第20章 TCP的成块数据流

20.1 引言

在第15章我们看到TFTP使用了停止等待协议。数据发送方在发送下一个数据块之前需要等待接收对已发送数据的确认。本章我们将介绍 TCP所使用的被称为滑动窗口协议的另一种形式的流量控制方法。该协议允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。

我们还将介绍TCP的PUSH标志，该标志在前面的许多例子中都出现过。此外，我们还要介绍慢启动，TCP使用该技术在一个连接上建立数据流，最后介绍成块数据流的吞吐量。

20.2 正常数据流

我们以从主机svr4单向传输8192个字节到主机bsdi开始。在bsdi上运行sock程序作为服务器：

```
bsdi %sock -i -s 7777
```

其中，标志-i和-s指示程序作为一个“吸收（sink）”服务器运行（从网络上读取并丢弃数据），服务器端口指明为7777。相应的客户程序运行为：

```
svr4 %sock -i -n8 bsdi 7777
```

该命令指示客户向网络发送8个1024字节的数据。图20-1显示了这个过程的时间系列。我们在输出的前3个报文段中显示了每一端MSS的值。

发送方首先传送3个数据报文段（4~6）。下一个报文段（7）仅确认了前两个数据报文段，这可以从其确认序号为2048而不是3073看出来。

报文段7的ACK的序号之所以是2048而不是3073是由以下原因造成的：当一个分组到达时，它首先被设备中断例程进行处理，然后放置到IP的输入队列中。三个报文段4、5和6依次到达并按接收顺序放到IP的输入队列。IP将按同样顺序将它们交给TCP。当TCP处理报文段4时，该连接被标记为产生一个经受时延的确认。TCP处理下一报文段（5），由于TCP现在有两个未完成的报文段需要确认，因此产生一个序号为2048的ACK（报文段7），并清除该连接产生经受时延的确认标志。TCP处理下一个报文段（6），而连接又被标志为产生一个经受时延的确认。在报文段9到来之前，由于时延定时器溢出，因此产生一个序号为3073的ACK（报文段8）。报文段8中的窗口大小为3072，表明在TCP的接收缓存中还有1024个字节的数据等待被应用程序读取。

报文段11~16说明了通常使用的“隔一个报文段确认”的策略。报文段11、12和13到达并被放入IP的接收队列。当报文段11被处理时，连接被标记为产生一个经受时延的确认。当报文段12被处理时，它们的ACK（报文段14）被产生且连接的经受时延的确认标志被清除。报文段13使得连接再次被标记为产生经受时延。但在时延定时器溢出之前，报文段15处理完毕，因此该确认立刻被发送。

