

## 第12章 T/TCP实现：TCP用户请求

### 12.1 概述

`tcp_usrreq`函数处理来自插口层的所有 `PRU_xxx`请求。在本章中我们仅仅介绍 `PRU_CONNECT`、`PRU_SEND`和`PRU_SEND_EOF`请求，因为T/TCP中只对这三个请求做了修改。我们也会介绍 `tcp_usrclosed`函数，当进程发送完数据时要调用这个函数。还有 `tcp_sysctl`函数也会介绍，它用来处理新的TCP中的`sysctl`变量。

我们不打算介绍 `tcp_ctloutput`函数(见卷2的30.6节)所需的修改，这个函数用于设置和读取两个新的插口选项：`TCP_NOPUSH`和`TCP_NOOPT`。所需的修改是非常细微具体的，只要阅读源代码就很容易理解。

### 12.2 PRU\_CONNECT请求

在Net/3中，大约需要 25行代码(卷2第808~809页)来处理 `tcp_usrreq`发出的 `PRU_CONNECT`请求。在T/TCP，大部分这些代码都移到了 `tcp_connect`函数中(下一节介绍)，只留下了图12-1所给出的代码。

```
137     case PRU_CONNECT:
138         if ((error = tcp_connect(tp, nam)) != 0)
139             break;
140         error = tcp_output(tp);
141         break;
```

*tcp\_usrreq.c*

图12-1 PRU\_CONNECT 请求

137-141 `tcp_connect`执行连接建立所需的步骤，`tcp_output`发出SYN报文段(主动打开)。

当某个进程调用 `connect`时，即使本地主机和待连接的对等端主机都支持 T/TCP，仍然要经历正常的三次握手过程。这是因为不可能用 `connect`函数传递数据，这样 `tcp_output`就仅仅发送 SYN。为了跳过三次握手过程，应用程序必须避免使用 `connect`，而是使用 `sendto`或`sendmsg`，并给定数据和対等端服务器的地址。

### 12.3 tcp\_connect函数

新的 `tcp_connect`函数执行主动打开所需的处理步骤。当进程调用 `connect` (`PRU_CONNECT`请求)或者当进程调用 `sendto`或`sendmsg`时，要改为调用该函数，指定待连接的对等端地址(`PRU_SEND`和`PRU_SEND_EOF`请求)。`tcp_connect`的第一部分在图12-2中给出。

#### 1. 绑定本地端口

308-312 `nam`指向一个Internet插口地址结构，其中包含待连接的服务器的IP地址和端口号。

如果还没有给插口指定一个本地端口(通常的情况),调用`in_pcbbind`就会分配一个端口(卷2第558页)。

## 2. 指定本地地址, 检查插口对的唯一性

313-323 如果还没有给插口绑定一个本地IP地址(通常的情况下),调用`in_pcbladdr`就可分配本地IP地址。`in_pcblookup`查找匹配的PCB,如果找到,就返回一个非空指针。仅仅在进程绑定了一个专门指定的本地端口时才可能找到一个匹配的PCB,因为如果`in_pcbbind`选择本地端口,就会选择一个目前不在使用的本地端口。但是在T/TCP中,更有可能的是一个客户端进程为一系列事务绑定同一个本地端口(见4.2节)。

## 3. 存在已有连接; 检查TIME\_WAIT状态是否可以截断

324-332 如果找到一个匹配的PCB,进行下面的三项测试:

- 1) PCB是否处于TIME\_WAIT状态;
- 2) 连接持续时间是否短于MSL;
- 3) 连接是否使用T/TCP(也就是说,是否从对等端收到了一个CC选项或CCnew选项)。

如果上述这三个条件同时为真,则调用`tcp_close`关闭现有的PCB。这就是我们在4.4节中讨论过的,当一个新的连接再次使用同一插口对并执行一次主动打开时,TIME\_WAIT状态的截断。

## 4. 在互连网PCB中完成插口对

333-336 如果本地地址还是通配符,则`in_pcbladdr`计算出的值存储在PCB中。外部地址和外部端口也存储在PCB中。

图12-2中的步骤与图7-5中的最后一部分相似。`tcp_connect`的最后一部分在图12-3中给出。这段代码与卷2第808~809页PRU\_CONNECT请求的最后一部分相似。

