于幼虫（larval）状态的SA。然后守护进程使用由内核提供的这个SPI与远端协商安全参数，接着使用SADB\_UPDATE更新该SA，使它进入成熟（mature）状态。动态创建的SA通常还有关联的软生命周期和硬生命周期。当任何一个生命周期结束时，内核将发送一个SADB\_EXPIRE消息，其中指出期满的是软生命周期还是硬生命周期。如果软生命周期结束，其SA就进入垂死（dying）状态，期间它仍然可以使用，不过内核应该为它获取一个新的SA。如果硬生命周期结束，其SA就进入死亡（dead）状态，这种状态的SA不能继续使用，必须从SADB中删除。

- ```

#include "unp.h"
#include <net/pfkeyv2.h>

/* include sadb_register */
void
sadb_register(int type)
{
    int s;
    char buf[4096]; /* XXX */
    struct sadb_msg msg;
    int goteof;
    int mypid;

    s = Socket(PF_KEY, SOCK_RAW, PF_KEY_V2);

    mypid = getpid();

    /* Build and write SADB_REGISTER request */
    bzero(&msg, sizeof(msg));
    msg.sadb_msg_version = PF_KEY_V2;
    msg.sadb_msg_type = SADB_REGISTER;
    msg.sadb_msg_satype = type;
    msg.sadb_msg_len = sizeof(msg) / 8;
    msg.sadb_msg_pid = mypid;
    printf("Sending register message:\n");
    print_sadb_msg(&msg, sizeof(msg));
    Write(s, &msg, sizeof(msg));

    printf("\nReply returned:\n");
    /* Read and print SADB_REGISTER reply, discarding any others */
    for (;;) {
        int msglen;
        struct sadb_msg *msgp;

        msglen = Read(s, &buf, sizeof(buf));
        msgp = (struct sadb_msg *)&buf;
        if (msgp->sadb_msg_pid == mypid &&
            msgp->sadb_msg_type == SADB_REGISTER) {
            print_sadb_msg(msgp, msglen);
            break;
        }
    }
    close(s);
}

/* end sadb_register */

int
main(int argc, char **argv)
{
    int satype = SADB_SATYPE_UNSPEC;
    int c;

    opterr = 0;           /* don't want getopt() writing to stderr */

```

```

while ( (c = getopt(argc, argv, "t:")) != -1) {
    switch (c) {
        case 't':
            if ((satype = getsatypename(optarg)) == -1)
                err_quit("invalid -t option %s", optarg);
            break;

        default:
            err_quit("unrecognized option: %c", c);
    }
}

if (satype == SADB_SATYPE_UNSPEC) {
    err_quit("must specify SA type");
}

sadb_register(satype);
}

```

- 密钥管理套接字用于在内核、密钥管理守护进程以及诸如路由守护进程等安全服务消费进程之间交换SA。SA既可以手工静态安装，也可以使用密钥协商协议自动动态安装
- 每个由进程发送的消息被内核回射到所有其他打开着的密钥管理套接字，不过任何含有敏感数据的扩展会被抹除

第20章 广播

- 不同的寻址方式

类 型	IPv4	IPv6	TCP	UDP	所标识接口数	递送到接口数
单播	•	•	•	•	一个	一个
任播	•	•	尚没有	•	一组	一组中的一个
多播	可选	•		•	一组	一组中的全体
广播	•			•	全体	全体

图20-1 不同的寻址方式

- 广播的用途
 - 在本地子网定位一个服务器主机，资源发现
 - 在有多个客户主机与单个服务器主机通信的局域网环境中尽量减少分组流通
 - ARP, 使用链路层广播
 - DHCP
 - NTP, 网络时间协议
 - 路由守护进程，在一个局域网上广播自己的路由表

20.2 广播地址

- 子网定向广播地址: {子网ID, -1},通常路由器不转发这种广播
- 受限广播地址: {-1,-1}或255.255.255.255, 路由器从不转发目的地址为255.255.255.255的IP数据报。该地址用于主机配置过程中IP数据报的目的地址，此时，主机可能还不知道它所在网络的子网掩码。这样的数据报仅出现在本地网络中

20.3 单播和广播的比较

- 单播IP数据报仅由通过目的IP地址指定的单个主机接收，子网上的其他主机都不受任何影响

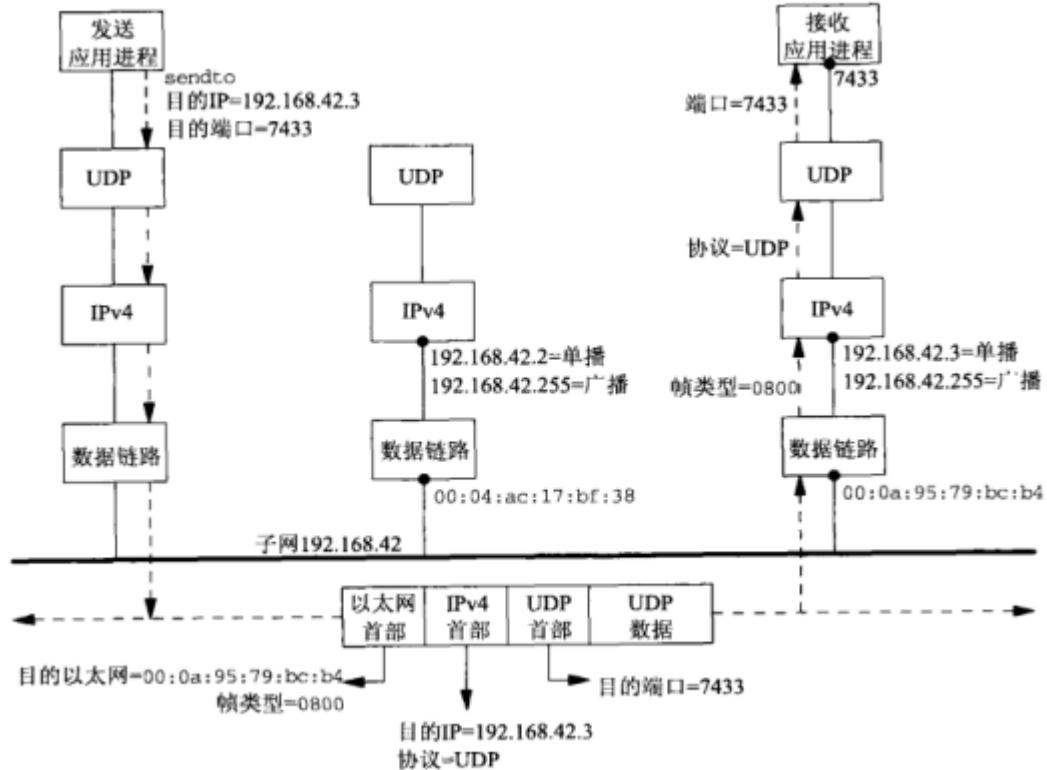


图20-3 UDP数据报单播示例

- 子网定向广播地址映射成全为1的以太网地址，使得子网上的每一个以太网接口都接收该帧

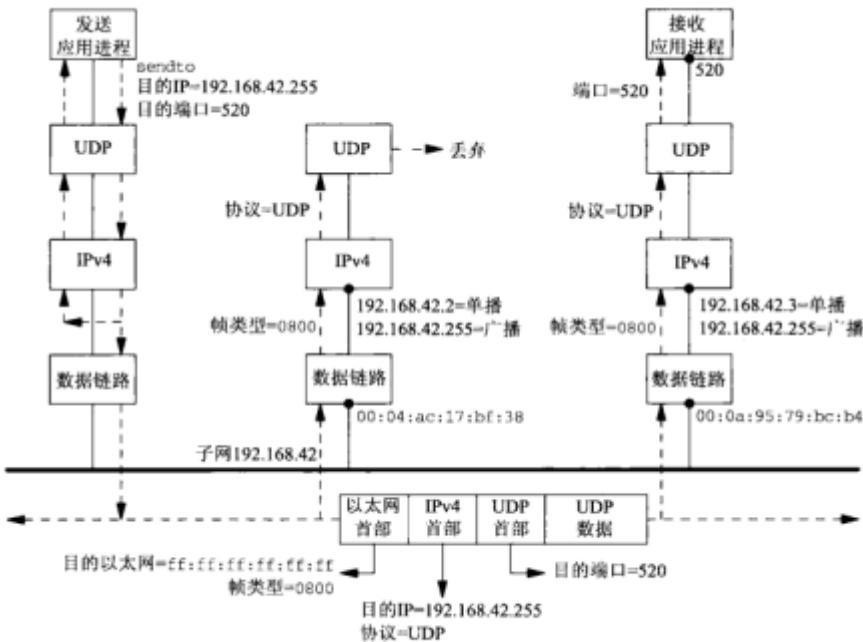


图20-4 UDP数据报广播示例

- 如果中间的主机没有任何应用进程绑定UDP端口520,该主机的UDP代码于是丢弃这个已收取的数据报，不能发送一个ICMP端口不可达消息
- 广播存在的根本问题: 子网上未参加相应广播应用的所有主机也不得不沿协议栈一路向上完整的处理收取的UDP广播数据报，直到丢弃为止

20.4 使用广播的dg_cli函数

- 通过设置SO_BROADCAST套接字选项允许发送广播数据报

```

• #include      "unp.h"

static void      recvfrom_alarm(int);

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int                  n;
    const int            on = 1;
    char                sendline[MAXLINE], recvline[MAXLINE + 1];
    socklen_t            len;
    struct sockaddr     *preply_addr;

    preply_addr = Malloc(servlen);

    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

    Signal(SIGALRM, recvfrom_alarm);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

        alarm(5);
        for ( ; ; ) {
            len = servlen;
            n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
            if (n < 0) {
                if (errno == EINTR)
                    break;           /* waited long enough for replies
*/
                else
                    err_sys("recvfrom error");
            } else {
                recvline[n] = 0;      /* null terminate */
                printf("from %s: %s",
                       Sock_ntop_host(preply_addr, len),
                recvline);
            }
        }
        free(preply_addr);
    }

    static void
    recvfrom_alarm(int signo)
    {
        return;          /* just interrupt the recvfrom() */
    }
}

```

- 广播数据报的目的主机是包括发送主机在内的接入同一个子网的所有主机，所有应答数据报都是单播的

- 某些内核不允许对广播数据报执行分片，如果其大小超过外出接口的MTU,发送它的系统调用就返回EMSGSIZE错误，使用SIOCGIFMTU ioctl确定外出接口的MTU

20.5 竞争状态

- 当多个进程访问共享的数据
- 当涉及信号处理时，信号会在程序执行过程中由内核随时随地递交
- 尽管我们的意图是让信号处理函数中断某个阻塞中的recvfrom,然而信号却可以在任何时刻被递交，当它被递交时，我们可能在无限for循环中的任何地方执行
- 阻塞和解阻塞信号(不正确方法)
 - 如果SIGALRM信号恰在recvfrom返回最后一个应答数据报之后与接着阻塞该信号之间递交，那么下一次调用recvfrom将永远阻塞

```

#include      "unp.h"

static void    recvfrom_alarm(int);

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int                  n;
    const int            on = 1;
    char                sendline[MAXLINE], recvline[MAXLINE + 1];
    sigset(SIGALRM,      sigset_alarm);
    socklen_t            len;
    struct sockaddr     *preply_addr;

    preply_addr = Malloc(servlen);

    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

    Sigemptyset(&sigset_alarm);
    Sigaddset(&sigset_alarm, SIGALRM);

    Signal(SIGALRM, recvfrom_alarm);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

        alarm(5);
        for ( ; ; ) {
            len = servlen;
            Sigprocmask(SIG_UNBLOCK, &sigset_alarm, NULL);
            n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr,
&len);
            Sigprocmask(SIG_BLOCK, &sigset_alarm, NULL);
            if (n < 0) {
                if (errno == EINTR)
                    break;           /* waited long enough for
replies */
            }
        }
    }
}

```

```

        else
            err_sys("recvfrom error");
    } else {
        recvline[n] = 0;           /* null terminate */
        printf("from %s: %s",
               Sock_ntop_host(preply_addr, len),
               recvline);
    }
}
free(preply_addr);
}

static void
recvfrom_alarm(int signo)
{
    return;                  /* just interrupt the recvfrom() */
}

```

- 用pselect阻塞和解阻塞信号

- pselect的关键点在于: 设置信号掩码、测试描述符、恢复信号掩码这3个操作在调用进程看来自成原子操作

```

#include      "unp.h"

static void      recvfrom_alarm(int);

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int
    const int          on = 1;
    char             sendline[MAXLINE], recvline[MAXLINE + 1];
    fd_set
    sigset_t          sigset_alm, sigset_empty;
    socklen_t          len;
    struct sockaddr *preply_addr;

    preply_addr = Malloc(servlen);

    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

    FD_ZERO(&rset);

    Sigemptyset(&sigset_empty);
    Sigemptyset(&sigset_alm);
    Sigaddset(&sigset_alm, SIGALRM);

    Signal(SIGALRM, recvfrom_alarm);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

        Sigprocmask(SIG_BLOCK, &sigset_alm, NULL);
    }
}

```

```

        alarm(5);
        for ( ; ; ) {
            FD_SET(sockfd, &rset);
            n = pselect(sockfd+1, &rset, NULL, NULL, NULL,
&sigset_empty);
            if (n < 0) {
                if (errno == EINTR)
                    break;
                else
                    err_sys("pselect error");
            } else if (n != 1)
                err_sys("pselect error: returned %d", n);

            len = servlen;
            n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr,
&len);
            recvline[n] = 0;           /* null terminate */
            printf("from %s: %s",
                   Sock_ntop_host(preply_addr, len), recvline);
        }
    }
    free(preply_addr);
}

static void
recvfrom_alarm(int signo)
{
    return;           /* just interrupt the recvfrom() */
}

```

- 使用sigsetjmp和siglpngjmp

```

○ #include      "unp.h"
#include      <setjmp.h>

static void                  recvfrom_alarm(int);
static sigjmp_buf      jmpbuf;

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int                  n;
    const int          on = 1;
    char                sendline[MAXLINE], recvline[MAXLINE + 1];
    socklen_t          len;
    struct sockaddr *preply_addr;

    preply_addr = Malloc(servlen);

    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

    Signal(SIGALRM, recvfrom_alarm);

```

```

        while (Fgets(sendline, MAXLINE, fp) != NULL) {

            Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

            alarm(5);
            for ( ; ; ) {
                if (sigsetjmp(jmpbuf, 1) != 0)
                    break;
                len = servlen;
                n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr,
&len);
                recvline[n] = 0;           /* null terminate */
                printf("from %s: %s",
                       Sock_ntop_host(preply_addr, len), recvline);
            }
        }
        free(preply_addr);
    }

    static void
    recvfrom_alarm(int signo)
{
    siglongjmp(jmpbuf, 1);
}

```

- 使用从信号处理函数到主控函数的IPC

```

○ #include      "unp.h"

static void      recvfrom_alarm(int);
static int       pipefd[2];

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int                         n, maxfdp1;
    const int                   on = 1;
    char                        sendline[MAXLINE], recvline[MAXLINE + 1];
    fd_set                      rset;
    socklen_t                   len;
    struct sockaddr *preply_addr;

    preply_addr = Malloc(servlen);

    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

    Pipe(pipefd);
    maxfdp1 = max(sockfd, pipefd[0]) + 1;

    FD_ZERO(&rset);

    Signal(SIGALRM, recvfrom_alarm);

```

```

        while (Fgets(sendline, MAXLINE, fp) != NULL) {
            Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

            alarm(5);
            for ( ; ; ) {
                FD_SET(sockfd, &rset);
                FD_SET(pipefd[0], &rset);
                if ( (n = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
                    if (errno == EINTR)
                        continue;
                    else
                        err_sys("select error");
                }

                if (FD_ISSET(sockfd, &rset)) {
                    len = servlen;
                    n = Recvfrom(sockfd, recvline, MAXLINE, 0,
preply_addr, &len);
                    recvline[n] = 0;           /* null terminate */
                    printf("from %s: %s",
                           Sock_ntop_host(preply_addr, len),
recvline);
                }

                if (FD_ISSET(pipefd[0], &rset)) {
                    Read(pipefd[0], &n, 1);      /* timer expired */
                    break;
                }
            }
            free(preply_addr);
        }

        static void
recvfrom_alarm(int signo)
{
    Write(pipefd[1], "", 1);          /* write one null byte to pipe */
    return;
}

```

- 让信号处理函数使用IPC通知主控函数定时器已到时，select同时测试套接字和管道的可读性

第21章 多播

- 多播地址标识一组IP接口，多播数据只应该由对它感兴趣的接口接收，多播既可以用于局域网，也可以跨广域网使用
- 9个套接字选项

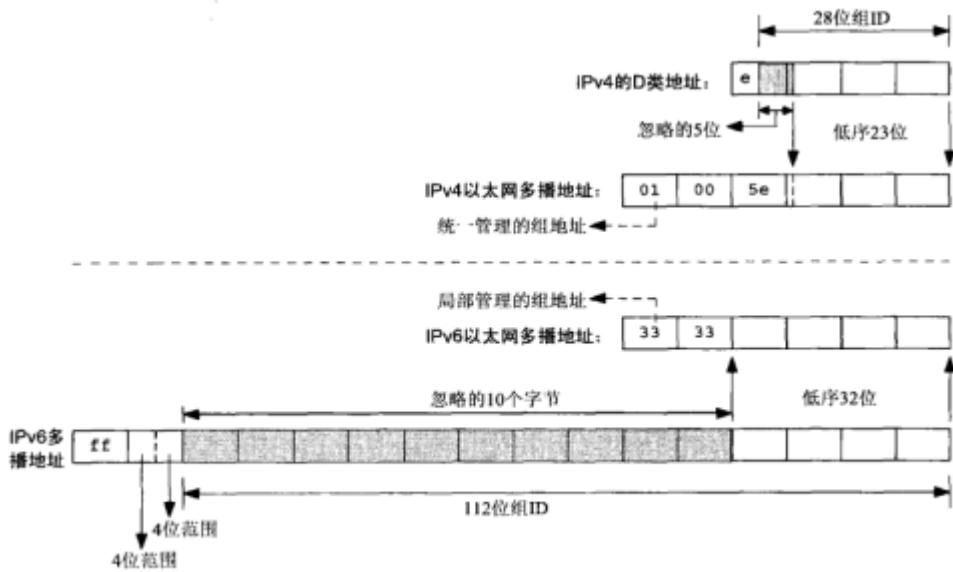


图21-1 IPv4和IPv6多播地址到以太网地址的映射

50

21.2 多播地址

- IPv4的D类地址
 - IPV4的D类地址，从224.0.0.0 到239.255.255.255是IPv4多播地址
 - 224.0.0.1是所有主机组，子网上所有具备多播能力的节点必须在所有具备多播能力的接口上加入该组
 - 224.0.0.2 是所有路由器组，子网上所有多播路由器必须在所有具备多播能力的接口上加入该组
 - 其余的为链路局部的多播地址，多播路由器从不转发以这些为目的地址的数据报
- IPv6多播地址
 - IPv6多播地址的高序字节值为ff

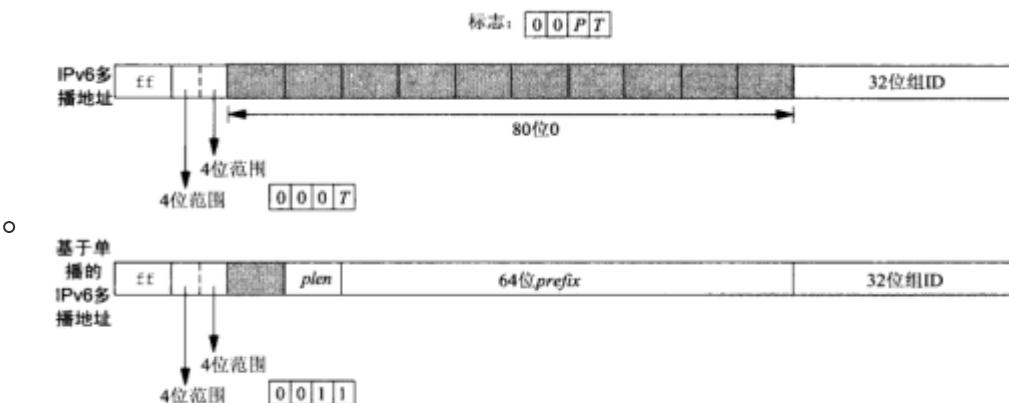


图21-2 IPv6多播地址格式

- ff01::1和ff02::1是所有节点组，子网上所有具备多播能力的节点必须在所有具备多播能力的接口上加入该组
- ff01::2 ff02::2和ff05::2是所有路由器组
- 多播地址的范围
 - IPv6多播地址显式存在一个4位的范围字段
 - 1: 接口局部的 (interface-local)。
 - 2: 链路局部的 (link-local)。
 - 4: 管区局部的 (admin-local)。
 - 5: 网点局部的 (site-local)。
 - 8: 组织机构局部的 (organization-local)。
 - 14: 全球或全局的 (global)。

14: 全球或全局的 (global)。

范 围	IPv6范围		IPv4
	TTL范围	可管理范围	
接口局部	1	0	
链路局部	2	1	224.0.0.0到224.0.0.255
网点局部	5	<32	239.255.0.0到239.255.255.255
组织机构局部	8		239.192.0.0到239.195.255.255
全球	14	≤255	224.0.1.0到238.255.255.255

图21-3 IPv4和IPv6多播地址范围

- 接口局部数据报不准由接口输出，链路局部数据报不可由路由器转发
- 多播会话: 一个多播地址和一个传输层端口的组合称为一个一个会话

21.3 局域网上多播和广播的比较

- 接收应用进程需要加入多播组，发送应用进程不必为此加入多播组
- 当我们告知一个以太网接口接收目的地址为某个特定以太网组地址的帧时，以太网接口对这个地址应用某个散列函数，当有一个目的地为某个组地址的帧在线缆上经过时，接口对齐目的地址应用同样的散列函数

