第23章 UDP：用户数据报协议

23.1 引言

用户数据报协议，即UDP，是一个面向数据报的简单运输层协议：进程的每次输出操作只产生一个UDP数据报，从而发送一个IP数据报。

进程通过创建一个Internet域内的SOCK_DGRAM类型的插口，来访问UDP。该类插口默认地称为无连接的(unconnected)。每次进程发送数据报时，必须指定目的IP地址和端口号。每次从插口上接收数据报时，进程可以从数据报中收到源IP地址和端口号。

我们在22.5节中提到，UDP插口也可以被连接到一个特殊的IP地址和端口号。这样，所有写到该插口上的数据报都被发往该目的地，而且只有来自该IP地址和端口号的数据报才被传给该进程。

本章讨论UDP的实现。

23.2 代码介绍

9个UDP函数在一个C文件中，2个UDP定义的头文件，如图23-1所示。

图23-2显示了6个主要的UDP函数与其他内核函数之间的关系。带阴影的椭圆是本章我们讨论的6个函数，另外还有其他3个函数是这6个函数经常调用的。

文　件	描　述
netinet/udp.h netinet/udp_var.h	udphdr结构定义 其他UDP定义
netinet/udp_usrreq.c	UDP函数

图23-1 本章中讨论的文件

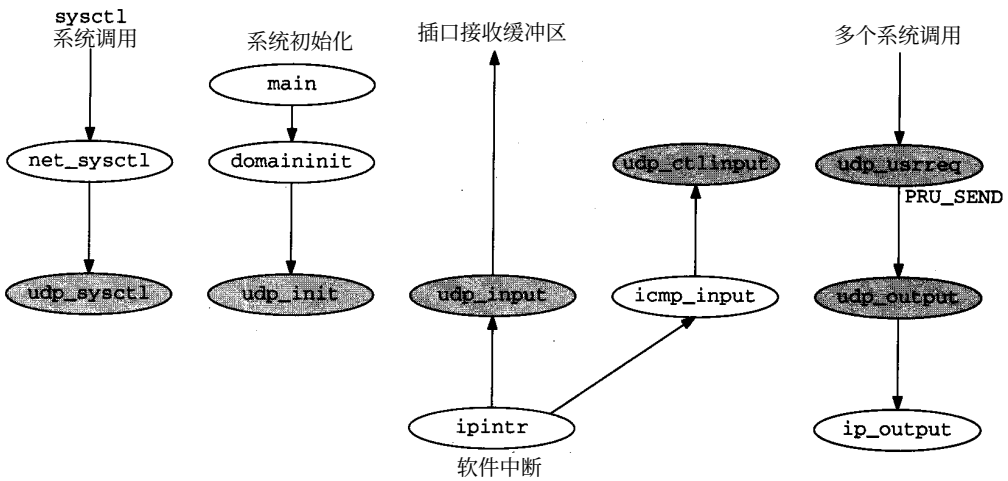

图23-2 UDP函数与内核其他函数之间的关系

23.2.1 全局变量

本章引入的全局变量，如图22-3所示。

变 量	数据类型	描 述
udb	Struct inpcb	UDP PCB表的表头
udp_last_inpcb	Struct inpcb *	指向最近收到的数据报的指针：“向后一个”高速缓存
udpcksum	Int	用于计算和验证UDP检验和的标志位
udp_in	Struct sockaddr_in	在输入时存放发送方的IP地址
udpstat	Struct udpstat	UDP统计(图23-4)
udp_recvspace	u_long	插口接收缓存的默认大小，41 600字节
udp_sendspace	u_long	插口发送缓存的默认大小，9 216字节

图23-3 本章中引入的全局变量

23.2.2 统计量

全局结构udpstat维护多种UDP统计量，如图23-4所示。讨论代码的过程中，我们会看到何时增加这些计数器的值。

udpstat成员	描 述	SNMP使用的
udps_badlen	收到所有数据长度大于分组的数据报个数	•
udps_badsum	收到有检验和错误的数据报个数	•
udps_fullsock	收到由于输入插口已满而没有提交的数据报个数	
udps_hdrops	收到分组小于首部的数据报个数	•
udps_ipackets	所有收到的数据报个数	•
udps_noport	收到在目的端口没有进程的数据报个数	•
udps_noportbcast	收到在目的端口没有进程的广播／多播数据报个数	•
udps_opackets	全部输出数据报的个数	•
udps_pcbcachemiss	收到的丢失pcb高速缓存的输入数据报个数	

图23-4 在udpstat 结构中维持的UDP统计

图23-5显示了执行netstat -s后输出的统计信息。

netstat -s 输出	udpstat 成员
18,575,142 datagrams received	udps_ipackets
0 with incomplete header	udps_hdrops
18 with bad data length field	udps_badlen
58 with bad checksum	udps_badsum
84,079 dropped due to no socket	udps_noport
446 broadcast/multicast datagrams dropped due to no socket	udps_noportbcast
5,356 dropped due to full socket buffers	udps_fullsock
18,485,185 delivered	(见正文)
18,676,277 datagrams output	udps_opackets

图23-5 UDP统计样本

提交的UDP数据报的个数(输出的倒数第二行)是收到的数据报总数(udps_ipackets)减去图23-5中它前面的6个变量。

23.2.3 SNMP变量

图23-6显示了UDP组中的四个简单SNMP变量，这四个变量在实现该变量的udpstat结构中计数。

图23-7显示了UDP监听器表，称为udpTable。SNMP为这个表返回的值是取自UDP PCB，而不是udpstat结构。

SNMP变量	udpstat成员	描　述
udpInDatagrams	udps_ipackets	收到的所有提交给进程的数据报个数
udpInErrors	udps_hdrops + udps_badsum+ udps_badlen	收到的由于某些原因不可提交的UDP数据报个数，这些原因中不包括在目的端口没有应用程序的原因(例如，UDP检验和差错)
udpNoPorts	udps_noport + udps_noportbcast	收到的所有目的端口没有应用进程的数据报
udpOutDatagrams	udps_opackets	发送的数据报的个数

图23-6 udp组中的简单SNMP变量

UDP监听器表，索引=<*udpLocalAddress*>.<*udpLocalPort*>		
SNMP变量	PCB变量	描　述
udpLocalAddress	inp_laddr	这个监听器的本地IP
udpLocalPort	inp_lport	这个监听器的本地端口号

图23-7 UDP监听器表：udpTable

23.3 UDP的protosw结构

图23-8显示了UDP的协议交换入口

成　员	inetsw[1]	描　述
pr_type	*SOCK_DGRAM*	UDP提供数据报分组服务
pr_domain	*&inetdomain*	UDP是Internet域的一部分
pr_protocol	*IPPROTO_UDP (17)*	出现在IP首部的ip_p字段
pr_flags	*PR_ATOMIC|PR_ADDR*	插口层标志，协议处理没有使用
pr_input	*Udp_input*	从IP层接收报文
pr_output	*0*	UDP没有使用
pr_ctlinput	*udp_ctlinput*	ICMP差错的控制输入函数
pr_ctloutput	*ip_ctloutput*	响应来自进程的管理请求
pr_usrreq	*udp_usrreq*	响应来自进程的通信请求
pr_init	*udp_init*	初始化UDP
pr_fasttimo	*0*	UDP没有使用
pr_slowtimo	*0*	UDP没有使用
pr_drain	*0*	UDP没有使用
pr_sysctl	*udp_sysctl*	对sysctl(8)系统调用

图23-8 UDP的protosw 结构

本章我们描述五个以udp_开头的函数。另外我们还要介绍第6个函数udp_output，它

不在协议交换入口，但当输出一个UDP数据报时，udp_usrreq会调用它。

23.4 UDP的首部

UDP首部定义成一个udphdr结构。图23-9是C结构，图23-10是UDP首部的图。

udp.h

```
39 struct udphdr {
40     u_short uh_sport;          /* source port */
41     u_short uh_dport;          /* destination port */
42     short   uh_ulen;           /* udp length */
43     u_short uh_sum;            /* udp checksum */
44 };
```

udp.h

图23-9 udphdr 结构

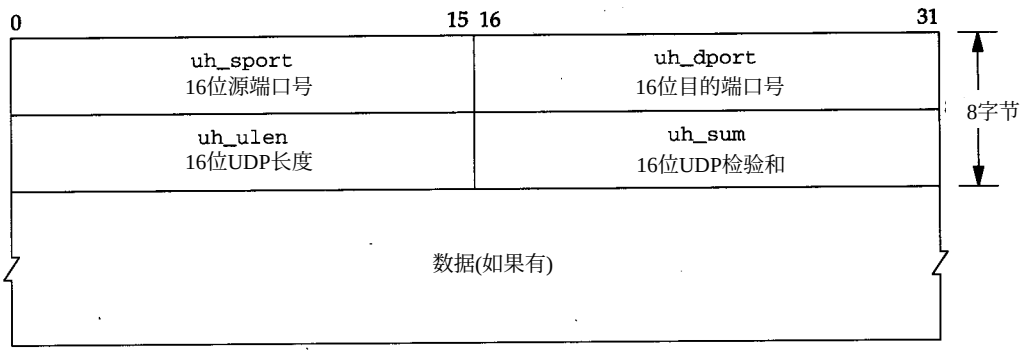

图23-10 UDP首部和可选数据

在源代码中，通常把UDP首部作为一个紧跟着UDP首部的IP首部来引用。这就是udp_input如何处理收到的IP数据报，以及udp_output如何构造外出的IP数据报。这种联合的IP/UDP首部是一个udpiphdr结构，如图23-11所示。

udp_var.h

```
38 struct udpiphdr {
39     struct ipovly ui_i;        /* overlaid ip structure */
40     struct udphdr ui_u;        /* udp header */
41 };
42 #define ui_next      ui_i.ih_next
43 #define ui_prev      ui_i.ih_prev
44 #define ui_x1        ui_i.ih_x1
45 #define ui_pr        ui_i.ih_pr
46 #define ui_len       ui_i.ih_len
47 #define ui_src       ui_i.ih_src
48 #define ui_dst       ui_i.ih_dst
49 #define ui_sport     ui_u.uh_sport
50 #define ui_dport     ui_u.uh_dport
51 #define ui_ulen      ui_u.uh_ulen
52 #define ui_sum       ui_u.uh_sum
```

udp_var.h

图23-11 udpiphdr 结构：联合的IP/UDP首部

20字节的IP首部定义成一个ipovly结构，如图23-12所示。

不幸的是，这个结构并不是一个真正的如图 8-8所示的IP首部。大小相同(20字节)，但字

段不同。我们将在23.6节讲UDP检验和的计算时回来讨论这个不同之处。

ip_var.h

```
38 struct ipovly {
39     caddr_t ih_next, ih_prev;    /* for protocol sequence q's */
40     u_char  ih_x1;               /* (unused) */
41     u_char  ih_pr;               /* protocol */
42     short   ih_len;              /* protocol length */
43     struct in_addr ih_src;       /* source internet address */
44     struct in_addr ih_dst;       /* destination internet address */
45 };
```

ip_var.h

图23-12 ipovly 结构

23.5 udp_init函数

domaininit函数在系统初始化时调用UDP的初始化函数(udp_init，图23-13)。

这个函数所做的唯一的工作是把头部PCB(udb)的向前和向后指针指向它自己。这是一个双向链表。

udb PCB的其他部分都被初始化成 0，尽管在这个头部 PCB中唯一使用的字段是inp_lport，它是要分配的下一个UDP临时端口号。在解习题22.4时，我们提到，因为这个本地端口号被初始化成0，所以第一个临时端口号将是1024。

udp_usrreq.c

```
50 void
51 udp_init()
52 {
53     udb.inp_next = udb.inp_prev = &udb;
54 }
```

udp_usrreq.c

图23-13 udp_init 函数

23.6 udp_output函数

当应用程序调用以下五个写函数中的任意一个时，发生UDP 输出。这五个函数是：send、sendto、sendmsg、write或writev。如果插口已连接上的，则可任意调用这五个函数，尽管用sendto或sendmsg不能指定目的地址。如果插口没有连接上，则只能调用 sendto和sendmsg，并且必须指定一个目的地址。图23-14总结了这五个函数，它们在终止时，都调用udp_output，该函数再调用ip_output。

