第6章 T/TCP的实现：路由表

6.1 概述

T/TCP需要在其每主机高速缓存中为每一个与之进行过通信的主机创建一个记录项。每个记录项包括图2-5所示的tao_cc、tao_ccsent和tao_mssopt三个变量。已有的IP路由表是每主机高速缓存的最合适位置。在 Net/3中，利用卷2第19章介绍的“克隆”标志，很容易为每一个主机创建一个每主机路由表记录项。

在卷2中，我们已经知道网际协议(没有T/TCP)利用了Net/3提供的一般路由表功能。卷2的图18-17说明了调用rn_addroute函数就可增加路由记录，调用rn_delete可以删除路由记录，调用rn_match可以查找路由记录，以及调用rn_walktree可以遍历整棵树(Net/3中用二叉树来存储其路由表，叫做基树(radix tree)。在TCP/IP中，除了这些一般功能外，不再需要有其他功能支持。然而在T/TCP中就不一样了。

既然一个主机可以在一个很短的时间内与成百上千的主机通信(例如几个小时，或者对于一个非常繁忙的WWW服务器来说可能不需要一个小时，详见14.10节的示例)，因此就需要有一些方法使每主机路由表中的路由记录超时作废。本章我们主要研究T/TCP协议在IP路由表中动态创建和删除每主机路由表记录项的功能。

> 卷2中的习题19.2给出了自动地为每一个与之通信的对等主机创建每主机路由表记录项的一个琐细方法。我们在本章中所叙述的方法在概念上与其非常相似，但对大多数TCP/IP路由都能自动进行。习题中创建的每主机路由是不会超时的；创建以后它们就一直存在，直到主机再次启动或者管理人员手工删除。这就需要有一个更好的方法来自动地管理所有的每主机路由。
>
> 并非每一个人都认为已有的路由表是开设T/TCP每主机高速缓存的好地方。另一个方法是将T/TCP每主机高速缓存在内核中作为其自身基树来存储。这项技术(一棵分立的基树)容易实现，利用了内核中已有的一般基树功能，在Net/3的网络文件系统NFS中就采用了这个方法。

6.2 代码介绍

C语言文件netinet/in_rmx.c中定义了T/TCP为TCP/IP的路由功能所增加的函数。这个文件中只包含了我们在本章中所介绍的专门用于Internet的函数。我们将不会介绍卷2第18、19和20章中所叙述的所有路由函数。

图6-1中给出了专门用于Internet的新增路由函数(在本章中介绍的函数用带阴影椭圆表示，函数名字用in_开头)和一般路由函数(这些函数的名字通常用rn_或rt开头)之间的关系。

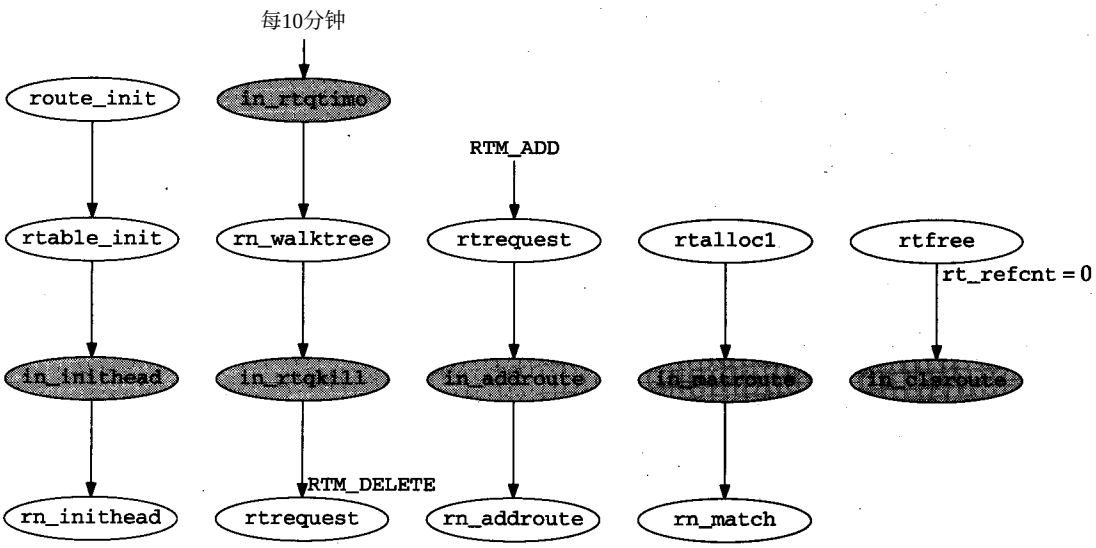

图6-1 专用于Internet的路由函数之间的关系

全局变量

图6-2中给出了专用于Internet的新增全局变量。

FreeBSD版允许系统管理员用sysctl程序修改图6-2中最后三个变量的值，程序要加前缀net.inet.ip。我们没有给出完成这个功能的程序代码，因为它只是对卷2图8-35中的ip_sysctl函数作了一些小小的补充。

