第11章　T/TCP实现：TCP输入

11.1　概述

T/TCP对TCP所做的大多数修改是在tcp_input函数中。在整个函数的前前后后都出现的修改是tcp_dooptions(10.9节)中新增的变量和返回值。我们不打算给出受这一变化而影响的每一块代码。

图11-1是卷2中图28-1的重写，T/TCP所做的修改用黑体字表示。

我们在说明tcp_input函数所做的修改时，按照各部分在整个函数中出现的顺序来分别介绍。

```
void
tcp_input()
{
    checksum TCP header and data;
    skip over IP/TCP headers in mbuf;
findpcb:
    locate PCB for segment;
    if (not found)
        goto dropwithreset;
    reset idle time to 0 and keepalive timer to 2 hours;
    process options if not LISTEN state;
    if (packet matched by header prediction) {
        completely process received segment;
        return;
    }

    switch (tp->t_state) {
    case TCPS_LISTEN:
        if SYN flag set, accept new connection request;
        perform TAO test;
        goto trimthenstep6;

    case TCPS_SYN_SENT:
        check CCecho option;
        if ACK of our SYN, connection completed;
trimthenstep6:
        trim any data not within window;
        if (ACK flag set)
            goto processack;
        goto step6;

    case TCPS_LAST_ACK:
    case TCPS_CLOSING:
    case TCPS_TIME_WAIT:
        check for new SYN as implied ACK of previous incarnation;
    }
```

图11-1　TCP输入处理步骤小结：T/TCP所做的修改用黑体表示

```
    process RFC 1323 timestamp;
    check CC option;
    check if some data bytes are within the receive window;
    trim data segment to fit within window;
    if (RST flag set) {
        process depending on state;
        goto drop;
    }
     if (ACK flag off)
        if (SYN_RCVD || half-synchronized)
            goto step6;
        else
            goto drop;
    if (ACK flag set) {
        if (SYN_RCVD state)
            passive open or simultaneous open complete;
        if (duplicate ACK)
            fast recovery algorithm;
processack:
        update RTT estimators if segment timed;
        if (no data was ACKed)
            goto step6;
        open congestion window;
        remove ACKed data from send buffer;
        change state if in FIN_WAIT_1, CLOSING, or LAST_ACK state;
    }
step6:
    update window information;
    process URG flag;
dodata:
    process data in segment, add to reassembly queue;
    if (FIN flag is set)
        process depending on state;
    if (SO_DEBUG socket option)
        tcp_trace(TA_INPUT);
    if (need output || ACK now)
        tcp_output();
    return;
dropafterack:
    tcp_output() to generate ACK;
    return;
dropwithreset:
    tcp_respond() to generate RST;
    return;
drop:
    if (SO_DEBUG socket option)
        tcp_trace(TA_DROP);
    return;
}
```

图11-1 (续)

11.2 预处理

定义了三个新的自动变量，其中之一是tcpopt结构，在tcp_dooptions中使用。下面的几行语句用于替换卷2第739页的第190行。

```
struct tcpopt to;               /* options in this segment */
struct rmxp_tao *taop;          /* pointer to our TAO cache entry */
struct rmxp_tao tao_noncached;  /* in case there's no cached entry */

bzero((char *)&to, sizeof(to));
tcpstat.tcps_rcvtotal++;
```

将tcpopt结构初始化为0是非常重要的：这样就会将to_cc字段(接收到的CC值)设置为0，表明它未定义。

在Net/3中，唯一回到标号findpcb的分支是在一个连接处于TIME_WAIT状态时又收到一个新的SYN报文段(卷2第765~766页)。因为下面的这两行代码有问题，因而该分支存在一个缺陷

```
m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
m->m_len  -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
```

这两行代码在findpcb后出现了两次，在goto后又执行了一次(这两行代码在卷2第751页出现了一次，在第752页又出现一次；这两处中只能有一处执行，决定于该报文段是否与首部所指示的相一致)。这在T/TCP之前并不会带来问题，因为SYN不携带数据，上述这个缺陷只在当一个连接处于TIME_WAIT状态又收到一个携带数据的新SYN时才会表现出来。然而在T/TCP中，还会有第2个回到findpcb的分支(在后面的图11-11中会说明，这个分支处理图4-7所示的隐式ACK)，并且要处理的SYN很可能携带数据。这样，在findpcb之前的上述这两行代码就必须删去，如图11-2所示。

tcp_input.c

```
274     /*
275      * Skip over TCP, IP headers, and TCP options in mbuf.
276      * optp & ti still point into TCP header, but that's OK.
277      */
278     m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
279     m->m_len -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);

280     /*
281      * Locate pcb for segment.
282      */
283   findpcb:
```

tcp_input.c

图11-2 tcp_input：在findpcb前修改mbuf指针和长度

这样就从卷2的第751页和第752页中删除上述的两行。

下一个修改位于卷2第744页的第327行，这段代码在一个新报文段到达监听插口时创建一个新插口。在t_state设置为TCPS_LISTEN以后，TF_NOPUSH和TF_NOOPT这两个标志必须从监听插口复制到新的插口：

```
tp->t_flags |= tp0->t_flags & (TF_NOPUSH|TF_NOOPT);
```

其中tp0是指向监听插口tcpcb的自动变量。

卷2第745页的第344~345行中对tcp_dooptions的调用要改为新的调用序列(10.9节)：

```
if (optp && tp->t_state != TCPS_LISTEN)
    tcp_dooptions(tp, optp, optlen, ti, &to);
```

