第3章　T/TCP使用举例

3.1　概述

本章中我们将通过几个T/TCP应用程序例子来学习如何使用这3个新引入的TCP选项。这几个例子说明，针对以下几种情形，T/TCP是如何处理的：

- 客户重新启动；
- 常规的T/TCP事务；
- 服务器收到一个过时的重复SYN报文段；
- 服务器重新启动；
- 请求或应答的长度超过报文段最大长度MSS；
- 与不支持T/TCP协议的主机的向下兼容；

下一章我们还将研究另外2个例子：SYN报文段到达服务器没有过时也不重复，但其到达的顺序错乱；客户对重复的服务器SYN/ACK响应的处理。

这些例子中的T/TCP客户是`bsdi`(图1-13)，而服务器则是`laptop`。这些主机上运行的T/TCP客户程序如图1-10所示；T/TCP服务器程序如图1-11所示。客户程序发出长度为300字节的请求，服务器则给出长度为400字节的应答。

在这些例子中，客户程序中支持RFC 1323的部分已经关闭。这样，在客户发起的SYN报文段中就不会含有窗口宽度和时间戳选项(由于只要客户不发送这两个选项，服务器的响应中也不会包含这两个选项，从而服务器是否支持RFC 1 323就是无关紧要的)。这样做是为了避免让那些与我们讨论的主题无关的因素把例子弄得太复杂。但在正常情况下，由于时间戳选项可以防止把重复的报文段误认为是当前连接的报文段，因而我们可以在T/TCP应用中支持RFC 1 323。也就是说，在宽带连接和大数据量传送的情况下，即便是T/TCP协议也一样需要防止序号重叠(PAWS，见卷1的24.6节)。

3.2　客户重新启动

客户一旦启动，客户-服务器事务过程也就开始了。客户程序调用`sendto`函数，即在路由表中为对端服务器增加一个表项，其中`tao_ccsent`的值初始化为0(表示未定义)。于是TCP协议就会发出CCnew选项而不是发出CC选项。服务器上的TCP协议收到CCnew选项后就执行常规的三次握手操作，其过程可见图3-1所示的Tcpdump的输出(不熟悉Tcpdump操作及其输出的读者可参见卷1的附录A。在跟踪观察这些分组的时候，不要忘了SYN和FIN在序号空间中各占用一个字节)。

从第1行的CCnew选项可以看出，客户端`tcp_ccgen`的值为1。在第2行，服务器对客户的CCnew给出了回应，服务器的`tcp_ccgen`值为18。服务器给客户的SYN发出确认，但不确认客户的数据。由于收到了客户的CCnew选项，即使服务器在其单机高速缓存中有该客户的表项，它也必须完成正常的三次握手过程。只有当三次握手完成以后，服务器的TCP协议才

```
1  0.0                bsdi.1024 > laptop.8888: SFP 36858825:36859125(300)
                                                win 8568 <mss 1460,nop,nop,ccnew 1>
2  0.020542 (0.0205)  laptop.8888 > bsdi.1024: S 76355292:76355292(0)
                                                ack 36858826 win 8712
                                                <mss 1460,nop,nop,cc 18,
                                                 nop,nop,ccecho 1>
3  0.021479 (0.0009)  bsdi.1024 > laptop.8888: F 301:301(0)
                                                ack 1 win 8712 <nop,nop,cc 1>
4  0.029471 (0.0080)  laptop.8888 > bsdi.1024: .
                                                ack 302 win 8412 <nop,nop,cc 18>
5  0.042086 (0.0126)  laptop.8888 > bsdi.1024: FP 1:401(400)
                                                ack 302 win 8712 <nop,nop,cc 18>
6  0.042969 (0.0009)  bsdi.1024 > laptop.8888: .
                                                ack 402 win 8312 <nop,nop,cc 1>
```

图3-1 T/TCP客户重启后向服务器发送一个事务

能把收到的300字节数据提交给当前的服务进程。

第3行显示的是三次握手过程的最后一个报文段：客户对服务器发出 SYN的确认。在这个报文段中客户将FIN重传，但不包括300字节数据。服务器收到该报文段后，立刻确认了收到的数据和FIN(第4行)。与一般的报文段不同的是，这个确认是即时发出的，没有被耽搁。这么做是为了防止客户第1行发出数据后超时而重传。

第5行显示的是服务器给出的应答以及服务器的 FIN，第6行中客户对服务器的 FIN和应答都做了确认。注意，第3、4、5和6行中都有CC选项，而CCnew和CCecho选项则分别只出现在第1和第2个报文段中。

从现在开始，我们不再专门地在 T/TCP报文段中标示NOP了，因为NOP不是必需的，而且会把图搞复杂。插入 NOP，使选项长度保持为 4字节整数倍的做法是出于对提高主机性能的考虑。

