

第27章 TCP的函数

27.1 引言

本章介绍多个TCP函数，它们为下两章进一步讨论TCP的输入打下了基础：

- `tcp_drain`是协议的资源耗尽处理函数，当内核的 `mbuf`用完时被调用。实际上，不做任何处理。
- `tcp_drop`发送RST来丢弃连接。
- `tcp_close`执行正常的TCP连接关闭操作：发送FIN，并等待协议要求的4次报文交换以终止连接。卷1的18.2节讨论了连接关闭时双方需要交换的4个报文。
- `tcp_mss`处理收到的MSS选项，并在TCP发送自己的MSS选项时计算应填入的MSS值。
- `tcp_ctlinput`在收到对应于某个TCP报文段的ICMP差错时被调用，它接着调用 `tcp_notify`处理ICMP差错。`tcp_quench`专门负责处理ICMP的源站抑制差错。
- `TCP_REASS`宏和 `tcp_reass`函数管理连接重组队列中的报文段。重组队列处理收到的乱序报文段，某些报文段还可能互相重复。
- `tcp_trace`向内核的TCP调试循环缓存中添加记录（插口选项 `SO_DEBUG`）。运行 `trpt` (8)程序可以打印缓存内容。

27.2 `tcp_drain`函数

`tcp_drain`是所有TCP函数中最简单的。它是协议的 `pr_drain`函数，在内核的 `mbuf`用完时，由 `m_reclaim`调用。图10-32中，`ip_drain`丢弃其重组队列中的所有数据报分片，而UDP则不定义自己的资源耗尽处理函数。尽管TCP也占用 `mbuf`——位于接收窗口内的乱序报文段——但Net/3实现的TCP并不丢弃这些 `mbuf`，即使内核的 `mbuf`已用完。相反，`tcp_drain`不做任何处理，假定收到的（但次序差错）的TCP报文段比IP分片重要。

27.3 `tcp_drop`函数

`tcp_drop`在整个系统中多次被调用，发送RST报文段以丢弃连接，并向应用进程返回差错。它与关闭连接（`tcp_disconnect`函数）不同，后者向对端发送FIN，并遵守TCP状态变迁图所规定的连接终止步骤。

图27-1列出了调用 `tcp_drop`的7种情况和相应的 `errno`参数。

图27-2给出了 `tcp_drop`函数。

202-213 如果TCP收到了一个SYN，连接被同步，则必须向对端发送RST。`tcp_drop`把状态设为CLOSED，并调用 `tcp_output`。从图24-16可知，CLOSED状态的 `tcp_outflags`数组中包含RST标志。

214-216 如果 `errno`等于ETIMEDOUT，且连接上曾收到过软差错（如EHOSTUNREACH），

软差错代码将取代内容不确定的 ETIMEDOUT，做为返回的插口差错。

217 tcp_close结束插口关闭操作。

函 数	errno	描 述
tcp_input	ENOBUFS	监听服务器收到SYN，但内核无法为t_template分配所需的mbuf
tcp_input	ECONNREFUSED	收到的RST是对本地发送的SYN的响应
tcp_input	ECONNRESET	在现存连接上收到了RST
tcp_timers	ETIMEDOUT	重传定时器连续超时13次，仍未收到对端的ACK(图25-25)
tcp_timers	ETIMEDOUT	连接建立定时器超时(图25-16)，或者保活定时器超时，且连续9次发送窗口探测报文段，对方均无响应
tcp_usrreq	ECONNABORTED	PRU_ABORT请求
tcp_usrreq	0	关闭插口，设定SO_LINGER选项，且拖延时间为0

图27-1 调用tcp_drop 函数和errno 参数

```

202 struct tcpcb *
203 tcp_drop(tp, errno)
204 struct tcpcb *tp;
205 int      errno;
206 {
207     struct socket *so = tp->t_inpcb->inp_socket;
208     if (TCPS_HAVERCVDSYN(tp->t_state)) {
209         tp->t_state = TCPS_CLOSED;
210         (void) tcp_output(tp);
211         tcpstat.tcps_drops++;
212     } else
213         tcpstat.tcps_conndrops++;
214     if (errno == ETIMEDOUT && tp->t_softerror)
215         errno = tp->t_softerror;
216     so->so_error = errno;
217     return (tcp_close(tp));
218 }

```

tcp_subr.c

tcp_subr.c

图27-2 tcp_drop 函数

27.4 tcp_close函数

通常情况下，如果应用进程被动关闭，且在 LAST_ACK 状态时收到了 ACK，tcp_input 将调用 tcp_close 关闭连接；或者当 2MSL 定时器超时，插口从 TIME_WAIT 状态变迁到 CLOSED 状态时，tcp_timers 也会调用 tcp_close。它也可以在其他状态被调用，一种可能是发生了差错，如上一小节讨论过的情况。tcp_close 释放连接占用的内存 (IP 和 TCP 首部模板、TCP 控制块、Internet PCB 和保存在连接重组队列中的所有乱序报文段)，并更新路由特性。

我们分3部分讲解这个函数，前两部分讨论路由特性，最后一部分介绍资源释放。

27.4.1 路由特性

rt_metrics 结构(图18-26)中保存了9个变量，有6个用于 TCP。其中8个变量可通过

route (8)命令读写(第9个, `rmx_pkssent`未使用):图27-3列出了这些变量。此外,运行route命令时,加入`-lock`选项,可以设置`rmx_locks`成员变量(图20-13)中对应的RTV_...比特,告诉内核不要更新对应的路由参数。

关闭TCP 插口时,如果下列条件满足:连接上传输的数据量足够生成有效的统计值,并且变量未被锁定,`tcp_close`将更新3个路由参数——已平滑的RTT估计器、已平滑的RTT平均偏差估计器和慢起动门限。

rt_metrics成员	tcp_close是否 保存该成员	tcp_mss是否 使用该成员	route(8)附加参数
<code>rmx_expire</code>			<code>-expire</code>
<code>rmx_hopcount</code>			<code>-hopcount</code>
<code>rmx_mtu</code>		•	<code>-mtu</code>
<code>rmx_recvpipe</code>		•	<code>-recvpipe</code>
<code>rmx_rtt</code>	•	•	<code>-rtt</code>
<code>rmx_rttvar</code>	•	•	<code>-rttvar</code>
<code>rmx_sendpipe</code>		•	<code>-sendpipe</code>
<code>rmx_ssthresh</code>	•	•	<code>-ssthresh</code>

图27-3 TCP用到的`rt_metrics` 结构中的变量

图27-4给出了`tcp_close`的第一部分。

1. 判断是否发送了足够的数据量

234-248 默认的发送缓存大小为8192字节(`sb_hiwat`),因此首先比较初始发送序号和连接上已发送的最大序号,测试是否已传输了131 072字节(16个完整的缓存)的数据。此外,插口还必须有一条非默认路由的缓存路由(参见习题19.2)。

请注意,如果传输的数据量在 $N \times 2^{32}$ ($N > 1$)和 $N \times 2^{32} + 131072$ ($N > 1$)之间,则因为序号可能回绕,比较时也许会出现问题,尽管可能性不大。但目前很少有连接会传输4 G的数据。

尽管Internet上存在大量的默认路由,缓存路由对于维护有效的路由表还是很有用的。如果主机长期与另外某个主机(或网络)交换数据,即使默认路由可用,也应运行route命令向路由表中添加源站选路和目的选路的路由,从而在整条连接上维护有效的路由信息(参见习题19.2)。这些信息在系统重启时丢失。

250 管理员可以锁定图27-3中的变量,防止内核修改它们。因此,代码在更新这些变量之前,必须先检查其锁定状态。

2. 更新RTT

251-264 `t_srtt`的单位为8个滴答(图25-19),而`rmx_rtt`的单位为微秒。因此,首先必须实现单位换算,`t_srtt`乘以1 000 000(`RTM_RTTUNIT`),除以2(滴答/秒)再乘以8,得到RTT的最新值。如果`rmx_rtt`值已存在,它被更新为最新值与原有值和的一半,即两者的平均值。如果不存在,最新值将直接赋给`rmx_rtt`变量。

