第28章 TCP的输入

28.1 引言

TCP输入处理是系统中最长的一部分代码，函数 tcp_input约有1100行代码。输入报文段的处理并不复杂，但非常繁琐。许多实现，包括 Net/3，都完全遵循RFC 793中定义的输入事件处理步骤，它详细定义了如何根据连接的当前状态，处理不同的输入报文段。

当收到的数据报的协议字段指明这是一个 TCP报文段时， ipintr(通过协议转换表中的pr_input函数)会调用tcp_input进行处理。tcp_input在软件中断一级执行。

函数非常长，我们将分两章讨论。图 28-1列出了tcp_input中的处理框架。本章将结束对RST报文段处理的讲解，从下一章开始介绍 ACK报文段的处理。

头几个步骤是非常典型的：对输入报文段做有效性验证 (检验和、长度等)，以及寻找连接的PCB。尽管后面还有大量代码，但通过“ 首部预测 (header prediction) ”(28.4节)，算法却有可能完全跳过后续的逻辑。首部预测算法是基于这样的假定：一般情况下，报文段既不会丢失，次序也不会错误，因此，对于给定连接， TCP总能猜到下一个接收报文段的内容。如果算法起作用，函数直接返回，这是tcp_input中最快的一条执行路径。

如果算法不起作用，函数在“ dodata ”处结束，测试几个标志，并且若需要对接收报文段作出响应，则调用tcp_output。

```
void
tcp_input()
{
    checksum TCP header and data;
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;
        goto trimthenstep6;
    case TCPS_SYN_SENT:
        if ACK of our SYN, connection completed;
trimthenstep6:
        trim any data not within window;
```

图28-1 TCP输入处理步骤小结

```
            goto step6;
        }
        process RFC 1323 timestamp;
        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;
        }                           /* Chapter 28 finishes here */
        if (ACK flag set) {         /* Chapter 29 starts here */
            if (SYN_RCVD state)
                simultaneous open complete;
            if (duplicate ACK)
                fast recovery algorithm;
            update RTT estimators if segment timed;
            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28-1 (续)

函数结尾处有3个标注，处理出现差错时控制会跳转到这些地方： dropafterack、dropwithreset和drop。标注中出现的“drop”指丢弃当前处理的报文段，而非丢弃连接。不过，当控制跳转到dropwithreset时，将发送RST，从而丢弃连接。

函数仅有的另一个分支是首部预测算法后的 switch语句，如果连接处于 LISTEN或SYN_SENT状态时收到了一个有效的SYN报文段，它负责分别进行处理。 trimthenstep6

处的代码结束后，跳转到step 6继续执行正常的流程。

28.2 预处理

图28-2的代码包含一些声明，并对收到的TCP报文段进行预处理。

1. 从第一个mbuf中获取IP和TCP首部

170-204 参数iphlen等于IP首部长度，包括可能的IP选项。如果长度大于20字节，可知存在IP选项，调用ip_stripoptions丢弃这些选项。TCP忽略除源选路之外的所有IP选项，源选路选项由IP特别保存(9.6节)，TCP能够读取其内容(图28-7)。如果簇中第一个mbuf的容量小于IP/TCP首部大小(40字节)，则调用m_pullup，试着把最初的40字节移入第一个mbuf中。

tcp_input.c

```
170 void
171 tcp_input(m, iphlen)
172 struct mbuf *m;
173 int     iphlen;
174 {
175     struct tcpiphdr *ti;
176     struct inpcb *inp;
177     caddr_t optp = NULL;
178     int     optlen;
179     int     len, tlen, off;
180     struct tcpcb *tp = 0;
181     int     tiflags;
182     struct socket *so;
183     int     todrop, acked, ourfinisacked, needoutput = 0;
184     short   ostate;
185     struct in_addr laddr;
186     int     dropsocket = 0;
187     int     iss = 0;
188     u_long  tiwin, ts_val, ts_ecr;
189     int     ts_present = 0;

190     tcpstat.tcps_rcvtotal++;
191     /*
192      * Get IP and TCP header together in first mbuf.
193      * Note: IP leaves IP header in first mbuf.
194      */
195     ti = mtod(m, struct tcpiphdr *);
196     if (iphlen > sizeof(struct ip))
197                 ip_stripoptions(m, (struct mbuf *) 0);
198     if (m->m_len < sizeof(struct tcpiphdr)) {
199         if ((m = m_pullup(m, sizeof(struct tcpiphdr))) == 0) {
200             tcpstat.tcps_rcvshort++;
201             return;
202         }
203         ti = mtod(m, struct tcpiphdr *);
204     }
```

tcp_input.c

图28-2 tcp_input 函数：变量声明及预处理

图28-3给出了函数下一部分的代码，验证TCP检验和及偏移字段。

2. 验证TCP检验和

205-217 tlen指TCP报文段的长度，即IP首部后的字节数。前面介绍过，IP已经从

ip_len中减去了IP的首部长度，因此，变量len就等于整个IP数据报的长度，即包括伪首部在内的需要计算检验和的数据长度。根据TCP检验和计算的要求，填充伪首部中的各个字段，如图23-19所示。

tcp_input.c

```
205     /*
206      * Checksum extended TCP header and data.
207      */
208     tlen = ((struct ip *) ti)->ip_len;
209     len = sizeof(struct ip) + tlen;
210     ti->ti_next = ti->ti_prev = 0;
211     ti->ti_x1 = 0;
212     ti->ti_len = (u_short) tlen;
213     HTONS(ti->ti_len);
214     if (ti->ti_sum = in_cksum(m, len)) {
215         tcpstat.tcps_rcvbadsum++;
216         goto drop;
217     }
218     /*
219      * Check that TCP offset makes sense,
220      * pull out TCP options and adjust length.       XXX
221      */
222     off = ti->ti_off << 2;
223     if (off < sizeof(struct tcphdr) || off > tlen) {
224         tcpstat.tcps_rcvbadoff++;
225         goto drop;
226     }
227     tlen -= off;
228     ti->ti_len = tlen;
```

tcp_input.c

图28-3 tcp_input 函数：验证TCP检验和及偏移字段

3. 验证TCP偏移字段

218-228 TCP的偏移字段，ti_off，是以32 bit为单位的TCP首部长度值，包括所有的TCP选项。把它乘以4(得到TCP报文段中第一个数据字节所在位置的偏移量)，并验证其有效性。偏移量必须大于等于标准TCP首部的大小(20字节)，并且小于等于TCP报文段的长度。

从TCP长度变量tlen中减去首部长度，得到报文段中携带的数据字节数 (可能为0)，并把这个值赋给tlen，以及TCP首部的变量ti_len。函数中会多次用到这个值。

图28-4给出了函数下一部分的代码：处理特定的TCP选项。

tcp_input.c

```
229     if (off > sizeof(struct tcphdr)) {
230         if (m->m_len < sizeof(struct ip) + off) {
231             if ((m = m_pullup(m, sizeof(struct ip) + off)) == 0) {
232                 tcpstat.tcps_rcvshort++;
233                 return;
234             }
235             ti = mtod(m, struct tcpiphdr *);
236         }
237         optlen = off - sizeof(struct tcphdr);
238         optp = mtod(m, caddr_t) + sizeof(struct tcpiphdr);
239         /*
240          * Do quick retrieval of timestamp options ("options
```

图28-4 tcp_input 函数：处理特定的TCP选项

```
241          * prediction?").  If timestamp is the only option and it's
242          * formatted as recommended in RFC 1323 Appendix A, we
243          * quickly get the values now and not bother calling
244          * tcp_dooptions(), etc.
245          */
246         if ((optlen == TCPOLEN_TSTAMP_APPA ||
247              (optlen > TCPOLEN_TSTAMP_APPA &&
248               optp[TCPOLEN_TSTAMP_APPA] == TCPOPT_EOL)) &&
249             *(u_long *) optp == htonl(TCPOPT_TSTAMP_HDR) &&
250             (ti->ti_flags & TH_SYN) == 0) {
251             ts_present = 1;
252             ts_val = ntohl(*(u_long *) (optp + 4));
253             ts_ecr = ntohl(*(u_long *) (optp + 8));
254             optp = NULL;        /* we've parsed the options */
255         }
256     }
```

tcp_input.c

图28-4 (续)

4. 把IP和TCP首部及选项放入第一个mbuf

230-236 如果首部长度大于20，说明存在TCP选项。必要时调用m_pullup，把标准IP首部、标准TCP首部的所有TCP选项放入簇中的第一个mbuf中。因为3部分数据最大只能为80字节(20+20+40)，因此，必定能够放入第一个存储数据分组首部的mbuf中。

此处能够造成m_pullup失败的惟一原因是IP数据分组的字节数小于20加上TCP首部长度，而且已通过TCP检验和的验证，我们认为m_pullup不可能失败。但有一点，图28-2中调用的m_pullup，将共享计数器tcps_rcvshort，因此，查看tcps_rcvshort并不能说明哪一个调用失败。不管怎样，从图24-5可知，即使收到九百万个TCP报文段之后，这个计数器仍旧为0。

5. 快速处理时间戳选项

237-255 optlen等于首部中TCP选项的长度，optp是指向第一个选项字节的指针。如果下列3个条件均为真，说明只存在时间戳选项，而且格式正确：

1) TCP选项长度等于12(TCPOLEN_TSTAMP_APPA)；或TCP选项长度大于12，但optp[12]等于选项结束字节。

2) 选项的头4个字节等于0x0101080a(TCPOPT_TSTAMP_HDR，在26.6节曾讨论过)。

3) SYN标志未置位(说明连接已建立，如果报文段中出现时间戳选项，意味着连接双方都同意使用这一选项)。

如果上述条件全部满足，则ts_present置为1；从接收报文段首部获取两个时间戳值，分别赋给ts_val和ts_ecr；optp置为空，因为所有选项已处理完毕。这种辨认时间戳的方法可以避免调用通用选项处理函数tcp_dooptions，从而使后者能够专门处理只出现在SYN报文段中的各种选项(MSS和窗口大小选项)。如果连接双方同意使用时间戳，那么在建立的连接上交换的几乎所有报文段中都可能带有时间戳选项，因此，必须加快其处理速度。

图28-5给出了函数下一部分的代码，寻找报文段的Internet PCB。

6. 保存输入标志，把字段转换为主机字节序

257-264 接收报文段中的标志(SYN、FIN等)被保存在本地变量tiflags中，因为函数在处理过程中会多次引用这些标志。TCP首部的两个16 bit字段和两个32 bit序号被转换回主机字

节序，而两个16 bit端口号则不做处理，依旧为网络字节序，因为 Internet PCB中的端口号是依照网络字节序存储的。

```
tcp_input.c
257     tiflags = ti->ti_flags;
258     /*
259      * Convert TCP protocol specific fields to host format.
260      */
261     NTOHL(ti->ti_seq);
262     NTOHL(ti->ti_ack);
263     NTOHS(ti->ti_win);
264     NTOHS(ti->ti_urp);
265     /*
266      * Locate pcb for segment.
267      */
268   findpcb:
269     inp = tcp_last_inpcb;
270     if (inp->inp_lport != ti->ti_dport ||
271         inp->inp_fport != ti->ti_sport ||
272         inp->inp_faddr.s_addr != ti->ti_src.s_addr ||
273         inp->inp_laddr.s_addr != ti->ti_dst.s_addr) {
274         inp = in_pcblookup(&tcb, ti->ti_src, ti->ti_sport,
275                            ti->ti_dst, ti->ti_dport, INPLOOKUP_WILDCARD);
276         if (inp)
277             tcp_last_inpcb = inp;
278         ++tcpstat.tcps_pcbcachemiss;
279     }
tcp_input.c
```

图28-5 tcp_input 函数：寻找报文段的Internet PCB

7. 寻找Internet PCB

265-279 TCP的缓存(tcp_last_inpcb)中保存了收到的最后一个报文段的PCB地址，采用的技术与UDP相同。TCP使用一对插口来识别连接，寻找PCB时插口对中4个元素的比较次序与udp_input相同。如果与TCP缓存中的记录不匹配，则调用 in_pcblookup，把新的PCB放入缓存。

TCP中不会出现我们在 UDP中曾遇到过的问题：由于高速缓存中存在通配项 (wildcard entry)，导致匹配成功率很低。因为只有处于监听状态的服务器，才可能在其插口中保存通配项。连接一旦建立，插口对的4个元素将全部填入确定值。从图 24-5可知，高速缓存命中率能够达到80%。

图28-6给出了函数下一部分的代码。