机敏的读者可能会注意到，客户端刚刚重新启动时，客户 TCP协议所用的初始序号(ISN)与卷1中习题18.1所讨论的一般模式不一样。而且，服务器的初始序号是个偶数，这在通常从伯克利演变而来的实现中是从来没有的。其原因在于这里的连接所使用的初始序号是随机选取的；而且每隔 500ms对内核的初始序号所加的增量也是随机的。这种改动有助于防护序号攻击，具体内容可见参考文献 [Bellovn 1989]。这种改动是1994年12月[Shimomura 1995]很有名的一次因特网侵入事件发生后，首先在BSD/OS 2.0，然后在4.4BSD-Lite2中加入的。

时间系列图

图3-2给出的是图3-1所描述的报文段交换过程的时序图。

图中，包含数据的第 1和第5这两个报文段用粗黑线标记。图的两侧还分别标注了客户和服务器收到报文段后各自发生的状态变迁。开始的时候，客户进程调用 `sendto`函数，并指定`MSG_EOF`标志，后进入SYN_SENT*状态。服务器收到并处理了第 3个报文段后发生了两次状态变迁。先是处理客户对服务器发出 SYN的确认后，连接的状态由 SYN_RCVD变迁到ESTABLISHED状态；紧接着处理客户发来的FIN又变迁到CLOSE_WAIT状态。当服务器向客户发出设置了`MSG_EOF`标志的应答后，即进入 LAST_ACK状态。注意，客户在第 3个报文段

中重传了FIN标志(回忆一下图2-7)。

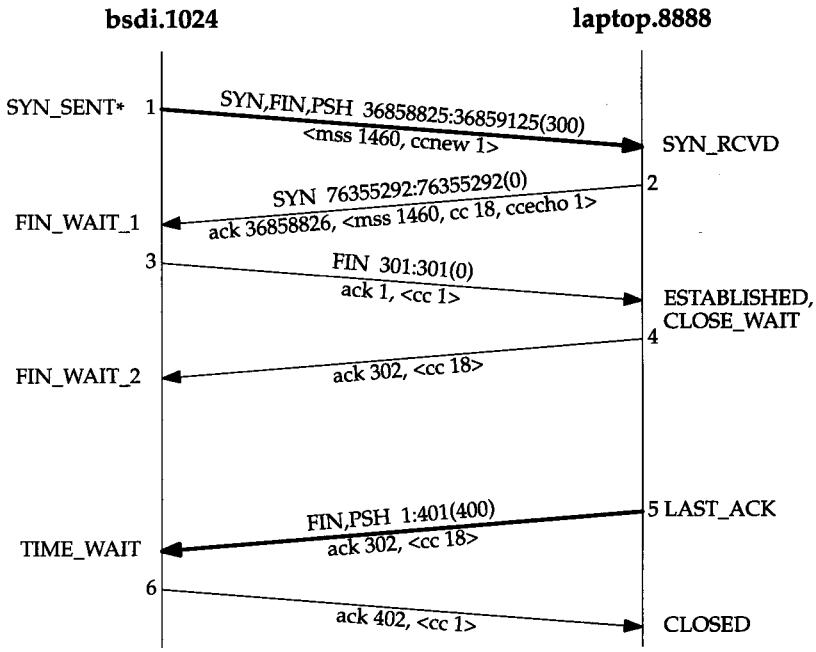

图3-2 图3-1中报文段交换过程的时间系列图

3.3 常规的T/TCP事务

下面我们还是在上面那对客户和服务器之间发起另一次事务。这一次客户在自己的单机高速缓存中取到该服务器的tao_ccsent值非0，于是就发出一个CC选项，其中下一个tcp_ccgen的值为2 (2表示这是客户端重新启动后TCP协议建立的第2个连接)。报文段交换的过程如图3-3所示。

```
1  0.0                  bsdi.1025 > laptop.8888: SFP 40203490:40203790(300)
                                                 win 8712 <mss 1460,cc 2>
2  0.026469 (0.0265)    laptop.8888 > bsdi.1025: SFP 79578838:79579238(400)
                                                 ack 40203792 win 8712
                                                 <mss 1460,cc 19,ccecho 2>
3  0.027573 (0.0011)    bsdi.1025 > laptop.8888: .
                                                 ack 402 win 8312 <cc 2>
```

图3-3 常规的T/TCP客户-服务器事务

这是一个常规的、仅包含3个报文段的最小规模T/TCP报文交换过程。图3-4显示了该次报文交换的时序图以及状态变迁过程。

客户发出包含有SYN标志、数据和FIN标志的报文段后进入SYN_SENT*状态。服务器收到该报文段，且TAO测试成功时，进入半同步的ESTABLISHED*状态。其中的数据经处理后交给服务器进程。接着处理完报文段的FIN标志后服务器进入CLOSE_WAIT*状态。由于还未发出SYN报文段，因而服务器一直都处于加星状态。当服务器发出应答并在其中设置MSG_EOF标志后，服务器端随即转入LAST_ACK*状态。如图2-7所示，这个状态的服务器发出的报文段中包含了SYN、FIN和ACK标志。