3. 更新平均偏差

265-273 更新平均偏差的算法与更新RTT的类似,也需要把单位为4个滴答的`t_rttvar`换算为以微秒为单位。

tcp_subr.c

```

225 struct tcpcb *
226 tcp_close(tp)
227 struct tcpcb *tp;
228 {
229     struct tcphdr *t;
230     struct inpcb *inp = tp->t_inpcb;
231     struct socket *so = inp->inp_socket;
232     struct mbuf *m;
233     struct rtentry *rt;
234
235     /*
236      * If we sent enough data to get some meaningful characteristics,
237      * save them in the routing entry. 'Enough' is arbitrarily
238      * defined as the sendpipe size (default 8K) * 16. This would
239      * give us 16 rtt samples assuming we only get one sample per
240      * window (the usual case on a long haul net). 16 samples is
241      * enough for the srtt filter to converge to within 5% of the correct
242      * value; fewer samples and we could save a very bogus rtt.
243      * Don't update the default route's characteristics and don't
244      * update anything that the user "locked".
245      */
246     if (SEQ_LT(tp->iss + so->so_snd.sb_hiwat * 16, tp->snd_max) &&
247         (rt = inp->inp_route.ro_rt) &&
248         ((struct sockaddr_in *) rt_key(rt))->sin_addr.s_addr != INADDR_ANY) {
249         u_long i;
250
251         if ((rt->rt_rmx.rmx_locks & RTV_RTT) == 0) {
252             i = tp->t_srtt *
253                 (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));
254             if (rt->rt_rmx.rmx_rtt && i)
255                 /*
256                  * filter this update to half the old & half
257                  * the new values, converting scale.
258                  * See route.h and tcp_var.h for a
259                  * description of the scaling constants.
260                  */
261                 rt->rt_rmx.rmx_rtt =
262                     (rt->rt_rmx.rmx_rtt + i) / 2;
263             else
264                 rt->rt_rmx.rmx_rtt = i;
265         }
266         if ((rt->rt_rmx.rmx_locks & RTV_RTTVAR) == 0) {
267             i = tp->t_rttvar *
268                 (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
269             if (rt->rt_rmx.rmx_rttvar && i)
270                 rt->rt_rmx.rmx_rttvar =
271                     (rt->rt_rmx.rmx_rttvar + i) / 2;
272             else
273                 rt->rt_rmx.rmx_rttvar = i;
274         }
275     }

```

tcp_subr.c

图27-4 tcp_close 函数：更新RTT和平均偏差

图27-5给出了tcp_close的下一部分代码，更新路由的慢启动门限。

274-283 满足下列条件时，慢启动门限被更新：(1)它被更新过(rmx_ssthresh非零)；(2)管理员规定了rmx_sendpipe，而snd_ssthresh的最新值小于rmx_sendpipe的一半。如同代码注释中指出的，TCP不会更新rmx_ssthresh值，除非因为数据分组丢失而不得不

这样做。从这个角度出发，除非十分必要，TCP不会修改门限值。

```

274      /*
275      * update the pipelimit (ssthresh) if it has been updated
276      * already or if a pipesize was specified & the threshold
277      * got below half the pipesize. I.e., wait for bad news
278      * before we start updating, then update on both good
279      * and bad news.
280      */
281      if ((rt->rt_rmx.rmx_locks & RTV_SSTHRESH) == 0 &&
282          (i = tp->snd_ssthresh) && rt->rt_rmx.rmx_ssthresh ||
283          i < (rt->rt_rmx.rmx_sendpipe / 2)) {
284          /*
285          * convert the limit from user data bytes to
286          * packets then to packet data bytes.
287          */
288          i = (i + tp->t_maxseg / 2) / tp->t_maxseg;
289          if (i < 2)
290              i = 2;
291          i *= (u_long) (tp->t_maxseg + sizeof(struct tcphdr));
292          if (rt->rt_rmx.rmx_ssthresh)
293              rt->rt_rmx.rmx_ssthresh =
294                  (rt->rt_rmx.rmx_ssthresh + i) / 2;
295          else
296              rt->rt_rmx.rmx_ssthresh = i;
297      }
298  }

```

tcp_subr.c

图27-5 tcp_close 函数：更新慢起动门限

284-290 变量snd_ssthresh以字节为单位，除以MSS(t_maxseg)得到报文段数，加上 $1/2t_maxseg$ 是为了保证总报文段容量必定大于snd_ssthresh字节。报文段数的下限为2个报文段。

291-297 MSS加上IP和TCP首部大小(40)，再乘以报文段数，利用得到的结果来更新rmx_ssthresh，采用的算法与图27-4中的相同(新值的1/2加上原有值的1/2)。

27.4.2 资源释放

图27-6给出了tcp_close的最后一部分，释放插口占用的内存资源。

```

299      /* free the reassembly queue, if any */
300      t = tp->seg_next;
301      while (t != (struct tcphdr *) tp) {
302          t = (struct tcphdr *) t->ti_next;
303          m = REASS_MBUF((struct tcphdr *) t->ti_prev);
304          remque(t->ti_prev);
305          m_freem(m);
306      }
307      if (tp->t_template)
308          (void) m_free(dtom(tp->t_template));
309      free(tp, M_PCB);
310      inp->inp_ppcb = 0;

```

tcp_subr.c

图27-6 tcp_close 函数：释放连接资源

```

311     soisdisconnected(so);
312     /* clobber input pcb cache if we're closing the cached connection */
313     if (inp == tcp_last_inpcb)
314         tcp_last_inpcb = &tc;
315     in_pcbdetach(inp);
316     tcpstat.tcps_closed++;
317     return ((struct tcpcb *) 0);
318 }

```

—tcp_subr.c

图27-6 (续)

1. 释放重组队列占用的mbuf

299-306 如果连接重组队列中还有报文段，则丢弃它们。重组队列用于存放收到位于接收窗口内、但次序差错的报文段。在等待接收的正常序列报文段到达之前，它们会一直保存在重组队列中；之后，报文段被重组并递交给应用程序。27.9节会详细讨论这一过程。

2. 释放首部模板和TCP控制块

307-309 调用 `m_free` 释放 IP 和 TCP 首部模板，调用 `free` 释放 TCP 控制块，调用 `sodisconnected` 发送 `PRU_DISCONNECT` 请求，标记插口已断开连接。

3. 释放PCB

310-318 如果插口的 Internet PCB 保存在 TCP 的高速缓存中，则把 TCP 的 PCB 链表表头赋给 `tcp_last_inpcb`，以清空缓存。接着调用 `in_pcbdetach` 释放 PCB 占用的内存。

27.5 tcp_mss函数

`tcp_mss` 被两个函数调用：

- 1) `tcp_output`，准备发送 SYN 时调用，以添加 MSS 选项；
 - 2) `tcp_input`，收到的 SYN 报文段中包含 MSS 选项时调用；
- `tcp_mss` 函数检查到达目的地的缓存路由，计算用于该连接的 MSS。

图27-7给出了 `tcp_mss` 第一部分的代码，如果 PCB 中没有到达目的地的路由，则设法得到所需的路由。

```

1391 int
1392 tcp_mss(tp, offer)
1393 struct tcpcb *tp;
1394 u_int offer;
1395 {
1396     struct route *ro;
1397     struct rentry *rt;
1398     struct ifnet *ifp;
1399     int rtt, mss;
1400     u_long bufsize;
1401     struct inpcb *inp;
1402     struct socket *so;
1403     extern int tcp_mssdflt;
1404
1405     inp = tp->t_inpcb;
1406     ro = &inp->inp_route;
1407
1408     if ((rt = ro->ro_rt) == (struct rentry *) 0) {

```

—tcp_input.c

图27-7 `tcp_mss` 函数：如果 PCB 中没有路由，则设法得到所需路由

```

1407      /* No route yet, so try to acquire one */
1408      if (inp->inp_faddr.s_addr != INADDR_ANY) {
1409          ro->ro_dst.sa_family = AF_INET;
1410          ro->ro_dst.sa_len = sizeof(ro->ro_dst);
1411          ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
1412              inp->inp_faddr;
1413          rtalloc(ro);
1414      }
1415      if ((rt = ro->ro_rt) == (struct rtable *) 0)
1416          return (tcp_mssdflt);
1417  }
1418  ifp = rt->rt_ifp;
1419  so = inp->inp_socket;

```

—tcp_input.c

图27-7 (续)

1. 如果需要，就获取路由

1391-1417 如果插口没有高速缓存路由，则调用 `rtalloc` 得到一条。与外出路由相关的接口指针存储在 `ifp` 中。外出接口是非常重要的，因为其 MTU 会影响 TCP 通告的 MSS。如果无法得到所需路由，函数就立即返回默认值 512 (`tcp_mssdflt`)。

图27-8给出了 `tcp_mss` 的下一部分代码，判断得到的路由是否有相应的参数表。如果有，则变量 `t_rttmin`、`t_srtt` 和 `t_rttvar` 将初始化为参数表中的对应值。

```

1420      /*
1421       * While we're here, check if there's an initial rtt
1422       * or rttvar. Convert from the route-table units
1423       * to scaled multiples of the slow timeout timer.
1424       */
1425      if (tp->t_srtt == 0 && (rtt = rt->rt_rmx.rmx_rtt)) {
1426          /*
1427           * XXX the lock bit for RTT indicates that the value
1428           * is also a minimum value; this is subject to time.
1429           */
1430          if (rt->rt_rmx.rmx_locks & RTV_RTT)
1431              tp->t_rttmin = rtt / (RTM_RTTUNIT / PR_SLOWHZ);
1432          tp->t_srtt = rtt / (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));

1433          if (rt->rt_rmx.rmx_rttvar)
1434              tp->t_rttvar = rt->rt_rmx.rmx_rttvar /
1435                  (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
1436          else
1437              /* default variation is +- 1 rtt */
1438              tp->t_rttvar =
1439                  tp->t_srtt * TCP_RTTVAR_SCALE / TCP_RTT_SCALE;

1440          TCPT_RANGESET(tp->t_rxtcur,
1441                      ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1,
1442                      tp->t_rttmin, TCPTV_REXMTMAX);
1443      }