11.3 首部预测

是否应用首部预测(卷2第748页)的第一项测试是检查隐藏状态标志是否关闭。如果这些标志中有任何一个处于打开状态，则需要用 tcp_input中的慢通道处理将其关闭。图 11-13给出了新的测试过程。

tcp_input.c

```
398     if (tp->t_state == TCPS_ESTABLISHED &&
399       (tiflags & (TH_SYN | TH_FIN | TH_RST | TH_URG | TH_ACK)) == TH_ACK &&
400         ((tp->t_flags & (TF_SENDSYN | TF_SENDFIN)) == 0) &&
401         ((to.to_flag & TOF_TS) == 0 ||
402          TSTMP_GEQ(to.to_tsval, tp->ts_recent)) &&
403     /*
404      * Using the CC option is compulsory if once started:
405      *   the segment is OK if no T/TCP was negotiated or
406      *   if the segment has a CC option equal to CCrecv
407      */
408     ((tp->t_flags & (TF_REQ_CC | TF_RCVD_CC)) != (TF_REQ_CC | TF_RCVD_CC) ||
409      (to.to_flag & TOF_CC) != 0 && to.to_cc == tp->cc_recv) &&
410         ti->ti_seq == tp->rcv_nxt &&
411         tiwin && tiwin == tp->snd_wnd &&
412         tp->snd_nxt == tp->snd_max) {
413         /*
414          * If last ACK falls within this segment's sequence numbers,
415          * record the timestamp.
416          * NOTE that the test is modified according to the latest
417          * proposal of the tcplw@cray.com list (Braden 1993/04/26).
418          */
419         if ((to.to_flag & TOF_TS) != 0 &&
420             SEQ_LEQ(ti->ti_seq, tp->last_ack_sent)) {
421             tp->ts_recent_age = tcp_now;
422             tp->ts_recent = to.to_tsval;
423         }
```

tcp_input.c

图11-3 tcp_inpu t：是否可以应用首部预测

1. 验证隐藏状态标志关闭

400 这里的第一项修改就是验证TF_SENDSYN和TF_SENDFIN标志是否同时处于关闭状态。

2. 检查时间戳选项(如果存在)

401-402 第2项修改与修改后的tcp_dooptions函数有关的是：不再测试ts_present，而是测试to_flag中的TOF_TS标志位，并且如果时间戳存在，它的值是在 to_tsval而不是ts_val中。

3. 如果使用T/TCP，就验证CC

403-409 最后，如果没有完成T/TCP协商(我们要求有CC选项，但另一个端没有发送，或者我们根本就没有要求)，则if测试继续进行。如果使用了 T/TCP，则接收到的报文段必须包含一个CC选项，且CC值必须等于cc_recv值，这样才继续进行if测试。

我们希望在简短的T/TCP事务中不要频繁使用首部预测。这是因为在一次最小的 T/TCP报文段交换中，其最初的两个报文段携带有控制标志 (SYN和FIN)，这会使图11-3中在第二项测

试失败。这些T/TCP报文段用tcp_input的慢通道进行处理。但是，在支持T/TCP的两个主机之间的长连接(例如成批的数据传送)可以使用CC选项，并从首部预测中获益。

4. 用接收到的时间戳更新ts_recent

413-423 ts_recent是否因该更新的测试与卷2第748页的第371~372行有所不同。图11-3中采用新测试代码的原因在卷2第694~695页有详细叙述。

11.4 被动打开的启动

我们现在替换掉卷2第755页的全部代码：处于LISTEN状态的插口处理所收到的SYN的代码的最后一部分。这是当服务器从一个客户端接收到一个SYN时被动打开的启动(我们不想重复卷2第753~754页中在该状态下进行初始化的代码)。图11-4给出了这段代码的第一部分。

tcp_input.c

```
545             tp->t_template = tcp_template(tp);
546             if (tp->t_template == 0) {
547                 tp = tcp_drop(tp, ENOBUFS);
548                 dropsocket = 0; /* socket is already gone */
549                 goto drop;
550             }
551             if ((taop = tcp_gettaocache(inp)) == NULL) {
552                 taop = &tao_noncached;
553                 bzero(taop, sizeof(*taop));
554             }
555             if (optp)
556                 tcp_dooptions(tp, optp, optlen, ti, &to);
557             if (iss)
558                 tp->iss = iss;
559             else
560                 tp->iss = tcp_iss;
561             tcp_iss += TCP_ISSINCR / 4;
562             tp->irs = ti->ti_seq;
563             tcp_sendseqinit(tp);
564             tcp_rcvseqinit(tp);
565             /*
566              * Initialization of the tcpcb for transaction:
567              *   set SND.WND = SEG.WND,
568              *   initialize CCsend and CCrecv.
569              */
570             tp->snd_wnd = tiwin;     /* initial send-window */
571             tp->cc_send = CC_INC(tcp_ccgen);
572             tp->cc_recv = to.to_cc;
```

tcp_input.c

图11-4 tcp_input：取TAO记录项，初始化事务的控制块

1. 取客户端的TAO记录项

551-554 tcp_gettaocache查找该客户端的TAO记录项。如果没有找到，在全部设置为0后使用自动变量。

2. 处理选项和初始化序号

555-564 tcp_dooptions处理所有的选项(由于连接处于LISTEN状态，这个函数在此之前是不会调用的)。初始化发送序号(iss)和初始接收序号(irs)。控制块中的所有序号变量都由tcp_sendseqinit和tcp_rcvseqinit进行初始化。

3. 更新发送窗

565-570 tiwin是在接收到的SYN中由客户端通告的窗口(卷2第742~743页)。它是新插口的初始化发送窗口。通常，发送窗口要一直等到收到了一个带有 ACK的报文段才会更新(卷2第785页)。但T/TCP要利用所收到的 SYN报文段中的发送窗口值，即使这个报文段不包含ACK。这个窗口影响到服务器端给出应答时可以立即发送给客户端的数据有多少 (T/TCP交换中最小三报文段的第2个报文段)。

4. 设置cc_send和cc_recv

571-572 cc_send设置为tcp_ccgen的值，并且如果CC选项存在，则cc_recv设置为CC值。如果CC选项不存在，因为在函数的一开始已经将 to初始化为0，所以cc_recv也是0(未定义)。