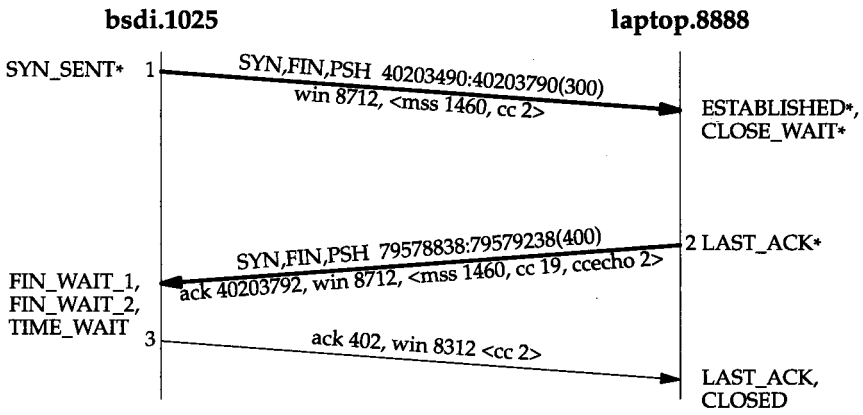

图3-4 图3-3中报文段交换的时间系列图

客户收到第2个报文段后，其中对SYN的确认使客户端的连接状态转入FIN_WAIT_1状态。接着客户处理报文段中对所发FIN的确认，并进入FIN_WAIT_2状态。服务器的应答则送到客户进程。然后，客户处理该报文段中服务器所发的FIN后进入TIME_WAIT状态。在这个最终的状态，客户发出对服务器所发FIN的确认。

服务器收到第3个报文段后，其中对服务器所发SYN的确认使服务器进入LAST_ACK状态，对服务器所发FIN的确认则使服务器进入CLOSED状态。

这个例子清晰地显示了在T/TCP事务过程中，收到一个报文段是怎样引起多次状态变迁的。它同时也显示了尚不处于ESTABLISHED状态的进程是如何接收数据的：客户进程半关闭（发出第1个报文段）了与服务器的连接，处于FIN_WAIT_1状态下，但仍然能接收数据（第2个报文段）。

3.4 服务器收到过时的重复SYN

如果服务器收到了一个看似过时的CC值该怎么办呢？我们让客户发出一个连接计数值CC为1的SYN报文段，这个值小于服务器刚刚从该客户收到的CC值(2，见图3-3)。事实上，这种情况也是可能发生的，比如：CC值等于1的这个报文段是客户和服务器之间此前某个连接的，它在网络上耽搁了一段时间，但还没有超过其报文段最大生存时间（发出后MSL秒），最终到达了服务器。

一个连接是由一对插口定义的，即包含客户端IP地址和端口号及服务器端IP地址和端口号的四元组。连接的新实例称为该连接的替身。

从图3-5我们可以看出，服务器收到一个CC值为1的SYN报文段后强迫执行三次握手操作，因为它无法判断该报文段是过时重复的还是新的。

由于激活了三次握手(这一点我们可以从服务器仅仅确认了客户的SYN而没有确认客户的数据来判断)，服务器的TCP协议在握手过程完全结束以后才会把300字节的数据提交给服务器进程。

本例中，第1个报文段就是一个过时的重复报文段(客户的TCP此时并不在等待对这个报文段中SYN的响应)，于是当第2个报文段中服务器发出的SYN/ACK到达时，客户端TCP协议的响应是要求重新建立连接RST(第3个报文段)。这样做也是理所应当的。服务器的TCP协议收到这个RST后就扔掉那300字节的数据，而且`accept`函数也不返回到服务器进程。

```
1  0.0                  bsdi.1027 > laptop.8888: SFP 80000000:80000300(300)
                                                 win 4096 <mss 1460,cc 1>
2  0.018391 (0.0184)    laptop.8888 > bsdi.1027: S 132492350:132492350(0)
                                                 ack 80000001 win 8712
                                                 <mss 1460,cc 21,ccecho 1>
3  0.019266 (0.0009)    bsdi.1027 > laptop.8888: R 80000001:80000001(0) win 0
```

图3-5 T/TCP服务器收到过时的重复SYN报文段

第1个报文段是由一个特殊的测试程序生成的。我们无法让客户的 T/TCP协议自己生成这样的报文段，而只能让它以过时的重复报文段出现。作者曾试着把内核的tcp_ccgen变量值改为 1，但是，正如我们将在图 12-3中看到的，当内核的tcp_ccgen小于它最近一次发给对端的CC值时，TCP协议自动地发出一个CCnew选项而不是发出一个CC选项。