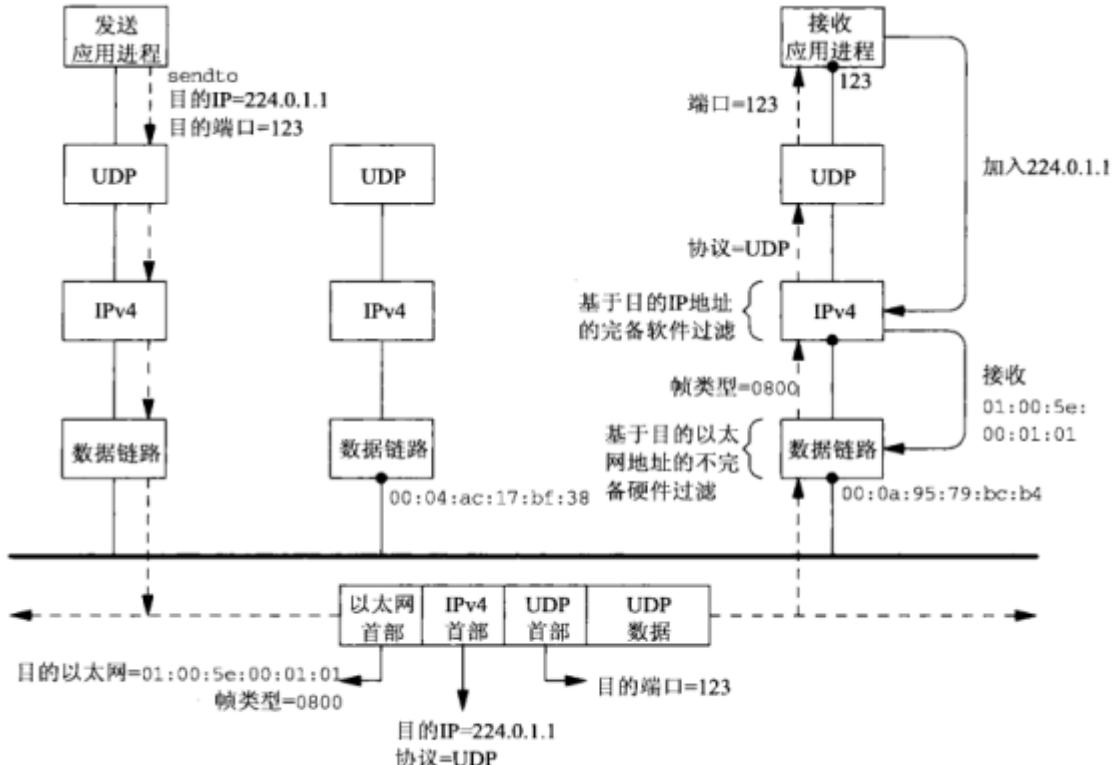
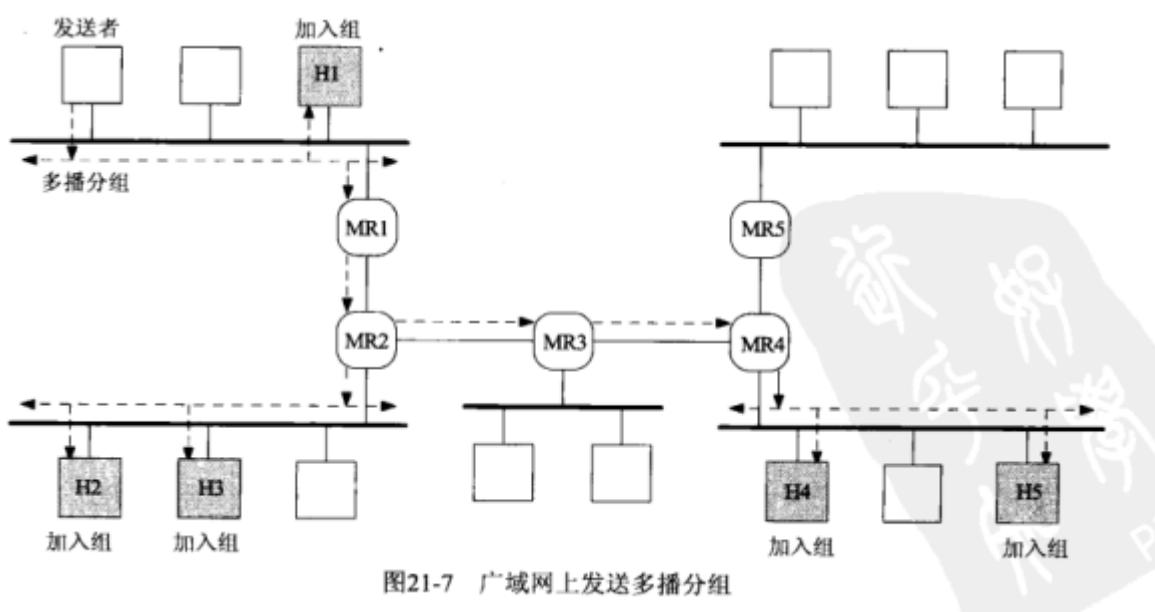
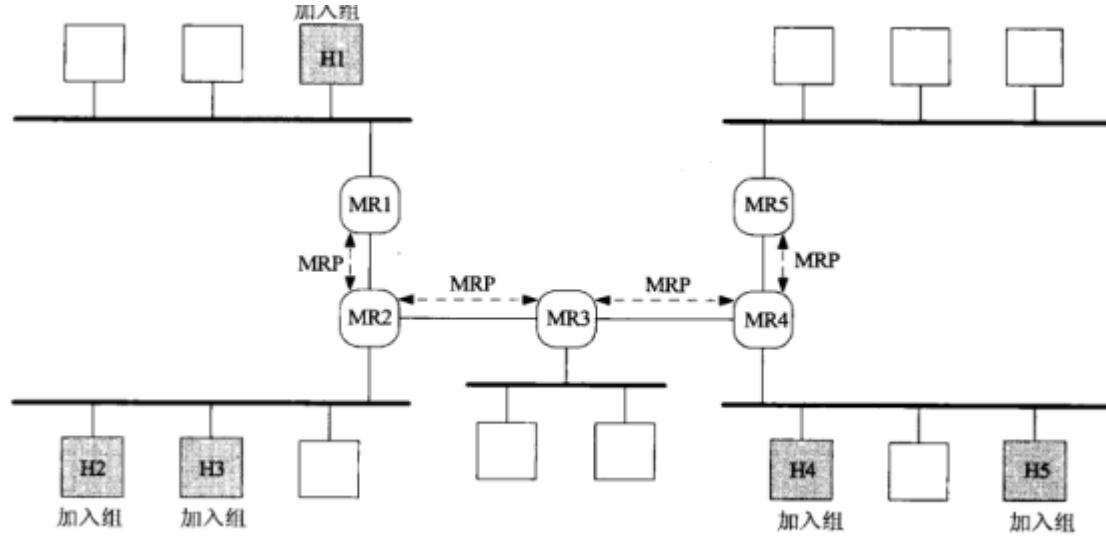


图21-4 UDP数据报多播示例

- (1) 运行所加入多播地址为225.0.1.1的某个应用进程的一个主机。既然多播地址组ID的高5位在到以太网地址的映射中被忽略，该主机的接口也将接收目的以太网地址为01:00:5e:00:01:01的帧。这种情况下，由该帧承载的分组将由IP层中的完备过滤丢弃。
- (2) 运行所加入多播地址符合以下条件的某个应用进程的一个主机：由这个多播地址映射成的以太网地址恰好和01:00:5e:00:01:01一样被该主机执行非完备过滤的接口散列到同一个结果。该接口也将接收目的以太网地址为01:00:5e:00:01:01的帧，直到由数据链路层或IP层丢弃。
- (3) 目的地为相同多播组(224.0.1.1)不同端口(譬如4000)的一个数据报。图21-4中右侧主机仍然接收该数据报，并由IP层接受并传递给UDP层，不过UDP层将丢弃它(假设绑定端口4000的套接字不存在)。

21.4 广域网上的多播

- 当某个主机上的一个进程加入一个多播组时，该主机向所有直接连接的多播路由器发送一个IGMP消息，告知它们本主机已加入了那个多播组，多播路由器随后使用MRP交换这些信息



21.5 源特定多播

- 源特定多播SSM把应用系统的源地址结合到组地址上

21.6 多播套接字选项

选项名	数据类型	说 明
IP_MULTICAST_IF	struct in_addr	指定外出多播数据报的默认接口
IP_MULTICAST_TTL	u_char	指定外出多播数据报的TTL
IP_MULTICAST_LOOP	u_char	开启或禁止外出多播数据报的回馈
IPV6_MULTICAST_IF	u_int	指定外出多播数据报的默认接口
IPV6_MULTICAST_HOPS	int	指定外出多播数据报的跳限
IPV6_MULTICAST_LOOP	u_int	开启或禁止外出多播数据报的回馈

图21-8 组成员无关多播套接字选项

选项名	数据类型	说 明
IP_ADD_MEMBERSHIP	struct ip_mreq	加入一个多播组
IP_DROP_MEMBERSHIP	struct ip_mreq	离开一个多播组
IP_BLOCK_SOURCE	struct ip_mreq_source	在一个已加入组上阻塞某个源
IP_UNBLOCK_SOURCE	struct ip_mreq_source	开通一个早先阻塞的源
IP_ADD_SOURCE_MEMBERSHIP	struct ip_mreq_source	加入一个源特定多播组
IP_DROP_SOURCE_MEMBERSHIP	struct ip_mreq_source	离开一个源特定多播组
IPV6_JOIN_GROUP	struct ipv6_mreq	加入一个多播组
IPV6_LEAVE_GROUP	struct ipv6_mreq	离开一个多播组
MCAST_JOIN_GROUP	struct group_req	加入一个多播组
MCAST_LEAVE_GROUP	struct group_req	离开一个多播组
MCAST_BLOCK_SOURCE	struct group_source_req	在一个已加入组上阻塞某个源
MCAST_UNBLOCK_SOURCE	struct group_source_req	开通一个早先阻塞的源
MCAST_JOIN_SOURCE_GROUP	struct group_source_req	加入一个源特定多播组
MCAST_LEAVE_SOURCE_GROUP	struct group_source_req	离开一个源特定多播组

图21-9 组成员相关多播套接字选项

- IP_ADD_MEMBERSHIP、IPV6_JOIN_GROUP MCAST_JOIN_GROUP
 - 在一个给定套接字上可以多次加入多播组，不过每次加入的必须是不同的多播地址，或者是在不同接口上的同一个多播地址
- IP_DROP_MEMBERSHIP IPV6_LEAVE_GROUP MCAST_LEAVE_GROUP
 - 离开指定的本地接口上不限源的多播组
- IP_BLOCK_SOURCE和MCAST_BLOCK_SOURCE
 - 在本套接字上阻塞接收来自某个源的多播分组
- IP_UNBLOCK 和MCAST_UNBLOCK_SPURCE
 - 开通被阻塞的源
- IP_ADD_SOURCE_MEMBERSHIP和MCAST_JOIN_SOURCE_GROUP
 - 加入一个特定于源的多播组
- IP_DROP_SOURCE_MEMBERSHIP和MCAST_LEAVE_SOURCE_GROUP
 - 离开一个特定于源的多播组
- IP_MULTICAST_IF和IPV6_MULTICAST_IF
 - 指定通过本套接字发送的多播数据的外出接口
- IP_MULTICAST_TTL和IPV6_MULTICAST_HOPS
 - 设置跳限，默认为1
- IP_MULTICAST_LOOP和IPV6_MULTICAST_LOOP
 - 开启或禁止多播数据报的本地自环

21.7 mcast_join和相关函数

- 使用多播的协议无关代码
- mcast_join

```
o /* include mcast_join1 */
#include      "unp.h"
#include      <net/if.h>

int
mcast_join(int sockfd, const SA *grp, socklen_t grplen,
           const char *ifname, u_int ifindex)
{
#ifdef MCAST_JOIN_GROUP
    struct group_req req;
    if (ifindex > 0) {
        req.gr_interface = ifindex;
    } else if (ifname != NULL) {
        if ( (req.gr_interface = if_nametoindex(ifname)) == 0) {
            errno = ENXIO; /* i/f name not found */
            return(-1);
        }
    } else
        req.gr_interface = 0;
    if (grplen > sizeof(req.gr_group)) {
        errno = EINVAL;
        return -1;
    }
    memcpy(&req.gr_group, grp, grplen);
    return (setsockopt(sockfd, family_to_level(grp->sa_family),
                      MCAST_JOIN_GROUP, &req, sizeof(req)));
#else
/* end mcast_join1 */

/* include mcast_join2 */
switch (grp->sa_family) {
case AF_INET: {
    struct ip_mreq      mreq;
    struct ifreq         ifreq;

    memcpy(&mreq.imr_multiaddr,
           &((const struct sockaddr_in *) grp)->sin_addr,
           sizeof(struct in_addr));

    if (ifindex > 0) {
        if (if_indextoname(ifindex, ifreq.ifr_name) == NULL) {
            errno = ENXIO; /* i/f index not found */
            return(-1);
        }
        goto doioctl;
    } else if (ifname != NULL) {
        strncpy(ifreq.ifr_name, ifname, IFNAMSIZ);
doioctl:
```

```

                if (ioctl(sockfd, SIOCGIFADDR, &ifreq) < 0)
                    return(-1);
                memcpy(&mreq.imr_interface,
                       &((struct sockaddr_in *) &ifreq.ifr_addr)-
>sin_addr,
                           sizeof(struct in_addr));
            } else
                mreq.imr_interface.s_addr = htonl(INADDR_ANY);

            return(setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                             &mreq, sizeof(mreq)));
        }
/* end mcast_join2 */

/* include mcast_join3 */
#ifndef IPV6
#ifndef IPV6_JOIN_GROUP           /* APIv0 compatibility */
#define IPV6_JOIN_GROUP           IPV6_ADD_MEMBERSHIP
#endif
#endif
case AF_INET6: {
    struct ipv6_mreq      mreq6;

    memcpy(&mreq6.ipv6mr_multiaddr,
           &((const struct sockaddr_in6 *) grp)->sin6_addr,
           sizeof(struct in6_addr));

    if (ifindex > 0) {
        mreq6.ipv6mr_interface = ifindex;
    } else if (ifname != NULL) {
        if ( (mreq6.ipv6mr_interface = if_nametoindex(ifname)) == 0)
{
            errno = ENXIO; /* i/f name not found */
            return(-1);
        }
    } else
        mreq6.ipv6mr_interface = 0;

    return(setsockopt(sockfd, IPPROTO_IPV6, IPV6_JOIN_GROUP,
                     &mreq6, sizeof(mreq6)));
}
#endif

default:
    errno = EAFNOSUPPORT;
    return(-1);
}
#endif
}
/* end mcast_join3 */

void
Mcast_join(int sockfd, const SA *grp, socklen_t grplen,
           const char *ifname, u_int ifindex)

```

```

{
    if (mcast_join(sockfd, grp, grplen, ifname, ifindex) < 0)
        err_sys("mcast_join error");
}

int
mcast_join_source_group(int sockfd, const SA *src, socklen_t srclen,
                        const SA *grp, socklen_t grplen,
                        const char *ifname, u_int ifindex)
{
#endif MCAST_JOIN_SOURCE_GROUP
    struct group_source_req req;
    if (ifindex > 0) {
        req.gsr_interface = ifindex;
    } else if (ifname != NULL) {
        if (req.gsr_interface = if_nametoindex(ifname)) == 0) {
            errno = ENXIO; /* i/f name not found */
            return(-1);
        }
    } else
        req.gsr_interface = 0;
    if (grplen > sizeof(req.gsr_group) || srclen > sizeof(req.gsr_source)) {
        errno = EINVAL;
        return -1;
    }
    memcpy(&req.gsr_group, grp, grplen);
    memcpy(&req.gsr_source, src, srclen);
    return (setsockopt(sockfd, family_to_level(grp->sa_family),
                      MCAST_JOIN_SOURCE_GROUP, &req, sizeof(req)));
}

#else
    switch (grp->sa_family) {
#endif IP_ADD_SOURCE_MEMBERSHIP
    case AF_INET: {
        struct ip_mreq_source    mreq;
        struct ifreq               ifreq;

        memcpy(&mreq.imr_multiaddr,
               &((struct sockaddr_in *) grp)->sin_addr,
               sizeof(struct in_addr));
        memcpy(&mreq.imr_sourceaddr,
               &((struct sockaddr_in *) src)->sin_addr,
               sizeof(struct in_addr));

        if (ifindex > 0) {
            if (if_indextoname(ifindex, ifreq.ifr_name) == NULL) {
                errno = ENXIO; /* i/f index not found */
                return(-1);
            }
            goto doioctl;
        } else if (ifname != NULL) {
            strncpy(ifreq.ifr_name, ifname, IFNAMSIZ);
    }
doioctl:
        if (ioctl(sockfd, SIOCGIFADDR, &ifreq) < 0)

```

```

                return(-1);
        memcpy(&mreq.imr_interface,
               &((struct sockaddr_in *) &ifreq.ifr_addr)-
>sin_addr,
               sizeof(struct in_addr));
    } else
        mreq.imr_interface.s_addr = htonl(INADDR_ANY);

    return(setsockopt(sockfd, IPPROTO_IP, IP_ADD_SOURCE_MEMBERSHIP,
                      &mreq, sizeof(mreq)));
}
#endif

#ifndef IPV6
case AF_INET6: /* IPv6 source-specific API is MCAST_JOIN_SOURCE_GROUP */
#endif
default:
    errno = EAFNOSUPPORT;
    return(-1);
}
#endif
}

void
Mcast_join_source_group(int sockfd, const SA *src, socklen_t srclen,
                       const SA *grp, socklen_t grplen,
                       const char *ifname, u_int ifindex)
{
    if (mcast_join_source_group(sockfd, src, srclen, grp, grplen,
                               ifname, ifindex) <
0)
        err_sys("mcast_join_source_group error");
}

int
mcast_block_source(int sockfd, const SA *src, socklen_t srclen,
                   const SA *grp, socklen_t grplen)
{
#ifndef MCAST_BLOCK_SOURCE
    struct group_source_req req;
    req.gsr_interface = 0;
    if (grplen > sizeof(req.gsr_group) || srclen > sizeof(req.gsr_source)) {
        errno = EINVAL;
        return -1;
    }
    memcpy(&req.gsr_group, grp, grplen);
    memcpy(&req.gsr_source, src, srclen);
    return (setsockopt(sockfd, family_to_level(grp->sa_family),
                      MCAST_BLOCK_SOURCE, &req, sizeof(req)));
#else
    switch (grp->sa_family) {
#ifndef IP_BLOCK_SOURCE
        case AF_INET: {

```

```

        struct ip_mreq_source    mreq;

        memcpy(&mreq.imr_multiaddr,
               &((struct sockaddr_in *) grp)->sin_addr,
               sizeof(struct in_addr));
        memcpy(&mreq.imr_sourceaddr,
               &((struct sockaddr_in *) src)->sin_addr,
               sizeof(struct in_addr));
        mreq.imr_interface.s_addr = htonl(INADDR_ANY);

        return(setsockopt(sockfd, IPPROTO_IP, IP_BLOCK_SOURCE,
                          &mreq, sizeof(mreq)));
    }
#endif

#ifndef IPV6
case AF_INET6: /* IPv6 source-specific API is MCAST_BLOCK_SOURCE */
#endif
default:
    errno = EAFNOSUPPORT;
    return(-1);
}
#endif
}

void
Mcast_block_source(int sockfd, const SA *src, socklen_t srclen,
                   const SA *grp, socklen_t grplen)
{
    if (mcast_block_source(sockfd, src, srclen, grp, grplen) < 0)
        err_sys("mcast_block_source error");
}

int
mcast_unblock_source(int sockfd, const SA *src, socklen_t srclen,
                     const SA *grp, socklen_t grplen)
{
#ifndef MCAST_UNBLOCK_SOURCE
    struct group_source_req req;
    req.gsr_interface = 0;
    if (grplen > sizeof(req.gsr_group) || srclen > sizeof(req.gsr_source)) {
        errno = EINVAL;
        return -1;
    }
    memcpy(&req.gsr_group, grp, grplen);
    memcpy(&req.gsr_source, src, srclen);
    return (setsockopt(sockfd, family_to_level(grp->sa_family),
                      MCAST_UNBLOCK_SOURCE, &req, sizeof(req)));
#else
    switch (grp->sa_family) {
#ifndef IP_UNBLOCK_SOURCE
        case AF_INET: {
            struct ip_mreq_source    mreq;

```

```

        memcpy(&mreq.imr_multiaddr,
               &((struct sockaddr_in *) grp)->sin_addr,
               sizeof(struct in_addr));
        memcpy(&mreq.imr_sourceaddr,
               &((struct sockaddr_in *) src)->sin_addr,
               sizeof(struct in_addr));
        mreq.imr_interface.s_addr = htonl(INADDR_ANY);

        return(setsockopt(sockfd, IPPROTO_IP, IP_UNBLOCK_SOURCE,
                          &mreq, sizeof(mreq)));
    }

#endif

#ifndef IPV6
    case AF_INET6: /* IPv6 source-specific API is MCAST_UNBLOCK_SOURCE */
#endif
default:
    errno = EAFNOSUPPORT;
    return(-1);
}
#endif
}

void
Mcast_unblock_source(int sockfd, const SA *src, socklen_t srclen,
                     const SA *grp, socklen_t grplen)
{
    if (mcast_unblock_source(sockfd, src, srclen, grp, grplen) < 0)
        err_sys("mcast_unblock_source error");
}

```

21.8 使用多播的dg_cli函数

- 分片数据报对于多播数据报不成问题

21.9 接收IP多播基础设施会话声明

21.10 发送和接收

21.11 SNTP: 简单网络时间协议

- 多播应用进程一开始就通过设置套接字选项请求加入赋予它的多播组，该请求告知IP层加入给定组，IP层再告知数据链路层接收发往相应硬件多播地址的多播帧，多播利用多数接口卡都提供的硬件过滤减少非期望分组的接收。

第22章 高级UDP套接字编程

22.2 接收标志、目的IP地址和接口索引

```
• /* include recvfrom_flags1 */
#include      "unp.h"
#include      <sys/param.h>           /* ALIGN macro for CMSG_NXTHDR() macro */

ssize_t
recvfrom_flags(int fd, void *ptr, size_t nbytes, int *flagsp,
               SA *sa, socklen_t *salenptr, struct unp_in_pktinfo *pktp)
{
    struct msghdr    msg;
    struct iovec     iov[1];
    ssize_t          n;

#ifndef HAVE_MSGHDR_MSG_CONTROL
    struct cmsghdr  *cmptr;
    union {
        struct cmsghdr   cm;
        char             control[CMSG_SPACE(sizeof(struct in_addr)) +
                               CMSG_SPACE(sizeof(struct
unp_in_pktinfo))];
    } control_un;

    msg.msg_control = control_un.control;
    msg.msg_controllen = sizeof(control_un.control);
    msg.msg_flags = 0;
#else
    bzero(&msg, sizeof(msg));           /* make certain msg_accrightslen = 0 */
#endif

    msg.msg_name = sa;
    msg.msg_namelen = *salenptr;
    iov[0].iov_base = ptr;
    iov[0].iov_len = nbytes;
    msg.msg iov = iov;
    msg.msg iovlen = 1;

    if ( (n = recvmsg(fd, &msg, *flagsp)) < 0)
        return(n);

    *salenptr = msg.msg_namelen;       /* pass back results */
    if (pktp)
        bzero(pktp, sizeof(struct unp_in_pktinfo));      /* 0.0.0.0, i/f = 0 */
/* end recvfrom_flags1 */

/* include recvfrom_flags2 */
#ifndef HAVE_MSGHDR_MSG_CONTROL
    *flagsp = 0;                      /* pass back results */
    return(n);
#else
```

```

        *flagsp = msg.msg_flags;           /* pass back results */
        if (msg.msg_controllen < sizeof(struct cmsghdr) ||
            (msg.msg_flags & MSG_CTRUNC) || pktp == NULL)
            return(n);

        for (cmptr = CMSG_FIRSTHDR(&msg); cmptr != NULL;
             cmptr = CMSG_NXTHDR(&msg, cmptr)) {

#ifdef IP_RECVDSTADDR
            if (cmptr->cmsg_level == IPPROTO_IP &&
                cmptr->cmsg_type == IP_RECVDSTADDR) {

                memcpy(&pktp->ipi_addr, CMSG_DATA(cmptr),
                       sizeof(struct in_addr));
                continue;
            }
#endif

#ifdef IP_RECVIF
            if (cmptr->cmsg_level == IPPROTO_IP &&
                cmptr->cmsg_type == IP_RECVIF) {
                struct sockaddr_dl      *sdl;

                sdl = (struct sockaddr_dl *) CMSG_DATA(cmptr);
                pktp->ipi_ifindex = sdl->sdl_index;
                continue;
            }
#endif
            err_quit("unknown ancillary data, len = %d, level = %d, type = %d",
                    cmptr->cmsg_len, cmptr->cmsg_level, cmptr->cmsg_type);
        }
        return(n);
#endif /* HAVE_MSGHDR_MSG_CONTROL */
    }
/* end recvfrom_flags2 */

ssize_t
Recvfrom_flags(int fd, void *ptr, size_t nbytes, int *flagsp,
               SA *sa, socklen_t *salenptr, struct unp_in_pktinfo *pktp)
{
    ssize_t          n;

    n = recvfrom_flags(fd, ptr, nbytes, flagsp, sa, salenptr, pktp);
    if (n < 0)
        err_quit("recvfrom_flags error");

    return(n);
}

```

22.3 数据报截断

- 当到达的一个UDP数据报超过应用进程提供的缓冲区容量时，`recvmsg`在其`msghdr`结构的`msg_flags`成员上设置`MSG_TRUNC`标志

22.4 何时用UDP代替TCP

- UDP支持广播和多播
- UDP没有连接建立和拆除
- 对于简单的请求—应答应用程序可以使用UDP,不过错误检测功能必须加到应用程序内部，错误检测至少涉及确认、超时和重传
- 对于海量数据传输不应该使用UDP

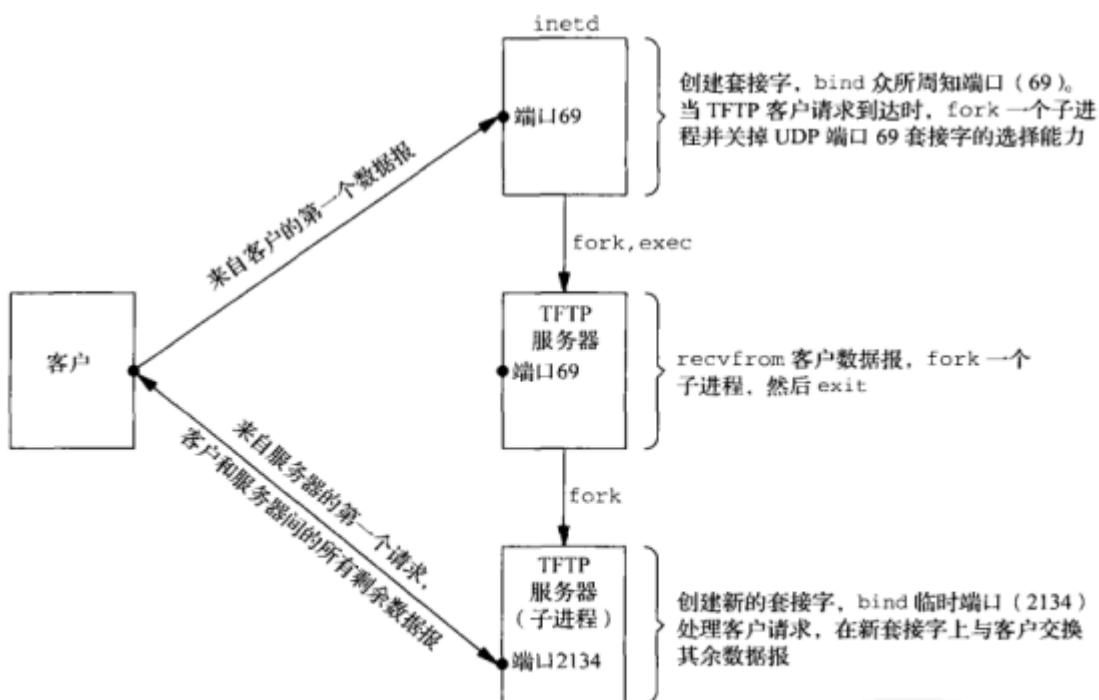
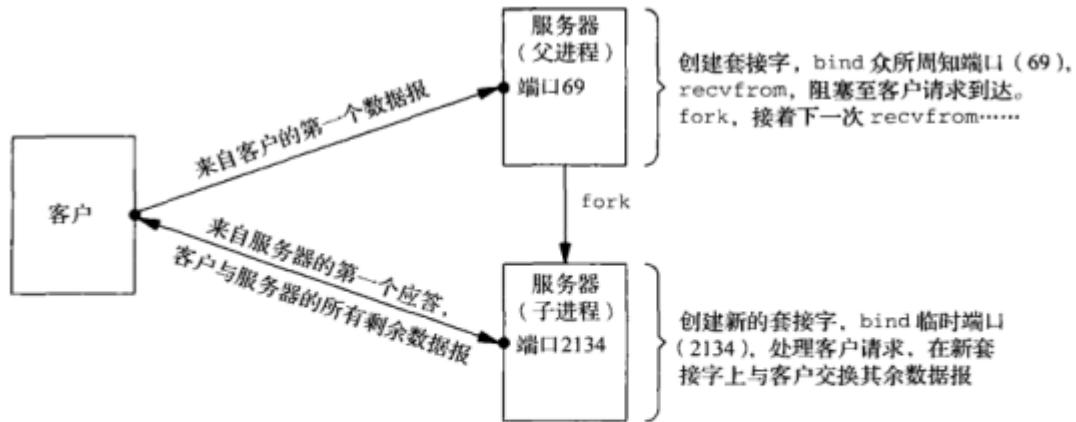
22.5 给UDP应用增加可靠性