```
tcp_input.c
280     /*
281      * If the state is CLOSED (i.e., TCB does not exist) then
282      * all data in the incoming segment is discarded.
283      * If the TCB exists but is in CLOSED state, it is embryonic,
284      * but should either do a listen or a connect soon.
285      */
286     if (inp == 0)
287         goto dropwithreset;
288     tp = intotcpcb(inp);
```

图28-6 tcp_input 函数：判断是否应丢弃报文段

```
289     if (tp == 0)
290         goto dropwithreset;
291     if (tp->t_state == TCPS_CLOSED)
292         goto drop;
293     /* Unscale the window into a 32-bit value. */
294     if ((tiflags & TH_SYN) == 0)
295         tiwin = ti->ti_win << tp->snd_scale;
296     else
297         tiwin = ti->ti_win;
```

tcp_input.c

图28-6 (续)

8. 丢弃报文段并生成RST

280-287 如果没有找到PCB，则丢弃输入报文段，并发送RST作为响应。例如，TCP收到了一个SYN，但报文段指定的服务器并不存在，则直接向对端发送 RST。回想一下，出现这种情况时UDP的处理方式，它将发送一个ICMP端口不可达差错。

288-290 如果PCB存在，但对应的TCP控制块不存在，可能插口已关闭(tcp_close释放TCP之后，才释放PCB)，则丢弃输入报文段，并发送RST作为响应。

9. 丢弃报文段且不发送响应

291-292 如果TCP控制块存在，但连接状态为CLOSED，说明插口已创建，且得到了本地地址和本地端口号，但还未调用connect或者listen。报文段被丢弃，且不发送任何响应。举例来说，如果客户向服务器发送的连接请求报文段到达时，服务器已调用了 bind，但还未调用listen。这种情况下，客户连接请求将超时，导致重传 SYN。

10. 不改变通告窗口大小

293-297 如果需要支持窗口大小选项，连接双方都必须在连接建立时通过窗口大小选项规定窗口缩放因子。如果报文段中包含 SYN，说明此时窗口缩放因子还未定义，因此，直接把TCP首部的窗口字段值复制给 tiwin；否则，首部中的16 bit数值应根据窗口缩放因子左移，得到32 bit的数值。

图28-7给出了函数的下一部分代码，如果选取了插口的 SO_DEBUG选项，或者插口正处于监听状态，则完成一些相应的预处理工作。

tcp_input.c

```
298     so = inp->inp_socket;
299     if (so->so_options & (SO_DEBUG | SO_ACCEPTCONN)) {
300         if (so->so_options & SO_DEBUG) {
301             ostate = tp->t_state;
302             tcp_saveti = *ti;
303         }
304         if (so->so_options & SO_ACCEPTCONN) {
305             so = sonewconn(so, 0);
306             if (so == 0)
307                 goto drop;
308             /*
309              * This is ugly, but ....
310              *
311              * Mark socket as temporary until we're
312              * committed to keeping it.  The code at
313              * 'drop' and 'dropwithreset' check the
```

图28-7 tcp_input 函数：处理调试选项和监听状态的插口

```
314                 * flag dropsocket to see if the temporary
315                 * socket created here should be discarded.
316                 * We mark the socket as discardable until
317                 * we're committed to it below in TCPS_LISTEN.
318                 */
319                dropsocket++;
320                inp = (struct inpcb *) so->so_pcb;
321                inp->inp_laddr = ti->ti_dst;
322                inp->inp_lport = ti->ti_dport;
323 #if BSD>=43
324                inp->inp_options = ip_srcroute();
325 #endif
326                tp = intotcpcb(inp);
327                tp->t_state = TCPS_LISTEN;

328                /* Compute proper scaling value from buffer space */
329                while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
330                       TCP_MAXWIN << tp->request_r_scale < so->so_rcv.sb_hiwat)
331                    tp->request_r_scale++;
332            }
333        }
```

tcp_input.c

图28-7 (续)

11. 如果选定了插口调试选项，则保存连接状态及IP和TCP首部

300-303 如果SO_DEBUG选项置位，则保存当前连接状态 (ostate)及IP和TCP首部(tcp_saveti)。函数结束时，这些信息将作为参数传给tcp_trace(图29-26)。

12. 如果监听插口收到了报文段，则创建新的插口

304-319 如果有报文段到达处于监听状态的插口(listen置位SO_ACCEPTCONN)，则调用sonewconn创建新的插口。发出PRU_ATTACH协议请求(图30-2)，分配Internet PCB和TCP控制块。在TCP最终接受连接请求之前，还需做更多的处理(如一个最基本的问题，报文段中是否包含SYN)，如果发现差错，置位dropsocket标志，控制跳转至标注“drop”和“dropwithreset”，丢弃新的插口。

320-326 inp和tp将指向新建的插口。本地地址和本地端口号直接从接收报文段TCP首部的目的地址和目的端口号字段中复制。如果输入数据报中有源选路的路由，TCP调用ip_srcroute，得到指向保存数据报源选路选项的mbuf的指针，并赋给inp_options。TCP向连接发送数据时，tcp_output会把源选路选项传给ip_output，使用与之相同的逆向路由。

327 新插口的状态设为LISTEN。如果接收报文段中包含SYN，控制将转到图28-16中的代码，完成连接建立请求的处理。

13. 计算窗口缩放因子

328-331 窗口缩放因子取决于接收缓存的大小。如果接收缓存大于通告窗口的最大值(65535，TCP_MAXWIN)，则左移65535，直到结果大于接收缓存大小，或者窗口缩放因子已等于最大值(14，TCP_MAX_WINSHIFT)。注意，窗口缩放因子的选取基于监听插口的接收缓存，也就是说，应用进程调用listen进入监听状态之前，应首先设定SO_RCVBUF插口选项，或者继承tcp_recvspace中的默认值。

窗口缩放因子最大值等于14，而65535×2^{14}等于1 073 725 440，已远远大于接收

缓存的最大值(Net/3中为2626 144)，因此，在窗口缩放因子远小于14时，循环即终止。参见习题28.1和28.2。

图28-8给出了TCP输入处理下一部分的代码。

tcp_input.c

```
334     /*
335      * Segment received on connection.
336      * Reset idle time and keepalive timer.
337      */
338     tp->t_idle = 0;
339     tp->t_timer[TCPT_KEEP] = tcp_keepidle;

340     /*
341      * Process options if not in LISTEN state,
342      * else do it below (after getting remote address).
343      */
344     if (optp && tp->t_state != TCPS_LISTEN)
345         tcp_dooptions(tp, optp, optlen, ti,
346                       &ts_present, &ts_val, &ts_ecr);
```

tcp_input.c

图28-8 tcp_inpu t函数：复位空闲时间和保活定时器，处理应用进程选项

14. 复位空闲时间和保活定时器

334-339 由于连接上收到了报文段，t_idle重设为0。保活定时器复位为2小时。

15. 如果不处于监听状态，处理TCP选项

340-346 如果TCP首部中有选项，并且连接状态不等于LISTEN，调用tcp_dooptions进行处理。前面介绍过，如果连接已建立，接收报文段中只存在时间戳选项，并且时间戳选项格式符合RFC 1323附录A的建议，这种情况在图28-4中已处理过，而且optp被置为空。如果插口处于监听状态，TCP把对端地址保存在PCB中之后，才会调用tcp_dooptions，这是因为处理MSS选项时需要了解到达对端的路由，具体代码如图28-17所示。

28.3 tcp_dooptions函数

函数处理Net/3支持的5个TCP选项(图26-4)：EOL、NOP、MSS、窗口大小和时间戳。图28-9给出了函数的第一部分。

tcp_input.c

```
1213 void
1214 tcp_dooptions(tp, cp, cnt, ti, ts_present, ts_val, ts_ecr)
1215 struct tcpcb *tp;
1216 u_char *cp;
1217 int     cnt;
1218 struct tcpiphdr *ti;
1219 int    *ts_present;
1220 u_long *ts_val, *ts_ecr;
1221 {
1222     u_short mss;
1223     int     opt, optlen;

1224     for (; cnt > 0; cnt -= optlen, cp += optlen) {
1225         opt = cp[0];
1226         if (opt == TCPOPT_EOL)
```

图28-9 tcp_dooptions 函数：处理EOL和NOP选项

```
1227            break;
1228        if (opt == TCPOPT_NOP)
1229            optlen = 1;
1230        else {
1231            optlen = cp[1];
1232            if (optlen <= 0)
1233                break;
1234        }
1235        switch (opt) {

1236        default:
1237            continue;
```

tcp_input.c

图28-9 (续)

1. 获取选项类型的长度

1213-1229 代码遍历TCP首部选项，遇到EOL(选项终止)时终止循环，函数返回；遇到NOP时，将其长度置为1，因为它后面不带长度字段（图26-16），控制转到switch语句的default子句，对其不做处理。

1230-1234 所有其他选项的长度保存在optlen中。

所有新增的Net/3不支持的TCP选项都被忽略。这是因为：

1) 将来定义的所有新选项都将带有长度字段(NOP和EOL是仅有的两个不带长度字段的选项)，而for语句的每次循环都跳过optlen字节。

2) switch语句的default子句忽略所有未知选项。

图28-10给出了tcp_dooptions最后一部分的代码，处理MSS、窗口大小和时间戳选项。

2. MSS选项

1238-1246 如果长度不等于4(TCPOLEN_MAXSEG)，或者报文段不带SYN标志，则忽略该选项。否则，复制两个MSS字节到本地变量，转换为主机字节序，调用tcp_mss完成处理。tcp_mss负责更新TCP控制块中的变量t_maxseg，即发向对端的报文段中允许携带的最大字节数。

3. 窗口大小选项

1247-1254 如果长度不等于4(TCPOLEN_WINDOW)，或者报文段不带SYN标志，则忽略该选项。Net/3置位TF_RCVD_SCALE，说明收到了一个窗口大小选项请求，并在requested_s_scale中保存缩放因子。由于cp[2]只有一个字节，因此，不存在边界问题。当连接转移到ESTABLISHED状态时，如果连接双方都同意支持窗口大小选项，则使用这一功能。

4. 时间戳选项

1255-1273 如果长度不等于10(TCPOLEN_TIMESTAMP)，则忽略该选项。否则，ts_present指向的标志被置位1，两个时间戳值分别保存在ts_val和ts_ecr所指向的变量中。如果收到的报文段带有SYN标志，Net/3置位TF_RCVD_TSTMP，说明收到了一个时间戳请求。ts_recent等于收到的时间戳值，ts_recent_age等于tcp_now，从系统初启到目前的时间，以500ms滴答为单位。

tcp_input.c

```
1238            case TCPOPT_MAXSEG:
1239                if (optlen != TCPOLEN_MAXSEG)
1240                    continue;
1241                if (!(ti->ti_flags & TH_SYN))
1242                    continue;
1243                bcopy((char *) cp + 2, (char *) &mss, sizeof(mss));
1244                NTOHS(mss);
1245                (void) tcp_mss(tp, mss);    /* sets t_maxseg */
1246                break;

1247            case TCPOPT_WINDOW:
1248                if (optlen != TCPOLEN_WINDOW)
1249                    continue;
1250                if (!(ti->ti_flags & TH_SYN))
1251                    continue;
1252                tp->t_flags |= TF_RCVD_SCALE;
1253                tp->requested_s_scale = min(cp[2], TCP_MAX_WINSHIFT);
1254                break;

