第25章　TCP的定时器

25.1　引言

从本章起，我们开始详细讨论TCP的实现代码，首先熟悉一下在绝大多数TCP函数里都会遇到的各种定时器。

TCP为每条连接建立了七个定时器。按照它们在一条连接生存期内出现的次序，简要介绍如下。

1)“连接建立(connection establishment)”定时器在发送SYN报文段建立一条新连接时启动。如果没有在75秒内收到响应，连接建立将中止。

2)“重传(retransmission)”定时器在TCP发送数据时设定。如果定时器已超时而对端的确认还未到达，TCP将重传数据。重传定时器的值(即TCP等待对端确认的时间)是动态计算的，取决于TCP为该连接测量的往返时间和该报文段已被重传的次数。

3)“延迟ACK(delayed ACK)”定时器在TCP收到必须被确认但无需马上发出确认的数据时设定。TCP等待200 ms后发送确认响应。如果，在这200 ms内，有数据要在该连接上发送，延迟的ACK响应就可随着数据一起发送回对端，称为捎带确认。

4)“持续(persist)”定时器在连接对端通告接收窗口为0，阻止TCP继续发送数据时设定。由于连接对端发送的窗口通告不可靠(只有数据才会被确认，ACK不会被确认)，允许TCP继续发送数据的后续窗口更新有可能丢失。因此，如果TCP有数据要发送，但对端通告接收窗口为0，则持续定时器启动，超时后向对端发送1字节的数据，判定对端接收窗口是否已打开。与重传定时器类似，持续定时器的值也是动态计算的，取决于连接的往返时间，在5秒到60秒之间取值。

5)“保活(keepalive)”定时器在应用进程选取了插口的SO_KEEPALIVE选项时生效。如果连接的连续空闲时间超过2小时，保活定时器超时，向对端发送连接探测报文段，强迫对端响应。如果收到了期待的响应，TCP可确定对端主机工作正常，在该连接再次空闲超过2小时之前，TCP不会再进行保活测试。如果收到的是其他响应，TCP可确定对端主机已重启。如果连续若干次保活测试都未收到响应，TCP就假定对端主机已崩溃，尽管它无法区分是主机故障(例如，系统崩溃而尚未重启)，还是连接故障(例如，中间的路由器发生故障或电话线断了)。

6) FIN_WAIT_2定时器。当某个连接从FIN_WAIT_1状态变迁到FIN_WAIT_2状态(图24-15)，并且不能再接收任何新数据时(意味着应用进程调用了close，而非shutdown，没有利用TCP的半关闭功能)，FIN_WAIT_2定时器启动，设为10分钟。定时器超时后，重新设为75秒，第二次超时后连接被关闭。加入这个定时器的目的是为了避免如果对端一直不发送FIN，某个连接会永远滞留在FIN_WAIT_2状态。

7) TIME_WAIT定时器，一般也称为2MSL定时器。2MSL指两倍的MSL，24.8节定义的最大报文段生存时间。当连接转移到TIME_WAIT状态，即连接主动关闭时，定时器启动。卷1

的18.6节详细说明了需要2MSL等待状态的原因。连接进入TIME_WAIT状态时，定时器设定为1分钟(Net/3选用30秒的MSL)，超时后，TCP控制块和Internet PCB被删除，端口号可重新使用。

TCP包括两个定时器函数：一个函数每200 ms调用一次(快速定时器)；另一个函数每500 ms调用一次(慢速定时器)。延迟ACK定时器与其他6个定时器有所不同：如果某个连接上设定了延迟ACK定时器，那么下一次200 ms定时器超时后，延迟的ACK必须被发送(ACK的延迟时间必须在0~200 ms之间)。其他的定时器每500 ms递减一次，计数器减为0时，就触发相应的动作。

25.2 代码介绍

当某个连接的TCP控制块中的TF_DELACK标志(图24-14)置位时，允许该连接使用延迟ACK定时器。TCP控制块中的t_timer数组包括4个(TCPT_NTIMERS)计数器，用于实现其他的6个定时器。图25-1列出了数组的索引。下面简单地介绍这6个计数器是如何实现除延迟ACK定时器外的其余6个定时器的。

常 量	值	描 述
TCPT_REXMT	0	重传定时器
TCPT_PERSIST	1	持续定时器
TCPT_KEEP	2	保活定时器或连接建立定时器
TCPT_2MSL	3	2MSL定时器或FIN_WAIT_2定时器

图25-1 t_timer 数组索引

t_timer中的每条记录，保存了定时器的剩余值，以500 ms为计时单位。如果等于零，则说明对应的定时器没有设定。由于每个定时器都是短整型，所以定时器的最大值只能设定为16 383.5秒，约为4.5小时。

	建连定时器	重传定时器	延迟ACK定时器	持续定时器	保活定时器	FIN_WAIT_2	2MSL
t_timer[TCPT_REXMT]		•					
t_timer[TCPT_PERSIST]				•			
t_timer[TCPT_KEEP]	•				•		
t_timer[TCPT_2MSL]						•	•
t_flags & TF_DELACK			•				
tcp_keepidle (2小时)					•		
tcp_keepintvl (75秒)					•	•	
tcp_maxidle (10分钟)					•	•	
2 * TCPTV_MSL (60秒)							•
TCPTV_KEEP_INIT (75秒)	•						

图25-2 七个TCP定时器的实现

请注意，图25-1中利用4个“定时计数器”实现了6个TCP“定时器”，这是因为有些定时器彼此间是互斥的。下面我们首先区分一下计数器与定时器。TCPT_KEEP计数器同时实现了保活定时器和连接建立定时器，因为这两个定时器永远不会同时出现在同一条连接上。类似地，2MSL定时器和FIN_WAIT_2定时器都由TCPT_2MSL计数器实现，因为一条连接在同一

时间内只可能处于其中的一种状态。图 25-2的第一行小结了7个TCP定时器的实现方式，第二行和第三行列出了其中4个定时器初始化时用到的3个全局变量(图24-3)和2个常量(图25-3)。注意，有2个全局变量同时被多个定时器使用。前面已讨论过，延迟 ACK定时器直接受控于TCP的200 ms定时器，在本章后续部分将讨论其他 2个定时器的时间长度是如何设定的。

图25-3列出了Net/3实现中基本的定时器取值。

常　量	500ms的时钟滴答数	秒　数	描　述
TCPTV_MSL	60	30	MSL，最大报文段生存时间
TCPTV_MIN	2	1	重传定时器最小值
TCPTV_REXMTMAX	128	64	重传定时器最大值
TCPTV_PERSMIN	10	5	持续定时器最小值
TCPTV_PERSMAX	120	60	持续定时器最大值
TCPTV_KEEP_INIT	150	75	连接建立定时器取值
TCPTV_KEEP_IDLE	14400	7200	第一次保活测试前连接的空闲时间 (2小时)
TCPTV_KEEPINTVL	150	75	对端无响应时保活测试间的间隔时间
TCPTV_SRTTBASE	0		特殊取值，意味着目前无连接 RTT样本
TCPTV_SRTTDFLT	6	3	连接无RTT样本时的默认值

图25-3　TCP实现中基本的定时器取值

图25-4列出了在代码中会遇到的其他定时器常量。

常　量	值	描　述
TCP_LINGERTIME	120	用于SO_LINGER插口选项的最大时间，以秒为单位
TCP_MAXRXTSHIFT	12	等待某个ACK的最大重传次数
TCPTV_KEEPCNT	8	对端无响应时，最大保活测试次数

图25-4　定时器常量

图25-5中定义的TCPT_RANGESET宏，给定时器设定一个给定值，并确认该值在指定范围内。

tcp_timer.h

```
102 #define TCPT_RANGESET(tv, value, tvmin, tvmax) { \
103     (tv) = (value); \
104     if ((tv) < (tvmin)) \
105         (tv) = (tvmin); \
106     else if ((tv) > (tvmax)) \
107         (tv) = (tvmax); \
108 }
```

tcp_timer.h

图25-5　TCPT_RANGESET　宏

从图25-3可知，重传定时器和持续定时器都有最大值和最小值限制，因为它们的取值都是基于测量的往返时间动态计算得到的，其他定时器均设为常值。

本章中将不讨论图25-4中列出的一个特殊定时器：插口的拖延定时器 (linger timer)，这是由插口选项SO_LINGER设置的。这是一个插口级的定时器，由系统函数close使用(15.15节)。在图30-12中读者将看到，插口关闭时， TCP会首先检查该选项是否置位，拖延时间是否为 0。

如果上述条件满足，将不采用TCP正常的关闭过程，连接直接被复位。

25.3 tcp_canceltimers函数

图25-6中定义了tcp_canceltimers函数。连接进入TIME_WAIT状态时，tcp_input在设定2MSL定时器之前，调用该函数。4个定时计数器清零，相应地关闭了重传定时器、持续定时器、保活定时器和FIN_WAIT_2定时器。

tcp_timer.c

```
107 void
108 tcp_canceltimers(tp)
109 struct tcpcb *tp;
110 {
111     int     i;

112     for (i = 0; i < TCPT_NTIMERS; i++)
113         tp->t_timer[i] = 0;
114 }
```

tcp_timer.c

图25-6 tcp_canceltimers 函数

25.4 tcp_fasttimo函数

图25-7定义了tcp_fasttimo函数。该函数每隔200 ms被pr_fasttimo调用一次，用于操作延迟ACK定时器。

tcp_timer.c

```
41 void
42 tcp_fasttimo()
43 {
44     struct inpcb *inp;
45     struct tcpcb *tp;
46     int     s = splnet();

47     inp = tcb.inp_next;
48     if (inp)
49         for (; inp != &tcb; inp = inp->inp_next)
50             if ((tp = (struct tcpcb *) inp->inp_ppcb) &&
51                 (tp->t_flags & TF_DELACK)) {
52                 tp->t_flags &= ~TF_DELACK;
53                 tp->t_flags |= TF_ACKNOW;
54                 tcpstat.tcps_delack++;
55                 (void) tcp_output(tp);
56             }
57     splx(s);
58 }
```

tcp_timer.c

图25-7 tcp_fasttimo 函数，每200 ms调用一次

函数检查TCP链表中每个具有对应TCP控制块的Internet PCB。如果TCP_DELACK标志置位，清除该标志，并置位TF_ACKNOW标志。调用tcp_output，由于TF_ACKNOW标志已置位，ACK被发送。