- 超时和重传: 用于处理丢失的数据报
- 序列号: 供客户验证一个应答是否匹配相应的请求
- 通过给每个分组增加一个时间戳并追踪RTT及其平均偏差这两个估算因子，动态地修改重传超时值，还给每个分组增加一个序列号以验证某个给定应答是期望的应答

22.6 捆绑接口地址

- 有些应用程序需要知道某个UDP数据报的目的IP地址和接收接口，开启`IP_RECVDSTADDR`和`IP_RECVIF`套接字选项可以作为辅助数据随每个数据报返回这些信息

22.7 并发UDP服务器



22.8 IPv6分组信息

- sendmsg
- recvmsg
-

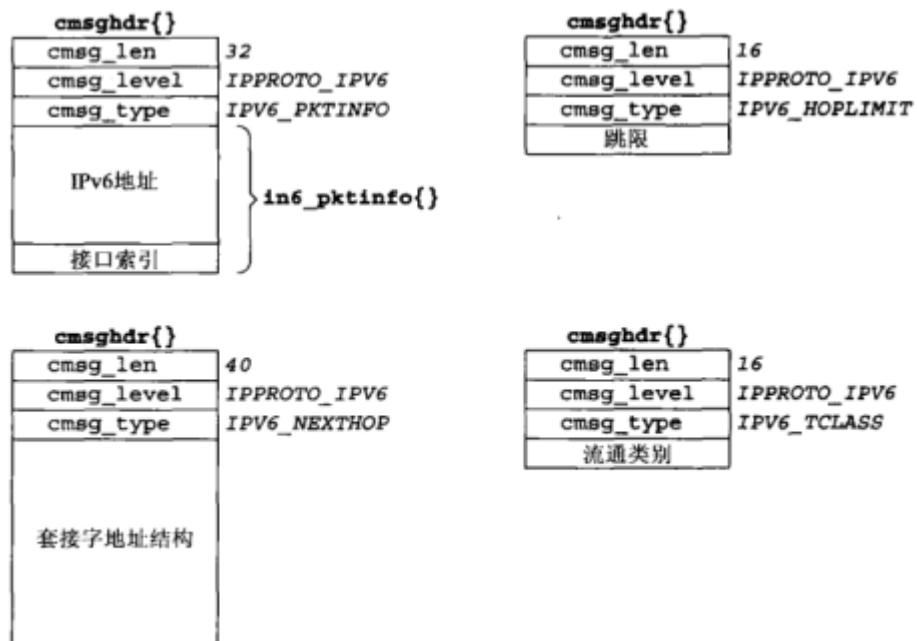


图22-21 IPv6分组信息的辅助数据

- 设置IPv6_PKTINFO套接字选项
- 开启IPV6_RECVPKTINFO套接字选项
- 外出和到达接口
 - 任何接口都不会被赋予0值索引
 - 如果ipi6_ifindex成员值为0,那么就由内核选择外出接口
- 源和目的IPv6地址
- 指定和接收跳限
 - IPV6_UNICAST_HOPS
 - IPV6_MULTICAST_HOPS
 - 跳限的正常值在0-255之间, 若为-1则告知内核使用默认值
- 指定下一跳地址
 - IPv6_NEXTHOP辅助数据对象将数据报的下一跳指定为一个套接字地址结构
- 指定和接收流通类别
 - IPv6_TCLASS辅助数据对象指定数据报的流通类别
 - 如果为某个给定分组指定流通类别, 那么只需要包含辅助数据, 如果为通过某个套接字的所有分组指定流通类别, 那么就以一个整数选项值设置IPv6_TCLASS套接字选项

22.9 IPv6路径MTU控制

- 以最小MTU发送
 - IPv6_USE_MIN_MTU套接字选项控制, 也可以作为辅助数据发送
- 接收路径MTU变动指示
 - 开启IPv6_RECVPATHMTU套接字选项以接收MTU变动通知, 本标志值使得任何时候路径MTU发生变动时作为辅助数据由recvmsg返回变动后的路劲MTU
- 确定当前路径MTU
 - IPV6_PATHMTU套接字选项确定某个已连接套接字的当前路径MTU
- 避免分片

- IPV6_DONTFRAG套接字选项用于关闭自动分片特性

第23章 高级SCTP套接字编程

- SCTP是一个面向消息的协议，递送给用户的是部分的或完整的消息
- SCTP还提供了从一到多式套接字抽取某个关联并使其成为一到一式套接字的方法，允许构造既可以迭代运行又可以并发运行的服务器程序

23.2 自动关闭的一到多式服务器程序

- 自动关闭允许SCTP端点指定某个关联可以保持空闲的最大秒钟数
- 开启必须显式地使用SCTP_AUTOCLOSE套接字选项

23.2 部分递送

- 当应用进程要求SCTP传输过大的消息时，SCTP可能采取部分递送措施
 - 所接收消息的缓冲区空间耗用量必须满足或超过某个门槛。
 - SCTP栈最多只能从该消息开始处顺序递送到首个缺失断片。
 - 一旦激发，其他消息必须等到当前消息已被完整地接收并递送给接收端应用进程之后才能被递送。也就是说过大的消息会阻塞通常情况下可以递送的所有其他消息的递送，包括其他流中的消息。
- 处理部分递送API

```

◦ #include      "unp.h"

static uint8_t *sctp_pdapi_readbuf=NULL;
static int sctp_pdapi_rdbuf_sz=0;

uint8_t *
pdapi_recvmsg(int sock_fd,
              int *rdlen,
              SA *from,
              int *from_len,
              struct sctp_sndrcvinfo *sri,
              int *msg_flags)
{
    int rdsz, left, at_in_buf;
    int frmlen=0;

    if (sctp_pdapi_readbuf == NULL) {
        sctp_pdapi_readbuf = (uint8_t *)Malloc(SCTP_PDAPI_INCR_SZ);
        sctp_pdapi_rdbuf_sz = SCTP_PDAPI_INCR_SZ;
    }
    at_in_buf = Sctp_recvmsg(sock_fd, sctp_pdapi_readbuf, sctp_pdapi_rdbuf_sz,
                           from, from_len,
                           sri, msg_flags);
    if(at_in_buf < 1){
        *rdlen = at_in_buf;
        return(NULL);
    }
}

```

```

        while((*msg_flags & MSG_EOR) == 0) {
            left = sctp_pdapi_rdbuf_sz - at_in_buf;
            if(left < SCTP_PDAPI_NEED_MORE_THRESHOLD) {
                sctp_pdapi_readbuf = realloc(sctp_pdapi_readbuf,
sctp_pdapi_rdbuf_sz+SCTP_PDAPI_INCR_SZ);
                if(sctp_pdapi_readbuf == NULL) {
                    err_quit("sctp_pdapi ran out of memory");
                }
                sctp_pdapi_rdbuf_sz += SCTP_PDAPI_INCR_SZ;
                left = sctp_pdapi_rdbuf_sz - at_in_buf;
            }
            rdsz = Sctp_recvmsg(sock_fd, &sctp_pdapi_readbuf[at_in_buf],
                                left, NULL, &frmlen, NULL, msg_flags);
            at_in_buf += rdsz;
        }
        *rdlen = at_in_buf;
        return(sctp_pdapi_readbuf);
    }
}

```

23.4 通知

- 应用进程可以预定7个通知
- 通知显示实用函数

- #include "unp.h"

```

void
print_notification(char *notify_buf)
{
    union sctp_notification *snp;
    struct sctp_assoc_change *sac;
    struct sctp_paddr_change *spc;
    struct sctp_remote_error *sre;
    struct sctp_send_failed *ssf;
    struct sctp_shutdown_event *sse;
    struct sctp_adaption_event *ae;
    struct sctp_pdapi_event *pdapi;
    const char *str;

    snp = (union sctp_notification *)notify_buf;
    switch(snp->sn_header.sn_type) {
    case SCTP_ASSOC_CHANGE:
        sac = &snp->sn_assoc_change;
        switch(sac->sac_state) {
        case SCTP_COMM_UP:
            str = "COMMUNICATION UP";
            break;
        case SCTP_COMM_LOST:
            str = "COMMUNICATION LOST";
            break;
        case SCTP_RESTART:
            str = "RESTART";
            break;
        }
    }
}

```

```

        break;
case SCTP_SHUTDOWN_COMP:
        str = "SHUTDOWN COMPLETE";
        break;
case SCTP_CANT_STR_ASSOC:
        str = "CAN'T START ASSOC";
        break;
default:
        str = "UNKNOWN";
        break;
} /* end switch(sac->sac_state) */
printf("SCTP_ASSOC_CHANGE: %s, assoc=0x%x\n", str,
       (uint32_t)sac->sac_assoc_id);
break;
case SCTP_PEER_ADDR_CHANGE:
spc = &snp->sn_paddr_change;
switch(spc->spc_state) {
case SCTP_ADDR_AVAILABLE:
        str = "ADDRESS AVAILABLE";
        break;
case SCTP_ADDR_UNREACHABLE:
        str = "ADDRESS UNREACHABLE";
        break;
case SCTP_ADDR_REMOVED:
        str = "ADDRESS REMOVED";
        break;
case SCTP_ADDR_ADDED:
        str = "ADDRESS ADDED";
        break;
case SCTP_ADDR_MADE_PRIM:
        str = "ADDRESS MADE PRIMARY";
        break;
default:
        str = "UNKNOWN";
        break;
} /* end switch(spc->spc_state) */
printf("SCTP_PEER_ADDR_CHANGE: %s, addr=%s, assoc=0x%x\n", str,
       Sock_ntop((SA *)&spc->spc_aaddr, sizeof(spc->spc_aaddr)),
       (uint32_t)spc->spc_assoc_id);
break;
case SCTP_REMOTE_ERROR:
sre = &snp->sn_remote_error;
printf("SCTP_REMOTE_ERROR: assoc=0x%x error=%d\n",
       (uint32_t)sre->sre_assoc_id, sre->sre_error);
break;
case SCTP_SEND_FAILED:
ssf = &snp->sn_send_failed;
printf("SCTP_SEND_FAILED: assoc=0x%x error=%d\n",
       (uint32_t)ssf->ssf_assoc_id, ssf->ssf_error);
break;
case SCTP_ADAPTION_INDICATION:
ae = &snp->sn_adaption_event;
printf("SCTP_ADAPTION_INDICATION: 0x%x\n",

```

```

        (u_int)ae->sai_adaption_ind);
    break;
case SCTP_PARTIAL_DELIVERY_EVENT:
    pdapi = &snp->sn_pdapi_event;
    if(pdapi->pdapi_indication == SCTP_PARTIAL_DELIVERY_ABORTED)
        printf("SCTP_PARTIAL_DELIVERY_ABORTED\n");
    else
        printf("Unknown SCTP_PARTIAL_DELIVERY_EVENT 0x%x\n",
               pdapi->pdapi_indication);
    break;
case SCTP_SHUTDOWN_EVENT:
    sse = &snp->sn_shutdown_event;
    printf("SCTP_SHUTDOWN_EVENT: assoc=0x%x\n",
           (uint32_t)sse->sse_assoc_id);
    break;
default:
    printf("Unknown notification event type=0x%x\n",
           snp->sn_header.sn_type);
}
}

```

23.5 无序的数据

- SCTP通常提供可靠的有序数据传输服务，不过也提供可靠的无序数据传输服务
- 指定MSG_UNORDERED标志发送的消息没有顺序限制
- 通常情况下，一个给定SCTP流中的所有数据都标以序列化以便排序，该标志使相应数据以无序方式发送，即不标以序列号，一到达对端就能被递送

23.6 捆绑地址子集

- TCP和UDP传统上只能捆绑单个地址，而不能捆绑一个地址子集。由SCTP提供的新的sctp_bindx函数调用允许应用进程捆绑不止一个地址，所有地址必须使用相同的端口

```

#include      "unp.h"

int
sctp_bind_arg_list(int sock_fd, char **argv, int argc)
{
    struct addrinfo *addr;
    char *bindbuf, *p, portbuf[10];
    int addrcnt=0;
    int i;

    bindbuf = (char *)Calloc(argc, sizeof(struct sockaddr_storage));
    p = bindbuf;
    sprintf(portbuf, "%d", SERV_PORT);
    for( i=0; i<argc; i++ ) {
        addr = Host_serv(argv[i], portbuf, AF_UNSPEC, SOCK_SEQPACKET);
        memcpy(p, addr->ai_addr, addr->ai_addrlen);
        freeaddrinfo(addr);
        addrcnt++;
        p += addr->ai_addrlen;
    }
}

```

```

    }
    Sctp_bindx(sock_fd, (SA *)bindbuf, addrcnt, SCTP_BINDX_ADD_ADDR);
    free(bindbuf);
    return(0);
}

```

23.7 确定对端和本端地址信息

- SCTP是一个多宿协议，找出一个关联的本地端点和远程端点所用的地址需要使用不同于单宿协议的机制

- #include "unp.h"

```

int
main(int argc, char **argv)
{
    int sock_fd;
    struct sockaddr_in servaddr;
    struct sctp_event_subscribe evnts;

    if(argc != 2)
        err_quit("Missing host argument - use '%s host'\n",
                 argv[0]);
    sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

/* include mod_client04 */
    bzero(&evnts, sizeof(evnts));
    evnts.sctp_data_io_event = 1;
    evnts.sctp_association_event = 1;
    Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS,
               &evnts, sizeof(evnts));

    sctpstr_cli(stdin, sock_fd, (SA *)&servaddr, sizeof(servaddr));
/* end mod_client04 */
    close(sock_fd);
    return(0);
}

```

- #include "unp.h"

```

void
sctpstr_cli(FILE *fp, int sock_fd, struct sockaddr *to, socklen_t tolen)
{
    struct sockaddr_in peeraddr;
    struct sctp_sndrcvinfo sri;
    char sendline[MAXLINE], recvline[MAXLINE];
    socklen_t len;
    int out_sz, rd_sz;
    int msg_flags;

```

```

bzero(&sri,sizeof(sri));
while (fgets(sendline, MAXLINE, fp) != NULL) {
    if(sendline[0] != '[') {
        printf("Error, line must be of the form '[streamnum]text'\n");
        continue;
    }
    sri.sinfo_stream = strtol(&sendline[1],NULL,0);
    out_sz = strlen(sendline);
    Sctp_sendmsg(sock_fd, sendline, out_sz,
                  to, tolen,
                  0, 0,
                  sri.sinfo_stream,
                  0, 0);
/* include mod_strcli1 */
    do {
        len = sizeof(peeraddr);
        rd_sz = Sctp_recvmsg(sock_fd, recvline, sizeof(recvline),
                             (SA *)&peeraddr, &len,
                             &sri,&msg_flags);
        if(msg_flags & MSG_NOTIFICATION)
            check_notification(sock_fd,recvline,rd_sz);
    } while (msg_flags & MSG_NOTIFICATION);
    printf("From str:%d seq:%d (assoc:0x%x):",
           sri.sinfo_stream,sri.sinfo_ssn,
           (u_int)sri.sinfo_assoc_id);
    printf("%..*s",rd_sz,recvline);
/* end mod_strcli1 */
    }
}

```

- #include "unp.h"

```

void
check_notification(int sock_fd,char *recvline,int rd_len)
{
    union sctp_notification *snp;
    struct sctp_assoc_change *sac;
    struct sockaddr_storage *sal,*sar;
    int num_rem, num_loc;

    snp = (union sctp_notification *)recvline;
    if(snp->sn_header.sn_type == SCTP_ASSOC_CHANGE) {
        sac = &snp->sn_assoc_change;
        if((sac->sac_state == SCTP_COMM_UP) ||
           (sac->sac_state == SCTP_RESTART)) {
            num_rem = sctp_getpaddrs(sock_fd,sac->sac_assoc_id,&sar);
            printf("There are %d remote addresses and they are:\n",
                   num_rem);
            sctp_print_addresses(sar,num_rem);
            sctp_freepaddrs(sar);

            num_loc = sctp_getladdrs(sock_fd,sac->sac_assoc_id,&sal);

```

```

        printf("There are %d local addresses and they are:\n",
               num_loc);
        sctp_print_addresses(sal, num_loc);
        sctp_freeLaddrs(sal);
    }
}

}

```

- ```
#include "unp.h"

void
sctp_print_addresses(struct sockaddr_storage *addrs, int num)
{
 struct sockaddr_storage *ss;
 int i, salen;

 ss = addrs;
 for(i=0; i<num; i++){
 printf("%s\n", Sock_ntop((SA *)ss, salen));
#endif HAVE_SOCKADDR_SA_LEN
 salen = ss->ss_len;
#else
 switch(ss->ss_family) {
 case AF_INET:
 salen = sizeof(struct sockaddr_in);
 break;
#endif IPV6
 case AF_INET6:
 salen = sizeof(struct sockaddr_in6);
 break;
#endif
 default:
 err_quit("sctp_print_addresses: unknown AF");
 break;
 }
#endif
 ss = (struct sockaddr_storage *)((char *)ss + salen);
 }
}
```

## 23.8 给定IP地址找出关联ID

- 从IP地址到关联ID的转换

- ```
#include      "unp.h"

sctp_assoc_t
sctp_address_to_associd(int sock_fd, struct sockaddr *sa, socklen_t salen)
{
    struct sctp_paddrparams sp;
    int siz;
```

```

    siz = sizeof(struct sctp_paddrparams);
    bzero(&sp,siz);
    memcpy(&sp.spp_address,sa,salen);
    sctp_opt_info(sock_fd,0,
                  SCTP_PEER_ADDR_PARAMS, &sp, &siz);
    return(sp.spp_assoc_id);
}

```

23.9 心博和地址不可达

- SCTP提供类似TCP的保持存活选项的心搏机制，出错门限是认定这个对端地址不可达之前必须发生的心搏遗失即超时重传次数，由心搏检测到该对端地址再次变为可达时，该地址重新开始活跃
- 心搏控制实用函数

```

○ #include      "unp.h"

int heartbeat_action(int sock_fd, struct sockaddr *sa, socklen_t salen,
                     u_int value)
{
    struct sctp_paddrparams sp;
    int siz;

    bzero(&sp,sizeof(sp));
    sp.spp_hbinterval = value;
    memcpy((caddr_t)&sp.spp_address,sa,salen);
    Setsockopt(sock_fd,IPPROTO_SCTP,
               SCTP_PEER_ADDR_PARAMS, &sp, sizeof(sp));
    return(0);
}

```

23.10 关联剥离

- 一到多式接口相比一到一式接口存在的优势
 - 只需维护单个描述符。
 - 允许编写简单的迭代服务器程序。
 - 允许应用进程在四路握手的第三个和第四个分组发送数据，只需使用sendmsg或sctp_sendmsg隐式建立关联就行。
 - 无需跟踪传输状态。也就是说应用进程只需在套接字描述符上执行一个接收调用就可以接收消息，之前不必执行传统的connect或accept调用。
- 一到多式存在的缺陷：造成难以编写并发服务器程序
- 增设sctp_peeloff函数，该函数取一个一到多式套接字描述符和一个关联ID，返回一个新的仅仅附以给定关联的一到一式套接字描述符，原始的一到多式套接字继续开放，它代表的其他关联均不受此影响
- 一个并发SCTP服务器程序

```

○ #include      "unp.h"

int
main(int argc, char **argv)
{

```

```

int sock_fd,msg_flags,connfd,childpid;
sctp_assoc_t assoc;
char readbuf[BUFFSIZE];
struct sockaddr_in servaddr, cliaddr;
struct sctp_sndrcvinfo sri;
struct sctp_event_subscribe evnts;
socklen_t len;
size_t rd_sz;

sock_fd = Socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

Bind(sock_fd, (SA *) &servaddr, sizeof(servaddr));

bzero(&evnts, sizeof(evnts));
evnts.sctp_data_io_event = 1;
Setsockopt(sock_fd, IPPROTO_SCTP, SCTP_EVENTS,
           &evnts, sizeof(evnts));

Listen(sock_fd, LISTENQ);
/* include mod_servfork */
for ( ; ; ) {
    len = sizeof(struct sockaddr_in);
    rd_sz = Sctp_recvmsg(sock_fd, readbuf, sizeof(readbuf),
                          (SA *)&cliaddr, &len,
                          &sri,&msg_flags);
    Sctp_sendmsg(sock_fd, readbuf, rd_sz,
                  (SA *)&cliaddr, len,
                  sri.sinfo_ppid,
                  sri.sinfo_flags,
                  sri.sinfo_stream,
                  0, 0);
    assoc = sctp_address_to_associd(sock_fd,(SA *)&cliaddr,len);
    if((int)assoc == 0){
        err_ret("Can't get association id");
        continue;
    }
    connfd = sctp_peeloff(sock_fd,assoc);
    if(connfd == -1){
        err_ret("sctp_peeloff fails");
        continue;
    }
    if((childpid = fork()) == 0) {
        Close(sock_fd);
        str_echo(connfd);
        exit(0);
    } else {
        Close(connfd);
    }
}

```

```

/* end mod_servfork */
}

```

23.11 定时控制

- 一些定时控制量，它们影响SCTP端点需多久才能声称某个关联或某个对端地址已经失效

字 段	说 明	默 认 值	单 位
srto_min	最小重传超时	1000	毫秒
srto_max	最大重传超时	60000	毫秒
srto_initial	初始重传超时	3000	毫秒
sinit_max_init_timeo	INIT阶段最大重传超时	60000	毫秒
sinit_max_attempts	INIT的最大重传次数	8	次数
app_pathmaxrxt	每个地址的最大重传次数	5	次数
sasoc_asocmaxrxt	每个关联的最大重传次数	10	次数

图23-16 SCTP中控制定时的字段

23.12 何时改用SCTP代替TCP

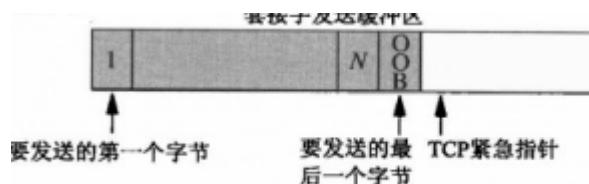
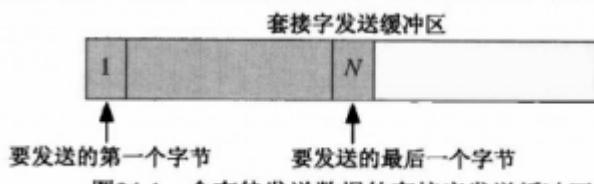
- SCTP的益处
 - 直接支持多宿，一个端点可以利用它的多个直接连接的网络获得额外的可靠性
 - 可以消除头端阻塞，应用进程可以使用单个SCTP关联并行地传输多个数据元素
 - 保持应用层消息边界
 - 提供无序消息服务
 - 提供部分可靠服务，允许SCTP发送端为每个消息指定一个生命期
 - 提供TCP的许多特性：正面确认，重传丢失数据、重排数据、窗口式流量控制、慢启动、拥塞避免、选择性确认
- SCTP不提供TCP的半关闭状态
- SCTP不提供TCP的紧急数据
- 不能从SCTP真正获益的是那些确实必须使用面向字节流传输服务的应用，如telnet，TCP能够比SCTP更高效地把字节流分割分装到TCP分节中
- SCTP忠实地保持消息边界，当每个消息的长度仅仅是一个字节时mSCTP封装消息到数据块的效率很低