```

—tcp_input.c

图27-8 `tcp_mss` 函数：判断路由是否有相应的 RTT 参数表

2. 初始化已平滑的 RTT 估计器

1420-1432 如果连接上不存在 RTT 样本值 (`t_srtt=0`)，并且 `rmx_rtt` 非零，则将后者赋

给已平滑的RTT估计器 t_srtt 。如果路由参数表锁定标志的 RTV_RTT 比特置位,表明连接的最小RTT(t_rttmin)也应初始化为 rmx_rtt 。前面介绍过, $tcp_newtcpcb$ 把 t_rttmin 初始化为2个滴答。

rmx_rtt (以微秒为单位)转换为 t_srtt (以8个滴答为单位),这是图27-4的反变换。注意, t_rttmin 等于 t_srtt 的1/8,因为前者没有除以缩放因子 TCP_RTT_SCALE 。

3. 初始化已平滑的RTT平均偏差估计器

1433-1439 如果存储的 rmx_rttvar (以微秒为单位)值非零,将其转换为 t_rttvar (以4个滴答为单位)。但如果为零,则 t_rttvar 等于 t_rtt ,即偏差等于均值。已平滑的RTT平均偏差估计器默认设置为 ± 1 RTT。由于 t_rttvar 的单位为4个滴答,而 t_rtt 的单位为8个滴答, t_srtt 值也必须做相应转换。

4. 计算初始RTO

1440-1442 计算当前的RTO,并存储在 t_rxtcu 中,采用下列算式更新:

$$RTO = srtt + 2 \times rttvar$$

计算第一个RTO时,乘数取2,而非4,上式与图25-21中用到的算式相同。将缩放关系代入,得到:

$$RTO = \frac{t_srtt}{8} + 2 \times \frac{t_rttvar}{4} = \frac{\frac{t_srtt}{4} + t_rttvar}{2}$$

即为 $TCPT_RANGESET$ 的第二个参数。

图27-9给出了 tcp_mss 的下一部分,计算MSS。

```

1444  /*
1445   * if there's an mtu associated with the route, use it
1446   */
1447   if (rt->rt_rmx.rmx_mtu)
1448       mss = rt->rt_rmx.rmx_mtu - sizeof(struct tcphdr);
1449   else {
1450       mss = ifp->if_mtu - sizeof(struct tcphdr);
1451 #if (MCLBYTES & (MCLBYTES - 1)) == 0
1452     if (mss > MCLBYTES)
1453         mss &= ~(MCLBYTES - 1);
1454 #else
1455     if (mss > MCLBYTES)
1456         mss = mss / MCLBYTES * MCLBYTES;
1457 #endif
1458     if (!in_localaddr(inp->inp_faddr))
1459         mss = min(mss, tcp_mssdflt);
1460 }

```

tcp_input.c

图27-9 tcp_mss 函数: 计算mss

5. 从路由表中的MTU得到MSS

1444-1450 如果路由表中的MTU有值,则将其赋给 mss 。如果没有,则 mss 初始值等于外接口的MTU值减去40(IP和TCP首部默认值)。对于以太网,MSS初始值应为1460。

6. 减小MSS, 令其等于MCLBYTES的倍数

1451-1457 如果 mss 大于MCLBYTES,则减小 mss 的值,令其等于最接近的

MCLBYTES(mbuf簇大小)的整数倍。如果MCLBYTES值(通常等于1024或2048)与MCLBYTES值减1逻辑与后等于0,说明MCLBYTES等于2的倍数。例如,1024(0x400)逻辑与1023(0x3ff)等于0。

代码通过清零mss的若干低位比特,将mss减小到最接近的MCLBYTES的倍数:如果mbuf簇大小为1024,mss与1023的二进制补码(0xfffffc00)逻辑与,低位的10 bit被清零。对于以太网,mss将从1460减至1024。如果mbuf簇大小为2048,与2047的二进制补码(0xffff8000)逻辑与,低位的11 bit被清零。对于令牌环,MTU大小为4464,上述运算将mss从4424减为4096。如果MCLBYTES不是2的倍数,代码用mss整数除以MCLBYTES后,再乘上MCLBYTES,从而将mss减小到最接近的MCLBYTES的倍数。

7. 判断目的地是本地地址还是远端地址

1458-1459 如果目的IP不是本地地址(in_localaddr返回零),且mss大于512(tcp_mssdflt),则将mss设为512。

IP地址是否为本地地址取决于全局变量subnetsarelocal,内核编译时把符号变量SUBNETSARELOCAL的值赋给它。默认值为1,意味着如果给定IP地址与主机任一接口的IP地址具有相同的网络ID,则被认为是一个本地地址。如果为0,则给定IP地址必须与主机任一接口的IP地址具有相同的网络号和子网号,才会被认为是一个本地地址。

对于非本地地址,将MSS最小化是为了避免IP数据报经广域网时被分片。绝大多数WAN链路的MTU只有1006,这是从ARPANET遗留下来的一个问题。在卷1的11.7节中讨论过,现代的多数WAN支持1500,甚至更大的MTU。感兴趣的读者还可阅读卷1的24.2节中讨论的路由MTU发现特性(RFC 1191, [Mogul and Deering 1990])。Net/3不支持路由MTU发现。

图27-10给出了tcp_mss最后一部分的代码。

8. 对端的MSS用作上限

1461-1472 如果tcp_mss被tcp_input调用,参数offer非零,等于对端通告的mss值。如果mss大于对端通告的值,则将offer赋给它。例如,如果函数计算得到的mss等于1024,但对端通告的值只有512,则mss必须被设定为512。相反,如果mss等于536(即输出MTU等于576),而对端通告的值为1460,TCP仍旧使用536。只要不超过对端通告的值,mss可以取小于它的任何一个值。如果tcp_mss被tcp_output调用,offer等于0,用于发送MSS选项。注意,尽管mss的上限可变,其下限固定为32。

1473-1483 如果mss小于tcp_newtcpcb中设定的默认值t_maxseg(512),或者如果TCP正在处理收到的MSS选项(offer非零),则需执行下列步骤。首先,如果路由的rmx_sendpipe有值,则采用它做为发送缓存的高端(high-water)标志(图16-4)。如果缓存小于mss,则使用较小的值。除非是应用程序有意把发送缓存定得很小,或者管理员将rmx_sendpipe定得很小,这种情况一般不会发生,因为发送缓存的上限默认值为8192,大于绝大多数的mss。

9. 增加缓存大小,令其等于最近的MSS整数倍

1484-1489 增加缓存大小,令其等于最近的mss整数倍,上限为sb_max(Net/3中定义为262 144,即256×1024)。插口发送缓存的上限设定为sbreserve。例如,上限默认值等于

8192, 但对于以太网上的本地 TCP 传输, 其 mbuf 簇大小为 2048 (假定 mss 等于 1460), 代码把上限值增加到 8760 (等于 6×1460)。但对于非本地的连接, mss 等于 512, 上限值保持 8192 不变。

```

1461      /*
1462      * The current mss, t_maxseg, was initialized to the default value
1463      * of 512 (tcp_mssdflt) by tcp_newtcpcb().
1464      * If we compute a smaller value, reduce the current mss.
1465      * If we compute a larger value, return it for use in sending
1466      * a max seg size option, but don't store it for use
1467      * unless we received an offer at least that large from peer.
1468      * However, do not accept offers under 32 bytes.
1469      */
1470      if (offer)
1471          mss = min(mss, offer);
1472      mss = max(mss, 32);          /* sanity */
1473      if (mss < tp->t_maxseg || offer != 0) {
1474          /*
1475          * If there's a pipesize, change the socket buffer
1476          * to that size. Make the socket buffers an integral
1477          * number of mss units; if the mss is larger than
1478          * the socket buffer, decrease the mss.
1479          */
1480          if ((bufsize = rt->rt_rmx.rmx_sendpipe) == 0)
1481              bufsize = so->so_snd.sb_hiwat;
1482          if (bufsize < mss)
1483              mss = bufsize;
1484          else {
1485              bufsize = roundup(bufsize, mss);
1486              if (bufsize > sb_max)
1487                  bufsize = sb_max;
1488              (void) sbreserve(&so->so_snd, bufsize);
1489          }
1490          tp->t_maxseg = mss;
1491          if ((bufsize = rt->rt_rmx.rmx_rcvpipe) == 0)
1492              bufsize = so->so_rcv.sb_hiwat;
1493          if (bufsize > mss) {
1494              bufsize = roundup(bufsize, mss);
1495              if (bufsize > sb_max)
1496                  bufsize = sb_max;
1497              (void) sbreserve(&so->so_rcv, bufsize);
1498          }
1499      }
1500      tp->snd_cwnd = mss;
1501      if (rt->rt_rmx.rmx_ssthresh) {
1502          /*
1503          * There's some sort of gateway or interface
1504          * buffer limit on the path. Use this to set
1505          * the slow start threshold, but set the
1506          * threshold to no less than 2*mss.
1507          */
1508          tp->snd_ssthresh = max(2 * mss, rt->rt_rmx.rmx_ssthresh);
1509      }
1510      return (mss);
1511 }