5. 执行TAO测试

573-587 仅仅在报文段中包含有CC选项时才进行TAO测试。如果接收到的CC值非0且大于该客户端的缓存值(tao_cc)，则TAO测试成功。

6. TAO测试成功；更新客户端的TAO缓存

588-594 对这个客户端的缓存值进行更新，并且将连接状态设置为 ESTABLISHED*(隐藏状态变量在稍后的几行中设置，使之成为半同步加星状态)。

7. 决定是否延迟发送ACK

595-606 如果报文段中包含FIN，或者如果报文段中包含数据，那么客户端应用程序必须按使用T/TCP来编程(即调用sendto，并指定MSG_EOF，在此之前不能调用connect、write和shutdown)。在这种情况下， ACK要延迟发送，以便让服务器的应答来捎带服务器给出的SYN/ACK。

tcp_input.c

```
573             /*
574              * Perform TAO test on incoming CC (SEG.CC) option, if any.
575              * - compare SEG.CC against cached CC from the same host,
576              *  if any.
577              * - if SEG.CC > cached value, SYN must be new and is accepted
578              *  immediately: save new CC in the cache, mark the socket
579              *  connected, enter ESTABLISHED state, turn on flag to
580              *  send a SYN in the next segment.
581              *  A virtual advertised window is set in rcv_adv to
582              *  initialize SWS prevention.  Then enter normal segment
583              *  processing: drop SYN, process data and FIN.
584              * - otherwise do a normal 3-way handshake.
585              */
586             if ((to.to_flag & TOF_CC) != 0) {
587                 if (taop->tao_cc != 0 && CC_GT(to.to_cc, taop->tao_cc)) {
588                     /*
589                      * There was a CC option on the received SYN
590                      * and the TAO test succeeded.
591                      */
592                     tcpstat.tcps_taook++;
593                     taop->tao_cc = to.to_cc;
594                     tp->t_state = TCPS_ESTABLISHED;
595                     /*
596                      * If there is a FIN, or if there is data and the
597                      * connection is local, then delay SYN,ACK(SYN) in
```

图11-5 tcp_input ：对收到的报文段执行TAO测试

```
598                     * the hope of piggybacking it on a response
599                     * segment.  Otherwise must send ACK now in case
600                     * the other side is slow starting.
601                     */
602                    if ((tiflags & TH_FIN) ||
603                        (ti->ti_len != 0 && in_localaddr(inp->inp_faddr)))
604                        tp->t_flags |= (TF_DELACK | TF_SENDSYN);
605                    else
606                        tp->t_flags |= (TF_ACKNOW | TF_SENDSYN);
607                    tp->rcv_adv += tp->rcv_wnd;
608                    tcpstat.tcps_connects++;
609                    soisconnected(so);
610                    tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;
611                    dropsocket = 0;      /* committed to socket */
612                    tcpstat.tcps_accepts++;
613                    goto trimthenstep6;
614                } else if (taop->tao_cc != 0)
615                    tcpstat.tcps_taofail++;
```

tcp_input.c

图11-5 (续)

如果FIN标志未设置，但是报文段中包含数据，那么由于报文段中同时也包含了SYN标志，这很有可能是客户端发来的多报文段数据中的第一段。在这种情况下，如果客户端不在本地子网中(in_localaddr函数返回的是0)，这时因为客户端可能处于慢启动状态，确认不再延迟。

8. 设置rcv_adv

607 rcv_adv定义为所接收通告的最高序列号加1(卷2的图24-28)，但是在图11-4中的宏tcp_rcvseqinit将其初始化为接收序列号加1。在这个处理点上，rcv_wnd将是插口接收缓存的大小(卷2第752页)。这样，将rcv_wnd加到rcv_adv以后，后者刚好超出当前的接收窗口。rcv_adv必须在这里进行初始化，因为它的值要用在tcp_output的糊涂窗口避免机制中(卷2第700页)。rcv_adv在tcp_output快结束时设置，通常是在发送第一个报文段时(在这里应该是服务器对客户端SYN的响应SYN/ACK)。但是在T/TCP中，rcv_adv需要在tcp_output中进行第一次设置，因为我们可能在所发送的第一个报文段中发送数据。

9. 完成连接

608-609 递增tcps_connects和调用soisconnected通常是在接收到三次握手中的第三个报文段时进行的(卷2第774页)。既然连接已经完成，在T/TCP中这时就执行这两个步骤。

610-613 连接建立定时器设置为75秒，dropsocked标志设置为0，增加了标签为trimthenstep6的分支。

图11-6 给出了在LISTEN状态的插口收到一个SYN时的其余处理代码。

```
616                } else {
617                    /*
618                     * No CC option, but maybe CCnew:
619                     * invalidate cached value.
620                     */
621                    taop->tao_cc = 0;
622                }
```

tcp_input.c

图11-6 tcp_input：LISTEN处理：没有CC选项或TAO测试失败

```
623             /*
624              * TAO test failed or there was no CC option,
625              * do a standard 3-way handshake.
626              */
627             tp->t_flags |= TF_ACKNOW;
628             tp->t_state = TCPS_SYN_RECEIVED;
629             tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;
630             dropsocket = 0;      /* committed to socket */
631             tcpstat.tcps_accepts++;
632             goto trimthenstep6;
633         }
```

tcp_input.c

图11-6 (续)

10. 无CC选项；缓存设置为CC未定义

612-622 当CC选项不存在时，执行else语句(图11-5的第一个if语句在)。缓存中的CC值设置为0(即未定义)。如果该段程序中发现报文段中含有 CCnew选项，则当三次握手过程完成以后要更新缓存的CC值(图11-14)。

11. 要求执行三次握手

623-633 执行到这一点，要么报文段中没有 CC选项，要么有CC选项但TAO测试失败。在任何一种情况下，都要求执行三次握手过程。剩余的代码行与卷 2中图28-17的最后部分完全一样：设置TF_ACKNOW标志，状态设置为SYN_RCVD状态，这样就会立即发送SYN/ACK。