*tcp\_usrreq.c*

```

295 int
296 tcp_connect(tp, nam)
297 struct tcpcb *tp;
298 struct mbuf *nam;
299 {
300     struct inpcb *inp = tp->t_inpcb, *oinp;
301     struct socket *so = inp->inp_socket;
302     struct tcpcb *otpcb;
303     struct sockaddr_in *sin = mtod(nam, struct sockaddr_in *);
304     struct sockaddr_in *ifaddr;
305     int error;
306     struct rmxp_tao *taop;
307     struct rmxp_tao tao_noncached;

308     if (inp->inp_lport == 0) {
309         error = in_pcbbind(inp, NULL);
310         if (error)
311             return (error);
312     }
313     /*
314      * Cannot simply call in_pcbconnect, because there might be an
315      * earlier incarnation of this same connection still in
316      * TIME_WAIT state, creating an ADDRINUSE error.
317      */
318     error = in_pcbladdr(inp, nam, &ifaddr);
319     oinp = in_pcblookup(inp->inp_head,

```

图12-2 `tcp_connect` 函数：第一部分

```

320             sin->sin_addr, sin->sin_port,
321             inp->inp_laddr.s_addr != INADDR_ANY ?
322             inp->inp_laddr : ifaddr->sin_addr,
323             inp->inp_lport, 0);

324     if (oinp) {
325         if (oinp != inp && (otp = intotcpcb(oinp)) != NULL &&
326             otp->t_state == TCPS_TIME_WAIT &&
327             otp->t_duration < TCPTV_MSL &&
328             (otp->t_flags & TF_RCVD_CC))
329             otp = tcp_close(otp);
330         else
331             return (EADDRINUSE);
332     }
333     if (inp->inp_laddr.s_addr == INADDR_ANY)
334         inp->inp_laddr = ifaddr->sin_addr;
335     inp->inp_faddr = sin->sin_addr;
336     inp->inp_fport = sin->sin_port;

```

tcp\_usrreq.c

图12-2 (续)

```

337     tp->t_template = tcp_template(tp);
338     if (tp->t_template == 0) {
339         in_pcbdisconnect(inp);
340         return (ENOBUFS);
341     }
342     /* Compute window scaling to request. */
343     while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
344         (TCP_MAXWIN << tp->request_r_scale) < so->so_rcv.sb_hiwat)
345         tp->request_r_scale++;

346     soisconnecting(so);
347     tcpstat.tcps_connattempt++;
348     tp->t_state = TCPS_SYN_SENT;
349     tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;
350     tp->iss = tcp_iss;
351     tcp_iss += TCP_ISSINCR / 4;
352     tcp_sendseqinit(tp);

353     /*
354      * Generate a CC value for this connection and
355      * check whether CC or CCnew should be used.
356      */
357     if ((taop = tcp_gettaocache(tp->t_inpcb)) == NULL) {
358         taop = &tao_noncached;
359         bzero(taop, sizeof(*taop));
360     }
361     tp->cc_send = CC_INC(tcp_ccgen);
362     if (taop->tao_ccsent != 0 &&
363         CC_GEQ(tp->cc_send, taop->tao_ccsent)) {
364         taop->tao_ccsent = tp->cc_send;
365     } else {
366         taop->tao_ccsent = 0;
367         tp->t_flags |= TF_SENDCCNEW;
368     }

369     return (0);
370 }

```

tcp\_usrreq.c

图12-3 tcp\_connect 函数：第二部分

### 5. 初始化IP和TCP首部

337-341 `tcp_template`分配一个mbuf，用于缓存IP和TCP首部，并用尽可能多的信息来初始化这两个首部。

### 6. 计算窗口宽度因子

342-345 计算接收缓存的窗口宽度值。

### 7. 设置插口和连接的状态

346-349 `soisconnecting`在插口状态变量中设置特定的一些标志位，并设置TCP连接的状态为SYN\_SENT(如果进程给出MSG\_EOF标志，并调用`sendto`或者`sendmsg`，而不是调用`connect`，我们很快就会看到`tcp_usrclosed`设置TF\_SENDSYN隐藏状态标志，连接状态变迁到SYN\_SENT\*)。连接建立定时器初始化为75秒。