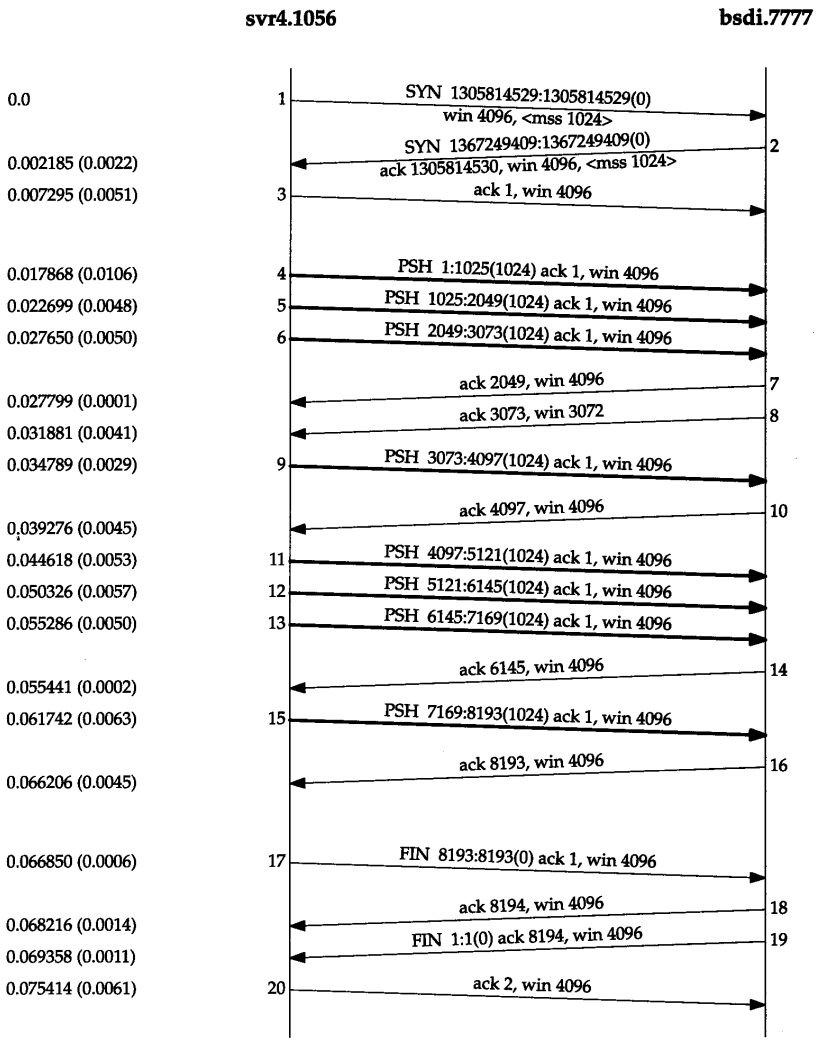

图20-1 从svr4 传输8192个字节到bsdi

注意到报文段7、14和16中的ACK确认了两个收到的报文段是很重要的。使用 TCP的滑动窗口协议时，接收方不必确认每一个收到的分组。在 TCP中，ACK是累积的——它们表示接收方已经正确收到了一直到确认序号减 1的所有字节。在本例中，三个确认的数据为 2048字节而两个确认的数据为1024字节（忽略了连接建立和终止中的确认）。

用tcpdump看到的是TCP的动态活动情况。我们在线路上看到的分组顺序依赖于许多无法控制的因素：发送方TCP的实现、接收方TCP的实现、接收进程读取数据（依赖于操作系统的调度）和网络的动态性（如以太网的冲突和退避等）。对这两个TCP而言，没有一种单一的、正确的方法来交换给定数量的数据。

为显示情况可能怎样变化，图 20-2显示了在同样两个主机之间交换同样数据时的另一个时间系列，它们是在图20-1所示的几分钟之后截获的。

一些情况发生了变化。这一次接收方没有发送一个序号为 3073的ACK，而是等待并发送序号为4097的ACK。接收方仅发送了4个ACK（报文段7、10、12和15）：三个确认了2048字

节，另一个确认了1024字节。最后1024字节数据的ACK出现在报文段17中，它与FIN的ACK一道发送（比较该图中的报文段17与图20-1中的报文段16和18）。