变 量	数据类型	说 明
rtq_timeout	int	in_rtqtimo运行的频率(默认值= 每一次10分钟)
rtq_toomany	int	在动态删除开始前有多少路由
rtq_reallyold	int	路由已经确实很陈旧时，存在了多长时间
rtq_minreallyold	int	rtq_reallyold最小值

图6-2 专用于Internet的全局路由变量

6.3 radix_node_head结构

在radix_node_head结构中新增加了一个指针 (卷2的图18-16)：rnh_close。当它指向in_clsroute时，除了指向某个IP路由表外，它的值总是空的，见后面的图 6-7中。

这个函数指针用在rtfree函数中。卷2的图19-5中的第108行和第109行之间要加上下面这些程序行，以说明并初始化自动变量 rnh：

```
struct radix_node_head *rnh = rt_tables[rt_key(rt)->sa_family];
```

以下3行程序则加在第112~113行之间：

```
if(rnh->rnh_close && rt-> rt- refcnt == 0) {
        rnh->rnh_close((struct radix_node *)rt, rnh);
}
```

如果这个函数指针非空，并且引用计数达到0，就要调用关闭函数。

6.4 rtentry结构

T/TCP需要在rtentry结构中增加两个路由标志(卷2第464页)。但是现有的rt_flags项是一个16位短整数，并且15位已经占用了(卷2第464页)。为此要在rtentry结构中新增一个标志项rt_prflags。

另一个解决的方法是将短整数rt_flags改为长整数，这种做法在将来的版本中可能会有。

T/TCP使用了rt_prflags的两个标志位。

- RTPRF_WASCLONED是由rtrequest设置的(卷2第488页第335~336行)，从设置了RTF_CLONING标志的记录项创建一个新的记录项时就要设置该标志。
- RTPRF_OURS是由in_clsroute设置的(图6-7)，当IP路由的最后一个克隆参考项关闭时就要设置该标志。这时，要设置一个定时器，以便在将来的某个时间将这个路由表项删除。

6.5 rt_metrics结构

T/TCP修改路由表的目的是在每个路由表记录项中存储附加的每主机信息，实际上也就是三个变量：tao_cc、tao_ccsent和tao_mssopt。为了容纳这些附加的信息，在rt_metrics结构(卷2第464页)中就需要有一个新的字段：

```
u_long rmx_filler[4];    /* protocal family specific metrics */
```

这就有了一个16字节的协议专用向量，T/TCP可以利用，如图6-3所示：

tcp_var.h

```
153 struct rmxp_tao {
154     tcp_cc  tao_cc;             /* latest CC in valid SYN from peer */
155     tcp_cc  tao_ccsent;         /* latest CC sent to peer */
156     u_short tao_mssopt;         /* latest MSS received from peer */
157 };

158 #define rmx_taop(r) ((struct rmxp_tao *)(r).rmx_filler)
```

tcp_var.h

图6-3 T/TCP用作TAO高速缓存的rmxp_tao 结构

153-157 tcp_cc数据类型用于连接计数，是用typedef定义的一个无符号长整数(类似于TCP的序号)。tcp_cc变量的值为0，表示它还未定义。

158 当给定一个指向rtentry结构的指针，宏rmx_taop就返回一个指向相应rmxp_tao结构的指针值。

6.6 in_inithead函数

卷2第504页详细介绍了Net/3中路由表初始化工作的所有步骤。T/TCP所做的第一项修改是将inetdomain结构中的dom_rtattach字段指向in_inithead，而不是指向rn_inithead(卷2第151页)。图6-4中给出了in_inithead函数。