图3-6所示的就是这对客户-服务器之间的下一次、也是常规的一次 T/TCP事务。正如我们所预期的，这是一个包含3个报文段的交换过程。

```
1  0.0                  bsdi.1026 > laptop.8888: SFP 101619844:101620144(300)
                                                 win 8712 <mss 1460,cc 3>
2  0.028214 (0.0282)    laptop.8888 > bsdi.1026: SFP 140211128:140211528(400)
                                                 ack 101620146 win 8712
                                                 <mss 1460,cc 22,ccecho 3>
3  0.029330 (0.0011)    bsdi.1026 > laptop.8888: .
                                                 ack 402 win 8312 <cc 3>
```

图3-6 常规的T/TCP客户-服务器事务

服务器希望这个客户发来的CC值大于2，因此收到CC值为3的SYN后TAO测试成功。

3.5 服务器重启动

现在我们将服务器重新启动，并让客户在服务器刚启动，即服务器监听进程刚开始运行的时候就立即发送一个事务请求。图 3-7为报文段交换的情况。

```
1  0.0                  bsdi.1027 > laptop.8888: SFP 146513089:146513389(300)
                                                 win 8712 <mss 1460,cc 4>
2  0.025420 (0.0254)    arp who-has bsdi tell laptop
3  0.025872 (0.0005)    arp reply bsdi is-at 0:20:af:9c:ee:95
4  0.033731 (0.0079)    laptop.8888 > bsdi.1027: S 27338882:27338882(0)
                                                 ack 146513090 win 8712
                                                 <mss 1460,cc 1,ccecho 4>
5  0.034697 (0.0010)    bsdi.1027 > laptop.8888: F 301:301(0)
                                                 ack 1 win 8712 <cc 4>
6  0.044284 (0.0096)    laptop.8888 > bsdi.1027: .
                                                 ack 302 win 8412 <cc 1>
7  0.066749 (0.0225)    laptop.8888 > bsdi.1027: FP 1:401(400)
                                                 ack 302 win 8712 <cc 1>
8  0.067613 (0.0009)    bsdi.1027 > laptop.8888: .
                                                 ack 402 win 8312 <cc 4>
```

图3-7 服务器刚刚重启动后，T/TCP的交换分组情况

由于客户并不知道服务器已经重新启动了，因而它发出的仍是一个常规的 T/TCP请求，其

中CC值为4(见第1行)。服务器重新启动使其ARP缓存中的客户硬件地址丢失，于是服务器发出一个ARP请求，客户给出应答。服务器强迫执行三次握手操作 (见第4行)，因为它不记得上次从该客户收到的连接计数值CC。

与我们在图3-1中看到的类似，客户发出一个带有 FIN标志的确认报文段完成三次握手过程，300字节的数据则不重传。只有当客户端的重传定时器超时时客户才会重传数据，我们将在图3-11中看到这种情况。收到这第 3个报文段后，服务器立即对数据和 FIN发出确认。服务器发出应答(见第7行)，第8行则是客户给出的确认。

看过图3-7那样的报文交换过程后，我们来看看这对客户和服务器之间接下来继续通信时的一个最小T/TCP事务，如图3-8所示。

```
1  0.0                  bsdi.1028 > laptop.8888: SFP 152213061:152213361(300)
                                                 win 8712 <mss 1460,cc 5>
2  0.034851 (0.0349)    laptop.8888 > bsdi.1028: SFP 32869470:32869870(400)
                                                 ack 152213363 win 8712
                                                 <mss 1460,cc 2,ccecho 5>
3  0.035955 (0.0011)    bsdi.1028 > laptop.8888: .
                                                 ack 402 win 8312 <cc 5>
```

图3-8 常规的T/TCP客户-服务器事务

3.6 请求或应答超出报文段最大长度MSS

到目前为止，在我们所举的所有例子中，无论是客户的请求报文段还是服务器的应答报文段都没有超过报文段最大长度(MSS)。如果客户要发送超出报文段最大长度的数据，而且也确信对等端支持T/TCP协议，那么它就会发出多个报文段。由于对等端的报文段最大长度存储在TAO高速缓存中(图2-5的tao_mssopt)，因而客户的TCP协议能够知道服务器的报文段最大长度，但无法知道服务器的接收窗口宽度 (卷1的18.4节和20.4节分别讨论了报文段最大长度和窗口宽度)。对一个特定的主机来说，报文段最大长度一般是个固定值，而此接收窗口的宽度却会随应用程序改变其插口接收缓存的大小而相应地变化。而且，即使对等端告知了一个较大的接收窗口(比如说，32 768字节)，但如果报文段最大长度为512字节，那么很可能会有一些中间的路由器无法处理客户一下子发给服务器的前 64个报文段(也即，TCP协议的慢启动是不能跳过的)。T/TCP协议加了两条限制来解决这些问题：

1) T/TCP协议将刚开始时的发送窗口宽度设定为 4 096字节。在Net/3中，这就是变量snd_wnd的值。该变量控制着TCP输出流可以发出多少数据。当对等端带有窗口通告的第1个报文段到达后，窗口宽度的初始值4 096将被改变为所需值。