为什么TCP的PCB链表中的某个Internet PCB会没有相应的TCP控制块(第50行的判断)？读者将在图30-11中看到，创建插口时(PRU_ATTACH请求响应socket系统调用)，首先创建

Inertnet PCB，之后才创建TCP控制块。两个操作间有可能会插入高优先级的时钟中断(图1-13)，该中断有可能调用tcp_fasttimo函数。

25.5 tcp_slowtimo函数

图25-8定义了tcp_slowtimo函数，每隔500ms被pr_slowtimo调用一次。它操作其他6个定时器：连接建立定时器、重传定时器、持续定时器、保活定时器、 FIN_WAIT_2定时器和2MSL定时器。

tcp_timer.c

```
64 void
65 tcp_slowtimo()
66 {
67     struct inpcb *ip, *ipnxt;
68     struct tcpcb *tp;
69     int     s = splnet();
70     int     i;

71     tcp_maxidle = TCPTV_KEEPCNT * tcp_keepintvl;
72     /*
73      * Search through tcb's and update active timers.
74      */
75     ip = tcb.inp_next;
76     if (ip == 0) {
77         splx(s);
78         return;
79     }
80     for (; ip != &tcb; ip = ipnxt) {
81         ipnxt = ip->inp_next;
82         tp = intotcpcb(ip);
83         if (tp == 0)
84             continue;
85         for (i = 0; i < TCPT_NTIMERS; i++) {
86             if (tp->t_timer[i] && --tp->t_timer[i] == 0) {
87                 (void) tcp_usrreq(tp->t_inpcb->inp_socket,
88                                   PRU_SLOWTIMO, (struct mbuf *) 0,
89                                   (struct mbuf *) i, (struct mbuf *) 0);
90                 if (ipnxt->inp_prev != ip)
91                     goto tpgone;
92             }
93         }
94         tp->t_idle++;
95         if (tp->t_rtt)
96             tp->t_rtt++;
97       tpgone:
98         ;
99     }
100    tcp_iss += TCP_ISSINCR / PR_SLOWHZ;     /* increment iss */
101    tcp_now++;                  /* for timestamps */
102    splx(s);
103 }
```

tcp_timer.c

图25-8 tcp_slowtimo 函数，每隔500 ms调用一次

71 tcp_maxidle初始化为10分钟，这是TCP向对端发送连接探测报文段后，收到对端主机响应前的最长等待时间。如图25-6所示，FIN_WAIT_2定时器也使用了这一变量。它的初始化语句可放到tcp_init中，因为其值可在系统初启时设定(见习题25.2)。

1. 检查所有TCP控制块中的所有定时器

72-89 检查TCP链表中每个具有对应TCP控制块的Internet PCB，测试每个连接的所有定时计数器，如果非0，计数器减1。如果减为0，则发送PRU_SLOWTIMO请求。后面会介绍该请求将调用tcp_timers函数。

tcp_usrreq的第四个入口参数是指向mbuf的指针。不过，在不需要mbuf指针的场合，这个参数实际被用于完成其他功能。tcp_slowtimo函数中利用它传递索引i，指出超时的是哪一个时钟。代码中把i强制转换为mbuf指针是为了避免编译错误。

2. 检查TCP控制块是否已被删除

90-93 在检查控制块中的定时器之前，先将指向下一个Internet PCB的指针保存在ipnxt中。每次PRU_SLOWTIMO请求返回后，tcp_slowtimo会检查TCP链表中的下一个PCB是否仍指向当前正处理的PCB。如果不是，则意味着控制块已被删除——也许2MSL定时器超时或重传定时器超时，并且TCP已放弃当前连接——控制转到tpgone，跳过当前控制块的其余定时器，并移至下一个PCB。

3. 计算空闲时间

94 当一个报文段到达当前连接，tcp_input清零控制块中的t_idle。从连接收到最后一个报文段起，每隔500ms t_idle递增一次。空闲时间统计主要有三个目的：(1)TCP在连接空闲2小时后发送连接探测报文段；(2)如果连接位于FIN_WAIT_2状态，且空闲10分钟后又空闲75秒，TCP将关闭该连接；(3)连接空闲一段时间后，tcp_output将返回慢启动状态。

4. 增加RTT计数器

95-96 如果需要测量某个报文段的RTT，tcp_output在发送该报文段时，初始化t_rtt计数器为1。它每500 ms递增一次，直至收到该报文段的确认。在tcp_slowtimo函数中，如果连接正对某个报文段计时，即t_rtt计数器非零，则递增t_rtt。

5. 递增初始发送序号

100 tcp_iss在tcp_init中初始化为1。每500 ms tcp_iss增加64 000: 128 000 (TCP_ISSINCR)除以2 (PR_SLOWHZ)。尽管看上去tcp_iss每秒钟仅递增两次，但实际速率可达每8微秒增加1。后面将介绍，无论主动打开或被动打开，只要建立了一条连接，tcp_iss就会增加64 000。

> RFC 793规定初始发送序号应该约每4微秒增加一次，或每秒钟250 000次。Net/3实现的增加速率只有规定的一半。

6. 递增RFC 1323规定的时间戳值

101 tcp_now在系统重启时初始化为0，每500 ms递增一次，用于实现RFC 1323中定义的时间戳[Jacobson, Barden和Borman 1992]。26.6节中将详细介绍这一功能。

75-79 请注意，如果主机上没有打开的连接(tcb.inp_next为空)，则tcp_iss和则tcp_now的递增将停止。这种状况只可能发生在系统初启时，因为在一个联网的UNIX系统中几乎不可能没有若干活跃的TCP服务器。

25.6 tcp_timers函数

tcp_timers函数在4个TCP定时计数器中的任何一个减为0时由TCP的PRU_SLOWTIMO请求处理代码调用(图30-10)：

```
case PRU_SLOWTIMO:
     tp = tcp_timers(tp, (int)nam);
```

整个函数的结构是一个switch语句，每个定时器对应一个case语句，如图25-9所示。

tcp_timer.c

```
120 struct tcpcb *
121 tcp_timers(tp, timer)
122 struct tcpcb *tp;
123 int     timer;
124 {
125     int     rexmt;
126     switch (timer) {

                /* switch cases */

256     }
257     return (tp);
258 }
```

tcp_timer.c

图25-9 tcp_timers 函数：总体框架

下面我们介绍其中3个定时计数器(5个TCP定时器)，重传定时器留待25.11节中再讨论。

25.6.1 FIN_WAIT_2和2MSL定时器

TCP的TCP2_2MSL定时计数器实现了两种定时器。

1) FIN_WAIT_2定时器。当tcp_input从FIN_WAIT_1状态变迁到FIN_WAIT_2状态，并且插口不再接收任何新数据(意味着应用进程调用了close，而不是shutdown，从而无法利用TCP的半关闭功能)时，FIN_WAIT_2定时器设定为10分钟(tcp_maxidle)。这样可以防止连接永远停留在FIN_WAIT_2状态。

2) 2MSL定时器。当TCP转移到TIME_WAIT状态，2MSL定时器设定为60秒。

图25-10列出了处理2MSL定时器的case语句——在该定时器减为0时执行。

tcp_timer.c

```
127         /*
128          * 2 MSL timeout in shutdown went off.  If we're closed but
129          * still waiting for peer to close and connection has been idle
130          * too long, or if 2MSL time is up from TIME_WAIT, delete connection
131          * control block.  Otherwise, check again in a bit.
132          */
133     case TCPT_2MSL:
134         if (tp->t_state != TCPS_TIME_WAIT &&
135             tp->t_idle <= tcp_maxidle)
136             tp->t_timer[TCPT_2MSL] = tcp_keepintvl;
137         else
138             tp = tcp_close(tp);
139         break;
```

tcp_timer.c

图25-10 tcp_timers 函数：2MSL定时器超时

1. 2MSL定时器

127-139 图25-10中的条件判断逻辑较为复杂，因为TCPT_2MSL计数器的两种不同用法混在了一起(习题25.4)。首先看TIME_WAIT状态，定时器60秒超时后，将调用tcp_close并释

放控制块。图25-11给出了典型的时间顺序，列出了2MSL定时器超时后的一系列函数调用。从图中可看出，如果某个定时器被设定为N秒($2\times N$滴答)，由于定时计数器的第一次递减将发生在其后的0~500 ms之间，定时器将在其后$2\times N-1$和$2\times N$个滴答之间的某个时刻超时。