五个函数终止调用sosend，并把一个指向msghdr结构的指针作为参数传给该函数。要输出的数据被分装在一个 mbuf链上，sosend把一个可选的目的地址和可选的控制信息放到mbuf中，并发布PRU_SEND请求。

UDP调用函数udp_output，该函数的第一部分如图23-15所示。四个参数分别是：inp，指向插口Internet PCB的指针；m，指向输出mbuf链的指针；addr，一个可选指针，指向某个mbuf，存放分装在一个sockaddr_in结构中的目的地址；control，一个可选指针，指向一个mbuf，其中存放着sendmsg中的控制信息。

1. 丢掉可选控制信息

333-344 m_freem丢弃可选的控制信息，不产生差错。UDP输出不使用任何控制信息。

注释*xxx*是因为忽略控制信息且不产生错误。其他协议如路由选择域和 TCP，当进程传递控制信息时，都会产生一个错误。

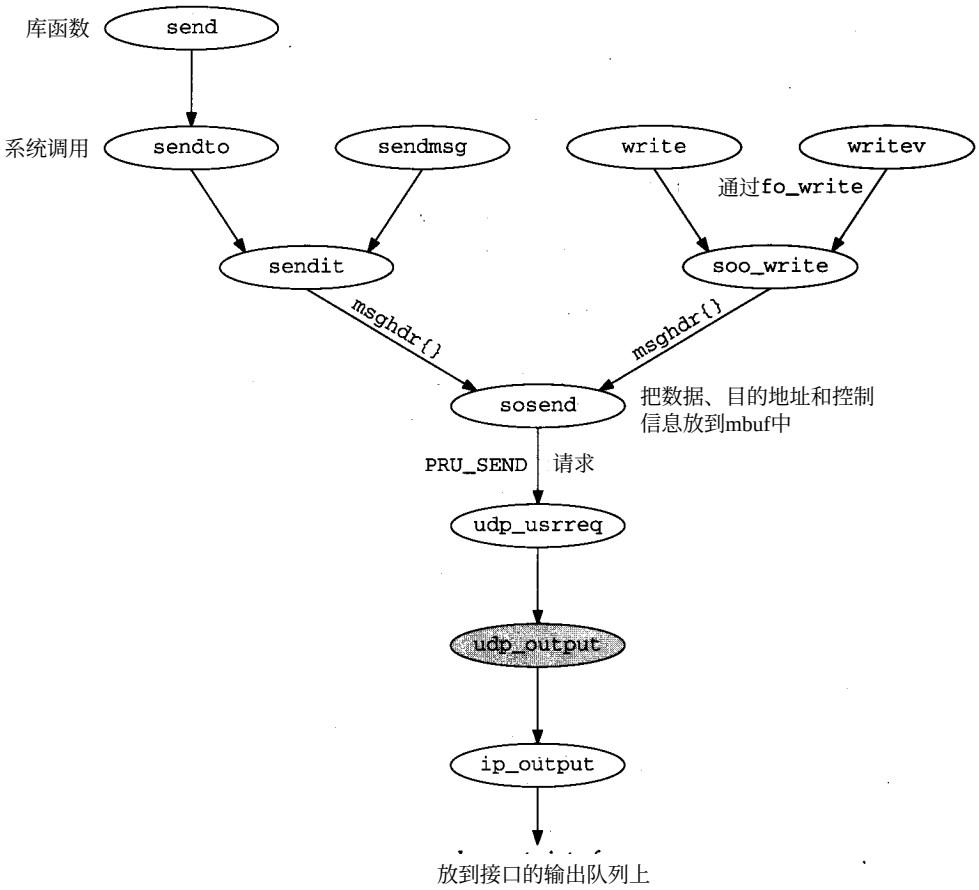

图23-14 五个写函数如何终止调用udp_output

2. 临时连接一个未连接上的插口

345-359 如果调用方为UDP数据报指定了目的地址(addr非空)，则插口是由in_pcbconnect临时连接到该目的地址的，并在该函数的最后被断连。在连接之前，要作一个检测，判断插口是否已经连接上。如果已连接上，则返回错误 EISCONN。这就是为什么sendto在已连接上的插口上指定目的地址时，会返回错误。

在临时连接插口之前，splnet停止IP的输入处理。这样做的原因是因为，临时连接将改变插口PCB中的外部地址、外部端口以及本地地址。如果在临时连接该 PCB的过程中处理某个收到的UDP数据报，可能把该数据报提交给错误的进程。把处理器设置成比 splnet优先，只能阻止软件中断引发执行IP输入程序(图1-12)，它不能阻止接口层接收进入的分组，并把它们放到IP的输入队列中。

[Partridge和Pink 1993] 注意到临时连接插口的这个操作开销很大，用去每个UDP传送将近三分之一的时间。

在临时连接之前，PCB的本地地址被保存在laddr中，因为如果它是通配地址，它将被

in_pcbconnect在调用in_pcbbind时改变。

如果进程调用了connect，则应用于目的地址的同一规则也将适用，因为两种情况都将调用in_pcbconnect。

udp_usrreq.c

```
333 int
334 udp_output(inp, m, addr, control)
335 struct inpcb *inp;
336 struct mbuf *m;
337 struct mbuf *addr, *control;
338 {
339     struct udpiphdr *ui;
340     int     len = m->m_pkthdr.len;
341     struct in_addr laddr;
342     int     s, error = 0;
343     if (control)
344         m_freem(control);       /* XXX */
345     if (addr) {
346         laddr = inp->inp_laddr;
347         if (inp->inp_faddr.s_addr != INADDR_ANY) {
348             error = EISCONN;
349             goto release;
350         }
351         /*
352          * Must block input while temporarily connected.
353          */
354         s = splnet();
355         error = in_pcbconnect(inp, addr);
356         if (error) {
357             splx(s);
358             goto release;
359         }
360     } else {
361         if (inp->inp_faddr.s_addr == INADDR_ANY) {
362             error = ENOTCONN;
363             goto release;
364         }
365     }
366     /*
367      * Calculate data length and get an mbuf for UDP and IP headers.
368      */
369     M_PREPEND(m, sizeof(struct udpiphdr), M_DONTWAIT);
370     if (m == 0) {
371         error = ENOBUFS;
372         goto release;
373     }

            /* remainder of function shown in Figure 23.20 */

409   release:
410     m_freem(m);
411     return (error);
412 }
```

udp_usrreq.c

图23-15 udp_output 函数：临时连接一个未连接上的插口

360-364 如果进程没有指定目的地址，并且插口没有连接上，则返回ENOTCONN。

3. 在前面加上IP/UDP首部

366-373 M_PREPEND在数据的前面为IP和UDP首部分配空间。图1-8是一种情况，假定mbuf链上的第一个mbuf已经没有空间存放首部的28个字节。习题23.1详细给出了其他情况。需要指定标志位M_DONTWAIT，因为如果插口是临时连接的，则IP处理被阻塞，所以M_PREPEND也应被阻塞。

早期的伯克利版本不正确地指定了这里的M_WAIT。

23.6.1 在前面加上IP/UDP首部和mbuf簇

在M_PREPEND宏和mbuf簇之间有一个微妙的交互。如果sosend把用户数据放到一个簇中，则该簇的最前面的56个字节(max_hdr，图7-17)没有使用，这就为以太网、IP和UDP首部提供了空间。避免M_PREPEND仅仅为存放这些首部而另外再分配一个mbuf。M_PREPEND调用M_LEADINGSPACE来计算在mbuf的前面有多大的空间可以使用：

```
#define M_LEADINGSPACE(m) \
   ((m)->m_flags & M_EXT ? /* (m)->m_data - (m)->m_ext_buf */ 0 : \
        (m)->m_flags & M_PKTHDR ? (m)->m_data - (m)->m_pktdat : \
        (m)->m_data - (m)->m_dat)
```

正确地计算出簇前面可用空间大小的代码被注释掉了，如果数据在簇内，该宏总是返回0。这意味着，当用户数据也在某个簇中时，M_PREPEND总是为协议首部分配一个新的mbuf，而不再使用sosend分配的用于存放首部的空间。

M_LEADINGSPACE中注释掉正确代码的原因是因为该簇可能被共享(2.9节)，而且，如果它被共享，使用簇中数据报前面的空间可能会擦掉其他数据。

UDP数据不共享簇，因为udp_output不保存数据的备份。但是TCP在它的发送缓存内保存数据备份(等待对该数据的确认)，而且如果数据不在簇内，则说明它是共享的。但tcp_output不调用M_LEADINGSPACE，因为sosend只为数据报协议在簇前面留56个字节，所以，tcp_output总是调用MGETHDR为协议首部分配一个mbuf。

23.6.2 UDP检验和计算和伪首部

在讨论udp_output的后一部分之前，我们描述一下UDP如何填充IP/UDP首部的某些字段，如何计算UDP检验和，以及如何传递IP/UDP首部及数据给IP输出的。这些工作很巧妙地使用了ipovly结构。

图23-16显示了udp_output在由m指向的mbuf链的第一个存储器上构造的28字节IP/UDP首部。没有阴影的字段是udp_output填充的，有阴影的字段是ip_output填充的。这个图显示了首部在线路上的格式。

UDP检验和的计算覆盖了三个区域：(1)一个12字节的伪首部，其中包含IP首部的字段；(2)8字节UDP首部，和(3)UDP数据。图23-17显示了用于检验和计算的12字节伪首部，以及UDP首部。用于计算检验和的UDP首部等价于出现在线路上的UDP首部(图23-16)。

