# 第21章 TCP的超时与重传

## 21.1 引言

TCP提供可靠的运输层。它使用的方法之一就是确认从另一端收到的数据。但数据和确认都有可能会丢失。TCP通过在发送时设置一个定时器来解决这种问题。如果当定时器溢出时还没有收到确认，它就重传该数据。对任何实现而言，关键之处就在于超时和重传的策略，即怎样决定超时间隔和如何确定重传的频率。

我们已经看到过两个超时和重传的例子：(1) 在6.5节的ICMP端口不能到达的例子中，看到TFTP客户使用UDP实现了一个简单的超时和重传机制：假定5秒是一个适当的时间间隔，并每隔5秒进行重传；(2) 在向一个不存在的主机发送ARP的例子中（第4.5节），我们看到当TCP试图建立连接的时候，在每个重传之间使用一个较长的时延来重传SYN。

对每个连接，TCP管理4个不同的定时器。

1) 重传定时器使用于当希望收到另一端的确认。在本章我们将详细讨论这个定时器以及一些相关的问题，如拥塞避免。

2) 坚持(persist)定时器使窗口大小信息保持不断流动，即使另一端关闭了其接收窗口。第22章将讨论这个问题。

3) 保活(keepalive)定时器可检测到一个空闲连接的另一端何时崩溃或重启。第23章将描述这个定时器。

4) 2MSL定时器测量一个连接处于TIME_WAIT状态的时间。我们在18.6节对该状态进行了介绍。

本章以一个简单的TCP超时和重传的例子开始，然后转向一个更复杂的例子。该例子可以使我们观察到TCP时钟管理的所有细节。可以看到TCP的典型实现是怎样测量TCP报文段的往返时间以及TCP如何使用这些测量结果来为下一个将要传输的报文段建立重传超时时间。接着我们将研究TCP的拥塞避免——当分组丢失时TCP所采取的动作——并提供一个分组丢失的实际例子，我们还将介绍较新的快速重传和快速恢复算法，并介绍该算法如何使TCP检测分组丢失比等待时钟超时更快。

## 21.2 超时与重传的简单例子

首先观察TCP所使用的重传机制，我们将建立一个连接，发送一些分组来证明一切正常，然后拔掉电缆，发送更多的数据，再观察TCP的行为。

```
bsdi % telnet svr4 discard
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
hello, world                          正常发送本行
and hi                                在发送本行前断连
Connection closed by foreign host.    9分钟后TCP放弃时输出
```

图21-1表示的是tcpdump的输出结果（已经去掉了bsdi设置的服务类型信息）。

```
1    0.0                bsdi.1029 > svr4.discard: S 1747921409:1747921409(0)
                                                    win 4096 <mss 1024>
2    0.004811 ( 0.0048) svr4.discard > bsdi.1029: S 3416685569:3416685569(0)
                                                    ack 1747921410
                                                    win 4096 <mss 1024>
3    0.006441 ( 0.0016) bsdi.1029 > svr4.discard: . ack 1 win 4096

4    6.102290 ( 6.0958) bsdi.1029 > svr4.discard: P 1:15(14) ack 1 win 4096
5    6.259410 ( 0.1571) svr4.discard > bsdi.1029: . ack 15 win 4096

6   24.480158 (18.2207) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
7   25.493733 ( 1.0136) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
8   28.493795 ( 3.0001) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
9   34.493971 ( 6.0002) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
10  46.484427 (11.9905) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
11  70.485105 (24.0007) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
12 118.486408 (48.0013) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
13 182.488164 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
14 246.489921 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
15 310.491678 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
16 374.493431 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
17 438.495196 (64.0018) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
18 502.486941 (63.9917) bsdi.1029 > svr4.discard: P 15:23(8) ack 1 win 4096
19 566.488478 (64.0015) bsdi.1029 > svr4.discard: R 23:23(0) ack 1 win 4096
```

图21-1 TCP超时和重传的简单例子

第1、2和3行表示正常的TCP连接建立的过程，第4行是“ hello, world ”（12个字符加上回车和换行）的传输过程，第 5行是其确认。接着我们从 svr4拔掉了以太网电缆，第 6行表示“ and hi ”将被发送。第7~18行是这个报文段的12次重传过程，而第19行则是发送方的TCP最终放弃并发送一个复位信号的过程。

现在检查连续重传之间不同的时间差，它们取整后分别为 1、3、6、12、24、48和多个64秒。在本章的后面，我们将看到当第一次发送后所设置的超时时间实际上为 1.5秒（它在首次发送后的1.0136秒而不是精确的1.5秒后，发生的原因我们已在图 18-7中进行了解释），此后该时间在每次重传时增加1倍并直至64秒。

这个倍乘关系被称为“ 指数退避 (exponential backoff) ”。可以将该例子与 6.5节中的TFTP例子比较，在那里每次重传总是在前一次的 5秒后发生。

首次分组传输（第6行，24.480秒）与复位信号传输（第 19行，566.488秒）之间的时间差约为9分钟，该时间在目前的TCP实现中是不可变的。

对于大多数实现而言，这个总时间是不可调整的。Solaris 2.2允许管理者改变这个时间（E.4节中的tcp_ip_abort_interval变量），且其默认值为2分钟，而不是最常用的9分钟。

## 21.3 往返时间测量

TCP超时与重传中最重要的部分就是对一个给定连接的往返时间（ RTT）的测量。由于路由器和网络流量均会变化，因此我们认为这个时间可能经常会发生变化， TCP应该跟踪这些变化并相应地改变其超时时间。

首先TCP必须测量在发送一个带有特别序号的字节和接收到包含该字节的确认之间的RTT。在上一章中，我们曾提到在数据报文段和 ACK之间通常并没有一一对应的关系。在图

20.1中，这意味着发送方可以测量到的一个RTT，是在发送报文段4（第1~1024字节）和接收报文段7（对1~1024字节的ACK）之间的时间，用M表示所测量到的RTT。

最初的TCP规范使TCP使用低通过滤器来更新一个被平滑的RTT估计器（记为*O*）。

$$R \leftarrow \alpha R + (1 - \alpha)M$$

这里的α是一个推荐值为0.9的平滑因子。每次进行新测量的时候，这个被平滑的RTT将得到更新。每个新估计的90%来自前一个估计，而10%则取自新的测量。

该算法在给定这个随RTT的变化而变化的平滑因子的条件下，RFC 793推荐的重传超时时间*RTO*（Retransmission TimeOut）的值应该设置为

$$RTO = R\beta$$

这里的β是一个推荐值为2的时延离散因子。

[Jacobson 1988] 详细分析了在RTT变化范围很大时，使用这个方法无法跟上这种变化，从而引起不必要的重传。正如Jacobson记述的那样，当网络已经处于饱和状态时，不必要的重传会增加网络的负载，对网络而言这就像在火上浇油一样。

除了被平滑的RTT估计器，所需要做的还有跟踪RTT的方差。在往返时间变化起伏很大时，基于均值和方差来计算*RTO*，将比作为均值的常数倍数来计算*RTO*能提供更好的响应。在[Jacobson 1988]中的图5和图6中显示了根据RFC 793计算的某些实际往返时间的*RTO*和下面考虑了往返时间的方差所计算的*RTO*的比较结果。

正如Jacobson所描述的，均值偏差是对标准偏差的一种好的逼近，但却更容易进行计算（计算标准偏差需要一个平方根）。这就引出了下面用于每个RTT测量*M*的公式。

$$Err = M - A$$

$$A \leftarrow A + gErr$$

$$D \leftarrow D + h(|Err| - D)$$

$$RTO = A + 4D$$

这里的*A*是被平滑的RTT（均值的估计器）而*D*则是被平滑的均值偏差。*Err*是刚得到的测量结果与当前的RTT估计器之差。*A*和*D*均被用于计算下一个重传时间（*RTO*）。增量*g*起平均作用，取为1/8（0.125）。偏差的增益是*h*，取值为0.25。当RTT变化时，较大的偏差增益将使*RTO*快速上升。