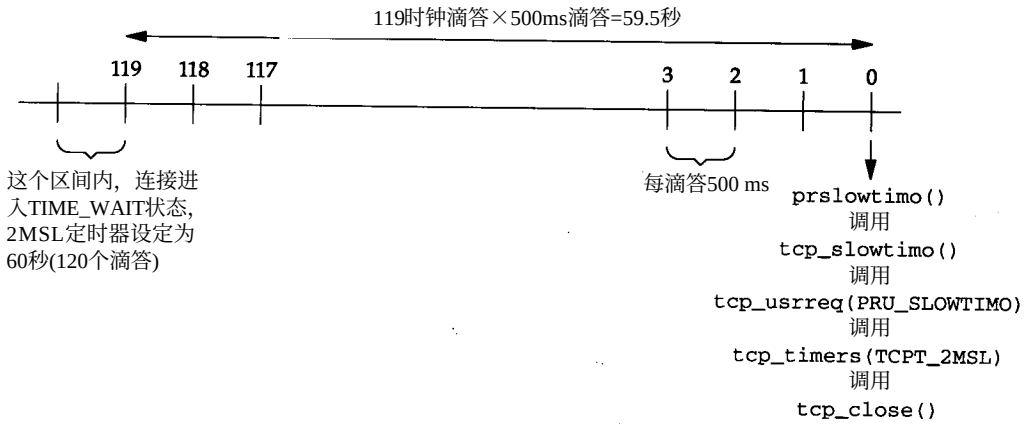

图25-11 TIME_WAIT状态下2MSL定时器的设定与超时

2. FIN_WAIT_2定时器

127-139 如果连接状态不是TIME_WAIT，TCPT_2MSL计数器表示FIN_WAIT_2定时器。只要连接的空闲时间超过10分钟(tcp_maxidle)，连接就会被关闭。但如果连接的空闲时间小于或等于10分钟，FIN_WAIT_2定时器将被设为75秒。图25-12给出了典型的时间顺序。

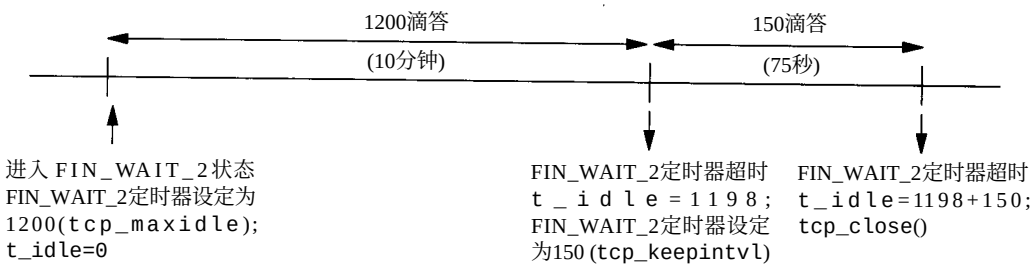

图25-12 FIN_WAIT_2定时器，避免永久滞留于FIN_WAIT_2状态

连接接收到一个ACK后，从FIN_WAIT_1状态变迁到FIN_WAIT_2状态(图24-15)，t_idle被置为0，FIN_WAIT_2定时器设为1200(tcp_maxidle)。图25-12中，向上的箭头指着10分钟定时起始时刻的右侧，强调定时计数器的第一次递减发生在定时器设定后的0~500 ms之间。1199个滴答后，定时器超时。从图25-8中可知，在四个定时计数器递减并做超时判定之后，t_idle才会增加，因此t_idle等于1198(我们假定连接在10分钟内一直空闲)。因为条件表达式“1198小于或等于1200”为真，FIN_WAIT_2定时器设为150(tcp_keepintvl)。定时器75秒后再次超时，假定连接一直空闲，t_idle应为1348，条件表达式为假，tcp_close被调用。

第一次10分钟定时后加入另一个75秒定时是因为除非持续空闲时间超过10分钟，否则处于FIN_WAIT_2状态的连接不会被关闭。如果第一个10分钟定时器还未超时，测试t_idle值是没有意义的，但只要过了这段时间，每隔75秒就会进行一次测试。由于有可能收到重复的报文段，即一个重复的ACK使得连接从FIN_WAIT_1状态变迁到FIN_WAIT_2状态，因此每收到一个报文段，10分钟等待将重新开始(因为t_idle重设为0)。

处于FIN_WAIT_2状态的连接在10分钟空闲后将被关闭，这一点并不符合协议规范，但在实际中是可行的。处于FIN_WAIT_2状态，应用进程调用close，连接上的所有数据都已发送并被确认，FIN已被对端确认，TCP等待对端应用进程调用close。如果对端进程永远不关闭它的连接，本地TCP将一直滞留在FIN_WAIT_2状态。应定义计数器保存由于这种原因而终止的连接数，从而了解这种状况出现的频率。

25.6.2 持续定时器

图25-13给出了处理持续定时器超时的case语句。

tcp_timer.c

```
210         /*
211          * Persistence timer into zero window.
212          * Force a byte to be output, if possible.
213          */
214     case TCPT_PERSIST:
215         tcpstat.tcps_persisttimeo++;
216         tcp_setpersist(tp);
217         tp->t_force = 1;
218         (void) tcp_output(tp);
219         tp->t_force = 0;
220         break;
```

tcp_timer.c

图25-13 tcp_timers 函数：持续定时器超时

强制发送窗口探测报文段

210-220 持续定时器超时后，由于对端已通告接收窗口为0，TCP无法向对端发送数据。此时，tcp_setpersist计算持续定时器的下一个设定值，并存储在TCPT_PERSIST计数器中。t_force标志置位，强制tcp_output发送1字节数据。

图25-14给出了局域网环境下，持续定时器的典型值，假定连接的重传时限为1.5秒(见卷1的图22-1)。

图25-14 持续定时器取值的时间表：探测对端接收窗口

一旦持续定时器取值达到60秒，TCP将每隔60秒发送一次窗口探测报文段。由于持续定时器取值的下限为5秒，上限为60秒，因此定时器头两次均设定为5秒，而不是1.5秒和3秒。从图中可知，定时器采用了指数退避策略，新的取值等于原有值乘以2，25.9节中将介绍这一算法的实现。

25.6.3 连接建立定时器和保活定时器

TCP的TCPT_KEEP计数器实现了两个定时器：

1) 当应用进程调用connect，连接转移到SYN_SENT状态(主动打开)，或者当连接从LISTEN状态变迁到SYN_RCVD状态(被动打开)时，SYN发送之后，将连接建立定时器设定为75秒(TCPTV_KEEP_INIT)。如果75秒内连接未能进入ESTABLISHED状态，则该连接被丢弃。

2) 收到一个报文段后，tcp_input将复位连接的保活定时器，重设为2小时(tcp_keepidle)，并清零连接的t_idle计数器。上述操作适用于系统中所有的TCP连接，无论是否置位了插口的保活选项。如果保活定时器超时(收到最后一个报文段2小时后)，并且置位了插口的保活选项，则TCP将向对端发送连接探测报文段。如果定时器超时，且未置位插口选项，则TCP将只复位定时器，重设为2小时。

图25-15给出了处理TCP的TCPT_KEEP计数器的case语句。

tcp_timer.c

```
221         /*
222          * Keep-alive timer went off; send something
223          * or drop connection if idle for too long.
224          */
225     case TCPT_KEEP:
226         tcpstat.tcps_keeptimeo++;
227         if (tp->t_state < TCPS_ESTABLISHED)
228             goto dropit;        /* connection establishment timer */
229         if (tp->t_inpcb->inp_socket->so_options & SO_KEEPALIVE &&
230             tp->t_state <= TCPS_CLOSE_WAIT) {
231             if (tp->t_idle >= tcp_keepidle + tcp_maxidle)
232                 goto dropit;
233             /*
234              * Send a packet designed to force a response
235              * if the peer is up and reachable:
236              * either an ACK if the connection is still alive,
237              * or an RST if the peer has closed the connection
238              * due to timeout or reboot.
239              * Using sequence number tp->snd_una-1
240              * causes the transmitted zero-length segment
241              * to lie outside the receive window;
242              * by the protocol spec, this requires the
243              * correspondent TCP to respond.
244              */
245             tcpstat.tcps_keepprobe++;
246             tcp_respond(tp, tp->t_template, (struct mbuf *) NULL,
247                         tp->rcv_nxt, tp->snd_una - 1, 0);
248             tp->t_timer[TCPT_KEEP] = tcp_keepintvl;
249         } else
250             tp->t_timer[TCPT_KEEP] = tcp_keepidle;
251         break;
252       dropit:
253         tcpstat.tcps_keepdrops++;
254         tp = tcp_drop(tp, ETIMEDOUT);
255         break;
```

tcp_timer.c

图25-15 tcp_timer 函数：保活时钟超时处理

1. 连接建立定时器75秒后超时

221-228 如果状态小于ESTABLISHED(图24-16)，TCPT_KEEP计数器代表连接建立定时器。定时器超时后，控制转到dropit，调用tcp_drop终止连接，给出差错代码ETIMEDOUT。我们将看到，ETIMEDOUT是默认差错码——例如，连接收到了某个差错报告，比如ICMP的主机不可达，返回应用进程的差错码将变为EHOSTUNREACH，而非默认差错码。

我们将在图30-4中看到，TCP发送SYN的同时初始化了两个定时器：正在讨论的连接建立定时器，设定为75秒，和重传定时器，保证对端无响应时可重传SYN。图25-16给出了这两个