1255            case TCPOPT_TIMESTAMP:
1256                if (optlen != TCPOLEN_TIMESTAMP)
1257                    continue;
1258                *ts_present = 1;
1259                bcopy((char *) cp + 2, (char *) ts_val, sizeof(*ts_val));
1260                NTOHL(*ts_val);
1261                bcopy((char *) cp + 6, (char *) ts_ecr, sizeof(*ts_ecr));
1262                NTOHL(*ts_ecr);

1263                /*
1264                 * A timestamp received in a SYN makes
1265                 * it ok to send timestamp requests and replies.
1266                 */
1267                if (ti->ti_flags & TH_SYN) {
1268                    tp->t_flags |= TF_RCVD_TSTMP;
1269                    tp->ts_recent = *ts_val;
1270                    tp->ts_recent_age = tcp_now;
1271                }
1272                break;
1273            }
1274        }
1275    }
```

tcp_input.c

图28-10 tcp_dooptions 函数：处理MSS、窗口大小选项和时间戳选项

28.4 首部预测

下面接着图28-8中的代码，继续介绍tcp_input函数。

首部预测最初由Van Jacobson提出，出现在4.3BSD Reno版中。除了下面要讨论的代码外，只有[Jacobson 1990b]还介绍过该算法，核心内容来自3张给出实现代码的幻灯片。

首部预测算法通过处理两种常见现象，简化单向数据传输的实现。

1) 如果TCP发送数据，连接上等待接收的下一个报文段是对已发送数据的 ACK。

2) 如果TCP接收数据，连接上等待接收的下一个报文段是顺序到达的数据报文段。

两种情况下，通过若干测试，判定收到的报文段是否是等待接收的报文段。如果是，则立即处理，比起本章接下来和下一章介绍的通用处理要快得多。

[Partridge 1993]介绍了一种基于Van Jacobson的研究成果，速度更快的TCP首部预测算法实现。

图28-11给出了首部预测的第一部分代码。

tcp_input.c

```
347     /*
348      * Header prediction: check for the two common cases
349      * of a uni-directional data xfer.  If the packet has
350      * no control flags, is in-sequence, the window didn't
351      * change and we're not retransmitting, it's a
352      * candidate.  If the length is zero and the ack moved
353      * forward, we're the sender side of the xfer.  Just
354      * free the data acked & wake any higher-level process
355      * that was blocked waiting for space.  If the length
356      * is non-zero and the ack didn't move, we're the
357      * receiver side.  If we're getting packets in order
358      * (the reassembly queue is empty), add the data to
359      * the socket buffer and note that we need a delayed ack.
360      */
361     if (tp->t_state == TCPS_ESTABLISHED &&
362      (tiflags & (TH_SYN | TH_FIN | TH_RST | TH_URG | TH_ACK)) == TH_ACK &&
363         (!ts_present || TSTMP_GEQ(ts_val, tp->ts_recent)) &&
364         ti->ti_seq == tp->rcv_nxt &&
365         tiwin && tiwin == tp->snd_wnd &&
366         tp->snd_nxt == tp->snd_max) {

367         /*
368          * If last ACK falls within this segment's sequence numbers,
369          *  record the timestamp.
370          */
371         if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
372             SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len)) {
373             tp->ts_recent_age = tcp_now;
374             tp->ts_recent = ts_val;
375         }
```

tcp_input.c

图28-11 tcp_input 函数：首部预测，第一部分

1. 判定收到的报文段是否是等待接收的报文段

347-366 下列6个条件必须全真，才能说明收到的报文段是连接正等待接收的数据报文段或ACK报文段：

1) 连接状态等于ESTABLISHED。

2) 下列4个控制标志必须不设定：SYN、FIN、RST、或URG。但ACK标志必须置位。换言之，TCP的6个控制标志中，ACK标志必须置位，前面列出的4个标志必须清除，PSH标志置位与否无关紧要(连接处于ESTABLISHED状态时，除非RST标志置位，一般情况下，ACK都会置位)。

3) 如果报文段带有时间戳选项，则最新时间戳值(ts_val)必须大于或等于连接上以前收到的时间戳值(ts_recent)。本质上说，这就是PAWS测试，28.7节将详细介绍PAWS。如果ts_val小于ts_recent，则新报文段是乱序报文段，因为它的发送时间早于连接上收到的上一个报文段。由于对端通常把时钟值填充到时间戳字段(Net/3的全局变量tcp_now)，收到的时间戳正常情况下应该是一个单调递增的序列。

并非每个顺序到达报文段中的时间戳都会增加。事实上，Net/3系统每500 ms增加一次时

钟值(tcp_now)，在这段时间间隔中，完全可能发送多个报文段。假定利用时间戳和序号构成一个64 bit的数值，序号放在低32 bit，时间戳放在高32 bit，对于每个顺序报文段，这个64 bit的值都至少会增加1(应考虑取模算法)。

4) 报文段的起始序号(ti_seq)必须等于连接上等待接收的下一个序号(rcv_nxt)。如果这个条件为假，那么收到的报文段是重传报文段或是乱序报文段。

5) 报文段通告的窗口大小必须非零(tiwin)，必须等于当前发送窗口(snd_wnd)。也就是说，无需更新当前发送窗口。

6) 下一个发送序号(snd_nxt)必须等于已发送的最大序号(snd_max)，也就是说，上一个发送报文段不是重传报文段。

2. 根据接收的时间戳更新ts_recent

367-375 如果存在时间戳选项，并且时间戳值满足图26-18中的测试条件，则把收到的时间戳值(ts_val)赋给ts_recent，并在ts_recent_age中保存当前时钟(tcp_now)。

前面讨论过图26-18中的时间戳有效性测试条件所存在的问题，并在图26-20中给出了正确的测试条件。但在首部预测算法的实现中，图26-20中的TSTMP_GEQ测试是多余的，因为图28-11起始处的if语句已完成了这一测试

图28-12给出了首部预测的下一部分代码，用于单向数据的发送方：处理输出数据的ACK。

tcp_input.c

```
376         if (ti->ti_len == 0) {
377             if (SEQ_GT(ti->ti_ack, tp->snd_una) &&
378                 SEQ_LEQ(ti->ti_ack, tp->snd_max) &&
379                 tp->snd_cwnd >= tp->snd_wnd) {
380                 /*
381                  * this is a pure ack for outstanding data.
382                  */
383                 ++tcpstat.tcps_predack;
384                 if (ts_present)
385                     tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
386                 else if (tp->t_rtt &&
387                           SEQ_GT(ti->ti_ack, tp->t_rtseq))
388                     tcp_xmit_timer(tp, tp->t_rtt);

389                 acked = ti->ti_ack - tp->snd_una;
390                 tcpstat.tcps_rcvackpack++;
391                 tcpstat.tcps_rcvackbyte += acked;
392                 sbdrop(&so->so_snd, acked);
393                 tp->snd_una = ti->ti_ack;
394                 m_freem(m);

395                 /*
396                  * If all outstanding data is acked, stop
397                  * retransmit timer, otherwise restart timer
398                  * using current (possibly backed-off) value.
399                  * If process is waiting for space,
400                  * wakeup/selwakeup/signal.  If data
401                  * is ready to send, let tcp_output
402                  * decide between more output or persist.
403                  */
404                 if (tp->snd_una == tp->snd_max)
405                     tp->t_timer[TCPT_REXMT] = 0;
```

图28-12 tcp_input 函数：首部预测，发送方处理

```
406                 else if (tp->t_timer[TCPT_PERSIST] == 0)
407                     tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;

408                 if (so->so_snd.sb_flags & SB_NOTIFY)
409                     sowwakeup(so);
410                 if (so->so_snd.sb_cc)
411                     (void) tcp_output(tp);
412                 return;
413             }
```

tcp_input.c

图28-12 (续)

3. 纯ACK测试

376-379 如果下列4个条件全真，则收到的是一个纯ACK报文段。

1) 报文段不携带数据(ti_len等于0)。

2) 报文段的确认字段(ti_ack)大于最大的未确认序号(snd_una)。由于测试条件是“大于”，而非“大于等于”，也就是要求收到的ACK必须确认未曾确认过的数据。

3) 报文段的确认字段(ti_ack)小于等于已发送的最大序号(snd_max)。

4) 拥塞窗口大于等于当前发送窗口(snd_wnd)，要求窗口完全打开，连接不处于慢起动或拥塞避免状态。

4. 更新RTT值

384-388 如果报文段携带时间戳选项，或者报文段中的确认序号大于某个计时报文段的起始序号，则调用tcp_xmit_timer更新RTT值。

5. 从发送缓存中删除被确认的字节

389-394 acked等于接收报文段确认字段所确认的字节数，调用sbdrop从发送缓存中删除这些字节。更新最大的未确认过的序号(snd_una)为报文段的确认字段值，释放保存接收报文段的mbuf链表(由于数据长度等于0，实际只有一个保存首部的mbuf)。

6. 终止重传定时器

395-407 如果接收报文段确认了所有已发送数据(snd_una等于snd_max)，则关闭重传定时器。若条件不满足，且持续定时器未设定，则重启重传定时器，时限设为t_rxtcur。

前面介绍过，tcp_output发送报文段时，只有重传定时器未启动，才会设定它。如果连续发送两个报文段，发送第一个报文段时定时器启动，发送第二个报文段时定时器不变。但如果只收到第一个报文段的确认，则重传定时器必须重启，防止第二个报文段丢失。

7. 唤醒等待进程

408-409 如果发送缓存修改后，有必要唤醒等待的应用进程，则调用sowwakeup。从图16-5可知，如果有应用进程正在等待缓存空间，或者设定了与缓存有关的select选项，或者正等待插口上的SIGIO，则SB_NOTIFY为真。

8. 生成更多的输出

410-411 如果发送缓存中有数据，则调用tcp_output，因为发送窗口已经向右移动。snd_una已增加，但snd_wnd未变化，因此，图24-17中的整个窗口将向右移动。

图28-13给出了首部预测的下一部分代码，接收方收到顺序到达的数据时进行的各种处理。

9. 测试收到报文段是否是连接等待接收的下一个报文段

414-416 如果下列4个条件均为真，则收到的报文段是连接上等待接收的下一报文段，并且插口缓存中的剩余空间能够容纳到达的数据。

1) 报文段的数据量(ti_len)大于0，即图28-12起始处if语句的else子句。

2) 确认字段(ti_ack)等于最大的未确认序号，即报文段未确认任何数据。

3) 连接乱序报文段的重组队列为空(seq_next等于tp)。

4) 接收缓存能够容纳报文段数据。

10. 完成接收数据的处理

423-435 等待接收序号(rcv_nxt)递增收到的数据字节数。从mbuf链中丢弃IP首部、TCP首部和所有TCP选项，将剩余的mbuf链附加到插口的接收缓存，调用sorwakeup唤醒接收进程。注意，代码没有调用TCP_REASS宏，因为宏代码中的条件判定已经包含在首部预测的测试条件中。设定延迟ACK标志，输入处理结束。

tcp_input.c

```
414         } else if (ti->ti_ack == tp->snd_una &&
415                    tp->seg_next == (struct tcpiphdr *) tp &&
416                    ti->ti_len <= sbspace(&so->so_rcv)) {
417             /*
418              * this is a pure, in-sequence data packet
419              * with nothing on the reassembly queue and
420              * we have enough buffer space to take it.
421              */
422             ++tcpstat.tcps_preddat;
423             tp->rcv_nxt += ti->ti_len;
424             tcpstat.tcps_rcvpack++;
425             tcpstat.tcps_rcvbyte += ti->ti_len;
426             /*
427              * Drop TCP, IP headers and TCP options then add data
428              * to socket buffer.
429              */
430             m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
431             m->m_len -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
432             sbappend(&so->so_rcv, m);
433             sorwakeup(so);
434             tp->t_flags |= TF_DELACK;
435             return;
436         }
437     }
```

tcp_input.c

图28-13 tcp_input 函数：首部预测的接收方处理

统计量

首部预测能在多大程度上改善系统性能？让我们做个简单的实验，跨越 LAN(bdsi和svr4间的双向通信)的数据传输，和跨越 WAN(vangogh.cs.berkeley.edu和ftp.uu.net之间的双向通信)的数据传输。运行netstat，得到类似于图24-5的输出，列出了两种情况下的首部预测寄存器的值。

跨越LAN传输时，无数据分组丢失，只有一些重复的 ACK。利用首部预测处理的报文段可占到97%~100%。跨越WAN时，比例有所降低，约为83%~99%之间。

请注意，首部预测的应用限定于单独的连接，无论主机是否收到了额外的 TCP流量，PCB

缓存必须在主机范围内共享。即使 TCP流量的丢失造成了 PCB缓存缺失，但如果给定连接上的数据分组未丢失，这条连接上的首部预测仍能工作。

28.5 TCP输入：缓慢的执行路径

下面讨论首部预测失败时的处理代码，`tcp_input`中较慢的一条执行路径。图 28-14给出了下一部分代码，为输入报文段的处理完成一些准备工作。

tcp_input.c