2) 只有当对等端不在本地时， T/TCP协议才使用慢启动方式开始通信。 TCP协议将snd_cwnd变量设置为1个报文段时就是慢启动。图10-14给出了本地/非本地测试程序，以内核的in_localaddr函数为基础。如果(a)与本机拥有相同的网络号和子网号，或者(b)虽然网络号相同子网号不同，但内核的 subnetsarelocal变量值非0，这样的对等主机就是本地主机。

   Net/3总是用慢启动方式开始每一条连接 (卷2第721页)，但这样就使客户在启动事务时无法连续发出多个报文段。折衷的结果是，允许向本地的对等主机发送多个报文段，但最多4 096 字节。

每次调用TCP协议的输出模块，它总是选择snd_wnd和snd_cwnd中较小的一个作为其可发送数据量的上限值。前者的初始值为TCP滑动窗口通告中的最大值，我们假设为65 535字节(如果使用窗口大小选项，那么这个最大值可以为 65 535×2^{14}，大约为1吉字节)。如果对等主机在本地，那么snd_wnd和snd_cwnd的初始值分别为4 096和65 535。TCP协议在连接刚开始时还未收到对方的窗口通告前，可以发出至多 4 096字节的数据。如果对方通告的窗口宽度为32 768字节，那么 TCP协议可以持续发送数据直到对等主机的接收窗口满为止 (因为32 768和65 535的最小值是32 768)。这样，TCP协议既可以避开慢启动过程，发送数据量又可以受限于对方通告的窗口宽度。

如果对等主机不在本地，那么snd_wnd的初始值仍为4 096，但snd_cwnd的初始值则为1个报文段(假设保存的对等主机报文段最大长度 MSS为512)。TCP协议在连接一开始的时候只能发出一个报文段，当收到对等主机的窗口通告后，每收到一个确认， snd_wnd的值就加1。这时慢启动机制在起作用，可以发出的数据量受限于拥塞窗口，直至拥塞窗口宽度超过了对等主机通告的接收窗口。

作为一个例子，我们对第1章中的T/TCP客户和服务器程序加以修改，使请求和应答中的数据量分别为3 300和3 400字节。图3-9给出了分组交换过程。

这个例子要显示T/TCP交换的多个报文段的序列号，恰好暴露了 Tcpdump的一个打印bug 。第6、第8和第10个报文段的确认号应当打印3 302而不是1。

```
1   0.0                 bsdi.1057 > laptop.8888: S 3846892142:3846893590(1448)
                                                 win 8712 <mss 1460,cc 7>
2   0.001556 (0.0016)   bsdi.1057 > laptop.8888: .  3846893591:3846895043(1452)
                                                 win 8712 <cc 7>
3   0.002672 (0.0011)   bsdi.1057 > laptop.8888: FP 3846895043:3846895443(400)
                                                 win 8712 <cc 7>

4   0.138283 (0.1356)   laptop.8888 > bsdi.1057: S 3786170031:3786170031(0)
                                                 ack 3846895444 win 8712
                                                 <mss 1460,cc 6,ccecho 7>
5   0.139273 (0.0010)   bsdi.1057 > laptop.8888: .
                                                 ack 1 win 8712 <cc 7>

6   0.179615 (0.0403)   laptop.8888 > bsdi.1057: . 1:1453(1452)
                                                 ack 1 win 8712 <cc 6>
7   0.180558 (0.0009)   bsdi.1057 > laptop.8888: .
                                                 ack 1453 win 7260 <cc 7>

8   0.209621 (0.0291)   laptop.8888 > bsdi.1057: . 1453:2905(1452)
                                                 ack 1 win 8712 <cc 6>
9   0.210565 (0.0009)   bsdi.1057 > laptop.8888: .
                                                 ack 2905 win 7260 <cc 7>

10  0.223822 (0.0133)   laptop.8888 > bsdi.1057: FP 2905:3401(496)
                                                 ack 1 win 8712 <cc 6>
11  0.224719 (0.0009)   bsdi.1057 > laptop.8888: .
                                                 ack 3402 win 8216 <cc 7>
```

图3-9 3 300字节的客户请求和3 400字节的服务器应答

由于客户知道服务器支持T/TCP协议，客户可以立即发出4 096字节。在前2.6 ms的时间里，客户发出了第1、第2和第3个报文段。第1个报文段携带了SYN标志、1 448字节数据和12字节TCP选项(报文段最大长度MSS和连接计数CC)。第2个报文段没有带标志，只有1 452字节数据

和8字节TCP选项。第3个报文段携带FIN和PSH标志、8字节TCP选项以及剩余的400字节数据。第2个报文段是唯一一个没有设置任何 TCP标志(共有6个标志)，甚至不带ACK标志的报文段。通常情况下，ACK标志总是携带的，除非是客户端主动打开，此时的报文段带有 SYN标志(在收到服务器的报文段之前，客户是绝不能发出任何确认的)。