第24章 带外数据

- 带外数据：一个连接的某端发生了重要的事情，而且该端希望迅速通告其对端，带外数据被认为具有比普通数据更高的优先级
- UDP没有实现带外数据

24.2 TCP带外数据

- TCP并没有真正的带外数据，不过提供了紧急模式
-



- TCP紧急指针对应一个TCP序列号，它是使用MSG_OOB标志写出的最后一个数据字节对应的序列号加1
- 即便数据的流动会因为TCP的流量控制而停止，紧急通知却总是无障碍地发送到对端
- 只有一个OOB标记，如果新的OOB字节在旧的OOB字节被读取之前就到达，旧的被丢弃
- 如果接收进程开启了SO_OOBINLINE套接字选项，那么由TCP紧急指针指向的实际数据字节将被留在通常的套接字接收缓冲区中，否则被放入一个独立的单字节带外缓冲区
- 当有新的紧急指针到达时，内核给接收套接字的属主进程发送SIGURG信号
 - (1) 当收到一个设置了URG标志的分节时，接收端TCP检查紧急指针，确定它是否指向新的带外数据，也就是判断本分节是不是首个到达的引用从发送端到接收端的数据流中特定字节的紧急模式分节。发送端TCP往往发送多个含有URG标志且紧急指针指向同一个数据字节的分节（通常是在一小段时间内）。这些分节中只有第一个到达的会导致通知接收进程有新的带外数据到达。
 - (2) 当有新的紧急指针到达时，接收进程被通知到。首先，内核给接收套接字的属主进程发送SIGURG信号，前提是接收进程（或其他进程）曾调用fcntl或ioctl为这个套接字建立了属主（图7-20），而且该属主进程已为这个信号建立了信号处理函数。其次，如果接收进程阻塞在select调用中以等待这个套接字描述符出现一个异常条件，select调用就返回。
- 一旦有新的紧急指针到达，不论由紧急指针指向的实际数据字节是否已经到达接收端TCP，这两个潜在通知接收进程的手段就发生动作。
- 只有一个OOB标记，如果新的OOB字节在旧的OOB字节被读取之前就到达，旧的OOB字节会被丢弃。
- (3) 当由紧急指针指向的实际数据字节到达接收端TCP时，该数据字节既可能被拉出带外，也可能被留在带内，即在线（inline）留存。SO_OOBINLINE套接字选项默认情况下是禁止的，对于这样的接收端套接字，该数据字节并不放入套接字接收缓冲区，而是被放入该连接的一个

- 简单的带外发送程序

```

○ #include      "unp.h"

int
main(int argc, char **argv)
{
    int          sockfd;

    if (argc != 3)
        err_quit("usage: tcpsend01 <host> <port#>");

    sockfd = Tcp_connect(argv[1], argv[2]);
  
```

```

        Write(sockfd, "123", 3);
        printf("wrote 3 bytes of normal data\n");
        sleep(1);

        Send(sockfd, "4", 1, MSG_OOB);
        printf("wrote 1 byte of OOB data\n");
        sleep(1);

        Write(sockfd, "56", 2);
        printf("wrote 2 bytes of normal data\n");
        sleep(1);

        Send(sockfd, "7", 1, MSG_OOB);
        printf("wrote 1 byte of OOB data\n");
        sleep(1);

        Write(sockfd, "89", 2);
        printf("wrote 2 bytes of normal data\n");
        sleep(1);

        exit(0);
    }
}

```

- 简单的带外接收程序

```

o #include      "unp.h"

int          listenfd, connfd;

void    sig_urg(int);

int
main(int argc, char **argv)
{
    int          n;
    char    buff[100];

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], NULL);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], NULL);
    else
        err_quit("usage: tcprecv01 [ <host> ] <port#>");

    connfd = Accept(listenfd, NULL, NULL);

    Signal(SIGURG, sig_urg);
    Fcntl(connfd, F_SETOWN, getpid());

    for ( ; ; ) {
        if ( (n = Read(connfd, buff, sizeof(buff)-1)) == 0) {
            printf("received EOF\n");
            exit(0);
        }
    }
}

```

```

        }
        buff[n] = 0;      /* null terminate */
        printf("read %d bytes: %s\n", n, buff);
    }

void
sig_urg(int signo)
{
    int          n;
    char     buff[100];

    printf("SIGURG received\n");
    n = Recv(connfd, buff, sizeof(buff)-1, MSG_OOB);
    buff[n] = 0;          /* null terminate */
    printf("read %d OOB byte: %s\n", n, buff);
}

```

- 发送进程带外数据的每次产生递交给接收进程的SIGURG信号，后者接着读入单个带外字节
- 改用select代替SIGURG信号重新编写带外接收程序

```

◦ #include      "unp.h"

int
main(int argc, char **argv)
{
    int          listenfd, connfd, n, justreadoob = 0;
    char     buff[100];
    fd_set  rset, xset;

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], NULL);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], NULL);
    else
        err_quit("usage: tcprecv03 [ <host> ] <port#>");

    connfd = Accept(listenfd, NULL, NULL);

    FD_ZERO(&rset);
    FD_ZERO(&xset);
    for ( ; ; ) {
        FD_SET(connfd, &rset);
        if (justreadoob == 0)
            FD_SET(connfd, &xset);

        Select(connfd + 1, &rset, NULL, &xset, NULL);

        if (FD_ISSET(connfd, &xset)) {
            n = Recv(connfd, buff, sizeof(buff)-1, MSG_OOB);
            buff[n] = 0;          /* null terminate */
            printf("read %d OOB byte: %s\n", n, buff);
            justreadoob = 1;
        }
    }
}

```

```

        FD_CLR(connfd, &xset);
    }

    if (FD_ISSET(connfd, &rset)) {
        if ( (n = Read(connfd, buff, sizeof(buff)-1)) == 0) {
            printf("received EOF\n");
            exit(0);
        }
        buff[n] = 0;      /* null terminate */
        printf("read %d bytes: %s\n", n, buff);
        justreadoob = 0;
    }
}
}

```

24.3 sockatmark函数

- 每收到一个带外数据时，就有一个与之关联的带外标记，这是发送进程发送带外字节时该字节在发送端普通数据流中的位置，在从套接字读入期间，接收进程通过调用sockatmark函数确定是否处于带外标记
- 带外标记的常见用法之一是接收进程特殊地对待所有数据，直到越过它
 - (1) 带外标记总是指向普通数据最后一个字节紧后的位置。这意味着，如果带外数据在线接收，那么如果下一个待读入的字节是使用MSG_OOB标志发送的，sockatmark就返回真。而如果SO_OOBINLINE套接字选项没有开启，那么，若下一个待读入的字节是跟在带外数据后发送的第一个字节，sockatmark就返回真。
 - (2) 读操作总是停在带外标记上（TCPv2第519~520页）。也就是说，如果在套接字接收缓冲区中有100个字节，不过在带外标记之前只有5个字节，而进程执行一个请求100个字节的read调用，那么返回的是带外标记之前的5个字节。这种在带外标记上强制停止读操作的做法使得进程能够调用sockatmark确定缓冲区指针是否处于带外标记。
- 调用sockatmark的接收程序

```

○ #include      "unp.h"

int
main(int argc, char **argv)
{
    int          listenfd, connfd, n, on=1;
    char        buff[100];

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], NULL);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], NULL);
    else
        err_quit("usage: tcprecv04 [ <host> ] <port#>");

    Setsockopt(listenfd, SOL_SOCKET, SO_OOBINLINE, &on, sizeof(on));

    connfd = Accept(listenfd, NULL, NULL);
    sleep(5);

    for ( ; ; ) {

```

```

        if (Sockatmark(connfd))
            printf("at OOB mark\n");

        if ( (n = Read(connfd, buff, sizeof(buff)-1)) == 0) {
            printf("received EOF\n");
            exit(0);
        }
        buff[n] = 0; /* null terminate */
        printf("read %d bytes: %s\n", n, buff);
    }
}

```

- 带外数据的另外两个特性
 - 即使因为流量控制而停止发送数据了，TCP仍然发送带外数据的通知(即它的紧急指针)
 - 在带外数据到达之前，接收进程可能被通知说发送进程已经发送了带外数据(使用SIGURG或select),如果接收进程接着指定MSG_OOB调用recv,而带外数据却未到达，将返回错误。解决办法是让接收进程通过读入已排队的普通数据，在套接字缓冲区中腾出空间，这将导致接收端TCP向发送端通告一个非0的窗口，最终允许发送端发送带外数据
- 一个给定TCP连接只有一个带外标记，如果在接收进程读入某个现有带外数据之前有新的带外数据到达，先前的标记就丢失

24.4 TCP带外数据小结

- 对于TCP的紧急模式，可以认为URG标志是通知，紧急指针是带外标记，数据字节是其本身
 - (1) 发送端进入紧急模式这个事实。接收进程得以通知这个事实的手段不外乎SIGURG信号或select调用。本通知在发送进程发送带外字节后由发送端TCP立即发送，因为我们在图24-11中看到，即使往接收端的任何数据发送因流量控制而停止了，TCP仍然发送本通知。本通知可能导致接收端进入某种特殊处理模式，以处理接收的任何后继数据。
 - (2) 带外字节的位置，也就是它相对于来自发送端的其余数据的发送位置：带外标记。
 - (3) 带外字节的实际值。既然TCP是一个不解释应用进程所发送数据的字节流协议，带外字节就可以是任何8位值。
- 每个连接只有一个TCP紧急指针，每个连接只有一个带外标记，每个连接只有一个单字节的带外缓冲区

24.5 客户/服务器心搏函数

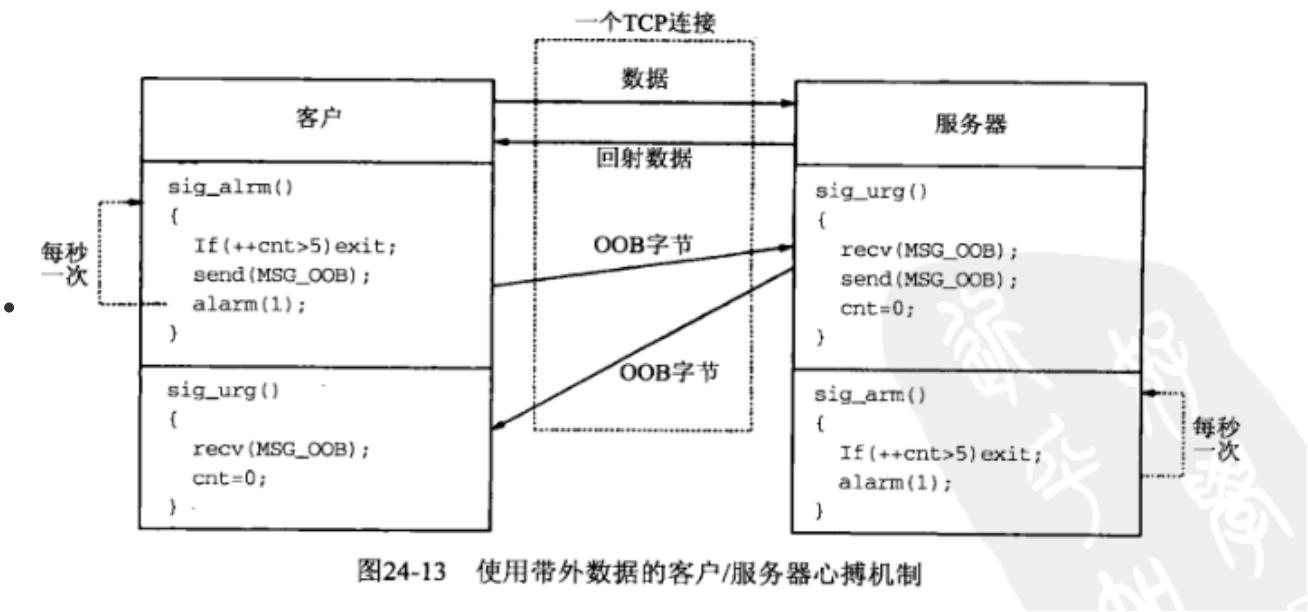


图24-13 使用带外数据的客户/服务器心搏机制

第25章 信号驱动式I/O

- 进程预先告知内核，使得当某个描述符上发生某事时，内核使用信号通知相关进程
- 信号驱动式I/O和异步I/O的差异

25.2 套接字的信号驱动式I/O

- 3个步骤
 - 建立SIGIO信号的信号处理函数
 - 设置该套接字的属主，使用fcntl的F_SETOWN命令
 - 开启该套接字的信号驱动式I/O, 使用fcntl的F_SETFL命令打开O_ASYNC标志
- 对于UDP套接字的SIGIO信号
 - SIGIO信号在以下事件发生
 - 数据报到达套接字
 - 套接字上发生异步错误
- 对于TCP套接字的SIGIO信号
 - 下列条件
 - 监听套接字上某个连接请求已经完成;
 - 某个断连请求已经发起;
 - 某个断连请求已经完成;
 - • 某个连接之半已经关闭;
 - 数据到达套接字;
 - 数据已经从套接字发送走（即输出缓冲区有空闲空间）;
 - 发生某个异步错误。
- 构建一个UDP服务器的两种方式

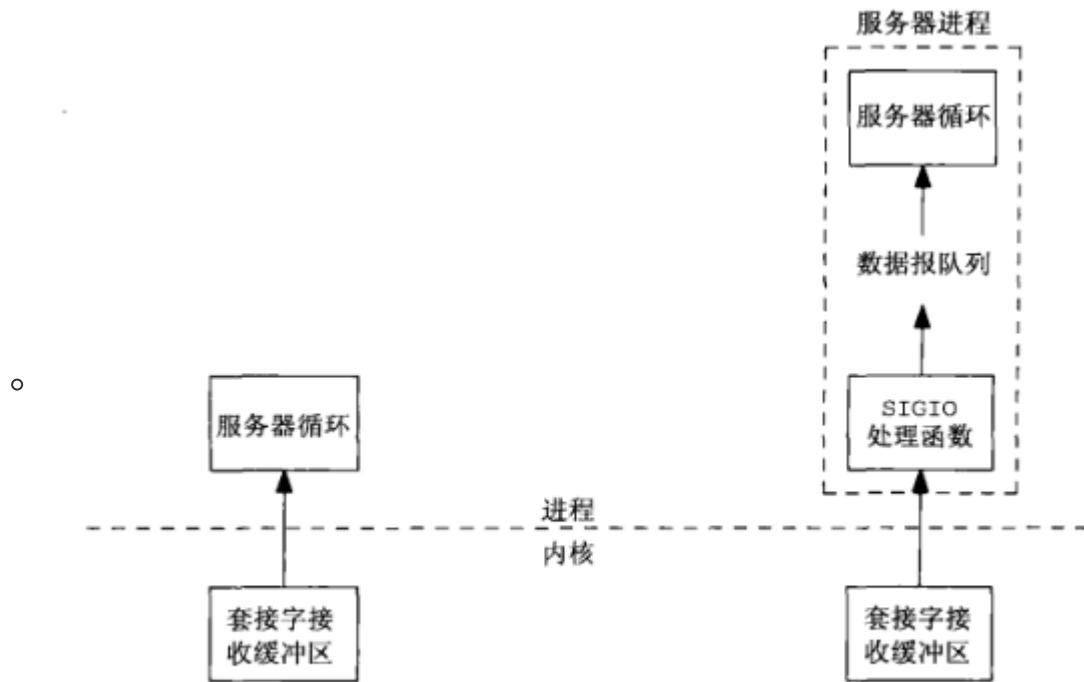


图25-1 构建一个UDP服务器的两种方式

25.3 使用SIGIO的UDP回射服务器程序

```

/* include dgecho1 */
#include      "unp.h"

static int          sockfd;

#define QSIZE     8           /* size of input queue */
#define MAXDG    4096         /* max datagram size */

typedef struct {
    void            *dg_data;           /* ptr to actual datagram */
    size_t          dg_len;            /* length of datagram */
    struct sockaddr *dg_sa;           /* ptr to sockaddr{} w/client's address */
    socklen_t        dg_salen;          /* length of sockaddr{} */
} DG;
static DG        dg[QSIZE];           /* queue of datagrams to process */
static long      cntread[QSIZE+1];      /* diagnostic counter */

static int        iget;               /* next one for main loop to process */
static int        iput;               /* next one for signal handler to read into */
static int        nqueue;             /* # on queue for main loop to process */
static socklen_t  clilen;             /* max length of sockaddr{} */

static void       sig_io(int);
static void       sig_hup(int);
/* end dgecho1 */

/* include dgecho2 */

```

```

void
dg_echo(int sockfd_arg, SA *pcliaddr, socklen_t clilen_arg)
{
    int                  i;
    const int            on = 1;
    sigset_t             zeromask, newmask, oldmask;

    sockfd = sockfd_arg;
    clilen = clilen_arg;

    for (i = 0; i < QSIZE; i++) { /* init queue of buffers */
        dg[i].dg_data = Malloc(MAXDG);
        dg[i].dg_sa = Malloc(clilen);
        dg[i].dg_salen = clilen;
    }
    igure = igure = nqueue = 0;

    Signal(SIGHUP, sig_hup);
    Signal(SIGIO, sig_io);
    Fcntl(sockfd, F_SETOWN, getpid());
    Ioctl(sockfd, FIOASYNC, &on);
    Ioctl(sockfd, FIONBIO, &on);

    Sigemptyset(&zeromask);           /* init three signal sets */
    Sigemptyset(&oldmask);
    Sigemptyset(&newmask);
    Sigaddset(&newmask, SIGIO);      /* signal we want to block */

    Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
    for ( ; ; ) {
        while (nqueue == 0)
            sigsuspend(&zeromask); /* wait for datagram to process */

        /* 4unblock SIGIO */
        Sigprocmask(SIG_SETMASK, &oldmask, NULL);

        Sendto(sockfd, dg[igure].dg_data, dg[igure].dg_len, 0,
                dg[igure].dg_sa, dg[igure].dg_salen);

        if (++igure >= QSIZE)
            igure = 0;

        /* 4block SIGIO */
        Sigprocmask(SIG_BLOCK, &newmask, &oldmask);
        nqueue--;
    }
}
/* end dgecho2 */

/* include sig_io */
static void
sig_io(int signo)
{

```

```

    ssize_t          len;
    int             nread;
    DG             *ptr;

    for (nread = 0; ; ) {
        if (nqueue >= QSIZE)
            err_quit("receive overflow");

        ptr = &dg[input];
        ptr->dg_salen = clilen;
        len = recvfrom(sockfd, ptr->dg_data, MAXDG, 0,
                        ptr->dg_sa, &ptr->dg_salen);
        if (len < 0) {
            if (errno == EWOULDBLOCK)
                break;           /* all done; no more queued to read */
            else
                err_sys("recvfrom error");
        }
        ptr->dg_len = len;

        nread++;
        nqueue++;
        if (++input >= QSIZE)
            input = 0;

    }
    cntread[nread]++;           /* histogram of # datagrams read per signal */
}

/* end sig_io */

/* include sig_hup */
static void
sig_hup(int signo)
{
    int          i;

    for (i = 0; i <= QSIZE; i++)
        printf("cntread[%d] = %ld\n", i, cntread[i]);
}
/* end sig_hup */

```

第26章 线程

- fork调用存在的问题
 - fork是昂贵的
 - fork返回之后父子进程之间信息的传递需要进程间通信(IPC)
- 同一进程内的所有线程共享相同的全局内存，存在同步问题
-

同一进程内的所有线程除了共享全局变量外还共享：

- 进程指令；
- 大多数数据；
- 打开的文件（即描述符）；
- 信号处理函数和信号处置；
- 当前工作目录；
- 用户ID和组ID。

不过每个线程有各自的：

- 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。

26.2 基本线程函数：创建和终止

- pthread_create
- pthread_join
- pthread_self
- pthread_detach
 - 一个线程或者是可汇合的，或者是脱离的。当一个可汇合的线程终止时，它的线程ID和退出状态将留存到另一个线程对它调用pthread_join。脱离的线程像守护进程，当它们终止时，所有相关资源都被释放，我们不能等待它们终止
- pthread_exit

26.3 使用线程的str_cli函数

```
#include      "unpthread.h"

void      *copyto(void *);

static int      sockfd;          /* global for both threads to access */
static FILE     *fp;

void
str_cli(FILE *fp_arg, int sockfd_arg)
{
    char      recvline[MAXLINE];
    pthread_t      tid;

    sockfd = sockfd_arg;      /* copy arguments to externals */
    fp = fp_arg;

    Pthread_create(&tid, NULL, copyto, NULL);
```

```

        while (Readline(sockfd, recvline, MAXLINE) > 0)
            Fputs(recvline, stdout);
    }

void *
copyto(void *arg)
{
    char    sendline[MAXLINE];

    while (Fgets(sendline, MAXLINE, fp) != NULL)
        Writen(sockfd, sendline, strlen(sendline));

    Shutdown(sockfd, SHUT_WR);      /* EOF on stdin, send FIN */

    return(NULL);
    /* 4return (i.e., thread terminates) when EOF on stdin */
}

```

26.4 使用线程的TCP回射服务器程序

- 创建新线程并不影响已打开描述符的引用计数
- 使用线程的TCP回射服务器程序

```

o #include      "unpthread.h"

static void      *doit(void *);           /* each thread executes this function */

int
main(int argc, char **argv)
{
    int                  listenfd, connfd;
    pthread_t             tid;
    socklen_t              addrlen, len;
    struct sockaddr *cliaddr;

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
    else
        err_quit("usage: tcperv01 [ <host> ] <service or port>");

    cliaddr = Malloc(addrlen);

    for ( ; ; ) {
        len = addrlen;
        connfd = Accept(listenfd, cliaddr, &len);
        Pthread_create(&tid, NULL, &doit, (void *) connfd);
    }
}

static void *

```

```

doit(void *arg)
{
    Pthread_detach(pthread_self());
    str_echo((int) arg);      /* same function as before */
    Close((int) arg);         /* done with connected socket */
    return(NULL);
}

```

- 给新线程传递参数

- malloc和free函数是不可重入的，在主线程正处于这两个函数之一的内部处理期间，从某个信号处理函数中调用这两个函数之一有可能导致灾难性的后果，因为这两个函数操纵相同的静态数据结构

```

#include      "unpthread.h"

static void      *doit(void *);           /* each thread executes this function */

int
main(int argc, char **argv)
{
    int                     listenfd, *iptr;
    thread_t                tid;
    socklen_t                addrlen, len;
    struct sockaddr *cliaddr;

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
    else
        err_quit("usage: tcpserv01 [ <host> ] <service or port>");

    cliaddr = Malloc(addrlen);

    for ( ; ; ) {
        len = addrlen;
        iptr = Malloc(sizeof(int));
        *iptr = Accept(listenfd, cliaddr, &len);
        Pthread_create(&tid, NULL, &doit, iptr);
    }
}

static void *
doit(void *arg)
{
    int                     connfd;

    connfd = *((int *) arg);
    free(arg);

    Pthread_detach(pthread_self());
    str_echo(connfd);          /* same function as before */
    Close(connfd);            /* done with connected socket */
}

```

```

        return(NULL);
}

```

- 线程安全函数

不必线程安全的版本	必须线程安全的版本	注 释
asctime ctime getc_unlocked getchar_unlocked getgrid getgrnam getlogin getpwnam getpwuid gmtime localtime putc_unlocked putchar_unlocked rand readdir strtok ttyname	asctime_r ctermid ctime_r getgrid_r getgrnam_r getlogin_r getpwnam_r getpwuid_r gmtime_r localtime_r rand_r readdir_r strtok_r tmpnam ttyname_r	仅当参数非空时才是线程安全的
gethostXXX getnetXXX getprotoXXX getservXXX inet_ntoa		

