第14章 IP多播选路

14.1 引言

前面两章讨论了在一个网络上的多播。本章我们讨论在整个互联网上的多播。我们将讨论mrouted程序的执行，该程序计算多播路由表，以及在网络之间转发多播数据报的内核函数。

从技术上说，多播分组(packet)被转发。本章我们假定每个多播分组中都包含一个完整数据报(也就是说，没有分片)，所以我们只用名词数据报(datagram)。Net/3转发IP分片，也转发IP数据报。

图14-1是mrouted的几个版本及它们和BSD版本的对应关系。mrouted版本包括用户级守护程序和内核级多播程序。

mrouted版本	描　述
1.2	修改4.3 BSD Tahoe版本
2.0	包括在4.4 BSD和Net/3中
3.3	修改SunOS 4.1.3

图14-1 mrouted 和IP多播版本

IP多播技术是一个活跃的研究和开发领域。本章讨论包括在Net/3中的多播软件的2.0版，但被认为已经过时了。3.3版的发行还有一段时间，因此无法在本书中完整地讨论，但我们在整个过程中将指出3.3版本的一些特点。

因为还没有广泛安装商用多播路由器，所以常用多播隧道连接标准IP单播互联网上的两个多播路由器，构造多播网络。Net/3支持多播隧道，并采用宽松源站记录路由(LSRR，Loose Source Record Route)选项(9.6节)构造多播隧道。一种更好的隧道技术把IP多播数据报封装在一个单播数据报里，3.3版的多播程序支持这一技术，但Net/3不支持。

与第12章一样，我们用通常名称运输层协议代指发送和接收多播数据报的协议，但UDP是唯一支持多播的Internet协议。

14.2 代码介绍

本章讨论的三个文件显示在图14-2中。

文　件	描　述
netinet/ip_mroute.h	多播结构定义
netinet/ ip_mroute.c	多播选路函数
netinet/raw_ip.c	多播选路选项

图14-2 本章讨论的文件

14.2.1 全局变量

多播选路程序所使用的全局变量显示在图14-3中。

变 量	数据类型	描 述
cached_mrt	struct mrt	多播选路的“后面一个”高速缓存
cached_origin	u_long	“后面一个”高速缓存的多播组
cached_originmask	u_long	“后面一个”高速缓存的多播组的掩码
mrtstat	struct mrtstat	多播选路统计
mrttable	struct mrt *[]	指向多播路由器的指针的散列表
numvifs	vifi_t	允许的多播接口数
viftable	struct vif[]	虚拟多播接口的数组

图14-3 本章介绍的全局变量

14.2.2 统计量

多播选路程序收集的所有统计信息都放在图 14-4的mrtstat结构中。图14-5是在执行netstat -gs命令后，输出的统计信息。

mrtstat成员	描 述	SNMP使用的
mrts_mrt_lookups	查找的多播路由数	
mrts_mrt_misses	高速缓存丢失的多播路由数	
mrts_grp_lookups	查找的组地址数	
mrts_grp_misses	高速缓存丢失的组地址数	
mrts_no_route	查找失败的多播路由数	
mrts_bad_tunnel	有错误的隧道选项的分组数	
mrts_cant_tunnel	没有空间存放隧道选项的分组数	

图14-4 本章收集的统计量

netstat -gs 输出	mrtstat 成员
multicast routing:	
329569328 multicast route lookups	mrts_mrt_lookups
9377023 multicast route cache misses	mrts_mrt_misses
242754062 group address lookups	mrts_grp_lookups
159317788 group address cache misses	mrts_grp_misses
65648 datagrams with no route for origin	mrts_no_route
0 datagrams with malformed tunnel options	mrts_bad_tunnel
0 datagrams with no room for tunnel options	mrts_cant_tunnel

图14-5 IP多播路由选择统计的例子

这些统计信息来自一个有两个物理接口和一个隧道接口的系统。它们说明， 98%的时间，在高速缓存中发现多播路由。组地址高速缓存的效率稍低一些，最高只有 34%。图14-34描述了路由缓存，图14-21描述了组地址高速缓存。

14.2.3 SNMP变量

多播选路没有标准的SNMP MIB，但 [McCloghrie和Farinacci 1994a] 和 [McCloghrie和Farinacci 1994b] 描述一些多播路由器的实验MIB。

14.3 多播输出处理(续)

12.15节讲到如何为输出的多播数据报选择接口。我们看到在 ip_moptions结构中

ip_output被传给一个明确的接口，或者ip_output在路由表中查找目的组，并使用在路由入口中返回的接口。

如果在选择了输出的接口后，ip_output回送该数据报，就把它放在所选输出接口等待输入处理，当ipintr处理它时，把它当作是要转发的数据报。图14-6显示了这个过程。

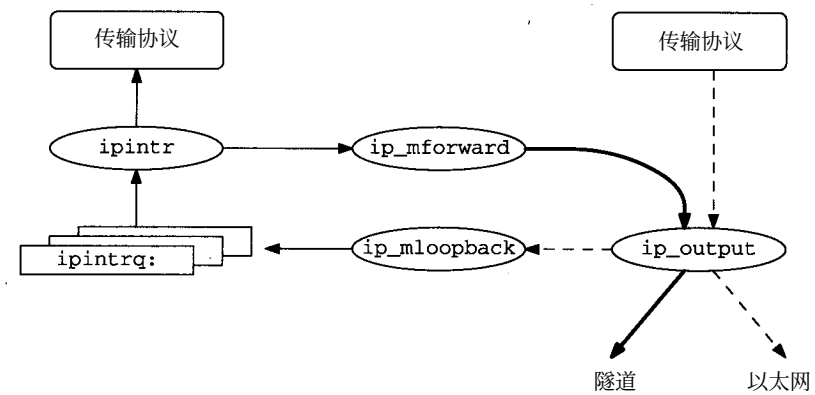

图14-6 有环回的多播输出处理

在图14-6中，虚线箭头代表原始输出的数据报，本例是本地以太网上的多播。ip_mloopback创建的备份由带箭头的细线表示；并作为输入被传给运输层协议。当ip_mforward决定通过系统上的另一个接口转发该数据报时，就产生第三个备份。图14-6中最粗的箭头代表第三个备份，在多播隧道上发送。

如果数据报不是回送的，则ip_output把它直接传给ip_mforward，ip_mforward复制并处理该数据报，就像它是从ip_output选定的接口上收到的一样。图14-7显示了这个过程。

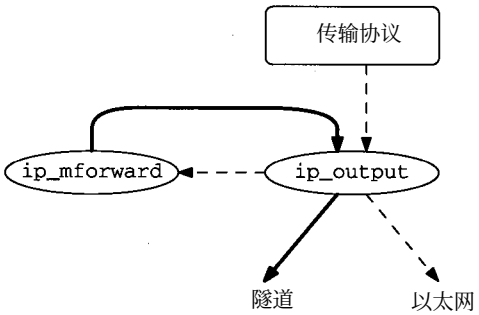

图14-7 没有环回的多播输出处理

一旦ip_mforward调用ip_output发送多播数据报，它就把IP_FORWARDING置位，这样，ip_output就不再把数据报传回给ip_mforward，以免导致无限循环。

图12-42显示了ip_mloopback。14.8节描述了ip_mforward。

14.4 mrouted守护程序

用户级进程mrouted守护程序允许和管理多播路由选择。mrouted实现IGMP协议的路由部分，并与其他多播路由器通信，实现网络间的多播路由选择。路由算法在mrouted上实现，但内核维护多播路由选择表，并转发数据报。

本书中我们只讨论支持mrouted的内核数据结构和函数——不讨论mrouted本身。我们讨论用于为数据报选择路由的截断逆向路径广播TRPB(Truncated Reverse Path Broadcast)算法[Deering和Cheriton 1990]，以及用于在多播路由器之间传递信息的距离向量多播选路协议DVMRP。我们力求使读者了解内核多播程序的工作原理。

RFC 1075 [Waitzman、Partidge 和Deering1988] 是DVMRP的一个老版本。mrouted实现了一个新的 DVMRP，还没有用 RFC文档写出来。目前，该算法和协议的最好的文档是mrouted发布的源代码。附录B指出在哪里能找到到源代码。

mrouted守护程序通过在一个IGMP插口上设置选项与内核通信 (第32章)。这些选项总结在图14-8中。

optname	optval类型	函 数	描 述
DVMRP_INIT		ip_mrouter_init	mrouted开始
DVMRP_DONE		ip_mrouter_done	mrouted被关闭
DVMRP_ADD_VIF	struct vifctl	add_vif	增加虚拟接口
DVMRP_DEL_VIF	vifi_t	del_vif	删除虚拟接口
DVMRP_ADD_LGRP	struct lgrplctl	add_lgrp	为某个接口增加多播组入口
DVMRP_DEL_LGRP	struct lgrplctl	del_lgrp	为某个接口删除多播组入口
DVMRP_ADD_MRT	struct mrtctl	add_mrt	增加多播路由
DVMRP_DEL_MRT	struct in addr	del_mrt	删除多播路由

图14-8 多播路由插口选项

图14-8显示的插口选项被setsockopt系统调用传给rip_ctloutput(32.8节)。图14-9显示了处理DVMRP_*xxx* 选项的rip_ctloutput部分。

raw_ip.c

```
173     case DVMRP_INIT:
174     case DVMRP_DONE:
175     case DVMRP_ADD_VIF:
176     case DVMRP_DEL_VIF:
177     case DVMRP_ADD_LGRP:
178     case DVMRP_DEL_LGRP:
179     case DVMRP_ADD_MRT:
180     case DVMRP_DEL_MRT:
181         if (op == PRCO_SETOPT) {
182             error = ip_mrouter_cmd(optname, so, *m);
183             if (*m)
184                 (void) m_free(*m);
185         } else
186             error = EINVAL;
187         return (error);
```

raw_ip.c

图14-9 rip_ctloutput 函数：DVMRP_ *xxx* 插口选项

173-187 当调用setsockopt时，op等于PRCO_SETOPT，而且所有选项都被传给ip_mrouter_cmd函数。对于getsockopt系统调用，op等于PRCO_GETOPT；对所有选项都返回EINVAL。

图14-10显示了ip_mrouter_cmd函数。

ip_mroute.c

```
84 int
85 ip_mrouter_cmd(cmd, so, m)
86 int     cmd;
87 struct socket *so;
88 struct mbuf *m;
89 {
90     int     error = 0;
```