```
438     /*
439      * Drop TCP, IP headers and TCP options.
440      */
441     m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
442     m->m_len -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);

443     /*
444      * Calculate amount of space in receive window,
445      * and then do TCP input processing.
446      * Receive window is amount of space in rcv queue,
447      * but not less than advertised window.
448      */
449     {
450         int     win;

451         win = sbspace(&so->so_rcv);
452         if (win < 0)
453             win = 0;
454         tp->rcv_wnd = max(win, (int) (tp->rcv_adv - tp->rcv_nxt));
455     }
```

tcp_input.c

图28-14 tcp_input 函数：丢弃IP和TCP首部

1. 丢弃IP和TCP首部，包括TCP选项

438-442 更新数据指针和mbuf链表中的第一个mbuf的长度，以跳过IP首部、TCP首部和所有TCP选项。因为`off`等于TCP首部长度，包括TCP选项，因此，表达式中减去了标准 TCP首部的大小(20字节)。

2. 计算接收窗口

443-455 `win`等于插口接收缓存中可用的字节数，`rcv_adv`减去`rcv_nxt`等于当前通告的窗口大小，接收窗口等于上述两个值中较大的一个，这是为了保证接收窗口不小于当前通告窗口的大小。此外，如果最后一次窗口更新后，应用进程从插口接收缓存中取走了数据，`win`可能大于通告窗口，因此，TCP最多能够接收`win`字节的数据(即使对端不会发送超过通告窗口大小的数据)。

因为函数后面的代码必须确定通告窗口中能放入多少数据 (如果有)，所以现在必须计算通告窗口的大小。落在通告窗口之外的接收数据被丢弃：落在窗口左侧的数据是已接收并确认过的数据，落在窗口右侧的数据是暂不允许对端发送的数据。

28.6 完成被动打开或主动打开

如果连接状态等于LISTEN或者SYN_SENT，则执行本节给出的代码。连接处于这两个状态时，等待接收的报文段为SYN，任何其他报文段将被丢弃。

28.6.1 完成被动打开

连接状态等于LISTEN时，执行图28-15中的代码，其中变量tp和inp指向图28-7所创建的新的插口，而非服务器的监听插口。

tcp_input.c

```
456     switch (tp->t_state) {

457         /*
458          * If the state is LISTEN then ignore segment if it contains an RST.
459          * If the segment contains an ACK then it is bad and send an RST.
460          * If it does not contain a SYN then it is not interesting; drop it.
461          * Don't bother responding if the destination was a broadcast.
462          * Otherwise initialize tp->rcv_nxt, and tp->irs, select an initial
463          * tp->iss, and send a segment:
464          *     <SEQ=ISS><ACK=RCV_NXT><CTL=SYN,ACK>
465          * Also initialize tp->snd_nxt to tp->iss+1 and tp->snd_una to tp->iss
466          * Fill in remote peer address fields if not previously specified.
467          * Enter SYN_RECEIVED state, and process any other fields of this
468          * segment in this state.
469          */
470     case TCPS_LISTEN:{
471             struct mbuf *am;
472             struct sockaddr_in *sin;

473             if (tiflags & TH_RST)
474                 goto drop;
475             if (tiflags & TH_ACK)
476                 goto dropwithreset;
477             if ((tiflags & TH_SYN) == 0)
478                 goto drop;
```

tcp_input.c

图28-15 tcp_input 函数：检测监听插口上是否收到了SYN

1. 丢弃RST、ACK或非SYN

473-478 如果接收报文段中带有RST标志，则丢弃它。如果带有ACK，则丢弃它并发送RST作为响应(建立连接的最初的SYN报文段是少数几个不允许携带ACK的报文段之一)。如果未带有SYN，则丢弃它。case子句的后续代码处理连接处于LISTEN状态时收到了SYN的状况。新的连接状态等于SYN_RCVD。

图28-16给出了case语句接下来的代码。

tcp_input.c

```
479             /*
480              * RFC1122 4.2.3.10, p. 104: discard bcast/mcast SYN
481              * in_broadcast() should never return true on a received
482              * packet with M_BCAST not set.
483              */
484             if (m->m_flags & (M_BCAST | M_MCAST) ||
485                 IN_MULTICAST(ti->ti_dst.s_addr))
486                 goto drop;

487             am = m_get(M_DONTWAIT, MT_SONAME);  /* XXX */
488             if (am == NULL)
489                 goto drop;
490             am->m_len = sizeof(struct sockaddr_in);
```

图28-16 tcp_input 函数：处理监听插口上收到的SYN报文段

```
491             sin = mtod(am, struct sockaddr_in *);
492             sin->sin_family = AF_INET;
493             sin->sin_len = sizeof(*sin);
494             sin->sin_addr = ti->ti_src;
495             sin->sin_port = ti->ti_sport;
496             bzero((caddr_t) sin->sin_zero, sizeof(sin->sin_zero));
497             laddr = inp->inp_laddr;
498             if (inp->inp_laddr.s_addr == INADDR_ANY)
499                 inp->inp_laddr = ti->ti_dst;
500             if (in_pcbconnect(inp, am)) {
501                 inp->inp_laddr = laddr;
502                 (void) m_free(am);
503                 goto drop;
504             }
505             (void) m_free(am);
```

tcp_input.c

图28-16 (续)

2. 如果是广播报文段或多播报文段，则丢弃它

479-486 如果数据报被发送到广播地址或多播地址，则丢弃它， TCP只支持点到点的应用。前面介绍过，根据数据帧携带的目的硬件地址， ether_input置位M_BCAST和M_MCAST标志， IN_MULTICAST宏可判定IP地址是否为D类地址。

注释引用了in_broadcast，因为Net/1代码(它不支持多播)在此处调用了这个函数，以检测目的IP地址是否为广播地址。 Net/2中改为根据目的硬件地址，通过ether_input设定M_BCAST和M_MCAST标志。

Net/3只测试目的硬件地址是否为广播地址，而且不调用 in_broadcast测试目的IP地址是否为广播地址。它假定除非目的硬件地址是广播地址，否则，目的 IP地址绝不可能是广播地址，从而避免调用 in_broadcast。另外，如果Net/3真的收到了一个数据帧，其目的硬件地址为单播地址，而目的 IP地址为广播地址，将执行图 28-16中的代码处理此种报文段。

目的地址参数IN_MULTICAST需要被转换为主机字节序。

3. 为客户端的IP地址和端口号分配mbuf

487-496 分配一个mbuf，保存sockaddr_in结构，其中带有客户端的 IP地址和端口号。IP地址从IP首部的源地址字段中复制，端口号从 TCP首部的源端口号字段中复制。这个结构用于把服务器的PCB连到客户，之后mbuf被释放。

注释中的“ XXX ”，是因为获取mbuf的消耗等同于之后调用in_pcbconnect的消耗。不过此处的代码位于 tcp_input中较慢的一条执行路径。从图 24-5可知，不足%2的接收报文段的处理中会用到这段处理代码。

4. 设定PCB中的本地地址

497-499 laddr是绑定在插口上的本地地址。如果服务器没有为插口绑定一个确定地址 (正常情况下)，IP首部的目的地址将成为 PCB中的本地地址。注意，不管数据报是在哪个端口收到的，都将保存IP首部中的目的地址。

注意，laddr不会是通配地址，因为图 28-7中的代码已将收到报文段中的目的 IP

地址赋给了它。

5. 填充PCB中的对端地址

500-505 调用in_pcbconnect，把服务器的PCB与客户相连，填充PCB中的对端地址和对端端口号。之后，释放mbuf。

图28-17给出了函数下一部分的代码，结束case语句的处理。

tcp_input.c

```
506             tp->t_template = tcp_template(tp);
507             if (tp->t_template == 0) {
508                 tp = tcp_drop(tp, ENOBUFS);
509                 dropsocket = 0; /* socket is already gone */
510                 goto drop;
511             }
512             if (optp)
513                 tcp_dooptions(tp, optp, optlen, ti,
514                               &ts_present, &ts_val, &ts_ecr);
515             if (iss)
516                 tp->iss = iss;
517             else
518                 tp->iss = tcp_iss;
519             tcp_iss += TCP_ISSINCR / 2;
520             tp->irs = ti->ti_seq;
521             tcp_sendseqinit(tp);
522             tcp_rcvseqinit(tp);
523             tp->t_flags |= TF_ACKNOW;
524             tp->t_state = TCPS_SYN_RECEIVED;
525             tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;
526             dropsocket = 0;     /* committed to socket */
527             tcpstat.tcps_accepts++;
528             goto trimthenstep6;
529         }
```

tcp_input.c

图28-17 tcp_input 函数：完成LISTEN状态下收到SYN报文段的处理

6. 分配并初始化IP和TCP首部模板

506-511 调用tcp_template创建IP和TCP首部的模板。图28-7中调用sonewconn时，为新连接分配了PCB和TCP控制块，但未分配首部模板。

7. 处理所有的TCP选项

512-514 如果存在TCP选项，则调用tcp_dooptions进行处理。图28-8中曾调用过一次tcp_dooptions，但只处理非LISTEN状态时的TCP选项。现在，插口处于监听状态， PCB中的对端地址已填入(tcp_mss函数中会用到对端地址)，调用tcp_dooptions：获取到达对端的路由；查看对端主机是本地结点还是远端结点 (选择MSS时，需考虑到对端的网络ID和子网ID)。

8. 初始化ISS

515-519 通常情况下，初始发送序号复制自全局变量 tcp_iss，之后增加 64 000 (TCP_ISSINCR除以2)。如果局部变量iss非零，则使用iss取代tcp_iss，初始化连接的发送序号。

出现以下事件序列时，会用到iss：

• 服务器的IP地址为128.1.2.3，端口号为27。

- IP地址等于192.3.4.5的客户与前述服务器建立了连接，客户端口号等于 3000。服务器的插口对为{128.1.2.3, 27, 192.3.4.5, 3000}。
- 服务器主动关闭了连接，上述插口对的状态转移到 TIME_WAIT。连接处于这种状态时，最后收到的序号保存在TCP控制块中。假设序号等于100 000。
- 连接离开TIME_WAIT状态之前，收到来自于同一客户主机、同一端口号 (192.3.4.5, 3000)的新的SYN，TCP寻找处于TIME_WAIT状态的连接所对应的PCB，而不是监听服务器的PCB。假定新SYN报文段的序号等于200 000。
- 因为连接状态不等于LISTEN，所以将不执行刚讨论过的图28-17中的代码，而是执行图28-28中的代码。我们将看到，其中包含了下列处理逻辑：如果新 SYN报文段的序号(200 000)大于客户最后发来的序号(100 000)，那么：(1)局部变量iss等于100 000加上128 000；(2)处于TIME_WAIT状态的连接被完全关闭(PCB和TCP控制块被删除)；(3)控制跳转到findpcb(图28-5)。
- 寻找服务器监听插口的PCB(假定监听服务器还在运行)，执行本节中介绍的代码。图28-17中的代码将使用局部变量iss(现在等于228 000)初始化新连接的tcp_iss。

RFC 1122中定义的这种处理逻辑，允许同一个客户和服务器重用同样的插口连接对，只要服务器主动关闭原有连接。它也解释了为什么只要有进程调用 connect，全局变量tcp_iss就递增64 000(图30-4)：为了确保在某个客户不断地重建与同一个服务器的连接的情况下，即使前一次连接上没有传输数据，甚至 500 ms定时器都未超时(定时器超时处理代码会增加tcp_iss)，新建连接仍可以使用较大的ISS。

9. 初始化控制块中的序号变量

520-522 图28-17中，初始接收序号复制自SYN报文段中的序号字段(irs)。下面两个宏初始化了TCP控制块中的相关变量。