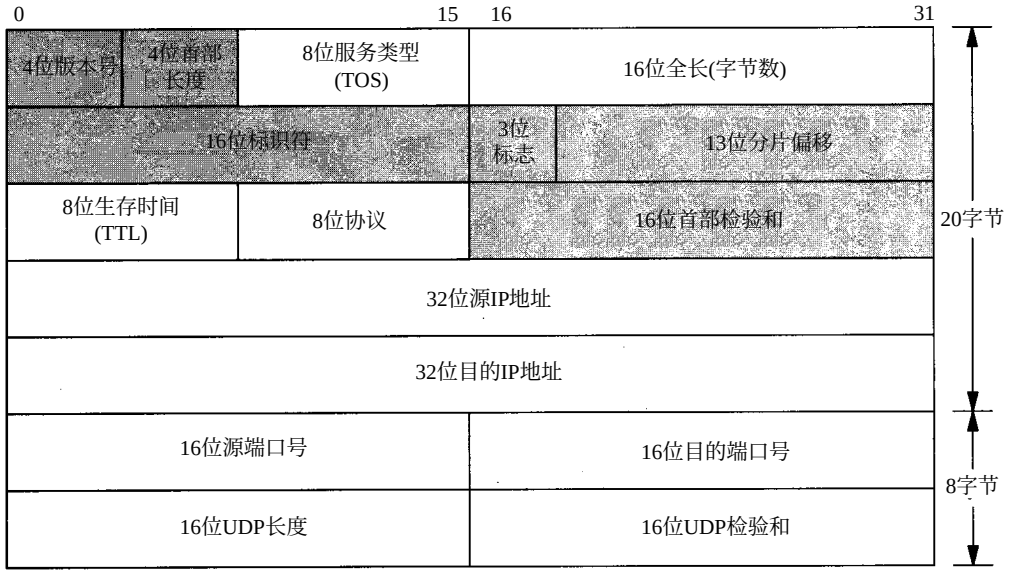

图23-16 IP/UDP首部：UDP填充没有阴影的字段，IP填充有阴影的字段

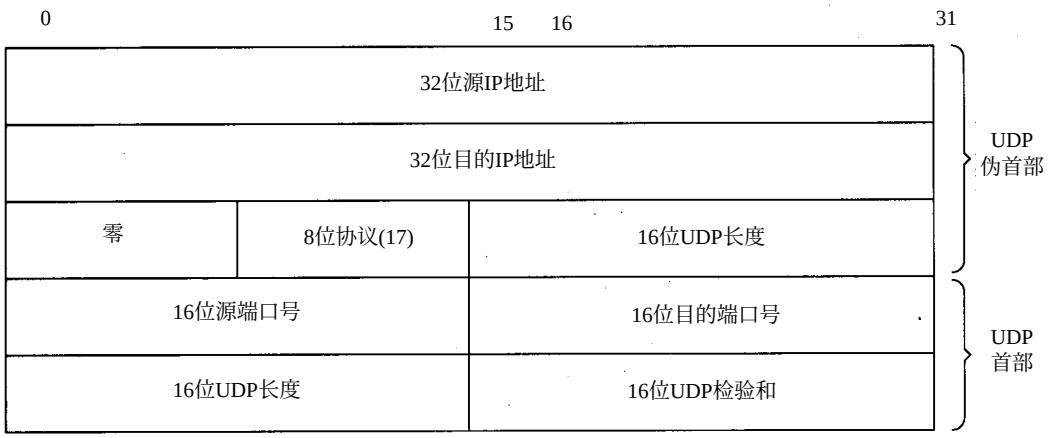

图23-17 检验和计算所使用的伪首部和UDP首部

在计算UDP检验和时使用以下三个事实：(1)在伪首部(图23-17)中的第三个32 bit字看起来与IP首部(图23-16)中的第三个32 bit字类似：两个8 bit值和一个16 bit值。(2)伪首部中三个32 bit值的顺序是无关的。事实上，Internet检验和的计算不依赖于所使用的16 bit值的顺序(8.7节)。(3)在检验和计算中另外再加上一个全0的32 bit字没有任何影响。

`udp_output`利用这三个事实，填充`udpiphdr`结构(图23-11)的字段，如图23-18所示。该结构包含在由`m`指向的mbuf链的第一个mbuf中。

在20字节IP首部(5个成员：`ui_x1`、`ui_pr`、`ui_len`、`ui_src`和`ui_dst`)中的最后三个32 bit字被用作检验和计算的伪首部。IP首部的前两个32 bit字(`ui_next`和`ui_prev`)也用在检验和计算中，但它们被初始化成0，所以不影响最后的检验和。

图23-19总结了我们描述的操作：

1) 图23-19中最上面的图是伪首部的协议定义，与图23-17对应。

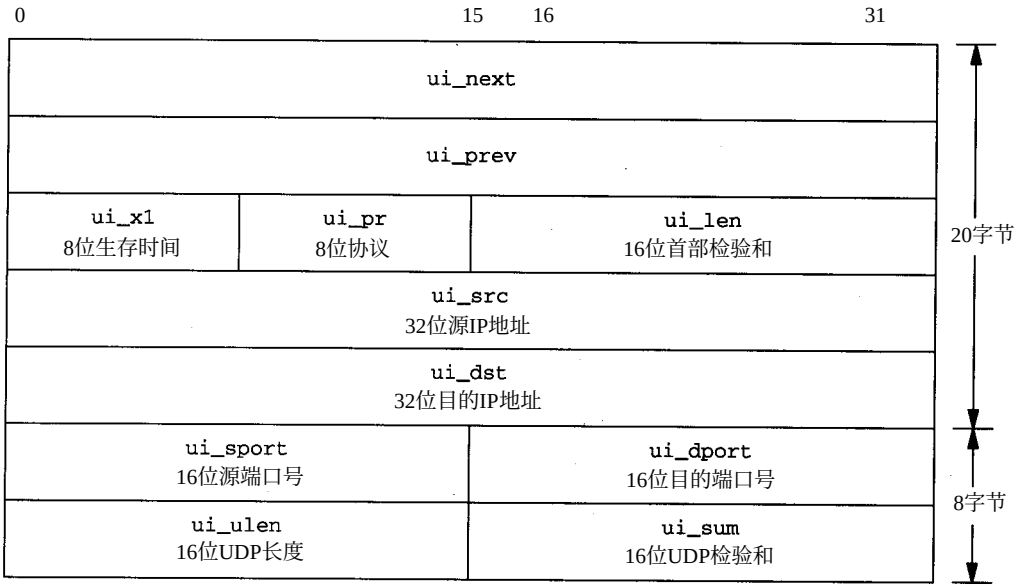

图23-18 udp_output 填充的udpiphdr

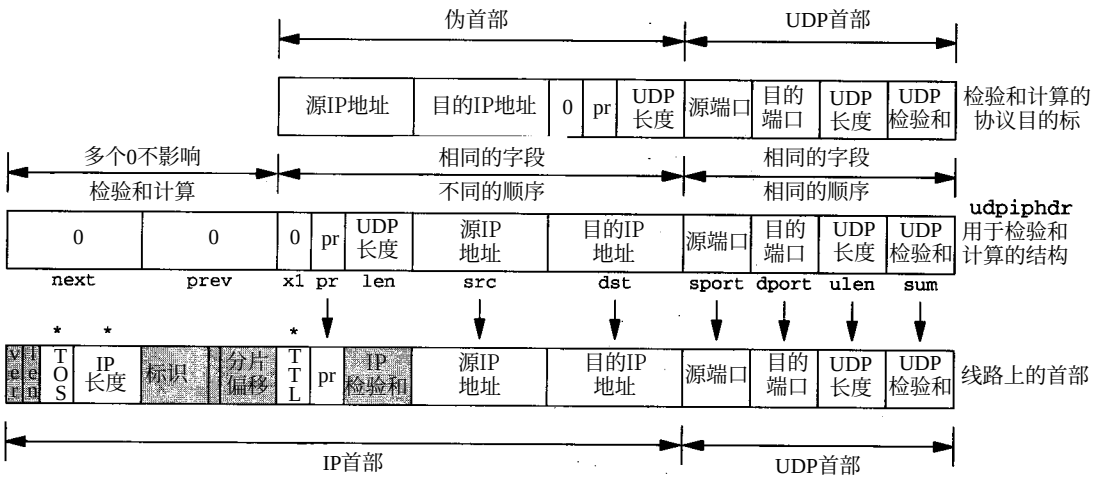

图23-19 填充IP/UDP首部和计算UDP检验和的操作

2) 中间的图是源代码使用的udpiphdr结构，与图23-11对应(为图的可读性，省略了所有成员的前缀ui_)。这是udp_output在mbuf链上的第一个mbuf中构造的结构，然后被用于计算UDP检验和。

3) 下面的图是出现在线路上的IP/UDP首部，与图23-16对应。上面有箭头的7个字段是udp_output在检验和计算之前填充的。上面有星号的3个字段是udp_output在检验和计算之后填充的。其他6个有阴影的字段是ip_output填充的。

图23-20是udp_output函数的后半部分。

1. 为检验和计算准备伪首部

374-387 把udpiphdr结构(图23-18)的所有成员设置成它们相应的值。PCB中的本地和外部插口已经是网络字节序，但必须把UDP的长度转换成网络字节序。UDP的长度是数据报的

字节数(len，可以是0)加上UDP 首部的大小(8)。UDP 长度字段在UDP 检验和计算中出现了两次：ui_len和ui_ulen。有一个是冗余的。

udp_usrreq.c

```
374     /*
375      * Fill in mbuf with extended UDP header
376      * and addresses and length put into network format.
377      */
378     ui = mtod(m, struct udpiphdr *);
379     ui->ui_next = ui->ui_prev = 0;
380     ui->ui_x1 = 0;
381     ui->ui_pr = IPPROTO_UDP;
382     ui->ui_len = htons((u_short) len + sizeof(struct udphdr));
383     ui->ui_src = inp->inp_laddr;
384     ui->ui_dst = inp->inp_faddr;
385     ui->ui_sport = inp->inp_lport;
386     ui->ui_dport = inp->inp_fport;
387     ui->ui_ulen = ui->ui_len;

388     /*
389      * Stuff checksum and output datagram.
390      */
391     ui->ui_sum = 0;
392     if (udpcksum) {
393         if ((ui->ui_sum = in_cksum(m, sizeof(struct udpiphdr) + len)) == 0)
394                 ui->ui_sum = 0xffff;
395     }
396     ((struct ip *) ui)->ip_len = sizeof(struct udpiphdr) + len;
397     ((struct ip *) ui)->ip_ttl = inp->inp_ip.ip_ttl;     /* XXX */
398     ((struct ip *) ui)->ip_tos = inp->inp_ip.ip_tos;     /* XXX */
399     udpstat.udps_opackets++;
400     error = ip_output(m, inp->inp_options, &inp->inp_route,
401             inp->inp_socket->so_options & (SO_DONTROUTE | SO_BROADCAST),
402                     inp->inp_moptions);

403     if (addr) {
404         in_pcbdisconnect(inp);
405         inp->inp_laddr = laddr;
406         splx(s);
407     }
408     return (error);
```

udp_usrreq.c

图23-20 udp_output 函数：填充首部、计算检验和并传给IP

2. 计算检验和

388-395 计算检验和时，首先把它设成0，然后调用in_cksum。如果UDP检验和是禁止的(一个坏的想法 —— 见卷1的11.3节)，则检验和的结果是0。如果计算的检验和是0，则在首部中保存16个1，而不是0(在求补运算中，全1和全0都是0)。这样，接收方就可以区分没有检验和的UDP分组(检验和字段为0)和有检验和的UDP分组了，后者的检验和值为0(16位的检验和是16个1)。