图14-10 ip_mrouter_cmd 函数

```
 91      if (cmd != DVMRP_INIT && so != ip_mrouter)
 92          error = EACCES;
 93      else
 94          switch (cmd) {
 95          case DVMRP_INIT:
 96              error = ip_mrouter_init(so);
 97              break;
 98          case DVMRP_DONE:
 99              error = ip_mrouter_done();
100              break;
101          case DVMRP_ADD_VIF:
102              if (m == NULL || m->m_len < sizeof(struct vifctl))
103                      error = EINVAL;
104              else
105                  error = add_vif(mtod(m, struct vifctl *));
106              break;
107          case DVMRP_DEL_VIF:
108              if (m == NULL || m->m_len < sizeof(short))
109                      error = EINVAL;
110              else
111                  error = del_vif(mtod(m, vifi_t *));
112              break;
113          case DVMRP_ADD_LGRP:
114              if (m == NULL || m->m_len < sizeof(struct lgrplctl))
115                      error = EINVAL;
116              else
117                  error = add_lgrp(mtod(m, struct lgrplctl *));
118              break;
119          case DVMRP_DEL_LGRP:
120              if (m == NULL || m->m_len < sizeof(struct lgrplctl))
121                      error = EINVAL;
122              else
123                  error = del_lgrp(mtod(m, struct lgrplctl *));
124              break;
125          case DVMRP_ADD_MRT:
126              if (m == NULL || m->m_len < sizeof(struct mrtctl))
127                      error = EINVAL;
128              else
129                  error = add_mrt(mtod(m, struct mrtctl *));
130              break;
131          case DVMRP_DEL_MRT:
132              if (m == NULL || m->m_len < sizeof(struct in_addr))
133                      error = EINVAL;
134              else
135                  error = del_mrt(mtod(m, struct in_addr *));
136              break;
137          default:
138              error = EOPNOTSUPP;
139              break;
140          }
141      return (error);
142  }
```

ip_mroute.c

图14-10 (续)

这些“选项”更像命令，因为它们引起内核更新多个数据结构。本章后面我们将使用命令(command)一词强调这个事实。

84-92 mrouted发布的第一个命令必须是 DVMRP_INIT。后续命令必须来自发布DVMRP_INIT的同一插口。当在其他插口上发布其他命令时，返回 EACCES。

94-142 switch语句的每个case语句检查每条命令中的数据量是否正确，然后调用匹配函数。如果不能识别该命令，则返回 EOPNOTSUPP。任何从匹配函数返回的错误都在 error中发布，并在函数的最后返回。

初始化时，mrouted发布DVMRP_INIT命令，调用图14-11显示的ip_mrouter_init。

ip_mroute.c

```
146 static int
147 ip_mrouter_init(so)
148 struct socket *so;
149 {
150     if (so->so_type != SOCK_RAW ||
151         so->so_proto->pr_protocol != IPPROTO_IGMP)
152         return (EOPNOTSUPP);
153     if (ip_mrouter != NULL)
154         return (EADDRINUSE);
155     ip_mrouter = so;
156     return (0);
157 }
```

ip_mroute.c

图14-11 ip_mrouter_init 函数：DVMRP_INIT 命令

146-157 如果不是在某个原始IGMP插口上发布命令，或者如果DVMRP_INIT已经被置位，则分别返回EOPNOTSUPP和EADDRINUSE。全局变量ip_mrouter保存指向某个插口的指针，初始化命令就是在该插口上发布的。必须在该插口上发布后续命令。以避免多个 mrouted进程的并行操作。

下面几节讨论其他DVMRP_*xxx*命令。

14.5 虚拟接口

当作为多播路由器运行时， Net/3接收到达的多播数据报，复制它们，并在一个或多个接口上转发备份。通过这种方式，数据报被转发给互联网上的其他多播路由器。

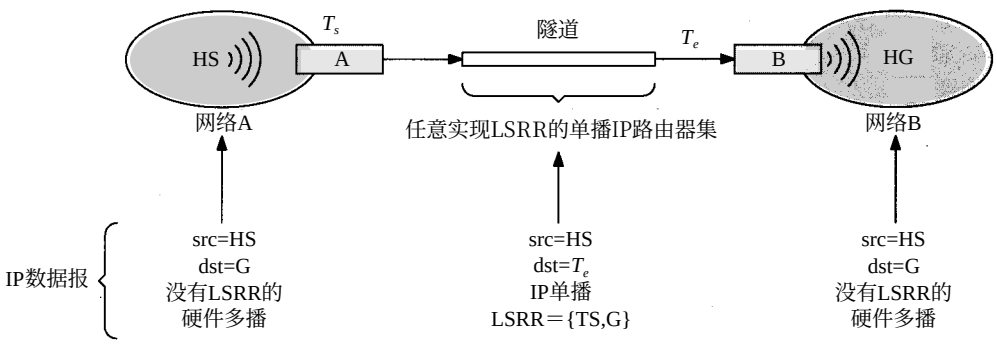

图14-12 多播隧道

输出的接口可以是一个物理接口，也可以是一个多播隧道。多播隧道的两端都与一个多播路由器上的某个物理接口相关。多播隧道使两个多播路由器，即使被不能转发多播数据报的路由器分隔，也能够交换多播数据报。图14-12是一个多播隧道连接的两个多播路由器。

图14-12中，网络A上的源主机HS正在向组G多播数据报。组G的唯一成员在网络B上，并通过一个多播隧道连接到网络A。路由器A接收多播(因为多播路由器接收所有多播)，查询它的多播路由选择表，并通过多播隧道转发该数据报。

隧道的开始是路由器A上的一个物理接口，以IP单播地址T_s标识。隧道的结束是网络B上的一个物理接口，以IP单播地址T_e标识。隧道本身是一个任意复杂的网络，由实现LSRR选项的IP单播路由器连接起来。图14-13显示IP LSRR选项如何实现多播隧道。

系统	IP首部		源路由选项		描　述
	ip_src	ip_dst	偏移	地　址	
HS	HS	G			在网络A上
T_s	HS	T_e	8	T_s • G	在隧道上
T_e	HS	G	12	T_s　见正文	在路由器B上ip_dooptions之后
T_e	HS	G			在路由器B上ip_mforward之后

图14-13　LSRR多播隧道选项

图14-13的第一行是HS在网络A上发送的多播数据报。路由器A全部接收，因为多播路由器接收本地连接的网络上的所有数据报。

为通过隧道发送数据报，路由器A在IP首部插入一个LSRR选项。第二行是在隧道上离开A时的数据报。LSRR选项的第一个地址是隧道的源地址，第二个地址是目的多播组地址。数据报的目的地址是T_e——隧道的另一端。LSRR偏移指向目的组。

经过隧道的数据报被转发，通过互联网，直到它到达路由器B上的隧道的另一端。

该图中的第三行是被路由器B上的ip_dooptions处理之后的数据报。记得第9章中讲到，ip_dooptions在ipintr检查数据报的目的地址之前处理LSRR选项。因为数据报的目的地址(T_e)和路由器B上的一个接口匹配，所以ip_dooptions把由选项偏移(本例中是G)标识的地址复制到IP首部的目的地址字段。在选项内，G被ip_rtaddr返回的地址取代，ip_rtaddr通常根据IP目的地址(本例中是G)为数据报选择输出的接口。这个地址是不相关的，因为ip_mforward将丢弃整个选项。最后，ip_dooptions把选项偏移向前移动。

图14-13的第四行是ipintr调用ip_mforward之后的数据报。在那里，LSRR选项被识别，并从数据报首部中移走。得到的数据报看起来就象原始多播数据报，由ip_mforward处理它，把它作为多播数据报在网络B上转发，并被HG收到。

用LSRR构造的多播隧道已经过时了。因为1993年3月发布了mrouted程序，该程序通过在IP多播数据报的首部前面加上另一个IP首部来构造隧道。新IP首部的协议设置为4，表明分组的内容是另一个IP分组。有关这个值的文档在RFC 1700——“IP中的IP”协议中。新版本的mrouted程序为了向后兼容，也支持LSRR隧道。

14.5.1　虚拟接口表

无论物理接口还是隧道接口，内核都为其在虚拟接口(virtual interface)表中维护一个入口，其中包含了只有多播使用的信息。每个虚拟接口都用一个vif结构表示(图14-14)。全局变量

viftable是一个这种结构的数组。数组的下标保存在无符号短整数 vifi_t变量中。

ip_mroute.h

```
105 struct vif {
106     u_char   v_flags;              /* VIFF_ flags */
107     u_char   v_threshold;          /* min ttl required to forward on vif */
108     struct in_addr v_lcl_addr;     /* local interface address */
109     struct in_addr v_rmt_addr;     /* remote address (tunnels only) */
110     struct ifnet *v_ifp;           /* pointer to interface */
111     struct in_addr *v_lcl_grps;    /* list of local grps (phyints only) */
112     int      v_lcl_grps_max;       /* malloc'ed number of v_lcl_grps */
113     int      v_lcl_grps_n;         /* used number of v_lcl_grps */
114     u_long   v_cached_group;       /* last grp looked-up (phyints only) */
115     int      v_cached_result;      /* last look-up result (phyints only) */
116 };
```

ip_mroute.h

图14-14 vif 结构

105-110 为v_flags定义的唯一的标志位是VIFF_TUNNEL。被置位时，该接口是一个到远程多播路由器的隧道。没有置位时，接口是在本地系统上的一个物理接口。 v_threshold是我们在12.9节描述的多播阈值。v_lcl_addr是与这个虚拟接口相关的本地接口的 IP地址。v_rmt_addr是一个IP多播隧道远端的单播IP地址。v_lcl_addr或者v_rmt_addr为非零，但不会两者都为非零。对物理接口， v_ifp非空，并指向本地接口的 ifnet结构。对隧道，v_ifp是空的。