第4个报文段是服务器的SYN报文段，它同时也对客户所发来的所有内容做出了确认，包括SYN标志、数据和FIN标志。在第5个报文段中，客户立即确认了服务器的 SYN报文段。

第6个报文段晚了40 ms才到达客户端，它携带了服务器应答的第 1段数据。客户立即对此给出了确认。第8~11报文段继续同样的过程。服务器的最后一个报文段 (第10行)带有FIN标志，客户发出的最后一个ACK报文段对这最后的数据以及 FIN标志做了确认。

一个问题是，为什么客户对 3个服务器应答报文中的前两个立即给出了确认，是因为它们在很短的时间(44ms)内就到达了吗？答案在 TCP_REASS宏(卷2第726页)中，客户每收到一个带有数据的报文段就要调用该宏。由于连接的客户端处理完第 4个报文段后就进入了FIN_WAIT_2状态，于是在TCP_REASS宏中对连接是否处于ESTABLISHED状态的测试失败，从而使客户端立即发出 ACK而不是延迟一会儿再发。这一“特性”并非 T/TCP协议所独有，在Net/3的程序中，如果任何一端半关闭了TCP连接而进入FIN_WAIT_1或FIN_WAIT_2状态后，都会出现这种情形。从此以后，来自对等主机的每一个数据报文段都立即给予确认。TCP_REASS宏中对是否已进入ESTABLISHED状态的测试使协议无法在三次握手完成之前把数据提交给应用程序。实际上，当连接状态大于 ESTABLISHED时，没有必要立刻确认按序收到的每个报文段(即：应当修改这种测试)。

TCP_NOPUSH插口选项

运行该示例程序之前需要对客户程序再做一些修改。下面这段程序打开了 TCP_NOPUSH插口选项(T/TCP协议新引入的选项)：

```
int  n;

n = 1;
if (setsockopt(sockfd, IPPROTO_TCP, TCP_NOPUSH, (char *) &n, sizeof(n)) < 0)
    err_sys("TCP_NOPUSH error");
```

这段程序在图1-10中调用socket函数之后执行。设置该选项的目的是告诉 TCP协议不要仅仅为了清空发送缓存而发送报文段。

如果要了解设置该插口选项的原因，我们必须跟踪用户进程调用 sendto 函数发送3 300字节数据并设置MSG_EOF标志的请求后内核所执行的动作。

1) 内核最终要调用sosend函数(卷2的第16.7节)来处理输出请求。它把前 2 048字节数据放入一个mbuf簇中，并向TCP协议发出一个PRU_SEND请求。
2) 于是内核调用tcp_output函数(图12-4)。由于可以发送一个满长度 (full-sized)的报文段，因此发出mbuf簇中的前1 448字节数据，并设置 SYN标志(该报文段中包含12字节的TCP选项)。
3) 由于mbuf簇中还剩下600字节数据，于是再次循环调用 tcp_output函数。我们也许会认为Nagle算法将不会使另一个报文段发出去，但是注意卷 2第681页可以看到，第1次执行tcp_output函数后，idle变量的值为1。当程序发出长为1 448字节的第1个报文段后进入again分支时，idle变量没有重新计算。因此，程序在图 9-3所示程序

段(“发送方的糊涂窗口避免(sender silly window avoidance)”)中结束。如果idle变量为真，待发送的数据将把插口发送缓存清空，因此，决定是否发送报文段的是TF_NOPUSH标志的当前值。

在T/TCP协议引入这个标志以前，如果某个报文段要清空插口的发送缓存，并且Nagle算法允许，这段程序就总是会发送一个不满长的报文段。但是如果应用程序设置了TF_NOPUSH标志(利用新的TF_NOPUSH插口选项)，这时TCP协议就不会仅仅为清空发送缓存而强迫发出数据。TCP协议将允许现有的数据与后面写操作补充来的数据结合起来，以期发出较大的报文段。

4) 如果应用程序设置了TCP_NOPUSH标志，那就不会发送报文段，tcp_output函数返回，程序执行的控制权又回到sosend函数。

   如果应用程序没有设置TCP_NOPUSH标志，那么协议就发出那个600字节的报文段，并在其中设置PSH标志。

5) sosend函数把剩余的1 252字节数据放入一个mbuf簇，并发出一个PRU_SEND_EOF请求(图5-2)，该请求再次结束tcp_output函数的调用。然而在这次调用之前，已经调用过tcp_usrclosed函数(图12-4)，使连接的状态由SYN_SENT变迁至SYN_SENT*(图12-5)。设置了TF_NOPUSH标志后，当前插口发送缓存中共有1 852字节的数据，于是协议又发出一个满长度的报文段，该报文段包含1 452字节数据和8字节TCP选项(如图3-9所示)。发出该报文段的原因就是因为它是满长度的(亦即：Nagle算法不起作用)。尽管SYN_SENT*状态的标志中包含有FIN标志(图2-7)，但由于发送缓存中还有额外的数据，因此FIN标志被关掉了(卷2第683页)。