```

tcp_input.c

图27-10 tcp_mss 函数：结束处理

1490 由于 t_maxseg 已小于默认值 (512), 或者由于收到了对端发送的 MSS 选项, 所以应更

新它。

1491-1499 对接收缓存的处理与发送缓存相同。

10. 初始化拥塞窗口和慢启动门限

1500-1509 拥塞窗口的值, `snd_cwnd`, 等于一个最大报文段长度。如果路由表中的 `rmx_ssthresh` 非零, 慢启动门限(`snd_ssthresh`)初始化为该值, 但应保证其下限为两个最大报文段长度。

1510 函数最后返回 `mss`。`tcp_input` 忽略这一返回值(图28-10, 因为它已收到对端的 MSS 选项), 但图26-23中, `tcp_output` 将它用作 MSS 通告。

举例

下面通过一个连接建立的实例说明 `tcp_mss` 的操作过程。连接建立过程中, 它会被调用两次: 发送 SYN 时和收到对端带有 MSS 选项的 SYN 时。

1) 创建插口, `tcp_newtcpcb` 初始化 `t_maxseg` 为 512。

2) 应用进程调用 `connect`。为了在 SYN 报文段中加入 MSS 选项, `tcp_output` 调用 `tcp_mss`, 参数 `offer` 等于零。假定目的 IP 为本地以太网地址, `mbuf` 簇大小为 2048, 执行图 27-9 中的代码后, `mss` 等于 1460。由于 `offer` 等于零, 图 27-10 中的代码不修改 `mss` 值, 函数返回 1460。因为 1460 大于默认值 (512) 而且未收到对端的 MSS 选项, 缓存大小不变。`tcp_output` 发送 MSS 选项, 通告 MSS 大小为 1460。

3) 对端发送响应 SYN, 通告 `mss` 大小为 1024。`tcp_input` 调用 `tcp_mss`, 参数 `offer` 等于 1024。图 27-9 的代码逻辑仍旧设定 `mss` 为 1460, 但在图 27-10 起始处的 `min` 语句将 `mss` 减小为 1024。因为 `offer` 非零, 缓存大小增加至最近的 1024 的整数倍(等于 8192)。`t_maxseg` 更新为 1024。

初看上去, `tcp_mss` 的逻辑存在问题: TCP 向对端通告 `mss` 大小为 1460, 之后从对端收到的 `mss` 只有 1024。尽管 TCP 只能发送 1024 字节的报文段, 对端却能够发送 1460 字节的报文段。读者可能会认为发送缓存应等于 1024 的倍数, 而接收缓存则应等于 1460 的倍数。但图 27-10 中的代码却将两个缓存大小都设为对端通告的 `mss` 的倍数。这是因为尽管 TCP 通告 `mss` 为 1460, 但对端通告的 `mss` 仅为 1024, 对端有可能不会发送 1460 字节的报文段, 而将发送报文段限制为 1024 字节。

27.6 tcp_ctlinput 函数

回想图 22-32 中, `tcp_ctlinput` 处理 5 种类型的 ICMP 差错: 目的地不可达、数据报参数错、源站抑制、数据报超时和重定向。所有重定向差错会上交给相应的 TCP 或 UDP 进行处理。

对于其他 4 种差错, 仅当它们是被 TCP 报文段引发的, 才会调用 `tcp_ctlinput` 进行处理。

图 27-11 给出了 `tcp_ctlinput` 函数, 它与图 23-30 的 `udp_ctlinput` 函数类似。

365-366 在逻辑上, `tcp_ctlinput` 与 `udp_ctlinput` 的唯一区别是如何处理 ICMP 源站抑制差错。因为 `inetctlerrmap` 等于 0, UDP 忽略源站抑制差错。TCP 检查源站抑制差错, 并把 `notify` 函数的默认值 `tcp_notify` 改为 `tcp_quench`。

```

355 void
356 tcp_ctlinput(cmd, sa, ip)
357 int      cmd;
358 struct sockaddr *sa;
359 struct ip *ip;
360 {
361     struct tcphdr *th;
362     extern struct in_addr zero_in_addr;
363     extern u_char inetctlerrmap[];
364     void (*notify) (struct inpcb *, int) = tcp_notify;

365     if (cmd == PRC_QUENCH)
366         notify = tcp_quench;
367     else if (!PRC_IS_REDIRECT(cmd) &&
368             ((unsigned) cmd > PRC_NCMLS || inetctlerrmap[cmd] == 0))
369         return;
370     if (ip) {
371         th = (struct tcphdr *) ((caddr_t) ip + (ip->ip_hl << 2));
372         in_pcbnotify(&tcb, sa, th->th_dport, ip->ip_src, th->th_sport,
373                     cmd, notify);
374     } else
375         in_pcbnotify(&tcb, sa, 0, zero_in_addr, 0, cmd, notify);
376 }

```

tcp_subr.c

图27-11 tcp_ctlinput 函数

27.7 tcp_notify函数

tcp_notify被tcp_ctlinput调用，处理目的地不可达、数据报参数错、数据报超时和重定向差错。与UDP的差错处理函数相比，它要复杂得多，因为TCP必须灵活地处理连接上收到的各种软差错。图27-12给出了tcp_notify函数。

```

328 void
329 tcp_notify(inp, error)
330 struct inpcb *inp;
331 int      error;
332 {
333     struct tcpcb *tp = (struct tcpcb *) inp->inp_ppcb;
334     struct socket *so = inp->inp_socket;

335     /*
336      * Ignore some errors if we are hooked up.
337      * If connection hasn't completed, has retransmitted several times,
338      * and receives a second error, give up now. This is better
339      * than waiting a long time to establish a connection that
340      * can never complete.
341      */
342     if (tp->t_state == TCPS_ESTABLISHED &&
343         (error == EHOSTUNREACH || error == ENETUNREACH ||
344          error == EHOSTDOWN)) {
345         return;
346     } else if (tp->t_state < TCPS_ESTABLISHED && tp->t_rxtshift > 3 &&
347                tp->t_softerror)
348         so->so_error = error;
349 }

```

tcp_subr.c

图27-12 tcp_notify 函数

```

349     else
350         tp->t_softerror = error;
351         wakeup((caddr_t) & so->so_timeo);
352         sorwakeup(so);
353         sowwakeup(so);
354 }

```

—tcp_subr.c

图27-12 (续)

328-345 如果连接状态为 ESTABLISHED，则忽略 EHOSTUNREACH、ENETUNREACH 和 EHOSTDOWN 差错代码。

处理这3个差错是4.4BSD中新增的功能。Net/2及早期版本在连接的软差错变量 (t_softerror) 中记录这些差错，如果连接最终失败，则向应用进程返回相应的差错码。回想一下，tcp_xmit_timer 在收到一个 ACK，确认未发送过的报文段时，复位 t_softerror 为零。

346-353 如果连接还未建立，而且 TCP 已经至少4次重传了当前报文段，t_softerror 中已存在差错记录，则最新的差错将被保存在插口的 so_error 变量中，从而应用进程可以调用 select 对插口进行读写。如果上述条件不满足，当前差错将仍旧保存在 t_softerror 中。我们在 tcp_drop 函数中讨论过，如果连接最终由于超时而被丢弃，tcp_drop 会把 t_softerror 赋给插口差错变量 errno。任何在插口上等待接收或发送数据的应用进程会被唤醒，并得到相应的差错代码。

27.8 tcp_quench 函数

tcp_quench 的函数代码在图 27-13 中给出。TCP 在两种情况下调用它：当连接上收到源站抑制差错时，由 tcp_input 调用。当 ip_output 返回 ENOBUFS 差错代码时，由 tcp_output 调用。

```

381 void
382 tcp_quench(inp, errno)
383 struct inpcb *inp;
384 int         errno;
385 {
386     struct tcpcb *tp = intotcp(inp);
387     if (tp)
388         tp->snd_cwnd = tp->t_maxseg;
389 }

```

—tcp_subr.c

—tcp_subr.c

图27-13 tcp_quench 函数

拥塞窗口设定为最大报文段长度，强迫 TCP 执行慢启动。慢启动门限不变（与 tcp_timers 处理重传超时的思想相同），因此，窗口大小将成指数地增加，直至达到 snd_ssthresh 门限或发生拥塞。

27.9 TCP_REASS 宏和 tcp_reass 函数

TCP 报文段有可能乱序到达，因此，在数据上交给应用进程之前，TCP 必须设法恢复正确

的报文段次序。例如，如果接收方的接收窗口大小为 4096，等待接收的下一个序号为 0。收到的第一个报文段携带 0 ~ 1023 字节的数据（次序正确），第二个报文段携带了 2048 ~ 3071 字节的数据，很明显，第二个报文段到达的次序差错。如果乱序报文段位于接收窗口内，TCP 并不丢弃它，而是将其保存在连接的重组队列中，继续等待中间缺失的报文段（携带 1024 ~ 2047 字节的报文段）。这一节我们将讨论处理 TCP 重组队列的代码，为后两章讨论 `tcp_input` 打下基础。

如果假定某个 mbuf 中包含 IP 首部、TCP 首部和 4 字节的用户数据（回想图 2-14 的左半部分），如图 27-14 所示。此外还假定数据的序号依次为 7、8、9 和 10。