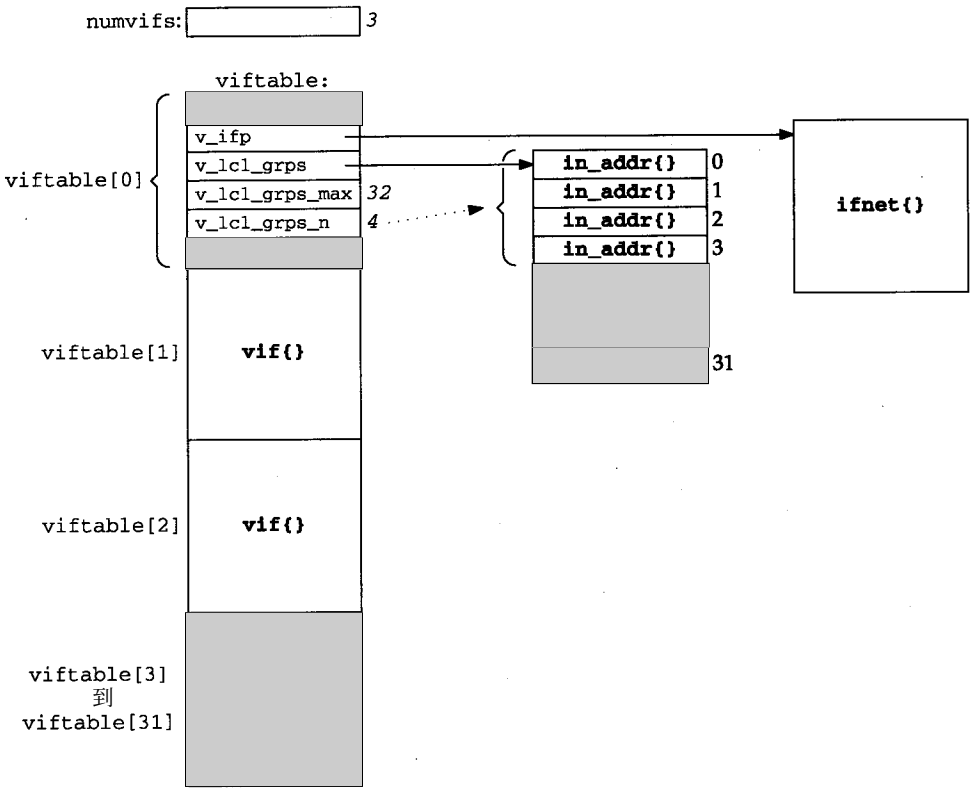

图14-15 viftable 数组

111-116 v_lcl_grps指向一个IP 多播组地址数组，这个数组记录了在连到的接口上的成员组列表。对隧道来说，v_lcl_grps总是空的。数组的大小保存在v_lcl_grps_max中，被使用的入口数保存在 v_lcl_grps_n中。数组随着组成员关系表的增大而增长。v_cached_group和v_cached_result实现“一个入口”高速缓存，其中记录的是最近一次查找得到的组。

图14-15说明了viftable，它最多有32个(MAXVIFS)入口。viftable[2]是正在使用的最后一个入口，所以numvifs是3。编译内核时固定了表的大小。图中还显示了表的第一个入口的vif结构的几个成员。v_ifp指向一个ifnet结构，v_lcl_grps指向in_addr结构中的一个数组。数组有32(v_lcl_grps_max)个入口，其中只用了4个(v_lcl_grps_n)。

mrouted通过DVMRP_ADD_VIF和DVMRP_DEL_VIF命令维护viftable。通常，当mrouted开始运行时，会把本地系统上有多播能力的接口加入表中。当 mrouted阅读自己的配置文件，通常是/etc/mrouted.conf时，会把多播隧道加入表中。这个文件中的命令也可能从虚拟接口表中删除物理接口，或者改变与接口有关的多播信息。

mrouted用DVMRP_ADD_VIF命令把ctl结构(图14-16)传给内核。它指示内核在虚拟接口表中加入一个接口项。

ip_mroute.h

```
76 struct vifctl {
77     vifi_t  vifc_vifi;          /* the index of the vif to be added */
78     u_char  vifc_flags;         /* VIFF_ flags (Figure 14.14) */
79     u_char  vifc_threshold;     /* min ttl required to forward on vif */
80     struct in_addr vifc_lcl_addr;    /* local interface address */
81     struct in_addr vifc_rmt_addr;    /* remote address (tunnels only) */
82 };
```

ip_mroute.h

图14-16 vifctl 结构

78-82 vifc_vifi识别viftable中虚拟接口的下标。其他 4个成员，vifc_flags、vifc_threshold、vifc_lcl_addr和vifc_rmt_addr，被add_vif函数复制到vif函数中。

14.5.2 add_vif函数

图14-17是add_vif函数。

ip_mroute.c

```
202 static int
203 add_vif(vifcp)
204 struct vifctl *vifcp;
205 {
206     struct vif *vifp = viftable + vifcp->vifc_vifi;
207     struct ifaddr *ifa;
208     struct ifnet *ifp;
209     struct ifreq ifr;
210     int     error, s;
211     static struct sockaddr_in sin =
212     {sizeof(sin), AF_INET};

213     if (vifcp->vifc_vifi >= MAXVIFS)
214         return (EINVAL);
215     if (vifp->v_lcl_addr.s_addr != 0)
216         return (EADDRINUSE);
```

图14-17 add_vif 函数：DVMRP_ADD_VIF 命令

```
217     /* Find the interface with an address in AF_INET family */
218     sin.sin_addr = vifcp->vifc_lcl_addr;
219     ifa = ifa_ifwithaddr((struct sockaddr *) &sin);
220     if (ifa == 0)
221         return (EADDRNOTAVAIL);
222     s = splnet();
223     if (vifcp->vifc_flags & VIFF_TUNNEL)
224         vifp->v_rmt_addr = vifcp->vifc_rmt_addr;
225     else {
226         /* Make sure the interface supports multicast */
227         ifp = ifa->ifa_ifp;
228         if ((ifp->if_flags & IFF_MULTICAST) == 0) {
229             splx(s);
230             return (EOPNOTSUPP);
231         }
232         /*
233          * Enable promiscuous reception of all IP multicasts
234          * from the interface.
235          */
236         satosin(&ifr.ifr_addr)->sin_family = AF_INET;
237         satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
238         error = (*ifp->if_ioctl) (ifp, SIOCADDMULTI, (caddr_t) & ifr);
239         if (error) {
240             splx(s);
241             return (error);
242         }
243     }
244     vifp->v_flags = vifcp->vifc_flags;
245     vifp->v_threshold = vifcp->vifc_threshold;
246     vifp->v_lcl_addr = vifcp->vifc_lcl_addr;
247     vifp->v_ifp = ifa->ifa_ifp;
248     /* Adjust numvifs up if the vifi is higher than numvifs */
249     if (numvifs <= vifcp->vifc_vifi)
250         numvifs = vifcp->vifc_vifi + 1;
251     splx(s);
252     return (0);
253 }
```

ip_mroute.c

图14-17 (续)

1. 验证下标

202-216 如果mrouted指定的vifc_vifi中的下标太大，或者该表入口已经被使用，则分别返回EINVAL或EADDRINUSE。

2. 本地物理接口

217-221 ifa_ifwithaddr取得vifc_lcl_addr中的单播IP地址，并返回一个指向相关ifnet结构的指针。这就标识出这个虚拟接口要用的物理接口。如果没有匹配的接口，返回EADDRNOTAVAIL。

3. 配置隧道接口

222-224 对于隧道，它的远端地址被从vifctl结构中复制到接口表的vif结构中。

4. 配置物理接口

225-243 对于物理接口，链路级驱动程序必须支持多播。 SIOCADDMULTI命令与

INADDR_ANY一起配置接口，开始接收所有IP多播数据报(图12-32)，因为它是一个多播路由器。当ipintr把到达数据报传给ip_mforward时，被ip_mforward转发。

5. 保存多播信息

244-253 其他接口信息被从vifctl结构复制到vif结构。如果需要，更新numvifs，记录正在使用的虚拟接口数。

14.5.3 del_vif函数

图14-18显示的del_vif函数从虚拟接口表中删除表项。当 mrouted设置DVMRP_DEL_VIF命令时，调用该函数。

1. 验证下标

257-268 如果传给del_vif的下标大于正在使用的最大下标，或者指向一个没有使用的入口，则分别返回EINVAL和EADDRNOTAVAIL。

ip_mroute.c

```
257 static int
258 del_vif(vifip)
259 vifi_t *vifip;
260 {
261     struct vif *vifp = viftable + *vifip;
262     struct ifnet *ifp;
263     int      i, s;
264     struct ifreq ifr;

265     if (*vifip >= numvifs)
266         return (EINVAL);
267     if (vifp->v_lcl_addr.s_addr == 0)
268         return (EADDRNOTAVAIL);

269     s = splnet();

270     if (!(vifp->v_flags & VIFF_TUNNEL)) {
271         if (vifp->v_lcl_grps)
272             free(vifp->v_lcl_grps, M_MRTABLE);
273         satosin(&ifr.ifr_addr)->sin_family = AF_INET;
274         satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
275         ifp = vifp->v_ifp;
276         (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);
277     }
278     bzero((caddr_t) vifp, sizeof(*vifp));

279     /* Adjust numvifs down */
280     for (i = numvifs - 1; i >= 0; i--)
281         if (viftable[i].v_lcl_addr.s_addr != 0)
282             break;
283     numvifs = i + 1;

284     splx(s);
285     return (0);
286 }
```

ip_mroute.c

图14-18 del_vif 函数：DVMRP_DEL_VIF 命令

2. 删除接口

269-278 对于物理接口，释放本地多播组表， SIOCADDMULTI禁止接收所有多播数据报，bzero对viftable的入口清零。

3. 调整接口计数

279-286 for循环从以前活动的最大入口开始向后直到第一个入口为止，搜索出第一个活动的入口。对没有使用的入口，v_lcl_addr(一个in_addr结构)的成员s_addr是0。相应地更新numvifs，函数返回。

14.6 IGMP(续)

第13章侧重于IGMP协议的主机部分，mrouted实现了这个协议的路由器部分。mrouted必须为每个物理接口记录哪个多播组有成员在连到的网络上。 mrouted每120秒多播一个IGMP_HOST_MEMBERSHIP_QUERY数据报，并把IGMP_HOST_MEMBERSHIP_REPORT的结果汇编到与每个网络相关的成员关系数组中。这个数组不是我们在第13章讲的成员关系表。

mrouted根据收集到的信息构造多播路由选择表。多播组表也提供信息，用来抑制向没有目的组成员的多播互联网区进行多播。

只为物理接口维护这样的成员关系数组。对其他多播路由器来说，隧道是点到点接口，所以无需组成员关系信息。

我们在图14-14中看到，v_lcl_grps指向一个IP多播组数组。mrouted用DVMRP_ADD_LGRP和DVMRP_DEL_LGRP命令维护这个表。两个命令都带了一个 lgrpctl结构(图14-19)。

ip_mroute.h

```
87 struct lgrplctl {
88     vifi_t  lgc_vifi;
89     struct in_addr lgc_gaddr;
90 };
```