### 8. 初始化序号

350-352 从全局变量`tcp_iss`中复制初始发送序号，然后该全局变量值要增加，即加上除以4后的TCP\_ISSINCR。发送序号由`tcp_sendseqinit`初始化。

我们在3.2节中讨论过的ISS随机化在宏TCP\_ISSINCR中实现。

### 9. 生成CC值

353-361 读取对等端的TAO缓存记录项。全局变量`tcp_ccgen`值加上CC\_INC(见8.2节)后存储在T/TCP的变量`tcp_ccgen`中。如同我们以前所述，不论是否使用了CC选项，主机每建立一个连接，`tcp_ccgen`就要加1。

### 10. 确定是否使用CC或CCnew选项

362-368 如果对应这个主机的TAO缓存(`tao_ccsent`)非0(说明与该主机之间已经不是第一次连接)，并且`cc_send`的值大于或等于`tao_ccsent`(CC值还没有回到0，继续循环)，这时发出一个CC选项并用新的CC值更新TAO缓存。否则发送一个新的CCnew选项，并将`tao_ccsent`设置为0(即未定义)。

回想图4-12中，那里的情况可以作为上述if条件中的第二部分不成立的一个实例：最后一次发送这个主机的CC值是1(`tao_ccsent`)，但`tcp_ccgen`(对这个连接来说，变为`cc_send`)的当前值是2 147 483 648。这样，T/TCP就必须发送CCnew选项而不是CC选项，因为如果我们发出的CC选项值为2 147 483 648，而对方主机还在其缓存中记着我们上次发送的CC值(即1)，那个主机会强制执行三次握手操作，因为CC值已经回到0并继续循环。对方主机无法区分CC值为2 147 483 648的SYN是否是一个过时的重复报文段。而且，如果我们发送了CC选项，即使三次握手过程顺利完成，对方主机也不会更新对应于本主机的缓存记录项(请再看看图4-12)。如果发送的是CCnew选项，客户端强制执行三次握手操作，并且会使服务器在三次握手操作完成后更新对应于本主机的缓存值。

Bob Braden的T/TCP实现是在`tcp_output`中测试是发送CC选项还是CCnew选项，而不是在这个函数中。这就导致了一个微小的缺陷，见下面的解释 [Olah 1995]。考虑图4-11，但假定报文段1被中途的某个路由器丢弃。报文段2~4如图所示，从客户端口1601发起的连接成功地建立。客户端发出的下一个报文段是重传的报文段1，但其中包含一个取值为15的CCnew选项。假设该报文段成功地收到，服务器强制执行三次握手，完成以后，服务器将对应于该客户端的CC缓存值更新为15。如果此后

网络交付了一个过时的重复报文段 2，其中的 CC 值为 5000，服务器收到后就会收下。解决的方法是在客户端执行主动打开时判断是发送 CC 选项还是 CCnew 选项，而不是在 tcp\_output 函数中发送报文段时判断。

## 12.4 PRU\_SEND 和 PRU\_SEND\_EOF 请求

在卷 2 第 811 页中，对 PRU\_SEND 请求的处理仅仅是先调用 sbappend，然后再调用 tcp\_output。在 T/TCP 中，对这个请求的处理还是一样，只是代码中加上了对 PRU\_SEND\_EOF 请求的处理，如图 12-4 所示。我们可以看到，对 TCP，PRU\_SEND\_EOF 请求是在指定了 MSG\_EOF 标志(见图 5-2)并且当最后一个 mbuf 发送给协议时由 sosend 产生的。

```

189     case PRU_SEND_EOF:
190     case PRU_SEND:
191         sbappend(&so->so_snd, m);
192         if (nam && tp->t_state < TCPS_SYN_SENT) {
193             /*
194              * Do implied connect if not yet connected,
195              * initialize window to default value, and
196              * initialize maxseg/maxopd using peer's cached
197              * MSS.
198              */
199             error = tcp_connect(tp, nam);
200             if (error)
201                 break;
202             tp->snd_wnd = TTCP_CLIENT_SND_WND;
203             tcp_mssrcvd(tp, -1);
204         }
205         if (req == PRU_SEND_EOF) {
206             /*
207              * Close the send side of the connection after
208              * the data is sent.
209              */
210             socantsendmore(so);
211             tp = tcp_usrclosed(tp);
212         }
213         if (tp != NULL)
214             error = tcp_output(tp);
215         break;