```
#define tcp_rcvseqinit(tp) \
     (tp)->rcv_adv = (tp)->rcv_nxt = (tp)->irs + 1

#define tcp_sendseqinit(tp) \
     (tp)->snd_una = (tp)->snd_nxt = (tp)->snd_max = (tp)->snd_up = \
          (tp)->iss
```

因为SYN占据一个序号，所以第一个宏表达式需加1。

10. 确认SYN并更新状态

523-525 因为对于SYN的确认必须立即发送，所以置位TF_ACKNOW标志。连接状态转移到SYN_RCVD，连接建立定时器设为75秒(TCPTV_KEEP_INIT)。因为TF_ACKNOW置位，函数结束时将调用tcp_output。从图24-16可知，此种tcp_outflags会导致发送携带SYN和ACK的报文段。

526-528 现在，TCP结束了从图28-7开始的新插口的创建，drop插口标志被清除。控制跳转到trimthenstep6处，完成SYN报文段的处理。前面介绍过，SYN报文段能够携带数据，尽管只有等连接进入ESTABLISHED状态后，数据才会被提交给应用程序。

28.6.2 完成主动打开

图28-18给出了连接进入SYN_SENT状态后，处理代码的第一部分。TCP等待接收SYN。

tcp_input.c

```
530         /*
531          * If the state is SYN_SENT:
532          *  if seg contains an ACK, but not for our SYN, drop the input.
533          *  if seg contains an RST, then drop the connection.
534          *  if seg does not contain SYN, then drop it.
535          * Otherwise this is an acceptable SYN segment
536          *  initialize tp->rcv_nxt and tp->irs
537          *  if seg contains ack then advance tp->snd_una
538          *  if SYN has been acked change to ESTABLISHED else SYN_RCVD state
539          *  arrange for segment to be acked (eventually)
540          *  continue processing rest of data/controls, beginning with URG
541          */
542     case TCPS_SYN_SENT:
543         if ((tiflags & TH_ACK) &&
544             (SEQ_LEQ(ti->ti_ack, tp->iss) ||
545              SEQ_GT(ti->ti_ack, tp->snd_max)))
546             goto dropwithreset;
547         if (tiflags & TH_RST) {
548             if (tiflags & TH_ACK)
549                 tp = tcp_drop(tp, ECONNREFUSED);
550             goto drop;
551         }
552         if ((tiflags & TH_SYN) == 0)
553             goto drop;
```

tcp_input.c

图28-18 tcp_input 函数：判定收到的SYN是否是所需的响应

1. 验证收到的ACK

530-546 当应用进程主动打开，TCP发送SYN时，从图30-4可知，连接的iss将等于全局变量tcp_iss，宏tcp_sendseqinit(前一节结尾给出了定义)被执行。假设ISS等于365，图28-19给出了tcp_output发送SYN后的发送序号变量。

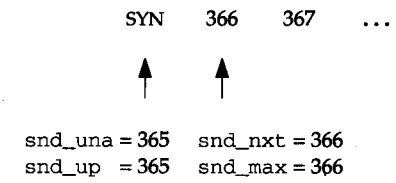

图28-19 ISS等于365的SYN发送后的发送序号变量

tcp_sendseqinit初始化图28-19中的4个变量为365，接着图26-31中的代码在发送SYN之后，把其中两个增至366。因此，如果图28-18中的接收报文段包含ACK，并且确认字段小于等于iss(365)，或者大于snd_max(366)，ACK无效，丢弃报文段并发送RST作为响应。注意，连接处于SYN_SENT状态时，收到的报文段中无需携带ACK。它可以只包括SYN，这种情况称为同时打开(simultaneous open)(图24-15)。

2. 处理并丢弃RST报文段

547-551 如果接收报文段中带有RST，则丢弃它。但首先应查看ACK标志，因为如果报文段同时携带了有效的ACK(已验证过)和RST，则说明对端拒绝本次连接请求，通常是因为服务器进程未运行。这种情况下，tcp_drop设定插口的so_error变量，并向调用connect的应用进程返回差错。

3. 判定SYN标志是否置位

552-553 如果收到报文段中的SYN标志未置位，则丢弃它。

这个case语句的其余代码用于处理本地发送连接请求后，收到对端响应的SYN报文段(及可选的ACK)的情况。图28-20给出了tcp_input下一部分的代码，继续处理SYN。

tcp_input.c

```
554         if (tiflags & TH_ACK) {
555             tp->snd_una = ti->ti_ack;
556             if (SEQ_LT(tp->snd_nxt, tp->snd_una))
557                 tp->snd_nxt = tp->snd_una;
558         }
559         tp->t_timer[TCPT_REXMT] = 0;
560         tp->irs = ti->ti_seq;
561         tcp_rcvseqinit(tp);
562         tp->t_flags |= TF_ACKNOW;
563         if (tiflags & TH_ACK && SEQ_GT(tp->snd_una, tp->iss)) {
564             tcpstat.tcps_connects++;
565             soisconnected(so);
566             tp->t_state = TCPS_ESTABLISHED;
567             /* Do window scaling on this connection? */
568             if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
569                 (TF_RCVD_SCALE | TF_REQ_SCALE)) {
570                 tp->snd_scale = tp->requested_s_scale;
571                 tp->rcv_scale = tp->request_r_scale;
572             }
573             (void) tcp_reass(tp, (struct tcpiphdr *) 0,
574                              (struct mbuf *) 0);
575             /*
576              * if we didn't have to retransmit the SYN,
577              * use its rtt as our initial srtt & rtt var.
578              */
579             if (tp->t_rtt)
580                 tcp_xmit_timer(tp, tp->t_rtt);
581         } else
582             tp->t_state = TCPS_SYN_RECEIVED;
```

tcp_input.c

图28-20 tcp_input 函数：发送连接请求后，收到对端响应的SYN

4. 处理ACK

554-558 如果报文段中有ACK，令snd_una等于报文段的确认字段。以图28-19为例，snd_una应更新为366，因为确认字段惟一有效的值就是366。如果snd_nxt小于snd_una(在图28-19的例子中不可能发生)，令snd_nxt等于snd_una。

5. 关闭连接建立定时器

559 连接建立定时器被关闭。

此处代码有错误。连接建立定时器只有在ACK标志置位时才能被关闭，因为收到一个不带ACK的SYN报文段，只是说明连接双方同时打开，而不意味着对端已收到了SYN。

6. 初始化接收序号

560-562 初始接收序号从接收报文段的序号字段中复制。tcp_rcvseqinit宏(上一节结束时给出了定义)初始化rcv_adv和rcv_nxt为接收序号加1。置位TF_ACKNOW标志，从而在函数结尾处调用tcp_output，发送报文段携带的确认字段应等于rcv_nxt(图26-27)，确认刚收到的SYN。

563-564 如果接收报文段带有ACK，并且snd_una大于连接的ISS，主动打开处理完毕，连接进入ESTABLISHED状态。

第二个测试条件其实是多余的。图28-20起始处，如果ACK标志置位，snd_una将等于接收报文段的确认字段值。另外，图28-18中紧跟着case语句的if语句验证了收到的确认字段大于ISS。所以，此处只要ACK置位，就可以确保snd_una大于ISS。

7. 连接建立

565-566 soisconnected设定插口进入连接状态，TCP连接的状态转移到ESTABLISHED。

8. 查看窗口大小选项

567-572 如果TCP在本地SYN中加入窗口大小选项，并且收到的SYN中也包含了这一选项，使用窗口缩放功能，设定snd_scale和rcv_scale。因为tcp_newtcpcb初始化TCP控制块为0，所以，如果不使用窗口大小选项，这两个变量的默认值为0。

9. 向应用进程提交队列中的数据

573-574 由于数据可能在连接未建立之前到达，调用tcp_reass把数据放入接收缓存，第二个参数为空。

测试条件其实不必要的。因为TCP刚收到带有ACK的SYN报文段，状态从SYN_SENT转移到ESTABLISHED。即使有数据出现在SYN中，也会被暂时搁置，直到函数快结束，控制转到dodata标注时才会被处理。如果TCP收到不带ACK的SYN(同时打开)，即使报文段携带数据，也会被暂时搁置，等到收到了ACK，连接从SYN_RCVD转移到ESTABLISHED之后，才会被处理。

尽管SYN中可以携带数据，并且Net/3能够正确处理这样的报文段，但Net/3自己不会产生这样的报文段。

10. 更新RTT估计器值

575-580 如果确认的SYN正被计时，tcp_xmit_timer将根据得到的对SYN报文段的测量值初始化RTT估计器值。

TCP在此处忽略收到的时间戳选项，只查看t_rtt计数器。TCP主动打开时，在第一个SYN中加入时间戳选项(图26-24)，如果对端也同意采用时间戳，就会在它响应的SYN中回应收到的时间戳(参见图28-10，Net/3在SYN中回应收到的时间戳)。因此，TCP在此处可以使用收到的时间戳，而不用t_rtt，但因为两者的精度相同(500ms)，在这一点上时间戳并无优势可言。使用时间戳，而非t_rtt计数器的真正好处在于高速网络中同时发送大量数据时，能提供更多的RTT测量值和更好的估计器值(希望如此)。

11. 同时打开

581-582 如果TCP在SYN_SENT状态收到不带ACK的SYN，则称为同时打开，连接转移到SYN_RCVD状态。

图28-21给出了函数的下一部分代码，处理SYN中可能携带的数据。图28-17结尾处，代码跳转至trimthenstep6标注处，这里也有类似的情况。

tcp_input.c

```
583       trimthenstep6:
584         /*
585          * Advance ti->ti_seq to correspond to first data byte.
586          * If data, trim to stay within window,
587          * dropping FIN if necessary.
588          */
```

图28-21 tcp_input 函数：接收SYN的通用处理

```
589             ti->ti_seq++;
590             if (ti->ti_len > tp->rcv_wnd) {
591                 todrop = ti->ti_len - tp->rcv_wnd;
592                 m_adj(m, -todrop);
593                 ti->ti_len = tp->rcv_wnd;
594                 tiflags &= ~TH_FIN;
595                 tcpstat.tcps_rcvpackafterwin++;
596                 tcpstat.tcps_rcvbyteafterwin += todrop;
597             }
598             tp->snd_wl1 = ti->ti_seq - 1;
599             tp->rcv_up = ti->ti_seq;
600             goto step6;
601         }
```

tcp_input.c

图28-21 (续)

584-589 报文段序号加1，以计入SYN。如果SYN带有数据，ti_seq现在应等于数据第一个字节的序号。

12. 丢弃落在接收窗口外的数据

590-597 ti_len等于报文段中的数据字节数。如果它大于接收窗口，超出部分的数据(ti_len减去rcv_wnd)将被m_adj丢弃。函数参数为负值，所以，将从mbuf链尾部起逆向删除数据(图2-20)。更新ti_len，等于数据删除后mbuf中剩余的数据量。清除FIN标志，这是因为FIN可能跟在最后一个数据字节之后，落在接收窗口外而被丢弃。

如果SYN是对本地连接请求的响应，且携带的数据过多，则说明对端收到的SYN报文段中带有窗口通告，但对端忽略了通告的窗口大小，并禁止不规范的行为。但如果主动打开的SYN报文段中带有大量数据，则说明对端还未收到窗口通告，所以不得不猜测SYN中能够携带多少数据。

13. 强制更新窗口变量

598-599 snd_wl1等于接收序号减1。从图29-15中可看到，这将强制更新3个窗口变量：snd_wnd、snd_wl1和snd_wl2。接收紧急指针(rcv_up)等于接收序号。控制跳转到标注step6处，与RFC 793定义的步骤相对应，我们将在图29-15中详细讨论。

28.7 PAWS：防止序号回绕

图28-22给出了tcp_input下一部分的代码，处理可能出现的序号回绕：RFC 1323中定义的PAWS算法。请回想一下我们在26.6节关于时间戳的讨论。

tcp_input.c