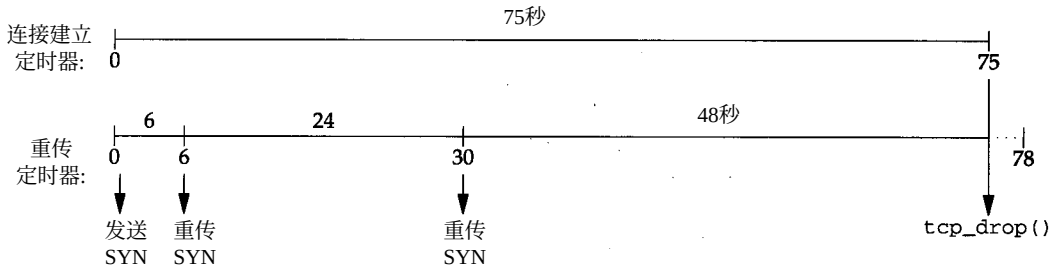

图25-16 SYN发送后：连接建立定时器和重传定时器

定时器。

对于一个新连接，重传定时器初始化为6 秒(图25-19)，后续值分别为24秒和48秒，25.7节中将详细讨论定时器取值的计算方法。重传定时器使得 SYN报文段在0秒、6秒和30秒处连续三次被重传。在75秒处，也就是重传定时器再次超时之前 3秒钟，连接建立定时器超时，调用tcp_drop终止连接。

2. 保活定时器在2小时空闲后超时

229-230 所有连接上的保活定时器在连续 2小时空闲后超时，无论连接是否选取了插口的SO_KEEPALIVE选项。如果插口选项置位，并且连接处于 ESTABLISHED状态或CLOSE_WAIT状态(图24-15)，TCP将发送连接探测报文段。但如果应用进程调用了 close(状态大于CLOSE_WAIT)，即使连接已空闲了2小时，TCP也不会发送连接探测报文段。

3. 无响应时丢弃连接

231-232 如果连接总的空闲时间大于或等于2小时(tcp_keepidle)加10分钟(tcp_maxidle)，连接将被丢弃。也就是说，对端无响应时， TCP最多发送9个连接探测报文段，间隔75秒(tcp_keepintvl)。TCP在确认连接已死亡之前必须发送多个连接探测报文段的一个原因是，对端的响应很可能是不带数据的纯 ACK报文段，TCP无法保证此类报文段的可靠传输，因此，连接探测报文段的响应有可能丢失。

4. 进行保活测试

233-248 如果TCP进行保活测试的次数还在许可范围之内， tcp_respond将发送连接探测报文段。报文段的确认字段(tcp_respond的第四个参数)填入rcv_nxt，期待接收的下一序号；序号字段填入snd_una -1，即对端已确认过的序号(图24-17)。由于这一特定序号落在接收窗口之外，对端必然会发送 ACK，给定它所期待的下一序号。

图25-17小结了保活定时器的用法

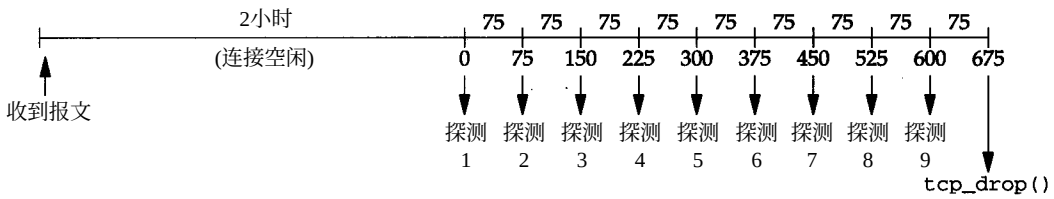

图25-17 保活定时器小结：判定对端是否可达

从0秒起，每隔75秒连续9次发送连接探测报文段，直至 600秒。675秒时(定时器2小时超时后的11.25分钟)连接被丢弃。请注意，尽管常量TCPTV_KEEPCNT(图25-4)的值设为8，却发

送了9次报文段，这是因为代码首先完成定时器递减、与0比较并做可能的处理后才递增变量t_idle(图25-8)。当tcp_input接收了一个报文段，就会复位保活定时器为14400(tcp_keepidle)，并清零t_idle。下一次调用tcp_slowtimo时，定时器减为14339而t_idle增为1。约2小时后，定时器从1减为0时将调用tcp_timers，而此时t_idle的值将为14339。图25-18列出了每次调用tcp_timers时t_idle的取值。

探测次数	图25-17中的时间	t_idle
1	0	14399
2	75	14549
3	150	14699
4	225	14849
5	300	14999
6	375	15149
7	450	15299
8	525	15449
9	600	15599
	675	15749

图25-18 调用tcp_timers 处理保活定时器时t_idle 的取值

图25-15中的代码一直等待t_idle大于或等于15600(tcp_keepidle+tcp_maxidle)，这一事件只可能发生在图25-17中的675秒处，即连续发送了9次连接探测报文段之后。

5. 复位保活定时器

249-250 如果插口选项未置位，或者连接状态大于CLOSE_WAIT，连接的保活定时器将复位，重设为2小时(tcp_keepidle)。

遗憾的是，计数器tcp_keepdrops(253行)不加区分地统计TCPT_KEEP定时计数器的两种不同用法所造成的连接丢弃：连接建立计数器和保活计数器。

25.7 重传定时器的计算

到目前为止，讨论过的定时器的取值都是固定的：延迟ACK 200ms，连接建立定时器75秒，保活定时器2小时等等。最后两个定时器——重传定时器和持续定时器——的取值依于连接上测算得到的RTT。在讨论实现定时器时限计算和设定的代码之前，首先应理解连接RTT的测算方法。

TCP的一个基本操作是在发送了需对端确认的报文段后，设置重传定时器。如果在定时器时限范围内未收到ACK，该报文段被重发。TCP要求对端确认所有数据报文段，不携带数据的报文段则无需确认(例如纯ACK报文段)。如果估算的重传时间过小，响应到达前即超时，造成不必要的重传；如果过大，在报文段丢失之后，发送重传报文段之前将等待一段额外的时间，降低了系统的效率。更为复杂的是，主机间的往返时间动态改变，且变化范围显著。

Net/3中TCP计算重传时限(RTO)时不仅要测量数据报文段的往返时间(*nticks*)，还要记录已平滑的RTT估计器(*srtt*)和已平滑的RTT平均偏差估计器(*rttvar*)。平均偏差是标准方差的良好近似，计算较为容易，无需标准方差的求平方根运算。[Jacobson 1988b]讨论了RTT测算的其他细节，给出下面的公式：

$$delta = nticks - srtt$$

$$srtt \leftarrow srtt + g \times delta$$

$$rttvar \leftarrow rttvar + h(|delta| - rttvar)$$

$$RTO = srtt + 4 \times rttvar$$

*delta*是最新测量的往返时间(*nticks*)与当前已平滑的RTT估计器(*srtt*)间的差值。g是用到RTT估计器的增益，设为1/8。h是用到平均偏差估计器的增益，设为1/4。这两个增益和*RTO*计算中的乘数4有意取为2的乘方，从而无需乘、除法，只需简单的移位操作就能够完成运算。

[Jacobson 1988b]规定RTO算式应使用2×*rttvar*，但经过进一步的研究，[Jacobson 1990d]更正为 4×*rttvar*，即Net/1实现中采用的算式。

下面首先介绍TCP重传定时器计算中用到的各种变量和算式，它们在 TCP代码中出现的频率很高。图25-19列出了控制块中与重传定时器有关的变量。

tcpcb的成员	单　位	tcp_newtcpcb初始值	秒　数	描　述
t_srtt	滴答×8	0		已平滑的RTT估计器：*srtt*×8
t_rttvar	滴答×4	24	3	已平滑的RTT平均偏差估计器：*rttvar*×4
t_rxtcur	滴答	12	6	当前重传时限：*RTO*
t_rttmin	滴答	2	1	重传时限最小值
t_rxtshift	不用	0		tcp_backoff[ ]数组索引(指数退避)

图25-19　用于重传定时器计算的控制块变量

tcp_backoff数组将在25.9节末尾定义。tcp_newtcpcb函数设定这些变量的初始值，实现代码将在下一节详细讨论。对变量t_rxtshift中的shift及其上限TCP_MAXRXTSHIFT的命名并不十分准确。它指的并不是比特移位，而是如图 25-19中所声明的，指数组索引。

TCP时限计算中不易理解的地方是已平滑的 RTT估计器和已平滑的RTT平均偏差估计器(t_rtt和t_rttvar)在C代码中都定义为整型，而不是浮点型。这样可以避免内核中的浮点运算，代价是增加了代码的复杂性。

为了区分缩放前和缩放后(scaled)的变量，斜体变量*srtt*和*rttvar*表示前面公式中未缩放的变量，t_srtt和t_rttvar表示TCP控制块中缩放后的变量。

图25-20列出了将遇到的四个常量，它们分别定义了 t_srtt的缩放因子和t_rttvar的缩放因子，分别为8和4。

常　量	值	描　述
TCP_RTT_SCALE	8	相乘：t_srtt=*srtt*×8
TCP_RTT_SHIFT	3	移位：t_srtt=*srtt*<<3
TCP_RTTVAR_SCALE	4	相乘：t_rttvar=*rttvar*×4
TCP_RTTVAR_SHIFT	2	移位：t_rttvar=*rttvar*<<2