[Jacobson 1988]指明在计算*RTO*时使用2D，但经过后来更深入的研究，[Jacobson1990c]将该值改为4D，也就是在BSD Net/1的实现中使用的那样。

Jacobson指明了一种使用整数运算来计算这些公式的方法，并被许多实现所采用（这也就是*g*, *h*和倍数4均是2的乘方的一个原因，这样一来计算均可只通过移位操作而不需要乘、除运算来完成）。

将Jacobson与最初的方法比较，我们发现被平滑的均值计算公式是类似的（α是1减去增益*g*），而增益可使用不同的值。而且Jacobson计算*RTO*的公式依赖于被平滑的RTT和被平滑的均值偏差，而最初的方法则使用了被平滑的RTT的一个倍数。

在看完下一节中的例子时，我们将看到这些估计器是如何被初始化的。

### Karn算法

在一个分组重传时会产生这样一个问题：假定一个分组被发送。当超时发生时，*RTO*正

如21.2节中显示的那样进行退避，分组以更长的*RTO*进行重传，然后收到一个确认。那么这个ACK是针对第一个分组的还是针对第二个分组呢？这就是所谓的重传多义性问题。

[Karn and Partridge 1987]规定，当一个超时和重传发生时，在重传数据的确认最后到达之前，不能更新RTT估计器，因为我们并不知道ACK对应哪次传输（也许第一次传输被延迟而并没有被丢弃，也有可能第一次传输的ACK被延迟）。

并且，由于数据被重传，*RTO*已经得到了一个指数退避，我们在下一次传输时使用这个退避后的*RTO*。对一个没有被重传的报文段而言，除非收到了一个确认，否则不要计算新的*RTO*。

## 21.4 往返时间RTT的例子

在本章中，我们将使用以下这些例子来检查TCP的超时和重传、慢启动以及拥塞避免等方方面面的实现细节。

使用sock程序和如下的命令来将32768字节的数据从主机slip发送到主机vangogh.cs.berkeley.edu上的丢弃服务。

```
slip %sock -D -i -n32 vangogh.cs.berkeley.edu discard
```

在扉页前图中，可以看到slip通过两个SLIP链路与140.252.1以太网相连，并从这里通过Internet到达目的地。通过使用两个9600 b/s的SLIP链路，我们期望能够得到一些可测量的时延。

该命令执行32个写1024字节的操作。由于slip和bsdi之间的MTU为296字节，因此这些操作会产生128个报文段，每个报文段包含256字节的用户数据。整个传输过程的时间约为45秒，我们观察到了一个超时和三次重传。

当该传输过程进行时，我们在slip上使用tcpdump来截获所有的发送和接收的报文段，并通过使用-D选项来打开插口排错功能（见A.6节），这样便可以通过运行一个修改后的trpt(8)程序来打印出连接控制块中与RTT、慢启动及拥塞避免等有关的多个变量。

对于给出的跟踪结果，我们不能够完全进行显示，相反，我们将在介绍本章时看到它的各个部分。图21-2显示的是前5秒中的数据和确认的传输过程。与前面tcpdump的输出相比，我们已对其显示稍微进行了修改。虽然我们仅能够在运行tcpdump的主机上测量分组发送和接收的时间，但在本图中我们希望显示出分组正在网络中传输（它们确实存在，因为这个局域网连接与共享式的以太网并不一样）以及接收主机何时可能产生ACK（在本图中去掉了所有的窗口大小通告。主机slip总是通告窗口大小为4096，而vangogh则总是通告窗口大小为8192）。

还需要注意的是在本图中我们已经将报文段按照在主机slip上发送和接收的序号记为1~13和15。这与在这个主机上所收集的tcpdump的输出结果有关。

### 21.4.1 往返时间RTT的测量

在图21-2左边的时间轴上有三个括号，它们表明为进行RTT计算对哪些报文段进行了计时，并不是所有的报文段都被计时。

大多数源于伯克利的TCP实现在任何时候对每个连接仅测量一次RTT值。在发送一个报文段时，如果给定连接的定时器已经被使用，则该报文段不被计时。

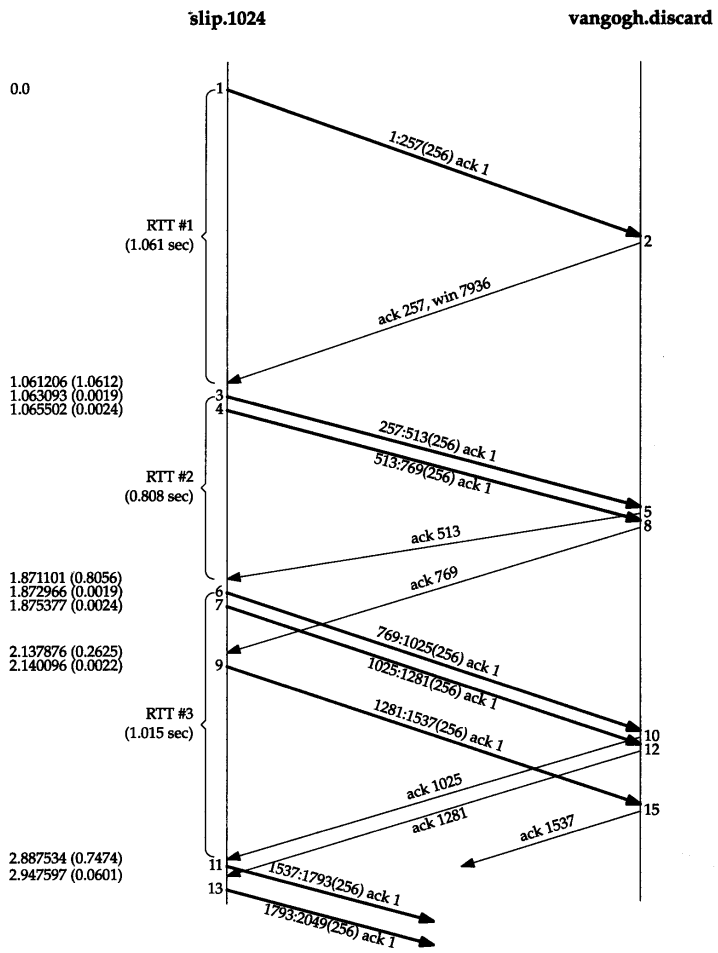


图21-2 分组交换和RTT测量

在每次调用500 ms的TCP的定时器例程时，就增加一个计数器来完成计时。这意味着，如果一个报文段的确认在它发送550 ms后到达，则该报文段的往返时间RTT将是1个滴答（即500 ms）或是2个滴答(即1000 ms)。

对每个连接而言，除了这个滴答计数器，报文段中数据的起始序号也被记录下来。当收到一个包含这个序号的确认后，该定时器就被关闭。如果ACK到达时数据没有被重传，则被平滑的RTT和被平滑的均值偏差将基于这个新测量进行更新。

图21-2中连接上的定时器在发送报文段1时启动，并在确认（报文段2）到达时终止。尽管它的RTT是1.061秒（tcpdump的输出），但插口排错的信息显示该过程经历了3个TCP时钟滴答，即RTT为1500 ms。

下一个被计时的是报文段3。当2.4 ms后传输报文段4时，由于连接的定时器已经被启动，因此该报文段不能被计时。当报文段5到达时，确认了正在被计时的数据。虽然我们从tcpdump的输出结果可以看到其RTT是0.808秒，但它的RTT被计算为1个滴答（500 ms）。

定时器在发送报文段6时再次被启动，并在1.015秒后接收到它的确认（报文段10）时终止。测量到的RTT是2个滴答。报文段7和9不能被计时，因为定时器已经被使用。而且，当收