11.5 主动打开的启动

下一个case是SYN_SENT状态。TCP先前发送过一个SYN(一次主动打开)，现在是处理服务器的应答。图11-7给出了处理程序的第一部分。相应的Net/3代码从卷2第757页开始。

tcp_input.c

```
634         /*
635          * If the state is SYN_SENT:
636          *  if seg contains an ACK, but not for our SYN, drop the input.
637          *  if seg contains a RST, then drop the connection.
638          *  if seg does not contain SYN, then drop it.
639          * Otherwise this is an acceptable SYN segment
640          *  initialize tp->rcv_nxt and tp->irs
641          *  if seg contains ack then advance tp->snd_una
642          *  if SYN has been acked change to ESTABLISHED else SYN_RCVD state
643          *  arrange for segment to be acked (eventually)
644          *  continue processing rest of data/controls, beginning with URG
645          */
646     case TCPS_SYN_SENT:
647         if ((taop = tcp_gettaocache(inp)) == NULL) {
648             taop = &tao_noncached;
649             bzero(taop, sizeof(*taop));
650         }
651         if ((tiflags & TH_ACK) &&
652             (SEQ_LEQ(ti->ti_ack, tp->iss) ||
653              SEQ_GT(ti->ti_ack, tp->snd_max))) {
654             /*
655              * If we have a cached CCsent for the remote host,
656              * hence we haven't just crashed and restarted,
657              * do not send a RST.  This may be a retransmission
```

图11-7 tcp_input ：在SYN_SENT状态的初始处理

```
658             * from the other side after our earlier ACK was lost.
659             * Our new SYN, when it arrives, will serve as the
660             * needed ACK.
661             */
662            if (taop->tao_ccsent != 0)
663                goto drop;
664            else
665                goto dropwithreset;
666        }
667        if (tiflags & TH_RST) {
668            if (tiflags & TH_ACK)
669                tp = tcp_drop(tp, ECONNREFUSED);
670            goto drop;
671        }
672        if ((tiflags & TH_SYN) == 0)
673            goto drop;
674        tp->snd_wnd = ti->ti_win;    /* initial send window */
675        tp->cc_recv = to.to_cc; /* foreign CC */

676        tp->irs = ti->ti_seq;
677        tcp_rcvseqinit(tp);
```

tcp_input.c

图11-7 (续)

1. 取TAO缓存记录项

647-650 取该服务器的TAO缓存记录项。因为我们最近刚刚发送过SYN，应该有一个记录项。

2. 处理不正确的ACK

651-666 如果报文段中包含有ACK，但是其确认字段不正确(见卷2中图28-19对进行比较的几个字段的叙述)，我们的应答就依赖于是否已经为该主机缓存了 `tao_ccsent`。如果`cc_ccsent`非0，则丢弃该报文段，而不发送RST。这个处理步骤的代码段在图4-7中的标号为“discard”处。但如果`tao_ccsent`为0，我们丢弃该报文段，并发送RST，这是在该状态下对不正确ACK的正常TCP响应。

3. 检查RST

667-671 如果接收到的报文段中设置了RST标志，则丢弃报文段。另外，如果设置了ACK标志，则说明服务器的TCP主动拒绝连接，并将`ECONNREFUSED`错误返回给调用进程。

4. 必须设置SYN

672-673 如果SYN标志没有设置，则丢弃报文段。

674-677 初始发送窗口设置为报文段中通告的窗口宽度，并将 `cc_recv`设置为接收到的CC值(如果CC选项不存在，就为0)。`irs`是初始接收序号，宏`tcp_rcvseqinit`对控制块中的接收变量进行初始化。

代码中现在开始出现分支，决定于是否报文段中包含一个对所发SYN的确认ACK(通常情况下)，或者是否ACK标志没有打开(双方同时进行打开的情况较少发生)。图11-18给出了通常的情况。

```
678        if (tiflags & TH_ACK) {
679            /*
680             * Our SYN was acked.  If segment contains CCecho
681             * option, check it to make sure this segment really
```

tcp_input.c

图11-8 tcp_input：在SYN_SENT状态处理SYN/ACK响应

```
682              * matches our SYN.  If not, just drop it as old
683              * duplicate, but send an RST if we're still playing
684              * by the old rules.
685              */
686             if ((to.to_flag & TOF_CCECHO) &&
687                 tp->cc_send != to.to_ccecho) {
688                 if (taop->tao_ccsent != 0) {
689                     tcpstat.tcps_badccecho++;
690                     goto drop;
691                 } else
692                     goto dropwithreset;
693             }
694             tcpstat.tcps_connects++;
695             soisconnected(so);

696             /* Do window scaling on this connection? */
697             if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
698                 (TF_RCVD_SCALE | TF_REQ_SCALE)) {
699                 tp->snd_scale = tp->requested_s_scale;
700                 tp->rcv_scale = tp->request_r_scale;
701             }
702             /* Segment is acceptable, update cache if undefined. */
703             if (taop->tao_ccsent == 0)
704                 taop->tao_ccsent = to.to_ccecho;

705             tp->rcv_adv += tp->rcv_wnd;
706             tp->snd_una++;      /* SYN is acked */
707             /*
708              * If there's data, delay ACK; if there's also a FIN
709              * ACKNOW will be turned on later.
710              */
711             if (ti->ti_len != 0)
712                 tp->t_flags |= TF_DELACK;
713             else
714                 tp->t_flags |= TF_ACKNOW;
715             /*
716              * Received <SYN,ACK> in SYN_SENT[*] state.
717              * Transitions:
718              *  SYN_SENT  --> ESTABLISHED
719              *  SYN_SENT* --> FIN_WAIT_1
720              */
721             if (tp->t_flags & TF_SENDFIN) {
722                 tp->t_state = TCPS_FIN_WAIT_1;
723                 tp->t_flags &= ~TF_SENDFIN;
724                 tiflags &= ~TH_SYN;
725             } else
726                 tp->t_state = TCPS_ESTABLISHED;
```

tcp_input.c

图11-8 (续)

5. ACK标志是打开的

678 如果ACK标志处于开的状态，我们从图11-7中的`ti_ack`测试可以知道，ACK确认了我们的SYN。

6. 检查CCecho值是否存在

679-693 如果报文段中包含了CCecho选项，但CCecho的值与我们发出的不相等，则丢弃该报文段(除非另一端发生故障，否则，因为已经收到了对所发SYN的ACK，所以这是“决不应该发生的”)。如果我们并没有发送过CC选项(`tao_ccsent`为0)，那么就要发送RST。

