

## 第10章 T/TCP实现：TCP函数

### 10.1 概述

本章包括了 T/TCP作过修改的各个 TCP函数。也就是说，`tcp_output`(前一章)、`tcp_input`，和`tcp_usrreq`(后两章)以外的所有函数。本章定义了两个新的函数，`tcp_rtllookup`和`tcp_gettaocache`，用于在TAO缓存中查找记录项。

`tcp_close`函数修改以后，当使用 T/TCP的连接关闭时，可以在路由表中记录往返时间估计值(平滑的平均值和平均偏差估计)。常规协议只在连接上传送了至少 16个满数据报文段后才记录。然而，T/TCP通常只发送少量数据，但与同一对等端之间的这些不同连接的估计值应该保留下来。

T/TCP中对MSS选项的处理也有所改变。有一部分改变是为了在 Net/3中清理过载的`tcp_mss`函数，这样就把它分成了一个计算MSS以便发送的函数(`tcp_msssend`)和另一个处理接收到的MSS选项的函数(`tcp_mssrcvd`)。T/TCP同时也将从对等端收到的最新MSS值保存到TAO缓存记录项中。在接收到服务器的SYN和最新的MSS之前，如果要随SYN发送数据，T/TCP就用这个记录来初始化发送MSS。

Net/3中的`tcp_dooptions`函数修改以后能够识别三个新的T/TCP选项：CC、CCnew和CCecho。

### 10.2 `tcp_newtcpcb`函数

用PRU\_ATTACH请求创建新的插口时要调用该函数。图 10-1中的五行代码用来代替卷 2第 667页的第177~178行。

```

180      tp->t_maxseg = tp->t_maxopd = tcp_mssdflt;
181      if (tcp_do_rfc1323)
182          tp->t_flags = (TF_REQ_SCALE | TF_REQ_TSTMP);
183      if (tcp_do_rfc1644)
184          tp->t_flags |= TF_REQ_CC;

```

tcp\_subr.c

图10-1 `tcp_newtcpcb` 函数：T/TCP所做的修改

180 在前面图8-3有关的介绍中提到过，`t_maxopd`是每个报文段中可以发送的TCP选项加上数据的最大字节数。它和`t_maxseg`的默认值均为512(`tcp_mssdflt`)。由于这两个值相等，表明报文段中不能再有TCP选项。在后面的图10-13和图10-14中，如果时间戳选项或者CC选项(或者两者同时)需要在报文段中发送，就要减小`t_maxseg`的值。

183-184 如果全局变量`tcp_do_rfc1644`非零(它的默认值为1)，且设置了`TF_REQ_CC`标志，这将使`tcp_output`伴随SYN发出CC或CCnew选项(图9-6)。

## 10.3 tcp\_rtlookup函数

tcp\_mss(卷2第717~718页)执行的第一项操作是读取为该连接所缓存的路由 (存储在

```

46 struct route {
47     struct rentry *ro_rt;      /* pointer to struct with information */
48     struct sockaddr ro_dst;    /* destination of this route */
49 };

```

route.h

图10-2 route 结构

```

432 struct rentry *
433 tcp_rtlookup(inp)
434 struct inpcb *inp;
435 {
436     struct route *ro;
437     struct rentry *rt;
438
439     ro = &inp->inp_route;
440     rt = ro->ro_rt;
441     if (rt == NULL) {
442         /* No route yet, so try to acquire one */
443         if (inp->inp_faddr.s_addr != INADDR_ANY) {
444             ro->ro_dst.sa_family = AF_INET;
445             ro->ro_dst.sa_len = sizeof(ro->ro_dst);
446             ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
447                 inp->inp_faddr;
448             rtalloc(ro);
449             rt = ro->ro_rt;
450         }
451     }
452     return (rt);

```

tcp\_subr.c

图10-3 tcp\_rtlookup 函数

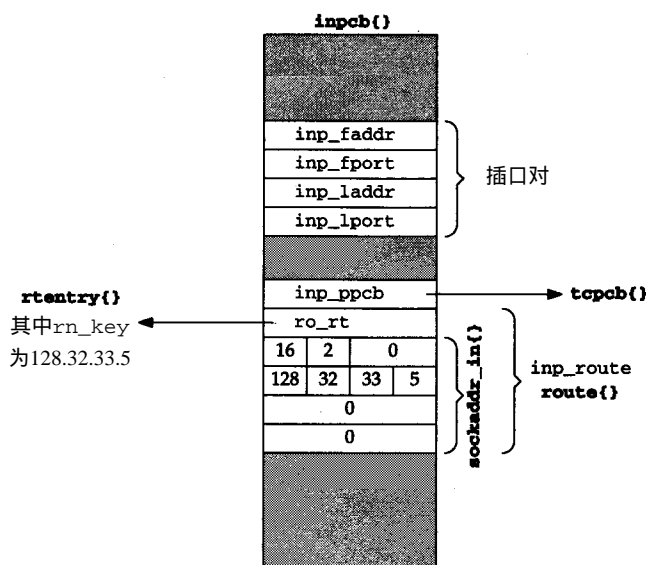


图10-4 在Internet PCB中缓存的路由全貌

Internet PCB的`inp_route`字段中), 如果该路由还没有缓存过, 则调用 `rtalloc` 查找路由。现在这项操作安排在另一个独立的函数 `tcp_rtlookup` 中实现, 我们将在图 10-3 中介绍。这样做是因为连接的路由表记录项中包括有 TAO 信息, T/TCP 需要更经常地执行这一项操作。

438-452 如果这个连接的路由还没有在缓存中记录, `rtalloc` 就计算出路由。但仅仅当 PCB 中的外部地址非 0 时才能计算路由。在调用 `rtalloc` 之前, 要先填写 `route` 结构中的 `sockaddr_in` 结构。