6) 程序又执行了一次循环从而再次调用tcp_output函数发送缓存中剩余的400字节数据。然而这一回FIN标志是打开的，因为发送缓存已经空了。尽管图9-3中的Nagle算法不允许发出数据，但由于设置了FIN标志，只有400字节的报文段还是发出去了(卷2第688页)。

本例中，给插口设置了TCP_NOPUSH属性之后，在报文段最大长度MSS为1 460字节的以太网上发出一个3 300字节的请求就引发出3个报文段，长度分别为1 448、1 452和400字节。如果不设置该选项，那么仍然会有3个报文段，但其长度分别为1 448、600和1 252字节。但如果请求的长度为3 600字节，则设置了TCP_NOPUSH选项时产生3个报文段(长度分别为1 448、1 452和700字节)，而不设置该选项就会产生4个报文段(长度分别为1 448、600、1 452和100字节)。

总之，当客户程序仅调用一次sendto函数发出请求时，通常应该设置TCP_NOPUSH插口选项。这样，当请求长度超过报文段最大长度MSS时，协议就会尽可能发出满长度的报文段。这样可以减少报文段的数量，减少的程度取决于每次发送的数据量。

3.7 向后兼容性

我们还需要研究一下如果客户用T/TCP协议给一台不支持T/TCP协议的主机发送数据会发生什么样的情况。

图3-10显示的就是主机bsdi上的T/TCP客户程序向主机svr4(一个运行System V Release 4的主机，不支持T/TCP)上的TCP服务器发起事务时，它们二者之间分组交换的情况。

```
1  0.0                  bsdi.1031 > svr4.8888: SFP 2672114321:2672114621(300)
                                                win 8568 <mss 1460,ccnew 10>
2  0.006265 (0.0063)    svr4.8888 > bsdi.1031: S 879930881:879930881(0)
                                                ack 2672114322 win 4096 <mss 1024>
3  0.007108 (0.0008)    bsdi.1031 > svr4.8888: F 301:301(0)
                                                ack 1 win 9216
4  0.012279 (0.0052)    svr4.8888 > bsdi.1031: .
                                                ack 302 win 3796
5  0.071683 (0.0594)    svr4.8888 > bsdi.1031: P 1:401(400)
                                                ack 302 win 4096
6  0.072451 (0.0008)    bsdi.1031 > svr4.8888: .
                                                ack 401 win 8816
7  0.078373 (0.0059)    svr4.8888 > bsdi.1031: F 401:401(0)
                                                ack 302 win 4096
8  0.079642 (0.0013)    bsdi.1031 > svr4.8888: .
                                                ack 402 win 9216
```

图3-10 T/TCP客户程序向TCP服务器发起事务

客户端的TCP程序发出的第1个报文段中包含有SYN、FIN和PSH标志，还包含了300字节数据。由于客户端的TCP协议在其TAO高速缓存中还没有该服务器主机svr4的连接计数CC值，因而它发出的报文段中带上了CCnew选项。图中第2行就是服务器对该报文段的响应，这是标准三次握手过程中的第2个报文段；而客户端在第3行中对该响应做出了确认。注意：第3行中没有重传数据。

服务器端收到第3行的报文段后，立即确认了客户一开始发过来的300字节数据和FIN标志(如卷2第791页所示，对FIN的确认从不推迟)。服务器端TCP将上述数据保存在队列中，直至三次握手过程结束才将其交给服务进程。

第5行显示的是服务器给出的响应(400字节数据)，客户端在第6行中立刻对此做出了确认。第7行显示的是服务器发出的FIN报文段，客户端同样也迅速地做出了确认。注意，服务器进程无法把第5行的数据和第7行的FIN结合在一起发送。

如果我们在同样还是这一对客户和服务器之间再发起一次事务，则报文段交换的顺序与上一次完全相同。由于在图3-10中服务器端并没有发回一个CCecho选项，因此客户端仍然无法向svr4主机发出带有CC选项的报文段，从而客户端发出的第1个报文段(即初始化报文段)仍然带有CCnew选项，其值为11。支持T/TCP的客户端总是发出CCnew选项的原因是，对不支持T/TCP的服务器，它从来不会更新在其单机高速缓存中的相关表项，因而tao_ccsent值总是0(未定义)。

在下面的例子(图3-11)中，服务器主机运行Solaris 2.4，这也是一个基于SVR4(与图3-10中的服务器一样)的系统，但二者的TCP/IP协议栈实现却完全不同。

第1~3行与图3-10中的相同：带有SYN、FIN、PSH标志和300字节数据的报文段，接着是服务器的SYN/ACK报文段，然后是客户的ACK报文段。这是一次正常的三次握手过程。同样，由于不知道该服务器的连接计数CC值，客户端TCP协议发出的是一个带有CCnew选项的报文段。