1. 执行路由表的初始化

222-225 rn_inithead用于分配并初始化一个radix_node_head结构。在Net/3中也就

这些功能。该函数的其他功能是 T/TCP新增加的，并且仅仅在“实”路由表初始化时执行。在NFS装载点初始化另一个路由表时也会调用这个函数。

2. 改变函数指针

226-229 rn_inithead还要将radix_node_head结构中取默认值的两个函数指针修改为rnh_addaddr和rnh_matchaddr。它们也是在卷2图18-17中给出的四个指针中的两个。这就使得在调用一般基结点函数前可以执行 Internet中专有的一些动作。 rnh_close函数指针是T/TCP中新加的。

in_rmx.c

```
218 int
219 in_inithead(void **head, int off)
220 {
221     struct radix_node_head *rnh;

222     if (!rn_inithead(head, off))
223         return (0);

224     if (head != (void **) &rt_tables[AF_INET])
225         return (1);             /* only do this for the real routing table */

226     rnh = *head;
227     rnh->rnh_addaddr = in_addroute;
228     rnh->rnh_matchaddr = in_matroute;
229     rnh->rnh_close = in_clsroute;
230     in_rtqtimo(rnh);            /* kick off timeout first time */
231     return (1);
232 }
```

in_rmx.c

图6-4 in_inithead 函数

3. 初始化超时函数

230 in_rtqtimo是超时函数，是第一次调用。这个函数的每一次调用，它总会安排在将来再次被调用。

6.7 in_addroute函数

用rtrequest可以创建一个新的路由表记录项，它们或者是 RTM_ADD命令的结果，也可能是RTM_RESOLVE命令的结果。这两个命令都会从已经存在并且设置了克隆标志的记录项中创建一个新的记录项(卷2第488~489页)。创建以后就要调用rnh_addaddr函数，我们在Internet协议中看到的是in_addroute函数。图6-5给出了这个新函数。

in_rmx.c

```
47 static struct radix_node *
48 in_addroute(void *v_arg, void *n_arg, struct radix_node_head *head,
49             struct radix_node *treenodes)
50 {
51     struct rtentry *rt = (struct rtentry *) treenodes;

52     /*
53      * For IP, all unicast non-host routes are automatically cloning.
54      */
55     if (!(rt->rt_flags & (RTF_HOST | RTF_CLONING))) {
56         struct sockaddr_in *sin = (struct sockaddr_in *) rt_key(rt);
57         if (!IN_MULTICAST(ntohl(sin->sin_addr.s_addr))) {
```

图6-5 in_addroute 函数

```
58              rt->rt_flags |= RTF_CLONING;
59          }
60      }
61      return (rn_addroute(v_arg, n_arg, head, treenodes));
62  }
```
in_rmx.c

图6-5 (续)

52-61 如果所增加的路由不是一个主机路由，也没有设置克隆标志，这时就要检查路由表的主键(IP地址)。如果IP地址不是一个多播地址，该新创建的路由表记录项就要设置克隆标志。rn_addroute为路由表增加记录项。

这个函数的功能是为所有非多播网络路由设置克隆标志，包括默认的路由。这个克隆标志的作用是为任何一个在路由表中能够查到一条非多播网络路由或默认路由的目的地址创建一个新的主机路由。这个新克隆的主机路由是在它第一次查找时创建的。

6.8 in_matroute函数

rtallocl(卷2第483页)在查找一个路由时调用了rnh_matchaddr指针所指向的函数(即图6-6中所示的in_matroute函数)。

```
68 static struct radix_node *
69 in_matroute(void *v_arg, struct radix_node_head *head)
70 {
71      struct radix_node *rn = rn_match(v_arg, head);
72      struct rtentry *rt = (struct rtentry *) rn;

73      if (rt && rt->rt_refcnt == 0) {      /* this is first reference */
74          if (rt->rt_prflags & RTPRF_OURS) {
75              rt->rt_prflags &= ~RTPRF_OURS;
76              rt->rt_rmx.rmx_expire = 0;
77          }
78      }
79      return (rn);
80 }
```
in_rmx.c

图6-6 in_matroute 函数

调用rn_match来查找路由

71-78 rn_match在路由表中查找路由。如果找到了一个路由并且其参考计数值为 0，这就是该路由表记录项的第一个参考路由。如果记录项已经超时，也就是说，如果设置了RTPRF_OURS标志，这时就要把这个标志将关闭，并且将 rmx_expire定时器设置为0。当路由已经关闭，但在删除前又重新使用该路由时，就往往会发生这种情况。