图 10-2 给出了 `route` 结构, 其中的一个结构在每个 Internet PCB 中都有。

图 10-4 给出了这个结构的全貌, 图中假定外部地址为 128.32.33.5。

## 10.4 tcp\_gettaocache 函数

一个给定主机的 TAO 信息保存在该主机的路由表记录项中, 确切地说, 是在 `rt_metrics` 结构的 `rmx_filler` 字段中(见 6.5 节)。图 10-5 所示的函数 `tcp_gettaocache` 返回指向该主机 TAO 缓存的指针。

```

458 struct rmxp_tao *                                     tcp_subr.c
459 tcp_gettaocache(inp)
460 struct inpcb *inp;
461 {
462     struct rtentry *rt = tcp_rtlookup(inp);
463     /* Make sure this is a host route and is up. */
464     if (rt == NULL ||
465         (rt->rt_flags & (RTF_UP | RTF_HOST)) != (RTF_UP | RTF_HOST))
466         return (NULL);
467     return (rmx_tao(r->rt_rmx));
468 }
tcp_subr.c

```

图 10-5 tcp\_gettaocache 函数

460-468 `tcp_rtlookup` 返回的指针指向外部主机的 `rtentry` 结构。如果查找成功, 并且 `RTF_UP` 和 `RTF_HOST` 标志均打开了, 则宏 `rmx_tao` (见图 6-3) 返回的指针指向 `rmxp_tao` 结构。

## 10.5 重传超时间隔的计算

Net/3 的 TCP 要测量数据报文段往返时间、跟踪平滑的 RTT 估计器 (`srtt`) 和平滑的平均偏差估计器 (`rttvar`), 并据此计算重传超时间隔 (RTO)。平均偏差是标准差的良好逼近, 比较容易计算, 因为与标准差不一样, 平均偏差不需要做平方根运算。文献 [Jacobson 1988] 给出了 RTT 测量的其他细节, 并导出以下的计算公式:

$$\begin{aligned}
 \text{delta} &= \text{data} - \text{srtt} \\
 \text{srtt} &= \text{srtt} + g \times \text{delta} \\
 \text{rttvar} &= \text{rttvar} + h(|\text{delta}| - \text{rttvar}) \\
 \text{RTO} &= \text{srtt} + 4 \times \text{rttvar}
 \end{aligned}$$

其中, `delta` 是刚刚得到的往返时间测量值 (`data`) 与当前的平滑的 RTT 估计器 (`srtt`) 之差; `g` 是应用于 RTT 估计器的增益, 等于 1/8; `h` 是应用于平均偏差估计器的增益, 等于 1/4。在 RTO 计算

中的两个增益和乘数4特意取为2的乘幂,因此可以通过移位操作来代替乘除运算。卷2的第25章给出了如何用定点整数来保存这些值的有关细节。

在常规的TCP连接中,在计算 $srtt$ 和 $rttvar$ 这两个估计器时,通常要对多个RTT取样,对于图1-9中的给定最小TCP连接来说,至少要有两个样本。而且,在一定条件下,Net/3将对相同主机之间的多个连接运用这两个估计器。这是`tcp_close`函数实现的,在一个连接关闭时,如果有关对等端的路由表记录项不是默认路由,并且至少得到了16个RTT样值。估计的结果存储在路由表记录项中`rt_metrics`结构的`rmx_rtt`和`rmx_rttvar`字段中。新连接建立时,`tcp_mssrcvd`(见10.8节)从路由表记录项中取出这两个值作为 $srtt$ 和 $rttvar$ 这两个估计器的初始值。

T/TCP中出现的的问题是,一个最小连接只有一个RTT测量值,而且少于16个样值是很正常的,因此在两个对等端之间相继建立拆除的T/TCP连接对上述测量和估计一点贡献也没有。这就意味着在T/TCP中,第一个报文段发出去时根本就不知道RTO的取值应该是多少。卷2的25.8节讨论了`tcp_newtcpcb`执行初始化时是怎样确定第一个RTO应该是6秒的。

让`tcp_close`在即使只收集到少于16个样值也存储对T/TCP连接的平滑估计结果并不难(在10.6节中我们会看到为此所做的修改),但问题是:如何将新估计值与以前的估计值进行归并?不幸的是,这仍然还是一个正在研究的问题[Paxson 1995a]。

为了理解各种不同的可能性,请考虑图10-6中的情况。从作者的一台主机上通过Internet向另一台主机上的回显服务器发送100个400字节长的UDP数据报(在一个工作日的下午,通常是Internet上最为拥挤的时候)。93个数据报有回显返回(还有7个不知在Internet的哪些地方丢失了),在图10-6中给出了前91个数据报。样值是在30分钟的时间内采集到的,前后数据报之间的时间间隔是在0~30秒之间均匀分布的随机数。实际的RTT是在客户主机上运行Tcpdump得到的。黑点就是测量得到的RTT。另外的三条实线(从上至下依次是RTO、 $srtt$ 和 $rttvar$ )是运用本

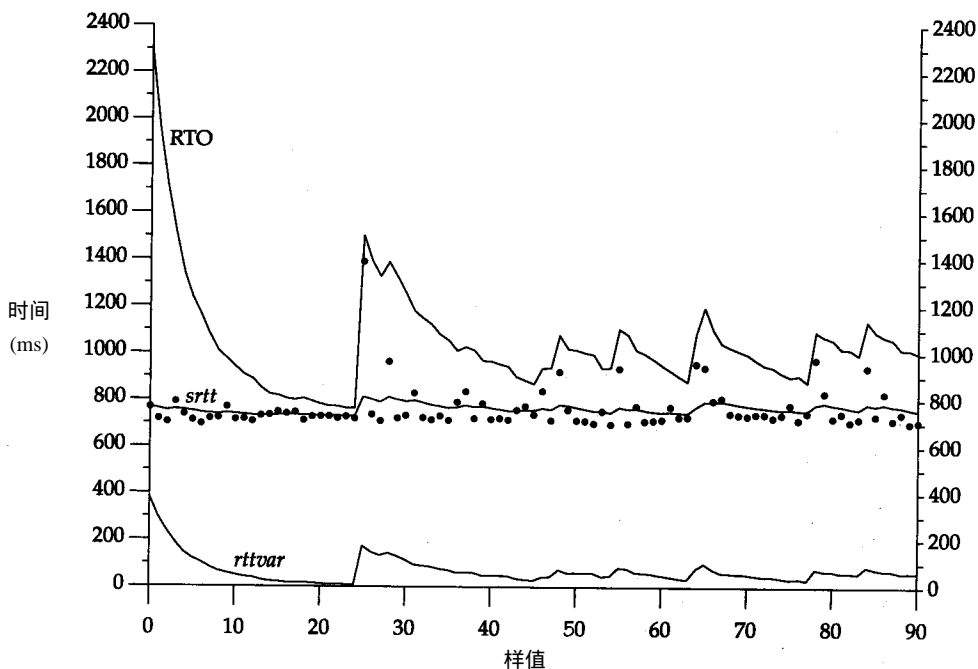


图10-6 RTT测量和对应的RTO、 $srtt$ 和 $rttvar$

节开头的公式从测得的 RTT 计算出来的。计算是用浮点算术完成的，而不是 Net/3 中实际所用的定点整数方法。图上所示的 RTO 就是从相应的数据点计算出来的值。也就是说，第一个数据点(大约 2200 ms)的 RTO 是从第一个数据点计算得来的，将用作下一个报文段发送时的 RTO。