变量udpcksum(图23-3)通常默认值为1，使能UDP检验和。对内核的编译可对4.2BSD兼容，把udpcksum初始化为0。

3. 填充UDP长度、TTL和TOS

396-398 指针ui指向一个指向某个标准IP首部的指针(ip)，UDP设置IP首部内的三个字段。IP长度字段等于UDP数据报中数据的个数加上IP/UDP首部大小28。注意，IP首部的这个字段

以主机字节序保存，不象首部其他多字节字段，是以网络字节序保存的。ip_output在发送之前，把它转换成网络字节序。

把IP首部里的TTL和TOS字段的值设成插口PCB中的值。在创建插口时，UDP设置这些默认值，进程可用setsockopt改变它们。因为这三个字段——IP长度、TTL和TOS——不是伪首部的内容，UDP检验和计算时也没有用到它们，所以，在计算检验和之后，调用ip_output之前，必须设置它们。

4. 发送数据报

400-402 ip_output发送数据报。第二个参数inp_options，是进程可用setsockopt设置的IP选项。这些IP选项是ip_output放置到IP首部中的。第三个参数是一个指向高速缓存在PCB中的路由的指针，第四个参数是插口选项。传给ip_output的唯一插口选项是SO_DONTROUTE(旁路选路表)和SO_BROADCAST(允许广播)。最后一个参数是一个指向该插口的多播选项的指针。

5. 断连临时连接的插口

403-407 如果插口是临时连接上的，则in_pcbdisconnect断连插口，本地IP地址在PCB中恢复，恢复中断级别到保存的值。

23.7 udp_input函数

进程调用五个写函数之一来驱动UDP输出。图23-14显示的函数都作为系统调用的组成部分被直接调用。另一方面，当IP在它的协议字段指定为UDP的输入队列上收到一个IP数据报时，才发生UDP的输入。IP通过协议交换表(图8-15)中的pr_input函数调用函数udp_input。因为IP的输入是在软件中断级，所以udp_input也在这一级上执行。udp_input的目标是把UDP数据报放置到合适的插口的缓存内，唤醒该插口上因输入阻塞的所有进程。

我们对udp_input函数的讨论分三个部分：

1) UDP对收到的数据报完成一般性的确认；

2) 处理目的地是单播地址的UDP数据报：找到合适的PCB，把数据报放到插口的缓存内，

3) 处理目的地是广播或多播地址的UDP数据报：必须把数据报提交给多个插口。

最后一步是新的，是为了在Net/3中支持多播，但占用了大约三分之一的代码。

23.7.1 对收到的UDP数据报的一般确认

图23-21是UDP输入的第一部分。

55-65 udp_input的两个参数是：m，一个指向包含了该IP数据报的mbuf链的指针；iphlen，IP首部的长度(包括可能的IP选项)。

1. 丢弃IP选项

67-76 如果有IP选项，则ip_stripoptions丢弃它们。正如注释中表明的，UDP应该保存IP选项的一个备份，使接收进程可以通过IP_RECVOPTS插口选项访问到它们，但这个还没有实现。

77-88 如果mbuf链上的第一个mbuf小于28字节(IP首部加上UDP首部的大小)，则m_pullup重新安排mbuf链，使至少有28个字节连续地存放在第一个mbuf中。

udp_usrreq.c

```
 55 void
 56 udp_input(m, iphlen)
 57 struct mbuf *m;
 58 int     iphlen;
 59 {
 60     struct ip *ip;
 61     struct udphdr *uh;
 62     struct inpcb *inp;
 63     struct mbuf *opts = 0;
 64     int     len;
 65     struct ip save_ip;
 66     udpstat.udps_ipackets++;
 67     /*
 68      * Strip IP options, if any; should skip this,
 69      * make available to user, and use on returned packets,
 70      * but we don't yet have a way to check the checksum
 71      * with options still present.
 72      */
 73     if (iphlen > sizeof(struct ip)) {
 74         ip_stripoptions(m, (struct mbuf *) 0);
 75         iphlen = sizeof(struct ip);
 76     }
 77     /*
 78      * Get IP and UDP header together in first mbuf.
 79      */
 80     ip = mtod(m, struct ip *);
 81     if (m->m_len < iphlen + sizeof(struct udphdr)) {
 82         if ((m = m_pullup(m, iphlen + sizeof(struct udphdr))) == 0) {
 83             udpstat.udps_hdrops++;
 84             return;
 85         }
 86         ip = mtod(m, struct ip *);
 87     }
 88     uh = (struct udphdr *) ((caddr_t) ip + iphlen);
 89     /*
 90      * Make mbuf data length reflect UDP length.
 91      * If not enough data to reflect UDP length, drop.
 92      */
 93     len = ntohs((u_short) uh->uh_ulen);
 94     if (ip->ip_len != len) {
 95         if (len > ip->ip_len) {
 96             udpstat.udps_badlen++;
 97             goto bad;
 98         }
 99         m_adj(m, len - ip->ip_len);
100         /* ip->ip_len = len; */
101     }
102     /*
103      * Save a copy of the IP header in case we want to restore
104      * it for sending an ICMP error message in response.
105      */
106     save_ip = *ip;
107     /*
108      * Checksum extended UDP header and data.
109      */
110     if (udpcksum && uh->uh_sum) {
```

图23-21 udp_input 函数：对收到的UDP数据报的一般确认

```
111         ((struct ipovly *) ip)->ih_next = 0;
112         ((struct ipovly *) ip)->ih_prev = 0;
113         ((struct ipovly *) ip)->ih_x1 = 0;
114         ((struct ipovly *) ip)->ih_len = uh->uh_ulen;
115         if (uh->uh_sum = in_cksum(m, len + sizeof(struct ip))) {
116             udpstat.udps_badsum++;
117             m_freem(m);
118             return;
119         }
120     }
```

udp_usrreq.c

图23-21 (续)

2. 验证UDP长度

333-344 与UDP 数据报相关的两个长度是：IP首部的长度字段(ip_len)和UDP首部的长度字段(uh_ulen)。前面讲到，ipintr在调用udp_input之前，从ip_len中抽取出IP首部的长度(图10-11)。比较这两个长度，可能有三种可能性：

1) ip_len等于uh_ulen。这是通常情况。

2) ip_len大于uh_ulen。IP首部太大，如图23-22所示。代码相信两个长度中小的那个(UDP首部长度)，m_adj从数据报的最后移走多余的数据字节。m_adj的第二个参数是负数，在图2-20中我们说，从mbuf链的最后截断数据。在这种情况下，UDP的长度字段出现冲突。如果是这样，假定发送方计算了UDP的检验和，则不久检验和会检测到这个错误，接收方也会验证检验和，从而丢弃该数据报。IP长度字段必须正确，因为IP根据接口上收到的数据量验证它，而强制的IP首部检验和覆盖了IP首部的长度字段。

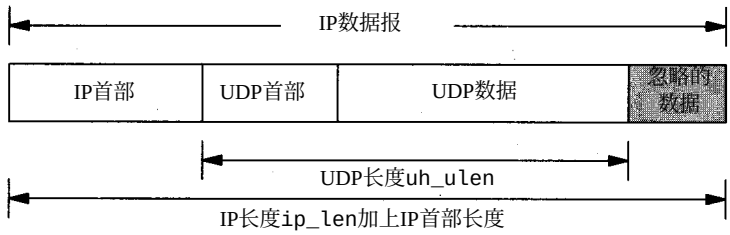

图23-22 UDP长度太小

3) ip_len小于uh_ulen。当UDP 首部的长度给定时，IP数据报比可能的小。图23-23显示了这种情况。这说明数据报有错误，必须丢弃，没有其他的选择：如果 UDP长度字段被破坏，用UDP检验和是无法检测到的。需要用正确的UDP长度来计算UDP检验和。

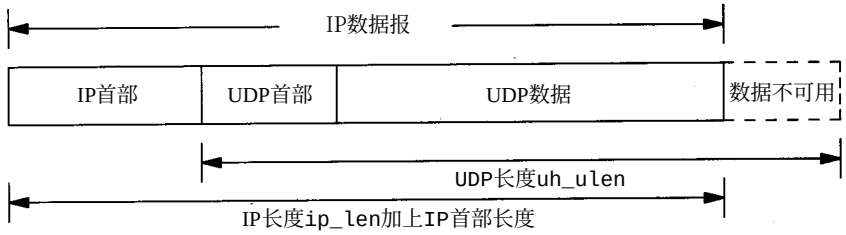

图23-23 UDP长度太大

正如我们提到的，UDP长度是冗余的。在第28章中我们将看到，TCP自己的首部内没有长度字段 —— 它用IP长度字段减去IP和TCP首部的长度，以此确定数据报内数

据的数量。为什么存在 UDP长度字段呢？可能是为了加上少量的差错检测，因为UDP检验和是可选的。

3. 保存IP首部的备份，验证UDP检验和

102-106 udp_input在验证检验和之前保存IP首部的备份，因为检验和计算会擦去原始 IP首部的一些字段。

110 只有当的UDP检验和(udpcksum)是内核允许的，并且发送方也计算了 UDP检验和(收到的检验和不为0)时，才验证检验和。

这个检测是不正确的。如果发送方计算了一个检验和，就应该验证它，不管外出的检验和是否被计算。变量 udpcksum应该只指定是否计算外出的检验和。不幸的是，许多厂商都复制了这个检测，尽管厂商已经改变它们产品的内核，却默认地允许UDP检验和。

111-120 在计算检验和之前， IP首部作为ipovly结构(图23-18)引用，所有字段的初始化都是udp_output在计算UDP检验和(上一节)时初始化的。

此时，如果数据报的目的地是一个广播或多播 IP地址，将执行特别的代码。我们把这段代码推迟到本节最后。

23.7.2 分用单播数据报

假定数据报的目的地是一个单播地址，图 23-24显示了执行的代码。

udp_usrreq.c

```
        /* demultiplex broadcast & multicast datagrams (Figure 23.26) */
206     /*
207      * Locate pcb for unicast datagram.
208      */
209     inp = udp_last_inpcb;
210     if (inp->inp_lport != uh->uh_dport ||
211         inp->inp_fport != uh->uh_sport ||
212         inp->inp_faddr.s_addr != ip->ip_src.s_addr ||
213         inp->inp_laddr.s_addr != ip->ip_dst.s_addr) {
214         inp = in_pcblookup(&udb, ip->ip_src, uh->uh_sport,
215                            ip->ip_dst, uh->uh_dport, INPLOOKUP_WILDCARD);
216         if (inp)
217             udp_last_inpcb = inp;
218         udpstat.udpps_pcbcachemiss++;
219     }
220     if (inp == 0) {
221         udpstat.udps_noport++;
222         if (m->m_flags & (M_BCAST | M_MCAST)) {
223             udpstat.udps_noportbcast++;
224             goto bad;
225         }
226         *ip = save_ip;
227         ip->ip_len += iphlen;
228         icmp_error(m, ICMP_UNREACH, ICMP_UNREACH_PORT, 0, 0);
229         return;
230     }
```

udp_usrreq.c

图23-24 udp_input 函数：分用单播数据报

1. 检查“向后一个”高速缓存

206-209 UDP维护一个指针，该指针指向最后在其上接收数据报的Internet PCB，udp_last_inpcb。在调用in_pcblookup之前，可能必须搜索UDP表上的PCB，把最近一次接收PCB的外部和本地地址以及端口号和收到数据报的进行比较。这称为“向后一个”高速缓存(one-behind cache)[Partridge和Pink 1993]。它是根据这样一个假设，即收到的数据报极有可能要发往上一个数据报发往的同一端口[Mogul 1991]。这个高速缓存技术是在4.3BSD Tahoe版本中引入的。