7. 标记插口为已连接和处理窗口宽度选项

694-701 插口标记为已经建立连接，并对窗口宽度选项进行处理 (如果存在)。

Bob Braden的 T/TCP实现有错误，不应在测试CCecho值之前就执行这两行代码。

8. 如果未定义，则更新TAO缓存

702-704 报文段可以接受，这样，如果这个服务器的 TAO缓存还没有定义(例如客户重新启动了，或发送了一个 CCnew选项)，我们就用接收到的CCecho值(如果CCecho选项不存在，它的值为0)对其进行更新。

9. 设置rcv_adv

705-706 更新rcv_adv，如图11-4所示。在发出的SYN得到确认后，snd_una(尚未确认的最小序号数据)加1。

10. 确定是否延迟发送ACK

707-714 如果服务器在其 SYN中发送数据，那我们就延迟发送 ACK；否则，立即发出ACK(因为这很可能是三次握手中的第二个报文段)。延迟发送ACK是因为，如果服务器发来的SYN中包含了数据，那么服务器很可能正在使用 T/TCP，这样就很可能还会接收到另外的报文段，其中包含剩余的应答数据，这时就没有必要立即发送 ACK。但如果这个报文段中同时还包含有服务器的FIN(最小的三报文段T/TCP交换中的第二个报文段)，图11-18中的代码会打开TF_ACKNOW标志，以便立即发送ACK。

715-726 我们知道t_state等于TCPS_SYN_SENT，但如果隐藏状态标志TF_SENDFIN也是打开的，我们的状态实际上是 SYN_SENT*。在这种情况下，我们的状态变迁到FIN_WAIT_1状态(如果看看RFC 1644中的状态变迁图就可以看出，这实际上是两个状态变迁的组合。在SYN_SENT*状态下接收到SYN就变迁到FIN_WAIT_1*状态，而对SYN的ACK则变迁到FIN_WAIT_1状态)。

对应于图11-8开头的if的else代码如图11-9所示。它对应的是两端同时打开：我们发出了一个SYN，然后收到了一个没有ACK的SYN。这个图取代了卷2第758页的第581~582行。

tcp_input.c

```
727         } else {
728             /*
729              * Simultaneous open.
730              * Received initial SYN in SYN-SENT[*] state.
731              * If segment contains CC option and there is a
732              * cached CC, apply TAO test; if it succeeds,
733              * connection is half-synchronized.
734              * Otherwise, do 3-way handshake:
735              *        SYN-SENT -> SYN-RECEIVED
736              *        SYN-SENT* -> SYN-RECEIVED*
737              * If there was no CC option, clear cached CC value.
738              */
739             tp->t_flags |= TF_ACKNOW;
740             tp->t_timer[TCPT_REXMT] = 0;
741             if (to.to_flag & TOF_CC) {
742                 if (taop->tao_cc != 0 && CC_GT(to.to_cc, taop->tao_cc)) {
743                     /*
744                      * update cache and make transition:
745                      *         SYN-SENT -> ESTABLISHED*
746                      *         SYN-SENT* -> FIN-WAIT-1*
```

图11-9 tcp_input ：同时打开

```
747                      */
748                     tcpstat.tcps_taook++;
749                     taop->tao_cc = to.to_cc;
750                     if (tp->t_flags & TF_SENDFIN) {
751                         tp->t_state = TCPS_FIN_WAIT_1;
752                         tp->t_flags &= ~TF_SENDFIN;
753                     } else
754                         tp->t_state = TCPS_ESTABLISHED;
755                     tp->t_flags |= TF_SENDSYN;
756                 } else {
757                     tp->t_state = TCPS_SYN_RECEIVED;
758                     if (taop->tao_cc != 0)
759                         tcpstat.tcps_taofail++;
760                 }
761             } else {
762                 /* CCnew or no option => invalidate cache */
763                 taop->tao_cc = 0;
764                 tp->t_state = TCPS_SYN_RECEIVED;
765             }
766         }
```

tcp_input.c

图11-9 (续)

11. 立即ACK和关闭重传定时器

739-740 立即发出ACK，并且关闭重传定时器。尽管定时器被关闭，但由于 TF_ACKNOW 标志是设置了的，在 tcp_input快结束时调用 tcp_output。在发送ACK时，因为至少有一个数据字节(SYN)已经发出且未得到确认，重新启动重传定时器。

12. 执行TAO测试

741-755 如果报文段中包含有CC选项，那么就要执行TAO测试：缓存的值(tao_cc)必须非0，接收到的CC值必须大于缓存中的值。如果通过了 TAO测试，缓存的值就要用接收到的 CC值进行更新，这时要么从 SYN_SENT状态变迁到ESTABLISHED*状态，要么从SYN-SNET*状态变迁到FIN_WAIT_1*状态。

13. TAO测试失败或没有CC选项

756-765 如果TAO测试失败，新的状态就是 SYN_RCVD。如果没有CC选项，则TAO缓存的内容置0(未定义)，新的状态也是SYN_RCVD。

图11-10给出了标号为 trimthenstep6的代码段，是在处理 LISTEN状态结束时的一个分支(见图11-5)。这个图中的大部分代码是从卷 2第759页复制过来的。

tcp_input.c

```
767     trimthenstep6:
768       /*
769        * Advance ti->ti_seq to correspond to first data byte.
770        * If data, trim to stay within window,
771        * dropping FIN if necessary.
772        */
773       ti->ti_seq++;
774       if (ti->ti_len > tp->rcv_wnd) {
775           todrop = ti->ti_len - tp->rcv_wnd;
776           m_adj(m, -todrop);
777           ti->ti_len = tp->rcv_wnd;
778           tiflags &= ~TH_FIN;
```

图11-10 tcp_input :处理完主动或被动打开后执行的 trimthenstep6 代码段

```
779             tcpstat.tcps_rcvpackafterwin++;
780             tcpstat.tcps_rcvbyteafterwin += todrop;
781         }
782         tp->snd_wl1 = ti->ti_seq - 1;
783         tp->rcv_up = ti->ti_seq;
784         /*
785          *  Client side of transaction: already sent SYN and data.
786          *  If the remote host used T/TCP to validate the SYN,
787          *  our data will be ACK'd; if so, enter normal data segment
788          *  processing in the middle of step 5, ack processing.
789          *  Otherwise, goto step 6.
790          */
791         if (tiflags & TH_ACK)
792             goto processack;
793         goto step6;
```

tcp_input.c

图11-10 (续)