尽管所测得的 RTT 值平均都在 800 ms 以下(作者的客户系统是通过拨号线上的 PPP 连接到 Internet 上的，穿越整个国家才能到达服务器)，第 26 个样值的 RTT 几乎达到 1400 ms，此后有少量的一些点在 1000 ms 左右。[Jacobson 1994] 指出，“只要有竞争的连接共享一条路由，瞬间 RTT 波动达到 2 倍最小值是完全正常的(它们仅仅表示另外一个连接的开始或丢失后重新开始)，因此，RTO 小于  $2 \times \text{RTT}$  从来就不会是合理的”。

当估计器有新值存储到路由表记录项中时，必须做出判断，对应于已经过去的历史，有多少信息是新的。这样，计算公式就为：

$$\text{savesrtt} = g \times \text{savesrtt} + (1 - g) \times \text{srtt}$$

$$\text{saverttvar} = g \times \text{saverttvar} + (1 - g) \times \text{rttvar}$$

这是一个低通滤波器，其中  $g$  是取值在 0~1 之间的过滤增益常量， $\text{savesrtt}$  和  $\text{saverttvar}$  是存储在路由表记录项中的数值。当 Net/3 用这些公式更新路由表记录时(当一个连接关闭，并假定得到了 16 个样值)，它采用的增益是 0.5：存储在路由表中的值有一半是路由表中的旧值，另有一半是当前估计的值。Bob Braden 的 T/TCP 代码中取增益为 0.75。

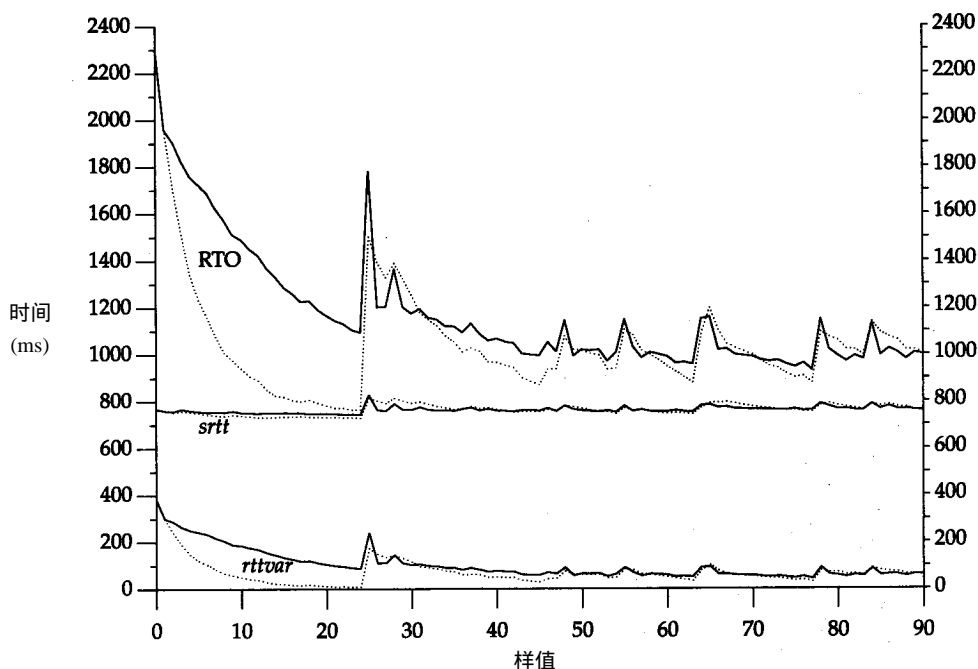


图10-7 TCP平滑与T/TCP平滑的比较

图10-7给出了从图10-6中的数据用常规 TCP 计算方法算出的结果与用滤波器增益 0.75 平滑的计算结果之间的比较。图中的三条虚线就是图10-6中的三个变量(RTO在最上方， $\text{srtt}$ 在中间， $\text{rttvar}$ 在底部)。三条实线则是假定每一个数据点都是一个独立 T/TCP 连接(每一个连接有一个 RTT 测量值)所对应的变量，并且采用滤波增益 0.75 进行了平滑。要知道有这样的差别：虚线对应的是一个 TCP 连接在 30 分钟内的 91 个 RTT 样本；而实线对应的则是在同样的 30 分钟内 91

个独立的T/TCP连接, 每个连接有一次RTT测量。实线同时还是91个连接的所有相继两个估计值归并后记录到两个路由度量值中的。

代表 $srtt$ 的实线和虚线差别不大, 但是代表 $rttvar$ 的实线和虚线之间就有明显的差别。 $rttvar$ 的实线(T/TCP情况)取值通常大于虚线(单个TCP连接), 使T/TCP的重传超时时间可以取更大的值。

还有其他因素也在影响T/TCP中的RTT测量。从客户端来看, 所测得的RTT通常包括服务器的处理时间或者服务器的延迟ACK定时值, 因为服务器的应答通常会延迟到这些事件发生后才给出。在Net/3中, 延迟ACK的定时器值是每200ms到时一次, 而RTT测量的时间单位为500 ms, 因此应答时延不会是一个大的因素。而且T/TCP报文段的处理常常会在TCP输入处理中遭遇慢通道(例如, 报文段常常不被用于首部预测), 会加大测得的RTT值(然而快通道与慢通道的差别相对于200 ms的延迟ACK定时器值来说很可能是可以忽略的)。最后, 如果存储在路由表中的值“过时”了(就是说, 其最后一次更新是在一个小时以前), 在当前事务完成以后, 或许应该用当前的测量值直接替换路由表中的值, 而不是用新的测量值与已有的测量值归并。