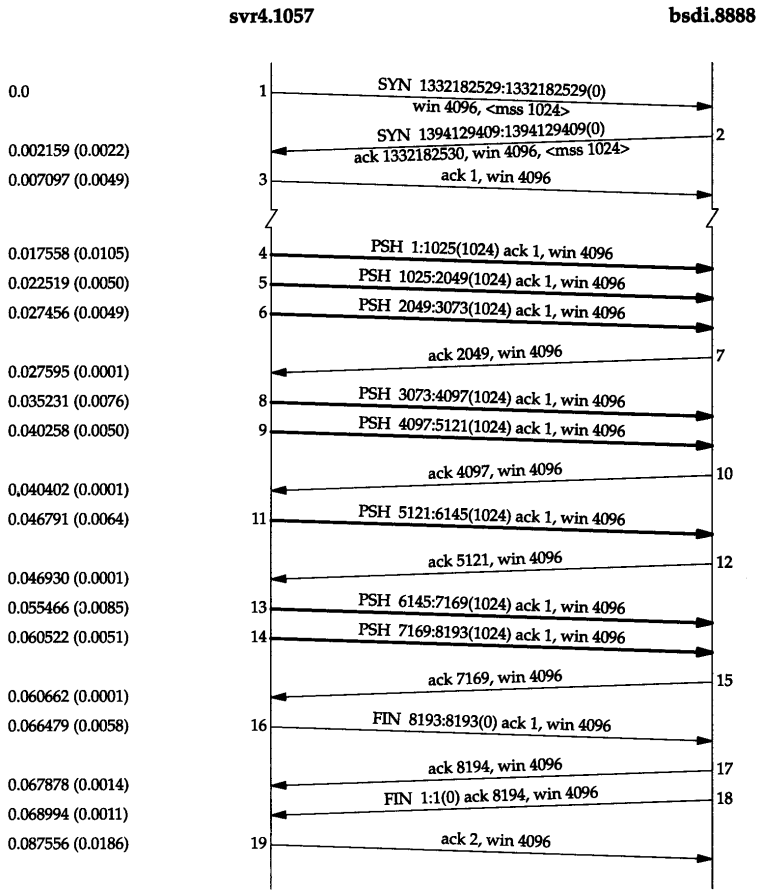

图20-2 从svr4 到bsdi 的另外8192字节数据的传输过程

快的发送方和慢的接收方

图20-3显示了另外一个时间系列。这次是从一个快的发送方（一个 Sparc工作站）到一个慢的接收方（配有慢速以太网卡的80386机器）。它的动态活动情况又有所不同。

发送方发送4个背靠背（back-to-back）的数据报文段去填充接收方的窗口，然后停下来等待一个ACK。接收方发送ACK（报文段8），但通告其窗口大小为0，这说明接收方已收到所有数据，但这些数据都在接收方的 TCP缓冲区，因为应用程序还没有机会读取这些数据。另一个ACK（称为窗口更新）在17.4 ms后发送，表明接收方现在可以接收另外的4096个字节的数据。虽然这看起来像一个 ACK，但由于它并不确认任何新数据，只是用来增加窗口的右边沿，因此被称为窗口更新。

发送方发送最后4个报文段（10~13），再次填充了接收方的窗口。注意到报文段13中包括两个比特标志：PUSH和FIN。随后从接收方传来另外两个 ACK，它们确认了最后的4096字节的数据（从4097到8192字节）和FIN（标号为8192）。