```
602     /*
603      * States other than LISTEN or SYN_SENT.
604      * First check timestamp, if present.
605      * Then check that at least some bytes of segment are within
606      * receive window.  If segment begins before rcv_nxt,
607      * drop leading data (and SYN); if nothing left, just ack.
608      *
609      * RFC 1323 PAWS: If we have a timestamp reply on this segment
610      * and it's less than ts_recent, drop it.
611      */
612     if (ts_present && (tiflags & TH_RST) == 0 && tp->ts_recent &&
```

图28-22 tcp_input 函数：处理时间戳选项

```
613         TSTMP_LT(ts_val, tp->ts_recent)) {

614         /* Check to see if ts_recent is over 24 days old.  */
615         if ((int) (tcp_now - tp->ts_recent_age) > TCP_PAWS_IDLE) {
616             /*
617              * Invalidate ts_recent.  If this segment updates
618              * ts_recent, the age will be reset later and ts_recent
619              * will get a valid value.  If it does not, setting
620              * ts_recent to zero will at least satisfy the
621              * requirement that zero be placed in the timestamp
622              * echo reply when ts_recent isn't valid.  The
623              * age isn't reset until we get a valid ts_recent
624              * because we don't want out-of-order segments to be
625              * dropped when ts_recent is old.
626              */
627             tp->ts_recent = 0;
628         } else {
629             tcpstat.tcps_rcvduppack++;
630             tcpstat.tcps_rcvdupbyte += ti->ti_len;
631             tcpstat.tcps_pawsdrop++;
632             goto dropafterack;
633         }
634     }
```

tcp_input.c

图28-22 (续)

1. 基本PAWS测试

602-613 如果存在时间戳，则调用tcp_dooptions设定ts_present。如果下列3个条件全真，则丢弃报文段：

1) RST标志未置位(参见习题28.8)。

2) TCP曾收到过对端发送的有效的时间戳(ts_recent非零)；并且

3) 当前报文段中的时间戳(ts_val)小于原先收到的时间戳。

PAWS算法基于这样的假定：对于高速连接， 32 bit时间戳值回绕的速度远小于32 bit序号回绕的速度。习题28.6说明，即使是最高的时钟计数器更新频率 (每毫秒加1)，时间戳的符号位也要24天才会回绕一次。而在千兆级网络中，序号可能 17秒就回绕一次(卷1的24.3节)。因此，如果报文段时间戳小于从同一个连接收到的最近一次的时间戳，说明是个重复报文段，应被丢弃(还需进行后续的时间戳过期测试)。尽管因为序号已过时， tcp_input也可将其丢弃，但PAWS算法能够有效地处理序号回绕速率很高的高速网。

注意， PAWS算法是对称的：它不仅丢弃重复的数据报文段，也丢弃重复的 ACK。PAWS处理所有收到的报文段，前面介绍过，首部预测代码也采用了 PAWS测试(图28-11)。

2. 检查过期时间戳

614-627 尽管可能性不大， PAWS测试还是会失败，因为连接有可能长时间空闲。收到的报文段并非重复报文段，但连接空闲时间过长，造成时间戳值回绕，从而小于从同一个连接收到的最近一次的时间戳。

无论何时， ts_recent保存接收报文段中的时间戳值， ts_recent_age记录当前时间(tcp_now)。如果ts_recent的最后一次更新发生在24天之前，则将其清零，不是一个有效的时间戳值。常量TCP_PAWS_IDLE定义为(24×24×60×60×2)，最后的乘数2指每秒钟2个滴答。这种情况下，不丢弃接收报文段，因为问题是时间戳过期，而非重复报文段。参见习

题28.6和28.7。

图28-23举例说明了时间戳过期问题。连接左侧的系统是一个非 Net/3的TCP实现，以RFC 1323中规定的最高速度，每毫秒更新一次时钟。连接右侧是 Net/3实现。

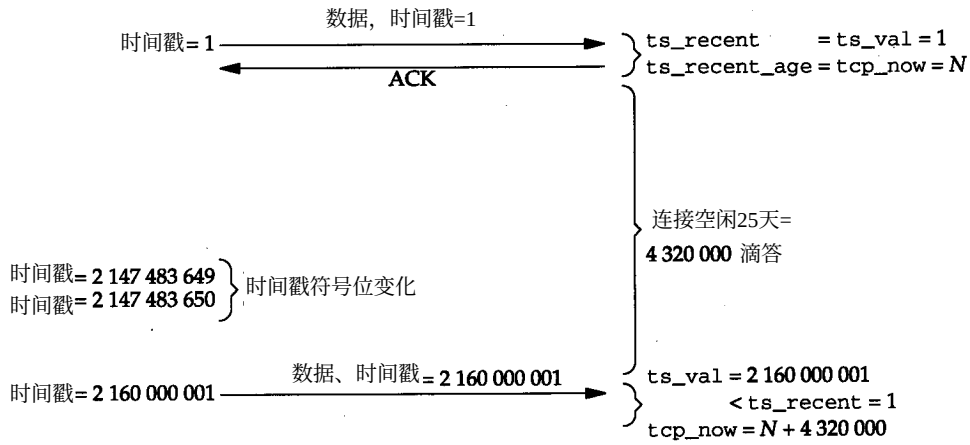

图28-23 过期时间戳举例

第一个数据报文段中携带的时间戳值等于1，所以ts_recent等于1，ts_recent_age等于当前时间(tcp_now)，如图28-11和图28-35所示。连接空闲25天，在此期间tcp_now增加了4 320 000 (25×24×60×60×1000)，对端的时间戳值增加了2 160 000 000 (25×24×60×60×1000)，时间戳值的符号位改变，即2 147 483 649大于1，而2 147 483 650小于1(回想图24-26)。因此，当收到的数据报文段中的时间戳等于 2 160 000 001时，调用TSTMP_LT宏进行比较，收到的时间戳小于ts_recent(1)，PAWS测试失败。但因为tcp_now减去ts_recent_age大于24天，说明造成失败的原因是连接空闲时间过长，报文段被接受。

3. 丢弃重复报文段

628-633 如果PAWS算法测试说明收到的是一个重复报文段，确认之后丢弃该报文段 (所有重复报文段都必须被确认)，不更新本地时间戳变量。

图24-5中，tcp_pawsdrop (22)远小于tcps_rcvduppack (46 953)。这可能是因为目前只有很少的系统支持时间戳，导致绝大多数重复报文段直到 TCP输出处理中才被发现和丢弃，而非PAWS。

28.8 裁剪报文段使数据在窗口内

本节讨论如何调整收到的报文段，确保它只携带能够放入接收窗口内的数据：

- 丢弃接收报文段起始处的重复数据；并且
- 从报文段尾部起，丢弃超出接收窗口的数据。

从而只剩下可放入接收窗口的新数据。图 28-24给出的代码，用于判定报文段起始处是否存在重复数据。

1. 查看报文段前部是否存在重复数据

635-636 如果接收报文段的起始序号 (ti_seq)小于等待接收的下一序号 (rcv_nxt)，则

todrop大于0，报文段前部有重复数据。这些数据已被确认并提交给应用进程 (图24-18)。

tcp_input.c

```
635     todrop = tp->rcv_nxt - ti->ti_seq;
636     if (todrop > 0) {
637         if (tiflags & TH_SYN) {
638             tiflags &= ~TH_SYN;
639             ti->ti_seq++;
640             if (ti->ti_urp > 1)
641                 ti->ti_urp--;
642             else
643                 tiflags &= ~TH_URG;
644             todrop--;
645         }
```

tcp_input.c

图28-24 tcp_input 函数：查看报文段起始处的重复数据

2. 丢弃重复SYN

637-645 如果SYN标志置位，它必然指向报文段的第一个数据序号，现已知是重复数据。清除SYN，报文段的起始序号加1，以越过重复的SYN。此外，如果接收报文段中的紧急指针大于1 (ti_urp)，则必须将其减1，因为紧急数据偏移量以报文段起始序号为基准。如果紧急指针等于0或者1，则不做处理，为防止出现等于1的情况，清除URG标志。最后，todrop减1(因为SYN占用一个序号)。

图28-25继续处理报文段前部的重复数据。

tcp_input.c

```
646         if (todrop >= ti->ti_len) {
647             tcpstat.tcps_rcvduppack++;
648             tcpstat.tcps_rcvdupbyte += ti->ti_len;
649             /*
650              * If segment is just one to the left of the window,
651              * check two special cases:
652              * 1. Don't toss RST in response to 4.2-style keepalive.
653              * 2. If the only thing to drop is a FIN, we can drop
654              *    it, but check the ACK or we will get into FIN
655              *    wars if our FINs crossed (both CLOSING).
656              * In either case, send ACK to resynchronize,
657              * but keep on processing for RST or ACK.
658              */
659             if ((tiflags & TH_FIN && todrop == ti->ti_len + 1)
660                 ) {
661                 todrop = ti->ti_len;
662                 tiflags &= ~TH_FIN;
663                 tp->t_flags |= TF_ACKNOW;
664             } else {
665                 /*
666                  * Handle the case when a bound socket connects
667                  * to itself. Allow packets with a SYN and
668                  * an ACK to continue with the processing.
669                  */
670                 if (todrop != 0 || (tiflags & TH_ACK) == 0)
671                     goto dropafterack;
672             }
673         } else {
674             tcpstat.tcps_rcvpartduppack++;
675             tcpstat.tcps_rcvpartdupbyte += todrop;
```

图28-25 tcp_input 函数：处理完全重复的报文段

```
676         }
677         m_adj(m, todrop);
678         ti->ti_seq += todrop;
679         ti->ti_len -= todrop;
680         if (ti->ti_urp > todrop)
681             ti->ti_urp -= todrop;
682         else {
683             tiflags &= ~TH_URG;
684             ti->ti_urp = 0;
685         }
686     }
```

tcp_input.c

图28-25 (续)

3. 判定报文段数据是否完全重复

646-648 如果报文段前部重复的数据字节数大于等于报文段大小，则是一个完全重复的报文段。

4. 判定重复FIN

649-663 接下来测试FIN是否重复，图28-26举例说明了这一情况。

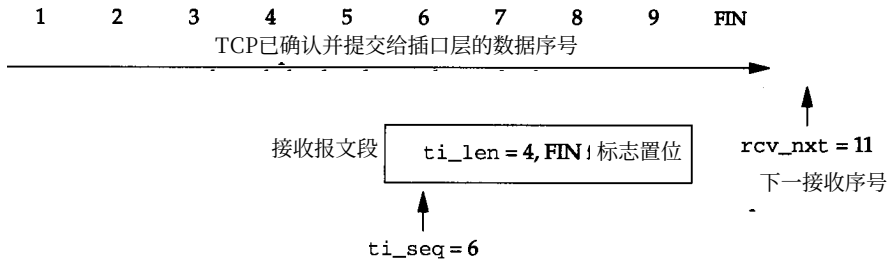

图28-26 举例：带有FIN标志的重复报文段

图28-26的例子中，todrop等于5，大于等于ti_len(4)。因为FIN置位，并且todrop等于ti_len加1，所以清除FIN标志，todrop重设为4，置位TF_ACKNOW，函数结束时立即发送ACK。这个例子也适用于其他报文段，如果ti_seq加上ti_len等于10。

代码的注释提到了4.2BSD实现中的保活定时器，Net/3省略了相关处理(if语句中的另一项测试)。

5. 生成重复ACK

664-672 如果todrop非零(报文段携带的全部是重复数据)，或者ACK标志未置位，则丢弃报文段，调用dropafterack生成ACK。出现这种情况，一般是因为对端未收到ACK，导致报文段重发。TCP生成新的ACK。

6. 处理同时打开或半连接

664-672 代码还处理同时打开，以及插口与自己建立连接的情况，将在下一节中详细讨论。如果todrop等于0(完全重复报文段中不包含数据)，且ACK标志置位，则继续下一步的处理。

if语句是4.4BSD版中新加的。早期的基于Berkeley的系统只是简单地跳转到dropafterack，即不处理同时打开，也不处理与自己建立连接的情况。

即使做了改进，这段代码仍有错误，我们在本节结束时将谈到这一点。

7. 收到部分重复报文段时，更新统计值

673-676 当todrop小于报文段长度时，执行else语句：报文段携带数据中只有部分重复。

8. 删除重复数据，更新紧急指针

677-685 调用m_adj，从mbuf链的首部开始删除重复数据，并相应地调整起始序号和长度。如果紧急指针指向的数据仍在 mbuf中，也需做相应的调整。否则，紧急指针清零，并清除URG标志。

图28-27给出了函数下一部分的代码，处理应用进程终止后到达的数据。

tcp_input.c