如RFC 1644中所指出的, 需要对TCP中的动态特性作更多的研究, 特别是T/TCP, 以及RTT估计。

## 10.6 tcp\_close函数

`tcp_close`的唯一改变是要为T/TCP事务记录RTT估计值, 即使还没有凑足16个样值。我们在前一节中已经叙述了这样做的原因。图10-8给出了代码。

```

252     if (SEQ_LT(tp->iss + so->so_snd.sb_hiwat * 16, tp->snd_max) && tcp_subr.c
253         (rt = inp->inp_route.ro_rt) &&
254         ((struct sockaddr_in *) rt_key(rt))->sin_addr.s_addr != INADDR_ANY) {

    /* pp. 895-896 of Volume 2 */

304     } else if (tp->cc_recv != 0 &&
305         (rt = inp->inp_route.ro_rt) &&
306         ((struct sockaddr_in *) rt_key(rt))->sin_addr.s_addr != INADDR_ANY) {
307         /*
308          * For transactions we need to keep track of srtt and rttvar
309          * even if we don't have 'enough' data for above.
310          */
311         u_long i;
312         if ((rt->rt_rmx.rmx_locks & RTV_RTT) == 0) {
313             i = tp->t_srtt *
314                 (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));
315             if (rt->rt_rmx.rmx_rtt && i)
316                 /*
317                  * Filter this update to 3/4 the old plus
318                  * 1/4 the new values, converting scale.
319                  */
320                 rt->rt_rmx.rmx_rtt =
321                     (3 * rt->rt_rmx.rmx_rtt + i) / 4;
322             else
323                 rt->rt_rmx.rmx_rtt = i;
324         }
    
```

图10-8 `tcp_close` 函数: 为T/TCP事务保存RTT估计值



```

325     if ((rt->rt_rmx.rmx_locks & RTV_RTTVAR) == 0) {
326         i = tp->t_rttvar *
327             (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
328         if (rt->rt_rmx.rmx_rttvar && i)
329             rt->rt_rmx.rmx_rttvar =
330                 (3 * rt->rt_rmx.rmx_rttvar + i) / 4;
331         else
332             rt->rt_rmx.rmx_rttvar = i;
333     }
334 }

```

—tcp\_subr.c

图10-8 (续)

### 1. 只对T/TCP事务进行更新

304-311 只有在连接中使用了T/TCP(cc\_recv非0)、有一路由表记录项存在及不是默认路由时才更新路由表记录项中的度量值。而且,只有当两个 RTT估计值没有加锁(RTV\_RTT和RTV\_RTTVAR位)时才更新。

### 2. 更新RTT

312-324 t\_srtt是以500 ms × 8为时间单位保存的, rmx\_rtt则以μs为单位保存。这样, t\_srtt就要乘1 000 000(RTM\_RTTUNIT)除2(时间单位/秒)再乘8。如果rmx\_rtt已经有值,新记录值就是旧值的四分之三加上新值的四分之一。这就是取滤波增益为 0.75,我们在前一节已讨论过。否则,直接将新值保存到 rmx\_rtt中。

### 3. 更新平均偏差

325-334 对平均偏差估计值应用同样的算法。它也以 ms为单位保存,需要将t\_rttvar中的单位时间 × 4。

## 10.7 tcp\_msssend函数

在Net/3中,有一个函数tcp\_mss(卷2的27.5节),在处理MSS选项时tcp\_input要调用它,在需要发送MSS选项时tcp\_output也要调用它。在T/TCP中,这个函数改名为tcp\_mssrcvd,在执行隐式连接建立时,收到SYN后,tcp\_input要调用它(在后面的图10-18中,确定是否需要在SYN中包含MSS选项),以及PRU\_SEND和PRU\_SEND\_EOF请求要调用它(见图12-4)。有一个新的函数tcp\_msssend,如图10-9所示,只有当发出了MSS选项时,才会被tcp\_output调用。

—tcp\_input.c

```

1911 int
1912 tcp_msssend(tp)
1913 struct tcpcb *tp;
1914 {
1915     struct rtentry *rt;
1916     extern int tcp_mssdflt;

1917     rt = tcp_rtlookup(tp->t_inpcb);
1918     if (rt == NULL)
1919         return (tcp_mssdflt);

1920     /*
1921      * If there's an mtu associated with the route, use it,

```

图10-9 tcp\_msssend 函数:返回MSS值,并在MSS选项中发出

```

1922      * else use the outgoing interface mtu.
1923      */
1924      if (rt->rt_rmx.rmx_mtu)
1925          return (rt->rt_rmx.rmx_mtu - sizeof(struct tcphdr));

1926      return (rt->rt_ifp->if_mtu - sizeof(struct tcphdr));
1927 }

```

tcp\_input.c

图10-9 (续)

### 1. 读取路由表记录项

1917-1919 为每一个对等主机搜索路由表, 如果没有找到记录项, 则返回默认值 512(tcp\_mssdflt)。除非对等主机不可达, 否则总是可以查找到一个路由表记录项的。

### 2. 返回MSS

1920-1926 如果路由表有一个关联的MTU(rt\_metrics结构中的rmx\_mtu字段, 系统管理员可以用route程序设置), 就返回该值。否则, 返回值就取输出接口的 MTU减去40(例如, 以太网上是1460)。因为路由已经由tcp\_rtlookup确定, 输出接口也是已知的。

在路由表中存储 MTU度量的另一个来源是利用路由 MTU发现过程(卷1的24.2节), 尽管Net/3中还不支持这种方法。

这个函数不同于通常的BSD做法。如果对等端是非本地主机(由in\_localaddr函数决定), 而且rmx\_mtu度量值为0, 则Net/3代码(卷2第719页)中总是将MSS取为512(tcp\_mssdflt)。

MSS选项的目的是告诉另一端, 该选项发送者准备接收多大报文段。RFC 793中指出, MSS选项“用于交流发送这个报文段的TCP的最大可接收报文段”。在一些实现中, 这可能受主机能够重装的最大的IP数据报限制。然而在当前的多数系统中, 合理的限制决定于输出接口的MTU, 因为如果需要分段并且发生报文段丢失, 则TCP的性能会下降。