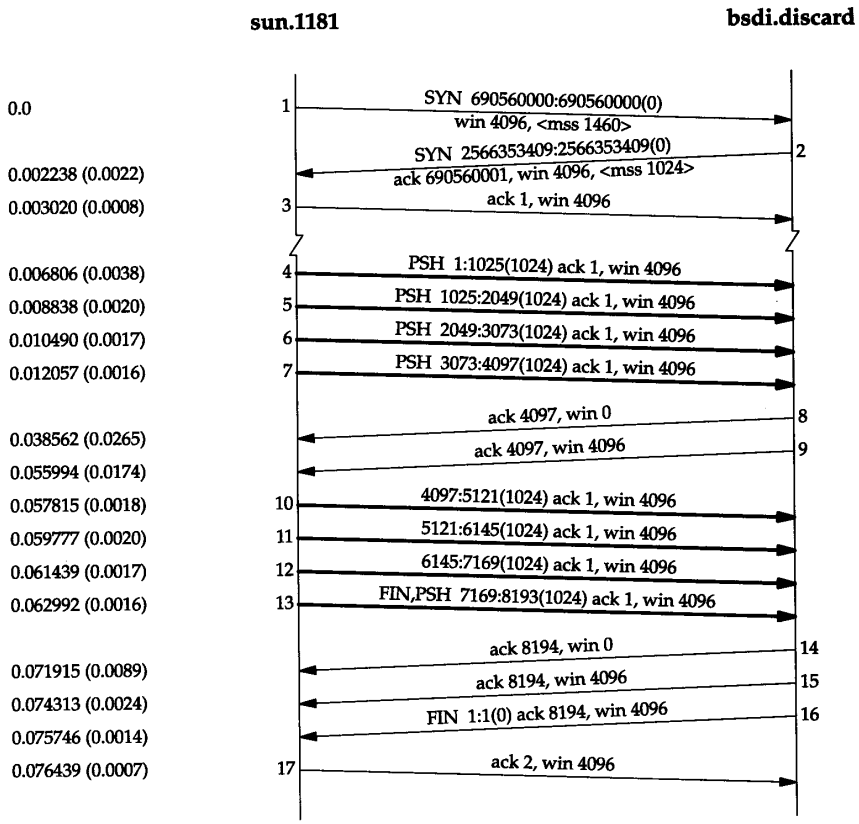

图20-3 从一个快发送方发送8192字节的数据到一个慢接收方

20.3 滑动窗口

图20-4用可视化的方法显示了我们在前一节观察到的滑动窗口协议。

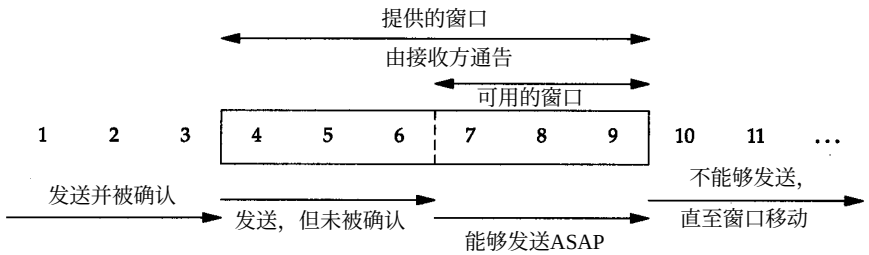

图20-4 TCP滑动窗口的可视化表示

在这个图中，我们将字节从1至11进行标号。接收方通告的窗口称为提出的窗口（offered window），它覆盖了从第4字节到第9字节的区域，表明接收方已经确认了包括第3字节在内的数据，且通告窗口大小为6。回顾第17章，我们知道窗口大小是与确认序号相对应的。发送方计算它的可用窗口，该窗口表明多少数据可以立即被发送。

当接收方确认数据后，这个滑动窗口不时地向右移动。窗口两个边沿的相对运动增加或减少了窗口的大小。我们使用三个术语来描述窗口左右边沿的运动：

1) 称窗口左边沿向右边沿靠近为窗口合拢。这种现象发生在数据被发送和确认时。

2) 当窗口右边沿向右移动时将允许发送更多的数据，我们称之为窗口张开。这种现象发生在另一端的接收进程读取已经确认的数据并释放了TCP的接收缓存时。

3) 当右边沿向左移动时，我们称之为窗口收缩。Host Requirements RFC强烈建议不要使用这种方式。但TCP必须能够在某一端产生这种情况时进行处理。第22.3节给出了这样的一个例子，一端希望向左移动右边沿来收缩窗口，但没能够这样做。

图20-5表示了这三种情况。因为窗口的左边沿受另一端发送的确认序号的控制，因此不可能向左边移动。如果接收到一个指示窗口左边沿向左移动的ACK，则它被认为是一个重复ACK，并被丢弃。