ip_mroute.h

图14-19 lgrpctl 结构

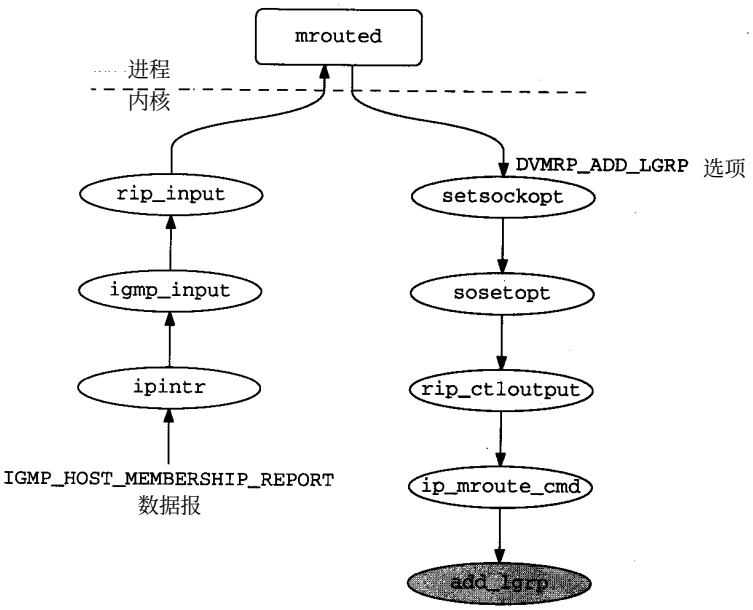

图14-20 IGMP报告处理

87-90 lgc_vifi和lgc_gaddr标识{接口，组}对。接口下标(无符号短整数lgc_vifi)标识一个虚拟接口，而不是物理接口。

当收到一个IGMP_HOST_MEMBERSHIP_REPORT时，调用图14-20所示的函数。

14.6.1 add_lgrp函数

mrouted检查到达IGMP报告的源地址，确定是哪个子网，从而确定报告是哪个接口接

ip_mroute.c

```
291 static int
292 add_lgrp(gcp)
293 struct lgrplctl *gcp;
294 {
295     struct vif *vifp;
296     int     s;

297     if (gcp->lgc_vifi >= numvifs)
298         return (EINVAL);

299     vifp = viftable + gcp->lgc_vifi;
300     if (vifp->v_lcl_addr.s_addr == 0 || (vifp->v_flags & VIFF_TUNNEL))
301         return (EADDRNOTAVAIL);

302     /* If not enough space in existing list, allocate a larger one */
303     s = splnet();
304     if (vifp->v_lcl_grps_n + 1 >= vifp->v_lcl_grps_max) {
305         int     num;
306         struct in_addr *ip;

307         num = vifp->v_lcl_grps_max;
308         if (num <= 0)
309             num = 32;           /* initial number */
310         else
311             num += num;         /* double last number */
312         ip = (struct in_addr *) malloc(num * sizeof(*ip),
313                                        M_MRTABLE, M_NOWAIT);
314         if (ip == NULL) {
315             splx(s);
316             return (ENOBUFS);
317         }
318         bzero((caddr_t) ip, num * sizeof(*ip));     /* XXX paranoid */
319         bcopy((caddr_t) vifp->v_lcl_grps, (caddr_t) ip,
320               vifp->v_lcl_grps_n * sizeof(*ip));

321         vifp->v_lcl_grps_max = num;
322         if (vifp->v_lcl_grps)
323             free(vifp->v_lcl_grps, M_MRTABLE);
324         vifp->v_lcl_grps = ip;

325         splx(s);
326     }
327     vifp->v_lcl_grps[vifp->v_lcl_grps_n++] = gcp->lgc_gaddr;

328     if (gcp->lgc_gaddr.s_addr == vifp->v_cached_group)
329         vifp->v_cached_result = 1;

330     splx(s);
331     return (0);
332 }
```

ip_mroute.c

图14-21 add_lgrp 函数：DVMRP_ADD_GLRP 命令

收的。根据这个信息，mrouted为该接口设置DVMRP_ADD_LGRP命令，更新内核中的成员关系表。这个信息也被送到多播路由选择算法，更新路由选择表。图 14-21显示了add_lgrp函数。

1. 验证增加请求

291-301 如果该请求标识了一个无效接口，就返回 EINVAL。如果没有使用该接口或它是一个隧道，则返回EADDRNOTAVAIL。

2. 如果需要，扩展组数组

302-326 如果新组无法放在当前的组数组中，就分配一个新的数组。第一次为接口调用add_lgrp函数时，分配一个能装32个组的数组。

每次数组被填满后，add_lgrp就分配一个两倍于前面数组大小的新数组。 Malloc负责分配， bzero负责清零， bcopy把旧数组中的内容复制到新数组中。更新最大入口数v_lcl_grps_max，释放旧数组(如果有的话)，把新数组和v_lcl_grps连接到vif入口。

“偏执狂(paranoid)”评论指出，无法保证malloc分配的内存全部是0。

3. 增加新的组

327-332 新组被复制到下一个可用的入口，如果高速缓存中已经存放了新组，就把高速缓存标记为有效。

查找高速缓存中包含一个地址 v_cached_group，以及一个高速缓存的查找结果v_cached_result。grplst_member函数在搜索成员关系数组之前，总是先查一下这个高速缓存。如果给定的组与v_cached_group匹配，就返回高速缓存的查找结果；否则，搜索成员关系数组。

14.6.2 del_lgrp函数

如果在270秒内，没有收到该组任何成员关系的报告，则每个接口的组信息超时。mrouted维护适当的定时器，并当信息超时后，发布 DVMRP_DEL_LGRP命令。图14-22显示了del_lgrp。

1. 验证接口下标

337-347 如果请求标识无效接口，叫返回 EINVAL。如果该接口没有使用或是一个隧道，则返回EADDRNOTAVAIL。

2. 更新查找高速缓存

348-350 如果要删除的组在高速缓存里，就把查找结果设成 0(假)。

3. 删除组

351-364 如果在成员关系表中没有找到该组，则在 error中发布EADDRNOTAVAIL。for循环搜索与该接口相关的成员关系数组。如果 same(是一个宏，用bcmp比较两个地址)为真，则清除error，把组计数器加1。bcopy移动后续的数组入口，删除该组， del_lgrp跳出该循环。

如果循环结束，没有找到匹配，则返回 EADDRNOTAVAIL；否则返回0。

ip_mroute.c

```
337 static int
338 del_lgrp(gcp)
339 struct lgrplctl *gcp;
340 {
341     struct vif *vifp;
342     int     i, error, s;

343     if (gcp->lgc_vifi >= numvifs)
344         return (EINVAL);
345     vifp = viftable + gcp->lgc_vifi;
346     if (vifp->v_lcl_addr.s_addr == 0 || (vifp->v_flags & VIFF_TUNNEL))
347         return (EADDRNOTAVAIL);

348     s = splnet();

349     if (gcp->lgc_gaddr.s_addr == vifp->v_cached_group)
350         vifp->v_cached_result = 0;

351     error = EADDRNOTAVAIL;
352     for (i = 0; i < vifp->v_lcl_grps_n; ++i)
353         if (same(&gcp->lgc_gaddr, &vifp->v_lcl_grps[i])) {
354             error = 0;
355             vifp->v_lcl_grps_n--;
356             bcopy((caddr_t) & vifp->v_lcl_grps[i + 1],
357                   (caddr_t) & vifp->v_lcl_grps[i],
358                   (vifp->v_lcl_grps_n - i) * sizeof(struct in_addr));
359             error = 0;
360             break;
361         }
362     splx(s);
363     return (error);
364 }
```

ip_mroute.c

图14-22 del_lgrp 函数：DVMRP_ DEL_LGRP命令

14.6.3 grplst_member函数

在转发多播时，查询成员关系数组，以免把数据报发到没有目的组成员的网络上。图 14-23显示的grplst_member函数，搜索整个表，寻找给定组地址。

ip_mroute.c

```
368 static int
369 grplst_member(vifp, gaddr)
370 struct vif *vifp;
371 struct in_addr gaddr;
372 {
373     int     i, s;
374     u_long  addr;

375     mrtstat.mrts_grp_lookups++;

376     addr = gaddr.s_addr;
377     if (addr == vifp->v_cached_group)
378         return (vifp->v_cached_result);

379     mrtstat.mrts_grp_misses++;

380     for (i = 0; i < vifp->v_lcl_grps_n; ++i)
```

图14-23 grplst_member 函数

```
381         if (addr == vifp->v_lcl_grps[i].s_addr) {
382             s = splnet();
383             vifp->v_cached_group = addr;
384             vifp->v_cached_result = 1;
385             splx(s);
386             return (1);
387         }
388     s = splnet();
389     vifp->v_cached_group = addr;
390     vifp->v_cached_result = 0;
391     splx(s);
392     return (0);
393 }
```

ip_mroute.c

图14-23 (续)

1. 检查高速缓存

368-379 如果请求的组在高速缓存中，则返回高速缓存的结果，不搜索成员关系数组。

2. 搜索成员关系数组

380-390 对数组进行线性搜索，确定组是否在其中。如果找到，就更新高速缓存以记录匹配的值，并返回1；如果没有找到，就更新高速缓存记录丢失的，并返回0。

14.7 多播选路

正如在本章开始提到的，我们不给出 `mrouted`实现的TRPB算法，但给出一个有关该机制的综述，描述内核的多播路由选择表和多播路由选择函数。图 14-24显示了一个我们用于解释该算法的示例多播网络。

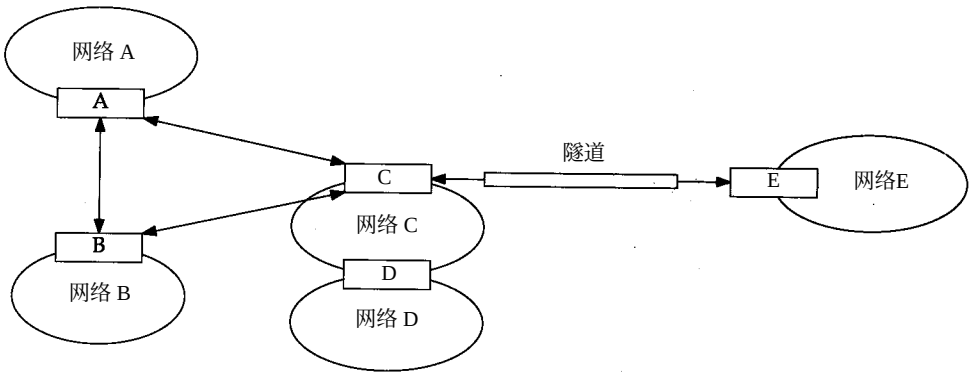

图14-24 多播网络示例

图14-24中，方框代表路由器，椭圆代表连接到路由器的多播网络。例如，路由器 D可以在网络D和网络C上多播。路由器C可以向网络C多播，通过点到点接口向路由器 A和B多播，并可以通过一个多播隧道向路由器E多播。

最简单的路由选择办法是，从互联网拓扑中选出一个子网，形成一个生成树。如果每个路由器都沿着生成树转发多播，则各路由器最终会收到数据报。图 14-25显示了示例网络的一个生成树。其中，网络A上的主机S是多播数据报的源。