```

tcp\_usrreq.c

tcp\_usrreq.c

图 12-4 PRU\_SEND 和 PRU\_SEND\_EOF 请求

### 1. 隐式连接建立

192-202 如果 nam 参数非空，进程就调用 sendto 或 sendmsg，并指定一个对等端地址。如果连接状态是 CLOSED 或 LISTEN，那么 tcp\_connect 就执行隐式连接建立。初始发送窗口设置为 4096(TTCP\_CLIENT\_SND\_WND)，因为在 T/TCP 中，客户端可以在收到服务器的窗口通告以前就发送数据(见 3.6 节)。

### 2. 为连接设置初始 MSS

203-204 调用 tcp\_mssrcvd 函数时第二个参数为 -1，表示我们还没有收到 SYN，所以用这个主机的缓存值(tao\_mssopt)作为初始 MSS。当 tcp\_mssrcvd 函数返回时，根据缓存的 tao\_mssopt 值或系统管理员在路由表记录项中设置的值(rt\_metrics 结构中的 rmx\_mtu

成员)设置变量 `t_maxseg` 和 `t_maxopd` 的值。如果并且当收到服务器发出的带有 MSS 选项的 SYN 时, `tcp_mssrcvd` 将再次被 `tcp_dooptions` 调用。因为在收到对等端的 MSS 选项之前就发出了数据,现在 T/TCP 需要在收到 SYN 之前就在 TCP 控制块中设置 MSS 变量的值。

### 3. 处理 MSG\_EOF 标志

205-212 如果进程指定了 MSG\_EOF 标志,这时 `socantsendmore` 就要设置插口的 `SS_CANTSENDMORE` 标志。然后 `tcp_usrclosed` 就把连接状态从 SYN\_SENT(由 `tcp_connect` 设置)变迁到 SYN\_SENT\* 状态。

### 4. 发送第一个报文段

213-214 `tcp_output` 检查是否应该发送报文段。在 T/TCP 客户端刚刚指定 MSG\_EOF 标志调用了 `sendto`(见图 1-10)时,这个调用就发出一个报文段,其中包含 SYN、数据和 FIN。

## 12.5 tcp\_usrclosed 函数

Net/3 中,在处理 PRU\_SHUTDOWN 请求时,该函数由 `tcp_disconnect` 调用。我们在图 12-4 中可以看到,在 T/TCP 中,这个函数也被 PRU\_SEND\_EOF 请求调用。图 12-5 给出了这个函数,替代卷 2 第 817 页中的代码。

```

533 struct tcpcb *                                     tcp_usrreq.c
534 tcp_usrclosed(tp)
535 struct tcpcb *tp;
536 {
537     switch (tp->t_state) {
538     case TCPS_CLOSED:
539     case TCPS_LISTEN:
540         tp->t_state = TCPS_CLOSED;
541         tp = tcp_close(tp);
542         break;
543     case TCPS_SYN_SENT:
544     case TCPS_SYN_RECEIVED:
545         tp->t_flags |= TF_SENDFIN;
546         break;
547     case TCPS_ESTABLISHED:
548         tp->t_state = TCPS_FIN_WAIT_1;
549         break;
550     case TCPS_CLOSE_WAIT:
551         tp->t_state = TCPS_LAST_ACK;
552         break;
553     }
554     if (tp && tp->t_state >= TCPS_FIN_WAIT_2)
555         soisdisconnected(tp->t_inpcb->inp_socket);
556     return (tp);
557 }

```

tcp\_usrreq.c

图12-5 tcp\_usrclosed 函数

541-546 在 T/TCP 中,通过设置 TF\_SENDFIN 状态标志,用户在 SYN\_SENT 或 SYN\_RVD 状态下发起关闭过程,将状态变迁到相应的加星状态。其余的状态变迁在 T/TCP 中没有改变。