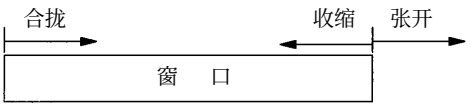

图20-5 窗口边沿的移动

如果左边沿到达右边沿，则称其为一个零窗口，此时发送方不能够发送任何数据。

一个例子

图20-6显示了在图20-1所示的数据传输过程中滑动窗口协议的动态性。

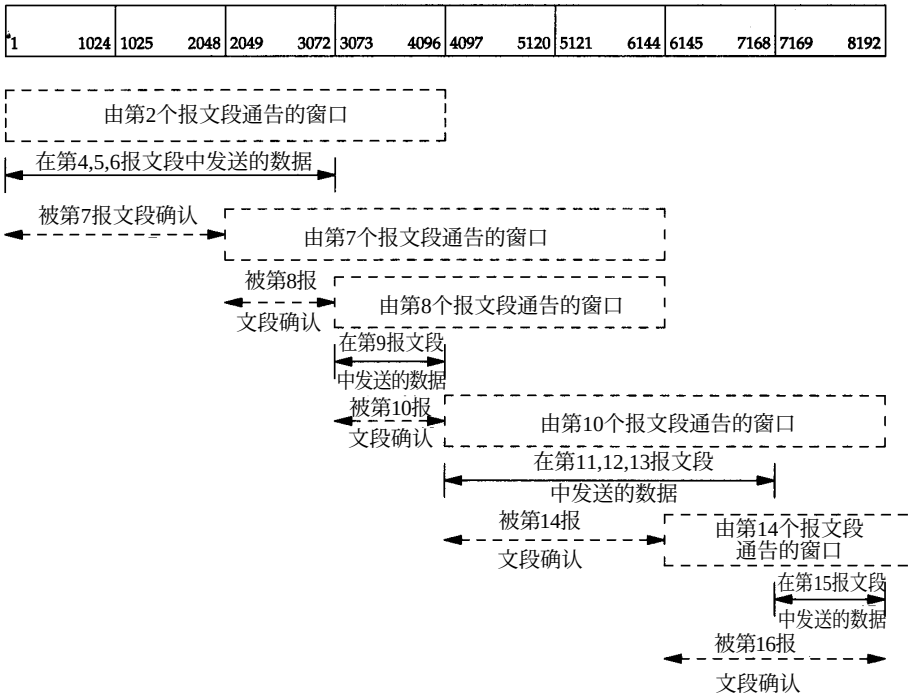

图20-6 图20-1的滑动窗口协议

以该图为例可以总结如下几点：

1) 发送方不必发送一个全窗口大小的数据。

2) 来自接收方的一个报文段确认数据并把窗口向右边滑动。这是因为窗口的大小是相对于确认序号的。

3) 正如从报文段7到报文段8中变化的那样，窗口的大小可以减小，但是窗口的右边沿却不能够向左移动。

4) 接收方在发送一个ACK前不必等待窗口被填满。在前面我们看到许多实现每收到两个报文段就会发送一个ACK。

下面我们可以看到更多的滑动窗口协议动态变化的例子。

20.4 窗口大小

由接收方提供的窗口的大小通常可以由接收进程控制，这将影响TCP的性能。

4.2BSD默认设置发送和接受缓冲区的大小为2048个字节。在4.3BSD中双方被增加为4096个字节。正如我们在本书中迄今为止所看到的例子一样，SunOS 4.1.3、BSD/386和SVR4仍然使用4096字节的默认大小。其他的系统，如Solaris 2.2、4.4BSD和AIX3.2则使用更大的默认缓存大小，如8192或16384等。

插口API允许进程设置发送和接收缓存的大小。接收缓存的大小是该连接上所能够通告的最大窗口大小。有一些应用程序通过修改插口缓存大小来增加性能。

[Mogul 1993]显示了在改变发送和接收缓存大小（在单向数据流的应用中，如文件传输，只需改变发送方的发送缓存和接收方的接收缓存大小）的情况下，位于以太网上的两个工作站之间进行文件传输时的一些结果。它表明对以太网而言，默认的4096字节并不是最理想的大小，将两个缓存增加到16384个字节可以增加约40%左右的吞吐量。在[Papadopoulos和Parulkar 1993]中也有相似的结果。

在20.7节中，我们将看到在给定通信媒体带宽和两端往返时间的情况下，如何计算最小的缓存大小。

一个例子

可以使用sock程序来控制这些缓存的大小。我们以如下方式调用服务器程序：

```
bsdi % sock -i -s -R6144 5555
```

该命令设置接收缓存为6144个字节（-R选项）。接着我们在主机sun上启动客户程序并使之发送8192个字节的数据：

```
sun % sock -i -n1 -w8192 bsdi 5555
```