图25-20　RTT均值与偏差的乘法与移位

25.8　tcp_newtcpcb算法

图25-21定义了tcp_newtcpcb，分配一个新的TCP控制块并完成初始化。创建新的插口时，TCP的PRU_ATTACH请求将调用它(图30-2)。调用者已事先为该连接分配了一个 Internet PCB，并在入口参数inp中包含指向该结构的指针。我们在这里给出函数代码，是因为它初始化了TCP的定时器变量。

167-175　内核函数malloc分配控制块所需内存，bzero清零新分配的内存块。

176　变量seg_next和seg_prev指向未按正常次序到达当前连接的报文段的重组队列。我们将在27.9节中详细讨论这一重组队列。

tcp_subr.c

```
167 struct tcpcb *
168 tcp_newtcpcb(inp)
169 struct inpcb *inp;
170 {
171     struct tcpcb *tp;

172     tp = malloc(sizeof(*tp), M_PCB, M_NOWAIT);
173     if (tp == NULL)
174         return ((struct tcpcb *) 0);
175     bzero((char *) tp, sizeof(struct tcpcb));
176     tp->seg_next = tp->seg_prev = (struct tcpiphdr *) tp;
177     tp->t_maxseg = tcp_mssdflt;
178     tp->t_flags = tcp_do_rfc1323 ? (TF_REQ_SCALE | TF_REQ_TSTMP) : 0;
179     tp->t_inpcb = inp;
180     /*
181      * Init srtt to TCPTV_SRTTBASE (0), so we can tell that we have no
182      * rtt estimate.  Set rttvar so that srtt + 2 * rttvar gives
183      * reasonable initial retransmit time.
184      */
185     tp->t_srtt = TCPTV_SRTTBASE;
186     tp->t_rttvar = tcp_rttdflt * PR_SLOWHZ << 2;
187     tp->t_rttmin = TCPTV_MIN;
188     TCPT_RANGESET(tp->t_rxtcur,
189                   ((TCPTV_SRTTBASE >> 2) + (TCPTV_SRTTDFLT << 2)) >> 1,
190                   TCPTV_MIN, TCPTV_REXMTMAX);

191     tp->snd_cwnd = TCP_MAXWIN << TCP_MAX_WINSHIFT;
192     tp->snd_ssthresh = TCP_MAXWIN << TCP_MAX_WINSHIFT;

193     inp->inp_ip.ip_ttl = ip_defttl;
194     inp->inp_ppcb = (caddr_t) tp;
195     return (tp);
196 }
```

tcp_subr.c

图25-21 tcp_newtcpcb 函数：创建并初始化一个新的TCP控制块

177-179 发送报文段的最大长度，t_maxseq，默认为512(tcp_mssdflt)。收到对端MSS选项后，它将被tcp_mss函数更改(新连接建立后，TCP也会向对端发送MSS选项)。如果配置要求系统实现 RFC 1313规定的可变窗口和时间戳功能 (图24-3中的全局变量tcp_do_rfc1313，默认值为1)，TF_REQ_SCALE和TF_REQ_TSTMP两个标志将被置位。TCP控制块中的t_inpcb指针将指向由调用者传来的Internet PCB。

180-185 初始化图 25-19中列出的四个变量 t_srtt、t_rttvar、t_rttmin和t_rxtcur。首先，已平滑的RTT估计器被设为0(TCPTV_SRTTBASE)，这个取值非常特殊，指明连接上还不存在RTT估计器。首次进行RTT测量时，tcp_xmit_timer函数将判定已平滑的RTT估计器是否等于0，以采取相应动作。

186-187 已平滑的RTT平均偏差估计器t_rttvar定义为24：3(tcp_rttdflt，图24-3)乘以2(PR_SLOWHZ)后左移2 bit(即乘以4)。由于t_rttvar是变量*rttvar*的4倍，也就等于6个滴答，即3秒钟。*RTO*的最小值，t_rttmin，为2个滴答。

188-190 变量t_rxtcu保存了当前 *RTO*值，以滴答为单位，最小值为 2个滴答(TCPTV_MIN)，最大值为128个滴答(TCPTV_REXMTMAX)。TCPT_RANGESET的第二个参数，表达式计算后等于12个滴答，即6秒钟，是连接的第一个*RTO*值。

理解上述C表达式和RTT缩放值的概念并不是一件容易的事，下面的讨论可能会对您有所

帮助。首先从原始的计算公式开始，并将缩放后的变量替代其中缩放前的变量。下面的算式用于计算第一个*RTO*，以乘数2替代了乘数4。

$$RTO=srtt+2\times rttvar$$

使用乘数2而非4是最初4.3BSD Tahoe实现的一个遗留问题[Paxson 1994]。

把下面两个缩放后的变量代入上式：

$$t\_srtt=8\times srtt$$

$$t\_rttvar=4\times rttvar$$

得到：

$$RTO=\frac{t\_srtt}{8}+2\times\frac{t\_rttvar}{4}=\frac{\frac{t\_srtt}{4}+t\_rttvar}{2}$$

也就是图25-21代码中TCPT_RANGESET第二个参数的表达式，只不过用常量——值为6个滴答的TCPTV_SRTTDFLT乘以4后(缩放运算)代替了变量t_rttvar。

191-192 拥塞窗口(snd_cwnd)和慢起动门限(snd_ssthresh)初始化为1 073 725 440 (约为1 G字节)，如是配置了动态窗口选项，这已是TCP窗口大小的上限(卷1的21.6节详细讨论了慢起动和避免拥塞策略)，即TCP首部窗口字段的最大值(65535，TCP_MAXWIN)乘以2^{14}，14是窗口缩放因子的最大值(TCP_MAX_WINSHIFT)。后面将看到，连接上发送或接收了一个SYN时，tcp_mss复位snd_cwnd为1。

193-194 Internet PCB中的IP TTL的默认值初始化为64(ip_defttl)，而PCB则指向新的TCP控制块。

代码中没有明确初始化的其他变量，如移位变量t_rxtshift，均为0，这是因为控制块内存分配后已由bzero清零。

25.9 tcp_setpersist函数

接下来要讨论的函数是tcp_setpersist，它用到了TCP的重传超时算法。从图25-13中可知，持续定时器超时后，将调用此函数。当TCP有数据要发送，而连接对端通告接收窗口为0时，持续定时器启动。图25-22给出了函数实现代码，计算并存储定时器的下个取值。

tcp_output.c

```
493 void
494 tcp_setpersist(tp)
495 struct tcpcb *tp;
496 {
497     t = ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1;

498     if (tp->t_timer[TCPT_REXMT])
499         panic("tcp_output REXMT");
500     /*
501      * Start/restart persistance timer.
502      */
503     TCPT_RANGESET(tp->t_timer[TCPT_PERSIST],
504                   t * tcp_backoff[tp->t_rxtshift],
505                   TCPTV_PERSMIN, TCPTV_PERSMAX);
506     if (tp->t_rxtshift < TCP_MAXRXTSHIFT)
507         tp->t_rxtshift++;
508 }
```

tcp_output.c

图25-22 tcp_setpersist 函数：计算并存储持续定时器的下一次取值

1. 确认重传定时器未设定

493-499 持续定时器设定之前，首先检查确认重传定时器未启动，这是因为两个定时器彼此互斥：如果数据已被发送，说明对端通告的接收窗口必然非零，但持续时钟仅当对端通告零接收窗口时才会设定。

2. 计算RTO

500-505 函数起始处，计算RTO值并存储到变量t中。使用的计算公式为

$$RTO = srtt + 2 \times rttvar$$

与上小节结束时讨论过的公式相同。通过变量替换可得到

$$RTO = \frac{\frac{t\_srtt}{4} + t\_rttvar}{2}$$

即变量t的计算式。

3. 指数退避算法

506-507 *RTO*计算中还用到了指数退避算法，将上式计算得到的*RTO*与tcp_backoff数组中的某个值相乘：

```
int tcp_backoff[ TCP_MAXRXTSHIFT + 1] =
          {1, 2, 4, 8, 16, 32, 64, 64, 64, 64, 64, 64, 64 };
```

tcp_output第一次为连接设置持续定时器的代码是：

```
        tp->t_rxtshift=0;
        tcp_setpersist(tp);
```

因此，第一次调用tcp_setpersist时，t_rxtshift= 0。由于tcp_backoff[0]=1，持续时限等于t。TCPT_RANGESET宏确保*RTO*值位于5秒~60秒之间。t_rxtshift每次增加1，直到最大值12(TCP_MAXRXTSHIFT)，tcp_backoff[12]是数组的最后一个元素。

25.10 tcp_xmit_timer函数

下一个讨论的函数，tcp_xmit_timer，在得到了一个RTT测量值，从而更新已平滑的RTT估计器(*srtt*)和平均偏差(*rttvar*)时被调用。

参数rtt传递了得到的RTT测量值。它的值为*nticks*+1 (与25.7节中的符号一致)，可以通过下面两种方法之一得到。

如果收到的报文段中存在时间戳选项， RTT测量值应等于当前时间(tcp_now)减去时间戳值。我们将在26.6节中讨论时间戳选项，现在只需了解tcp_now每500ms递增一次(图25-8)。发送报文段时，tcp_now做为时间戳被发送，连接对端在相应的ACK中回显该时间戳。

如果未使用时间戳，可以对数据报文计时。从图 25-8可知，连接上的计数器t_rtt每500 ms递增一次。在25.5节也曾提到，该计数器初始化为1，因此收到ACK时，该计数器中的值即为RTT测量值加1(以滴答为单位)。

tcp_input中调用tcp_xmit_timer的典型代码如下：

```
      if (ts_present)
           tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
      else if (tp->rtt  &&  SEQ_GT(ti->ti_ack, tp->t_rtseq))
           tcp_xmit_timer(tp, tp->t_rtt);
```