有关生成树的讨论，参见 [Tanenbaum 1989] 或 [Perlman 1992]。

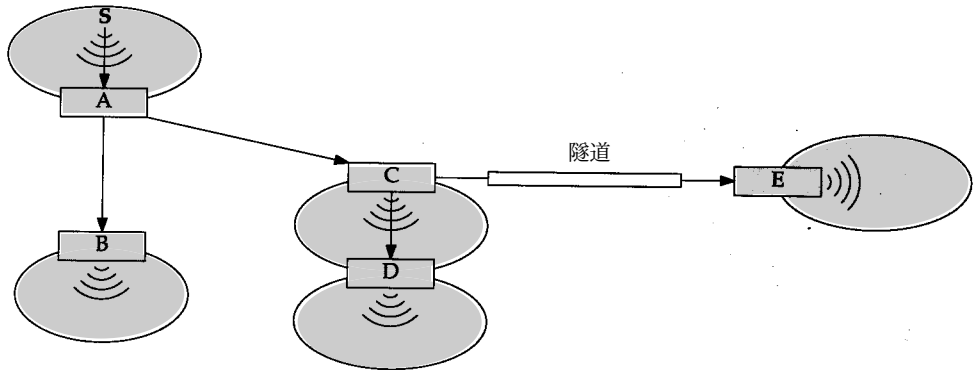

图14-25 网络A的生成树

这个生成树是根据从各网络回到网络A上的源站的最短逆向路径(*reverse path*)构造的。图14-25的生成树中，省略了路由器B和C之间的线路。源站和路由器A之间的箭头，以及路由器B和C之间的箭头，强调了多播网络是生成树的一部分。

如果用同一生成树转发来自网络C的数据报，为了在网络B上收到，数据报经过的转发路径将大于需要的长度。RFC 1075提出的算法为每个潜在的源站计算了一个单独的生成树，以避免这种情况。路由选择表为每条路由记录了一个网络号和子网掩码，所以一条路由可以应用到源子网内的任意主机。

因为构造生成树是为了给源站的数据报提供最短逆向路径，而每个网络都接收所有多播数据报，所以这个过程称为逆向路径广播(*reverse path broadcast*)即RPB。

RPB没有任何多播组成员信息，使许多数据报被不必要地转发到没有目的组成员的网络上。如果，除了计算生成树外，该路由选择算法还能记录哪些网络是叶子，注意到每个网络上的组成员关系，那么，连到叶子网络的路由器就可以避免把数据报转发到没有目的组成员的网络上去。这称为截断逆向路径广播(TRPB)，2.0版的`mrouted`在IGMP帮助下记录叶子网络上的成员关系，从而实现这一算法。

图14-26显示了TRPB算法的应用。多播来自网络C上的源站，并在网络B上有一个目的组成员。

我们用图14-26说明Net/3多播路由选择表中使用的名词。在这个例子中，有阴影的网络和

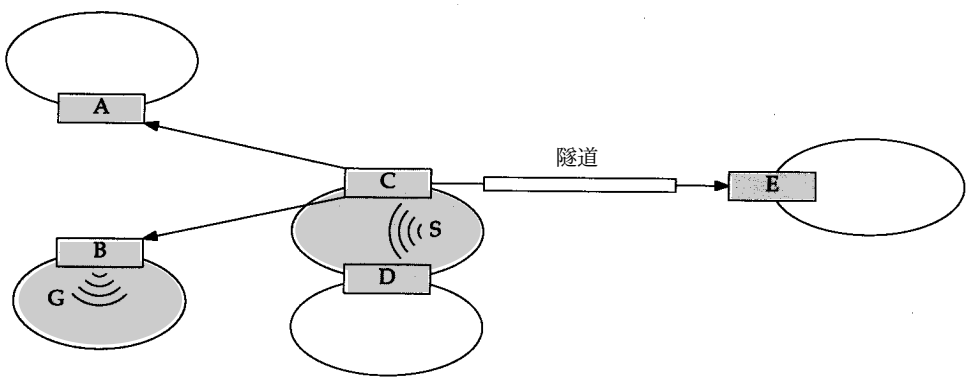

图14-26 网络C的TRPB路由选择

路由器收到来自网络C上源站的数据报。A和B之间的线路不属于生成树，C与D之间没有连接，因为C和D直接收到源站发送的多播。

在这个图中，网络A、B、D和E是叶子网络。路由器C接收多播，并通过连到路由器A、B和E的接口将其转发——尽管把它发给A和E都是浪费。这是TRPB算法的缺点。

路由器C上与网络C相关的接口叫做父亲，因为路由器C期望用它接收来自网络C的多播。从路由器C到路由器A、B和E的接口叫做儿子接口。对路由器A来说，点到点接口是来自C的源分组的父亲，到网络A的接口是儿子。接口相对于数据报的源站，被标识为父亲和儿子。只在相关的儿子接口上转发多播数据报，不在父亲接口上转发多播。

继续我们的例子，因为网络A、D和E是叶子网络，并且没有目的组成员，所以它们没有阴影。在路由器处截断生成树，也不把数据报转发到这些网络上去。路由器B把数据报转发到网络B上，因为B上有一个目的组成员。为实现截断算法，接收数据报的所有路由器都在自己的viftable中查询与每个虚拟接口相关的组表。

对该多播路由选择算法的最后一个改进叫做逆向路径多播 (reverse path multicasting，RPM)。RPM的目的是修剪(prune)各生成树，避免在没有目的组成员的分支上发送数据报。在图14-26中，RPM可以避免路由器C向A和E发送数据报，因为在这两个分支上没有目的多播组的成员。3.3版的mrouted实现了RPM。

图14-27是我们的示例网络，但这一次，只有那些RPM算法选路数据报能到达的路由器和网络才有阴影。

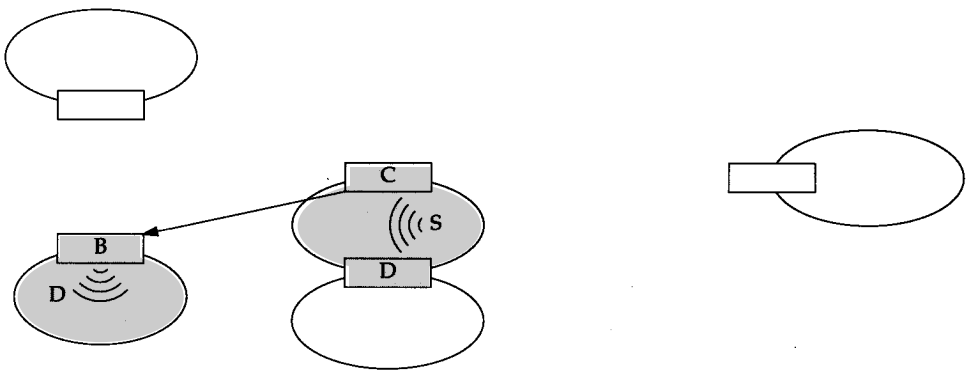

图14-27 网络C的RMP路由选择

为了计算生成树对应的路由表，多播路由器和邻近的多播路由器通信，发现多播互联网拓扑和多播组成员的位置。在Net/3中，用DVMRP进行这种通信。DVMRP作为IGMP数据报传送，发给224.0.0.4组，该组是给DVMRP通信保留的(图12-1)。

在图12-39中，我们看到，多播路由器总是接受到达的IGMP分组，把它们传给igmp_input和rip_input，然后mrouted在一个原始IGMP插口上读它们。mrouted把DVMRP报文发送到同一原始IGMP插口上的其他多播路由器。

实现这些算法需要的有关RPB、TRPB、RPM以及DVMRP报文的其他细节参见[Deering和Cheriton 1990]和mrouted的源代码。

Internet上还使用了其他多播路由选择协议。Proteon路由器实现了RFC 1584 [Moy 1994]提出的MOSPF协议。Cisco从操作软件的10.2版开始实现了PIM(Protocol Independent

Multicasting)。[Deering et al1994]描述了PIM。

14.7.1 多播选路表

现在我们描述Net/3中实现的多播路由选择。内核的多播路由选择表是作为一个有 64个入口的散列表实现的 (MRTHASHIZ)。该表保存在全局数组 mrttable中，每个入口指向一个mrt结构的链表，如图14-28所示。

ip_mroute.h

```
120 struct mrt {
121     struct in_addr mrt_origin;  /* subnet origin of multicasts */
122     struct in_addr mrt_originmask;  /* subnet mask for origin */
123     vifi_t  mrt_parent;         /* incoming vif */
124     vifbitmap_t mrt_children;   /* outgoing children vifs */
125     vifbitmap_t mrt_leaves;     /* subset of outgoing children vifs */
126     struct mrt *mrt_next;       /* forward link */
127 };
```

ip_mroute.h

图14-28 mrt 结构

120-127 mrtc_origin和mrtc_originmask标识表中的一个入口。mrtc_parent是虚拟接口的下标，该虚拟接口上预期有来自起点的所有多播数据报。 mrtc_children是一个位图，标识外出的接口。 mrtc_leaves也是一个位图，里面标识多播路由选择树中也是叶子的外出接口。当多条路由散列到同一个数组入口时，最后一个成员 mrt_next实现该入口的一个链表。

图14-29是多播选路表的整体结构。各 mrt结构都放在一个散列链上，该散列链与nethash(图14-31)函数返回的值对应。

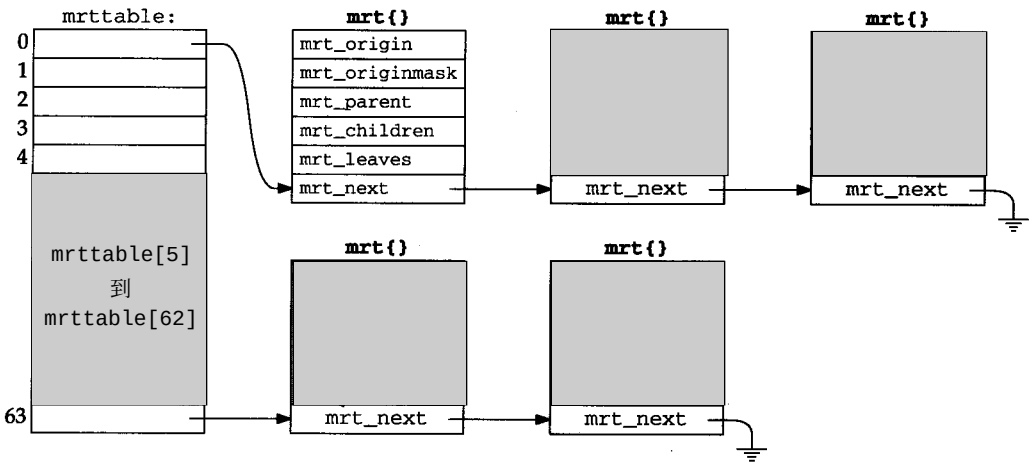

图14-29 多播选路表

内核维护的多播选路表是mrouted维护的多播选路表的一个子集，其中的信息足够内核支持多播转发。发送内核表更新和 DVMRP_ADD_MRT命令，其中包含图 14-30显示的mrtctl结构。