210-213 高速缓存的PCB和收到数据报之间的四个比较的次序是故意安排的。如果PCB不匹配，则应尽快结束比较。最大的可能性是目的端口号不相同——这就是为什么第一个检测它。不匹配的可能性最小的是本地地址，尤其在只有一个接口的主机，所以它是最后一个检测。

不幸的是，这种“向后一个”高速缓存技术代码，在实际中毫无用处[Partridge和Pink 1993]。最普通的UDP服务器类型只绑定它的知名端口，它的本地地址、外部地址和外部端口都是通配地址。最普通的UDP客户程序类型并不连接它的UDP插口；它用sendto指定每个数据报的目的地址。因此，大多数时间PCB内的inp_laddr、inp_faddr和inp_fport都是通配的。在高速缓存的比较中，收到数据报的这四个值永远都不是通配的，这意味着只有当指定PCB的四个本地和外部值是非通配时，高速缓存入口与收到数据报的比较才可能相等。这种情况只在连接上的UDP插口上发生。

在系统bsdi上，udpps_pcbcachemiss计数器是41 253，udps_ipackets计数器是42 485。小于3%缓存命中率。

netstat-s命令打印出udpstat结构(图23-5)的大多数字段。不幸的是，Net/3版本，以及多数厂家的版本都不打印udpps_pcbcachemiss。如果你想看它们的值，用调试器检查在运行的内核中的变量。

2. 搜索所有UDP的PCB

214-218 假定与高速缓存的比较失败，则in_pcblookup寻找一个匹配。指定INPLOOKUP_WILDCARD标志，允许通配匹配。如果找到一个匹配，则把指向该PCB的指针保存在udp_last_inpcb中，我们说它高速缓存了最后收到的UDP数据报的PCB。

3. 生成ICMP端口不可达差错

220-230 如果没找到匹配的PCB，UDP通常产生一个ICMP端口不可达差错。首先检测收到的mbuf链的m_flags，看看该数据报是否是要发送到一个链路级广播或多播地址。有可能会收到一个发送到链路级广播或多播地址的IP数据报，具有单播地址，此时不应该产生ICMP端口不可达差错。如果成功产生ICMP差错，则把IP首部恢复成收到它时的值(save_ip)，也把IP长度设置成它原来的值。

链路级广播或多播地址的检测是冗余的。icmp_error也做这个检测。这个冗余检测的唯一好处是，在udps_noportbcast计数器之外，还维护了udps_noport计数器。

把iphlen改回ip_len是一个错误。icmp_error也会做这项工作，使得ICMP差错返回的IP首部的IP长度字段是20字节，这太大了。可以在Traceroute程

序(卷1的第8章)中加上几行新程序，在最终到达目的主机后，打印出 ICMP端口不可达差错报文中的这个字段，可以测试系统是否有这个错误。

图23-25是处理单播数据报的代码，把数据报提交给与目的 PCB对应的插口。

udp_usrreq.c

```
231     /*
232      * Construct sockaddr format source address.
233      * Stuff source address and datagram in user buffer.
234      */
235     udp_in.sin_port = uh->uh_sport;
236     udp_in.sin_addr = ip->ip_src;

237     if (inp->inp_flags & INP_CONTROLOPTS) {
238         struct mbuf **mp = &opts;

239         if (inp->inp_flags & INP_RECVDSTADDR) {
240             *mp = udp_saveopt((caddr_t) & ip->ip_dst,
241                               sizeof(struct in_addr), IP_RECVDSTADDR);
242             if (*mp)
243                 mp = &(*mp)->m_next;
244         }
245 #ifdef notyet
246         /* IP options were tossed above */
247         if (inp->inp_flags & INP_RECVOPTS) {
248             *mp = udp_saveopt((caddr_t) opts_deleted_above,
249                               sizeof(struct in_addr), IP_RECVOPTS);
250             if (*mp)
251                 mp = &(*mp)->m_next;
252         }
253         /* ip_srcroute doesn't do what we want here, need to fix */
254         if (inp->inp_flags & INP_RECVRETOPTS) {
255             *mp = udp_saveopt((caddr_t) ip_srcroute(),
256                               sizeof(struct in_addr), IP_RECVRETOPTS);
257             if (*mp)
258                 mp = &(*mp)->m_next;
259         }
260 #endif
261     }
262     iphlen += sizeof(struct udphdr);
263     m->m_len -= iphlen;
264     m->m_pkthdr.len -= iphlen;
265     m->m_data += iphlen;
266     if (sbappendaddr(&inp->inp_socket->so_rcv, (struct sockaddr *) &udp_in,
267                      m, opts) == 0) {
268         udpstat.udps_fullsock++;
269         goto bad;
270     }
271     sorwakeup(inp->inp_socket);
272     return;

273   bad:
274     m_freem(m);
275     if (opts)
276         m_freem(opts);
277 }
```

udp_usrreq.c

图23-25 udp_input 函数：把单播数据报提交给插口

4. 返回源站IP地址和源站端口

231-236 收到的IP数据报的源站IP地址和源站端口被保存在全局 sockaddr_in结构中的

udp_in。在函数的后面，该结构作为参数传给了sbappendaddr。

采用全局变量保存IP地址和端口号不出现问题的原因是，udp_input是单线程的。当ipintr调用它时，它在返回之前完整地处理了收到的数据报。而且，sbappendaddr还把该插口结构从全局变量复制一个mbuf中。

5. IP_RECVDSTADDR插口选项

337-244 常数INP_CONTROLOPTS是三个插口选项的结合，进程可以设置这三个插口选项，通过系统调用recvmsg返回插口的控制信息(图22-5)。IP_RECVDSTADDR把收到的UDP数据报中的目的IP地址作为控制信息返回。函数udp_saveopt分配一个MT_CONTROL类型的mbuf，并把4字节的目的IP地址存放在该缓存中。我们在23.8节中介绍这个函数。

该插口选项与4.3BSD Reno一起出现，是为一般文件传输协议TFTP的应用程序设计的，它们不响应发给广播地址的客户程序请求。不幸的是，即使接收应用程序使用这个选项，也很难确定目的IP地址是否是一个广播地址(习题23.6)。

当4.4BSD中加上了多播功能后，这个代码只对目的地是单播地址的数据报有效。我们将在图23-26看到，对发给多播地址的广播数据报还没有实现这个选项，根本无法达到该选项的目的。

6. 未实现的插口选项

245-260 这段代码被注释掉了，因为它们不起作用。IP_RECVOPTS插口选项的原意是把收到数据报的IP选项作为控制信息返回，而IP_RECVRETOPTS插口选项返回源路由信息。三个IP_RECV插口选项对mp变量的操作构造了一个最多有三个mbuf的链表，该链表由sbappendaddr放置到插口的缓存。图23-25显示的代码只把一个选项作为控制信息返回，所以指向该mbuf的m_next总是一个空指针。

7. 把数据加到插口的接收队列中

262-272 此时，已经准备好把收到的数据报(m指向的mbuf链)以及一个表示发送方IP地址和端口的插口地址结构(udp_in)和一些可选的控制信息(opts指向的mbuf，目的IP地址)放到插口的接收队列中。这个工作由sbappendaddr完成。但在调用这个函数之前，要修正指针和缓存链上的第一个mbuf，忽略掉UDP和IP首部。返回之前，调用sorwakeup唤醒插口接收队列中的所有睡眠进程。

8. 返回差错

273-276 如果在UDP输入处理的过程中遇到错误，udp_input会跳转到bad标号语句，释放所有包含该数据报以及控制信息(如果有的话)的mbuf链。

23.7.3 分用多播和广播数据报

现在返回到udp_input处理发给广播或多播IP地址数据报的这部分代码。如图23-26所示。

121-138 正如注释所表明的，这些数据报被提交给匹配的所有插口，而不仅仅是一个插口。我们提到的UDP接口不够指的是除非连接上插口，否则进程没有能力在UDP插口上接收异步差错(特别是ICMP端口不可达差错)。我们22-11节讨论这个问题。

139-145 源站的IP地址和端口号被保存在全局sockaddr_in结构的udp_in中，传给sbappendaddr。更新mbuf链的长度和数据指针，忽略UDP和IP首部。

udp_usrreq.c

```
121     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr)) ||
122         in_broadcast(ip->ip_dst, m->m_pkthdr.rcvif)) {
123         struct socket *last;
124         /*
125          * Deliver a multicast or broadcast datagram to *all* sockets
126          * for which the local and remote addresses and ports match
127          * those of the incoming datagram.  This allows more than
128          * one process to receive multi/broadcasts on the same port.
129          * (This really ought to be done for unicast datagrams as
130          * well, but that would cause problems with existing
131          * applications that open both address-specific sockets and
132          * a wildcard socket listening to the same port -- they would
133          * end up receiving duplicates of every unicast datagram.
134          * Those applications open the multiple sockets to overcome an
135          * inadequacy of the UDP socket interface, but for backwards
136          * compatibility we avoid the problem here rather than
137          * fixing the interface.  Maybe 4.5BSD will remedy this?)
138          */
139         /*
140          * Construct sockaddr format source address.
141          */
142         udp_in.sin_port = uh->uh_sport;
143         udp_in.sin_addr = ip->ip_src;
144         m->m_len -= sizeof(struct udpiphdr);
145         m->m_data += sizeof(struct udpiphdr);
146         /*
147          * Locate pcb(s) for datagram.
148          * (Algorithm copied from raw_intr().)
149          */
150         last = NULL;
151         for (inp = udb.inp_next; inp != &udb; inp = inp->inp_next) {
152             if (inp->inp_lport != uh->uh_dport)
153                 continue;
154             if (inp->inp_laddr.s_addr != INADDR_ANY) {
155                 if (inp->inp_laddr.s_addr !=
156                     ip->ip_dst.s_addr)
157                     continue;
158             }
159             if (inp->inp_faddr.s_addr != INADDR_ANY) {
160                 if (inp->inp_faddr.s_addr !=
161                     ip->ip_src.s_addr ||
162                     inp->inp_fport != uh->uh_sport)
163                     continue;
164             }
165             if (last != NULL) {
166                 struct mbuf *n;
167                 if ((n = m_copy(m, 0, M_COPYALL)) != NULL) {
168                     if (sbappendaddr(&last->so_rcv,
169                                      (struct sockaddr *) &udp_in,
170                                      n, (struct mbuf *) 0) == 0) {
171                         m_freem(n);
172                         udpstat.udps_fullsock++;
173                     } else
174                         sorwakeup(last);
175                 }
176             }
177             last = inp->inp_socket;
```