如果报文段中存在时间戳(ts_present)，RTT测量值等于当前时间(tcp_now)减去回显

的时间戳(ts_ecr)再加1，RTT估计器将被更新(后面将介绍加1的原因)。

如果不存在时间戳，但收到的ACK报文确认了一个正在计时的数据报文，这种情况下RTT估计器也将被更新。每个TCP控制块(t_rtt)中只存在一个RTT计数器，因此，在一条连接上只可能对一个特定数据报文计时。这个报文发送时的起始序号存储在t_rtseq中，与收到的ACK比较，可以确定该报文对应ACK返回的时间。如果收到的确认序号(ti_ack)大于正在计时的数据报文起始序号(t_rtseq)，t_rtt即为RTT新的样本，从而更新RTT估计器。

在支持RFC 1323的时间戳功能之前，t_rtt是TCP测量RTT的唯一方法。但这个变量还用作确认报文段是否被计时的标志（图25-8)：如果t_rtt大于0，则tcp_slowtimo每隔500ms完成t_rtt的加1操作；因此，t_rtt非零时，它等于所用的滴答数再加1。我们将看到，tcp_xmit_timer函数中对得到的第二个参数减1，以纠正上述偏差。因此，使用时间戳时，向tcp_xmit_timer传送的第二个参数必须加1，以保持一致。

序号的大于判定是因为ACK是累积的：如果TCP发送并计时的报文序号为1~1024(t_rtseq等于1)，然后立即发送(但未计时)下一个报文序号为1025~2048，接着收到一个ACK报文，其ti_ack等于2049，它确认了序号1~2048，即同时确认了第一个计时报文和第二个未计时报文。注意，如果使用了RFC 1323定义的时间戳，则不存在序号比较问题。如果对端发送了时间戳选项，意味着它填入了回应时间(ts_ecr)，从而可直接计算RTT。

图25-23给出了函数更新RTT估算值的部分代码。

tcp_input.c

```
1310 void
1311 tcp_xmit_timer(tp, rtt)
1312 struct tcpcb *tp;
1313 short   rtt;
1314 {
1315     short   delta;

1316     tcpstat.tcps_rttupdated++;
1317     if (tp->t_srtt != 0) {
1318         /*
1319          * srtt is stored as fixed point with 3 bits after the
1320          * binary point (i.e., scaled by 8).  The following magic
1321          * is equivalent to the smoothing algorithm in rfc793 with
1322          * an alpha of .875 (srtt = rtt/8 + srtt*7/8 in fixed
1323          * point).  Adjust rtt to origin 0.
1324          */
1325         delta = rtt - 1 - (tp->t_srtt >> TCP_RTT_SHIFT);
1326         if ((tp->t_srtt += delta) <= 0)
1327             tp->t_srtt = 1;
1328         /*
1329          * We accumulate a smoothed rtt variance (actually, a
1330          * smoothed mean difference), then set the retransmit
1331          * timer to smoothed rtt + 4 times the smoothed variance.
1332          * rttvar is stored as fixed point with 2 bits after the
1333          * binary point (scaled by 4).  The following is
1334          * equivalent to rfc793 smoothing with an alpha of .75
1335          * (rttvar = rttvar*3/4 + |delta| / 4).  This replaces
1336          * rfc793's wired-in beta.
1337          */
```

图25-23 tcp_xmit_timer 函数：利用新的RTT测量值计算已平滑的RTT估计器

```
1338            if (delta < 0)
1339                delta = -delta;
1340            delta -= (tp->t_rttvar >> TCP_RTTVAR_SHIFT);
1341            if ((tp->t_rttvar += delta) <= 0)
1342                tp->t_rttvar = 1;
1343        } else {
1344            /*
1345             * No rtt measurement yet - use the unsmoothed rtt.
1346             * Set the variance to half the rtt (so our first
1347             * retransmit happens at 3*rtt).
1348             */
1349            tp->t_srtt = rtt << TCP_RTT_SHIFT;
1350            tp->t_rttvar = rtt << (TCP_RTTVAR_SHIFT - 1);
1351        }
```

tcp_input.c

图25-23 (续)

1. 更新已平滑的RTT估计器

1310-1325 前面已介绍过，tcp_newtcpcb初始化已平滑的RTT估计器(t_srtt)为0，指明连接上不存在RTT估计器。delta是RTT测量值与当前已平滑的RTT估计器间的差值，以未缩放的滴答为单位。t_srtt除以8，单位从缩放后的滴答转换为未缩放的滴答。

1326-1327 已平滑的RTT估计器用以下公式进行更新：

$$srtt \leftarrow srtt + g \times delta$$

由于增益g=1/8，公式变为

$$8 \times srtt \leftarrow 8 \times srtt + delta$$

也就是

$$t\_srtt \leftarrow t\_srtt + delta$$

1328-1342 已平滑的RTT平均偏差估计器的计算公式如下：

$$rttvar \leftarrow rttvar + h(|delta| - rttvar)$$

将h=1/4和缩放后的t_rttvar=4×$rttvar$代入，得到：

$$\frac{t\_rttvar}{4} \leftarrow \frac{t\_rttvar}{4} + \frac{|delta| - \frac{t\_rttvar}{4}}{4}$$

也就是：

$$t\_rttvar \leftarrow t\_rttvar + |delta| - \frac{t\_rttvar}{4}$$

最后一个表达式即为C代码中的表达式。

2. 第一次测量RTT时初始化平滑的估计器值

1343-1350 如果是首次测量某连接的RTT值，已平滑的RTT估计器初始化为测量得到的样本值。下面的计算用到了参数rtt，前面已介绍过rtt等于测量到的RTT值加1($nticks$+1)，而前面公式中用到的delta是从rtt中减1得到的。

$$srtt = nticks + 1$$

或

$$\frac{t\_srtt}{8} = nticks + 1$$

也就是

$$t\_srtt=(nticks+1)\times 8$$

平均偏差等于测量到的RTT值的一半：

$$rttvar = \frac{srtt}{2}$$

也就是

$$\frac{t\_rttvar}{4} = \frac{nticks+1}{2}$$

或者

$$t\_rttvar=(nticks+1)\times 2$$

代码中的注释指出，已平滑的平均偏差的这种初始取值使得 RTO的初始值等于$3\times srtt$。因为

$$RTO=srtt+4\times rttvar$$

替换掉rttvar，得到：

$$RTO = srtt + 4 \times \frac{srtt}{2}$$

也就是：

$$RTO=3\times srtt$$

图25-24给出了tcp_xmit_timer函数最后一部分的代码。

tcp_input.c

```
1352     tp->t_rtt = 0;
1353     tp->t_rxtshift = 0;

1354     /*
1355      * the retransmit should happen at rtt + 4 * rttvar.
1356      * Because of the way we do the smoothing, srtt and rttvar
1357      * will each average +1/2 tick of bias.  When we compute
1358      * the retransmit timer, we want 1/2 tick of rounding and
1359      * 1 extra tick because of +-1/2 tick uncertainty in the
1360      * firing of the timer.  The bias will give us exactly the
1361      * 1.5 tick we need.  But, because the bias is
1362      * statistical, we have to test that we don't drop below
1363      * the minimum feasible timer (which is 2 ticks).
1364      */
1365     TCPT_RANGESET(tp->t_rxtcur, TCP_REXMTVAL(tp),
1366                   tp->t_rttmin, TCPTV_REXMTMAX);

1367     /*
1368      * We received an ack for a packet that wasn't retransmitted;
1369      * it is probably safe to discard any error indications we've
1370      * received recently.  This isn't quite right, but close enough
1371      * for now (a route might have failed after we sent a segment,
1372      * and the return path might not be symmetrical).
1373      */
1374     tp->t_softerror = 0;
1375 }
```

tcp_input.c

图25-24 tcp_xmit_timer 函数：最后一部分

1352-1353 RTT计数器(t_rtt)和重传移位计数器(t_rxtshift)同时复位为0，为下一个报文的发送和计时做准备。

1354-1366 连接的下一个*RTO*(t_rxtcur)计算用到宏

```
#define TCP_REXMTVAL(tp) \
        (((tp)->t_srtt >> TCP_RTT_SHIFT) + (tp)->t_ttvar)
```

其实，这就是我们很熟悉的公式

$$RTO = srtt + 4 \times rttvar$$

用tcp_xmit_timer更新过的缩放后的变量替代上式中的 *srtt*和*rttvar*，得到宏的表达式：

$$RTO = \frac{t\_srtt}{8} + 4 \times \frac{t\_rttvar}{4} = \frac{t\_srtt}{8} + t\_rttvar$$

此外， *RTO*取值应在规定范围之内，最小值为连接上设定的最小 RTO(t_rttmin，t_newtcpcb初始化为2个滴答)，最大值为128个滴答(TCPTV_REXMTMAX)。

3. 清除软错误变量

1367-1374 由于只有当收到了已发送的数据报文的确认时，才会调用tcp_xmit_timer，如果连接上发生了软错误(t_softerror)，该错误将被丢弃。下一节中将详细讨论软错误。

25.11 重传超时：tcp_timers函数

我们现在回到tcp_timers函数，讨论25.6节中未涉及的最后一个case语句：处理重传定时器。如果在*RTO*内没有收到对端对一个已发送数据报的确认，则执行此段代码。