95-101 mrtctl结构的5个成员携带了我们谈到的mrouted和内核之间的信息(图14-28)。

多播选路表的键值是多播数据报的源 IP地址。nethash(图14-31)实现该用于该表的散列

算法。它接受源IP地址，并返回0~63之间的一个值(MRTHASHSIZ-1)。

ip_mroute.h

```
 95 struct mrtctl {
 96     struct in_addr mrtc_origin; /* subnet origin of multicasts */
 97     struct in_addr mrtc_originmask;     /* subnet mask for origin */
 98     vifi_t  mrtc_parent;         /* incoming vif */
 99     vifbitmap_t mrtc_children;   /* outgoing children vifs */
100     vifbitmap_t mrtc_leaves;     /* subset of outgoing children vifs */
101 };
```

ip_mroute.h

图14-30 mrtctl 结构

ip_mroute.c

```
398 static  u_long
399 nethash(in)
400 struct in_addr in;
401 {
402     u_long  n;

403     n = in_netof(in);
404     while ((n & 0xff) == 0)
405         n >>= 8;
406     return (MRTHASHMOD(n));
407 }
```

ip_mroute.c

图14-31 nethash 结构

398-407 in_netof返回in，主机部分设置为全0，在n中仅留下发送主机的A、B和C类网络。右移结果，直到低8位非零为止。MRTHASHMOD是

```
#define MRTHASHMOD(h) ((h) & (MRTHASHSIZ - 1))
```

把低8位与63进行逻辑与运算，留下低6位，这是0~63之间的一个整数。

用两个函数调用(nethash和in_netof)计算散列值，作为散列32 bit地址值太过昂贵了。

14.7.2 del_mrt函数

mrouted守护程序通过DVMRP_ADD_MRT和DVMRP_DEL_MRT命令在内核的多播选路表中增加或删除表项。图14-32显示了del_mrt函数。

ip_mroute.c

```
451 static int
452 del_mrt(origin)
453 struct in_addr *origin;
454 {
455     struct mrt *rt, *prev_rt;
456     u_long  hash = nethash(*origin);
457     int     s;

458     for (prev_rt = rt = mrttable[hash]; rt; prev_rt = rt, rt = rt->mrt_next)
459         if (origin->s_addr == rt->mrt_origin.s_addr)
460             break;
461     if (!rt)
462         return (ESRCH);

463     s = splnet();
```

图14-32 del_mrt 函数：DVMRP_DEL_MRT 命令

```
464     if (rt == cached_mrt)
465         cached_mrt = NULL;
466     if (prev_rt == rt)
467         mrttable[hash] = rt->mrt_next;
468     else
469         prev_rt->mrt_next = rt->mrt_next;
470     free(rt, M_MRTABLE);
471     splx(s);
472     return (0);
473 }
```

ip_mroute.c

图14-32 (续)

1. 找到路由入口

451-462 for循环从hash标识的入口开始(在nethash中定义时初始化)。如果没有找到入口，则返回ESRCH。

2. 删除路由入口

463-473 如果该入口在高速缓存中，则高速缓存也无效了。从散列链上把该入口断开，并且释放。当匹配入口在表的最前面时，需要if语句处理这一特殊情况。

14.7.3 add_mrt函数

add_mrt函数如图14-33所示。

ip_mroute.c

```
411 static int
412 add_mrt(mrtcp)
413 struct mrtctl *mrtcp;
414 {
415     struct mrt *rt;
416     u_long  hash;
417     int     s;
418     if (rt = mrtfind(mrtcp->mrtc_origin)) {
419         /* Just update the route */
420         s = splnet();
421         rt->mrt_parent = mrtcp->mrtc_parent;
422         VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
423         VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
424         splx(s);
425         return (0);
426     }
427     s = splnet();
428     rt = (struct mrt *) malloc(sizeof(*rt), M_MRTABLE, M_NOWAIT);
429     if (rt == NULL) {
430         splx(s);
431         return (ENOBUFS);
432     }
433     /*
434      * insert new entry at head of hash chain
435      */
436     rt->mrt_origin = mrtcp->mrtc_origin;
437     rt->mrt_originmask = mrtcp->mrtc_originmask;
438     rt->mrt_parent = mrtcp->mrtc_parent;
```

图14-33 add_mrt 函数：处理DVMRP_ADD_MRT 命令

```
439     VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
440     VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
441     /* link into table */
442     hash = nethash(mrtcp->mrtc_origin);
443     rt->mrt_next = mrttable[hash];
444     mrttable[hash] = rt;

445     splx(s);
446     return (0);
447 }
```

ip_mroute.c

图14-33 (续)

1. 更新存在的路由

411-427 如果请求的路由已经在路由表中，则把新的信息复制到该路由中，add_mrt返回。

2. 分配新路由

428-447 在新分配的mbuf中，根据增加请求传递的mrtctl结构，构造一个mrt结构。从mrtc_origin计算出散列下标，并把新路由插入散列链的第一个入口。

14.7.4 mrtfind函数

mrtfind函数负责搜索多播选路表。如图14-34所示。把数据报的源站地址传给mrtfind，mrtfind返回一个指向匹配mrt结构的指针；如果没有匹配，则返回一个空指针。

1. 检查路由查询高速缓存

477-488 把给定的源IP地址(orgin)与高速缓存中的原始掩码做逻辑与运算。如果结果与cached_origin匹配，则返回高速缓存的入口。

ip_mroute.c

```
477 static struct mrt *
478 mrtfind(origin)
479 struct in_addr origin;
480 {
481     struct mrt *rt;
482     u_int   hash;
483     int     s;

484     mrtstat.mrts_mrt_lookups++;

485     if (cached_mrt != NULL &&
486         (origin.s_addr & cached_originmask) == cached_origin)
487         return (cached_mrt);

488     mrtstat.mrts_mrt_misses++;

489     hash = nethash(origin);
490     for (rt = mrttable[hash]; rt; rt = rt->mrt_next)
491         if ((origin.s_addr & rt->mrt_originmask.s_addr) ==
492             rt->mrt_origin.s_addr) {
493             s = splnet();
494             cached_mrt = rt;
495             cached_origin = rt->mrt_origin.s_addr;
496             cached_originmask = rt->mrt_originmask.s_addr;
497             splx(s);
498             return (rt);
499         }
500     return (NULL);
501 }
```

ip_mroute.c

图14-34 mrtfind 函数

2. 检查散列表

489-501 nethash返回该路由入口的散列下标。for循环搜索散列链找到匹配的路由。当找到一个匹配时，更新高速缓存，返回一个指向该路由的指针。如果没有找到匹配，则返回一个空指针。

14.8 多播转发：ip_mforward函数

内核实现了整个多播转发。我们在图 12-39中看到，当ip_mrouter非空时，也就是mrouted在运行时，ipintr把到达数据报传给ip_mforward。

我们在图 12-40中看到，ip_output可以把本地主机产生的多播数据报传给 ip_mforward，由ip_mforward为这些数据报选路到除ip_output选定的接口以外的其他接口上去。

与单播转发不同，每当多播数据报被转发到某个接口上时，就为该数据报产生一个备份。例如，如果本地主机是一个多播路由器，并且连接到三个不同的网络，则系统产生的多播数据报被分别复制三份，在三个接口上等待输出。另外，如果应用程序设置了多播环回标志位，或者任何输出的接口也接收它自己的传送，则数据报也将被复制，等待输入。

图14-35显示了一个到达某个物理接口的多播数据报。

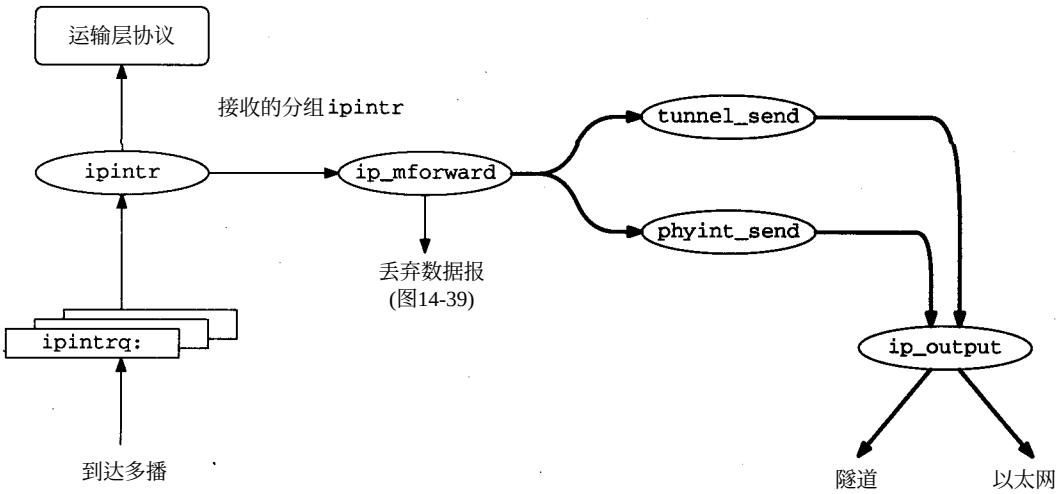

图14-35 到达某个物理接口的多播数据报

在图14-35中，数据报到达的接口是目的多播组的一个成员，所以数据报被传给运输层协议等待输入处理。该数据报也被传给 ip_mforward，在这里它被复制和转发到一个物理接口和一个隧道上(带粗线的箭头)，这两个必须都不和接收接口相同。

图14-36显示了一个到达某隧道的多播数据报。

在图14-36中，用带虚线的箭头表示与该隧道的本地端有关的物理接口，数据报就在这一接口上到达。数据报被传给ip_mforward，我们将在图14-37看到，因为分组到达一个隧道，所以ip_mforward返回一个非零值。这导致ipintr不再把该分组传给运输层协议。

ip_mforward从分组中取出隧道选项，查询多播选路表，并且，在本例中，还把分组转发到另一个隧道以及到达的物理接口上去，用带细线的箭头表示。这是可行的，因为多播选

路表是根据虚拟接口，而不是物理接口。

在图14-36中，我们假定物理接口是目的多播组的成员，所以ip_output把该数据报传给ip_mloopback，ip_mloopback把它送到队列中等待ipintr的处理(带粗线的箭头)。然后，分组又被传给ip_mforward，并被这个函数丢弃（练习14.4)。这一次，ip_mforward返回0(因为分组是在物理接口上到达的)，所以ipintr接受该数据报，并进行输入处理。