到报文段8（第769字节的确认）时，由于该报文段不是正在计时的数据的确认，因此什么也没有进行更新。

图21-3显示了本例中通过tcpdump的输出所得到的实际RTT与时钟滴答计数之间的关系。

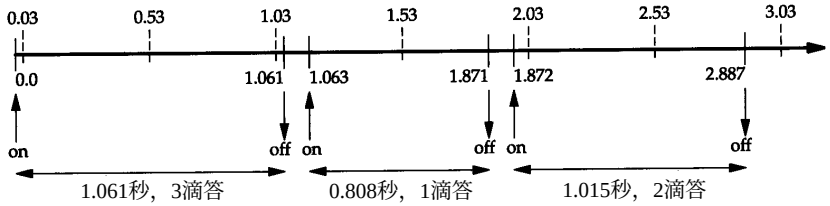


图21-3 RTT测量和时钟滴答

在图的上端表示间隔为500 ms的时钟滴答，图的下端表示tcpdump的输出时间及定时器何时被启动和关闭。在发送报文段 1和接收到报文段2之间经历了3个滴答，时间为1.061秒，因此假定第1个滴答发生在0.03秒处（第1个滴答一定在0~0.061秒之间）。接着该图表示了第2个被测量的RTT为什么被记为1个滴答，而第3个被记为2个滴答。

在这个完整的例子中， 128个报文段被传送，并收集了18个RTT采样。图21-4表示了测量的RTT（取自tcpdump的输出）和TCP为超时所使用的*RTO*（取自插口排错的输出）。在图21-2中，*x*轴从时间0开始，表示的是传输报文段1的时刻，而不是传输第1个SYN的时刻。

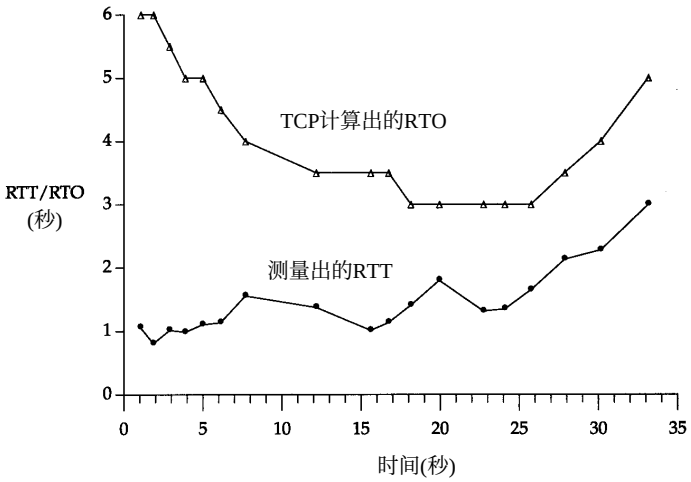


图21-4 测量出的RTT和TCP计算的*RTO*的例子

测量出RTT的前3个数据点对应图21-2所示的3个RTT。在时间10, 14和21处的间隔是由在这些时刻附近发生的重传（将在本章后面给出）引起的。 Karn算法在另一个报文段被发送和确认之前阻止我们更新估计器。同样注意到在这个实现中， TCP计算的*RTO*总是500 ms的倍数。

### 21.4.2 RTT估计器的计算

让我们来看一下RTT估计器（平滑的RTT和平滑的均值偏差）是如何被初始化和更新，以及每个重传超时是怎样计算的。

变量*A*和*D*分别被初始化为0和3秒。初始的重传超时使用下面的公式进行计算

$$RTO = A + 2D = 0 + 2 \times 3 = 6 \text{ s}$$

（因子2*D*只在这个初始化计算中使用。正如前面提到的，以后使用4*D*和*A*相加来计算*RTO*）。这就是传输初始SYN所使用的*RTO*。

结果是这个初始SYN丢失了，然后超时并引起了重传。图21-5给出了tcpdump输出文件中的前4行。

```
1  0.0                    slip.1024 > vangogh.discard: S 35648001:35648001(0)
                                                        win 4096 <mss 256>

2  5.802377 (5.8024)      slip.1024 > vangogh.discard: S 35648001:35648001(0)
                                                        win 4096 <mss 256>

3  6.269395 (0.4670)      vangogh.discard > slip.1024: S 1365512705:1365512705(0)
                                                        ack 35648002
                                                        win 8192 <mss 512>

4  6.270796 (0.0014)      slip.1024 > vangogh.discard: . ack 1 win 4096
```

图21-5 初始SYN的超时和重传

当超时在5.802秒后发生时，计算当前的*RTO*值为

$$RTO = A + 4D = 0 + 4 \times 3 = 12 \text{ s}$$

因此，应用于*RTO*的指数退避取为12。由于这是第1次超时，我们使用倍数2，因此下一个超时时间取值为24秒。再下一个超时时间的倍数为4，得出值为48秒（这些初始*RTO*，对于一个连接上的最初的SYN，取值为6秒，接下来为24秒，正是我们在图4-5中看到的）。

ACK在重传后467ms到达。*A*和*D*的值没有被更新，这是因为Karn算法对重传的处理比较模糊。下一个发送的报文段是第4行的ACK，但它只是一个ACK，所以没有被计时（只有数据报文段才会被计时）。

当发送第1个数据报文段时（图21-2中的报文段1），*RTO*没有改变，这同样是由于Karn算法。当前的24秒一直被使用，直到进行一个RTT测量。这意味着图21-4中时间0的*RTO*并不真的是24，但我们没有画出那个点。

当第1个数据报文段的ACK（图21-2中的报文段2）到达时，经历了3个时钟滴答，估计器被初始化为

$$A = M + 0.5 = 1.5 + 0.5 = 2$$

$$D = A/2 = 1$$

（因为经历3个时钟滴答，因此，*M*取值为1.5）。在前面，*A*和*D*初始化为0，*RTO*的初始计算值为3。这是使用第1个RTT的测量结果M对估计器进行首次计算的初始值。计算的*RTO*值为

$$RTO = A + 4D = 2 + 4 \times 1 = 6 \text{ s}$$

当第2个数据报文段的ACK（图21-2中的报文段5）到达时，经历了1个时钟滴答（0.5秒），估计器按如下更新：

$$Err = M - A = 0.5 - 2 = -1.5$$

$$A = A + gErr = 2 - 0.125 \times 1.5 = 1.8125$$

$$D = D + h(|Err| - D) = 1 + 0.25 \times (1.5 - 1) = 1.125$$

$$RTO = A + 4D = 1.8125 + 4 \times 1.125 = 6.3125$$

*Err*、*A*和*D*的定点表示与实际使用的定点计算（在简化浮点计算中表示过）有一些微小的差别。这些不同使*RTO*取值为6秒（而非6.3125秒），正如我们在图21-4中的时间1.871处所画的那样。

### 21.4.3 慢启动

我们在第20.6节介绍了慢启动算法，在图21-2中可再次看到它的工作过程。

连接上最初只允许传输一个报文段，然后在发送下一个报文段之前必须等待接收它的确认。当报文段2被接收后，就可以再发送两个报文段。

## 21.5 拥塞举例

现在观察一下数据报文段的传输过程。图 21-6显示了报文段中数据的起始序号与该报文段发送时间的对比图。它提供了一种较好的数据传输的可视化方法。通常代表数据的点将向上和向右移动，这些点的斜率就表示传输速率。当这些点向下和向右移动则表示发生了重传。

在21.4节开始时，我们曾提到整个传输的时间约为 45秒，但在本图中只显示了 35秒钟。这35秒只是数据报文段发送的时间。因为第 1个SYN看来是丢失了并被重传（见图 21-5），因此第1个数据报文段是在第1个SYN发送6.3秒后才发送的。而且，在发送最后一个数据报文段和FIN（图21-6中的34.1秒）之后，在接收方的FIN到达之前，又花费了另外的4.0秒接收来自接收方的最后14个ACK。