Solaris主机发出的每个报文段中携带的“不分段”标志(DF)，用于路径最大传输单元发现(RFC 1191 [Mogul and Deering 1990])。

```
1  0.0                  bsdi.1033 > sun.8888: SFP 2693814107:2693814407(300)
                                                win 8712 <mss 1460,ccnew 12>
2  0.002808 (0.0028)    sun.8888 > bsdi.1033: S 3179040768:3179040768(0)
                                                ack 2693814108 win 8760
                                                <mss 1460> (DF)
3  0.003679 (0.0009)    bsdi.1033 > sun.8888: F 301:301(0)
                                                ack 1 win 8760
4  1.287379 (1.2837)    bsdi.1033 > sun.8888: FP 1:301(300)
                                                ack 1 win 8760
5  1.289048 (0.0017)    sun.8888 > bsdi.1033: .
                                                ack 302 win 8760 (DF)
6  1.291323 (0.0023)    sun.8888 > bsdi.1033: P 1:401(400)
                                                ack 302 win 8760 (DF)
7  1.292101 (0.0008)    bsdi.1033 > sun.8888: .
                                                ack 401 win 8360
8  1.292367 (0.0003)    sun.8888 > bsdi.1033: F 401:401(0)
                                                ack 302 win 8760 (DF)
9  1.293151 (0.0008)    bsdi.1033 > sun.8888: .
                                                ack 402 win 8360
```

图3-11 T/TCP客户向Solaris 2.4上的TCP服务器发送事务请求

不幸的是，我们遇到了在 Solaris的TCP/IP实现中的一个 bug。因为这个 bug，服务器端TCP把第1行中的数据部分扔掉了 (第2个报文段中没有对该数据做确认)，造成客户端的TCP超时，并在第4行重传了数据，同时也重传了 FIN。接着，服务器端确认了客户端发来的数据和FIN(第5行)，然后服务器端在第 6行发出应答。客户端在第 7行对应答给出确认，紧接着是服务器发出FIN报文段(第8行)，最后是客户端的确认(第9行)。

RFC 793 [Postel 1981b]的第30页中指出：“ 尽管这些例子并不证明采用附带数据的报文段也能实现连接同步，但这样处理也完全是合法的，接收端的 TCP只有在搞清楚了数据是正确的以后才能将数据交付给用户 (即，接收端对数据进行缓存，等到连接状态进入 ESTABLISHED以后才能交付给用户) ”。该 RFC的第 66页还说，在LISTEN状态处理接收到的 SYN时，“ 任何其他控制信息和正文数据都要先放入队列待以后处理 ”。

有一个评论者声称，把上述现象叫做“ bug ”是不对的，因为 RFC中并没有强制要求服务器在处理 SYN的同时接受其中附带的数据。声明中还说， Solaris的实现是正确的，因为还没有向客户端通告接收窗口，这时服务器完全可以丢弃已到达的数据，因为这些数据都落在窗口之外。不管你如何评价这个特点 (作者仍然称它们为bug，SUN公司也已经为这个问题分配了一个 Bug ID，即1 222 490，因此也将会在今后的版本中进行修正)，处理这样的情况还要符合健壮性原则，该原则在 RFC 791 [Postel 1981a]中首次提出：“ 你有自由去决定接受什么，但你发送什么却必须遵守规定。”

3.8 小结

我们可以对本章中的例子做下面这样的总结：

1) 如果客户端丢失了服务器的状态信息 (例如，客户端重新启动)，那么客户端在主动打开

时将发出CCnew选项，从而强迫执行三次握手过程。

2) 如果服务器丢失了客户端的状态信息，或者服务器收到的 SYN报文段中的CC值小于期望的值，那么服务器返回给客户的响应将只是一个 SYN/ACK报文段，从而强迫执行三次握手过程。在这种情况下，直到三次握手过程完全结束以后，服务器的 TCP才会把客户在SYN报文段中附带的数据交给上层的服务器进程。

3) 如果服务器想在连接中使用 T/TCP协议，那么它总是用 CCecho选项对客户的 CC或CCnew选项作出应答。

4) 如果客户端和服务器端彼此都掌握有对方的状态信息，那么整个事务过程所收发的报文段个数将达到最少：3个(假设请求和响应的长度都小于或等于报文段最大长度 MSS)。此时收发的分组数最少，时延也最小：为 RTT + SPT。

以上这些例子同时也说明了 T/TCP协议中多个状态的变迁是如何发生的，以及如何使用那些新扩充的(加星的)状态。

如果客户端向一个不支持 T/TCP协议的主机发送带有 SYN、数据和FIN的报文段，那么采用伯克利网络代码的系统 (包括SVR4，但不包括Solaris)能够正确地将数据存储在队列中，直至三次握手过程完成。然而，其他的一些网络代码也有可能错误地把 SYN报文段中的数据扔掉，造成客户端超时，并重传数据。