```
687     /*
688      * If new data is received on a connection after the
689      * user processes are gone, then RST the other end.
690      */
691     if ((so->so_state & SS_NOFDREF) &&
692         tp->t_state > TCPS_CLOSE_WAIT && ti->ti_len) {
693         tp = tcp_close(tp);
694         tcpstat.tcps_rcvafterclose++;
695         goto dropwithreset;
696     }
```

tcp_input.c

图28-27 tcp_input 函数：处理应用进程终止后到达的数据

687-696 如果找不到插口的描述符，说明应用进程已关闭了连接 (连接状态等于图24-16中大于CLOSE_WAIT的5个状态中的任何一个)，若接收报文段中有数据，则连接被关闭。报文段被丢弃，输出RST做为响应。

因为TCP支持半关闭功能，如果应用进程意外终止 (也许被某个信号量终止)，做为进程终止的一部分，内核将关闭所有打开的描述符， TCP将发送FIN。连接转移到FIN_WAIT_1状态。因为FIN的接收者无法知道对端执行的是完全关闭，还是半关闭。如果它假定是半关闭，并继续发送数据，那么将收到图28-27中发送的FIN。

图28-28给出了函数下一部分的代码，从接收报文段中删除落在通告窗口右侧的数据。

tcp_input.c

```
697     /*
698      * If segment ends after window, drop trailing data
699      * (and PUSH and FIN); if nothing left, just ACK.
700      */
701     todrop = (ti->ti_seq + ti->ti_len) - (tp->rcv_nxt + tp->rcv_wnd);
702     if (todrop > 0) {
703         tcpstat.tcps_rcvpackafterwin++;
704         if (todrop >= ti->ti_len) {
705             tcpstat.tcps_rcvbyteafterwin += ti->ti_len;
706             /*
707              * If a new connection request is received
708              * while in TIME_WAIT, drop the old connection
709              * and start over if the sequence numbers
710              * are above the previous ones.
711              */
712             if (tiflags & TH_SYN &&
713                 tp->t_state == TCPS_TIME_WAIT &&
```

图28-28 tcp_input 函数：删除落在窗口右侧的数据

```
714                 SEQ_GT(ti->ti_seq, tp->rcv_nxt)) {
715                 iss = tp->rcv_nxt + TCP_ISSINCR;
716                 tp = tcp_close(tp);
717                 goto findpcb;
718             }
719             /*
720              * If window is closed can only take segments at
721              * window edge, and have to drop data and PUSH from
722              * incoming segments.  Continue processing, but
723              * remember to ack.  Otherwise, drop segment
724              * and ack.
725              */
726             if (tp->rcv_wnd == 0 && ti->ti_seq == tp->rcv_nxt) {
727                 tp->t_flags |= TF_ACKNOW;
728                 tcpstat.tcps_rcvwinprobe++;
729             } else
730                 goto dropafterack;
731         } else
732             tcpstat.tcps_rcvbyteafterwin += todrop;
733         m_adj(m, -todrop);
734         ti->ti_len -= todrop;
735         tiflags &= ~(TH_PUSH | TH_FIN);
736     }
```

tcp_input.c

图28-28 (续)

9. 计算落在通告窗口右侧的字节数

697-703 todrop等于接收报文段中落在通告窗口右侧的字节数。例如，在图 28-29中，todrop等于(6+5)减去(4+6)，即等于1。

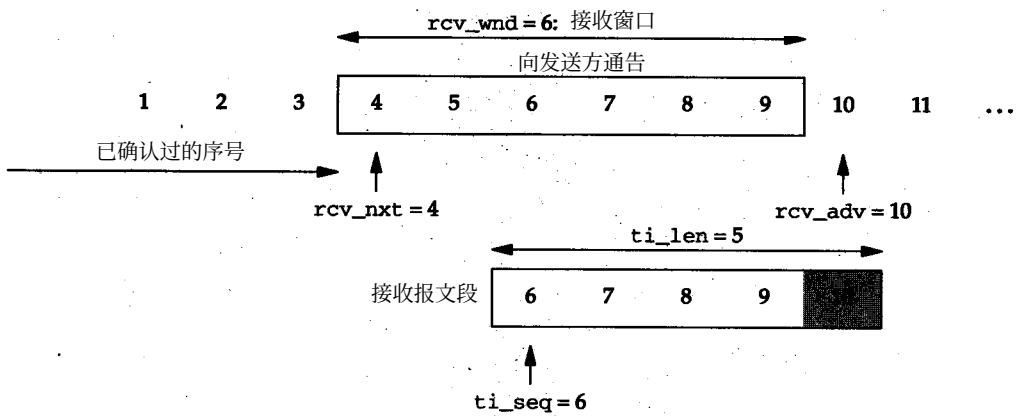

图28-29 举例：接收报文段部分数据落在窗口右侧

10. 如果连接处于TIME_WAIT状态，查看有无新的连接请求

704-718 如果todrop大于等于报文段长度，则丢弃整个报文段。如果下列3个条件全真：

1) SYN标志置位；并且

2) 连接处于TIME_WAIT状态；并且

3) 新的起始序号大于连接上最后收到的序号；

说明对端要求在已被关闭且正处于 TIME_WAIT状态的连接上重建连接。 RFC 1122允许这种情况，但要求新连接的ISS必须大于最后收到的序号(rcv_nxt)。TCP在rcv_nxt的基础上增加

128 000(TCP_ISSINCR)，得到执行图28-17中的代码时所使用的ISS。调用tcp_close释放处于TIME_WAIT状态的原有连接的PCB和TCP控制块。控制跳转到findpcb(图28-5)，寻找监听服务器的PCB(假定服务器仍在运行)。然后执行图28-7中的代码，为新连接创建新的插口，最后执行图28-16和图28-17中的代码，完成新连接请求的处理。

11. 判定是否为窗口探测报文段

719-728 如果接收窗口已关闭(rcv_wnd等于0)，且接收报文段中的数据从窗口最左端开始(rcv_nxt)，说明是对端发送的窗口探测报文段。TCP立即发送响应ACK，其中包含等待接收的序号。

12. 丢弃完全落在窗口之外的其他报文段

729-730 如果报文段整个落在窗口之外，且并非窗口探测报文段，则丢弃该报文段，并发送携带等待接收序号的ACK，作为响应。

13. 处理携带部分有效数据的报文段

731-735 通过m_adj，从mbuf链中删除落在窗口右侧的数据，并更新ti_len。如果接收报文段是对端发送的窗口探测报文段，m_adj将丢弃mbuf链中的所有数据，并将ti_len设为0，最后清除FIN和PSH标志。

何时丢弃ACK

图28-25中的代码有错误，在几种情况下，本应继续进行报文段处理，控制却跳转到dropafterack[Carlson 1993; Lanciani 1993]。系统实际运行时，如果连接双方重组队列中都存在缺失报文段，并都进入持续状态，将造成死锁，因为双方都将丢弃正常的ACK。

纠正的方法是简化图28-25起始处的代码。控制不再跳转到dropafterack，如果收到了完全重复报文段，则关闭FIN标志，并在函数结束时强迫立即发送ACK。删除图28-25中的646~676行的代码，而代之以图28-30中的代码。此外，新代码还更正了原代码中的另一个错误(习题28.9)。

```
if (todrop > ti->ti_len ||
    todrop == ti->ti_len && (tiflags & TH_FIN) == 0) {

    /*
     * Any valid FIN must be to the left of the window.
     * At this point the FIN must be a duplicate or
     * out of sequence; drop it.
     */
     tiflags &= ~TH_FIN;

    /*
     * Send an ACK to resynchronize and drop any data.
     * But keep on processing for RST or ACK.
     */
    tp->t_flags |= TF_ACKNOW;
    todrop = ti->ti_len;
    tcpstat.tcps_rcvdupbyte += todrop;
    tcpstat.tcps_rcvduppack++;

} else {
    tcpstat.tcps_rcvpartduppack++;
    tcpstat.tcps_rcvpartdupbyte += todrop;
}
```

图28-30 图28-28中646~676行代码的修正

28.9 自连接和同时打开

读者应首先理解插口与自己建立连接的步骤。接着会看到在 4.4BSD中，如何巧妙地通过一行代码修正图28-25中的错误，从而不仅能够处理自连接，还能处理 4.4BSD以前的版本中都无法正确处理的同时打开。

应用进程创建一个插口，并通过下列系统调用建立自连接： `socket`，`bind`绑定到一个本地端口(假定为3000)，之后`connect`试图与同一个本地地址和同一个端口号建立连接。如果`connect`成功，则插口已建立了与自己的连接：向这个插口写入的所有数据，都可以在同一插口上读出。这有点类似于全双工的管道，但只有一个，而非两个标识符。尽管很少有应用进程会这样做，但实际上它是一种特殊的同时打开，两者的状态变迁图相同。如果系统不允许插口建立自连接，那么它也很可能无法正确处理同时打开，而后者是 RFC 1122所要求的。有些人对于自连接能成功感到非常惊诧，因为只用了一个 Internet PCB和一个TCP控制块。不过，TCP是全双工的、对称的协议，它为每个方向上的数据流保留一份专有数据。

图28-31给出了应用进程调用 `connect`时的发送序号空间， SYN已发送，连接状态为SYN_SENT。

插口收到SYN后，执行图28-18和图28-20中的代码，但因为SYN中未包含ACK，连接状态转移到SYN_RCVD。从状态变迁图(图24-15)可知，与同时打开类似。图 28-32给出了接收序号空间。图28-20置位`TF_ACKNOW`，`tcp_output`生成的报文段将包含SYN和ACK(图24-16中的`tcp_outflags`)。SYN序号等于153，而确认序号等于154。

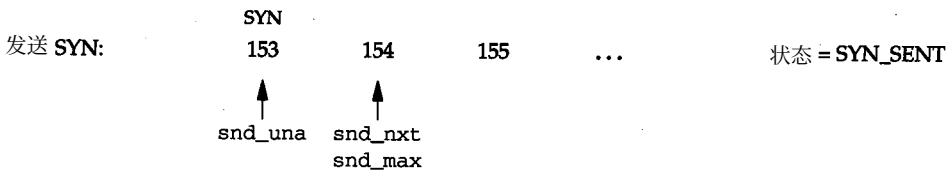

图28-31 自连接：SYN发送后的发送序号空间

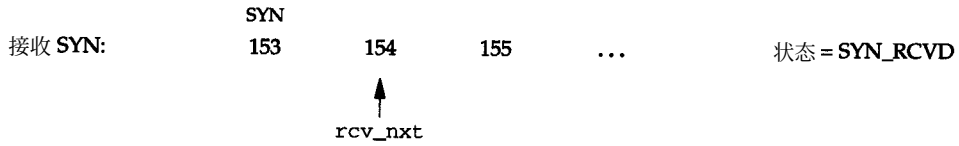

图28-32 自连接：收到的SYN处理完毕后的接收序号空间

与图28-20处理的正常情况相比，发送序号空间没有变化，只是连接状态等于 SYN_SENT。图28-33给出了收到同时带有SYN和ACK的报文段时，接收序号空间的状态。

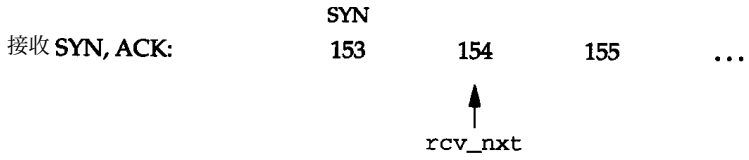

图28-33 收到带有SYN和ACK报文段时，接收序号空间的状态

因为连接状态等于SYN_RCVD，将执行图29-2中的代码处理收到的报文段，而不用我们在本章前面讨论过的处理主动打开或被动打开的代码。但在此之前，首先遇到的是图 28-24中的代码，而且从测试结果看似乎是一个重复 SYN：