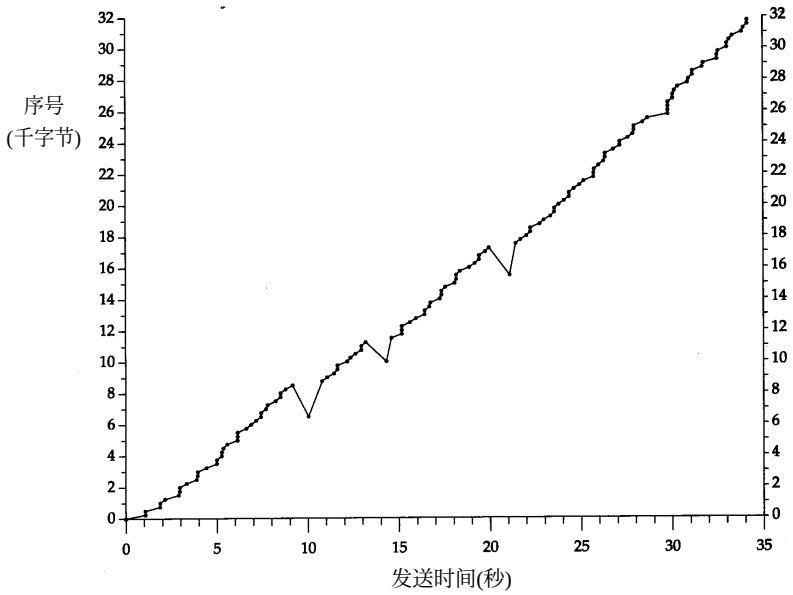


图21-6 从slip 发送32768个字节的数据到vangogh

可以立即看到图21-6中发生在时刻10，14和21附近的3个重传。我们还可以看到在这 3个点中只进行了一次报文段的重传，因为只有一个点下垂低于向上的斜率。

仔细检查一下这几个下垂点中的第 1个点（在10秒标记处的附近）。整理tcpdump的输出结果可以得到图21-7。

在这个图中，除了下面将要讨论的报文段 72，已经去掉了其他所有的窗口通告。主机slip总是通告窗口大小为4096，而主机vangogh则通告窗口为8192。该图中报文段的编号可以看作是图21-2的延续，在那里报文段的编号从 1开始。与图21-2一样，报文段根据在 slip上发送和接收的顺序进行编号， tcpdump在主机slip上运行。我们还去掉了一些与讨论无

关的段（第44, 47和49以及所有来自vangogh的ACK）。

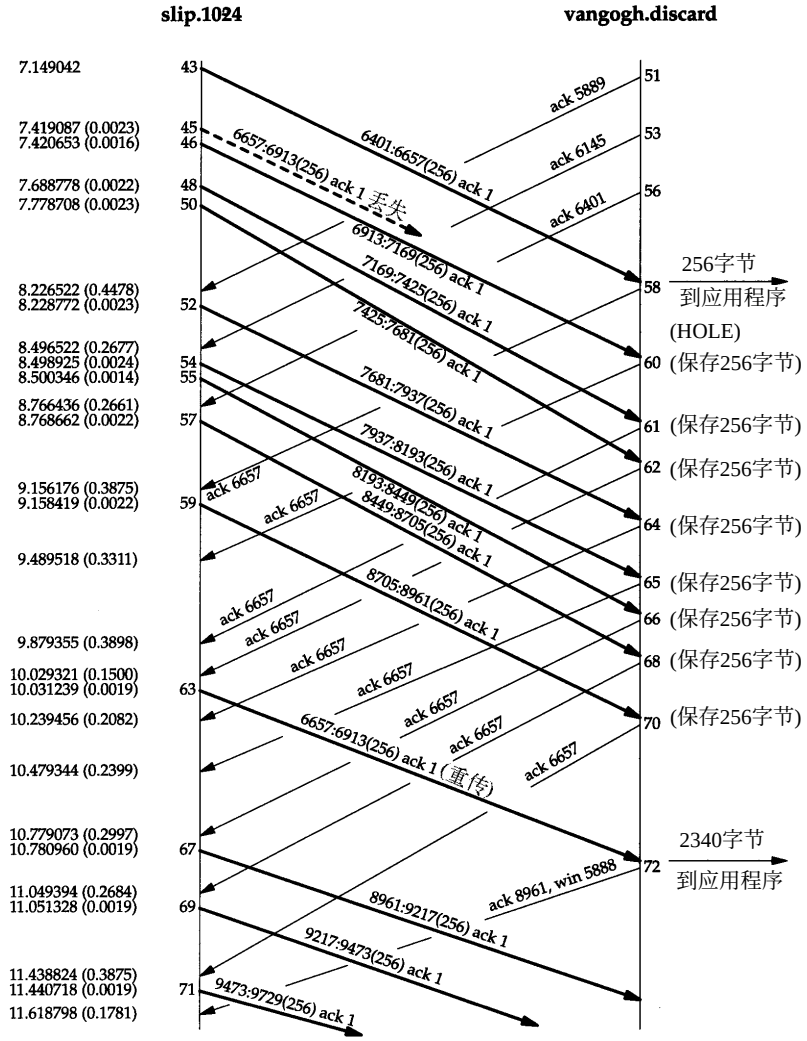


图21-7 10秒标记处附近重传的分组交换

看来报文段45丢失或损坏了，这一点无法从该输出上进行辨认。能够在主机slip上看到的是对第6657字节（报文段58）以前数据的确认（不包括字节6657在内）。紧接着的是带有相同序号的8个ACK。正是接收到报文段62，也就是第3个重复ACK，才引起自序号6657开始的数据报文段（报文段63）进行重传。的确，源于伯克利的TCP实现对收到的重复ACK进行计数，当收到第3个时，就假定一个报文段已经丢失并重传自那个序号起的一个报文段。这就是Jacobson的快速重传算法，该算法通常与他的快速恢复算法一起配合使用。我们在第21.7节中介绍这两个算法。

注意到在重传后（报文段63），发送方继续正常的数据传输（报文段67、69和71）。TCP不需要等待对方确认重传。

现在检查一下在接收端发生了什么。当按序收到正常数据（报文段43）后，接收TCP将255个字节的数据交给用户进程。但下一个收到的报文段（报文段46）是失序的：数据的开始

序号（6913）并不是下一个期望的序号（6657）。TCP保存256字节的数据，并返回一个已成功接收数据的最大序号加1（6657）的ACK。被vangogh接收到的后面7个报文段（48, 50, 52, 54, 55, 57和59）也是失序的，接收方TCP保存这些数据并产生重复ACK。

目前TCP尚无办法告诉对方缺少一个报文段，也无法确认失序数据。此时主机vangogh所能够做的就是继续发送确认序号为6657的ACK。

当缺少的报文段（报文段63）到达时，接收方TCP在其缓存中保存第6657~8960字节的数据，并将这2304字节的数据交给用户进程。所有这些数据在报文段72中进行确认。请注意此时该ACK通告窗口大小为5888（8192 - 2304），这是因为用户进程没有机会读取这些已准备好的2304字节的数据。

如果仔细检查图21-6中tcpdump的输出中第14和21秒附近的下垂点，我们会看到它们也是由于收到了3个重复ACK引起的，这表明一个分组已经丢失。在这些例子中只有一个分组被重传。

在介绍完拥塞避免算法后，将在第21.8节中继续讨论这个例子。

## 21.6 拥塞避免算法

在第20.6节介绍的慢启动算法是在一个连接上发起数据流的方法，但有时我们会达到中间路由器的极限，此时分组将被丢弃。拥塞避免算法是一种处理丢失分组的方法。该方法的具体描述见[Jacobson 1988]。

该算法假定由于分组受到损坏引起的丢失是非常少的（远小于1%），因此分组丢失就意味着在源主机和目的主机之间的某处网络上发生了拥塞。有两种分组丢失的指示：发生超时和接收到重复的确认（我们在21.5节看到这种现象。如果使用超时作为拥塞指示，则需要使用一个好的RTT算法，正如在21.3节中描述的那样）。