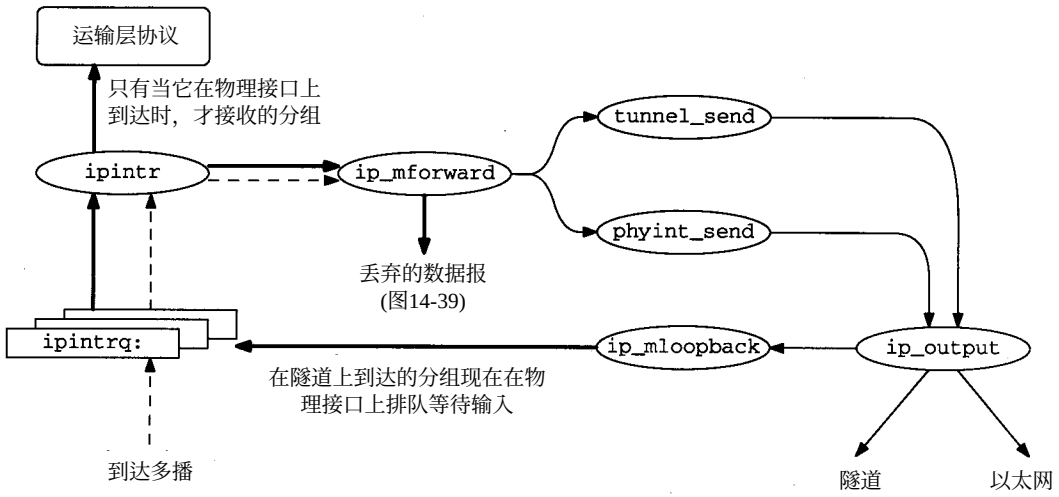

图14-36 到达某个多播隧道的多播数据报

我们分三部分说明多播转发程序：

- 隧道输入处理(图14-37)；
- 转发条件合格(图14-39)；和
- 转发到出去的接口上(图14-40)。

ip_mroute.c

```
516 int
517 ip_mforward(m, ifp)
518 struct mbuf *m;
519 struct ifnet *ifp;
520 {
521     struct ip *ip = mtod(m, struct ip *);
522     struct mrt *rt;
523     struct vif *vifp;
524     int     vifi;
525     u_char *ipoptions;
526     u_long  tunnel_src;

527     if (ip->ip_hl < (IP_HDR_LEN + TUNNEL_LEN) >> 2 ||
528         (ipoptions = (u_char *) (ip + 1))[1] != IPOPT_LSRR) {
529         /* Packet arrived via a physical interface. */
530         tunnel_src = 0;
531     } else {
532         /*
533          * Packet arrived through a tunnel.
534          * A tunneled packet has a single NOP option and a
```

图14-37 ip_mforward 函数：到达隧道

```
535          * two-element loose-source-and-record-route (LSRR)
536          * option immediately following the fixed-size part of
537          * the IP header.  At this point in processing, the IP
538          * header should contain the following IP addresses:
539          *
540          * original source           - in the source address field
541          * destination group         - in the destination address field
542          * remote tunnel end-point   - in the first  element of LSRR
543          * one of this host's addrs - in the second element of LSRR
544          *
545          * NOTE: RFC-1075 would have the original source and
546          * remote tunnel end-point addresses swapped.  However,
547          * that could cause delivery of ICMP error messages to
548          * innocent applications on intermediate routing
549          * hosts!  Therefore, we hereby change the spec.
550          */
551         /* Verify that the tunnel options are well-formed.  */
552         if (ipoptions[0] != IPOPT_NOP ||
553             ipoptions[2] != 11 ||    /* LSRR option length    */
554             ipoptions[3] != 12 ||    /* LSRR address pointer */
555             (tunnel_src = *(u_long *) (&ipoptions[4])) == 0) {
556             mrtstat.mrts_bad_tunnel++;
557             return (1);
558         }
559         /* Delete the tunnel options from the packet. */
560         ovbcopy((caddr_t) (ipoptions + TUNNEL_LEN), (caddr_t) ipoptions,
561                 (unsigned) (m->m_len - (IP_HDR_LEN + TUNNEL_LEN)));
562         m->m_len -= TUNNEL_LEN;
563         ip->ip_len -= TUNNEL_LEN;
564         ip->ip_hl -= TUNNEL_LEN >> 2;
565     }
```

ip_mroute.c

图14-37 (续)

516-526 ip_mforward的两个参数是：一个指向包含该数据报的 mbuf链的指针；另一个是指向接收接口ifnet结构的指针。

1. 到达物理接口

527-530 为了区分在同一物理接口上到达的多播数据报是否经过隧道，要检查 IP首部的特征LSRR选项。如果首部太小，无法包含该选项；或者该选项不是以一个后面跟着一个 LSRR选项的NOP开始，就假定该数据报是在一个物理接口上到达的，并把 tunnel_src设为0。

2. 到达隧道

531-558 如果数据报看起来像是从隧道上到达的，就检查选项，验证格式是否正确。如果选项的格式不符合多播隧道，则 ip_mforward返回1，指示应该把该数据报丢弃。图 14-38是隧道选项的结构。

在图14-38中，我们假定数据报里没有其他选项，但不是必须这样的。任何其他IP选项都可能出现在 LSRR选项的后面，因为隧道开始端的多播路由器总是把 LSRR选项插在所有其他选项之前。

3. 删除隧道选项

559-565 如果选项正确，就把后面的选项和数据向前移动，调整 mbuf首部的m_len和IP 首部的ip_len和ip_hl的值，然后删除隧道选项(图14-38)。

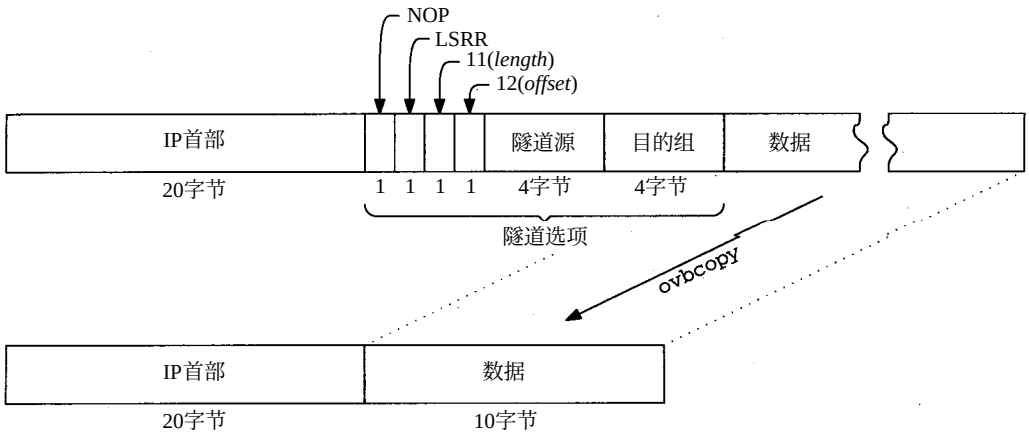

图14-38 多播隧道选项

ip_mforward经常把tunnel_source作为返回值。当数据报从隧道上到达时，这个值只能是非零的。当ip_mforward返回非零值时，它的调用方就丢弃该数据报。对 ipintr来说，这意味着在隧道上到达的一个数据报被传给 ip_mforward，并且被ipintr丢弃。转发程序取出隧道信息，复制数据报，用 ip_output将其发送出去；如果接口是目的多播组的成员，则ip_output调用ip_mloopback。

ip_mforward的下一部分显示在图 14-39中，在这部分程序中，如果数据报不符合转发的条件，就丢弃它。

ip_mroute.c

```
566     /*
567      * Don't forward a packet with time-to-live of zero or one,
568      * or a packet destined to a local-only group.
569      */
570     if (ip->ip_ttl <= 1 ||
571         ntohl(ip->ip_dst.s_addr) <= INADDR_MAX_LOCAL_GROUP)
572         return ((int) tunnel_src);

573     /*
574      * Don't forward if we don't have a route for the packet's origin.
575      */
576     if (!(rt = mrtfind(ip->ip_src))) {
577         mrtstat.mrts_no_route++;
578         return ((int) tunnel_src);
579     }
580     /*
581      * Don't forward if it didn't arrive from the parent vif for its origin.
582      */
583     vifi = rt->mrt_parent;
584     if (tunnel_src == 0) {
585         if ((viftable[vifi].v_flags & VIFF_TUNNEL) ||
586             viftable[vifi].v_ifp != ifp)
587             return ((int) tunnel_src);
588     } else {
589         if (!(viftable[vifi].v_flags & VIFF_TUNNEL) ||
590             viftable[vifi].v_rmt_addr.s_addr != tunnel_src)
591             return ((int) tunnel_src);
592     }
```

ip_mroute.c

图14-39 ip_mforward 函数：转发可行性检查

4. 超时的TTL或本地多播

566-572 如果ip_ttl是0或1，那么数据报已经到了生存期的最后，不再转发它。如果目的组小于或等于INADDR_MAX_LOCAL_GROUP(几个224.0.0.x组，图12-1)，则不允许数据报离开本地网络，也不转发它。在两种情况下，都把tunnel_src返回给调用方。

3.3版的mrouted支持对某些目的多播组的管理辖域。可把接口配置成丢弃所有寻址到这些组的数据报，与224.0.0.x组的自动辖域类似。

5. 没有路由可用

573-579 如果mrtfind无法根据数据报中的源地址找到一条路由，则函数返回。没有路由，多播路由器无法确定把数据报转发到哪个接口上去。这种情况可能发生在，比如，多播数据报在mrouted更新多播选路表之前到达。