6.9 in_clsroute函数

我们曾经提到过， T/TCP在radix_node_head结构中增加了一个新的函数指针rnh_close。当参考计数值为零时，这个函数就要在 rtfree中调用，这又将调用in_clsroute函数，如图6-7所示。

1. 检查标志

93-99 要做以下的测试：路由必须是正常的， RTF_HOST标志必须是打开的(即这不是一个

in_rmx.c

```
 89 static void
 90 in_clsroute(struct radix_node *rn, struct radix_node_head *head)
 91 {
 92     struct rtentry *rt = (struct rtentry *) rn;
 93     if (!(rt->rt_flags & RTF_UP))
 94         return;
 95     if ((rt->rt_flags & (RTF_LLINFO | RTF_HOST)) != RTF_HOST)
 96         return;
 97     if ((rt->rt_prflags & (RTPRF_WASCLONED | RTPRF_OURS))
 98         != RTPRF_WASCLONED)
 99         return;
100     /*
101      * If rtq_reallyold is 0, just delete the route without
102      * waiting for a timeout cycle to kill it.
103      */
104     if (rtq_reallyold != 0) {
105         rt->rt_prflags |= RTPRF_OURS;
106         rt->rt_rmx.rmx_expire = time.tv_sec + rtq_reallyold;
107     } else {
108         rtrequest(RTM_DELETE,
109                   (struct sockaddr *) rt_key(rt),
110                   rt->rt_gateway, rt_mask(rt),
111                   rt->rt_flags, 0);
112     }
113 }
```

in_rmx.c

图6-7 in_clsroute 函数

网络路由)，RTF_LLINFO标志必须是关闭的(对ARP记录项，该标志要打开)，RTPRF_WASCLONED必须是打开的(记录项是克隆的)，RTPRF_OURS必须是关闭的(该记录项还未超时)。如果这些测试中有任何一项失败，函数都将结束并返回。

2. 设置路由表记录项的终止时间

100-112 在通常的情况下，如果rtq_reallyold非零，就要打开RTPRF_OURS标志，并且要将rmx_expire时间值设置为当前时钟的秒值(time.tv_sec)加上rtq_reallyold值(一般为3 600秒，即1小时)。如果系统管理员用sysctl程序将rtq_reallyold的值设置为0，该路由就会立即被rtrequest删除。

6.10 in_rtqtimo函数

图6-4中，in_inithead首次调用in_rtqtimo函数。每一次调用执行in_rtqtimo时，它都会自动安排在rtq_timeout(默认值为600秒或者10分钟)后再次得到调用。

in_rtqtimo的目的是(用一般的rn_walktree函数)找遍整个IP路由表，对每一个记录项调用in_rtqkill。in_rtqkill要决定是否删除相应记录项。需要从 in_rtqtimo传递有关信息给 in_rtqkill(回顾图6-1)，或者反过来。传递是通过给 rn_walktree的第三个变量实现的。这个变量是 rn_walktree传递给in_rtqkill的一个指针。由于该变量是一个指针，所以信息可以在 in_rtqtimo和in_rtqkill之间的任何一个方向上传递。

in_rtqtimo传递给rn_walktree的指针指向rtqk_arg结构，这个结构如图6-8所示。

in_rmx.c

```
114 struct rtqk_arg {
115     struct radix_node_head *rnh;     /* head of routing table */
116     int     found;              /* #entries found that we're timing out */
117     int     killed;             /* #entries deleted by in_rtqkill */
118     int     updating;           /* set when deleting excess entries */
119     int     draining;           /* normally 0 */
120     time_t  nextstop;           /* time when to do it all again */
121 };
```