拥塞避免算法和慢启动算法是两个目的不同、独立的算法。但是当拥塞发生时，我们希望降低分组进入网络的传输速率，于是可以调用慢启动来作到这一点。在实际中这两个算法通常在一起实现。

拥塞避免算法和慢启动算法需要对每个连接维持两个变量：一个拥塞窗口*cwnd*和一个慢启动门限*ssthresh*。这样得到的算法的工作过程如下：

1) 对一个给定的连接，初始化*cwnd*为1个报文段，*ssthresh*为65535个字节。

2) TCP输出例程的输出不能超过*cwnd*和接收方通告窗口的大小。拥塞避免是发送方使用的流量控制，而通告窗口则是接收方进行的流量控制。前者是发送方感受到的网络拥塞的估计，而后者则与接收方在该连接上的可用缓存大小有关。

3) 当拥塞发生时（超时或收到重复确认），*ssthresh*被设置为当前窗口大小的一半（*cwnd*和接收方通告窗口大小的最小值，但最少为2个报文段）。此外，如果是超时引起了拥塞，则*cwnd*被设置为1个报文段（这就是慢启动）。

4) 当新的数据被对方确认时，就增加*cwnd*，但增加的方法依赖于我们是否正在进行慢启动或拥塞避免。如果*cwnd*小于或等于*ssthresh*，则正在进行慢启动，否则正在进行拥塞避免。慢启动一直持续到我们回到当拥塞发生时所处位置的半时候才停止（因为我们记录了在步骤2中给我们制造麻烦的窗口大小的一半），然后转为执行拥塞避免。

慢启动算法初始设置*cwnd*为1个报文段，此后每收到一个确认就加1。正如20.6节描述的

那样，这会使窗口按指数方式增长：发送1个报文段，然后是2个，接着是4个……。

拥塞避免算法要求每次收到一个确认时将*cwnd*增加1/*cwnd*。与慢启动的指数增加比起来，这是一种加性增长(additive increase)。我们希望在一个往返时间内最多为*cwnd*增加1个报文段（不管在这个RTT中收到了多少个ACK），然而慢启动将根据这个往返时间中所收到的确认的个数增加*cwnd*。

所有的4.3BSD版本和4.4BSD都在拥塞避免中将增加值不正确地设置为1个报文段的一小部分（即一个报文段的大小除以8），这是错误的，并在以后的版本中不再使用[Floyd 1994]。但是，为了和（不正确的）实现的结果对应，我们在将来的计算中给出了这个细节。

在[Leffler et al. 1989]中介绍的4.3BSD Tahoe版本仅在对方处于一个不同的网络上时才进行慢启动。而4.3BSD Reno版本改变了这种做法，因此，慢启动总是被执行。

图21-8是慢启动和拥塞避免的一个可视化描述。我们以段为单位来显示 *cwnd*和*ssthresh*，但它们实际上都是以字节为单位进行维护的。

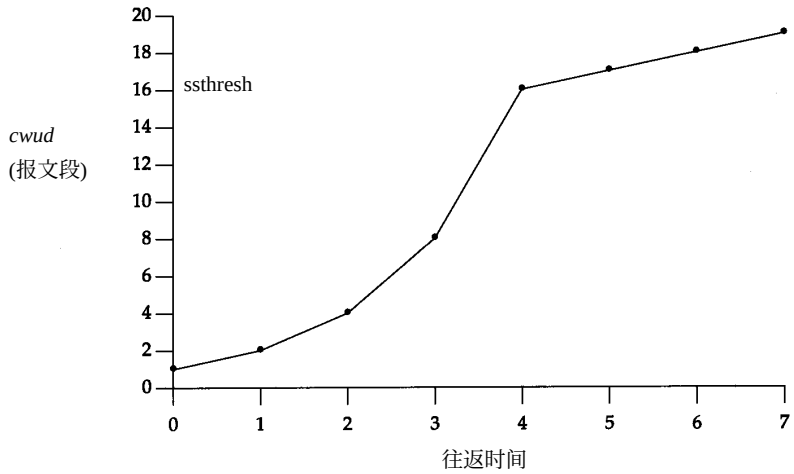


图21-8 慢启动和拥塞避免的可视化描述

在该图中，假定当*cwnd*为32个报文段时就会发生拥塞。于是设置 *ssthresh*为16个报文段，而*cwnd*为1个报文段。在时刻 0发送了一个报文段，并假定在时刻 1接收到它的ACK，此时*cwnd*增加为2。接着发送了2个报文段，并假定在时刻2接收到它们的ACK，于是*cwnd*增加为4（对每个ACK增加1次）。这种指数增加算法一直进行到在时刻 3和4之间收到8个ACK后*cwnd*等于*ssthresh*时才停止，从该时刻起， *cwnd*以线性方式增加，在每个往返时间内最多增加 1个报文段。

正如我们在这个图中看到的那样，术语“慢启动”并不完全正确。它只是采用了比引起拥塞更慢些的分组传输速率，但在慢启动期间进入网络的分组数增加的速率仍然是在增加的。只有在达到*ssthresh*拥塞避免算法起作用时，这种增加的速率才会慢下来。

## 21.7 快速重传与快速恢复算法

拥塞避免算法的修改建议1990年提出 [Jacobson 1990b]。在我们的例子（见21.5节）中已

经可以看到这些实施中的修改。

在介绍修改之前，我们认识到在收到一个失序的报文段时， TCP立即需要产生一个ACK（一个重复的ACK）。这个重复的ACK不应该被迟延。该重复的ACK的目的在于让对方知道收到一个失序的报文段，并告诉对方自己希望收到的序号。

由于我们不知道一个重复的ACK是由一个丢失的报文段引起的，还是由于仅仅出现了几个报文段的重新排序，因此我们等待少量重复的ACK到来。假如这只是一些报文段的重新排序，则在重新排序的报文段被处理并产生一个新的ACK之前，只可能产生1~2个重复的ACK。如果一连串收到3个或3个以上的重复ACK，就非常可能是一个报文段丢失了（我们在21.5节中见到过这种现象）。于是我们就重传丢失的数据报文段，而无需等待超时定时器溢出。这就是快速重传算法。接下来执行的不是慢启动算法而是拥塞避免算法。这就是快速恢复算法。

在图21-7中可以看到在收到3个重复的ACK之后没有执行慢启动。相反，发送方进行重传，接着在收到重传的ACK以前，发送了3个新的数据的报文段（报文段67, 69和71）。

在这种情况下没有执行慢启动的原因是由于收到重复的ACK不仅仅告诉我们一个分组丢失了。由于接收方只有在收到另一个报文段时才会产生重复的ACK，而该报文段已经离开了网络并进入了接收方的缓存。也就是说，在收发两端之间仍然有流动的数据，而我们不想执行慢启动来突然减少数据流。

这个算法通常按如下过程进行实现：

1) 当收到第3个重复的ACK时，将*ssthresh*设置为当前拥塞窗口*cwnd*的一半。重传丢失的报文段。设置*cwnd*为*ssthresh*加上3倍的报文段大小。

2) 每次收到另一个重复的ACK时，*cwnd*增加1个报文段大小并发送1个分组（如果新的*cwnd*允许发送）。

3) 当下一个确认新数据的ACK到达时，设置*cwnd*为*ssthresh*（在第1步中设置的值）。这个ACK应该是在进行重传后的一个往返时间内对步骤1中重传的确认。另外，这个ACK也应该是对丢失的分组和收到的第1个重复的ACK之间的所有中间报文段的确认。这一步采用的是拥塞避免，因为当分组丢失时我们将当前的速率减半。