6. 在没有想到的接口上到达

580-592 如果数据报到达某个物理接口，但系统本来预想它应该到达某个隧道或其他物理接口，则ip_mforward返回；如果数据报到达某个隧道，但系统本来预想它应该在某个物理接口或其他隧道上到达，则ip_mforward也返回。产生这些情况的原因是，当组成员关系或网络的物理拓扑发生变化后，正在更新选路表时，数据报到达。

ip_mforward的最后一部分(图14-40)把该数据报在多播路由入口所指定的每个输出接口上发送。

ip_mroute.c

```
593     /*
594      * For each vif, decide if a copy of the packet should be forwarded.
595      * Forward if:
596      *       - the ttl exceeds the vif's threshold AND
597      *       - the vif is a child in the origin's route AND
598      *       - ( the vif is not a leaf in the origin's route OR
599      *           the destination group has members on the vif )
600      *
601      * (This might be speeded up with some sort of cache -- someday.)
602      */
603     for (vifp = viftable, vifi = 0; vifi < numvifs; vifp++, vifi++) {
604         if (ip->ip_ttl > vifp->v_threshold &&
605             VIFM_ISSET(vifi, rt->mrt_children) &&
606             (!VIFM_ISSET(vifi, rt->mrt_leaves) ||
607              grplst_member(vifp, ip->ip_dst))) {
608             if (vifp->v_flags & VIFF_TUNNEL)
609                 tunnel_send(m, vifp);
610             else
611                 phyint_send(m, vifp);
612         }
613     }

614     return ((int) tunnel_src);
615 }
```

ip_mroute.c

图14-40 ip_mforward 函数：转发

593-615 对viftable中的每个接口，如果以下条件满足，则在该接口上发送数据报：

- 数据报的TTL大于接口的多播阈值；
- 接口是该路由的子接口；以及
- 接口没有和某个叶子网络相连。

如果该接口是一个叶子，那么只有当网络上有目的多播组成员时(也就是说，grplst_member返回一个非零值)，才输出该数据报。

tunnel_send在隧道接口上转发该数据报；用phyint_send在物理接口上转发。

14.8.1 phyint_send函数

为在物理接口上发送多播数据报，phyint_send(图14-41)在它传给ip_output的ip_moptions结构中，明确指定了输出接口。

ip_mroute.c

```
616 static void
617 phyint_send(m, vifp)
618 struct mbuf *m;
619 struct vif *vifp;
620 {
621     struct ip *ip = mtod(m, struct ip *);
622     struct mbuf *mb_copy;
623     struct ip_moptions *imo;
624     int     error;
625     struct ip_moptions simo;

626     mb_copy = m_copy(m, 0, M_COPYALL);
627     if (mb_copy == NULL)
628         return;

629     imo = &simo;
630     imo->imo_multicast_ifp = vifp->v_ifp;
631     imo->imo_multicast_ttl = ip->ip_ttl - 1;
632     imo->imo_multicast_loop = 1;

633     error = ip_output(mb_copy, NULL, NULL, IP_FORWARDING, imo);
634 }
```

ip_mroute.c

图14-41 phyint_send 函数

616-634 m_copy复制输出的数据报。ip_moptions结构设置为强制在选定的接口上传送该数据报。递减TTL，允许多播环回。

数据报被传给ip_output。IP_FORWARDING标志位避免产生无限回路，使ip_output再次调用ip_mforward。

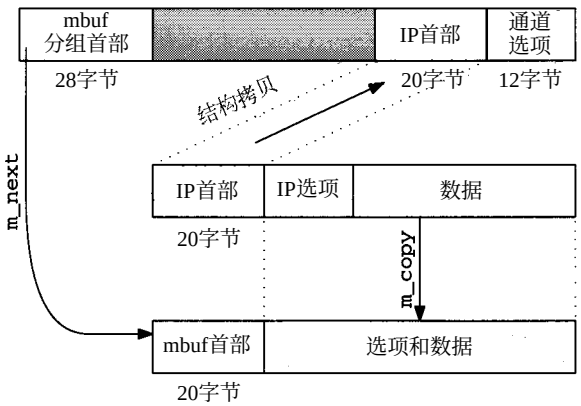

图14-42 插入隧道选项

14.8.2 tunnel_send函数

为了在隧道上发送数据报，tunnel_send(图14-43)必须构造合适的隧道选项，并将其插到输出数据报的首部。图14-42显示了tunnel_send如何为隧道准备分组。

ip_mroute.c

```
635 static void
636 tunnel_send(m, vifp)
637 struct mbuf *m;
638 struct vif *vifp;
639 {
640     struct ip *ip = mtod(m, struct ip *);
641     struct mbuf *mb_copy, *mb_opts;
642     struct ip *ip_copy;
643     int     error;
644     u_char *cp;

645     /*
646      * Make sure that adding the tunnel options won't exceed the
647      * maximum allowed number of option bytes.
648      */
649     if (ip->ip_hl > (60 - TUNNEL_LEN) >> 2) {
650         mrtstat.mrts_cant_tunnel++;
651         return;
652     }
653     /*
654      * Get a private copy of the IP header so that changes to some
655      * of the IP fields don't damage the original header, which is
656      * examined later in ip_input.c.
657      */
658     mb_copy = m_copy(m, IP_HDR_LEN, M_COPYALL);
659     if (mb_copy == NULL)
660         return;
661     MGETHDR(mb_opts, M_DONTWAIT, MT_HEADER);
662     if (mb_opts == NULL) {
663         m_freem(mb_copy);
664         return;
665     }
666     /*
667      * Make mb_opts be the new head of the packet chain.
668      * Any options of the packet were left in the old packet chain head
669      */
670     mb_opts->m_next = mb_copy;
671     mb_opts->m_len = IP_HDR_LEN + TUNNEL_LEN;
672     mb_opts->m_data += MSIZE - mb_opts->m_len;
```

ip_mroute.c

图14-43 tunnel_send 函数：验证和分配新首部

1. 隧道选项合适吗

635-652 如果IP首部内没有隧道选项的空间，tunnel_send立即返回，不再在隧道上转发该数据报。可能在其他接口上转发。

2. 复制数据报，为新首部和隧道选项分配mbuf

653-672 在调用m_copy时，复制的开始偏移是20(IP_HDR_LEN)。产生的mbuf链中包含了数据报的选项和数据报，但没有IP首部。mb_opts指向MGETHDR分配的一个新的数据报首部，这个新的数据报首部被放在mb_copy的前面。然后调整m_len和m_data的值，以容纳IP首部和隧道选项。

tunnel_send的第二部分，如图14-44所示，修改输出分组的首部，并发送该分组。

ip_mroute.c

```
673     ip_copy = mtod(mb_opts, struct ip *);
674     /*
675      * Copy the base ip header to the new head mbuf.
676      */
677     *ip_copy = *ip;
678     ip_copy->ip_ttl--;
679     ip_copy->ip_dst = vifp->v_rmt_addr;     /* remote tunnel end-point */
680     /*
681      * Adjust the ip header length to account for the tunnel options.
682      */
683     ip_copy->ip_hl += TUNNEL_LEN >> 2;
684     ip_copy->ip_len += TUNNEL_LEN;
685     /*
686      * Add the NOP and LSRR after the base ip header
687      */
688     cp = (u_char *) (ip_copy + 1);
689     *cp++ = IPOPT_NOP;
690     *cp++ = IPOPT_LSRR;
691     *cp++ = 11;                 /* LSRR option length */
692     *cp++ = 8;                  /* LSSR pointer to second element */
693     *(u_long *) cp = vifp->v_lcl_addr.s_addr;   /* local tunnel end-point */
694     cp += 4;
695     *(u_long *) cp = ip->ip_dst.s_addr;     /* destination group */

696     error = ip_output(mb_opts, NULL, NULL, IP_FORWARDING, NULL);
697 }
```

ip_mroute.c

图14-44 tunnel_send 函数：构造首部和发送

3. 修改IP首部

673-679 从原始mbuf链中把原始IP 首部复制到新分配的mbuf首部中。减少该首部的TTL，把目的地址改成隧道另一端的接口地址。

4. 构造隧道选项

680-664 调整ip_hl和ip_len的值以容纳隧道选项。隧道选项紧跟在 IP 首部的后面：一个NOP，后面是LSRR码，LSRR选项的长度(11字节)，以及一个指向选项第二个地址的指针 (8字节)。源路由包括了本地隧道端点和后面的目的多播组地址 (图14-13)。

5. 发送经过隧道处理的数据报

665-697 现在，这个数据报看起来像一个有 LSRR选项的单播数据报，因为它的目的地址是隧道另一端的单播地址。ip_output发送该数据报。当数据报到达隧道的另一端时，隧道选项被剥离，另一端可能会通过其他隧道将数据报继续转发。

14.9 清理：ip_mrouter_done函数

当mrouted结束时，它发布DVMRP_DONE命令，ip_mrouter_done函数(图14-45)处理这个命令。

161-186 这个函数在splnet上运行，避免与多播转发代码的任何交互。对每个物理多播接口，释放本地组表，并发布 SIOCDELMULTI命令，阻止接收多播数据报 (练习14.3) 。bzero清零整个viftable数组，并把numvifs设置成0。

187-198 释放多播选路表中的所有活动入口，bzero清零整个表，清零缓存，置位ip_mrouter。

多播选路表中的每个入口都可能是入口链表的第一个。这段代码只释放表的第一个入口，引起内存泄露。

ip_mroute.c

```
161 int
162 ip_mrouter_done()
163 {
164     vifi_t  vifi;
165     int     i;
166     struct ifnet *ifp;
167     int     s;
168     struct ifreq ifr;
169     s = splnet();
170     /*
171      * For each phyint in use, free its local group list and
172      * disable promiscuous reception of all IP multicasts.
173      */
174     for (vifi = 0; vifi < numvifs; vifi++) {
175         if (viftable[vifi].v_lcl_addr.s_addr != 0 &&
176             !(viftable[vifi].v_flags & VIFF_TUNNEL)) {
177             if (viftable[vifi].v_lcl_grps)
178                 free(viftable[vifi].v_lcl_grps, M_MRTABLE);
179             satosin(&ifr.ifr_addr)->sin_family = AF_INET;
180             satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
181             ifp = viftable[vifi].v_ifp;
182             (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);
183         }
184     }
185     bzero((caddr_t) viftable, sizeof(viftable));
186     numvifs = 0;
187     /*
188      * Free any multicast route entries.
189      */
190     for (i = 0; i < MRTHASHSIZ; i++)
191         if (mrttable[i])
192             free(mrttable[i], M_MRTABLE);
193     bzero((caddr_t) mrttable, sizeof(mrttable));
194     cached_mrt = NULL;
195     ip_mrouter = NULL;
196     splx(s);
197     return (0);
198 }
```

ip_mroute.c

图14-45 ip_mrouter_done 函数：DVMRP_DONE 命令

14.10 小结

本章我们描述了网际多播的一般概念和支持它的 Net/3内核中心专用函数。我们没有讨论mrouted的实现，有兴趣的读者可以得到源代码。

我们描述了虚拟接口表，讨论了物理接口和隧道之间的区别，以及 Net/3中用于实现隧道的LSRR选项。

我们说明了RPB、TRPB和RPM算法，描述了根据TRPB转发多播数据报的内核表，还讨论了父网络和叶子网络。

习题

14.1 在图14-25中，需要多少多播路由？

14.2 为什么splnet和splx保护对图14-23中组成员关系高速缓存的更新？

14.3 当某个接口用IP_ADD_MEMBERSHIP选项明确加入一个多播组后，如果向它发布SIOCDELMULTI，会发生什么？

14.4 当某个上隧道上到达一个数据报，并被ip_mforward接收后，可能会在转发到某个物理接口时，被ip_output环回。为什么当环回分组到达该物理接口时，ip_mforward会丢弃它呢？

14.5 重新设计组地址高速缓存，提高它的效率。