下面的注释摘抄于Bob Braden的T/TCP源码修改:“非常不幸, 使用TCP选项要求对BSD作可观的修改, 因为它对MSS的处理是错误的。BSD总是要发出MSS选项, 并且对非本地网络的主机, 这个选项的值是536。这是对MSS选项用途的误解, 这个选项是要告诉发送者, 接收者准备处理什么。这时发送主机要决定用多大的MSS, 既要考虑它接收的MSS选项, 还要考虑到路由情况。当有了MTU发现以后, 这个路由很可能有一个大于536的MTU; 这样, BSD就会降低吞吐率。因此, 这个程序只确定了应该发送什么样的MSS选项: 本地接口的MTU减去40。”(这段注释中讲到的值536应为512)。

我们在下一节(图10-12)中会看到, 如果对等端是非本地主机, MSS选项的接收者才把MSS减到512。

## 10.8 tcp\_mssrcvd函数

在执行隐式连接建立时, 收到SYN以后的tcp\_input要调用tcp\_mssrcvd, PRU\_SEND和PRU\_SEND\_EOF也都要调用它。该函数与卷2中的tcp\_mss函数相似, 但是它们之间还是有足够的差别, 能够完成我们所需的全部功能。这个函数的主要目标是设置两个变量, 一个是t\_maxseg(我们在每个报文段中发送的最大数据量), 另一个是t\_maxopd(在每个报文段中发送的数据加选项的最大长度)。图10-10给出了这个函数的第一部分。



tcp\_input.c

```

1755 void
1756 tcp_mssrcvd(tp, offer)
1757 struct tcpcb *tp;
1758 int      offer;
1759 {
1760     struct rtentry *rt;
1761     struct ifnet *ifp;
1762     int      rtt, mss;
1763     u_long   bufsize;
1764     struct inpcb *inp;
1765     struct socket *so;
1766     struct rmxp_tao *taop;
1767     int      origoffer = offer;
1768     extern int tcp_mssdflt;
1769     extern int tcp_do_rfc1323;
1770     extern int tcp_do_rfc1644;

1771     inp = tp->t_inpcb;
1772     if ((rt = tcp_rtlookup(inp)) == NULL) {
1773         tp->t_maxopd = tp->t_maxseg = tcp_mssdflt;
1774         return;
1775     }
1776     ifp = rt->rt_ifp;
1777     so = inp->inp_socket;

1778     taop = rmxp_tao(rt->rt_rmx);
1779     /*
1780      * Offer == -1 means we haven't received a SYN yet;
1781      * use cached value in that case.
1782      */
1783     if (offer == -1)
1784         offer = taop->tao_mssopt;
1785     /*
1786      * Offer == 0 means that there was no MSS on the SYN segment,
1787      * or no value in the TAO Cache. We use tcp_mssdflt.
1788      */
1789     if (offer == 0)
1790         offer = tcp_mssdflt;
1791     else
1792         /*
1793          * Sanity check: make sure that maxopd will be large
1794          * enough to allow some data on segments even if all
1795          * the option space is used (40 bytes). Otherwise
1796          * funny things may happen in tcp_output.
1797          */
1798         offer = max(offer, 64);
1799     taop->tao_mssopt = offer;

```

tcp\_input.c

图10-10 tcp\_mssrcvd 函数：第一部分

### 1. 取对等端的路由及其TAO缓存

1771-1777 tcp\_rtlookup查找到达对等端的路由。如果由于某种原因，查找路由失败了，t\_maxseg和t\_maxopd就同时设置为512(tcp\_mssdflt)。

1778-1799 taop指向该对等端的TAO缓存，位于路由表的记录项中。如果因为用户进程调用了sendto(一次隐式连接建立，是PRU\_SEND和PRU\_SEND\_EOF请求的一部分)而调用tcp\_mssrcvd，则offer设置为TAO缓存中保存的值。如果TAO中的该值为0，offer就设置为512。TAO缓存中的值被更新。

图10-11给出了该函数的第二部分, 与卷2第718页完全相同。

```

1800      /*
1801      * While we're here, check if there's an initial rtt
1802      * or rttvar. Convert from the route-table units
1803      * to scaled multiples of the slow timeout timer.
1804      */
1805      if (tp->t_srtt == 0 && (rtt = rt->rt_rmx.rmx_rtt)) {
1806          /*
1807          * XXX the lock bit for RTT indicates that the value
1808          * is also a minimum value; this is subject to time.
1809          */
1810          if (rt->rt_rmx.rmx_locks & RTV_RTT)
1811              tp->t_rttmin = rtt / (RTM_RTTUNIT / PR_SLOWHZ);
1812          tp->t_srtt = rtt / (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));
1813          if (rt->rt_rmx.rmx_rttvar)
1814              tp->t_rttvar = rt->rt_rmx.rmx_rttvar /
1815                  (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
1816          else
1817              /* default variation is +- 1 rtt */
1818              tp->t_rttvar =
1819                  tp->t_srtt * TCP_RTTVAR_SCALE / TCP_RTT_SCALE;
1820          TCPT_RANGESET(tp->t_rxtcur,
1821                      ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1,
1822                      tp->t_rttmin, TCPTV_REXMTMAX);
1823      }

```

tcp\_input.c

图10-11 tcp\_mssrcvd 函数: 用路由表度量值初始化 RTT 变量

1800-1823 如果还没有该连接的 RTT 测量值( $t\_srtt$  为 0), 并且  $rmx\_rtt$  度量值为非 0, 这时变量  $t\_srtt$ 、 $t\_rttvar$  和  $t\_rxtcur$  就用路由表记录项中保存的度量值初始化。

如果路由表度量值加锁标志中的  $RTV\_RTT$  位已经设置, 则它表明还要用  $rmx\_rtt$  来初始化这次连接的最小 RTT ( $t\_rttmin$ )。默认情况下,  $t\_rttmin$  初始化为两个时钟步进, 这为系统管理员替换该默认值提供了一个方法。