在下一节中我们将看到变量*cwnd*和*ssthresh*的计算过程。

快速重传算法最早出现在4.3BSD Tahoe版本中，但它随后错误地使用了慢启动。快速恢复算法出现在4.3BSD Reno版本中。

## 21.8 拥塞举例(续)

通过使用`tcmdump`和插口排错选项（在第21.4节进行了介绍）来观察一个连接，就会在发送每一个报文段时看到*cwnd*和*ssthresh*的值。如果MSS为256字节，则*cwnd*和*ssthresh*的初始值分别为256和65535字节。每当收到一个ACK时，我们可以看到*cwnd*增加了一个MSS，取值分别为512, 768, 1024, 1280等。假定不会发生拥塞，则最终拥塞窗口将超过接收方的通告窗口，意味着通告窗口将对数据流进行限制。

一个更有趣的例子是观察在拥塞发生时的情况。使用与21.4节同样的例子。当这个例子运行时发生了4次拥塞。为建立连接而发送的初始SYN有一个因超时而引起的重传（见图21-5），接着在数据传输过程中有3个分组丢失（见图21-6）。

图21-9显示了当初始SYN重传并接着发送了前7个数据报文段时变量*cwnd*和*ssthresh*的值（在图21-2中显示了最初的数据报文段及其ACK之间的交换过程）。使用`tcpdump`的记号来表示数据字节：1:257(256)表示第1~256字节。

当SYN的超时发生时，*ssthresh*被置为其最小取值（512字节，在本例中表示2个报文段）。为进入慢启动阶段，*cwnd*被置为1个报文段（256字节，与当前值一致）。

当收到SYN和ACK时，没有对这两个变量做任何修改，因为新的数据还没有被确认。

当ACK 257到达时，因为*cwnd*小于等于*ssthresh*，因此仍然处于慢启动阶段，于是将*cwnd*增加256字节。当收到ACK 513时，进行同样的处理。

当ACK 769到达时，我们不再处于慢启动状态，而是进入了拥塞避免状态。新的*cwnd*值按以下方法计算：

$$cwnd \leftarrow cwnd + \frac{segsize \times segsize}{cwnd} + \frac{segsize}{8}$$

考虑到*cwnd*实际上以字节而非以报文段来维护，因此这就是我们前面提到的增加1/*cwnd*。在这个例子中我们计算

$$cwnd \leftarrow 768 + \frac{256 \times 256}{768} + \frac{256}{8}$$

为885字节（使用整数算法）。当下一个ACK 1025到达时，我们计算

$$cwnd \leftarrow 885 + \frac{256 \times 256}{885} + \frac{256}{8}$$

为991字节（在这些表达式中包括了不正确的256/8项来匹配实现计算的数值，正如我们在前面标注的那样）。

| 报文段号 (图21-2) | 行为 | | | 变量 | |
|---|---|---|---|---|---|
| | 发送 | 接收 | 注释 | *cwnd* | *ssthresh* |
| | SYN | | 初始化 | 256 | 65535 |
| | SYN | | 超时重传 | 256 | 512 |
| | ACK | SYN, ACK | | | |
| 1 | 1:257(256) | | | | |
| 2 | | ACK 257 | 慢启动 | 512 | 512 |
| 3 | 257:513(256) | | | | |
| 4 | 513:769(256) | | | | |
| 5 | | ACK 513 | 慢启动 | 768 | 512 |
| 6 | 769:1025(256) | | | | |
| 7 | 1025:1281(256) | | | | |
| 8 | | ACK 769 | cong. avoid | 885 | 512 |
| 9 | 1281:1537(256) | | | | |
| 10 | | ACK 1025 | cong. avoid | 991 | 512 |
| 11 | 1537:1793(256) | | | | |
| 12 | | ACK 1281 | cong. avoid | 1089 | 512 |

图21-9 拥塞避免的例子

这个*cwnd*持续增加一直到在图21-6所示的发生在10秒左右的第1次重传。图21-10是使用与图21-6相同数据得到的图表，并给出了*cwnd*增加的数值。

本图中*cwnd*的前6个值就是我们为图21-9所计算的数值。在这个图中，要想直观分辨出在慢启动过程中的指数增加和在拥塞避免过程中的线性增加之间的区别是不可能的，因为慢启动的过程太快。

我们需要解释在重传的3个点上所发生的情况。回想起每个重传都是因为收到 3个重复的ACK，表明1个分组丢失了。这就是21.7节的快速重传算法。*ssthresh*立即设置为当重传发生时正在起作用的窗口大小的一半，但是在接收到重复 ACK的过程中*cwnd*允许保持增加，这是因为每个重复的ACK表示1个报文段已离开了网络（接收TCP已缓存了这个报文段，等待所缺数据的到达）。这就是快速恢复算法。

与图20-9类似，图21-10表示了*cwnd*和*ssthresh*的数值。第一列上的报文段编号与图21-7对应。

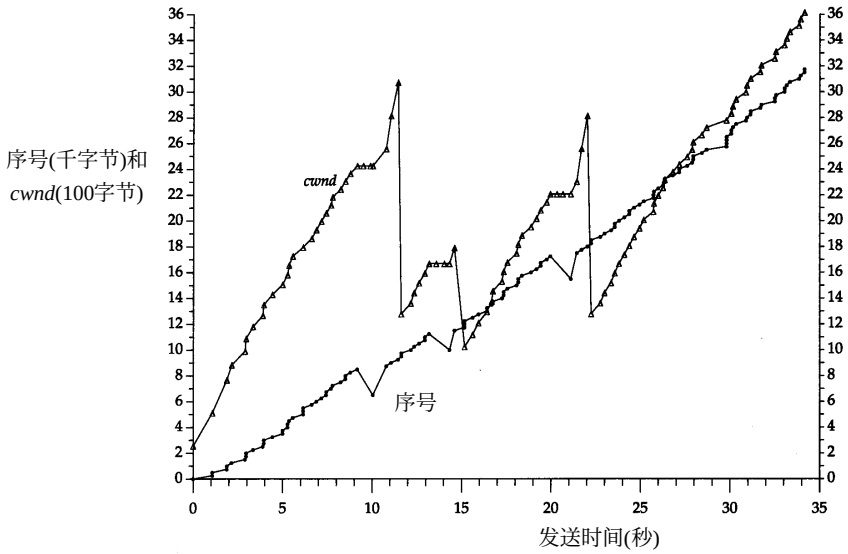


图21-10 当数据被发送时的发送序号和*cwnd*的取值

| 报文段号 (图21-7) | 行 为 | | | 变 量 | |
|---|---|---|---|---|---|
| | 发送 | 接收 | 注释 | *cwnd* | *ssthresh* |
| 58 | | ACK 6657 | 新数据的确认 | 2426 | 512 |
| 59 | 8705:8961(256) | | | | |
| 60 | | ACK 6657 | 重复 ACK #1 | 2426 | 512 |
| 61 | | ACK 6657 | 重复 ACK #2 | 2426 | 512 |
| 62 | | ACK 6657 | 重复 ACK #3 | 1792 | 1024 |
| 63 | 6657:6913(256) | | 重传 | | |
| 64 | | ACK 6657 | 重复 ACK #4 | 2048 | 1024 |
| 65 | | ACK 6657 | 重复 ACK #5 | 2304 | 1024 |
| 66 | | ACK 6657 | 重复 ACK #6 | 2560 | 1024 |
| 67 | 8961:9217(256) | | | | |
| 68 | | ACK 6657 | 重复 ACK #7 | 2816 | 1024 |
| 69 | 9217:9473(256) | | | | |
| 70 | | ACK 6657 | 重复 ACK #8 | 3072 | 1024 |
| 71 | 9473:9729(256) | | | | |
| 72 | | ACK 8961 | 新数据的确认 | 1280 | 1024 |

图21-11 拥塞避免的例子

*cwnd*的值一直持续增加，从图21-9中对应于报文段12的最终取值（1089）到图21-11中对应于报文段58的第一个取值（2426），而*ssthresh*的值则保持不变（512），这是因为在此过程中没有出现过重传。

当最初的2个重复的ACK（报文段60和61）到达时它们被计数，而 *cwnd*保持不变（也就是图21-10中处理重传之前的平坦的一段）。然而，当第3个重复的ACK到达时，*ssthresh*被置