图26-5 线程安全函数

26.5 线程特定数据

- 使用线程特定数据
- 改变调用顺序
- 改变接口的结构，避免使用静态变量
-

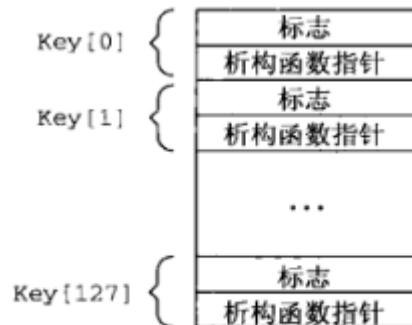


图26-7 线程特定数据的可能实现

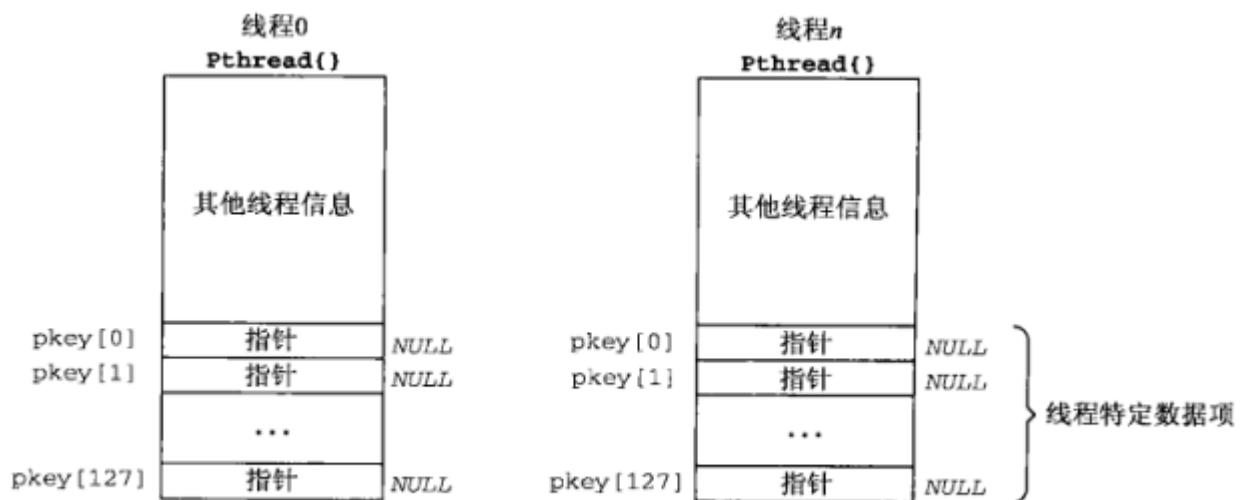


图26-8 系统维护的关于每个线程的信息

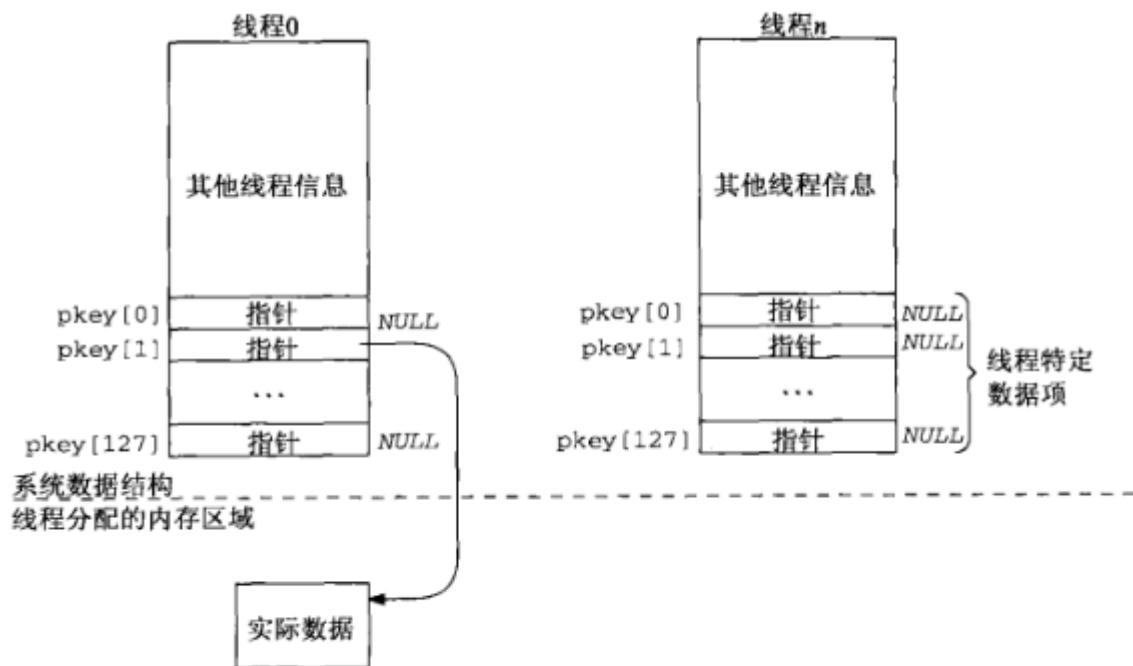


图26-9 把malloc到的内存区和线程特定数据指针相关联

- pthread_setspecific只是在pthread结构中把对应指定键的指针设置为指向分配的内存区，pthread_getspecific所做的只是返回对应指定键的指针

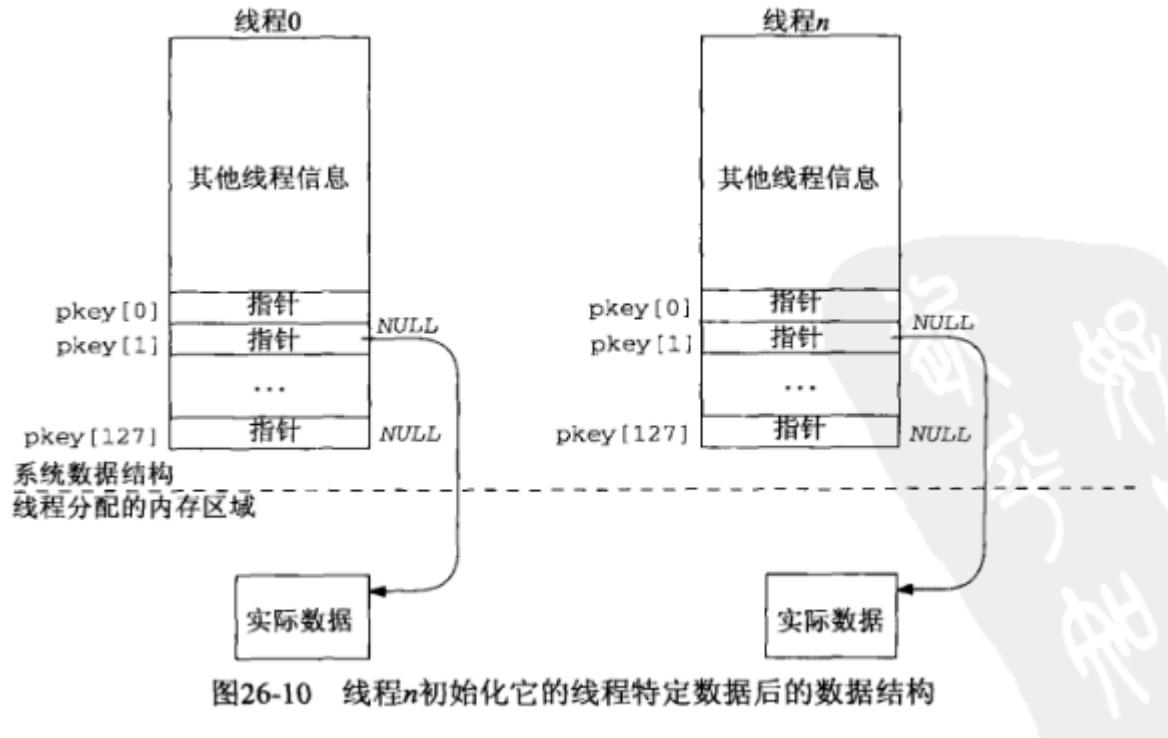


图26-10 线程n初始化它的线程特定数据后的数据结构

- `pthread_once`
- `pthread_key_create`
- 这两个函数的典型用法如下所示（不考虑出错返回）。

```

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 values pointed to by ptr */
}

```

- 线程安全的readline函数

```

o /* include readline1 */
#include      "unpthread.h"

static pthread_key_t    rl_key;
static pthread_once_t   rl_once = PTHREAD_ONCE_INIT;

static void
readline_destructor(void *ptr)
{
    free(ptr);
}

static void
readline_once(void)
{
    Pthread_key_create(&rl_key, readline_destructor);
}

typedef struct {
    int      rl_cnt;                      /* initialize to 0 */
    char   *rl_bufptr;                    /* initialize to rl_buf */
    char    rl_buf[MAXLINE];
} Rline;
/* end readline1 */

/* include readline2 */
static ssize_t
my_read(Rline *tsd, int fd, char *ptr)
{
    if (tsd->rl_cnt <= 0) {
again:
    if ( (tsd->rl_cnt = read(fd, tsd->rl_buf, MAXLINE)) < 0) {
        if (errno == EINTR)
            goto again;
        return(-1);
    } else if (tsd->rl_cnt == 0)
        return(0);
    tsd->rl_bufptr = tsd->rl_buf;
}
    tsd->rl_cnt--;
    *ptr = *tsd->rl_bufptr++;
    return(1);
}

ssize_t
readline(int fd, void *vptr, size_t maxlen)
{
    size_t          n, rc;
    char   c, *ptr;
    Rline   *tsd;

    Pthread_once(&rl_once, readline_once);

```

```

        if ( (tsd = pthread_getspecific(rl_key)) == NULL) {
            tsd = Calloc(1, sizeof(Rline));           /* init to 0 */
            Pthread_setspecific(rl_key, tsd);
        }

        ptr = vptr;
        for (n = 1; n < maxlen; n++) {
            if ( (rc = my_read(tsd, fd, &c)) == 1) {
                *ptr++ = c;
                if (c == '\n')
                    break;
            } else if (rc == 0) {
                *ptr = 0;
                return(n - 1);                      /* EOF, n - 1 bytes read */
            } else
                return(-1);                         /* error, errno set by read() */
        }

        *ptr = 0;
        return(n);
    }
/* end readline2 */

ssize_t
Readline(int fd, void *ptr, size_t maxlen)
{
    ssize_t          n;

    if ( (n = readline(fd, ptr, maxlen)) < 0)
        err_sys("readline error");
    return(n);
}

```

26.6 Web客户与同时连接

```

/* include web1 */
#include      "unpthread.h"
#include      <thread.h>           /* Solaris threads */

#define MAXFILES      20
#define SERV          "80"      /* port number or service name */

struct file {
    char  *f_name;                  /* filename */
    char  *f_host;                 /* hostname or IP address */
    int   f_fd;                    /* descriptor */
    int   f_flags;                 /* F_xxx below */
    pthread_t   f_tid;              /* thread ID */
} file[MAXFILES];
#define F_CONNECTING  1      /* connect() in progress */
#define F_READING     2      /* connect() complete; now reading */
#define F_DONE         4      /* all done */

```

```

#define GET_CMD           "GET %s HTTP/1.0\r\n\r\n"

int          nconn, nfiles, nlefttoconn, nlefttoread;

void      *do_get_read(void *);
void    home_page(const char *, const char *);
void    write_get_cmd(struct file *);

int
main(int argc, char **argv)
{
    int                  i, n, maxnconn;
    pthread_t      tid;
    struct file    *fptr;

    if (argc < 5)
        err_quit("usage: web <#conns> <IPaddr> <homepage> file1 ...");
    maxnconn = atoi(argv[1]);

    nfiles = min(argc - 4, MAXFILES);
    for (i = 0; i < nfiles; i++) {
        file[i].f_name = argv[i + 4];
        file[i].f_host = argv[2];
        file[i].f_flags = 0;
    }
    printf("nfiles = %d\n", nfiles);

    home_page(argv[2], argv[3]);

    nlefttoread = nlefttoconn = nfiles;
    nconn = 0;
/* end web1 */
/* include web2 */
    while (nlefttoread > 0) {
        while (nconn < maxnconn && nlefttoconn > 0) {
            /* 4find a file to read */
            for (i = 0 ; i < nfiles; i++)
                if (file[i].f_flags == 0)
                    break;
            if (i == nfiles)
                err_quit("nlefttoconn = %d but nothing found", nlefttoconn);

            file[i].f_flags = F_CONNECTING;
            Pthread_create(&tid, NULL, &do_get_read, &file[i]);
            file[i].f_tid = tid;
            nconn++;
            nlefttoconn--;
        }

        if ( (n = thr_join(0, &tid, (void **) &fptr)) != 0)
            errno = n, err_sys("thr_join error");
    }
}

```

```

        nconn--;
        nlefttoread--;
        printf("thread id %d for %s done\n", tid, fptr->f_name);
    }

    exit(0);
}
/* end web2 */

/* include do_get_read */
void *
do_get_read(void *vptr)
{
    int                     fd, n;
    char                    line[MAXLINE];
    struct file             *fptr;

    fptr = (struct file *) vptr;

    fd = Tcp_connect(fptr->f_host, SERV);
    fptr->f_fd = fd;
    printf("do_get_read for %s, fd %d, thread %d\n",
           fptr->f_name, fd, fptr->f_tid);

    write_get_cmd(fptr);      /* write() the GET command */

    /* 4Read server's reply */
    for ( ; ; ) {
        if ( (n = Read(fd, line, MAXLINE)) == 0)
            break;          /* server closed connection */

        printf("read %d bytes from %s\n", n, fptr->f_name);
    }
    printf("end-of-file on %s\n", fptr->f_name);
    Close(fd);
    fptr->f_flags = F_DONE;      /* clears F_READING */

    return(fptr);              /* terminate thread */
}
/* end do_get_read */

/* include write_get_cmd */
void
write_get_cmd(struct file *fptr)
{
    int         n;
    char     line[MAXLINE];

    n = snprintf(line, sizeof(line), GET_CMD, fptr->f_name);
    Writen(fptr->f_fd, line, n);
    printf("wrote %d bytes for %s\n", n, fptr->f_name);

    fptr->f_flags = F_READING;           /* clears F_CONNECTING */
}

```

```

}

/* end write_get_cmd */

/* include home_page */
void
home_page(const char *host, const char *fname)
{
    int          fd, n;
    char        line[MAXLINE];

    fd = Tcp_connect(host, SERV); /* blocking connect() */

    n = snprintf(line, sizeof(line), GET_CMD, fname);
    Writen(fd, line, n);

    for ( ; ; ) {
        if ( (n = Read(fd, line, MAXLINE)) == 0)
            break;           /* server closed connection */

        printf("read %d bytes of home page\n", n);
        /* do whatever with data */
    }
    printf("end-of-file on home page\n");
    Close(fd);
}
/* end home_page */

```

26.7 互斥锁

- 使用一个互斥锁保护共享变量，访问该变量的前提条件是持有该互斥锁
- pthread_mutex_lock
- pthread_mutex_unlock

26.8 条件变量

- 需要一个让主循环进入睡眠，直到某个线程通知它有事可做才醒来的办法。条件变量结合互斥锁能够处理这个功能，互斥锁提供互斥机制，条件变量提供信号机制
- pthread_cond_wait 把调用线程投入睡眠并释放调用线程持有的互斥锁，当调用线程后来从这个函数返回时，该线程再次持有该互斥锁
- pthread_cond_signal
- pthread_cond_broadcast
- pthread_cond_timewait

26.9 Web客户与同时连接

```

#include      "unpthread.h"
#include      <thread.h>           /* Solaris threads */

#define MAXFILES      20
#define SERV          "80"     /* port number or service name */

```

```

struct file {
    char *f_name;                      /* filename */
    char *f_host;                      /* hostname or IP address */
    int f_fd;                          /* descriptor */
    int f_flags;                       /* F_xxx below */
    pthread_t f_tid;                  /* thread ID */
} file[MAXFILES];
#define F_CONNECTING     1      /* connect() in progress */
#define F_READING        2      /* connect() complete; now reading */
#define F_DONE            4      /* all done */
#define F_JOINED          8      /* main has pthread_join'ed */

#define GET_CMD           "GET %s HTTP/1.0\r\n\r\n"

int             nconn, nfiles, nlefttoconn, nlefttoread;

int             ndone;                /* number of terminated threads */
pthread_mutex_t ndone_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  ndone_cond = PTHREAD_COND_INITIALIZER;

void *do_get_read(void *);
void home_page(const char *, const char *);
void write_get_cmd(struct file *);

int
main(int argc, char **argv)
{
    int             i, maxnconn;
    pthread_t       tid;
    struct file    *fptr;

    if (argc < 5)
        err_quit("usage: web <#conns> <IPaddr> <homepage> file1 ...");
    maxnconn = atoi(argv[1]);

    nfiles = min(argc - 4, MAXFILES);
    for (i = 0; i < nfiles; i++) {
        file[i].f_name = argv[i + 4];
        file[i].f_host = argv[2];
        file[i].f_flags = 0;
    }
    printf("nfiles = %d\n", nfiles);

    home_page(argv[2], argv[3]);

    nlefttoread = nlefttoconn = nfiles;
    nconn = 0;
/* include web2 */
    while (nlefttoread > 0) {
        while (nconn < maxnconn && nlefttoconn > 0) {
            /* 4find a file to read */
            for (i = 0 ; i < nfiles; i++)

```

```

        if (file[i].f_flags == 0)
            break;
        if (i == nfiles)
            err_quit("nlefttoconn = %d but nothing found", nlefttoconn);

        file[i].f_flags = F_CONNECTING;
        Pthread_create(&tid, NULL, &do_get_read, &file[i]);
        file[i].f_tid = tid;
        nconn++;
        nlefttoconn--;
    }

    /* 4Wait for thread 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);

            if (&file[i] != fptr)
                err_quit("file[i] != fptr");
            fptr->f_flags = F_JOINED;           /* clears F_DONE */
            ndone--;
            nconn--;
            nlefttoread--;
            printf("thread %d for %s done\n", fptr->f_tid, fptr->f_name);
        }
    }
    Pthread_mutex_unlock(&ndone_mutex);
}

exit(0);
}
/* end web2 */

void *
do_get_read(void *vptr)
{
    int                      fd, n;
    char                     line[MAXLINE];
    struct file              *fptr;

    fptr = (struct file *) vptr;

    fd = Tcp_connect(fptr->f_host, SERV);
    fptr->f_fd = fd;
    printf("do_get_read for %s, fd %d, thread %d\n",
           fptr->f_name, fd, fptr->f_tid);

    write_get_cmd(fptr);    /* write() the GET command */
}

```

```

        /* 4Read server's reply */
    for ( ; ; ) {
        if ( (n = Read(fd, line, MAXLINE)) == 0)
            break;           /* server closed connection */

        printf("read %d bytes from %s\n", n, fptr->f_name);
    }
    printf("end-of-file on %s\n", fptr->f_name);
    Close(fd);
    fptr->f_flags = F_DONE;           /* clears F_READING */

    Pthread_mutex_lock(&ndone_mutex);
    ndone++;
    Pthread_cond_signal(&ndone_cond);
    Pthread_mutex_unlock(&ndone_mutex);

    return(fptr);                  /* terminate thread */
}

void
write_get_cmd(struct file *fptr)
{
    int          n;
    char    line[MAXLINE];

    n = snprintf(line, sizeof(line), GET_CMD, fptr->f_name);
    Writen(fptr->f_fd, line, n);
    printf("wrote %d bytes for %s\n", n, fptr->f_name);

    fptr->f_flags = F_READING;           /* clears F_CONNECTING */
}

void
home_page(const char *host, const char *fname)
{
    int          fd, n;
    char    line[MAXLINE];

    fd = Tcp_connect(host, SERV);      /* blocking connect() */

    n = snprintf(line, sizeof(line), GET_CMD, fname);
    Writen(fd, line, n);

    for ( ; ; ) {
        if ( (n = Read(fd, line, MAXLINE)) == 0)
            break;           /* server closed connection */

        printf("read %d bytes of home page\n", n);
        /* do whatever with data */
    }
    printf("end-of-file on home page\n");
    Close(fd);
}

```

第27章 IP选项

27.2 IPv4选项

- IPv4的选项字段跟在20字节IPv4首部固定部分之后
- NOP: 单字节选项
- EOL: 单字节选项
- LSRR
- SSRR
- Timestamp
- Record route

(1) **NOP: no-operation**。单字节选项，典型用途是为某个后续选项落在4字节边界上提供填充。

(2) **EOL: end-of-list**。单字节选项，终止选项的处理。既然各个IP选项的总长度必须为4字节的倍数，因此最后一个有效选项之后可能跟以0~3个EOL字节。

(3) **LSRR: loose source and record route** (TCPv1的8.5节)。我们稍后给出使用本选项的一个例子。

(4) **SSRR: strict source and record route** (TCPv1的8.5节)：我们稍后给出使用本选项的一个例子。

(5) **Timestamp** (TCPv1的7.4节)。

(6) **Record route** (TCPv1的7.3节)。

(7) **Basic security** (已作废)。

(8) **Extended security** (已作废)。

(9) **Stream identifier** (已作废)。

(10) **Router alert**。这是在RFC 2113 [Katz 1997] 中叙述的一种选项。包含该选项的IP数据报要求所有转发路由器都查看其内容。

- 读取和设置IP选项字段使用getsockopt和setsockopt
- 使用setsockopt设置IP选项之后，在相应套接字上发送的所有IP数据报都将包括这些选项
- 可以设置IP选项的套接字包括TCP UDP 和原始IP套接字

27.3 IPv4源路径选项

- 源路径是由IP数据报的发送者指定的一个IP地址列表
- IPv4源路径称为源和记录路径SRR

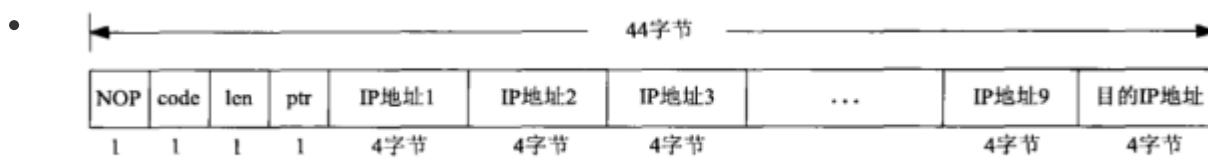


图27-1 向内核传递的源路径

```
/* include inet_srcrt_init */
#include      "unp.h"
#include      <netinet/in_systm.h>
#include      <netinet/ip.h>

static u_char    *optr;           /* pointer into options being formed */
static u_char    *lenptr;         /* pointer to length byte in SRR option */
static int        ocnt;          /* count of # addresses */
```

```

u_char *
inet_srcrt_init(int type)
{
    optr = Malloc(44);           /* NOP, code, len, ptr, up to 10 addresses */
    bzero(optr, 44);            /* guarantees EOLs at end */
    ocnt = 0;
    *optr++ = IPOPT_NOP;        /* NOP for alignment */
    *optr++ = type ? IPOPT_SSRR : IPOPT_LSRR;
    lenptr = optr++;            /* we fill in length later */
    *optr++ = 4;                /* offset to first address */

    return(optr - 4);           /* pointer for setsockopt() */
}
/* end inet_srcrt_init */

/* include inet_srcrt_add */
int
inet_srcrt_add(char *hostptr)
{
    int                                     len;
    struct addrinfo      *ai;
    struct sockaddr_in   *sin;

    if (ocnt > 9)
        err_quit("too many source routes with: %s", hostptr);

    ai = Host_serv(hostptr, NULL, AF_INET, 0);
    sin = (struct sockaddr_in *) ai->ai_addr;
    memcpy(optr, &sin->sin_addr, sizeof(struct in_addr));
    freeaddrinfo(ai);

    optr += sizeof(struct in_addr);
    ocnt++;
    len = 3 + (ocnt * sizeof(struct in_addr));
    *lenptr = len;
    return(len + 1);           /* size for setsockopt() */
}
/* end inet_srcrt_add */

/* include inet_srcrt_print */
void
inet_srcrt_print(u_char *ptr, int len)
{
    u_char          c;
    char           str[INET_ADDRSTRLEN];
    struct in_addr hop1;

    memcpy(&hop1, ptr, sizeof(struct in_addr));
    ptr += sizeof(struct in_addr);

    while ( (c = *ptr++) == IPOPT_NOP)
        ;                      /* skip any leading NOPs */
}

```

```

    if (c == IPOPT_LSRR)
        printf("received LSRR: ");
    else if (c == IPOPT_SSRR)
        printf("received SSRR: ");
    else {
        printf("received option type %d\n", c);
        return;
    }
    printf("%s ", Inet_ntop(AF_INET, &hop1, str, sizeof(str)));

    len = *ptr++ - sizeof(struct in_addr); /* subtract dest IP addr */
    ptr++; /* skip over pointer */
    while (len > 0) {
        printf("%s ", Inet_ntop(AF_INET, ptr, str, sizeof(str)));
        ptr += sizeof(struct in_addr);
        len -= sizeof(struct in_addr);
    }
    printf("\n");
}
/* end inet_srcrt_print */

```

27.4 IPv6扩展首部

- (1) 步跳选项 (hop_by_hop options)。如果有的话步跳选项必须紧跟40字节的IPv6首部。目前没有定义可供应用程序使用的这类选项。
- (2) 目的地选项 (destination options)。目前没有定义可供应用程序使用的这类选项。
- (3) 路径首部 (routing header)。这是一个源路由选项，在概念上类似于我们在27.3节讲解的IPv4源路径选项。
- (4) 分片首部 (fragmentation header)。该首部由对IPv6数据报执行分片的主机自动产生，然后由最终目的主机在重组片段时处理。
- (5) 认证首部 (authentication header, AH)。该首部的用法在RFC 2402 [Kent and Atkinson 1998b] 中说明。
- (6) 安全净荷封装 (encapsulating security payload, ESP)。该首部的用法在RFC 2406 [Kent and Atkinson 1998c] 中说明。

27.5 IPv6步跳选项和目的地选项

27.6 IPv6路由首部

27.7 IPv6粘附选项

第28章 原始套接字

- 有了原始套接字，进程可以读与写ICMPv4 IGMPv4和ICMPv6等分组
- 有了原始套接字，进程可以读写内核不处理其协议字段的IPv4数据报
- 有了原始套接字，进程还可以使用IP_HDRINCL套接字选项自行构造IPv4首部

28.2 原始套接字创建

- sockfd=socket(AF_INET, SOCK_RAW, protocol); 创建一个IPv4原始套接字，只有超级用户才能创建
- 因为原始套接字不存在端口的概念，bind函数仅仅设置本地地址，connect函数仅仅设置外地地址