图10-12给出了  $tcp\_mssrcvd$  的第三部分, 用于设置自动变量  $mss$  的值。

```

1824      /*
1825      * If there's an mtu associated with the route, use it.
1826      */
1827      if (rt->rt_rmx.rmx_mtu)
1828          mss = rt->rt_rmx.rmx_mtu - sizeof(struct tcpiphdr);
1829      else {
1830          mss = ifp->if_mtu - sizeof(struct tcpiphdr);
1831          if (!in_localaddr(inp->inp_faddr))
1832              mss = min(mss, tcp_mssdflt);
1833      }
1834      mss = min(mss, offer);

1835      /*
1836      * t_maxopd contains the maximum length of data AND options
1837      * in a segment; t_maxseg is the amount of data in a normal
1838      * segment. We need to store this value (t_maxopd) apart
1839      * from t_maxseg, because now every segment can contain options
1840      * therefore we normally have somewhat less data in segments.
1841      */
1842      tp->t_maxopd = mss;

```

tcp\_input.c

图10-12 tcp\_mssrcvd 函数: 计算 mss 变量的值

1824-1834 如果该路由关联于一个MTU(rmx\_mtu度量值),那就用这个值。否则, mss就取输出接口的MTU减去40。另外,如果对等端是在另一个网络,或者也可能在另一个子网(由in\_localaddr函数决定)中,这时mss的最大值取为512(tcp\_mssdflt)。如果路由表记录项中已经保存有MTU,那就不再进行本地-非本地测试。

## 2. 设置t\_maxopd

1835-1842 t\_maxopd设置为mss,包括了数据和选项的最大报文段长度。

图10-13给出的是第四部分代码,将mss减去在每一个报文段中都有的选项长度。

```

1843      /*                                     tcp_input.c
1844      * Adjust mss to leave space for the usual options. We're
1845      * called from the end of tcp_dooptions so we can use the
1846      * REQ/RCVD flags to see if options will be used.
1847      */
1848      /*
1849      * In case of T/TCP, origoffer == -1 indicates that no segments
1850      * were received yet (i.e., client has called sendto). In this
1851      * case we just guess, otherwise we do the same as before T/TCP.
1852      */
1853      if ((tp->t_flags & (TF_REQ_TSTMP | TF_NOOPT)) == TF_REQ_TSTMP &&
1854          (origoffer == -1 ||
1855           (tp->t_flags & TF_RCVD_TSTMP) == TF_RCVD_TSTMP))
1856          mss -= TCPOLEN_TSTAMP_APPA;

1857      if ((tp->t_flags & (TF_REQ_CC | TF_NOOPT)) == TF_REQ_CC &&
1858          (origoffer == -1 ||
1859           (tp->t_flags & TF_RCVD_CC) == TF_RCVD_CC))
1860          mss -= TCPOLEN_CC_APPA;

1861      #if (MCLBYTES & (MCLBYTES - 1)) == 0
1862          if (mss > MCLBYTES)
1863              mss &= ~(MCLBYTES - 1);
1864      #else
1865          if (mss > MCLBYTES)
1866              mss = mss / MCLBYTES * MCLBYTES;
1867      #endif

```

图10-13 tcp\_mssrcvd 函数：根据选项减小mss

## 3. 如果使用时间戳选项就减小mss

1843-1856 如果下面中的任何一个条件为真,则 mss就减去时间戳选项的长度(TCPOLEN\_TSTAMP\_APPA,即12字节):

- 1) 本地端将使用时间戳选项(TF\_REQ\_TSTAMP),并且还没有收到另一端发来的mss选项(origoffer等于-1);或
- 2) 已经收到另一个端发来的时间戳选项。

在代码的注释中指出,由于tcp\_mssrcvd是在tcp\_dooptions结束时所有的选项处理完以后调用的(见图10-18),因此第二项测试是成功的。

## 4. 如果使用CC选项,就减少mss

1857-1860 通过相似的逻辑,mss的值减去8字节(TCPOLEN\_CC\_APPA)。

这两个长度名称中出现术语APPA是因为,RFC 1323的附录A中建议在时间戳选项前面置两个空字符NOP,以便两个4字节时间戳值的长度都能取4字节的整数倍。

同时RFC 1644也有一个附录A，它对选项排列没有说什么。无论怎样，在三个CC选项的任一个前面都置两个NOP是有一定道理的，如图9-6所示。

### 5. 舍入MSS为MCLBYTES的倍数

1861-1867 mss要舍入取整为MCLBYTES的整数倍，即每个mbuf簇的字节数(通常为1024或2048)。

这段代码有一个糟糕的优化企图，即如果MCLBYTES是2的整数幂，则可以用逻辑操作来代替乘法或除法运算。自从Net/1开始，它就已经是一条弯路，应该清除掉。

图10-14给出了tcp\_mssrcvd代码的最后一部分，用于设置发送缓存和接收缓存的大小。

```

1868      /*
1869      * If there's a pipesize, change the socket buffer
1870      * to that size. Make the socket buffers an integral
1871      * number of mss units; if the mss is larger than
1872      * the socket buffer, decrease the mss.
1873      */
1874      if ((bufsize = rt->rt_rmx.rmx_sendpipe) == 0)
1875          bufsize = so->so_snd.sb_hiwat;
1876      if (bufsize < mss)
1877          mss = bufsize;
1878      else {
1879          bufsize = roundup(bufsize, mss);
1880          if (bufsize > sb_max)
1881              bufsize = sb_max;
1882          (void) sbreserve(&so->so_snd, bufsize);
1883      }
1884      tp->t_maxseg = mss;

1885      if ((bufsize = rt->rt_rmx.rmx_rcvpipe) == 0)
1886          bufsize = so->so_rcv.sb_hiwat;
1887      if (bufsize > mss) {
1888          bufsize = roundup(bufsize, mss);
1889          if (bufsize > sb_max)
1890              bufsize = sb_max;
1891          (void) sbreserve(&so->so_rcv, bufsize);
1892      }
1893      /*
1894      * Don't force slow-start on local network.
1895      */
1896      if (!in_localaddr(inp->inp_faddr))
1897          tp->snd_cwnd = mss;

1898      if (rt->rt_rmx.rmx_ssthresh) {
1899          /*
1900          * There's some sort of gateway or interface
1901          * buffer limit on the path. Use this to set
1902          * the slow start threshold, but set the
1903          * threshold to no less than 2*mss.
1904          */
1905          tp->snd_ssthresh = max(2 * mss, rt->rt_rmx.rmx_ssthresh);
1906      }
1907  }