图20-7显示了结果。

首先注意到的是在报文段2中提供的窗口大小为6144字节。由于这是一个较大的窗口，因此客户立即连续发送了6个报文段（4~9），然后停止。报文段10确认了所有的数据（从第1到6144字节），但提供的窗口大小却为2048，这很可能是接收程序没有机会读取多于2048字节的数据。报文段11和12完成了客户的数据传输，且最后一个报文段带有FIN标志。

报文段13包含与报文段10相同的确认序号，但通告了一个更大的窗口大小。报文段14确认了最后的2048字节的数据和FIN，报文段15和16仅用于通告一个更大的窗口大小。报文段17和18完成通常的关闭过程。

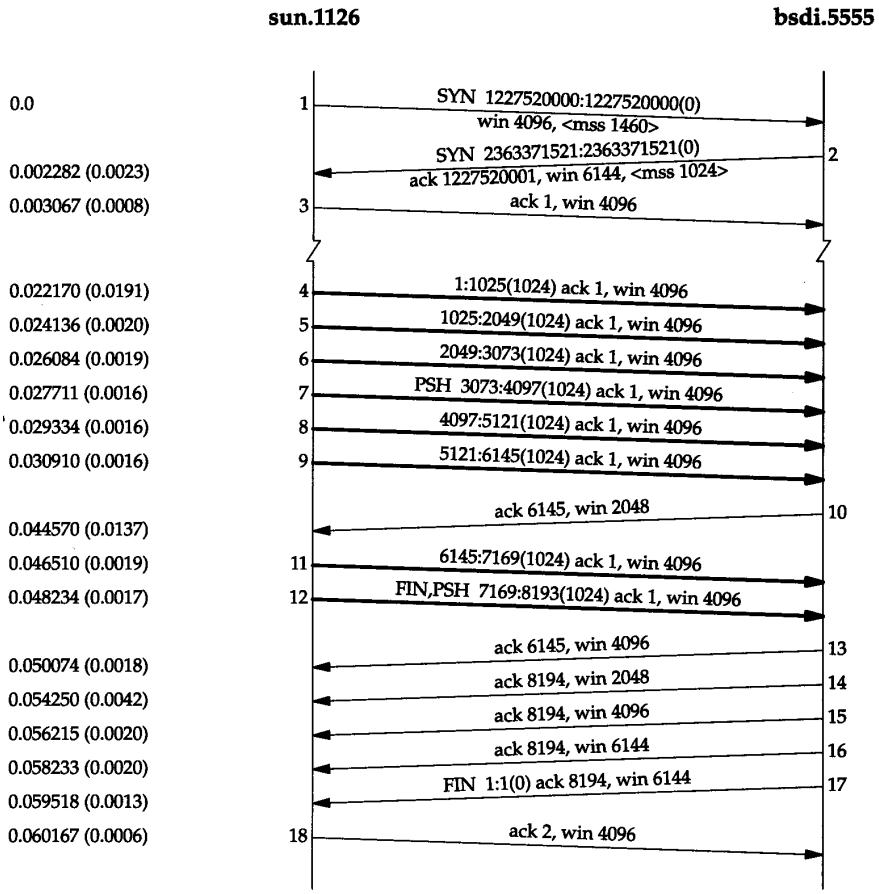

图20-7 接收方提供一个6144字节的接收窗口的情况下的数据传输

20.5 PUSH标志

在每一个TCP例子中，我们都看到了PUSH标志，但一直没有介绍它的用途。发送方使用该标志通知接收方将所收到的数据全部提交给接收进程。这里的数据包括与 PUSH一起传送的数据以及接收方TCP已经为接收进程收到的其他数据。