28.3 原始套接字输出

- 原始套接字的输出遵循以下规则
 - 普通输出通过调用sendto或sendmsg并指定目的IP地址完成。如果套接字已经连接，那么也可以调用write、writev或send。
 - 如果IP_HDRINCL套接字选项未开启，那么由进程让内核发送的数据的起始地址指的是IP首部之后的第一个字节，因为内核将构造IP首部并把它置于来自进程的数据之前。内核把所构造IPv4首部的协议字段设置成来自socket调用的第三个参数。
 - 如果IP_HDRINCL套接字选项已开启，那么由进程让内核发送的数据的起始地址指的是IP首部的第一个字节。进程调用输出函数写出的数据量必须包括IP首部的大小。整个IP首部由进程构造，不过(a) IPv4标识字段可置为0，从而告知内核设置该值，(b) IPv4首部校验和字段总是由内核计算并存储，(c) IPv4选项字段是可选的。
 - 内核会对超出外出接口MTU的原始分组执行分片。
- IPv6的差异
 - 协议首部中的所有字段均采用网络字节序
 - 通过IPv6原始套接字无法读入或写出完整的IPv6分组(包括IPv6首部和任何扩展首部)。IPv6首部的几乎所有字段及所有扩展首部都可以通过套接字选项或辅助数据由应用进程指定或获取。如果应用进程需要读入或写出完整的IPv6数据报，必须使用数据链路访问
- IPv6_CHECKSUM套接字选项

28.4 原始套接字输入

- 大多数ICMP分组在内核处理完其中的ICMP消息后传递到原始套接字
- 所有IGMP分组在内核完成处理其中的IGMP消息后传递到原始套接字
- 内核不认识其协议字段的所有IP数据报传递到原始套接字
 - 当内核有一个需传递到原始套接字的IP数据报时，它将检查所有进程上的所有原始套接字，以寻找所有匹配的套接字。每个匹配的套接字将被递送以该IP数据报的一个副本。内核对每个原始套接字均执行如下3个测试，只有这3个测试结果均为真，内核才把接收到的数据报递送到这个套接字。
 - 如果创建这个原始套接字时指定了非0的协议参数(socket的第三个参数)，那么接收到的数据报的协议字段必须匹配该值，否则该数据报不递送到这个套接字。
 - 如果这个原始套接字已由bind调用绑定了某个本地IP地址，那么接收到的数据报的目的IP地址必须匹配这个绑定地址，否则该数据报不递送到这个套接字。
 - 如果这个原始套接字已由connect调用指定了某个外地IP地址，那么接收到的数据报的源IP地址必须匹配这个已连接地址，否则该数据报不递送到这个套接字。
- 往一个原始IPv4套接字递送一个接收到的数据报，传递到该套接字所在进程的都是包括IP首部在内的完整数据报，对于IPv6套接字，传递到套接字的只是扣除了IPv6首部和所有扩展首部的净荷
- 定义原始套接字的目的在于提供一个访问某个协议的接口，就像该协议在内核中提供了这个接口那样
- ICMPv6类型过滤
 - ICMPv6在功用上是ICMPv4的超集，它把ARP和IGMP的功能也包括在内
 - struct icmp6_filter

```

#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_WILPPASS(int msgtype, const struct icmp6_filter *filt);
int ICMP6_FILTER_WILLBLOCK(int msgtype, const struct icmp6_filter *filt);

```

均返回：若过滤器放行（或阻止）给定消息类型则为1，否则为0

28.5 ping程序

- 开发一个同时支持IPv4和IPv6的ping程序版本
- ping程序往某个IP地址发送一个ICMP回送请求，该节点则以一个ICMP回射应答响应

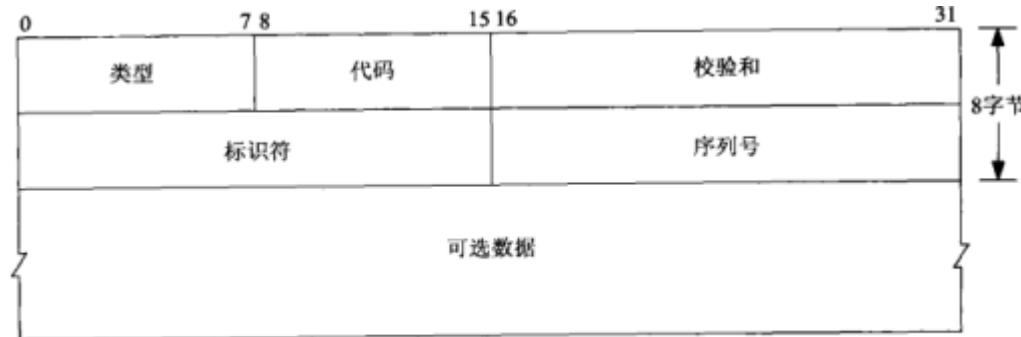


图28-1 ICMPv4和ICMPv6回射请求和回射应答消息的格式

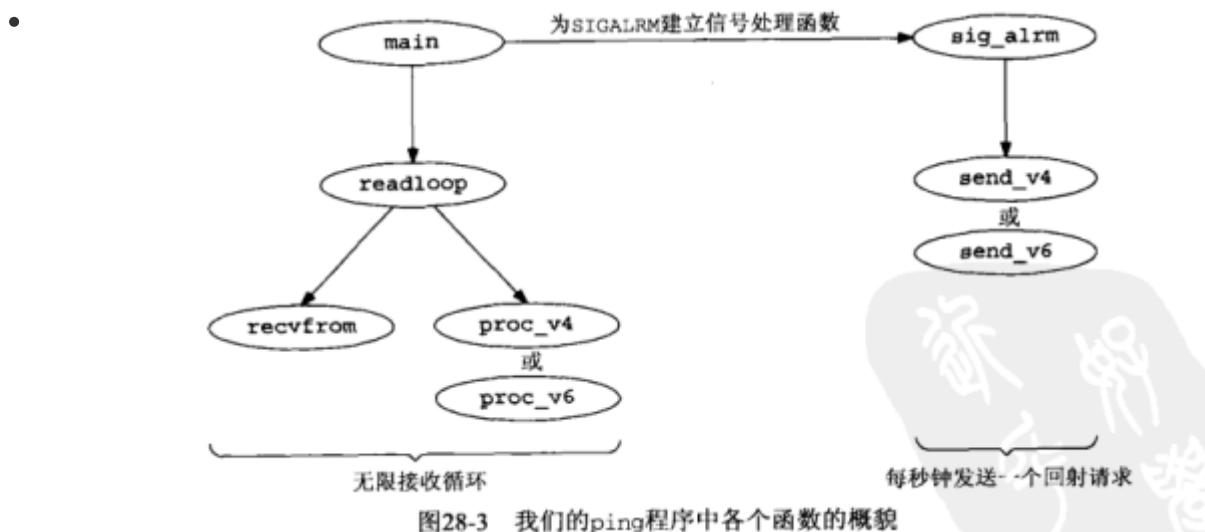


图28-3 我们的ping程序中各个函数的概貌

```

• #include "ping.h"

struct proto proto_v4 = { proc_v4, send_v4, NULL, NULL, NULL, 0, IPPROTO_ICMP };

#ifndef IPV6
struct proto proto_v6 = { proc_v6, send_v6, init_v6, NULL, NULL, 0, IPPROTO_ICMPV6 };
#endif

int datalen = 56; /* data that goes with ICMP echo request */

int
main(int argc, char **argv)
{
    int

```

```

    struct addrinfo *ai;
    char *h;

    opterr = 0;           /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "v")) != -1) {
        switch (c) {
        case 'v':
            verbose++;
            break;

        case '?':
            err_quit("unrecognized option: %c", c);
        }
    }

    if (optind != argc-1)
        err_quit("usage: ping [ -v ] <hostname>");
    host = argv[optind];

    pid = getpid() & 0xffff;          /* ICMP ID field is 16 bits */
    Signal(SIGALRM, sig_alarm);

    ai = Host_serv(host, NULL, 0, 0);

    h = Sock_ntop_host(ai->ai_addr, ai->ai_addrlen);
    printf("PING %s (%s): %d data bytes\n",
           ai->ai_canonname ? ai->ai_canonname : h,
           h, datalen);

    /* Initialize according to protocol */
    if (ai->ai_family == AF_INET) {
        pr = &proto_v4;
#define IPV6
    } else if (ai->ai_family == AF_INET6) {
        pr = &proto_v6;
        if (IN6_IS_ADDR_V4MAPPED(&(((struct sockaddr_in6 *)
                                      ai->ai_addr)-
>sin6_addr)))
            err_quit("cannot ping IPv4-mapped IPv6 address");
#endif
    } else
        err_quit("unknown address family %d", ai->ai_family);

    pr->sasend = ai->ai_addr;
    pr->sarecv = Calloc(1, ai->ai_addrlen);
    pr->salen = ai->ai_addrlen;

    readloop();

    exit(0);
}

```

- readloop函数

- ```

o #include "ping.h"

void
readloop(void)
{
 int size;
 char recvbuf[BUFSIZE];
 char controlbuf[BUFSIZE];
 struct msghdr msg;
 struct iovec iov;
 ssize_t n;
 struct timeval tval;

 sockfd = Socket(pr->sasend->sa_family, SOCK_RAW, pr->icmppproto);
 setuid(getuid()); /* don't need special permissions any more
*/
 if (pr->finit)
 (*pr->finit)();

 size = 60 * 1024; /* OK if setsockopt fails */
 setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));

 sig_alarm(SIGALRM); /* send first packet */

 iov.iov_base = recvbuf;
 iov.iov_len = sizeof(recvbuf);
 msg.msg_name = pr->sarecv;
 msg.msg_iov = &iov;
 msg.msg iovlen = 1;
 msg.msg_control = controlbuf;
 for (; ;) {
 msg.msg_namelen = pr->salen;
 msg.msg_controllen = sizeof(controlbuf);
 n = recvmsg(sockfd, &msg, 0);
 if (n < 0) {
 if (errno == EINTR)
 continue;
 else
 err_sys("recvmsg error");
 }
 Gettimeofday(&tval, NULL);
 (*pr->fproc)(recvbuf, n, &msg, &tval);
 }
}

```

- proc\_v4函数

- ```

o #include      "ping.h"

void
proc_v4(char *ptr, ssize_t len, struct msghdr *msg, struct timeval *tvrecv)

```

```

{
    int                     hlen1, icmplen;
    double                  rtt;
    struct ip               *ip;
    struct icmp              *icmp;
    struct timeval   *tvsend;

    ip = (struct ip *) ptr;           /* start of IP header */
    hlen1 = ip->ip_hl << 2;         /* length of IP header */
    if (ip->ip_p != IPPROTO_ICMP)
        return;                      /* not ICMP */

    icmp = (struct icmp *) (ptr + hlen1); /* start of ICMP header */
    if ((icmplen = len - hlen1) < 8)
        return;                      /* malformed packet */

    if (icmp->icmp_type == ICMP_ECHOREPLY) {
        if (icmp->icmp_id != pid)
            return;                  /* not a response to our
ECHO_REQUEST */
        if (icmplen < 16)
            return;                  /* not enough data to use */

        tvsend = (struct timeval *) icmp->icmp_data;
        tv_sub(tvrecv, tvsend);
        rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

        printf("%d bytes from %s: seq=%u, ttl=%d, rtt=%.3f ms\n",
               icmplen, Sock_ntop_host(pr->sarecv, pr->salen),
               icmp->icmp_seq, ip->ip_ttl, rtt);

    } else if (verbose) {
        printf(" %d bytes from %s: type = %d, code = %d\n",
               icmplen, Sock_ntop_host(pr->sarecv, pr->salen),
               icmp->icmp_type, icmp->icmp_code);
    }
}

```

o

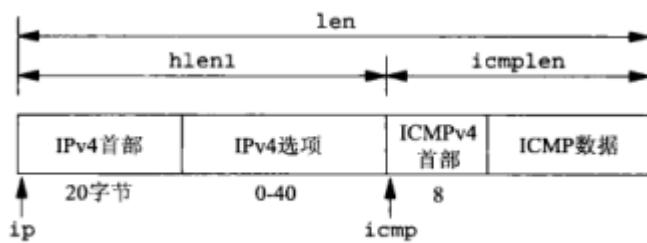


图28-9 处理ICMPv4应答涉及的首部、指针和长度

- o 如果本主机上同时运行着多个ping进程，那么每个进程都得到内核接收到的所有ICMP消息的一个副本
- init_v6函数

o #include "ping.h"

```

void
init_v6()
{
#ifdef IPV6
    int on = 1;

    if (verbose == 0) {
        /* install a filter that only passes ICMP6_ECHO_REPLY unless verbose
 */
        struct icmp6_filter myfilt;
        ICMP6_FILTER_SETBLOCKALL(&myfilt);
        ICMP6_FILTER_SETPASS(ICMP6_ECHO_REPLY, &myfilt);
        setsockopt(sockfd, IPPROTO_IPV6, ICMP6_FILTER, &myfilt,
sizeof(myfilt));
        /* ignore error return; the filter is an optimization */
    }

    /* ignore error returned below; we just won't receive the hop limit */
#endif IPV6_RECVHOPLIMIT
    /* RFC 3542 */
    setsockopt(sockfd, IPPROTO_IPV6, IPV6_RECVHOPLIMIT, &on, sizeof(on));
#else
    /* RFC 2292 */
    setsockopt(sockfd, IPPROTO_IPV6, IPV6_HOPLIMIT, &on, sizeof(on));
#endif
#endif
}

```

- #include "ping.h"

```

void
proc_v6(char *ptr, ssize_t len, struct msghdr *msg, struct timeval* tvrecv)
{
#ifdef IPV6
    double rtt;
    struct icmp6_hdr *icmp6;
    struct timeval *tvsend;
    struct cmsghdr *cmsg;
    int hlim;
    icmp6 = (struct icmp6_hdr *) ptr;
    if (len < 8)
        return; /* malformed packet */

    if (icmp6->icmp6_type == ICMP6_ECHO_REPLY) {
        if (icmp6->icmp6_id != pid)
            return; /* not a response to our
ECHO_REQUEST */
        if (len < 16)
            return; /* not enough data to use */

        tvsend = (struct timeval *) (icmp6 + 1);
        tv_sub(tvrecv, tvsend);
    }
}

```

```

    rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

    hlim = -1;
    for (cmsg = CMSG_FIRSTHDR(msg); cmsg != NULL; cmsg =
CMSG_NXTHDR(msg, cmsg)) {
        if (cmsg->cmsg_level == IPPROTO_IPV6 &&
            cmsg->cmsg_type == IPV6_HOPLIMIT) {
            hlim = *(u_int32_t *)CMSG_DATA(cmsg);
            break;
        }
    }
    printf("%d bytes from %s: seq=%u, hlim=%",
           len, Sock_ntop_host(pr->sarecv, pr->salen),
           icmp6->icmp6_seq);
    if (hlim == -1)
        printf("????"); /* ancillary data missing */
    else
        printf("%d", hlim);
    printf(", rtt=%.3f ms\n", rtt);
} else if (verbose) {
    printf(" %d bytes from %s: type = %d, code = %d\n",
           len, Sock_ntop_host(pr->sarecv, pr->salen),
           icmp6->icmp6_type, icmp6->icmp6_code);
}
#endif /* IPV6 */
}

```

- o

```
#include      "ping.h"

void
sig_alrm(int signo)
{
    (*pr->fsend)();

    alarm(1);
    return;
}
```

- o

```
#include      "ping.h"

void
send_v4(void)
{
    int                  len;
    struct icmp       *icmp;

    icmp = (struct icmp *) sendbuf;
    icmp->icmp_type = ICMP_ECHO;
    icmp->icmp_code = 0;
    icmp->icmp_id = pid;
    icmp->icmp_seq = nsent++;
    memset(icmp->icmp_data, 0xa5, datalen); /* fill with pattern */
```

```

    Gettimeofday((struct timeval *) icmp->icmp_data, NULL);

    len = 8 + datalen;                      /* checksum ICMP header and data */
    icmp->icmp_cksum = 0;
    icmp->icmp_cksum = in_cksum((u_short *) icmp, len);

    Sendto(sockfd, sendbuf, len, 0, pr->sasend, pr->salen);
}

```

- 网际网校验和是被校验的各个16位值的二进制反码和，本算法适用于IPv4 ICMPv4 IGMPv4 ICMPv6 UDP和TCP等首部的校验和字段
- in_cksum函数用于计算校验和

```

o #include "unp.h"

uint16_t
in_cksum(uint16_t *addr, int len)
{
    int                  nleft = len;
    uint32_t             sum = 0;
    uint16_t             *w = addr;
    uint16_t             answer = 0;

    /*
     * Our algorithm is simple, using a 32 bit accumulator (sum), we add
     * sequential 16 bit words to it, and at the end, fold back all the
     * carry bits from the top 16 bits into the lower 16 bits.
     */
    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }

    /* 4mop up an odd byte, if necessary */
    if (nleft == 1) {
        *(unsigned char *)&answer = *(unsigned char *)w ;
        sum += answer;
    }

    /* 4add back carry outs from top 16 bits to low 16 bits */
    sum = (sum >> 16) + (sum & 0xffff);      /* add hi 16 to low 16 */
    sum += (sum >> 16);                     /* add carry */
    answer = ~sum;                           /* truncate to 16 bits */
    return(answer);
}

```

- #include "ping.h"

```

void
send_v6()
{
#endif IPV6

```

```

    int                                len;
    struct icmp6_hdr      *icmp6;

    icmp6 = (struct icmp6_hdr *) sendbuf;
    icmp6->icmp6_type = ICMP6_ECHO_REQUEST;
    icmp6->icmp6_code = 0;
    icmp6->icmp6_id = pid;
    icmp6->icmp6_seq = nsent++;
    memset((icmp6 + 1), 0xa5, datalen);      /* fill with pattern */
    Gettimeofday((struct timeval *) (icmp6 + 1), NULL);

    len = 8 + datalen;                  /* 8-byte ICMPV6 header */

    Sendto(sockfd, sendbuf, len, 0, pr->sasend, pr->salen);
    /* 4kernel calculates and stores checksum for us */
#endif /* IPV6 */
}

```

28.6 traceroute程序

- 一个同时支持IPv4和IPv6的版本
- traceroute允许我们确定IP数据报从本地主机游历到某个远程主机所经过的路劲
- traceroute使用IPv4的TTL字段或IPv6的跳限字段以及两种ICMP消息，它一开始向目的地发送一个TTL为1的UDP数据报，这个数据报导致第一跳路由器返回一个ICMP传输超时错误，接着递增TTL一次发送一个UDP数据报，从而逐步确定下一跳路由器，当某个UDP数据报到达最终目的地时，目标是由这个主机返回一个ICMP端口不可达错误
- main函数

```

o #include      "trace.h"

struct proto    proto_v4 = { icmpcode_v4, recv_v4, NULL, NULL, NULL, NULL, 0,
                           IPPROTO_ICMP, IPPROTO_IP,
                           IP_TTL };

#ifndef IPV6
struct proto    proto_v6 = { icmpcode_v6, recv_v6, NULL, NULL, NULL, NULL, 0,
                           IPPROTO_ICMPV6,
                           IPPROTO_IPV6, IPV6_UNICAST_HOPS };
#endif

int            datalen = sizeof(struct rec); /* defaults */
int            max_ttl = 30;
int            nprobes = 3;
u_short        dport = 32768 + 666;

int
main(int argc, char **argv)
{
    int

```

```

    struct addrinfo *ai;
    char *h;

    opterr = 0;                  /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "m:v")) != -1) {
        switch (c) {
        case 'm':
            if ( (max_ttl = atoi(optarg)) <= 1)
                err_quit("invalid -m value");
            break;

        case 'v':
            verbose++;
            break;

        case '?':
            err_quit("unrecognized option: %c", c);
        }

    }

    if (optind != argc-1)
        err_quit("usage: traceroute [ -m <maxttl> -v ] <hostname>");
    host = argv[optind];

    pid = getpid();
    Signal(SIGALRM, sig_alarm);

    ai = Host_serv(host, NULL, 0, 0);

    h = Sock_ntop_host(ai->ai_addr, ai->ai_addrlen);
    printf("traceroute to %s (%s): %d hops max, %d data bytes\n",
           ai->ai_canonname ? ai->ai_canonname : h,
           h, max_ttl, datalen);

    /* initialize according to protocol */
    if (ai->ai_family == AF_INET) {
        pr = &proto_v4;
#ifdef IPV6
    } else if (ai->ai_family == AF_INET6) {
        pr = &proto_v6;
        if (IN6_IS_ADDR_V4MAPPED(&(((struct sockaddr_in6 *)ai->ai_addr)->sin6_addr)))
            err_quit("cannot traceroute IPv4-mapped IPv6 address");
#endif
    } else
        err_quit("unknown address family %d", ai->ai_family);

    pr->sasend = ai->ai_addr;             /* contains destination address */
    pr->sarecv = Calloc(1, ai->ai_addrlen);
    pr->salast = Calloc(1, ai->ai_addrlen);
    pr->sabind = Calloc(1, ai->ai_addrlen);
    pr->salen = ai->ai_addrlen;

```

```

    traceloop();

    exit(0);
}

```

- traceloop函数发送UDP数据报并读取返送的ICMP出错消息

- 需要两个套接字，从中读入所有返送ICMP消息的一个原始套接字，从中以不断递增的TTL写出探测分组的一个UDP套接字

```

◦ #include      "trace.h"

void
traceloop(void)
{
    int                      seq, code, done;
    double                   rtt;
    struct rec               *rec;
    struct timeval           tvrecv;

    recvfd = Socket(pr->sasend->sa_family, SOCK_RAW, pr->icmppproto);
    setuid(getuid());          /* don't need special permissions anymore */

#ifndef IPV6
    if (pr->sasend->sa_family == AF_INET6 && verbose == 0) {
        struct icmp6_filter myfilt;
        ICMP6_FILTER_SETBLOCKALL(&myfilt);
        ICMP6_FILTER_SETPASS(ICMP6_TIME_EXCEEDED, &myfilt);
        ICMP6_FILTER_SETPASS(ICMP6_DST_UNREACH, &myfilt);
        setsockopt(recvfd, IPPROTO_IPV6, ICMP6_FILTER,
                    &myfilt, sizeof(myfilt));
    }
#endif

    sendfd = Socket(pr->sasend->sa_family, SOCK_DGRAM, 0);

    pr->sabind->sa_family = pr->sasend->sa_family;
    sport = (getpid() & 0xffff) | 0x8000; /* our source UDP port # */
    sock_set_port(pr->sabind, pr->salen, htons(sport));
    Bind(sendfd, pr->sabind, pr->salen);

    sig_alrm(SIGALRM);

    seq = 0;
    done = 0;
    for (ttl = 1; ttl <= max_ttl && done == 0; ttl++) {
        Setsockopt(sendfd, pr->ttllevel, pr->ttloptname, &ttl, sizeof(int));
        bzero(pr->salast, pr->salen);

        printf("%2d ", ttl);
        fflush(stdout);

        for (probe = 0; probe < nprobes; probe++) {
            rec = (struct rec *) sendbuf;

```

```

        rec->rec_seq = ++seq;
        rec->rec_ttl = ttl;
        Gettimeofday(&rec->rec_tv, NULL);

        sock_set_port(pr->sasend, pr->salen, htons(dport + seq));
        Sendto(sendfd, sendbuf, datalen, 0, pr->sasend, pr->salen);

        if ( (code = (*pr->recv)(seq, &tvrecv)) == -3)
                printf(" *"); /* timeout, no reply */
        else {
                char     str[NI_MAXHOST];

                if (sock_cmp_addr(pr->sarecv, pr->salast, pr->salen)
!= 0) {
                        if (getnameinfo(pr->sarecv, pr->salen, str,
sizeof(str),
NULL, 0, 0)
== 0)
                                printf(" %s (%s)", str,
Sock_ntop_host(pr-
>sarecv, pr->salen));
                        else
                                printf(" %s",
Sock_ntop_host(pr-
>sarecv, pr->salen));
                }
                memcpy(pr->salast, pr->sarecv, pr->salen);
        }
        tv_sub(&tvrecv, &rec->rec_tv);
        rtt = tvrecv.tv_sec * 1000.0 + tvrecv.tv_usec /
1000.0;
        printf(" %.3f ms", rtt);

        if (code == -1) /* port unreachable; at
destination */
                done++;
        else if (code >= 0)
                printf(" (ICMP %s)", (*pr->icmpcode)(code));
        }
        fflush(stdout);
    }
    printf("\n");
}
}