```

图10-14 tcp\_mssrcvd 函数：设置发送和接收缓存的大小

### 6. 改变插口发送缓存的大小

1868-1883 系统管理员可以用route程序设置rmx\_sendpipe和rmx\_recvpipe这两个度量值。bufsize的值就设置为rmx\_sendpipe的值(如果已有定义),或者当前插口发送缓存的高位值。如果bufsize的值小于mss, mss值就减小为取bufsize的值(这是一种强迫MSS取比给定目标的默认值还小的取值方法)。否则, bufsize的值放大,取mss的整数倍(插口缓存的大小总是取报文段长度的整数倍)。上限为sb\_max,在Net/3就是262 144。插口缓存的高位值由sbreserve设置。

### 7. 设置t\_maxseg

1884 t\_maxseg设置为TCP将发给对等端的最大数据量(不包括常规选项)。

### 8. 改变插口接收缓存的大小

1885-1892 插口接收缓存的高位值可以用类似的逻辑来设置。例如,对于以太网上的本地连接来说,假定时间戳选项和CC选项同时都在用,则t\_maxopd将是1460, t\_maxseg为1440(见图2-4)。插口发送缓存和接收缓存都将从它们的默认值 8192(卷2的图16-4)舍入到8640(1440 × 6)。

### 9. 非本地对等端才有的慢启动

1893-1897 如果对等端不是在本地的网络中(in\_localaddr为假),则把拥塞窗口(snd\_cwnd)设置为1个报文段就开始了慢启动过程。

仅仅当对等端在非本地网中才强迫使用慢启动是T/TCP修改后的结果。这就使T/TCP的客户端或服务端可以向本地对等端发送多个报文段,又不需要慢启动所要求的额外RTT等待时间(见3.6节)。在Net/3中,总是执行慢启动过程(卷2第721页)。

### 10. 设置慢启动门限

1898-1906 如果慢启动门限度量值(rmx\_ssthresh)非0, snd\_ssthresh就设置取该值。

我们在图3-1和图3-3中可以看到MSS和TAO缓存与接收缓存大小之间的交叉影响。在图3-1中,客户端执行了一次隐式连接建立, PRU\_SEND\_EOF请求调用tcp\_mssrcvd,其中offer为-1,该函数查找到对应服务器的tao\_mssopt值为0(因为客户端刚刚重启)。取MSS为默认值512,因为只使用了CC选项(在第2章的例子中,时间戳无效),减去8字节后变为504。注意,8192舍入为504的整数倍后为8568,这是客户端SYN所通告的窗口。然而,当服务器调用tcp\_mssrcvd时,它已经接收到客户端的SYN,其中给定MSS为1460。这个值减去8字节(选项长度)后为1452,8192舍入到1452的整数倍后为8172。这是服务器的SYN中通告的窗口。当客户端处理完服务器的SYN后(图中第三段),客户端再次调用tcp\_mssrcvd,这一次offer为1460。这就将客户端的t\_maxopd增大至1460,客户端的t\_maxseg则增大至1452,客户端的接收缓存因舍入而增至8172。这就是客户端在对服务器的SYN作出ACK时通告的窗口。

在图3-3中,当客户端执行了隐式连接建立时,tao\_mssopt值为1460——最近一次从对等端收到的值。客户端通告的窗口为8712,1452的整数倍且大于8192。

## 10.9 tcp\_dooptions函数

在Net/1和Net/2版中,tcp\_dooptions只能识别NOP、EOL和MSS选项,并且函数有3个参数。在Net/3中增加了对窗口宽度和时间戳选项的支持后,参数的数量也增加到7个(卷2第

745~746页),其中有3个就是为了时间戳选项而加的。现在又需要支持 CC、CCnew和CCecho选项,参数的数量不是增加反而减少到了5个,因为采用了另一种技术来返回选项是否存在以及它们各自的取值信息。

图10-15给出了 `tcptopt` 结构。其中的一个结构是在 `tcp_input`(唯一可以调用 `tcp_dooptions`的函数)中分配的,并且将指向该结构的指针传给 `tcp_dooptions`,该函数填写结构的内容。在处理接收到的报文段时, `tcp_input`要用到存储在该结构中的值。

```

138 struct tcptopt {                                     tcp_var.h
139     u_long  to_flag;                                /* TOF_*** flags */
140     u_long  to_tsval;                                /* timestamp value */
141     u_long  to_tsecr;                                /* timestamp echo reply */
142     tcp_cc  to_cc;                                   /* CC or CCnew value */
143     tcp_cc  to_ccecho;                               /* CCecho value */
144 };                                                     tcp_var.h

```

图10-15 `tcptopt` 结构,由 `tcp_dooptions` 填写数据

图10-16给出了 `to_flag` 字段可以组合出的4个值。

to_flag	说 明
TOF_CC	CC选项存在
TOF_CCNEW	CCnew选项存在
TOF_CCECHO	CCecho选项存在
TOF_TS	时间戳选项存在

图10-16 `to_flag` 的取值

图10-17给出了这个函数的参数说明。前4个参数与 Net/3 中的相同,但第5个参数替换了 Net/3 版本中的最后3个参数。

```

1520 void                                                     tcp_input.c
1521 tcp_dooptions(tp, cp, cnt, ti, to)
1522 struct tcpcb *tp;
1523 u_char *cp;
1524 int cnt;
1525 struct tcpiphdr *ti;
1526 struct tcptopt *to;
1527 {                                                         tcp_input.c

```

图10-17 `tcp_dooptions` 函数:参数

因为处理 EOL、NOP、MSS、窗口宽度和时间戳选项的代码与卷2第745~747页的代码几乎相同,所以这里不再重复介绍(差别主要在于对新参数的处理,我们刚刚讨论过)。图10-18给出了这个函数的最后一部分代码,它们用于 T/TCP 处理3个新的选项。

#### 1. 检查长度和是否处理选项

1580-1584 选项长度要验证(所有3个CC选项的长度必须都是6)。处理接收到的CC选项时,我们也必须发送相应选项(如果内核的 `tcp_do_rfc1644` 标志已经设置,则 `tcp_newtcpcb` 要设置 `TF_REQ_CC` 标志),并且 `TF_NOOPT` 标志不能设置(最后这个标志不允许 TCP 在其 SYN 中发送任何选项)。