图23-26 udp_input 函数：分用广播或多播数据报

```
178             /*
179              * Don't look for additional matches if this one does
180              * not have either the SO_REUSEPORT or SO_REUSEADDR
181              * socket options set.  This heuristic avoids searching
182              * through all pcbs in the common case of a non-shared
183              * port.  It assumes that an application will never
184              * clear these options after setting them.
185              */
186             if ((last->so_options & (SO_REUSEPORT | SO_REUSEADDR) == 0))
187                 break;
188         }
189         if (last == NULL) {
190             /*
191              * No matching pcb found; discard datagram.
192              * (No need to send an ICMP Port Unreachable
193              * for a broadcast or multicast datgram.)
194              */
195             udpstat.udps_noportbcast++;
196             goto bad;
197         }
198         if (sbappendaddr(&last->so_rcv, (struct sockaddr *) &udp_in,
199                          m, (struct mbuf *) 0) == 0) {
200             udpstat.udps_fullsock++;
201             goto bad;
202         }
203         sorwakeup(last);
204         return;
205     }
```

udp_usrreq.c

图23-26 (续)

146-164 大的for循环扫描每个 UDP PCB，寻找所有匹配 PCB。这种分用不调用in_pcblookup，因为它只返回一个PCB，而广播或多播数据报可能需要提交给多个 PCB。

如果PCB的本地端口和收到数据报的本地端口不匹配，则忽略该入口。如果 PCB的本地端口不是通配地址，则把它和目的 IP地址比较，如果不相等则跳过该入口。如果 PCB内的外部地址不是通配地址，就把它和源站 IP地址比较，如果不匹配，则外部端口也必须和源站端口匹配。最后一个检测假定，如果插口连接到某个外部 IP地址，则它也必须连接到一个外部端口，反之亦然。这与in_pcblookup函数的逻辑相同。

165-177 如果这不是第一个匹配(last非空)，则把该数据报放到上一个匹配的接收队列中。因为当sbappendaddr完成后要释放mbuf链，所以m_copy先要做个备份。sorwakeup唤醒所有等待这个数据的进程，last保存指向匹配的socket结构的指针。

使用变量last避免调用m_copy函数(因为要复制整个mbuf链，所以耗费很大)，除非有多个接收方接收该数据报。在通常只有一个接收方的情况下，for循环必须把last设成指向一个匹配PCB，当循环终止时，sbappendaddr把mbuf链放到插口的接收队列中——不做备份。

178-188 如果匹配的插口没有设置SO_REUSEPORT或SO_REUSEADDR插口选项，则没必要再找其他匹配，终止该循环。在循环的外部，调用 sbappendaddr把数据报放到这个插口的接收队列中。

189-197 如果在循环的最后，last为空，没找到匹配，则并不产生 ICMP差错，因为该数据报是发给广播或多播IP地址。

198-204 最后的匹配入口(可能是唯一的匹配入口)把原来的数据报(m)放到它的接收队列中。在调用sorwakeup后，udp_input返回，因为完成了对广播或多播数据报的处理。

函数的其他部分(图23-24)处理单播数据报。

23.7.4 连接上的UDP插口和多接口主机

在使用连接上的UDP插口与多接口主机上的一个进程交换数据报时，有一个微妙的问题。来自对等实体的数据报可能到达时具有不同的源站 IP地址，不能提交给连接上的插口。

考虑图23-27所示的例子。

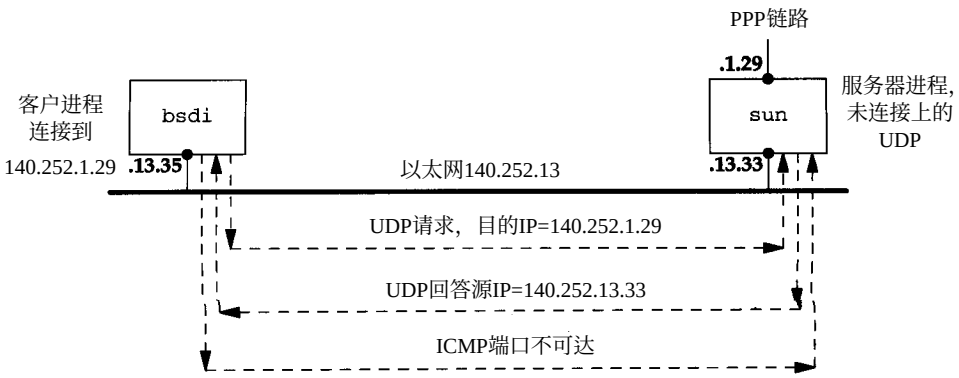

图23-27 连接上的UDP插口向一个多接口主机发送数据报的例子

有三个步骤：

1) bsdi上的客户程序创建一个UDP插口，并把它连接到140.252.1.29，这是sun上的PPP接口，而不是以太网接口。客户程序在插口上把数据报发给服务器。

Sun上的服务器接收并收下该数据报，即使到达接口与目的 IP地址不同(sun是一个路由器，所以不管它实现的是弱端系统模型或强端系统模型都没有关系)。数据报被提交给在未连接上的UDP插口上等待客户请求的服务器。

2) 服务器发一个回答，因为是在一个未连接上的 UDP插口上发送的，所以由内核根据外出的接口(140.252.13.33)选择回答的目的IP地址。请求的目的IP地址不作为回答的源站地址。

bsdi收到回答时，因为 IP地址不匹配，所以不把它提交给客户程序的连接上的 UDP接口。

3) 因为无法分用回答，所以 bsdi产生一个ICMP端口不可达差错(假定在bsdi上没有其他进程符合接收该数据报的条件)。

这个例子的问题在于，服务器并不把请求中的目的 IP地址作为回答的源站 IP地址。如果它这样做，就不存在这个问题了，但这个办法并不简单 —— 见习题23.10。我们将在图28-16中看到，如果一个 TCP服务器没有明确地把一个本地 IP地址绑定它的插口上，它就把来自客户的目的IP地址用作来自它自己的源IP地址。

23.8 udp_saveopt函数

如果进程指定了IP_RECVDSTADDR插口选项，则udp_input调用udp_saveopt，从收到的数据报中接收目的IP地址：

```
*mp = udp_saveopt((caddr_t) & ip_dst, sizeof(struct in_addr),
                  IP_RECVDSTADDR);
```

图23-28显示了这个函数。

udp_usrreq.c

```
278 /*
279  * Create a "control" mbuf containing the specified data
280  * with the specified type for presentation with a datagram.
281  */
282 struct mbuf *
283 udp_saveopt(p, size, type)
284 caddr_t p;
285 int     size;
286 int     type;
287 {
288     struct cmsghdr *cp;
289     struct mbuf *m;

290     if ((m = m_get(M_DONTWAIT, MT_CONTROL)) == NULL)
291         return ((struct mbuf *) NULL);
292     cp = (struct cmsghdr *) mtod(m, struct cmsghdr *);
293     bcopy(p, CMSG_DATA(cp), size);
294     size += sizeof(*cp);
295     m->m_len = size;
296     cp->cmsg_len = size;
297     cp->cmsg_level = IPPROTO_IP;
298     cp->cmsg_type = type;
299     return (m);
300 }
```

udp_usrreq.c

图23-28 udp_saveopt 函数：用控制信息创建mbuf

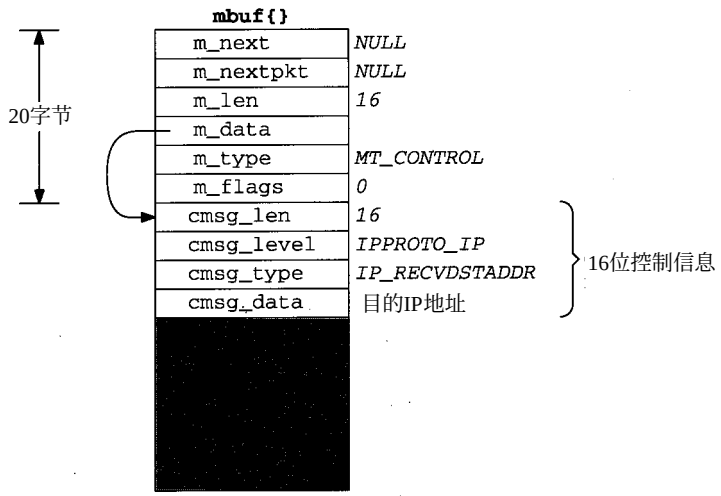

图23-29 把收到的数据报的目的地址作为控制信息保存的 mbuf

276-286 参数包括p，一个指向存储在mbuf中的信息的指针(收到的数据报的目的IP地址)；size，字节数大小(在这个例子中是4，IP地址的大小)；以及type，控制信息的类型(IP_RECVDSTADDR)。

290-299 分配一个mbuf，并且因为是在软件中断级执行代码，所以指定M_DONTWAIT。指针cp指向mbuf的数据部分，是一个指向cmsghdr结构(图16-14)的指针。bcopy把IP首部中

的IP地址复制到cmsghdr结构的数据部分。然后设置紧跟在cmsghdr结构后面的mbuf的长度(在本例中设成16)。图23-29是mbuf的最后一个状态。

cmsg_len字段包含了cmsghdr的长度(12)加上cmsg_data字段的长度(本例中是4)。如果应用程序调用recvmsg接收控制信息，则它必须检查cmsghdr结构，确定cmsg_data字段的类型和长度。

23.9 udp_ctlinput函数

当icmp_input收到一个ICMP差错(目的主机不可达、参数问题、重定向、源站抑制和超时)时，调用相应协议的pr_ctlinput函数：

```
if (ctlfunc = inetsw[ ip_protox[icp->icmp_ip.ip_p] ].pr_ctlinput)
    (*ctlfunc)(code, (struct sockaddr *)&icmpsrc, &icp->icmp_ip);
```

对于UDP，调用图22-32显示的函数udp_ctlinput。我们将在图23-30中给出这个函数。

314-322 参数包括cmd，图11-19的一个PRC_*xxx*常数；sa，一个指向sockaddr_in结构的指针，该结构含有ICMP报文的源站IP地址；ip，一个指向引起差错的IP首部的指针。对于目的站不可达、参数问题、源站抑制和超时差错，ip指向引起差错的IP首部。但当pfctlinput为重定向(图22-32)调用udp_ctlinput时，sa指向一个sockaddr_in结构，该结构中包含要被重定向的目的地址，ip是一个空指针。最后一种情况没有信息丢失，因为我们在22.11节看到，重定向应用于所有连接到目的地址的TCP和UDP插口。但对其他差错，如端口不可达，需要非空的第三个参数，因为协议跟在IP首部后面的协议首部包含了不可达端口。

323-325 如果差错不是重定向，并且PRC_*xxx*的值太大或全局数组inetctlerrmap中没有差错码，则忽略该ICMP差错。为理解这个检测，我们来看一下对收到的ICMP所做的处理：