图25-25小结了重传定时器的操作。假定 tcp_output计算的报文首次重传时限为 1.5秒，这是LAN的典型值(参见卷1的图21-1)。

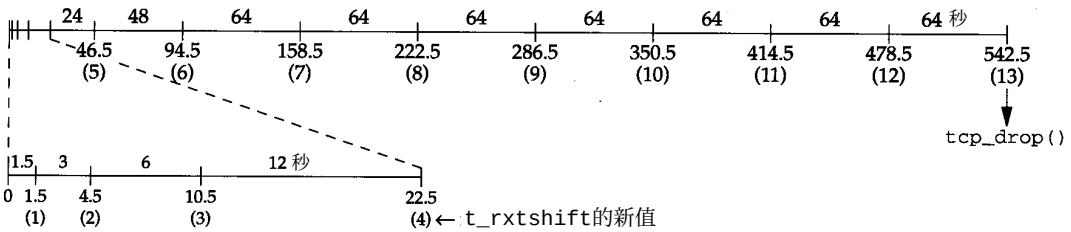

图25-25 发送数据时重传定时器小结

x轴为时间轴，以秒为单位，标注依次为： 0、1.5、4.5等等。这些数字的下方，给出了代码中用到的t_rxtshift的值。连续12次重传后，总共为542.5秒(约9分钟)，TCP将放弃并丢弃连接。

RFC 793建议在建立新连接时，无论主动打开或被动打开，应定义一个参数规定TCP发送数据的总时限，也就是TCP在放弃发送并丢弃连接之前试图传输给定数据报文的总时间。推荐的默认值为5分钟。

RFC 1122要求应用程序必须为连接指定一个参数，限定 TCP总的重传次数或者TCP试图发送数据的总时间。这个参数如果设为“ 无限 ”，那么TCP永不会放弃，还可能不允许终端用户终止连接。

在代码中可看到， Net/3不支持应用程序的上述控制权： TCP放弃传输之前的重传次数是固定的(12)，所用的总时间取决于RTT。

图25-26给出了重传超时case语句的前半部分。

tcp_timer.c

```
140         /*
141          * Retransmission timer went off.  Message has not
142          * been acked within retransmit interval.  Back off
143          * to a longer retransmit interval and retransmit one segment.
144          */
145     case TCPT_REXMT:
146         if (++tp->t_rxtshift > TCP_MAXRXTSHIFT) {
147             tp->t_rxtshift = TCP_MAXRXTSHIFT;
148             tcpstat.tcps_timeoutdrop++;
149             tp = tcp_drop(tp, tp->t_softerror ?
150                           tp->t_softerror : ETIMEDOUT);
151             break;
152         }
153         tcpstat.tcps_rexmttimeo++;
154         rexmt = TCP_REXMTVAL(tp) * tcp_backoff[tp->t_rxtshift];
155         TCPT_RANGESET(tp->t_rxtcur, rexmt,
156                       tp->t_rttmin, TCPTV_REXMTMAX);
157         tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
158         /*
159          * If losing, let the lower level know and try for
160          * a better route.  Also, if we backed off this far,
161          * our srtt estimate is probably bogus.  Clobber it
162          * so we'll take the next rtt measurement as our srtt;
163          * move the current srtt into rttvar to keep the current
164          * retransmit times until then.
165          */
166         if (tp->t_rxtshift > TCP_MAXRXTSHIFT / 4) {
167             in_losing(tp->t_inpcb);
168             tp->t_rttvar += (tp->t_srtt >> TCP_RTT_SHIFT);
169             tp->t_srtt = 0;
170         }
171         tp->snd_nxt = tp->snd_una;
172         /*
173          * If timing a segment in this window, stop the timer.
174          */
175         tp->t_rtt = 0;
```

tcp_timer.c

图25-26 tcp_timers 函数：重传定时器超时，前半部分

1. 递增移位计数器

146 重传移位计数器(t_rxtshift)在每次重传时递增，如果大于12(TCP_MAXRXTSHIFT)，连接将被丢弃。图25-25给出了t_rxtshift每次重传时的取值。请注意两种丢弃连接的区别，由于收不到对端对已发送数据报文的确认而造成的丢弃连接，和由于保活定时器的作用，在长时间空闲且收不到对端响应时丢弃连接。两种情况下，TCP都会向应用进程报告ETIMEDOUT差错，除非连接收到了一个软错误。

2. 丢弃连接

147-152 软错误指不会导致TCP终止已建立的连接或正试图建立的连接的错误，但系统会记录出现的软错误，以备TCP将来放弃连接时参考。例如，如果TCP重传SYN报文段，试图建立新的连接，但未收到响应，TCP将向应用进程报告ETIMEDOUT差错。但如果在重传期间，收到一个ICMP“主机不可达”差错代码，tcp_notify会在t_softerror中存储这一软错误。如果TCP最终决定放弃重传，返回给应用进程的差错代码将为EHOSTUNREACH，而不是

ETIMEDOUT，从而向应用进程提供了更多的信息。如果TCP发送SYN后，对端的响应为RST，这是个硬错误，连接立即被终止，返回差错代码ECONNRFUSED(图28-18)。

3. 计算新的*RTO*

153-157 利用TCP_REXMTVAL宏实现指数退避，计算新的*RTO*值。代码中，给定报文第一次重传时t_rxtshift等于1，因此，*RTO*值为TCP_REXMTVAL计算值的两倍。新的*RTO*值存储在t_rxtcur中，供连接的重传定时器——t_timer[TCPT_REXMT]——使用，tcp_input在启动重传定时器时会用到它(图28-12和图29-6)。

4. 向IP询问更换路由

158-167 如果报文段已重传4次以上，in_losing将释放缓存中的路由(如果存在)，tcp_output再次重传该报文时(图25-27中case语句的结尾处)，将选择一条新的，也许好一些的路由。从图25-25可看到，每次重传定时器超时时，如果重传时限已超过22.5秒，将调用in_losing。

5. 清除RTT估计器

168-170 代码中，已平滑的RTT估计器(t_srtt)被置为0(t_newtcpcb中曾将其初始化为0)，强迫tcp_xmit_timer将下一个RTT测量值做为已平滑的RTT估计器，这是因为报文段重传次数已超过4次，意味着TCP的已平滑的RTT估计器可能已失效。若重传定时器再次超时，进入case语句后，将利用TCP_REXMTVAL计算新的*RTO*值。由于t_srtt被置为0，新的计算值应与本次重传中的计算值相同，再利用指数退避算法加以修正(图25-28中，在42.464秒处的重传很好地说明了上面讨论的概念)。

再次计算*RTO*时，利用公式

$$RTO = \frac{\texttt{t\_srtt}}{8} + \texttt{t\_rttvar}$$

由于t_srtt等于0，*RTO*取值不变。如果报文的重传定时器再次超时(图25-28中从84.064秒到217.84秒)，case语句再次被执行，t_srtt等于0，t_rttvar不变。

6. 强迫重传最早的未确认数据

171 下一个发送序号(snd_nxt)被置为最早的未确认的序号(snd_una)。回想图24-17中，snd_nxt大于snd_una。把snd_nxt回移，将重传最早的未确认过的报文。

7. Karn算法

172-175 RTT计数器，t_rtt，被置为0。Karn算法认为由于该报文即将重传，对该报文的计时也就失去了意义。即使收到了ACK，也无法区分它是对第一次报文，还是对第二次报文的确认。[Karn and Partridge 1987]和卷1的21.3节中都介绍了这一算法。因此，TCP只对未重传报文计时，利用t_rtt计数器得到样本值，并据此修正RTT估计器。在后面的图29-6中将看到，如何使用RFC 1323的时间戳功能取代Karn算法。

25.11.1 慢起动和避免拥塞

图25-27给出了case语句的后半部分，实现慢起动和避免拥塞，并重传最早的未确认过的报文。

由于重传定时器超时，网络中很可能发生了拥塞。这种情况下，需要用到TCP的拥塞避免算法。如果最终收到了对端发送的确认，TCP采用慢起动算法以较慢的速率继续进行数据

传输。卷1的20.6节和21.6节详细讨论了这两种算法。

176-205 win被置为现有窗口大小(接收方通告的窗口大小snd_wnd和发送方拥塞窗口大小snd_cwnd，两者之中的较小值)的一半，以报文为单位，而非字节(因此除以t_maxseg)，最小值为2。它的值等于网络拥塞时现有窗口大小的一半，也就是慢起动门限，t_ssthresh(以字节为单位，因此乘以t_maxseq)。拥塞窗口的大小，snd_cwnd，被置为只容纳1个报文，强迫执行慢起动。上述做法假定造成网络拥塞的原因之一是本地数据发送太快，因此在拥塞发生时，必须降低发送窗口大小。

这段代码放在一对括号中，是因为它是在4.3BSD和Net/1实现之间添加的，并要求有自己的局部变量(win)。

206 连续重复ACK计数器，t_dupacks(用于29.4节中将介绍的快速重传算法)被置为0。我们将在第29章中介绍它在TCP快速重传和快速恢复算法中的用途。