```
todrop = rcv_nxt - rcv_seq
       = 154 - 153
       = 1
```

因为SYN标志置位，清除该标志，ti_seq等于154，todrop等于0。但因为todrop等于报文段长度(0)，图28-25开始处的测试条件为真，从而判定是一个重复报文段，执行注释为“处理绑定插口自连接的情况”的代码。早期的 TCP实现直接跳到dropafterack，略过了SYN_RCVD状态的处理逻辑，不可能建立连接。相反，即使todrop等于0，且ACK标志置位(本例中两个条件都成立)，Net/3仍旧继续处理收到的报文段，从而进入函数后面对SYN_RCVD状态的处理，连接转移到ESTABLISHED状态。

图28-34给出了自连接处理中函数调用的情况，是非常有意思的。

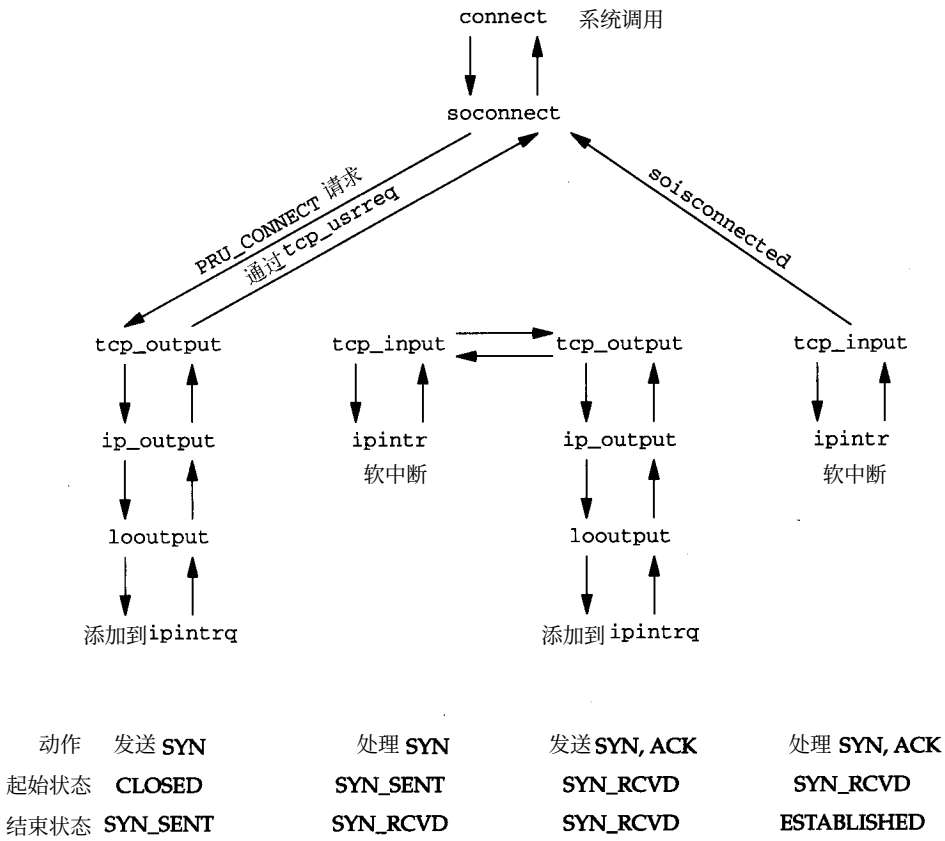

图28-34 自连接处理中的函数调用序列

操作顺序从左至右，首先应用进程调用 connect，发出PRU_CONNECT请求，经协议栈发送SYN。因为报文段发向主机自己的IP地址，直接通过环回接口加入到 ipintrq，并生成一个软中断。

系统在软中断处理中调用 ipintr，ipintr调用tcp_input，tcp_input再调用

tcp_output，经协议栈发送带有ACK的SYN。这个报文段也经由环回接口加入到ipintrq，并生成一个软中断。系统调用 ipintr处理软中断， ipintr调用tcp_input，连接进入ESTABLISHED状态。

28.10 记录时间戳

图28-35给出了tcp_input下一部分的代码，处理收到的时间戳选项。

tcp_input.c

```
737     /*
738      * If last ACK falls within this segment's sequence numbers,
739      * record its timestamp.
740      */
741     if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
742         SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len +
743                ((tiflags & (TH_SYN | TH_FIN)) != 0))) {
744         tp->ts_recent_age = tcp_now;
745         tp->ts_recent = ts_val;
746     }
```

tcp_input.c

图28-35 tcp_input 函数：记录时间戳

737-746 如果收到的报文段中带有时间戳，时间戳值保存在 ts_recent中。我们在26.6节曾讨论过Net/3的处理代码有错误。如果FIN和SYN标志均未置位，表达式

```
((tiflags & (TH_SYN|TH_FIN)) != 0)
```

等于0；如果有一个置位，则等于1。

28.11 RST处理

图28-36给出了处理RST标志的switch语句，取决于当前的连接状态。

1. SYN_RCVD状态

759-761 插口差错代码设定为ECONNREFUSED，控制向前跳转若干行，关闭插口。在两种状况下，连接进入此状态。一般地讲，连接收到 SYN后，从LISTEN转移到SYN_RCVD状态。TCP发送带有ACK的SYN做为响应，但接着却收到了对端的 RST。此时， so引用的插口是在图28-7中调用sonewconn新创建的。因为dropsocket为真，在标注drop处，插口被丢弃，监听插口不受影响。这也是图 24-15中状态从SYN_RCVD转回LISTEN的原因。

另一种情况是，应用进程调用 connect后，出现同时打开，状态也转移到 SYN_RCVD。收到RST后，向应用进程返回插口差错。

tcp_input.c

```
747     /*
748      * If the RST bit is set examine the state:
749      *    SYN_RECEIVED state:
750      *  If passive open, return to LISTEN state.
751      *  If active open, inform user that connection was refused.
752      *    ESTABLISHED, FIN_WAIT_1, FIN_WAIT2, CLOSE_WAIT states:
753      *  Inform user that connection was reset, and close tcb.
754      *    CLOSING, LAST_ACK, TIME_WAIT states
755      *  Close the tcb.
756      */
```

图28-36 tcp_input 函数：处理RST标志

```
757     if (tiflags & TH_RST)
758         switch (tp->t_state) {

759         case TCPS_SYN_RECEIVED:
760             so->so_error = ECONNREFUSED;
761             goto close;

762         case TCPS_ESTABLISHED:
763         case TCPS_FIN_WAIT_1:
764         case TCPS_FIN_WAIT_2:
765         case TCPS_CLOSE_WAIT:
766             so->so_error = ECONNRESET;
767           close:
768             tp->t_state = TCPS_CLOSED;
769             tcpstat.tcps_drops++;
770             tp = tcp_close(tp);
771             goto drop;

772         case TCPS_CLOSING:
773         case TCPS_LAST_ACK:
774         case TCPS_TIME_WAIT:
775             tp = tcp_close(tp);
776             goto drop;
777         }
```

tcp_input.c

图28-36 (续)

2. 其他状态

762-777 如果在ESTABLISHED、FIN_WAIT_1、FIN_WAIT_2或CLOSE_WAIT状态收到RST，则返回差错代码ECONNRESET。如果状态为CLOSING、LAST_ACK或TIME_WAIT，由于应用进程已关闭插口，无需返回差错代码。

如果允许RST终止处于TIME_WAIT状态的连接，那么TIME_WAIT状态也就没有存在的必要。RFC 1337 [Braden 1992]讨论了这一点，及其他取消TIME_WAIT状态的可能状况，建议不允许RST永久终止处于TIME_WAIT状态的连接。参见习题28.10中的例子。

图28-37给出了函数下一部分的代码，验证SYN是否出错，ACK是否存在。

tcp_input.c

```
778     /*
779      * If a SYN is in the window, then this is an
780      * error and we send an RST and drop the connection.
781      */
782     if (tiflags & TH_SYN) {
783         tp = tcp_drop(tp, ECONNRESET);
784         goto dropwithreset;
785     }
786     /*
787      * If the ACK bit is off we drop the segment and return.
788      */
789     if ((tiflags & TH_ACK) == 0)
790         goto drop;
```

tcp_input.c

图28-37 tcp_input 函数：处理带有多余SYN或者缺少ACK的报文段

778-785 如果SYN标志依旧置位，说明出现了差错，连接被丢弃，返回差错代码ECONNRESET。

786-790 如果ACK标志未置位，则报文段被丢弃。我们将在下一章讨论函数剩余部分的代码，其中假定ACK标志均置位。

28.12 小结

本章详细介绍了TCP输入处理的前半部分，下一章将继续讨论函数剩余的部分。

本章介绍了如何验证报文段检验和，处理各种 TCP选项，处理发起和结束连接建立的SYN报文段，从报文段头尾两个方向删除无效数据，及处理 RST标志。

首部预测算法处理正常情况的数据流是非常有效的，执行速度最快。尽管我们讨论的多数处理逻辑用于覆盖所有可能发生的情况，但多数报文段都是正常的，只需很少的处理步骤。

习题

28.1 假定Net/3中插口缓存最大等于262 444，基于图28-7的算法，得到的窗口缩放因子是多少？

28.2 假定Net/3中插口缓存最大等于262 444，如果往返时间等于60ms，可能的最大吞吐量是多少？(提示：见卷1的图24-5及带宽的解)

28.3 为什么图28-10中，调用bcopy获取时间戳值？

28.4 我们在26.6节中提到，TCP要求的时间戳选项格式与RFC 1323附录A中定义的不同。尽管TCP能够正确处理时间戳，但由于采用与标准不同的格式，会付出什么代价？

28.5 处理PRU_ATTACH请求时会分配 PCB和TCP控制块，为什么不接着调用tcp_template分配首部模板？而是直至收到了SYN，在图28-17中才进行这一操作。

28.6 阅读RFC 1323，理解为什么图28-22中选取24天做为空闲时间的界限？

28.7 在图28-22中，如果连接空闲时间超过 24天，tcp_now-ts_recent_age与TCP_PAWS_IDLE的比较，会出现符号位回绕的问题。 Net/3中采取500ms做为时间戳单位，会在什么时间出现问题？

28.8 阅读RFC 1323，回答为什么图28-22中PAWS测试不包括RST报文段？

28.9 客户发送了SYN，服务器响应SYN/ACK。客户转移到ESTABLISHED状态，并发送响应ACK。但这个ACK丢失，服务器重发 SYN/ACK。描述一下客户收到重发的SYN/ACK时的处理步骤。

28.10 客户和服务器已建立了连接，服务器主动关闭。连接正常终止，服务器上的插口对转移到TIME_WAIT状态。在服务器的2MSL定时器超时前，同一客户 (客户端的同一个插口对)向服务器发送SYN，但起始序号小于连接上最后收到的序号。会发生什么？