1) icmp_input把ICMP类型和码转换成一个PRC_*xxx*差错码。

2) 把PRC_*xxx*差错码传给协议的控制输入函数。

3) Internet PCB协议(TCP和UDP)用inetctlerrmap把PRC_*xxx*差错码映射到一个Unix的errno值，这个值被返回给进程。

udp_usrreq.c

```
314 void
315 udp_ctlinput(cmd, sa, ip)
316 int     cmd;
317 struct sockaddr *sa;
318 struct ip *ip;
319 {
320     struct udphdr *uh;
321     extern struct in_addr zeroin_addr;
322     extern u_char inetctlerrmap[];

323     if (!PRC_IS_REDIRECT(cmd) &&
324         ((unsigned) cmd >= PRC_NCMDS || inetctlerrmap[cmd] == 0))
325         return;
326     if (ip) {
327         uh = (struct udphdr *) ((caddr_t) ip + (ip->ip_hl << 2));
328         in_pcbnotify(&udb, sa, uh->uh_dport, ip->ip_src, uh->uh_sport,
329                      cmd, udp_notify);
330     } else
331         in_pcbnotify(&udb, sa, 0, zeroin_addr, 0, cmd, udp_notify);
332 }
```

udp_usrreq.c

图23-30 udp_ctlinput 函数：处理收到的ICMP差错

图11-1和图11-2总结了ICMP报文的处理。

回到图 23-30，我们可以看到如何处理响应 UDP数据报的 ICMP源站抑制报文。icmp_input把ICMP报文转换成差错PRC_QUENCH，并调用udp_ctlinput。但因为在图11-2中，这个ICMP差错的errno行是空白，所以忽略该差错。

326-331 in_pcbnotify函数把该ICMP差错通知给恰当的PCB。如果udp_ctlinput的第三个参数非空，则把引起差错数据报的源和目的UDP端口以及源IP地址，传给in_pcbnotify。

udp_notify函数

in_pcbnotify函数的最后一个参数是一个指向函数的指针，in_pcbnotify为每个准备接收差错的PCB调用该函数。对UDP，该函数是udp_notify，如图23-31所示。

301-313 该函数的第二个参数errno保存在插口的so_error变量中。通过设置这个插口变量，使插口变成可读，并且如果进程调用 select，插口也可写。然后唤醒插口上所有正在等待接收或发送的进程接收该差错。

udp_usrreq.c

```
305 static void
306 udp_notify(inp, errno)
307 struct inpcb *inp;
308 int     errno;
309 {
310     inp->inp_socket->so_error = errno;
311     sorwakeup(inp->inp_socket);
312     sowwakeup(inp->inp_socket);
313 }
```

udp_usrreq.c

图23-31 udp_notify函数：通知进程一个异步差错

23.10 udp_usrreq函数

许多操作都要调用协议的用户请求函数。从图 23-14我们看到，在某个UDP插口上调用五个写函数中的任意一个，都以请求 PRU_SEND调用UDP 的用户请求函数结束。

图23-32显示了udp_usrreq的开始和结束。switch单独在后面的图中给出。图 15-17显示了该函数的参数。

udp_usrreq.c

```
417 int
418 udp_usrreq(so, req, m, addr, control)
419 struct socket *so;
420 int     req;
421 struct mbuf *m, *addr, *control;
422 {
423     struct inpcb *inp = sotoinpcb(so);
424     int     error = 0;
425     int     s;

426     if (req == PRU_CONTROL)
427         return (in_control(so, (int) m, (caddr_t) addr,
428                            (struct ifnet *) control));
429     if (inp == NULL && req != PRU_ATTACH) {
430         error = EINVAL;
431         goto release;
432     }
```

图23-32 udp_usrreq 函数体

```
433     /*
434      * Note: need to block udp_input while changing
435      * the udp pcb queue and/or pcb addresses.
436      */
437     switch (req) {

                        /* switch cases */

522     default:
523         panic("udp_usrreq");
524     }

525   release:
526     if (control) {
527         printf("udp control data unexpectedly retained\n");
528         m_freem(control);
529     }
530     if (m)
531         m_freem(m);
532     return (error);
533 }
```

udp_usrreq.c

图23-32 (续)

417-428 PRU_CONTROL请求来自ioctl系统调用。函数in_control完整地处理该请求。

429-432 在函数的开头定义inp时，把插口指针转换成PCB指针。唯一允许PCB指针为空的时候是创建新插口时(PRU_ATTACH)。

433-436 注释表明，只要在UDP PCB表中增加或删除表项，代码必须由splnet保护起来。这是因为udp_usrreq是作为系统调用的一部分来调用的，在它修改PCB的双重链表时，不能被UDP输入中断(被IP输入作为软件中断调用)。在修改PCB的本地或外部地址或端口时，也必须阻塞UDP输入，避免in_pcblookup不正确地提交收到的UDP数据报。

我们现在讨论每个case语句。图23-33语句中的PRU_ATTACH请求，来自socket系统调用。

```
438     case PRU_ATTACH:
439         if (inp != NULL) {
440             error = EINVAL;
441             break;
442         }
443         s = splnet();
444         error = in_pcballoc(so, &udb);
445         splx(s);
446         if (error)
447             break;
448         error = soreserve(so, udp_sendspace, udp_recvspace);
449         if (error)
450             break;
451         ((struct inpcb *) so->so_pcb)->inp_ip.ip_ttl = ip_defttl;
452         break;

453     case PRU_DETACH:
454         udp_detach(inp);
455         break;
```

udp_usrreq.c

图23-33 udp_usrreq 函数：PRU_ATTACH 和PRU_DETACH 请求

438-447 如果插口结构已经指向一个PCB，则返回EINVAL。in_pcballoc分配一个新的PCB，把它加到UDP PCB表的前面，把插口结构和PCB链接到一起。

448-450 soreserve为插口的发送和接收缓存保留缓存空间。如图 16-7所示，soreserve只是实施系统的限制，并没有真正分配缓存空间。发送和接收缓存的默认大小分别是9216字节(udp_sendspace)和41 600字节(udp_recvspace)。前者允许最大9200字节的数据报(在NFS分组中，有8 KB的数据)，加上16字节目的地址的sockaddr_in结构。后者允许插口上一次最多有40个1024字节的数据报排队。进程可调用setsockopt改变这些值。

451-452 进程通过setsockopt函数可以改变PCB中原型IP首部的两个字段：TTL和TOS。TTL默认值是64(ip_defttl)，TOS的默认值是0(普通服务)，因为in_pcballoc把PCB初始化为0。

453-455 close系统调用发布PRU_DETACH请求，调用图23-34所示的udp_detach函数。本节后面的PRU_ABORT请求也调用这个函数。

udp_usrreq.c

```
534 static void
535 udp_detach(inp)
536 struct inpcb *inp;
537 {
538     int     s = splnet();

539     if (inp == udp_last_inpcb)
540         udp_last_inpcb = &udb;
541     in_pcbdetach(inp);
542     splx(s);
543 }
```

udp_usrreq.c

图23-34 udp_detach 函数：删除一个UDP PCB

如果最后收到的PCB指针(“向后一个”缓存)指向一个已分离的PCB，则把缓存的指针设成指向UDP 表的表头(udb)。函数in_pcbdetach从UDP 表中移走PCB，并释放该PCB。

回到udp_usrreq，PRU_BIND请求是系统调用bind的结果，而PRU_LISTEN请求是系统调用listen的结果。如图23-35所示。

456-460 in_pcbbind完成所有PRU_BIND请求的工作。

461-463 对无连接协议来说，PRU_LISTEN请求是无效的——只有面向连接的协议才使用它。

udp_usrreq.c

```
456     case PRU_BIND:
457         s = splnet();
458         error = in_pcbbind(inp, addr);
459         splx(s);
460         break;

461     case PRU_LISTEN:
462         error = EOPNOTSUPP;
463         break;
```

udp_usrreq.c

图23-35 udp_usrreq 函数：PRU_BIND 和PRU_LISTEN 请求

前面提到，一个UDP应用程序，客户或服务器 (通常是客户)，可以调用connect。它修改插口发送或接收的外部 IP地址和端口号。图 23-6显示了PRU_CONNECT、PRU_CONNECT2和PRU_ACCEPT请求。

464-474 如果插口已经连接上，则返回EISCONN。在这个时候，不应该连接上插口，因为在一个已经连接上的UDP插口上调用connect，会在生成PRU_CONNECT请求之前生成PRU_DISCONNECT请求。否则，由in_pcbconnect完成所有工作。如果没有遇到任何错误，soisconnected就把该插口结构标记成已经连接上的。

475-477 socketpair系统调用发布PRU_CONNECT2请求，只适用于Unix域的协议。

478-480 PRU_ACCEPT请求来自系统调用accept，只适用于面向连接的协议。

udp_usrreq.c

```
464     case PRU_CONNECT:
465         if (inp->inp_faddr.s_addr != INADDR_ANY) {
466             error = EISCONN;
467             break;
468         }
469         s = splnet();
470         error = in_pcbconnect(inp, addr);
471         splx(s);
472         if (error == 0)
473             soisconnected(so);
474         break;

475     case PRU_CONNECT2:
476         error = EOPNOTSUPP;
477         break;

478     case PRU_ACCEPT:
479         error = EOPNOTSUPP;
480         break;
```

udp_usrreq.c

图23-36 udp_usrreq 函数：PRU__CONNECT 、PRU__CONNECT2 和PRU__ACCEPT 请求

对于UDP插口，有两种情况会产生PRU_DISCONNECT请求：

1) 当关闭了一个连接上的UDP插口时，在PRU_DETACH之前调用PRU_DISCONNECT。

2) 当在一个已经连接上的UDP插口上发布connect时，soconnect在PRU_CONNECT请求之前发布PRU_DISCONNECT请求。

PRU_DISCONNECT请求如图23-37所示。

udp_usrreq.c

```
481     case PRU_DISCONNECT:
482         if (inp->inp_faddr.s_addr == INADDR_ANY) {
483             error = ENOTCONN;
484             break;
485         }
486         s = splnet();
487         in_pcbdisconnect(inp);
488         inp->inp_laddr.s_addr = INADDR_ANY;
489         splx(s);
490         so->so_state &= ~SS_ISCONNECTED;    /* XXX */
491         break;
```

udp_usrreq.c

图23-37 udp_usrreq 函数：PRU_DISCONNECT 请求

如果插口没有连接上，则返回ENOTCONN。否则，in_pcbdisconnect把外部IP地址设成0.0.0.0，把外部地址设成0。本地地址也被设成0.0.0.0，因为connect可能已经设置了这个PCB变量。

调用shutdown说明进程数据发送结束，产生PRU_SHUTDOWN请求，尽管对UDP插口来

说，很少有进程发布这个系统调用。图 23-38显示了PRU_SHUTDOWN、PRU_SEND和PRU_ABORT请求。

492-494 socantsendmore设置插口的标志，阻止其他更多输出。

495-496 图23-14显示了五个写函数如何调用udp_surreq，发布PRU_SEND请求。udp_output发送该数据报，udp_usrreq返回，避免执行release标号语句(图23-32)，因为还不能释放包含数据的mbuf链(m)。IP输出把这个mbuf链加到合适的接口输出队列中，当发送完数据后，由设备驱动器释放mbuf链。

udp_usrreq.c

```
492     case PRU_SHUTDOWN:
493         socantsendmore(so);
494         break;

495     case PRU_SEND:
496         return (udp_output(inp, m, addr, control));

497     case PRU_ABORT:
498         soisdisconnected(so);
499         udp_detach(inp);
500         break;
```

udp_usrreq.c

图23-38 udp_usrreq 函数体：PRU_SHUTDOWN 、PRU_SEND 和PRU_ABORT 请求

内核中UDP输出的唯一缓冲是在接口的输出队列中。如果插口的发送缓存内有存放数据报和目的地址的空间，则sosend调用udp_usrreq，该函数调用udp_output。图23-20显示，udp_output继续调用ip_output，ip_output为以太网调用ether_output，把数据报放到接口的输出队列中(如果有空间)。如果进程调用sendto的动作比接口快，就可以发送该数据报，ether_output返回ENOBUFS，并被返回给进程。