## 2. 设置相应标志并复制4字节值

1585-1588 设置相应的to\_flag值。四个字节的选项值存储在tcptopt结构的to\_cc字段中,并且要先转换成主机的字节顺序。

1589-1595 如果这是一个SYN报文段,要为该连接设置TF\_RCVD\_CC标志,因为收到了CC选项。

## 3. CCnew和CCecho选项

1596-1623 CCnew和CCecho选项的处理步骤与CC选项的相似。但因为CCnew和CCecho选项仅在SYN报文段中有效,所以要附加一项检测,检查报文段中是否包含SYN标志。

尽管TOF\_CCNEW标志都有正确设置,但从来不去检查它。这是因为在图11-6中,如果CC选项不存在,则缓存的CC值是无效的(即需设置为0)。如果存在CCnew选项,则cc\_recv仍然有正确设置(注意,在图10-18中,CC和CCnew选项都在to\_cc中存储其值),并且当三次握手完成时(图11-14),所缓存的值tao\_cc是从cc\_recv中复制过来的。

## 4. 处理收到的MSS

1625-1626 局部变量mss记录的或者是MSS选项的值(如果选项存在),或者是表示选项不存在的0值。在这两种情况下,tcp\_mssrcvd都要设置变量t\_maxseg和t\_maxopd的值。在tcp\_dooptions快结束时要调用该函数,因为如图10-13所示,tcp\_mssrcvd使用了TF\_RCVD\_TSTMP和TF\_RCVD\_CC标志。

## 10.10 tcp\_reass函数

当服务器收到包含数据的SYN时,假定TAO测试失败或报文段中不包含CC选项,那么tcp\_input就将数据存入缓存队列,等待三次握手过程的完成。在图11-6中,协议的状态设置为SYN\_RCVD,程序有一个分支trimthenstep6,在标号为dodata的程序行(卷2第790页),宏TCP\_REASS发现协议状态不是ESTABLISHED,因此调用tcp\_reass将报文段存入该连接的失序报文队列(其中的数据并非真的失序;只是因为它是在三次握手过程完成之前到达的。然而,卷2的图27-19底部的两个统计计数器tcps\_rcvooack和tcps\_rcvooobyte的累进是不正确的)。

tcp\_input.c

```

1580     case TCPOPT_CC:
1581         if (optlen != TCPOLEN_CC)
1582             continue;
1583         if ((tp->t_flags & (TF_REQ_CC | TF_NOOPT)) != TF_REQ_CC)
1584             continue; /* we're not sending CC opts */
1585         to->to_flag |= TOF_CC;
1586         bcopy((char *) cp + 2, (char *) &to->to_cc,
1587             sizeof(to->to_cc));
1588         NTOHL(to->to_cc);
1589         /*
1590          * A CC or CCnew option received in a SYN makes
1591          * it OK to send CC in subsequent segments.
1592          */
1593         if (ti->ti_flags & TH_SYN)
1594             tp->t_flags |= TF_RCVD_CC;

```

图10-18 tcp\_dooptions 函数:新T/TCP选项的处理

```

1595         break;

1596     case TCPOPT_CCNEW:
1597         if (optlen != TCPOLEN_CC)
1598             continue;
1599         if ((tp->t_flags & (TF_REQ_CC | TF_NOOPT)) != TF_REQ_CC)
1600             continue; /* we're not sending CC opts */
1601         if (!(ti->ti_flags & TH_SYN))
1602             continue;
1603         to->to_flag |= TOF_CCNEW;
1604         bcopy((char *) cp + 2, (char *) &to->to_cc,
1605             sizeof(to->to_cc));
1606         NTOHL(to->to_cc);
1607         /*
1608          * A CC or CCnew option received in a SYN makes
1609          * it OK to send CC in subsequent segments.
1610          */
1611         tp->t_flags |= TF_RCVD_CC;
1612         break;

1613     case TCPOPT_CCECHO:
1614         if (optlen != TCPOLEN_CC)
1615             continue;
1616         if (!(ti->ti_flags & TH_SYN))
1617             continue;
1618         to->to_flag |= TOF_CCECHO;
1619         bcopy((char *) cp + 2, (char *) &to->to_ccecho,
1620             sizeof(to->to_ccecho));
1621         NTOHL(to->to_ccecho);
1622         break;
1623     }
1624 }
1625 if (ti->ti_flags & TH_SYN)
1626     tcp_mssrcvd(tp, mss); /* sets t_maxseg */
1627 }

```

tcp\_input.c

图10-18 (续)

当对服务器所发的SYN的ACK(通常是三次握手中的第三个报文段)姗姗来迟时,执行卷2第774页的case TCPS\_SYN\_RECEIVED语句,使连接进入到ESTABLISHED状态,然后调用tcp\_reass函数将队列中的数据交付给进程,该函数第二个参数为0。但在图11-14中,我们会看到,如果新的报文段中有数据,或者如果设置了FIN标志,就跳过对tcp\_reass函数的调用,因为这两种情况的任何一种都会引起对标号为dodata的TCP\_REASS函数的调用。问题是,如果新的报文段完全与以前的报文段重复,则对TCP\_REASS函数的调用不会强行将队列中的数据交付给进程。

修改tcp\_reass函数只需做很小的改变:将卷2第729页的第106行的return改为执行标号为present的分支。

## 10.11 小结

给定主机的TAO信息保存在路由表的记录项中。函数tcp\_gettaocache读取为某主机缓存的TAO数据,但如果在PCB的路由缓存中尚不存在相应的路由,则调用tcp\_rtlookup来查找主机。

T/TCP修改`tcp_close`函数，在路由表中为 T/TCP连接保存两个估计值 *srtt*和*rttvar*，即使连接中只传送了不到 16个满长度的报文段。这样就使与该主机的下一次 T/TCP连接可以在开始时使用这两个估计值(假设路由表记录项在下次连接时还没有超时)。

Net/3的函数`tcp_mss`在T/TCP中分成了两个函数：`tcp_mssrcvd`和`tcp_msssend`。前者在收到MSS选项后调用，后者在发出MSS选项时调用。后者与通常的BSD做法的不同之处在于，它一般声明其MSS为输出接口的MTU减去TCP和IP首部的长度。BSD系统会向非本地对等主机声明取值为512的MSS。

Net/3中的`tcp_dooptions`函数在T/TCP中也有改变。函数的若干个参数取消了，用一个结构来代替。这就使函数可以处理新的选项(例如T/TCP新增加的3个选项)，而不需增加参数。