in_rmx.c

图6-8 rtqk_arg 结构：in_rtqtimo 与in_rtqkill 之间传递的信息

研究in_rtqtimo函数时，我们可以看到这些字段是怎样用的。如图 6-9所示。

in_rmx.c

```
159 static void
160 in_rtqtimo(void *rock)
161 {
162     struct radix_node_head *rnh = rock;
163     struct rtqk_arg arg;
164     struct timeval atv;
165     static time_t last_adjusted_timeout = 0;
166     int     s;

167     arg.rnh = rnh;
168     arg.found = arg.killed = arg.updating = arg.draining = 0;
169     arg.nextstop = time.tv_sec + rtq_timeout;
170     s = splnet();
171     rnh->rnh_walktree(rnh, in_rtqkill, &arg);
172     splx(s);

173     /*
174      * Attempt to be somewhat dynamic about this:
175      * If there are 'too many' routes sitting around taking up space,
176      * then crank down the timeout, and see if we can't make some more
177      * go away.  However, we make sure that we will never adjust more
178      * than once in rtq_timeout seconds, to keep from cranking down too
179      * hard.
180      */
181     if ((arg.found - arg.killed > rtq_toomany) &&
182         (time.tv_sec - last_adjusted_timeout >= rtq_timeout) &&
183         rtq_reallyold > rtq_minreallyold) {
184         rtq_reallyold = 2 * rtq_reallyold / 3;
185         if (rtq_reallyold < rtq_minreallyold)
186             rtq_reallyold = rtq_minreallyold;

187         last_adjusted_timeout = time.tv_sec;
188         log(LOG_DEBUG, "in_rtqtimo: adjusted rtq_reallyold to %d\n",
189             rtq_reallyold);
190         arg.found = arg.killed = 0;
191         arg.updating = 1;
192         s = splnet();
193         rnh->rnh_walktree(rnh, in_rtqkill, &arg);
194         splx(s);
195     }
196     atv.tv_usec = 0;
197     atv.tv_sec = arg.nextstop;
198     timeout(in_rtqtimo, rock, hzto(&atv));
199 }
```

in_rmx.c

图6-9 in_rtqtimo 函数

1. 设置rtqk_arg结构并调用rn_walktree

167-172 rtqk_arg结构的初始化包括：在IP路由表的首部设置rnh，计数器found和killed清零，draining和update标志清零，将nextstop设置为当前时间(秒级)加上rtq_timeout(600秒，即10分钟)。rn_walktree要找遍整个IP路由表，对每一个记录项调用in_rtqkill(图6-11)。

2. 检查路由表记录项是否过多

173-189 如果以下三个条件满足，就说明路由表中的记录项过多：

1) 已经超时但仍未删除的路由表记录项数(found减去killed)超过了rtq_toomany(默认值为128)。

2) 上一次执行本项操作至今所经过的秒数超过了rtq_timeout(600秒，即10分钟)

3) rtq_really超过了rtq_minreallyold(默认值为10)。

如果以上条件全部成立，则将rtq_reallyold设置为其当前值的2/3(用整数除法)。由于该值的初始值为3 600秒(60分钟)，因此它的取值就会分别是3600、2400、1600、1066和710，等等。但是该值不允许低于rtq_minreallyold(默认值为10秒)。当前时间值记录在静态变量last_adjusted_timeout中，并且要有一个调试消息发送给syslogd守护程序([Stevens 1992]的13.4.2节中给出了如何用log函数发送消息给syslogd守护程序)。这段代码以及减少rtq_reallyold值的目的是缩短路由表的处理周期，在路由表中记录项过多时删除过时的路由。

190-195 rtqk_arg结构中的计数器found和killed又初始化为0，updating标志此时设置为1，再次调用rn_walktree。

196-198 in_reqkill函数将rtqk_arg结构中的nextstop字段设置为下次调用in_rtqtimo的时间。内核的timeout函数会安排这个事件在需要的时候发生。

每10分钟就游历整个路由表一遍需要多大的开销？很明显这依赖于路由表中记录项的数目。在14.10节中我们模拟了一个繁忙的Web服务器中的T/TCP路由表大小，发现即使24小时内服务器要与5 000个不同的客户连接，并且主机路由的超时间隔为1小时，路由表中也从不会超过550个记录项。目前，一些Internet主干路由器中有成千上万条的路由表记录项，但是它们不是主机，而是路由器。我们并不会希望主干路由器支持T/TCP，因此也不必有规律地游历这样一个非常大的路由表以删除过时的路由。

6.11 in_rtqkill函数

rn_walktree要调用in_rtqkill函数，其中rn_walktree又是被in_rtqtimo调用的。我们在图6-11中所示的程序in_rtqkill就用于在必要时删除IP路由表记录项。

1. 只处理已经超时的记录项

134-135 这个函数只对设置了RTPRF_OURS标志的记录项进行处理，也就是说，只处理已经被in_clsroute关闭了的记录项(即它们的参考计数值已经达到零)和已经过了一个超时间隔(通常为1小时)而过期的记录项。这个函数不影响正在使用的路由 (因为这些路由的RTPRF_OURS标志不会打开的)。