14. 是客户端则不要跳过ACK处理

784-793 如果ACK标志打开，我们就是事务过程中的客户端。也就是说，我们发送的SYN得到了ACK，我们是从SYN_SENT状态变迁到当前状态的，而不是从LISTEN状态变迁来的。在这种情况下，我们不能执行step6分支，因为那样就会跳过对ACK的处理过程(见图11-1)，而如果我们在SYN报文段中发送了数据，就需要对数据的ACK进行处理(常规的TCP会在这里跳过对ACK的处理过程，因为它从来不会随SYN一起发送数据)。

处理过程中的下一个步骤是T/TCP中新加的。通常，卷2第753页开始的switch语句中只有LISTEN和SYN_SENT状态这两个case处理代码(这两种情况我们都刚刚介绍过)。T/TCP增加了LAST_ACK、CLOSING和TIME_WAIT状态这三段case处理代码，如图11-11所示。

tcp_input.c

```
794         /*
795          * If the state is LAST_ACK or CLOSING or TIME_WAIT:
796          *  if segment contains a SYN and CC [not CCnew] option
797          *  and peer understands T/TCP (cc_recv != 0):
798          *              if state == TIME_WAIT and connection duration > MSL,
799          *                  drop packet and send RST;
800          *
801          *      if SEG.CC > CCrecv then is new SYN, and can implicitly
802          *          ack the FIN (and data) in retransmission queue.
803          *                  Complete close and delete TCPCB.  Then reprocess
804          *                  segment, hoping to find new TCPCB in LISTEN state;
805          *
806          *      else must be old SYN; drop it.
807          *      else do normal processing.
808          */
809     case TCPS_LAST_ACK:
810     case TCPS_CLOSING:
811     case TCPS_TIME_WAIT:
812         if ((tiflags & TH_SYN) &&
813             (to.to_flag & TOF_CC) && tp->cc_recv != 0) {
814             if (tp->t_state == TCPS_TIME_WAIT &&
815                 tp->t_duration > TCPTV_MSL)
816                 goto dropwithreset;
817             if (CC_GT(to.to_cc, tp->cc_recv)) {
818                 tp = tcp_close(tp);
```

图11-11 tcp_input ：LAST_ACK、CLOSING和TIME_WAIT状态的初始处理

```
819                 tcpstat.tcps_impliedack++;
820                 goto findpcb;
821             } else
822                 goto drop;
823         }
824         break;                  /* continue normal processing */
825     }
```

tcp_input.c

图11-11 (续)

812-813 只有在接收到的报文段中包含了SYN和CC选项，并且我们已经有该主机的缓存CC值(cc_recv非0)，才执行接下来的特殊测试。同时知道要进入三种状态之一，TCP已发出了一个FIN，并接收到一个FIN(图2-6)。在LAST_ACK和CLOSING状态下，TCP等待对其所发FIN的ACK。所以要执行的测试是在TIME_WAIT状态下收到新的SYN时是否可以安全地截断TIME_WAIT状态，或者在LAST_ACK或CLOSING状态下收到一个新的SYN是否隐含着我们所发送FIN的ACK。

15. 如果持续时间大于MSL就不允许截断TIME_WAIT状态

814-816 通常，处于TIME_WAIT状态下的连接是允许接收新SYN的(卷2第765~766页)。这是从伯克利演变来的系统所允许的隐式截断TIME_WAIT状态，至少从NET/1以后就是这样了(卷1的习题18.5的解答就说明了这一特性)。如果连接处于TIME_WAIT状态的持续时间大于MSL，上述做法在T/TCP中是不允许的，这时要发送RST。我们在4.4节中讲到过这个限制。

16. 新SYN是现存连接的隐含ACK

817-820 如果接收到的CC值大于缓存的CC值，则TAO测试成功(即这是一个新的SYN)。这时关闭当前连接，回头执行findpcb分支，希望找到一个处于LISTEN状态的插口来处理新的SYN。图4-7给出了服务器插口的一个例子，在处理隐含的ACK时，插口处于LAST_ACK状态。

11.6 PAWS：防止序号重复

卷2第740页的PAWS测试没有变化——就是处理时间戳的代码。图11-12所示的测试在这些时间戳测试之后执行，验证接收到的CC。

tcp_input.c

```
860     /*
861      * T/TCP mechanism:
862      *   If T/TCP was negotiated, and the segment doesn't have CC
863      *   or if its CC is wrong, then drop the segment.
864      *   RST segments do not have to comply with this.
865      */
866     if ((tp->t_flags & (TF_REQ_CC | TF_RCVD_CC)) == (TF_REQ_CC | TF_RCVD_CC) &&
867         ((to.to_flag & TOF_CC) == 0 || tp->cc_recv != to.to_cc) &&
868         (tiflags & TH_RST) == 0) {
869         tcpstat.tcps_ccdrop++;
870         goto dropafterack;
871     }
```

tcp_input.c

图11-12 tcp_input：验证接收到的CC

860-871 如果使用T/TCP(TF_REQ_CC和TF_RCVD_CC选项同时打开)，这时接收到的报文段必须包含CC选项，且CC值必须等于该连接所用的值(cc_recv)；否则，报文段就是过时

重复的，要丢弃(但要给出确认，因为所有重复的报文段都需要确认)。如果报文段中包含了RST，就不丢弃，允许处理该报文段的函数稍后可以对 RST进行处理。

11.7 ACK处理

在卷2第771页上，RST处理后，如果ACK标志没有打开，报文段就被丢弃。这是常规的TCP处理过程。T/TCP改变这一点，如图11-13所示。

tcp_input.c

```
1024    /*
1025     * If the ACK bit is off: if in SYN-RECEIVED state or SENDSYN
1026     * flag is on (half-synchronized state), then queue data for
1027     * later processing; else drop segment and return.
1028     */
1029    if ((tiflags & TH_ACK) == 0) {
1030        if (tp->t_state == TCPS_SYN_RECEIVED ||
1031            (tp->t_flags & TF_SENDSYN))
1032            goto step6;
1033        else
1034            goto drop;
1035    }
```

tcp_input.c

图11-13 tcp_input ：处理没有ACK标志的报文段

1024-1035 如果ACK标志关闭，并且状态是SYN_RCVD，或者TF_SESNDSYN标志打开(即半同步)，则执行step6分支，而不是丢弃该报文段。这样做处理的是在连接建立前、但第一个SYN之后、不带ACK的数据报文段到达的情况(例如图3-9的第2报文段和第3报文段)。