497-500 在UDP插口上从不发布PRU_ABORT请求。但如果发布，则断连插口，分离PCB。

PRU_SOCKADDR和PRU_PEERADDR请求分别来自系统调用 getsockname和getpeername。这两个请求和PRU_SENSE请求一起，如图23-39所示。

udp_usrreq.c

```
501     case PRU_SOCKADDR:
502         in_setsockaddr(inp, addr);
503         break;

504     case PRU_PEERADDR:
505         in_setpeeraddr(inp, addr);
506         break;

507     case PRU_SENSE:
508         /*
509          * fstat: don't bother with a blocksize.
510          */
511         return (0);
```

udp_usrreq.c

图23-39 udp_usrreq 函数体：PRU_SOCKADDR 、PRU_PEERADDR 和PRU_SENSE 请求

501-506 函数in_setsockaddr和in_setpeeraddr从PCB中取得信息，并把结果保存在addr参数中。

507-511 系统调用fstat产生PRU_SENSE请求。该函数返回OK，但并不返回其他信息。我们将在后面看到，TCP把发送缓存的大小作为stat结构的st_blksize元素返回。

图23-40显示了其他7个PRU_*xxx*请求，UDP插口不支持。

对最后两个请求的处理略微有些不同，因为PRU_RCVD不把指向mbuf的指针(m是一个非空指针)作为参数传递，而PRU_RCVOOB则传递指向协议mbuf的指针来填充。两种情况下，立即返回错误，不终止switch语句的执行，释放mbuf链。调用方用PRU_RCVOOB释放它分配的mbuf。

udp_usrreq.c

```
512     case PRU_SENDOOB:
513     case PRU_FASTTIMO:
514     case PRU_SLOWTIMO:
515     case PRU_PROTORCV:
516     case PRU_PROTOSEND:
517         error = EOPNOTSUPP;
518         break;

519     case PRU_RCVD:
520     case PRU_RCVOOB:
521         return (EOPNOTSUPP);    /* do not free mbuf's */
```

udp_usrreq.c

图23-40 udp_usrreq 函数体：不支持的7个请求

23.11 udp_sysctl函数

UDP的sysctl函数只支持一个选项， UDP检验和标志位。系统管理员可以禁止用sysctl(8)程序使能或禁止UDP检验和。图23-41显示了udp_sysctl函数。该函数调用sysctl_int取得或设置整数udpcksum的值。

udp_usrreq.c

```
547 udp_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
548 int    *name;
549 u_int   namelen;
550 void   *oldp;
551 size_t *oldlenp;
552 void   *newp;
553 size_t  newlen;
554 {
555     /* All sysctl names at this level are terminal. */
556     if (namelen != 1)
557         return (ENOTDIR);

558     switch (name[0]) {
559     case UDPCTL_CHECKSUM:
560         return (sysctl_int(oldp, oldlenp, newp, newlen, &udpcksum));
561     default:
562         return (ENOPROTOOPT);
563     }
564     /* NOTREACHED */
565 }
```

udp_usrreq.c

图23-41 udp_sysctl 函数

23.12 实现求精

23.12.1 UDP PCB高速缓存

在22.12节中，我们讲到PCB搜索的一般性质，以及代码是如何线性搜索协议的PCB表的。现在我们把它和图23-24中UDP使用的“向后一个”高速缓存结合起来。

“向后一个”高速缓存的问题发生在当高速缓存的PCB中有通配值时(本地地址，外部地址

或外部端口)：高速缓存的值永远不和收到的数据报匹配。[Partridge和Pink 1993] 测试的一个解决办法是，修改高速缓存，不比较通配值。也就是说，不再把 PCB中的外部地址和数据报的源地址进行比较，而是只有当PCB中的外部地址不是通配地址时，才比较这两个值。

这个办法有一个微妙的问题[Partridge和Pink 1993]。假定有两个插口绑定到本地端口555上。其中一个有三个通配成份，而另一个已经连接到外部地址 128.1.2.3，外部端口1600。如果我们高速缓存第一个PCB，且有一个数据报来自128.1.2.3，端口1600，则不能仅仅因为高速缓存的值具有通配外部地址就不比较外部地址。这叫做高速缓存隐藏 (cache hiding)。在这个例子中，高速缓存的PCB隐藏了另一个更好匹配的PCB。

为解决高速缓存隐藏，当在高速缓存加上或删除一个入口时，要做更多的工作。不能高速缓存那些可能隐藏其他 PCB的PCB。但这很简单，因为普通情形是每个本地端口都有一个插口。刚才我们给的例子中，两个插口都绑定到本地端口 555，尽管可能(尤其在一个多接口主机上)，但很少见。

[Partridge和Pink 1993]的另一个提高测试的也是记录最后发送的数据报的 PCB。这是[Mogul 1991]提出的，指出在所有收到的数据报中，一半都是对最后发送的数据报的回答。在这里高速缓存隐藏也是个问题，所以不高速缓存那些可能隐藏其他 PCB的PCB。

在通用系统上测试[Partridge和Pink 1993] 的两种高速缓存结果是，100 000个左右收到的UDP数据报显示出57%命中最近收到PCB高速缓存，30%命中最近发送PCB高速缓存。相比于没有高速缓存的版本，`udp_input`使用的CPU时间减少了一半还多。

这两种高速缓存还在某种程度上依赖于位置：刚刚到达的 UDP数据报极大可能来自与最近收到或发送UDP数据报相同的对等实体上。后者对发送一个数据报并等待回答的请求—应答应用程序很典型。[McKenney和Dove 1992] 显示某些应用程序，如联机交易处理(OLTP)系统的数据入口，没有产生 [Partridge和Pink 1993] 观察到的很高的命中率。正如我们在22.12节中提到的，对于具有上千个OLTP连接的系统来说，把PCB放在哈希链上，相对于最近收到和最近发送高速缓存而言，性能提高了一个数量级。

23.12.2 UDP检验和

提高实现性能的下一个领域是把进程和内核之间的数据复制与检验和计算结合起来。Net/3中，在输出操作中，每个数据都被处理两遍：一次是从进程复制到mbuf中(`uiomove`函数，被`sosend`调用)；另一次是计算UDP检验和(函数`in_cksum`被`udp_output`调用)。输入跟输出一样。

[Partridge和Pink1993] 修改了图 23-14的UDP输出处理，调用一个 UDP专有函数`udp_sosend`，而不是`sosend`。这个新函数计算UDP 首部和内嵌的伪首部的检验和(不调用通用的`in_cksum`函数)，然后用特殊函数`in_uiomove`把数据从进程复制到一个mbuf链上(不是通用函数`uiomove`)，由这个新函数复制数据，更新检验和。采用这个技术，花在复制数据和计算检验和的时间减少了40%到45%。

在接收方情况就不同了。UDP 计算UDP首部和伪首部的检验和，移走UDP首部，把数据报在合适的插口上排队。当应用程序读取数据报时，`soreceive`的一个特殊版本(`udp_soreceive`)在把数据复制到用户高速缓存的同时，计算检验和。但是，如果检验和不正确，在整个数据报被复制到用户高速缓存之前，检测不到错误。对于普通的阻塞插口来说，`udp_soreceive`仅仅等待下一个数据报的到达。但是若插口是无阻塞的，且下一个数据报还没有准备好传给进程，就必须返回差错 `EWOULDBLOCK`。对于无阻塞读的UDP插口来说，

这意味着插口接口的两个变化：

1) select函数可以指示无阻塞UDP 插口可读，但如果检验和失败，其中一个读函数依然要返回错误EWOULDBLOCK。

2) 因为是在数据报被复制到用户高速缓存之后检测到检验和错误，所以即使读没有返回数据，应用程序的高速缓存也被改变了。

即使是阻塞插口，如果有检验和错误的数据报包含了 100字节的数据，而下一个没有错误的数据报包含40字节的数据，则recvfrom的返回长度是40，但跟在用户高速缓存后面的 60字节没有改变。

[Partridge和Pink1993] 在六台不同计算机上，对单纯复制和有检验和的复制的计时作了比较。结果显示，在许多体系结构的机器上，在复制操作中计算检验和无需额外时间。这种情况是在内存访问速度和CPU处理速度正确匹配的系统上的，目前许多 RISC处理器都符合条件。

23.13 小结

UDP 是一个无连接的简单协议，这是我们为什么在 TCP之前讨论它的原因。 UDP输出很简单：IP和UDP首部放在用户数据的前面，尽可能填满首部，把结果传递给 ip_output。唯一复杂的是UDP检验和计算，包括只为计算 UDP检验和而加上一个伪首部。我们将在第 26章遇到用于计算TCP检验和的伪首部。

当udp_input收到一个数据报时，它首先完成一个常规确认 (长度和检验和)；然后的处理根据目的IP地址是单播地址、广播或多播地址而不同。最多把单播数据报提交给一个进程，但多播或广播数据报可能会被提交给多个进程。“ 向后一个 ” 高速缓存适用于单播，其中维护着一个指向在其上接收数据报的最近 Internet PCB的指针。但是，我们也看到，由于 UDP应用程序普遍使用通配地址，所以这个高速缓存技术实际上毫无用处。

调用udp_ctlinput函数处理收到的ICMP报文， udp_usrreq函数处理来自插口层的PRU_*xxx*请求。

习题

23.1 列出udp_output传给ip_output的mbuf链的五种类型(提示：看看sosend)。

23.2 当进程为外出的数据报指定了IP选项时，上一题会是什么答案？

23.3 UDP 客户需要调用bind吗？为什么？

23.4 如果插口没有连接上，并且图 23-15中调用M_PREPEND失败，那么在udp_output里，处理器优先级会发生什么变化？

23.5 udp_output不检测目的端口0。它可能发送一个具有目的端口 0的UDP数据报吗？

23.6 假定当把一个数据报发送到一个广播地址时， IP_RECVDSTADDR插口选项有效，你如何确定这个地址是否是一个广播地址？

23.7 谁释放udp_saveopt(图23-38)分配的mbuf？

23.8 进程如何断连连接上的 UDP插口？也就是说，进程调用 connect并与对等实体交换数据报，然后进程要断连插口。允许它调用 sendto，并向其他主机发送数据报。

23.9 我们在图22-25的讨论中，注意到一个用外部IP地址255.255.255.255调用connect的UDP应用程序，在接口上发送时，是把该接口对应的广播地址作为目的IP地址。如果UDP应用使用未连接的插口，用目的地址255.255.255.255调用sendto，会发生什么情况？