图 24-12 中定义的 `tcpihdr` 结构里包含了 `ipovly` 和 `tcphdr` 两个结构，`tcphdr` 结构在图 24-12 中给出。图 27-14 只列出了与重组有关的一些变量：`ti_next`、`ti_prev`、`ti_len`、`ti_dport` 和 `ti_seq`。头两个指针指向由给定连接所有乱序报文段组成的双向链表。链表头保存在连接的 TCP 控制块中：结构的头两个成员变量为 `seg_next` 和 `seg_prev`。`ti_next` 和 `ti_prev` 指针与 IP 首部的头 8 个字节重复，只要数据报到达了 TCP，就不再需要这些内容。`ti_len` 等于 TCP 数据的长度，TCP 计算检验和之前首先计算并存储这个字段。

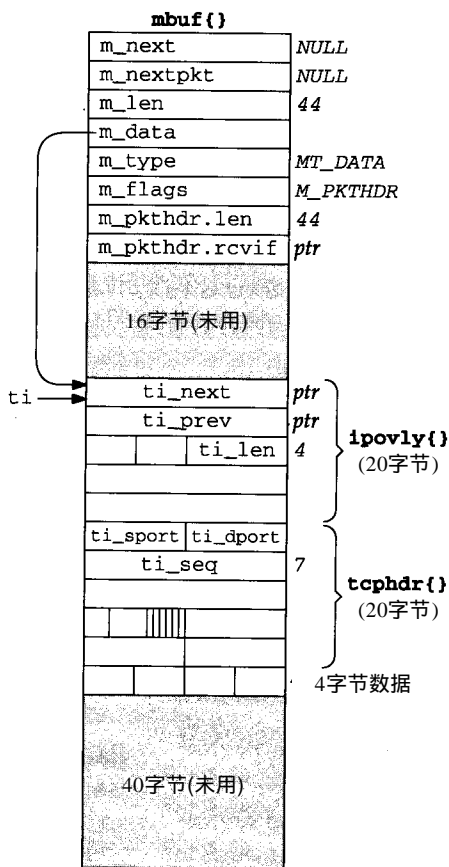


图 27-14 举例：带有 4 字节数据的 IP 和 TCP 首部

27.9.1 TCP_REASS 宏

`tcp_input` 收到数据后，就调用图 27-15 中的宏 `TCP_REASS`，把数据放入连接的重组队列。TCP_REASS 只在一种情况下被调用：参见图 29-22。

54-63 `tp` 是指向连接 TCP 控制块的指针，`ti` 是指向接收报文段的 `tcpihdr` 结构的指针。如果下列 3 个条件均为真：

- 1) 报文段到达次序正确（序号 `ti_seq` 等于连接上等待接收的下一序号，`rcv_nxt`）；并且
- 2) 连接的重组队列为空（`seg_next` 指向自己，而不是某个 mbuf）；并且
- 3) 连接处于 ESTABLISHED 状态。

则执行下列步骤：设定延迟 ACK 标志；更新 `rcv_nxt`，增加报文段携带的数据长度；如果报文段 TCP 首部中 FIN 标志置位，则 `flags` 参数中增加 `TH_FIN` 标志；更新两个统计值；数据放入插口的接收缓存；唤醒所有在插口上等待接收的应用进程。

```
53 #define TCP_REASS(tp, ti, m, so, flags) { \
54     if ((ti)->ti_seq == (tp)->rcv_nxt && \
55         (tp)->seg_next == (struct tcphdr *) (tp) && \
56         (tp)->t_state == TCPS_ESTABLISHED) { \
57         tp->t_flags |= TF_DELACK; \
58         (tp)->rcv_nxt += (ti)->ti_len; \
59         flags = (ti)->ti_flags & TH_FIN; \
60         tcpstat.tcps_rcvpack++; \
61         tcpstat.tcps_rcvbyte += (ti)->ti_len; \
62         sbappend(&(so)->so_rcv, (m)); \
63         sorwakeup(so); \
64     } else { \
65         (flags) = tcp_reass((tp), (ti), (m)); \
66         tp->t_flags |= TF_ACKNOW; \
67     } \
68 }
```

tcp_input.c

tcp_input.c

图27-15 TCP_REASS 宏：向连接的重组队列中添加数据

必须满足前述3个条件的原因是：首先，如果数据次序差错，则必须将其放入重组队列，直至收到了中间缺失的报文段，才能把数据提交给应用进程。第二，即使当前数据到达次序正确，但如果重组队列中已存在乱序数据，则新的数据有可能就是所需的缺失数据，从而能够向应用进程同时提交多个报文段中的数据；第三，尽管允许请求建立连接的 SYN 报文段中携带数据，但这些数据在连接进入 ESTABLISHED 状态之前，必须保存在重组队列中，不允许直接提交给应用进程。

64-67 如果这3个条件不是同时满足，则 TCP_REASS 宏调用 TCP_REASS 函数，向重组队列中添加数据。由于收到的报文段如果不是乱序报文段，就有可能是所需的缺失报文段，因此，置位 TF_ACKNOW，要求立即发送 ACK。TCP 的一个重要特性是收到乱序报文段时，必须立即发送 ACK，这有助于快速重传算法 (29.4 节) 的实现。

在讨论 TCP_REASS 函数代码之前，需要先了解图 27-14 中 TCP 首部的两个端口号，ti_sport 和 ti_dport，所起的作用。其实，只要找到了 TCP 控制块并调用了 TCP_REASS，就不再需要它们了。因此，TCP 报文段放入重组队列时，可以把对应 mbuf 的地址存储在这两个端口号变量中。对于图 27-14 中的报文段，无需这样做，因为 IP 和 TCP 的首部都存储在 mbuf 的数据部分，可直接使用 dtom 宏。但我们在 2.6 节讨论 m_pullup 时曾指出，如果 IP 和 TCP 的首部保存在簇中 (如图 2-16 所示，对于最大长度报文这是很正常的)，dtom 宏将无法使用。我们在该节中曾提到，TCP 把从 TCP 首部指向 mbuf 的后向指针 (back pointer) 存储在 TCP 的两个端口号字段中。

图 27-16 举例说明了这一技术的用法，利用它处理连接上的两个乱序报文段，每个报文段都存储在一个 mbuf 簇中。乱序报文段双向链表的表头是连接的 TCP 控制块中的 seg_next 成员变量。为简化起见，图中未标出 seg_prev 指针和指向链表最后一个报文段的 ti_next 指针。

接收窗口等待接收的下一个序号为 l(rcv_nxt)，但我们假定这个报文段丢失了。接着又收到了两个报文段，携带 1461~4380 字节的数据，这是两个乱序报文段。TCP 调用 m_devget 把它们放入 mbuf 簇中，如图 2-16 所示。