11.8 完成被动打开和同时打开

如卷2的第29章一样，继续对 ACK进行处理。第774页的大部分代码还是一样的 (删除第806行)，但用图11-14的代码替代其中的813~815行。这时我们处于SYN_RCVD状态，处理的是完成三次握手的最后一个ACK。这是在服务器上对连接的常规处理过程。

1. 如果缓存的CC值未定义，就更新

1057-1064 读取这个对等端的TAO记录项，如果所缓存的CC值为0(未定义)，则用接收到的CC值更新。注意，只有在缓存的值未定义时才执行更新操作。回顾前面，图 11-6的代码在CC选项不存在时明确地将tao_cc设置为0(这样，当三次握手完成时就会进行更新)。但是如果TAO测试失败，也就不会修改 tao_cc的值。后面这个动作实际上就是收到了一个失序的SYN，不应引起缓存tao_cc的改变，如我们在图4-11中所述。

2. 变迁到新状态

1065-1074 从SYN_RCVD状态变迁到ESTABLISHED状态，是服务器完成三次握手过程的常规TCP状态变迁。因为进程已经用 MSG_EOF标志关闭了用于发送的半个连接，连接状态从SYN_RCVD*变迁到FIN_WAIT_1状态。

tcp_input.c

```
1057        /*
1058         * Upon successful completion of 3-way handshake,
1059         * update cache.CC if it was undefined, pass any queued
1060         * data to the user, and advance state appropriately.
```

图11-14 tcp_input ：被动打开或同时打开的完成

```
1061          */
1062         if ((taop = tcp_gettaocache(inp)) != NULL &&
1063             taop->tao_cc == 0)
1064             taop->tao_cc = tp->cc_recv;

1065         /*
1066          * Make transitions:
1067          *      SYN-RECEIVED  -> ESTABLISHED
1068          *      SYN-RECEIVED* -> FIN-WAIT-1
1069          */
1070         if (tp->t_flags & TF_SENDFIN) {
1071             tp->t_state = TCPS_FIN_WAIT_1;
1072             tp->t_flags &= ~TF_SENDFIN;
1073         } else
1074             tp->t_state = TCPS_ESTABLISHED;

1075         /*
1076          * If segment contains data or FIN, will call tcp_reass()
1077          * later; if not, do so now to pass queued data to user.
1078          */
1079         if (ti->ti_len == 0 && (tiflags & TH_FIN) == 0)
1080             (void) tcp_reass(tp, (struct tcpiphdr *) 0,
1081                              (struct mbuf *) 0);
1082         tp->snd_wl1 = ti->ti_seq - 1;
1083         /* fall into ... */
```

tcp_input.c

图11-14 (续)

3. 检查数据或FIN

1075-1081 如果报文段中包含有数据或FIN标志，那么在标号为dodata的代码行就要调用宏TCP_REASS(回顾图11-1)将数据交付给用户进程。卷2第790页给出了在标号dodata处对这个宏的调用，这段代码在T/TCP中没有改变。否则就要调用tcp_reass，其第二个参数为0，将队列中的所有数据交付给用户进程。

11.9 ACK处理(续)

快速重传和快速恢复算法(卷2的29.4节)保持不变。图11-15中的代码插入在卷2第779页的899~900行之间。

tcp_input.c

```
1168         /*
1169          *  If we reach this point, ACK is not a duplicate,
1170          *     i.e., it ACKs something we sent.
1171          */
1172         if (tp->t_flags & TF_SENDSYN) {
1173             /*
1174              *   T/TCP: Connection was half-synchronized, and our
1175              *   SYN has been ACK'd (so connection is now fully
1176              *   synchronized).  Go to non-starred state and
1177              *   increment snd_una for ACK of SYN.
1178              */
1179             tp->t_flags &= ~TF_SENDSYN;
1180             tp->snd_una++;
1181         }
1182       processack:
```

tcp_input.c

图11-15 tcp_input ：如果TF_SENDSYN 打开了，就将其关闭

1. 关闭隐藏状态标志TF_SENDSYN

1168-1181 如果隐藏状态标志TF_SENDSYN处于打开状态，则它将被关闭。这是因为接收到的ACK确认了已经发送出去的一些东西，连接已不再是半同步。因为 SYN已经得到确认，并且SYN占用了1字节序号空间，snd_una加1。

图11-16插在卷2第780~781页的926~927行之间。

```
                                                                    tcp_input.c
1210            /*
1211             * If no data (only SYN) was ACK'd,
1212             *    skip rest of ACK processing.
1213             */
1214            if (acked == 0)
1215                goto step6;
                                                                    tcp_input.c
```

图11-16 tcp_input ：如果没有对数据的ACK就跳过剩ACK处理过程

2. 如果没有对数据的ACK，就跳过剩余ACK处理过程

1210-1215 如果没有对数据的确认(仅对我们的SYN给出了确认)，ACK处理过程的剩余部分就跳过去。跳过去的处理代码包括打开拥塞窗口和将已得到确认的数据从发送缓存中移去。