## 12.6 tcp\_sysctl函数

在为T/TCP而做修改时，用sysctl程序修改TCP变量的能力也同时加上了。T/TCP对此功能并没有严格要求，但这个功能提供了改变特定TCP变量值的一个简便方法，而不必再使用调试程序对内核进行修补。TCP变量都以前缀net.inet.tcp来标识访问。在TCP protosw结构的pr\_sysctl字段中(卷2第641页)记录着指向该函数的一个指针。图12-6给出了这个函数。

570-572 目前只支持三个变量，但是很容易加上更多的变量。

```
tcp_usrreq.c
561 int
562 tcp_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
563 int *name;
564 u_int namelen;
565 void *oldp;
566 size_t *oldlenp;
567 void *newp;
568 size_t newlen;
569 {
570     extern int tcp_do_rfc1323;
571     extern int tcp_do_rfc1644;
572     extern int tcp_mssdflt;

573     /* All sysctl names at this level are terminal. */
574     if (namelen != 1)
575         return (ENOTDIR);

576     switch (name[0]) {
577     case TCPCTL_DO_RFC1323:
578         return (sysctl_int(oldp, oldlenp, newp, newlen, &tcp_do_rfc1323));
579     case TCPCTL_DO_RFC1644:
580         return (sysctl_int(oldp, oldlenp, newp, newlen, &tcp_do_rfc1644));
581     case TCPCTL_MSSDFLT:
582         return (sysctl_int(oldp, oldlenp, newp, newlen, &tcp_mssdflt));
583     default:
584         return (ENOPROTOOPT);
585     }
586     /* NOTREACHED */
587 }
```

图12-6 tcp\_sysctl 函数

## 12.7 T/TCP的前景

有一件有趣的事，看看在RFC 1323中定义的TCP修改方案的普及，实际上是关于窗口宽度和时间戳选项的变化。这些变化受日益增长的网络速度(T3电话线路和FDDI)以及潜在的长时延路由(卫星线路)等的驱动。Thomas Skibo为SGI工作站所完成的修改是最早的实现之一。然后他又在伯克利Net/2版中做了这些修改，使这些修改在1992年5月可以公开得到(图1-16中详细给出了各个BSD版本之间的区别及其发行时间)。大约一年以后(1993年4月)，Bob Braden和Liming Wei公布了SunOS 4.1.1中类似于RFC 1323的源码修改。1993年8月，伯克利把Skibo的修改加到了4.4BSD版中，这使公众在1994年4月可以得到4.4BSD-Lite版。到1995年，有一些销售商已经加上了对RFC 1323的支持，另有一些销售商则宣称准备加上对RFC 1323的支

持。但RFC 1323并不是很通用的，特别是PC机上的实现(事实上，在14.6节中我们会看到，只有不到2%的客户遇到过发送窗口宽度和时间戳选项的特殊WWW服务器)。

T/TCP很可能会走类似的路。1994年9月的第一次实现(见1.9节)只是对SunOS 4.1.3的源码做了修改，大多数用户对此都不是很感兴趣，除非他们在使用SunOS的源码。然而这只不过是T/TCP设计者的一个参考实现。普遍存在的80×86硬件平台上的FreeBSD实现(引入了SunOS源码中的修改部分)在1995年的早期就可以公开得到了，它应该会将T/TCP传播到很多的用户。

本书这部分章节的目的是用T/TCP实例来说明为什么T/TCP是对TCP的很有价值的改进，给出文档的细节并解释源码的变化。如同RFC 1323中的修改一样，T/TCP实现与非T/TCP实现可以互通，仅仅当两端同时都支持CC选项时才使用它。

## 12.8 小结

`tcp_connect`函数是新的，已经有了T/TCP所需的修改，显式`connect`要调用它，隐式连接建立(指定目标地址的`sendto`或者`sendmsg`)也调用它。如果连接使用的是T/TCP，并且持续时间短于MSL，则该函数允许还处于TIME\_WAIT状态的连接再次建立新连接。

PRU\_SEND\_EOF请求是新的，它在最后一次调用协议输出并且应用程序指定了MSG\_EOF标志时由插口层产生。该请求允许采用隐式连接建立，并且在指定了MSG\_EOF标志时还调用`tcp_usrclosed`。

对`tcp_usrclosed`函数所做的唯一修改是允许一个进程可以关闭尚处于SYN\_SENT或SYN\_RCVD状态的连接。这时要设置隐藏标志TF\_SENDFIN。