为*cwnd*的一半（四舍五入到报文段大小的下一个倍数），而*cwnd*被置为*ssthresh*加上所收到的重复的ACK数乘以报文段大小（也即1024加上3倍的256），然后发送重传数据。

又有5个重复的ACK到达（报文段64~66, 68和70），每次*cwnd*增加1个报文段长度。最后一个新的ACK（报文段72段）到达时，*cwnd*被置为*ssthresh*（1024）并进入正常的拥塞避免过程。由于*cwnd*小于等于*ssthresh*（现在相等），因此报文段的大小增加到 *cwnd*，取值为1280。当下一个新的ACK到达（没有在图21-11中表示出来）时，*cwnd*大于*ssthresh*，取值为1363。

在快速重传和快速恢复阶段，我们收到报文段 66、68和70中的重复的ACK后才发送新的数据，而不是在接收到报文段 64和65中重复的ACK之后就发送。这是*cwnd*的取值与未被确认的数据大小比较的结果。当报文段 65到达时，*cwnd*为2048，但未被确认的数据有 2304字节（9个报文段：46, 48, 50, 52, 54, 55, 57, 59和63），因此不能发送任何数据。当报文段65到达后，*cwnd*被置为2304，此时我们仍不能进行发送。但是当报文段 66到达时，*cwnd*为2560，所以我们可以发送1个新的数据报文段。类似地，当报文段 68到达时，*cwnd*等于2816，该数值大于未被确认的2560字节的数据大小，因此我们可以发送另 1个新的数据报文段。报文段 70到达时也进行了类似的处理。

在图21-10中的时刻14.3发生下一个重传，也是因为收到了 3个重复的ACK。因此当另一个ACK到达时，可以看到*cwnd*以同样的方式增长，之后降低到1024。

图21-10中的时刻21.1也是因为收到了重复的 ACK而引起了重传。在重传后收到了 3个重复的ACK，因此观察到*cwnd*增加3个，之后降低到1280。在传输的后面部分，*cwnd*以线性方式增加到最终值3615。

## 21.9 按每条路由进行度量

较新的TCP实现在路由表项中维持许多我们在本章已经介绍过的指标。当一个 TCP连接关闭时，如果已经发送了足够多的数据来获得有意义统计资料，且目的结点的路由表项不是一个默认的表项，那么下列信息就保存在路由表项中以备下次使用：被平滑的 RTT、被平滑的均值偏差以及慢启动门限。所谓“ 足够多的数据 ”是指 16个窗口的数据，这样就可得到 16个RTT采样，从而使被平滑的RTT过滤器能够集中在正确结果的5%以内。

而且，管理员可以使用 route(8)命令来设置给定路由的度量：前一段中给出的三个指标以及MT、输出的带宽时延乘积（见第 20.7节）和输入的带宽时延乘积。

当建立一个新的连接时，不论是主动还是被动，如果该连接将要使用的路由表项已经有这些度量的值，则用这些度量来对相应的变量进行初始化。

## 21.10 ICMP的差错

让我们来看一下TCP是怎样处理一个给定的连接返回的 ICMP的差错。TCP能够遇到的最常见的ICMP差错就是源站抑制、主机不可达和网络不可达。

当前基于伯克利的实现对这些错误的处理是：

- 一个接收到的源站抑制引起拥塞窗口 *cwnd*被置为1个报文段大小来发起慢启动，但是慢启动门限*ssthresh*没有变化，所以窗口将打开直至它或者开放了所有的通路（受窗口大小和往返时间的限制）或者发生了拥塞。
- 一个接收到的主机不可达或网络不可达实际上都被忽略，因为这两个差错都被认为是

短暂现象。这有可能是由于中间路由器被关闭而导致选路协议要花费数分钟才能稳定到另一个替换路由。在这个过程中就可能发生这两个 ICMP差错中的一个，但是连接并不必被关闭。相反， TCP试图发送引起该差错的数据，尽管最终有可能会超时（回想图21-1中TCP在9分钟内没有放弃的情况）。当前基于伯克利的实现记录发生的ICMP差错，如果连接超时， ICMP差错被转换为一个更合适的的差错码而不是“连接超时”。

早期的BSD实现在任何时候收到一个主机不可达或网络不可达的ICMP差错时会不正确的放弃连接。

### 一个例子

可以通过在连接中拨号SLIP链路的断开来观察一个ICMP主机不可达的差错是如何被处理的。建立一个从主机slip到主机aix的连接（从扉页前的图中可以看到这个连接经过了我们的拨号SLIP链路）。在建立连接并发送一些数据之后，在路由器 sun和netb之间的SLIP链路被断开，这引起sun上的默认路由表项（见9.2节）被移去。我们希望sun对目的为140.252.1以太网的IP数据报响应ICMP主机不可达。希望观察TCP如何处理这些ICMP差错。

下面是主机slip的交互会话：

```
slip % sock aix echo                    运行sock程序
test line                               键入本行
test line                               和它的回显
                                        此时挂断SLIP链路
another line                            然后键入本行并观察其行为
                                        SLIP链路此时重新建立，
another line                            该行及其回显被交换
line number 3
line number 3
the last line
                                        此时挂断SLIP链路，且没有重新建立
read error: No route to host            TCP最终放弃
```

图21-12显示了在路由器bsdi上截获的tcpdump的相应输出（去掉了连接建立和所有的窗口通告）。我们连接到在主机 aix上的回显服务器并键入“ test line ”（第1行），它被回显（第2行）且回显被确认（第3行），接着我们断开了SLIP链路。

我们键入“ another line ”（第3行之后）并希望看到TCP超时和重传报文。的确，这一行在收到应答前被发送了6次。第4~13行显示了第1次传输和接着的4次重传，每个都产生了一个来自路由器sun的ICMP主机不可达。这正是我们所希望的：从 slip来的IP数据报发往路由器bsdi（这是一个指向sun的默认路由器），并到达检测到链路中断的sun。

在发生这些重传时， SLIP链路又被连通，在第14行的重传被交付。第15行是来自aix的回显，而第16行是对这个回显的确认。

这表明TCP忽略ICMP主机不可达的差错并坚持重传。我们也可以观察到所预期的在每一次重传超时中的指数退避：第1次约为2.5秒，接着乘2（约5秒），乘4（约10秒），乘8（约20秒），乘14（约40秒）。

接着我们键入输入的第3行（“ line number 3 ”）并看到它在第17行被发送，在第18行回显，并在第19行对回显进行确认。

```
1     0.0                  slip.1035 > aix.echo: P 1:11(10) ack 1
2     0.212271 (  0.2123)  aix.echo > slip.1035: P 1:11(10) ack 11
3     0.310685 (  0.0984)  slip.1035 > aix.echo: . ack 11
```

SLIP链路此时被挂断

```
4   174.758100 (174.4474)  slip.1035 > aix.echo: P 11:24(13) ack 11
5   174.759017 (  0.0009)  sun > slip: icmp: host aix unreachable

6   177.150439 (  2.3914)  slip.1035 > aix.echo: P 11:24(13) ack 11
7   177.151271 (  0.0008)  sun > slip: icmp: host aix unreachable

8   182.150200 (  4.9989)  slip.1035 > aix.echo: P 11:24(13) ack 11
9   182.151189 (  0.0010)  sun > slip: icmp: host aix unreachable

10  192.149671 (  9.9985)  slip.1035 > aix.echo: P 11:24(13) ack 11
11  192.150608 (  0.0009)  sun > slip: icmp: host aix unreachable

12  212.148783 ( 19.9982)  slip.1035 > aix.echo: P 11:24(13) ack 11
13  212.149786 (  0.0010)  sun > slip: icmp: host aix unreachable
```