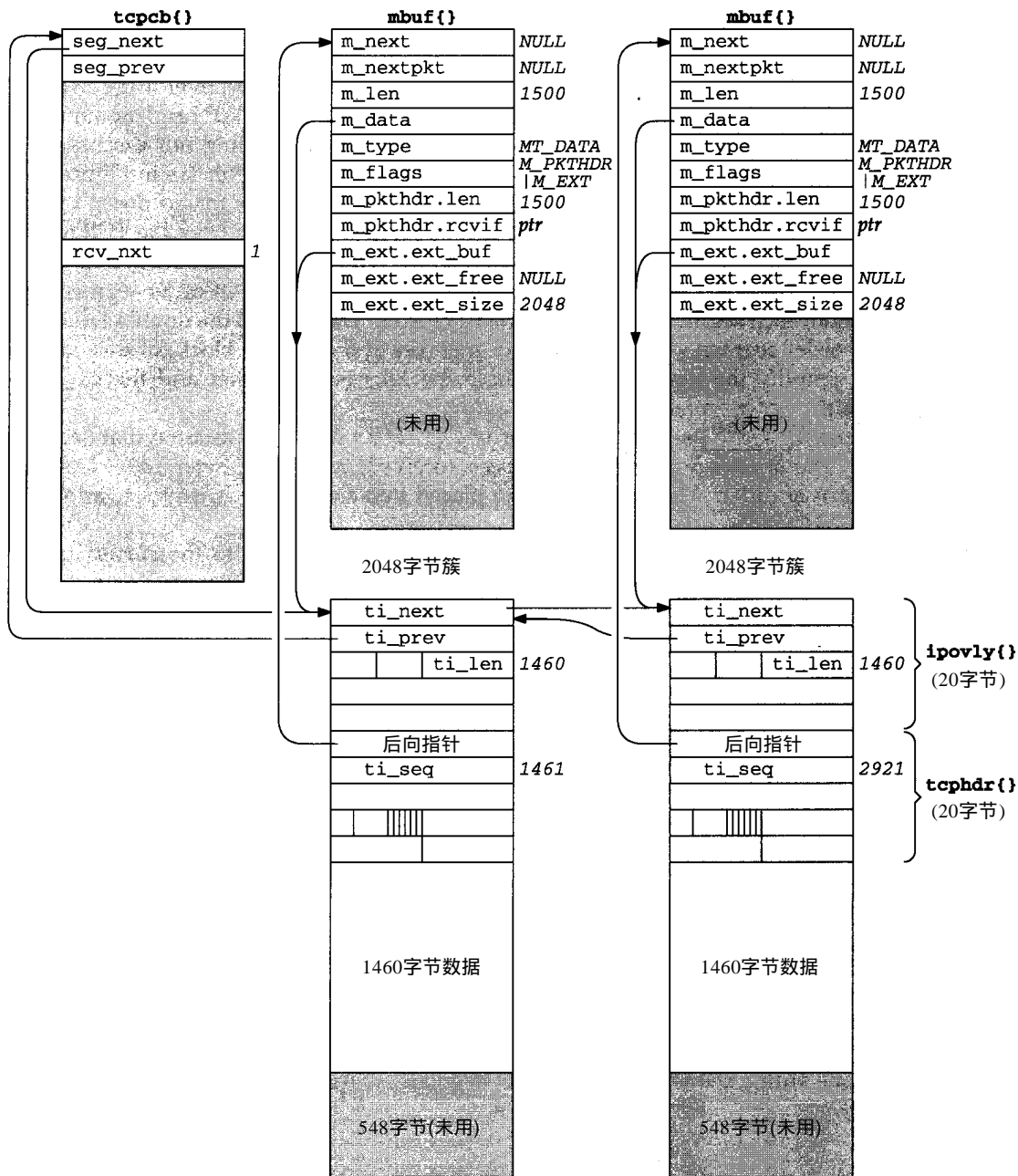


图27-16 两个乱序TCP报文段存储在mbuf簇中

TCP首部的头32 bit存储指向对应mbuf的指针，下面介绍的TCP_REASS函数将用到这个后向指针。

27.9.2 TCP_REASS函数

图27-17给出了TCP_REASS函数的第一部分。参数包括：`tp`，指向TCP控制块的指针；`ti`，指向接收报文段IP和TCP首部的指针；`m`，指向存储接收报文段的mbuf链表的指针。前

面曾提到过，*ti*既可以指向由*m*所指向的mbuf的数据区，也可以指向一个簇。

```

69 int
70 tcp_reass(tp, ti, m)
71 struct tcpcb *tp;
72 struct tcphdr *ti;
73 struct mbuf *m;
74 {
75     struct tcphdr *q;
76     struct socket *so = tp->t_inpcb->inp_socket;
77     int flags;
78
79     /*
80      * Call with ti==0 after become established to
81      * force pre-ESTABLISHED data up to user socket.
82      */
83     if (ti == 0)
84         goto present;
85
86     /*
87      * Find a segment that begins after this one does.
88      */
89     for (q = tp->seg_next; q != (struct tcphdr *) tp;
90          q = (struct tcphdr *) q->ti_next)
91         if (SEQ_GT(q->ti_seq, ti->ti_seq))
92             break;

```

tcp_input.c

tcp_input.c

图27-17 TCP_REASS 函数：第一部分

69-83 后面将看到，TCP收到一个对SYN的确认时，*tcp_input*将调用TCP_REASS，并传递一个空的*ti*指针(图28-20和图29-2)。这意味着连接已建立，可以把SYN报文段中携带的数据(TCP_REASS已将其放入重组队列)提交给应用程序。连接未建立之前，不允许这样做。标志“present”位于图27-23中。

84-90 遍历从*seg_next*开始的乱序报文段双向链表，寻找序号大于接收报文段序号(*ti_seq*)的第一个报文段。注意，for循环体中只包含一个if语句。

图27-18的例子中，新报文段到达时重组队列中已有两个报文段。图中标出了指针*q*，指向链表的下一个报文段，带有字节10~15。此外，图中标出了两个指针*ti_next*和*ti_prev*，起始序号(*ti_seq*)、长度(*ti_len*)和数据字节的序号。由于这些报文段较小，每个报文段很可能存储在单一的mbuf中，如图27-14所示。

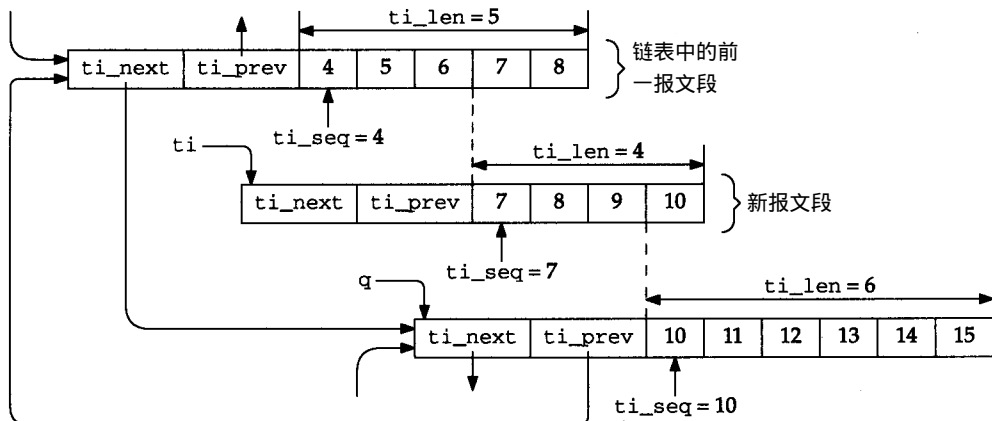


图27-18 存储重复报文段的重组队列举例

图27-19给出了TCP_REASS下一部分的代码

```

91      /*
92      * If there is a preceding segment, it may provide some of
93      * our data already. If so, drop the data from the incoming
94      * segment. If it provides all of our data, drop us.
95      */
96      if ((struct tcphdr *) q->ti_prev != (struct tcphdr *) tp) {
97          int i;
98          q = (struct tcphdr *) q->ti_prev;
99          /* conversion to int (in i) handles seq wraparound */
100         i = q->ti_seq + q->ti_len - ti->ti_seq;
101         if (i > 0) {
102             if (i >= ti->ti_len) {
103                 tcpstat.tcps_rcvduppack++;
104                 tcpstat.tcps_rcvduppack += ti->ti_len;
105                 m_freem(m);
106                 return (0);
107             }
108             m_adj(m, i);
109             ti->ti_len -= i;
110             ti->ti_seq += i;
111         }
112         q = (struct tcphdr *) (q->ti_next);
113     }
114     tcpstat.tcps_rcvooack++;
115     tcpstat.tcps_rcvooack += ti->ti_len;
116     REASS_MBUF(ti) = m; /* XXX */

```

tcp_input.c

图27-19 TCP_REASS 函数：第二部分

91-107 如果双向链表中q指向的报文段前还存在报文段，则该报文段有可能与新报文段重复，因此，挪动指针q，令其指向q的前一个报文段(图27-18中携带字节4~8的报文段)，计算重复的字节数，并存储在变量i中：

```

i = q->ti_seq + q->ti_len - ti->ti_seq;
  = 4 + 5 - 7
  = 2

```

如果i大于0，则链表中原有报文段与新报文段携带的数据间存在重复，如例子中给出的报文段。如果重复的字节数(i)大于或等于新报文段的大小，即新报文段中所有的数据都已包含在原有报文段中，新报文段是重复报文段，应予以丢弃。

108-112 如果只有部分数据重复(如图27-18所示)，m_adj丢弃新报文段起始i字节的数据，并相应更新新报文段的序号和长度。挪动q指针，指向链表中的下一个报文段。图27-20给出了图27-18中各报文段和变量此时的状态。

116 mbuf的地址m存储在TCP首部的源端口号和目的端口号中，也就是我们在本节前面曾提到的后向指针，防止TCP首部被存放在mbuf簇中，而无法使用dtom宏。宏REASS_MBUF定义为：

```
#define REASS_MBUF(ti) (*(struct mbuf **)&((ti)->ti_t))
```

ti_t是一个tcphdr结构(图24-12)，最初的两个成员变量是两个16bit的端口号。请注意图27-19中的注释“XXX”，其中隐含了这样一个假定，指针能够存放在两个端口号占用的32bit空间中。