```

- recv_v4函数

-

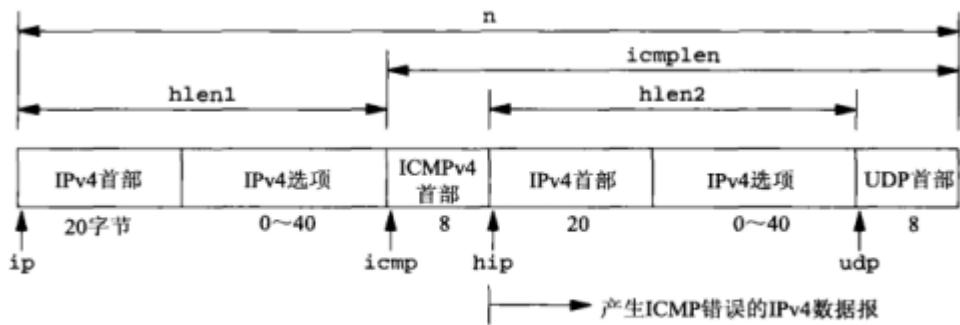


图28-21 处理ICMPv4错误涉及的首部、指针和长度

```

○ #include      "trace.h"

extern int gotalarm;

/*
 * Return: -3 on timeout
 *          -2 on ICMP time exceeded in transit (caller keeps going)
 *          -1 on ICMP port unreachable (caller is done)
 *          >= 0 return value is some other ICMP unreachable code
 */

int
recv_v4(int seq, struct timeval *tv)
{
    int                      hlen1, hlen2, icmplen, ret;
    socklen_t                 len;
    ssize_t                   n;
    struct ip                *ip, *hip;
    struct icmp               *icmp;
    struct udphdr             *udp;

    gotalarm = 0;
    alarm(3);
    for ( ; ; ) {
        if (gotalarm)
            return(-3);           /* alarm expired */
        len = pr->salen;
        n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0, pr->sarecv, &len);
        if (n < 0) {
            if (errno == EINTR)
                continue;
            else
                err_sys("recvfrom error");
        }

        ip = (struct ip *) recvbuf;      /* start of IP header */
        hlen1 = ip->ip_hl << 2;         /* length of IP header */

        icmp = (struct icmp *) (recvbuf + hlen1); /* start of ICMP header */
        if ((icmplen = n - hlen1) < 8)
            continue;                  /* not enough to
look at ICMP header */
    }
}

```



```

*/
    Gettimeofday(tv, NULL);           /* get time of packet arrival */
    return(ret);
}

```

```

o #include      "trace.h"

extern int gotalarm;

/*
 * Return: -3 on timeout
 *          -2 on ICMP time exceeded in transit (caller keeps going)
 *          -1 on ICMP port unreachable (caller is done)
 *          >= 0 return value is some other ICMP unreachable code
 */

int
recv_v6(int seq, struct timeval *tv)
{
#ifdef IPV6
    int                               hlen2, icmp6len, ret;
    ssize_t                           n;
    socklen_t                         len;
    struct ip6_hdr                  *ip6;
    struct icmp6_hdr                *icmp6;
    struct udphdr                   *udp;

    gotalarm = 0;
    alarm(3);
    for ( ; ; ) {
        if (gotalarm)
            return(-3);           /* alarm expired */
        len = pr->salen;
        n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0, pr->sarecv, &len);
        if (n < 0) {
            if (errno == EINTR)
                continue;
            else
                err_sys("recvfrom error");
        }

        icmp6 = (struct icmp6_hdr *) recvbuf; /* ICMP header */
        if ( (icmp6len = n) < 8)
            continue;           /* not enough to
look at ICMP header */

        if (icmp6->icmp6_type == ICMP6_TIME_EXCEEDED &&
            icmp6->icmp6_code == ICMP6_TIME_EXCEED_TRANSIT) {
            if (icmp6len < 8 + sizeof(struct ip6_hdr) + 4)
                continue;           /* not enough data
to look at inner header */

        ip6 = (struct ip6_hdr *) (recvbuf + 8);

```

```

        hlen2 = sizeof(struct ip6_hdr);
        udp = (struct udphdr *) (recvbuf + 8 + hlen2);
        if (hip6->ip6_nxt == IPPROTO_UDP &&
            udp->uh_sport == htons(sport) &&
            udp->uh_dport == htons(dport + seq))
            ret = -2;                                /* we hit an intermediate
router */
break;

} else if (icmp6->icmp6_type == ICMP6_DST_UNREACH) {
    if (icmp6len < 8 + sizeof(struct ip6_hdr) + 4)
        continue;                            /* not enough data
to look at inner header */

        hip6 = (struct ip6_hdr *) (recvbuf + 8);
        hlen2 = sizeof(struct ip6_hdr);
        udp = (struct udphdr *) (recvbuf + 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)
                ret = -1;                  /* have reached destination
*/
        else
            ret = icmp6->icmp6_code;           /* 0, 1, 2,
... */
break;
}
} else if (verbose) {
    printf(" (from %s: type = %d, code = %d)\n",
           Sock_ntop_host(pr->sarecv, pr->salen),
           icmp6->icmp6_type, icmp6->icmp6_code);
}
/* Some other ICMP error, recvfrom() again */
}
alarm(0);                                /* don't leave alarm running
*/
Gettimeofday(tv, NULL);                  /* get time of packet arrival */
return(ret);
#endif
}

```

28.7 一个ICMP消息守护程序

- 在UDP套接字上接收异步ICMP错误(即ICMP出错消息)一直来是一个问题，ICMP错误由内核收取之后很少被递送到需要了解它们的应用进程
-

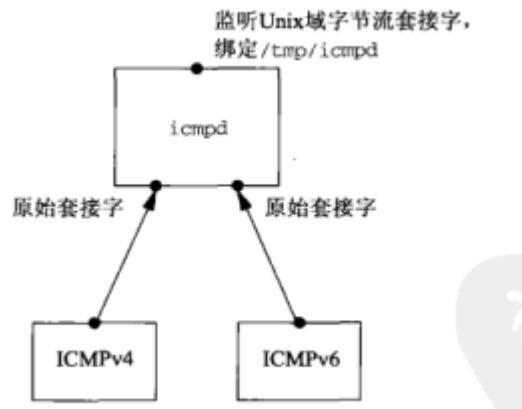


图28-26 icmpd守护进程：初始创建的套接字

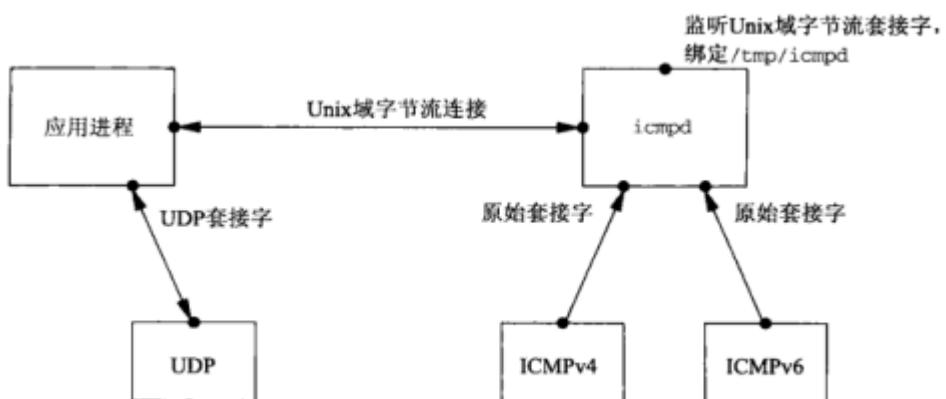


图28-27 应用进程创建自身的UDP套接字和到icmpd的Unix域连接

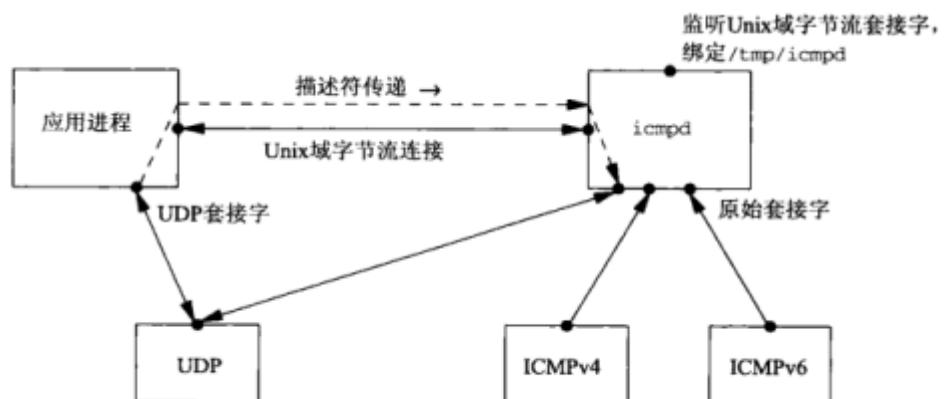


图28-28 应用进程跨Unix域连接把UDP套接字传递给icmpd

- `icmpd`一旦收取由该应用进程通过绑定在它的UDP套接字上的端口发送的UDP数据报所引发的任何ICMP错误，就通过Unix域连接向该应用进程发送一个消息

```

        struct timeval tv;
        struct icmpd_err icmpd_err;
        struct sockaddr_un sun;

        Sock_bind_wild(sockfd, pservaddr->sa_family);

        icmpfd = Socket(AF_LOCAL, SOCK_STREAM, 0);
        sun.sun_family = AF_LOCAL;
        strcpy(sun.sun_path, ICMPD_PATH);
        Connect(icmpfd, (SA *)&sun, sizeof(sun));
        Write_fd(icmpfd, "1", 1, sockfd);
        n = Read(icmpfd, recvline, 1);
        if (n != 1 || recvline[0] != '1')
            err_quit("error creating icmp socket, n = %d, char = %c",
                     n, recvline[0]);

        FD_ZERO(&rset);
        maxfdp1 = max(sockfd, icmpfd) + 1;
/* end dgcli011 */

/* include dgcli012 */
        while (Fgets(sendline, MAXLINE, fp) != NULL) {
            Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

            tv.tv_sec = 5;
            tv.tv_usec = 0;
            FD_SET(sockfd, &rset);
            FD_SET(icmpfd, &rset);
            if ( (n = Select(maxfdp1, &rset, NULL, NULL, &tv)) == 0) {
                fprintf(stderr, "socket timeout\n");
                continue;
            }

            if (FD_ISSET(sockfd, &rset)) {
                n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
                recvline[n] = 0; /* null terminate */
                Fputs(recvline, stdout);
            }

            if (FD_ISSET(icmpfd, &rset)) {
                if ( (n = Read(icmpfd, &icmpd_err, sizeof(icmpd_err))) == 0)
                    err_quit("ICMP daemon terminated");
                else if (n != sizeof(icmpd_err))
                    err_quit("n = %d, expected %d", n, sizeof(icmpd_err));
                printf("ICMP error: dest = %s, %s, type = %d, code = %d\n",
                       Sock_ntop(&icmpd_err.icmpd_dest, icmpd_err.icmpd_len),
                       strerror(icmpd_err.icmpd_errno),
                       icmpd_err.icmpd_type, icmpd_err.icmpd_code);
            }
        }
/* end dgcli012 */

```

- icmpd守护进程

```

o /* include icmpd1 */
#include      "icmpd.h"

int
main(int argc, char **argv)
{
    int          i, sockfd;
    struct sockaddr_un sun;

    if (argc != 1)
        err_quit("usage: icmpd");

    maxi = -1;                                /* index into client[] array
*/
    for (i = 0; i < FD_SETSIZE; i++)
        client[i].connfd = -1; /* -1 indicates available entry */
    FD_ZERO(&allset);

    fd4 = Socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    FD_SET(fd4, &allset);
    maxfd = fd4;

#ifndef IPV6
    fd6 = Socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);
    FD_SET(fd6, &allset);
    maxfd = max(maxfd, fd6);
#endif

    listenfd = Socket(AF_UNIX, SOCK_STREAM, 0);
    sun.sun_family = AF_LOCAL;
    strcpy(sun.sun_path, ICMPD_PATH);
    unlink(ICMPD_PATH);
    Bind(listenfd, (SA *)&sun, sizeof(sun));
    Listen(listenfd, LISTENQ);
    FD_SET(listenfd, &allset);
    maxfd = max(maxfd, listenfd);
/* end icmpd1 */

/* include icmpd2 */
    for ( ; ; ) {
        rset = allset;
        nready = Select(maxfd+1, &rset, NULL, NULL, NULL);

        if (FD_ISSET(listenfd, &rset))
            if (readable_listen() <= 0)
                continue;

        if (FD_ISSET(fd4, &rset))
            if (readable_v4() <= 0)
                continue;

```

```

#define IPV6

        if (FD_ISSET(fd6, &rset))
            if (readable_v6() <= 0)
                continue;

#endif

        for (i = 0; i <= maxi; i++) { /* check all clients for data */
            if ( (sockfd = client[i].connfd) < 0)
                continue;
            if (FD_ISSET(sockfd, &rset))
                if (readable_conn(i) <= 0)
                    break;                                /* no more
readable descriptors */
            }
        }

        exit(0);
    }
/* end icmpd2 */

```

- o `#include "icmpd.h"`

```

int
readable_listen(void)
{
    int                  i, connfd;
    socklen_t      clilen;

    clilen = sizeof(cliaddr);
    connfd = Accept(listenfd, (SA *)&cliaddr, &clilen);

    /* find first available client[] structure */
    for (i = 0; i < FD_SETSIZE; i++)
        if (client[i].connfd < 0) {
            client[i].connfd = connfd;      /* save descriptor */
            break;
        }
    if (i == FD_SETSIZE) {
        close(connfd);           /* can't handle new client, */
        return(--nready);        /* rudely close the new connection */
    }
    printf("new connection, i = %d, connfd = %d\n", i, connfd);

    FD_SET(connfd, &allset);      /* add new descriptor to set */
    if (connfd > maxfd)
        maxfd = connfd;          /* for select() */
    if (i > maxi)
        maxi = i;                /* max index in client[]
array */

    return(--nready);
}

```

```

o /* include readable_conn1 */
#include      "icmpd.h"

int
readable_conn(int i)
{
    int                      unixfd, recvfd;
    char                     c;
    ssize_t                  n;
    socklen_t                len;
    struct sockaddr_storage ss;

    unixfd = client[i].connfd;
    recvfd = -1;
    if ( (n = Read_fd(unixfd, &c, 1, &recvfd)) == 0) {
        err_msg("client %d terminated, recvfd = %d", i, recvfd);
        goto clientdone;          /* client probably terminated */
    }

    /* 4data from client; should be descriptor */
    if (recvfd < 0) {
        err_msg("read_fd did not return descriptor");
        goto clienterr;
    }
/* end readable_conn1 */

/* include readable_conn2 */
    len = sizeof(ss);
    if (getsockname(recvfd, (SA *) &ss, &len) < 0) {
        err_ret("getsockname error");
        goto clienterr;
    }

    client[i].family = ss.ss_family;
    if ( (client[i].lport = sock_get_port((SA *)&ss, len)) == 0) {
        client[i].lport = sock_bind_wild(recvfd, client[i].family);
        if (client[i].lport <= 0) {
            err_ret("error binding ephemeral port");
            goto clienterr;
        }
    }
    Write(unixfd, "1", 1); /* tell client all OK */
    Close(recvfd);           /* all done with client's UDP socket */
    return(--nready);

clienterr:
    Write(unixfd, "0", 1); /* tell client error occurred */
clientdone:
    Close(unixfd);
    if (recvfd >= 0)
        Close(recvfd);
    FD_CLR(unixfd, &allset);
    client[i].connfd = -1;

```

```

        return(--nready);
}
/* end readable_conn2 */

```

```

o /* include readable_v41 */
#include      "icmpd.h"
#include      <netinet/in_systm.h>
#include      <netinet/ip.h>
#include      <netinet/ip_icmp.h>
#include      <netinet/udp.h>

int
readable_v4(void)
{
    int                      i, hlen1, hlen2, icmplen, sport;
    char                     buf[MAXLINE];
    char                     srcstr[INET_ADDRSTRLEN],
    dststr[INET_ADDRSTRLEN];
    ssize_t                  n;
    socklen_t                len;
    struct ip                *ip, *hip;
    struct icmp               *icmp;
    struct udphdr             *udp;
    struct sockaddr_in         from, dest;
    struct icmpd_err           icmpd_err;

    len = sizeof(from);
    n = Recvfrom(fd4, buf, MAXLINE, 0, (SA *) &from, &len);

    printf("%d bytes ICMPv4 from %s:",
           n, Sock_ntop_host((SA *) &from, len));

    ip = (struct ip *) buf;          /* start of IP header */
    hlen1 = ip->ip_hl << 2;          /* length of IP header */

    icmp = (struct icmp *) (buf + hlen1); /* start of ICMP header */
    if ((icmplen = n - hlen1) < 8)
        err_quit("icmplen (%d) < 8", icmplen);

    printf(" type = %d, code = %d\n", icmp->icmp_type, icmp->icmp_code);
/* end readable_v41 */

/* include readable_v42 */
    if (icmp->icmp_type == ICMP_UNREACH ||
        icmp->icmp_type == ICMP_TIMXCEED ||
        icmp->icmp_type == ICMP_SOURCEQUENCH) {
        if (icmplen < 8 + 20 + 8)
            err_quit("icmplen (%d) < 8 + 20 + 8", icmplen);

        hip = (struct ip *) (buf + hlen1 + 8);
        hlen2 = hip->ip_hl << 2;
        printf("\tsrcip = %s, dstip = %s, proto = %d\n",
               Inet_ntop(AF_INET, &hip->ip_src, srcstr, sizeof(srcstr)),

```

```

                Inet_ntop(AF_INET, &hip->ip_dst, dststr, sizeof(dststr)),
                hip->ip_p);
        if (hip->ip_p == IPPROTO_UDP) {
                udp = (struct udphdr *) (buf + hlen1 + 8 + hlen2);
                sport = udp->uh_sport;

                /* 4find client's Unix domain socket, send headers
 */
                for (i = 0; i <= maxi; i++) {
                        if (client[i].connfd >= 0 &&
                                client[i].family == AF_INET &&
                                client[i].lport == sport) {

                                bzero(&dest, sizeof(dest));
                                dest.sin_family = AF_INET;
                                dest.sin_len = sizeof(dest);
#endif
                                memcpy(&dest.sin_addr, &hip->ip_dst,
                                       sizeof(struct in_addr));
                                dest.sin_port = udp->uh_dport;

                                icmpd_err.icmpd_type = icmp->icmp_type;
                                icmpd_err.icmpd_code = icmp->icmp_code;
                                icmpd_err.icmpd_len = sizeof(struct
sockaddr_in);
                                memcpy(&icmpd_err.icmpd_dest, &dest,
                                       sizeof(dest));

                                /* 4convert type & code to
reasonable errno value */
                                icmpd_err.icmpd_errno = EHOSTUNREACH; /*

default */
                                if (icmp->icmp_type == ICMP_UNREACH) {
                                        if (icmp->icmp_code ==
ICMP_UNREACH_PORT)
                                                icmpd_err.icmpd_errno =
ECONNREFUSED;
                                        else if (icmp->icmp_code ==
ICMP_UNREACH_NEEDFRAG)
                                                icmpd_err.icmpd_errno =
EMSGSIZE;
                                }
                                Write(client[i].connfd, &icmpd_err,
                                       sizeof(icmpd_err));
}
}
}
return(--nready);
}
/* end readable_v42 */

```

```

o /* include readable_v61 */

#include      "icmpd.h"
#include      <netinet/in_systm.h>
#include      <netinet/ip.h>
#include      <netinet/ip_icmp.h>
#include      <netinet/udp.h>

#ifndef IPV6
#include      <netinet/ip6.h>
#include      <netinet/icmp6.h>
#endif

int
readable_v6(void)
{
#ifndef IPV6
    int                                i, hlen2, icmp6len, sport;
    char                               buf[MAXLINE];
    char                               srcstr[INET6_ADDRSTRLEN],
    dststr[INET6_ADDRSTRLEN];
    ssize_t                            n;
    socklen_t                          len;
    struct ip6_hdr                     *ip6, *hip6;
    struct icmp6_hdr                  *icmp6;
    struct udphdr                      *udp;
    struct sockaddr_in6                from, dest;
    struct icmpd_err                  icmpd_err;

```

len = sizeof(from);
n = Recvfrom(fd6, buf, MAXLINE, 0, (SA *) &from, &len);

printf("%d bytes ICMPv6 from %s:",
n, Sock_ntop_host((SA *) &from, len));

icmp6 = (struct icmp6_hdr *) buf; /* start of ICMPv6 header */
if ((icmp6len = n) < 8)
err_quit("icmp6len (%d) < 8", icmp6len);

printf(" type = %d, code = %d\n", icmp6->icmp6_type, icmp6->icmp6_code);

/* end readable_v61 */

/* include readable_v62 */
if (icmp6->icmp6_type == ICMP6_DST_UNREACH ||
icmp6->icmp6_type == ICMP6_PACKET_TOO_BIG ||
icmp6->icmp6_type == ICMP6_TIME_EXCEEDED) {
if (icmp6len < 8 + 8)
err_quit("icmp6len (%d) < 8 + 8", icmp6len);

hip6 = (struct ip6_hdr *) (buf + 8);
hlen2 = sizeof(struct ip6_hdr);
printf("\tsrcip = %s, dstip = %s, next hdr = %d\n",
Inet_ntop(AF_INET6, &hip6->ip6_src, srcstr,
sizeof(srcstr)),

```

        Inet_ntop(AF_INET6, &hip6->ip6_dst, dststr,
sizeof(dststr)),
                hip6->ip6_nxt);
if (hip6->ip6_nxt == IPPROTO_UDP) {
    udp = (struct udphdr *) (buf + 8 + hlen2);
    sport = udp->uh_sport;

        /* 4find client's Unix domain socket, send headers
*/
        for (i = 0; i <= maxi; i++) {
            if (client[i].connfd >= 0 &&
                client[i].family == AF_INET6 &&
                client[i].lport == sport) {

                bzero(&dest, sizeof(dest));
                dest.sin6_family = AF_INET6;

#ifdef HAVE_SOCKADDR_SA_LEN
                dest.sin6_len = sizeof(dest);
#endif
                memcpy(&dest.sin6_addr, &hip6->ip6_dst,
                       sizeof(struct in6_addr));
                dest.sin6_port = udp->uh_dport;

                icmpd_err.icmpd_type = icmp6->icmp6_type;
                icmpd_err.icmpd_code = icmp6->icmp6_code;
                icmpd_err.icmpd_len = sizeof(struct
sockaddr_in6);
                memcpy(&icmpd_err.icmpd_dest, &dest,
                       sizeof(dest));

                    /* 4convert type & code to
reasonable errno value */
                icmpd_err.icmpd_errno = EHOSTUNREACH; /* */
default */
                if (icmp6->icmp6_type == ICMP6_DST_UNREACH
&&
ICMP6_DST_UNREACH_NOPORT)
                    icmp6->icmp6_code ==
ECONNREFUSED;
                if (icmp6->icmp6_type ==
ICMP6_PACKET_TOOBIG)
                    icmpd_err.icmpd_errno =
EMSGSIZE;
                write(client[i].connfd, &icmpd_err,
sizeof(icmpd_err));
            }
        }
    }
return(--nready);
#endif

```

```
/* end readable_v62 */
```

第29章 数据链路访问

- 能够监视数据链路层接收的分组，使得诸如tcpdump之类的程序能够在普通计算机系统上运行
- 能够作为普遍应用进程而不是内核的一部分运行某些程序，如RARP服务器