136-146 如果设置了draining标志(在当前的实现中是永远不会设置的)，或者超时间隔已到(rmx_expire时间小于当前时间)，相应的路由就被rtrequest删除。rtqk_arg结构中

的found字段累计已经设置了RTPRF_OURS标志位的路由表记录项数，killed字段则用于累计被删除的记录项数。

147-151 else语句在当前记录项还没有超时时执行。如果设置了updating标志(图6-9中我们已经看到，当过期的路由太多时就会设置该标志，或者下一次对整个路由表进行处理时也会设置该标志)，并且还远未到过期时间(一定是在将来某一时刻，以便相减时产生一个正值)，这时就将过期时间重新设置为当前时间加上rtq_reallyold。考虑图6-10所示的例子就容易理解了。

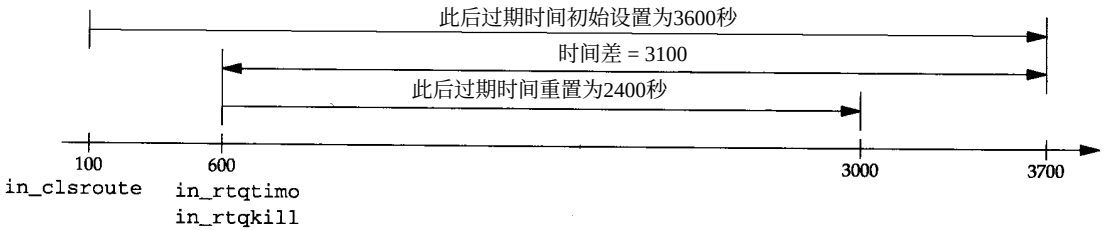

图6-10 in_rtqkill 重新设置过期时间

in_rmx.c

```
127 static int
128 in_rtqkill(struct radix_node *rn, void *rock)
129 {
130     struct rtqk_arg *ap = rock;
131     struct radix_node_head *rnh = ap->rnh;
132     struct rtentry *rt = (struct rtentry *) rn;
133     int     err;

134     if (rt->rt_prflags & RTPRF_OURS) {
135         ap->found++;

136         if (ap->draining || rt->rt_rmx.rmx_expire <= time.tv_sec) {
137             if (rt->rt_refcnt > 0)
138                 panic("rtqkill route really not free");

139             err = rtrequest(RTM_DELETE,
140                             (struct sockaddr *) rt_key(rt),
141                             rt->rt_gateway, rt_mask(rt),
142                             rt->rt_flags, 0);
143             if (err)
144                 log(LOG_WARNING, "in_rtqkill: error %d\n", err);
145             else
146                 ap->killed++;
147         } else {
148             if (ap->updating &&
149                 (rt->rt_rmx.rmx_expire - time.tv_sec > rtq_reallyold)) {
150                 rt->rt_rmx.rmx_expire = time.tv_sec + rtq_reallyold;
151             }
152             ap->nextstop = lmin(ap->nextstop, rt->rt_rmx.rmx_expire);
153         }
154     }
155     return (0);
156 }
```

in_rmx.c

图6-11 in_rtqkill 函数

图中x轴为时间，单位是秒。一个路由在时刻100时被in_clsroute关闭(当它的参考计

数值达到零时)，同时rtq_reallyold有了初始值3600(1小时)。这样，这个路由的过期时间就为3 700。但在时刻600，执行了in_rtqtimo，并且路由未删除 (因为它的过期时间是在3 100秒，还未到)，但由于路由记录项太多，使得 in_rtqtimo将rtq_reallyold的值重置为2 400、将updating设置为1，并且rn_walktree再次处理整个 IP路由表。此时in_rtqkill发现updating已经是1并且路由将在3100秒时过期。因为3100大于2400，过期时间就要重置为过2400秒以后，也就是在3000秒的时刻。路由表变大时，过期时间也就变短。

2. 计算下一个过期时间

152-153 每当发现一个记录项已经过期但其过期时间还未到时，就要执行这段代码。nextstop要设置为其当前值与路由表记录项过期时间中的最小值。前面讲过， nextstop的初始值是由in_rtqtimo设置的，设置值为当前时间加上 rtq_timeout的值(即10分钟以后)。