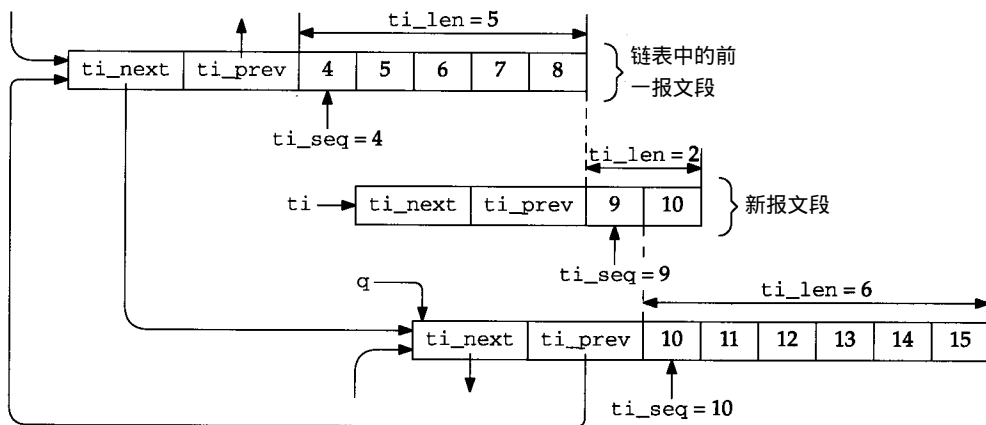


图27-20 删除新报文段中的字节7和8后，更新图27-18

图27-21给出了tcp_reass的第三部分，删除重组队列下一报文段中可能的重复字节。

117-135 如果还有后续报文段，则计算新报文段与下一报文段间重复的字节数，并存储在变量*i*中。还是以图27-18中的报文段为例，得到：

$$\begin{aligned} i &= 9 + 2 - 10 \\ &= 1 \end{aligned}$$

因为序号10的字节同时存在于两个报文段中。

根据*i*值的大小，有可能出现3种情况：

1) 如果*i*小于等于0，无重复。

2) 如果*i*小于下一报文段的字节数(*q*→*ti_len*)，则有部分重复，调用m_adj，从该报文段中丢弃起始的*i*字节。

3) 如果*i*大于等于下一报文段的字节数，则出现完全重复，从链表中删除该报文段。

136-139 代码最后调用insque，把新报文段插入连接的重组双向链表中。图27-22给出了图27-18中各报文段和变量此时的状态。

```

117  /*
118  * While we overlap succeeding segments trim them or,
119  * if they are completely covered, dequeue them.
120  */
121  while (q != (struct tcpiphdr *) tp) {
122      int i = (ti->ti_seq + ti->ti_len) - q->ti_seq;
123      if (i <= 0)
124          break;
125      if (i < q->ti_len) {
126          q->ti_seq += i;
127          q->ti_len -= i;
128          m_adj(REASS_MBUF(q), i);
129          break;
130      }
131      q = (struct tcpiphdr *) q->ti_next;
132      m = REASS_MBUF((struct tcpiphdr *) q->ti_prev);
133      remque(q->ti_prev);

```

图27-21 TCP_REASS 函数：第三部分

```

134     m_freem(m);
135 }
136 /*
137  * Stick new segment in its place.
138  */
139 insque(ti, q->ti_prev);

```

tcp_input.c

图27-21 (续)

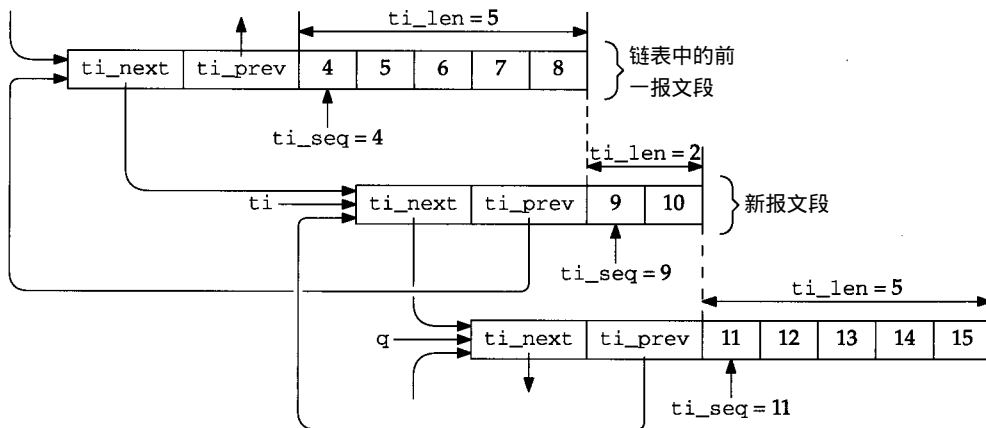


图27-22 丢弃所有重复字节后，更新图 27-20

图27-23给出了tcp_reass最后部分的代码，如果可能，向应用进程递交数据。

```

140 present:
141 /*
142  * Present data to user, advancing rcv_nxt through
143  * completed sequence space.
144  */
145 if (TCPS_HAVERCVDSYN(tp->t_state) == 0)
146     return (0);
147 ti = tp->seg_next;
148 if (ti == (struct tcpiphdr *) tp || ti->ti_seq != tp->rcv_nxt)
149     return (0);
150 if (tp->t_state == TCPS_SYN_RECEIVED && ti->ti_len)
151     return (0);
152 do {
153     tp->rcv_nxt += ti->ti_len;
154     flags = ti->ti_flags & TH_FIN;
155     remque(ti);
156     m = REASS_MBUF(ti);
157     ti = (struct tcpiphdr *) ti->ti_next;
158     if (so->so_state & SS_CANTRCVMORE)
159         m_freem(m);
160     else
161         sbappend(&so->so_rcv, m);
162 } while (ti != (struct tcpiphdr *) tp && ti->ti_seq == tp->rcv_nxt);
163 sorwakeup(so);
164 return (flags);
165 }

```

tcp_input.c

图27-23 tcp_reass 函数：第四部分

145-146 如果连接还没有收到SYN(连接处于LISTEN状态或SYN_SENT状态),不允许向应用进程提交数据,函数返回。当函数被宏 `TCP_REASS`调用时,返回值0被赋给宏的参数 `flags`。这种做法带来的副作用是可能会清除FIN标志。当宏 `TCP_REASS`被图29-22的代码调用时,如果接收报文段包含了SYN、FIN和数据(尽管不常见,但却是有有效的报文段),会出现这种情况。

147-149 `ti`设定为链表的第一个报文段。如果链表为空,或者第一个报文段的起始序号(`ti->ti_seq`)不等于连接等待接收的下一序号(`rcv_nxt`),则函数返回0。如果第二个条件为真,说明在等待接收的下一序号与已收到的数据之间仍然存在缺失报文段。例如,图 27-22中,如果携带4~8字节的报文段是链表的起始报文段,但 `rcv_nxt`等于2,字节2和3仍旧缺失,因此,不能把4~15字节提交给应用进程。返回值0将清除FIN标志(如果该标志设定),这是因为还有未收到的数据,所以暂时不能处理FIN。

150-151 如果连接处于SYN_RCVD状态,且报文段长度非零,则函数返回0。如果两个条件均为真,说明插口在监听过程中收到了携带数据的SYN报文段。数据将保存在连接队列中,等待三次握手过程结束。

152-164 循环从链表的第一个报文段开始(从前面的测试条件可知,它携带数据的次序已经正确),把数据放入插口的接收缓存,并更新 `rcv_nxt`。当链表为空,或者链表下一报文段的序号又出现差错,即当前处理报文段与下一报文段间存在缺失报文段时,循环结束。此时, `flags`变量(函数的返回值)等于0或者为 `TH_FIN`,取决于放入插口接收缓存的最后一个报文段中是否带有FIN标志。

在所有mbuf都放入插口的接收缓存后, `sorwakeup`唤醒所有在插口上等待接收数据的应用进程。

27.10 tcp_trace函数

图26-32中,在向IP递交报文段之前, `tcp_output`调用了 `tcp_trace`函数:

```
if (so->so_options & SO_DEBUG)
    tcp_trace(TA_OUTPUT, tp->t_state, tp, ti, 0);
```

在内核的环形缓存中添加一条记录,这些记录可通过 `trpt` (8)程序读取。此外,如果内核编译时定义了符号 `TCPDEBUG`,并且变量 `tcpconsdebug`非零,则信息将输出到系统控制台。

任何进程都可以设定TCP的插口选项 `SO_DEBUG`,要求TCP把信息存储到内核的环形缓存中。但只有特权进程或系统管理员才能运行 `trpt`,因为它必须读取系统内存才能获取这些信息。

尽管可以为任何类型的插口设定 `SO_DUBUG`选项(如UDP或原始IP),但只有TCP才会处理它。

这些信息被保存在 `tcp_debug`结构中,如图27-24所示。

35-43 `tcp_debug`很大(196字节),因为它包含了其他两个结构:保存IP和TCP首部的 `tcpihdr`和完整的TCP控制块 `tcpcb`。由于保存了TCP控制块,其中的任何变量都可通过 `trpt`打印出来。也就是说,如果 `trpt`标准输出中没有包含读者感兴趣的信息,可修改源代码以打印控制块中任何想要的信息(Net/3版支持这种修改)。图25-28中的RTT变量就是通过这种方式得到的。

```

35 struct tcp_debug {
36     n_time    td_time;           /* iptime(): ms since midnight, UTC */
37     short     td_act;           /* TA_XXX value (Figure 27.25) */
38     short     td_ostate;        /* old state */
39     caddr_t    td_tcb;          /* addr of TCP connection block */
40     struct tcpiphdr td_ti;      /* IP and TCP headers */
41     short     td_req;           /* PRU_XXX value for TA_USER */
42     struct tcpcb td_cb;         /* TCP connection block */
43 };

53 #define TCP_NDEBBUG 100
54 struct tcp_debug tcp_debug[TCP_NDEBBUG];
55 int      tcp_debx;

```

tcp_debug.h

图27-24 tcp_debug 结构

53-55 图27-24还定义了数组 `tcp_debug`，也就是前面提到的环形缓存。数组指针 (`tcp_debx`)初始化为零，该数组约占 20 000 字节。

内核只调用了 `tcp_trace` 4次，每次调用都会在结构的 `td_act` 变量中存入一个不同的值，如图27-25所示。

td_act	描 述	参 考
TA_DROP	当输入报文段被丢弃时，被 <code>tcp_input</code> 调用	图29-27
TA_INPUT	输入处理完毕后，调用 <code>tcp_output</code> 之前	图29-26
TA_OUTPUT	调用 <code>ip_output</code> 发送报文段之前	图26-32
TA_USER	RPU_XXX 请求处理完毕后，被 <code>tcp_usrreq</code> 调用	图30-1