208 tcp_output重新发送包含最早的未确认序号的报文，即由于重传定时器超时引发了报文重传。

tcp_timer.c

```
176         /*
177          * Close the congestion window down to one segment
178          * (we'll open it by one segment for each ack we get).
179          * Since we probably have a window's worth of unacked
180          * data accumulated, this "slow start" keeps us from
181          * dumping all that data as back-to-back packets (which
182          * might overwhelm an intermediate gateway).
183          *
184          * There are two phases to the opening: Initially we
185          * open by one mss on each ack.  This makes the window
186          * size increase exponentially with time.  If the
187          * window is larger than the path can handle, this
188          * exponential growth results in dropped packet(s)
189          * almost immediately.  To get more time between
190          * drops but still "push" the network to take advantage
191          * of improving conditions, we switch from exponential
192          * to linear window opening at some threshhold size.
193          * For a threshhold, we use half the current window
194          * size, truncated to a multiple of the mss.
195          *
196          * (the minimum cwnd that will give us exponential
197          * growth is 2 mss.  We don't allow the threshhold
198          * to go below this.)
199          */
200         {
201             u_int   win = min(tp->snd_wnd, tp->snd_cwnd) / 2 / tp->t_maxseg;
202             if (win < 2)
203                 win = 2;
204             tp->snd_cwnd = tp->t_maxseg;
205             tp->snd_ssthresh = win * tp->t_maxseg;
206             tp->t_dupacks = 0;
207         }
208         (void) tcp_output(tp);
209         break;
```

tcp_timer.c

图25-27 tcp_timer 函数：重传定时器超时，后半部分

25.11.2 精确性

TCP维护的这些估计器的精确性如何呢？首先应指出，因为RTT以500 ms为测量单位，是非常不精确的。已平滑的RTT估计器和平均偏差的精确性要高一些(缩放因子为8和4)，但也不够，LAN的RTT是毫秒级，横跨大陆的RTT约为60ms左右。这些估计器仅仅给出了RTT的上限，从而在设定重传定时器时，可以不考虑由于重传时限过小而造成不必要的重传。

[Brakmo, O'Malley, and Peterson 1994]描述的TCP实现，能够提供高精度的RTT样本。他们的做法是，发送报文段时记录系统时钟读数(精度比以500 ms为测量单位要高得多)，收到ACK时再次读取系统时钟，从而得到高精度的RTT。

Net/3支持的时间戳功能(26.6节)本来可以提供较高精度的RTT，但Net/3将时间戳的精度也定为500 ms。

25.12 一个RTT的例子

下面讨论一个具体的例子，说明上述计算是如何进行的。我们从主机`bsdi`向`vangogh.cs.berkeley.edu`发送12288字节的数据。在发送过程中，故意断开工作中的PPP链路，之后再恢复，看看TCP如何处理报文的超时与重传。为发送数据，我们运行自己的`sock`程序(参见卷1的附录C)，加-D选项，置位插口的`SO_DEBUG`选项(27.10节)。传输结束后，运行`trpt` (8)程序检查留在内核的环形缓存中的调试记录，之后打印TCP控制块中我们感兴趣的时钟变量。

图25-28列出了各变量在不同时刻的值。我们用M：N表示序号M~N-1已被发送。本例中的每个报文段都携带了512字节的数据。符号“ACK M”表示ACK报文的确认字段为M。标注“实际差值(ms)”栏列出了RTT定时器打开时刻和关闭时刻间的时间差值。标注“`rtt` (参数)”栏列出了调用`tcp_xmit_timer`时第二个参数的值：RTT定时器打开时刻和关闭时刻间的滴答数再加1。

`tcp_newtcpcb`函数完成`t_srtt`、`t_rttvar`和`t_rxtcur`的初始化，时刻0.0对应的即为变量初始值。

第一个计时报文是最初的SYN报文，365 ms后收到了对端的ACK，调用`tcp_xmit_timer`，`rtt`参数值为2。由于这是第一个RTT测量值(`t_srtt=0`)，执行图25-23中的`else`语句，计算RTT估计器初始值。

携带1~512字节的数据报文是第二个计时报文，1.259秒时收到对应的ACK，RTT估计器被更新。

从接下来的三个报文可看出，连续报文是如何被确认的。1.260秒时发送携带513~1024字节的报文，并启动定时器。之后又发送了携带1025~1526字节的报文，在2.206秒时收到了对端的ACK，同时确认了已发送的两个报文。RTT估计器被更新，因为ACK确认了正计时报文的起始序号(513)。

2.206秒时发送携带1537~2048字节的报文，并启动定时器。3.132秒时收到对应的ACK，RTT估计器被更新。

对3.132秒时发送的报文段计时，重传定时器设为5个滴答(`t_rxtcur`的当前值)。这时，路由器`sun`和`netb`间的PPP链路中断，几分钟后恢复正常。重传定时器在6.064秒超时，执行图25-26中的代码更新RTT变量。`t_rxtshift`从0增至1，`t_rxtcur`置为10个滴答(指数退

避)，重传最早的未确认过的序号 (snd_una=3073)。5秒钟后，定时器再次超时，t_rxtshift递增为2，重传定时器设为20个滴答。

发送时间	发送	接收	RTT定时器	实际时间差(ms)	rtt参数	t_srtt(8个滴答)	t_rttvar(4个滴答)	t_rxtcur(滴答)	t_rxtshift
0.0	SYN		on			0	24	12	
0.365		SYN,ACK	off	365	2	16	4	6	
0.365	ACK								
0.415	1:513		on						
1.259		ack 513	off	844	2	15	4	5	
1.260	513:1025		on						
1.261	1025:1537								
2.206		ack 1537	off	946	3	16	4	6	
2.206	1537:2049		on						
2.207	2049:2561								
2.209	2561:3073								
3.132		ack 2049	off	926	3	16	3	5	
3.132	3073:3585		on						
3.133	3585:4097								
3.736		ack 2561							
3.736	4097:4609								
3.737	4609:5121								
3.739		ack 3073							
3.739	5121:5633								
3.740	5633:6145								
6.064	3073:3585		off			16	3	10	1
11.264	3073:3585		off			16	3	20	2
21.664	3073:3585		off			16	3	40	3
42.464	3073:3585		off			0	5	80	4
84.064	3073:3585		off			0	5	128	5
150.624	3073:3585		off			0	5	128	6
217.184	3073:3585		off			0	5	128	7
217.944		ack 6145							
217.944	6145:6657		on						
217.945	6657:7169								
218.834		ack 6657	off	890	3	24	6	9	
218.834	7169:7681		on						
218.836	7681:8193								
219.209		ack 7169							
219.209	8193:8705								
219.760		ack 7681	off	926	2	22	7	9	
219.760	8705:9217		on						
220.103		ack 8705							
220.103	9217:9729								
220.105	9729:10241								
220.106	10241:10753								
220.821		ack 9217	off	1061	3	22	6	8	
220.821	10753:11265		on						
221.310		ack 9729							
221.310	11265:11777								
221.312		ack 10241							
221.312	11777:12289								
221.674		ack 10753							
221.955		ack 11265	off	1134	3	22	5	7	

图25-28 实例中的RTT变量值和估计器

42.464秒时，重传定时器再次超时，t_srtt清零，t_rttvar置为5。我们在图25-26的讨论中提到过，此时t_rxtcur运算得到的结果相同(因此，下一次运算的结果应为160)。但

由于t_srtt重置为0，下一次更新RTT估计器时(218.834秒)，与建立一条新的连接相类似，得到的RTT测量值将成为新的已平滑的RTT估计器。

之后继续进行数据传输，并且又多次更新了RTT估计器。

25.13 小结

内核每隔200 ms和500 ms，分别调用tcp_fasttimo函数和tcp_slowtimo函数。这两个函数负责维护TCP为连接建立的各种定时器。

TCP为每条连接维护下列7个定时器：

- 连接建立定时器；
- 重传定时器；
- 延迟ACK定时器；
- 持续定时器；
- FIN_WAIT_2定时器；
- 2MSL定时器；

延迟ACK定时器与其他6个定时器不同，设置它时意味着下一次TCP200 ms定时器超时时，延迟的ACK报文必须被发送。其他6个定时器都是计数器，每次TCP 500 ms定时器超时时，计数器减1。任何一个计数器减为0时，触发TCP完成相应动作：丢弃连接、重传报文、发送连接探测报文等等，这些内容本章中都有详细讨论。由于某些定时器是彼此互斥的，代码用4个计数器实现了这6个定时器，复杂性有所增加。

本章还介绍了重传定时器取值的标准计算方法。TCP为每条连接维护两个RTT估计器：已平滑的RTT估计器(*srtt*)和已平滑的RTT平均偏差估计器(*rttvar*)。尽管算法简单清楚，但由于使用了缩放因子(在不使用内核浮点运算的情况下保证足够的精度)，使得代码较为复杂。

习题

25.1 TCP快速超时处理函数的效率如何？(提示：参考图24-5中列出的延迟ACK的次数)有没有另外的实现方式？

25.2 为什么在tcp_slowtimo函数，而不是在tcp_init函数中初始化tcp_maxidle？

25.3 tcp_slowtimo递增t_idle，前面已介绍过t_idle用于计数从连接上收到最后一个报文起到当前为止的滴答数。TCP是否需要计数从连接上发送最后一个报文段起计时的空闲时间？

25.4 重写图25-10中的代码，分离TCPT_2MSL计数器两种不同用法的处理逻辑。

25.5 图25-12中，连接进入FIN_WIN_2状态75秒后收到一个重复的ACK。会发生什么？

25.6 应用程序设置SO_KEEPALIVE选项时连接已空闲了1小时。第一次连接探测报文在何时发送，1小时后还是2小时后？

25.7 为什么tcp_rttdflt是一个全局变量，而非常量？

25.8 重写与习题25.6有关的代码，实现另一种结果。