29.2 BPF: BSD分组过滤器

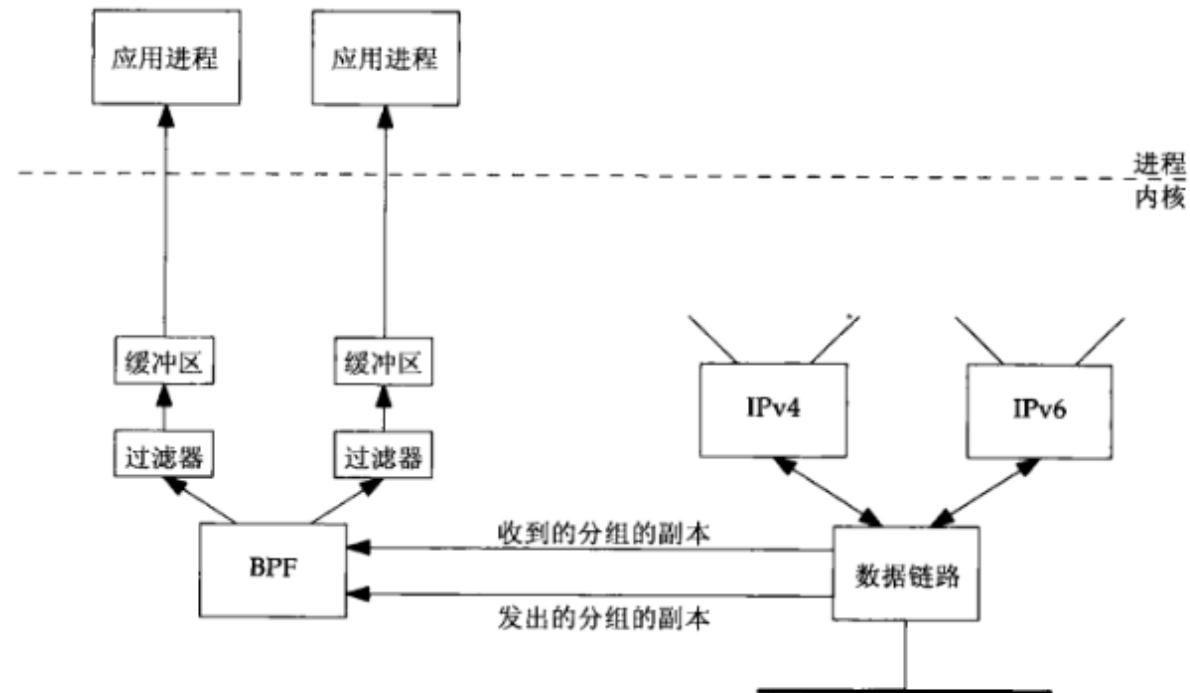


图29-1 使用BPF截获分组

- 为了访问BPF, 必须打开一个当前关闭着的BPF设备, 如/dev/bpf0, 然后使用ioctl命令来设置, 使用read和write执行I/O

29.3 DLPI: 数据链路提供者接口

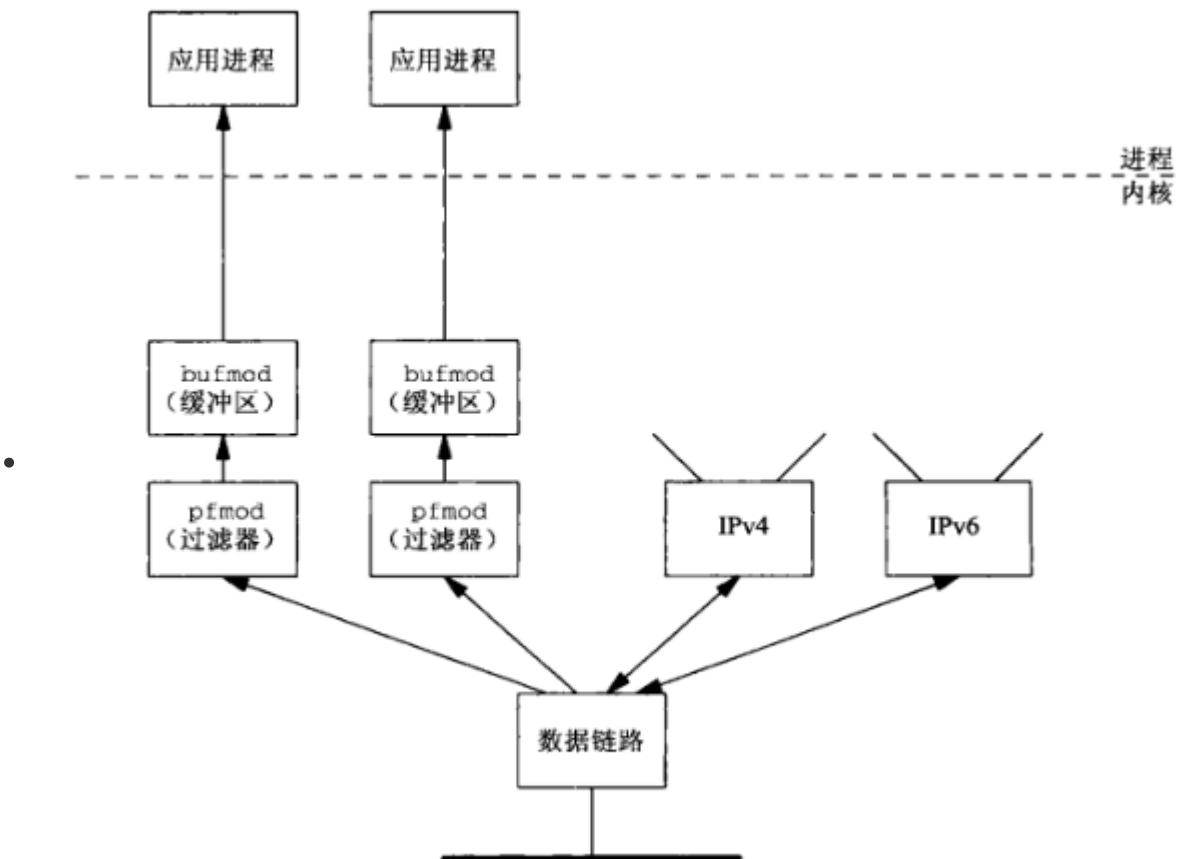


图29-2 使用DLPI、pfmod和bufmod捕获分组

29.4 Linux: SOCK_PACKET和PF_PACKET

- `fd=socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))`
- `fd=socket(AF_INET, SOCK_PACKET, htons(ETH_P_ALL))`

29.5 libpcap: 分组捕获函数库

29.6 libnet: 分组构造与输出函数库

29.6 检查UDP的校验和字段

•

我们将自行构造UDP数据报（即DNS查询），并把它写出到一个原始套接字。这个查询使用普通的UDP套接字就可以发送，不过我们想展示如何使用IP_HDRINCL套接字选项构造一个完整的IP数据报。

另一方面，我们无法在从普通UDP套接字读入时获取UDP校验和，使用原始套接字也无法读入UDP或TCP分组（28.4节）。因此我们必须使用分组捕获机制获取含有名字服务器的应答的完整UDP数据报。

我们还检查所获取 UDP 首部中的校验和字段，如果其值为 0，那么该名字服务器没有开启 UDP 校验和。我们还给出使用 libnet 编写的相同程序。

图29-3 汇总了本程序的操作。

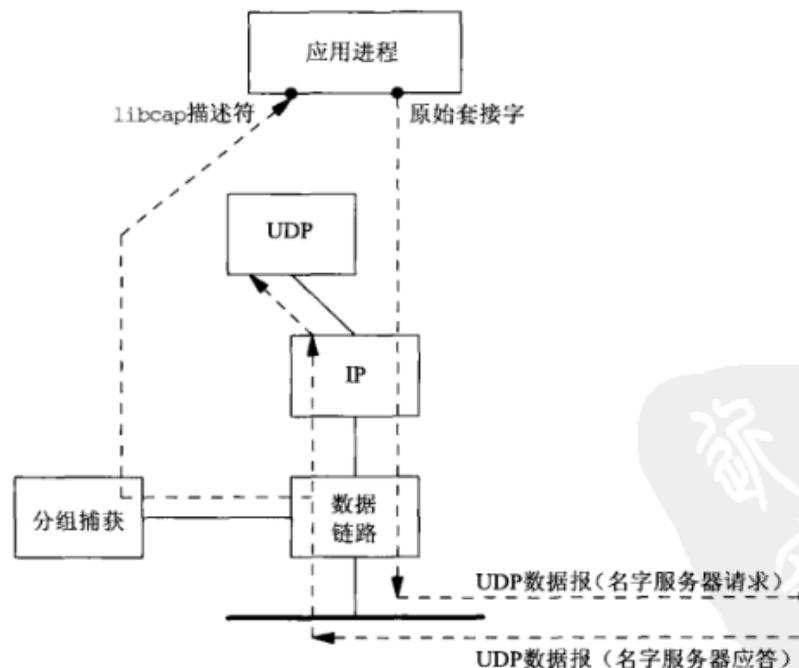


图29-3 检查某个名字服务器是否开启UDP校验和的应用程序

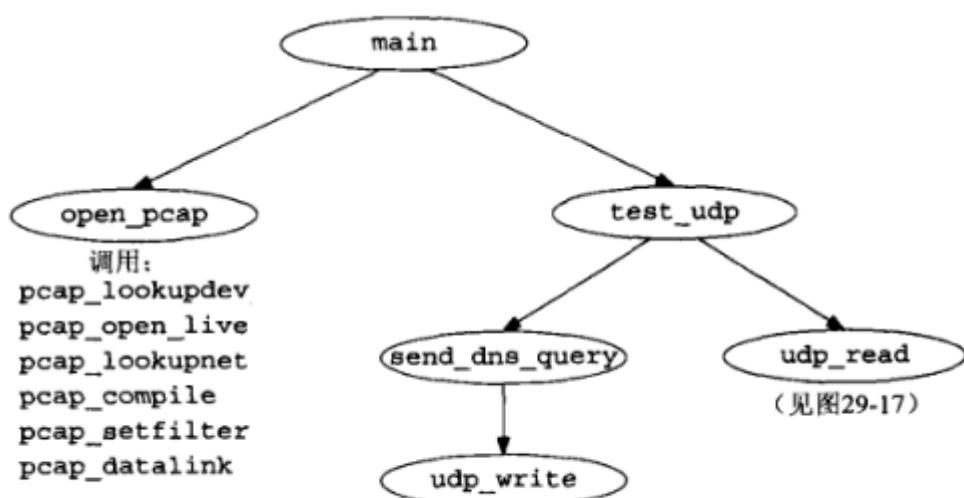


图29-4 `udpcksum`程序中的函数汇总

```

• /* include main1 */
#include      "udpcksum.h"

           /* Define global variables */
struct sockaddr *dest, *local;
  
```

```

struct sockaddr_in locallookup;
socklen_t destlen, locallen;

int          datalink;           /* from pcap_datalink(), in <net/bpf.h> */
char *device;                 /* pcap device */
pcap_t *pd;                   /* packet capture struct pointer */
int          rawfd;            /* raw socket to write on */
int          snaplen = 200;    /* amount of data to capture */
int          verbose;
int          zerosum;          /* send UDP query with no checksum */

static void usage(const char *);

int
main(int argc, char *argv[])
{
    int          c, lopt=0;
    char         *ptr, localname[1024], *localport;
    struct addrinfo *aip;
/* end main1 */

/* include main2 */
    opterr = 0;                /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "0i:l:v")) != -1) {
        switch (c) {

            case '0':
                zerosum = 1;
                break;

            case 'i':
                device = optarg;             /* pcap device */
                break;

            case 'l':                  /* local IP address and port #: a.b.c.d.p */
/*
                if ( (ptr = strrchr(optarg, '.')) == NULL)
                    usage("invalid -l option");

                *ptr++ = 0;                /* null replaces
final period */
                localport = ptr;           /* service name or port
number */
                strncpy(localname, optarg, sizeof(localname));
                lopt = 1;
                break;

            case 'v':
                verbose = 1;
                break;

            case '?':
                usage("unrecognized option");
        }
    }
}

```

```

        }

    }

/* end main2 */
/* include main3 */
    if (optind != argc-2)
        usage("missing <host> and/or <serv>");

        /* 4convert destination name and service */
    aip = Host_serv(argv[optind], argv[optind+1], AF_INET, SOCK_DGRAM);
    dest = aip->ai_addr;           /* don't freeaddrinfo() */
    destlen = aip->ai_addrlen;

    /*
     * Need local IP address for source IP address for UDP datagrams.
     * Can't specify 0 and let IP choose, as we need to know it for
     * the pseudoheader to calculate the UDP checksum.
     * If -l option supplied, then use those values; otherwise,
     * connect a UDP socket to the destination to determine the right
     * source address.
     */
    if (lopt) {
        /* 4convert local name and service */
        aip = Host_serv(localname, localport, AF_INET, SOCK_DGRAM);
        local = aip->ai_addr;           /* don't freeaddrinfo() */
        locallen = aip->ai_addrlen;
    } else {
        int s;
        s = Socket(AF_INET, SOCK_DGRAM, 0);
        Connect(s, dest, destlen);
        /* kernel chooses correct local address for dest */
        locallen = sizeof(locall lookup);
        local = (struct sockaddr *)&locall lookup;
        Getsockname(s, local, &locallen);
        if (locall lookup.sin_addr.s_addr == htonl(INADDR_ANY))
            err_quit("Can't determine local address - use -l\n");
        close(s);
    }

    open_output();          /* open output, either raw socket or libnet */

    open_pcap();           /* open packet capture device */

    setuid(getuid());      /* don't need superuser privileges anymore */

    Signal(SIGTERM, cleanup);
    Signal(SIGINT, cleanup);
    Signal(SIGHUP, cleanup);

    test_udp();

    cleanup(0);
}
/* end main3 */

```

```

static void
usage(const char *msg)
{
    err_msg(
"usage: udpcksum [ options ] <host> <serv>\n"
"options: -0      send UDP datagram with checksum set to 0\n"
"          -i s   packet capture device\n"
"          -l a.b.c.d.p local IP=a.b.c.d, local port=p\n"
"          -v      verbose output"
);

    if (msg[0] != 0)
        err_quit("%s", msg);
    exit(1);
}

```

- /* include open_pcap */

```

#include      "udpcksum.h"

#define CMD           "udp and src host %s and src port %d"

void
open_pcap(void)
{
    uint32_t             localnet, netmask;
    char                cmd[MAXLINE], errbuf[PCAP_ERRBUF_SIZE],
                        str1[INET_ADDRSTRLEN],
str2[INET_ADDRSTRLEN];
    struct bpf_program   fcode;

    if (device == NULL) {
        if ( (device = pcap_lookupdev(errbuf)) == NULL)
            err_quit("pcap_lookup: %s", errbuf);
    }
    printf("device = %s\n", device);

    /* 4hardcode: promisc=0, to_ms=500 */
    if ( (pd = pcap_open_live(device, snaplen, 0, 500, errbuf)) == NULL)
        err_quit("pcap_open_live: %s", errbuf);

    if (pcap_lookupnet(device, &localnet, &netmask, errbuf) < 0)
        err_quit("pcap_lookupnet: %s", errbuf);
    if (verbose)
        printf("localnet = %s, netmask = %s\n",
               Inet_ntop(AF_INET, &localnet, str1, sizeof(str1)),
               Inet_ntop(AF_INET, &netmask, str2, sizeof(str2)));

    snprintf(cmd, sizeof(cmd), CMD,
             Sock_ntop_host(dest, destlen),
             ntohs(sock_get_port(dest, destlen)));
    if (verbose)
        printf("cmd = %s\n", cmd);
}

```

```

    if (pcap_compile(pd, &fcode, cmd, 0, netmask) < 0)
        err_quit("pcap_compile: %s", pcap_geterr(pd));

    if (pcap_setfilter(pd, &fcode) < 0)
        err_quit("pcap_setfilter: %s", pcap_geterr(pd));

    if ( (datalink = pcap_datalink(pd)) < 0)
        err_quit("pcap_datalink: %s", pcap_geterr(pd));
    if (verbose)
        printf("datalink = %d\n", datalink);
}
/* end open_pcap */

/* include next_pcap */
char *
next_pcap(int *len)
{
    char                         *ptr;
    struct pcap_pkthdr      hdr;

    /* keep looping until packet ready */
    while ( (ptr = (char *) pcap_next(pd, &hdr)) == NULL)
        ;

    *len = hdr.caplen;      /* captured length */
    return(ptr);
}
/* end next_pcap */

```

- /* include sig_alrm */

```

#include      "udpcksum.h"
#include      <setjmp.h>

static sigjmp_buf      jmpbuf;
static int              canjump;

void
sig_alrm(int signo)
{
    if (canjump == 0)
        return;
    siglongjmp(jmpbuf, 1);
}
/* end sig_alrm */

/* include test_udp */
void
test_udp(void)
{
    volatile int    nsent = 0, timeout = 3;
    struct udphdr *ui;

    Signal(SIGALRM, sig_alrm);

```

```

    if (sigsetjmp(jmpbuf, 1)) {
        if (nsent >= 3)
            err_quit("no response");
        printf("timeout\n");
        timeout *= 2;           /* exponential backoff: 3, 6, 12 */
    }
    canjump = 1;      /* siglongjmp is now OK */

    send_dns_query();
    nsent++;

    alarm(timeout);
    ui = udp_read();
    canjump = 0;
    alarm(0);

    if (ui->ui_sum == 0)
        printf("UDP checksums off\n");
    else
        printf("UDP checksums on\n");
    if (verbose)
        printf("received UDP checksum = %x\n", ntohs(ui->ui_sum));
}
/* end test_udp */

```

- #include "udpcksum.h"

```

/*
 * Build a DNS A query for "a.root-servers.net" and write it to
 * the raw socket.
 */

/* include send_dns_query */
void
send_dns_query(void)
{
    size_t          nbytes;
    char           *buf, *ptr;

    buf = Malloc(sizeof(struct udpihdr) + 100);
    ptr = buf + sizeof(struct udpihdr);/* leave room for IP/UDP headers */

    *((uint16_t *) ptr) = htons(1234);      /* identification */
    ptr += 2;
    *((uint16_t *) ptr) = htons(0x0100);      /* flags: recursion desired */
    ptr += 2;
    *((uint16_t *) ptr) = htons(1);           /* # questions */
    ptr += 2;
    *((uint16_t *) ptr) = 0;                  /* # answer RRs */
    ptr += 2;
    *((uint16_t *) ptr) = 0;                  /* # authority RRs */
    ptr += 2;

```

```

*((uint16_t *) ptr) = 0;                                /* # additional RRs */
ptr += 2;

memcpy(ptr, "\001a\014root-servers\003net\000", 20);
ptr += 20;
*((uint16_t *) ptr) = htons(1);                         /* query type = A */
ptr += 2;
*((uint16_t *) ptr) = htons(1);                         /* query class = 1 (IP addr) */
ptr += 2;

nbytes = (ptr - buf) - sizeof(struct udphdr);
udp_write(buf, nbytes);
if (verbose)
    printf("sent: %d bytes of data\n", nbytes);
}

/* end send_dns_query */

```

- ```

#include "udpcsum.h"

/* include open_output_raw */
int rawfd; /* raw socket to write on */

void
open_output(void)
{
 int on=1;
 /*
 * Need a raw socket to write our own IP datagrams to.
 * Process must have superuser privileges to create this socket.
 * Also must set IP_HDRINCL so we can write our own IP headers.
 */
 rawfd = Socket(dest->sa_family, SOCK_RAW, 0);

 Setsockopt(rawfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
}
/* end open_output_raw */

/*
 * "buf" points to an empty IP/UDP header,
 * followed by "ulen" bytes of user data.
 */

/* include udp_write */
void
udp_write(char *buf, int userlen)
{
 struct udphdr *ui;
 struct ip *ip;

 /* 4fill in and checksum UDP header */
 ip = (struct ip *) buf;
 ui = (struct udphdr *) buf;

```

```

 bzero(ui, sizeof(*ui));
 /* 8add 8 to userlen for pseudoheader length */
 ui->ui_len = htons((uint16_t) (sizeof(struct udphdr) + userlen));
 /* 8then add 28 for IP datagram length */
 userlen += sizeof(struct udphdr);

 ui->ui_pr = IPPROTO_UDP;
 ui->ui_src.s_addr = ((struct sockaddr_in *) local)->sin_addr.s_addr;
 ui->ui_dst.s_addr = ((struct sockaddr_in *) dest)->sin_addr.s_addr;
 ui->ui_sport = ((struct sockaddr_in *) local)->sin_port;
 ui->ui_dport = ((struct sockaddr_in *) dest)->sin_port;
 ui->ui_ulen = ui->ui_len;
 if (zerosum == 0) {
#if 1 /* change to if 0 for Solaris 2.x, x < 6 */
 if ((ui->ui_sum = in_cksum((u_int16_t *) ui, userlen)) == 0)
 ui->ui_sum = 0xffff;
#else
 ui->ui_sum = ui->ui_len;
#endif
 }

 /* 4fill in rest of IP header; */
 /* 4ip_output() calcuates & stores IP header checksum */
 ip->ip_v = IPVERSION;
 ip->ip_hl = sizeof(struct ip) >> 2;
 ip->ip_tos = 0;
#if defined(linux) || defined(__OpenBSD__)
 ip->ip_len = htons(userlen); /* network byte order */
#else
 ip->ip_len = userlen; /* host byte order */
#endif
 ip->ip_id = 0; /* let IP set this */
 ip->ip_off = 0; /* frag offset, MF and DF flags */
 ip->ip_ttl = TTL_OUT;

 Sendto(rawfd, buf, userlen, 0, dest, destlen);
 }
/* end udp_write */

```

- #include "udpcksum.h"

```

struct udphdr *udp_check(char *, int);

/*
 * Read from the network until a UDP datagram is read that matches
 * the arguments.
 */

/* include udp_read */
struct udphdr *
udp_read(void)
{
 int

```

```

 char *ptr;
 struct ether_header *eptr;

 for (; ;) {
 ptr = next_pcap(&len);

 switch (datalink) {
 case DLT_NULL: /* loopback header = 4 bytes */
 return(udp_check(ptr+4, len-4));

 case DLT_EN10MB:
 eptr = (struct ether_header *) ptr;
 if (ntohs(eptr->ether_type) != ETHERTYPE_IP)
 err_quit("Ethernet type %x not IP", ntohs(eptr-
>ether_type));
 return(udp_check(ptr+14, len-14));

 case DLT_SLIP: /* SLIP header = 24 bytes */
 return(udp_check(ptr+24, len-24));

 case DLT_PPP: /* PPP header = 24 bytes */
 return(udp_check(ptr+24, len-24));

 default:
 err_quit("unsupported datalink (%d)", datalink);
 }
 }
 }

/* end udp_read */

/*
 * Check the received packet.
 * If UDP and OK, return pointer to packet.
 * If ICMP error, return NULL.
 * We assume the filter picks out desired UDP datagrams.
 */

/* include udp_check */
struct udiphdr *
udp_check(char *ptr, int len)
{
 int hlen;
 struct ip *ip;
 struct udiphdr *ui;
/* *INDENT-OFF* */

 if (len < sizeof(struct ip) + sizeof(struct udphdr))
 err_quit("len = %d", len);
/* *INDENT-ON* */

 /* minimal verification of IP header */
 ip = (struct ip *) ptr;
 if (ip->ip_v != IPVERSION)

```

```

 err_quit("ip_v = %d", ip->ip_v);
 hlen = ip->ip_hl << 2;
/* *INDENT-OFF* */
 if (hlen < sizeof(struct ip))
 err_quit("ip_hl = %d", ip->ip_hl);
 if (len < hlen + sizeof(struct udphdr))
 err_quit("len = %d, hlen = %d", len, hlen);
/* *INDENT-ON* */

 if ((ip->ip_sum = in_cksum((uint16_t *) ip, hlen)) != 0)
 err_quit("ip checksum error");

 if (ip->ip_p == IPPROTO_UDP) {
 ui = (struct udiphdr *) ip;
 return(ui);
 } else
 err_quit("not a UDP packet");
}
/* end udp_check */

```

## 29.8 总结

- 原始套接字使得我们有能力读写内核不理解的IP数据报，数据链路层访问则把这个能力进一步扩展成读与写任何类型的数据链路帧，而不仅仅是IP数据报，tcpdump也许是直接访问数据链路层的最常用程序

## 第30章 客户/服务器程序设计范式

- 9个不同的服务器程序设计范式

| 行 号 | 服务器描述                      | 进程控制CPU时间（秒数，与基准之差） |       |        |       |
|-----|----------------------------|---------------------|-------|--------|-------|
|     |                            | Solaris             | DUnix | BSD/OS | 第3版   |
| 0   | 迭代服务器（测量基准，无进程控制）          | 0.0                 | 0.0   | 0.0    | 0.0   |
| 1   | 并发服务器，为每个客户请求fork一个进程      | 504.2               | 168.9 | 29.6   | 20.90 |
| 2   | 预先派生子进程，每个子进程调用accept      |                     | 6.2   | 1.8    | 1.80  |
| 3   | 预先派生子进程，以文件上锁方式保护accept    | 25.2                | 10.0  | 2.7    | 2.07  |
| 4   | 预先派生子进程，以线程互斥锁上锁方式保护accept | 21.5                |       |        | 1.75  |
| 5   | 预先派生子进程，由父进程向子进程传递套接字描述符   | 36.7                | 10.9  | 6.1    | 2.58  |
| 6   | 并发服务器，为每个客户请求创建一个线程        | 18.7                | 4.7   |        | 0.99  |
| 7   | 预先创建线程，以互斥锁上锁方式保护accept    | 8.6                 | 3.5   |        | 1.93  |
| 8   | 预先创建线程，由主线程调用accept        | 14.5                | 5.0   |        | 2.05  |

图30-1 本章所讨论各个范式服务器的测时结果比较

- 迭代服务器
- 并发服务器，每个客户请求fork一个子进程
- 预先派生子进程，每个子进程无保护地调用accept
- 预先派生子进程，使用文件上锁保护accept
- 预先派生子进程，使用线程互斥锁上锁保护accept
- 预先派生子进程，父进程向子进程传递套接字描述符

- 并发服务器，每个客户请求创建一个线程
- 预先创建线程服务器，使用互斥锁上锁保护accept
- 预先创建线程服务器，由主线程调用accept
  - 当系统负载较轻时，每来一个客户请求现场派生一个子进程为之服务的传统并发服务器程序模型就足够了。这个模型甚至可以与inetd结合使用，也就是inetd处理每个连接的接受。我们的其他意见是就重负荷运行的服务器而言的，譬如Web服务器。
  - 相比传统的每个客户fork一次设计范式，预先创建一个子进程池或一个线程池的设计范式能够把进程控制CPU时间降低10倍或以上。编写这些范式的程序并不复杂，不过需超越本章所给例子的是：监视闲置子进程个数，随着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同一个accept调用中，另一些实现却要求包绕accept调用安置某种类型的锁加以保护。文件上锁或Pthread互斥锁上锁都可以使用。
- 让所有子进程或线程自行调用accept通常比让父进程或主线程独自调用accept并把描述符传递给子进程或线程来得简单而快速。
- 由于潜在select冲突的原因，让所有子进程或线程阻塞在同一个accept调用中比让它们阻塞在同一个select调用中更可取。
- 使用线程通常远快于使用进程。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支持，还可能取决于为服务每个客户需激活其他什么程序（若有其他程序需激活的话）。举例来说，如果accept客户连接的服务器调用fork和exec（譬如说inetd超级守护进程），那么fork一个单线程的进程可能快于fork一个多线程的进程。

## 第31章 流

---