图27-25 td_act 值及相应的 tcp_trace 调用

图27-26给出了 `tcp_trace` 函数的主要部分，我们忽略了直接输出到控制台的那部分代码。

48-133 在函数被调用时，`ostate` 中保存了连接的前一个状态，与连接的当前状态 (保存在控制块中) 相比较，可了解连接的状态变迁状况。图 27-25 中，`TA_OUTPUT` 不改变连接状态，但其他3个调用则会导致状态的转移。

```

48 void
49 tcp_trace(act, ostate, tp, ti, req)
50 short  act, ostate;
51 struct tcpcb *tp;
52 struct tcpiphdr *ti;
53 int     req;
54 {
55     tcp_seq seq, ack;
56     int     len, flags;
57     struct tcp_debug *td = &tcp_debug[tcp_debx++];

58     if (tcp_debx == TCP_NDEBBUG)
59         tcp_debx = 0;           /* circle back to start */

60     td->td_time = iptime();
61     td->td_act = act;

```

tcp_debug.c

图27-26 tcp_trace 函数：在内核的环形缓存中保存信息

```

62     td->td_ostate = ostate;
63     td->td_tcb = (caddr_t) tp;
64     if (tp)
65         td->td_cb = *tp;          /* structure assignment */
66     else
67         bzero((caddr_t) & td->td_cb, sizeof(*tp));
68     if (ti)
69         td->td_ti = *ti;          /* structure assignment */
70     else
71         bzero((caddr_t) & td->td_ti, sizeof(*ti));
72     td->td_req = req;

73 #ifdef TCPDEBUG
74     if (tcpconsdebug == 0)
75         return;

        /* output information on console */

132 #endif
133 }

```

tcp_debug.c

图27-26 (续)

输出举例

图27-27列出了tcpdump输出的前4行，反映25.12节例子中的三次握手过程和发送的第一个数据报文段(卷1附录A提供了tcpdump输出格式的细节)。

```

1  0.0                bsdi.1025 > vangogh.discard: S 20288001:20288001(0)
                                win 4096 <mss 512>
2  0.362719 (0.3627)  vangogh.discard > bsdi.1025: S 3202722817:3202722817(0)
                                ack 20288002 win 8192
                                <mss 512>
3  0.364316 (0.0016)  bsdi.1025 > vangogh.discard: . ack 1 win 4096
4  0.415859 (0.0515)  bsdi.1025 > vangogh.discard: . 1:513(512) ack 1 win 4096

```

图27-27 反映图25-28实例的tcpdump 输出

图27-28列出了与之对应的trpt的输出。

图27-28的输出与正常的trpt输出相比略有一些不同：32 bit的数字序号显示为无符号整数(trpt将其差错地打印为有符号整数)；有些trpt按16进制输出的值被改为10进制；为了编制图25-28，作者人为地把从t_rtt到t_rxtcur的值加入到trpt中。

```

953738 SYN_SENT: output 20288001:20288005(4) @0 (win=4096)
<SYN> -> SYN_SENT
rcv_nxt 0, rcv_wnd 0
snd_una 20288001, snd_nxt 20288002, snd_max 20288002
snd_wll 0, snd_wl2 0, snd_wnd 0
REXMT=12 (t_rxtshift=0), KEEP=150

```

图27-28 反映图25-28实例的trpt 输出

```

t_rtt=1, t_srtt=0, t_rttvar=24, t_rxtcur=12

953739 CLOSED: user CONNECT -> SYN_SENT
rcv_nxt 0, rcv_wnd 0
snd_una 20288001, snd_nxt 20288002, snd_max 20288002
snd_wll 0, snd_wl2 0, snd_wnd 0
REXMT=12 (t_rxtshift=0), KEEP=150
t_rtt=1, t_srtt=0, t_rttvar=24, t_rxtcur=12

954103 SYN_SENT: input 3202722817:3202722817(0) @20288002 (win=8192)
<SYN,ACK> -> ESTABLISHED
rcv_nxt 3202722818, rcv_wnd 4096
snd_una 20288002, snd_nxt 20288002, snd_max 20288002
snd_wll 3202722818, snd_wl2 20288002, snd_wnd 8192
KEEP=14400
t_rtt=0, t_srtt=16, t_rttvar=4, t_rxtcur=6

954103 ESTABLISHED: output 20288002:20288002(0) @3202722818 (win=4096)
<ACK> -> ESTABLISHED
rcv_nxt 3202722818, rcv_wnd 4096
snd_una 20288002, snd_nxt 20288002, snd_max 20288002
snd_wll 3202722818, snd_wl2 20288002, snd_wnd 8192
KEEP=14400
t_rtt=0, t_srtt=16, t_rttvar=4, t_rxtcur=6

954153 ESTABLISHED: output 20288002:20288514(512) @3202722818 (win=4096)
<ACK> -> ESTABLISHED
rcv_nxt 3202722818, rcv_wnd 4096
snd_una 20288002, snd_nxt 20288514, snd_max 20288514
snd_wll 3202722818, snd_wl2 20288002, snd_wnd 8192
REXMT=6 (t_rxtshift=0), KEEP=14400
t_rtt=1, t_srtt=16, t_rttvar=4, t_rxtcur=6

```

图27-28 (续)

在时刻953 738，发送SYN。注意，代码中的时间变量有8位数字，以毫秒为单位，这里只输出了低6位。输出的结束序号(20 288 005)是差错的。SYN中确实携带了4字节的内容，但并非数据，而是MSS选项。重传定时器设定为6秒(REXMT)，保活定时器为75秒(KEEP)，这些定时器值均以500 ms滴答为单位。t_rtt等于1，意味对该报文段计时，测量RTT样本值。

发送SYN是为了响应应用进程的connect调用。一毫秒后，这次系统调用的信息被写入内核的环形缓存。尽管是因为应用进程调用了connect，才导致发送SYN报文段，但TCP在处理完PRU_CONNECT请求后，才调用tcp_trace，环形缓存中实际写入了两条记录。此外，应用进程调用connect时，连接状态为CLOSED，发送完SYN后，状态变迁至SYN_SENT，这也是两条记录仅有的不同之处。

第三条记录，时刻954 103，与第一条记录相隔365 ms (tcpdump显示时间差为362.7ms)，即为图25-28中“实际时间差(ms)”一栏的填充值。收到带有SYN和ACK的报文段后，连接状态从SYN_SENT转移到ESTABLISHED。因为计时报文段已得到确认，更新RTT估计器值。

第四条记录反映了三次握手过程中的第三个报文段：确认对端的SYN。因为是纯ACK报文段，不用对它计时(rtt等于0)，它在时刻954 103被发送。connect系统调用返回，应用进程接着调用write发送数据，产生TCP输出。

第五条记录反映了这个数据报文段，在时刻954 153，三次握手结束后50 ms，被发送。它携带50字节的数据，起始序号为20 288 002。重传定时器设为3秒，需要计时。

应用进程继续调用write发送数据。尽管不再显示更多记录，但很明显，接下来的3条记

录也都是在TCP处理完PRU_SEND请求后写入环形缓存的。第一次PRU_SEND请求，生成我们已看到的第一个512字节的输出报文段，其他3次请求不会引发TCP输出报文段，此时连接正处于慢起动状态。只生成4条记录是因为，图25-28的例子中的TCP发送缓存大小只有4096，mbuf簇大小为1024。一旦发送缓存被占满，应用进程就进入休眠状态。

27.11 小结

本章介绍了各种TCP函数，为后续章节打下基础。

TCP连接正常关闭时，向对端发送FIN，并等待4次报文交换过程结束。它被丢弃时，只需发送RST。

路由表中的每条记录都包含8个变量，其中有3个在连接关闭时更新，有6个用于新连接的建立，从而内核能够跟踪与同一目标之间建立的正常连接的某些特性，如RTT估计器值和慢起动门限。系统管理员可以设置或锁定部分变量，如MTU、接收管道大小和发送管道大小，这些特性会影响到达该目标的连接的性能。

TCP对收到的ICMP差错有一定的容错性——不会导致TCP终止已建立的连接。Net/3处理ICMP差错的方式与早期的Berkeley版本不同。

TCP报文段可能乱序到达，并包含重复数据，TCP必须处理这些异常现象。TCP为每条连接维护一个重组队列，保存乱序报文段，处理之后再提交给应用进程。

最后介绍了选定插口选项SO_DEBUG时，内核中保存的信息。除某些程序如tcpdump之外，这些内容也是很有用的调试工具。

习题

- 27.1 为什么图27-1中最后一行的errno等于0？
- 27.2 rmx_rtt中存储的最大值是多少？
- 27.3 为了保存某个给定主机的路由信息（图27-3），我们用手工在本地的路由表中添加一条到达该主机的路由。之后，运行FTP客户程序，向这台主机发送足够多的数据，如图27-4所要求的。但终止FTP客户程序后，检查路由表，到达该主机的所有变量依旧为0。出了什么问题？