SLIP链路此时被建立

```
14  252.146774 ( 39.9970)  slip.1035 > aix.echo: P 11:24(13) ack 11
15  252.439257 (  0.2925)  aix.echo > slip.1035: P 11:24(13) ack 24
16  252.505331 (  0.0661)  slip.1035 > aix.echo: . ack 24

17  261.977246 (  9.4719)  slip.1035 > aix.echo: P 24:38(14) ack 24
18  262.158758 (  0.1815)  aix.echo > slip.1035: P 24:38(14) ack 38
19  262.305086 (  0.1463)  slip.1035 > aix.echo: . ack 38
```

SLIP链路此时被挂断

```
20  458.155330 (195.8502)  slip.1035 > aix.echo: P 38:52(14) ack 38
21  458.156163 (  0.0008)  sun > slip: icmp: host aix unreachable

22  461.136904 (  2.9807)  slip.1035 > aix.echo: P 38:52(14) ack 38
23  461.137826 (  0.0009)  sun > slip: icmp: host aix unreachable

24  467.136461 (  5.9986)  slip.1035 > aix.echo: P 38:52(14) ack 38
25  467.137385 (  0.0009)  sun > slip: icmp: host aix unreachable

26  479.135811 ( 11.9984)  slip.1035 > aix.echo: P 38:52(14) ack 38
27  479.136647 (  0.0008)  sun > slip: icmp: host aix unreachable

28  503.134816 ( 23.9982)  slip.1035 > aix.echo: P 38:52(14) ack 38
29  503.135740 (  0.0009)  sun > slip: icmp: host aix unreachable
```

在这里14行输出结果被删除

```
44  1000.219573 ( 64.0959)  slip.1035 > aix.echo: P 38:52(14) ack 38
45  1000.220503 (  0.0009)  sun > slip: icmp: host aix unreachable

46  1064.201281 ( 63.9808)  slip.1035 > aix.echo: R 52:52(0) ack 38
47  1064.202182 (  0.0009)  sun > slip: icmp: host aix unreachable
```

图21-12 TCP对接收到的ICMP主机不可达差错的处理

现在我们希望观察在接收到ICMP主机不可达后，TCP重传并放弃的情况。于是再次断开SLIP链路，之后键入“the last line”，并观察到在TCP放弃之前该行被发送了13次（我们已经从结果中删除了第30~43行，它们是额外的重传）。

然而，我们所观察到的现象是sock程序在最终放弃时打印出来的差错信息：“没有到达主机的路由”。这与Unix的ICMP主机不可达的差错类似（图6-12）。这表明TCP保存了它在连接上收到的ICMP差错，并在最终放弃时打印出该差错，而不是“连接超时”。

最后，注意到第22~46行与第6~14行不同的重传间隔。看起来我们键入的第3行在第17~19行被发送和确认时（无任何重传），TCP更新了它的估计器。最初的重传超时时间现在是3秒，后续取值为6, 12, 24, 48，直至上限64。

## 21.11 重新分组

当TCP超时并重传时，它不一定要重传同样的报文段。相反， TCP允许进行重新分组而发送一个较大的报文段，这将有助于提高性能（当然，这个较大的报文段不能够超过接收方声明的MSS）。在协议中这是允许的，因为 TCP是使用字节序号而不是报文段序号来进行识别它所要发送的数据和进行确认。

在实际中，可以很容易地看到这一点。我们使用 sock程序连接到丢弃服务器并键入一行。接着拔掉以太网电缆并再键入一行。当这一行被重传时，键入第 3行。我们预期下一个重传包含第2次和第3次键入的数据。

```
bsdi %sock svr4 discard
hello there                          第一行发送成功
                                     接着我们断开以太网电缆
line number 2                        本行被重传
and 3                                在第2行发送成功之前键入本行
                                     接着重新连接以太网电缆
```

图21-13显示了tcpdump的输出（去掉了连接建立、连接终止以及所有的窗口通告）。

```
1   0.0                 bsdi.1032 > svr4.discard: P 1:13(12) ack 1
2   0.140489 ( 0.1405)  svr4.discard > bsdi.1032: . ack 13
                                  此时断开以太网电缆
3  26.407696 (26.2672)  bsdi.1032 > svr4.discard: P 13:27(14) ack 1
4  27.639390 ( 1.2317)  bsdi.1032 > svr4.discard: P 13:27(14) ack 1
5  30.639453 ( 3.0001)  bsdi.1032 > svr4.discard: P 13:27(14) ack 1
                                  此时键入第3行
6  36.639653 ( 6.0002)  bsdi.1032 > svr4.discard: P 13:33(20) ack 1
7  48.640131 (12.0005)  bsdi.1032 > svr4.discard: P 13:33(20) ack 1
                                  此时重新连接以太网电缆
8  72.640768 (24.0006)  bsdi.1032 > svr4.discard: P 13:33(20) ack 1
9  72.719091 ( 0.0783)  svr4.discard > bsdi.1032: . ack 33
```

图21-13 TCP对数据的重新分组

第1行和第2行显示了头一行（“ hello there ”）被发送及其ACK。接着我们拔掉以太网电缆并键入“ line number 2 ”（14字节，包括换行）。这些数据在第3行被发送，并在第4和第5行被重传。

在第6行重传前，我们键入“ and 3 ”（6个字节，包括换行），并观察到这个重传包括 20个字节：键入的两行。当ACK在第9行到达时，它确认了这20字节的数据。

## 21.12 小结

本章提供了对TCP超时和重传机制的详细研究。使用的第 1个例子是一个丢失的建立连接的SYN，并观察了在随后的重传和超时中怎样使用指数退避方式。

TCP计算往返时间并使用这些测量结果来维护一个被平滑的 RTT估计器和被平滑的均值偏差估计器。这两个估计器用来计算下一个重传时间。许多实现对每个窗口仅测量一次 RTT。Karn算法在分组丢失时可以不测量RTT就能解决重传的二义性问题。

详细例子包括3个丢失的分组，使我们看到TCP的许多实际算法：慢启动、拥塞避免、快速重传和快速恢复。我们也能够使用拥塞窗口和慢启动门限来手工计算 TCP RTT估计器，并将这些值与跟踪输出的实际数据进行比较。

以多种ICMP差错对TCP连接的影响以及TCP怎样允许对数据进行重新分组来结束本章。我们观察到“软”的ICMP差错没有引起TCP连接终止，但这些差错被保存以便在连接非正常中止时能够报告这些软差错。

## 习题

21.1 在图21-5中第1个超时时间计算为6秒而第2个为12秒。如果初始SYN的确认在12秒超时溢出时还没有到达，则下一次超时在什么时候发生？

21.2 在图21-5后面的讨论中，我们提到计算的超时间隔分别为图 4-5中表示的6、24和48秒。但是如果观察一个从SVR4系统到一个不存在的主机的连接，则超时间隔分别为 6, 12, 24和48秒。请问发生了什么情况？

21.3 按下面的描述比较TCP滑动窗口协议与TFTP的停止等待协议的性能。在本章中，我们在35秒（图21-6）内传输32768字节的数据，其中链路的平均RTT是1.5秒（图21-4）。计算在同样条件下TFTP需要多长时间？

21.4 在第21.7节，我们提到过收到一个重复的 ACK是因为一个报文段丢失或重新进行排序。在21.5节我们看到1个丢失的报文段产生一些重复的 ACK。请画图表示重新排序也会产生一些重复的ACK。

21.5 在图21-6中的时刻28.8和29.8之间有一个显而易见的点，请问这是不是一个重传？

21.6 在21.6节我们提到过，如果目的地址位于一个不同的网络上， 4.3BSD Tahoe版本只执行慢启动。你认为在这里“不同的网络”是由什么决定的？（提示：参看附录 E）。

21.7 在20.2节我们提到过，在正常情况下， TCP每隔一个报文段进行一次确认，但是在图 21-2中，我们看到接收方对每个报文段都进行了确认，请解释其中的原因？

21.8 如果默认路由占优势，那么每路由（per-route）的度量是否真的有用？