想想图6-12所示的例子。*x*轴代表时间，单位为秒，黑点的时刻为 0、600、……，等等，是调用in_rtqtimo函数的时刻。

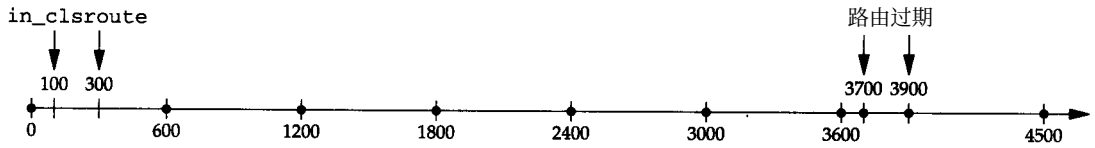

图6-12 根据路由过期时间执行 in_rtqtimo

in_addroute创建一个IP路由，然后在时刻100时被in_clsroute关闭。它的过期时间设置为3700(1小时以后)。在时刻300创建了第二个路由，然后被关闭，其过期时间设为 3900。in_rtqtimo函数每十分钟就执行一次，分别是在时刻 0、600、1200、1800、2400、3000和3600等。从时刻0至时刻3000，nextstop的值设置为当前时刻加上 600，这样在时刻3000为这两个路由分别调用in_rtqkill时，nextstop就改为了3600，因为3600小于3700和3900。但是在时刻3600为这两个路由分别又调用 in_rtqkill时，nextstop就设置为3700，因为3700小于3900，也小于4200。这就意味着在时刻3700将再次调用in_rtqtimo，而不是在时刻4200。此外，当in_rtqkill在时刻3700被调用时，因为另一个路由需要在时刻 3900过期，这就要将 nextstop设置为 3900。假设没有别的 IP路由要过期，在时刻 3900执行了in_rtqtimo以后，它将在4500、5100、……，等等时刻再次执行。

过期时间的交互影响

在路由表记录项的过期时间和 rt_metrics结构中的rmx_expire字段之间会有一些微小的交互影响。首先，地址解析协议 ARP也同样用该字段实现 ARP记录项的超时 (卷2的第21章)。这意味着路由表中有关本地子网中某一主机的路由表记录项 (以及与其相关的TAO信息)在该主机的ARP记录项被删除时也会同时被删除，通常是每 20分钟执行一次删除。这个间隔比in_rtqkill(1小时)所用的默认过期时间要短得多。回顾前面应该记得， in_clsroute明确地忽略已设置了 RTM_LLINFO标志的ARP记录项(图6-7)，让ARP对它们执行超时处理，而不用in_rtqkill。

其次，执行route程序去读取并打印一个克隆的 T/TCP路由表记录项的度量数据和过期

时间的副作用是会重置其过期时间。这种情况是这样的。假设有一个使用过的路由关闭了 (其参考计数值变为零)。关闭时，其过期时间设置为 1小时以后。59分钟过去了，但就在它即将过期前的1分钟，调用了route程序来打印这个路由的度量数据。以下是执行 route程序时需要调用的内核函数：route_output调用rtallocl，rtallocl又调用in_matroute(Internet专有的函数rnh_matchaddr)，in_matroute函数加大了参考计数值，即从 0到1。当这些操作全部完成后，假设参考值又从 1回到0，rtfree就调用in_clsroute，而in_clsroute将过期时间重置为1小时以后。

6.12 小结

在T/TCP中，我们为rt_metrics结构增加了16字节。其中的10个字节被T/TCP用作TAO缓存：

- tao_cc，从对等端收到的最后一个有效SYN中的CC值；
- tao_ccsent，发给对等端的最后一个CC值；
- tao_mssopt，从对等端收到的最后一个MSS值。

在radix_node_head结构中新增加了一个函数指针：rnh_close字段，当路由的参考计数值达到0时，就要调用该指针所指的函数(如果有定义)。

专门为Internet协议增加了四个新的函数：

1) in_inithead用于初始化Internet的radix_node_head结构，设置我们现在讲述的这四个函数指针。
2) in_addroute在IP路由表中增加新路由时调用。它为每一个非主机路由和非多播地址路由的IP路由打开克隆标志。
3) in_matroute在每次查找到IP路由时调用。如果路由被in_clsroute函数设置为超时，就要把它的过期时间重置为0，因为这个路由又有用了。
4) in_clsroute在IP路由的最后一个参考也被关闭时调用。它将路由的过期时间设置为1小时以后。我们也看到了，如果路由表过大时，过期时间要缩短。