这项测试和程序分支在T/TCP中不存在。这纠正了在14.12节的最后讨论的一个程序缺陷，在那里连接的服务器端通过发送两个背靠背段来执行慢启动，而不是一个报文段。

第2个变化如图11-17所示，用于替代卷2的图29-12中的代码。这时我们处于CLOSING状态并对ACK进行处理，处理结果是将连接的状态变迁到 TIME_WAIT。T/TCP允许截断TIME_WAIT状态(见4.4节)。

```
                                                                    tcp_input.c
1266                /*
1267                 * In CLOSING STATE in addition to the processing for
1268                 * the ESTABLISHED state if the ACK acknowledges our FIN
1269                 * then enter the TIME-WAIT state, otherwise ignore
1270                 * the segment.
1271                 */
1272            case TCPS_CLOSING:
1273                if (ourfinisacked) {
1274                    tp->t_state = TCPS_TIME_WAIT;
1275                    tcp_canceltimers(tp);
1276                    /* Shorten TIME_WAIT [RFC 1644, p.28] */
1277                    if (tp->cc_recv != 0 && tp->t_duration < TCPTV_MSL)
1278                        tp->t_timer[TCPT_2MSL] = tp->t_rxtcur * TCPTV_TWTRUNC;
1279                    else
1280                        tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1281                    soisdisconnected(so);
1282                }
1283                break;
                                                                    tcp_input.c
```

图11-17 tcp_input :在CLOSING状态收到ACK：设置TIME_WAIT定时器

1276-1280 如果我们从对等端接收到一个CC值，并且连接的持续时间少于MSL，这时TIME_WAIT定时器就设置为当前重传超时的TCPTV_TWTRUNC(8)倍。否则，TIME_WAIT定时器设置为通常的2倍MSL。

11.10 FIN处理

TCP输入处理的接下来的三部分(更新窗口信息、紧急模式处理和接收数据处理)在T/TCP中都没有改变(回忆图11-1)。再回顾卷2的29.9节，如果设置了FIN标志，但因为序号空间的空洞，它不会得到确认，那一节中的代码就用于清除FIN标志。因此，在这里我们知道FIN是要等待确认的。

FIN处理过程的另一个变化如图11-18所示。这个修改用于替代卷2第791页的第1123行。

tcp_input.c

```
1407     /*
1408      * If FIN is received ACK the FIN and let the user know
1409      * that the connection is closing.
1410      */
1411     if (tiflags & TH_FIN) {
1412         if (TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1413             socantrcvmore(so);
1414             /*
1415              *  If connection is half-synchronized
1416              *  (i.e., TF_SENDSYN flag on) then delay the ACK
1417              *  so it may be piggybacked when SYN is sent.
1418              *  Else, since we received a FIN, no more
1419              *  input can be received, so we send the ACK now.
1420              */
1421             if (tp->t_flags & TF_SENDSYN)
1422                 tp->t_flags |= TF_DELACK;
1423             else
1424                 tp->t_flags |= TF_ACKNOW;
1425             tp->rcv_nxt++;
1426         }
```

tcp_input.c

图11-18 tcp_input：确定是否迟延发送FIN的ACK

1. 决定是否延迟发送ACK

1414-1424 如果连接是半同步的(隐藏状态标志TF_SENDSYN打开)，ACK就要延迟发送，试图在数据报文段中捎带ACK。这是一种典型的情况，处于LISTEN状态的T/TCP服务器收到SYN，这样图11-5中的代码就会设置TF_SENDSYN标志。注意，图中代码已将延迟发送ACK的标志打开，但这里是TCP要根据已经设置了的FIN标志确定怎样去做。如果TF_SENDSYN标志没有打开，则ACK不能延迟。

常规的变迁是从FIN_WAIT_2状态到TIME_WAIT状态，在T/TCP中这也需要修改，以便使TIME_WAIT状态可能被截断(见4.4节)。图11-19给出了这些修改，用于取代卷2第792页的1142~1152行。

tcp_input.c

```
1443             /*
1444              * In FIN_WAIT_2 state enter the TIME_WAIT state,
1445              * starting the time-wait timer, turning off the other
1446              * standard timers.
1447              */
1448         case TCPS_FIN_WAIT_2:
1449             tp->t_state = TCPS_TIME_WAIT;
1450             tcp_canceltimers(tp);
1451             /* Shorten TIME_WAIT [RFC 1644, p.28] */
```

图11-19 tcp_input：变迁到TIME_WAIT状态以便可能截断超时间隔

```
1452            if (tp->cc_recv != 0 && tp->t_duration < TCPTV_MSL) {
1453                tp->t_timer[TCPT_2MSL] = tp->t_rxtcur * TCPTV_TWTRUNC;
1454                /* For transaction client, force ACK now. */
1455                tp->t_flags |= TF_ACKNOW;
1456            } else
1457                tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1458            soisdisconnected(so);
1459            break;
```

tcp_input.c

图11-19 (续)

2. 设置TIME_WAIT超时间隔

1451-1453 如图11-17所示，只有当我们从对等端收到一个 CC选项，并且连接时间短于MSL时，TIME_WAIT状态才能截断。

3. 强迫立即发送FIN的ACK

1454-1455 这个变迁通常是在T/TCP的客户端，当收到服务器的响应以及服务器的 SYN和FIN时发生的。服务器的FIN应该立即给出ACK，因为两端都已经发送了 FIN，已经没有理由再延迟发送ACK了。

在两个地方要重启动 TIME_WAIT定时器：处于TIME_WAIT状态时接收到ACK和处于TIME_WAIT状态时接收到FIN(卷2第784页和第792页)。T/TCP没有修改这些代码。这表明，即使状态 TIME_WAIT被截断，如果在这时收到重复的 ACK或FIN，定时器就要在2MSL时重启动，而不是在截断后的值。重启动定时器所需要的信息在截断后的值时也能得到(即控制块)，但是由于对等端必须重传，更保守的做法是不截断TIME_WAIT状态。

11.11 小结

T/TCP所做的修改大部分都是在 `tcp_input`中，并且其中的大部分修改都与打开新连接有关。

在LISTEN状态收到SYN时要执行TAO测试。如果报文段通过了这个测试，报文段就不是过时的重复报文段，三次握手也就不需要了。在 SYN_SENT状态收到SYN时，CCecho选项(如果存在)就用于验证该 SYN不是过时的重复报文段。当处于 LAST_ACK、CLOSING和TIME_WAIT状态收到SYN时，很有可能SYN是一个隐含的ACK，可以完成现存连接的关闭。

当主动关闭一个连接时，如果连接的持续时间短于 MSL，则TIME_WAIT状态被截断。