在最初的TCP规范中，一般假定编程接口允许发送进程告诉它的 TCP何时设置PUSH标志。例如，在一个交互程序中，当客户发送一个命令给服务器时，它设置 PUSH标志并停下来等待服务器的响应（在习题 19.1中我们假定当发送 12字节的请求时客户设置 PUSH标志）。通过允许客户应用程序通知其TCP设置PUSH标志，客户进程通知TCP在向服务器发送一个报文段时不要因等待额外数据而使已提交数据在缓存中滞留。类似地，当服务器的 TCP接收到一个设置了PUSH标志的报文段时，它需要立即将这些数据递交给服务器进程而不能等待判断是否还会有额外的数据到达。

然而，目前大多数的 API没有向应用程序提供通知其 TCP设置PUSH标志的方法。的确，许多实现程序认为PUSH标志已经过时，一个好的TCP实现能够自行决定何时设置这个标志。

如果待发送数据将清空发送缓存，则大多数的源于伯克利的实现能够自动设置 PUSH标志。这意味着我们能够观察到每个应用程序写的数据均被设置了 PUSH标志，因为数据在写的时候

就立即被发送。

代码中的注释表明该算法对那些只有在缓存被填满或收到一个PUSH标志时才向应用程序提交数据的TCP实现有效。

使用插口API通知TCP设置正在接收数据的PUSH标志或得到该数据是否被设置PUSH标志的信息是不可能的。

由于源于伯克利的实现一般从不将接收到的数据推迟交付给应用程序，因此它们忽略所接收的PUSH标志。

举例

在图20-1中我们观察到所有8个数据报文段（4~6、9、11~13和15）的PUSH标志均被置1，这是因为客户进行了8次1024字节数据的写操作，并且每次写操作均清空了发送缓存。

再次观察图20-7，我们预计报文段12中的PUSH标志被置1，因为它是最后一个报文段。为什么发送方知道有更多的数据需要发送还设置报文段 7中的PUSH标志呢？这是因为虽然我们指定写的是8192个字节的数据，但发送方的发送缓存却是4096个字节。

值得注意的另外一点是在图 20-7中的第14、15和16这三个连续的确认报文段。在图 20-3中我们也观察到了两个连续的ACK，但那是因为接收方已经通告其窗口为0（使发送方停止）。当窗口张开时，需要发送另一个窗口非0的ACK来使发送方重新启动。可是，在图20-7中，窗口的大小从来没有达到过0。然而，当窗口大小增加了2048个字节的时候，另一个ACK(报文段15和16)被发送以通知对方窗口被更新（在报文段 15和16中，这两个窗口更新是不需要的，因为已经收到了对方的FIN，表明它不会再发送任何数据）。许多TCP实现在窗口大小增加了两个最大报文段长度（本例中为2048字节，因为MSS为1024字节）或者最大可能窗口的50%（本例中为2048字节，因为最大窗口大小为4096字节）时发送这个窗口更新。在第22.3节详细考察糊涂窗口综合症的时候，我们还会看到这种现象。

作为PUSH标志的另一个例子，再次回到图20-3。我们之所以看到前4个报文段（4~7）的标志被设置，是因为它们每一个均使TCP产生了一个报文段并提交给IP层。但是随后，TCP停下来等待一个确认来移动4096字节的窗口。在此期间，TCP又得到了应用程序的最后4096个字节的数据。当窗口张开时（报文段9），发送方TCP知道它有4个可立即发送的报文段，因此它只设置了最后一个报文段（13）的PUSH标志。

20.6 慢启动

迄今为止，在本章所有的例子中，发送方一开始便向网络发送多个报文段，直至达到接收方通告的窗口大小为止。当发送方和接收方处于同一个局域网时，这种方式是可以的。但是如果在发送方和接收方之间存在多个路由器和速率较慢的链路时，就有可能出现一些问题。一些中间路由器必须缓存分组，并有可能耗尽存储器的空间。 [Jacobson 1988]证明了这种连接方式是如何严重降低了TCP连接的吞吐量的。

现在，TCP需要支持一种被称为“慢启动(slow start)”的算法。该算法通过观察到新分组进入网络的速率应该与另一端返回确认的速率相同而进行工作。

慢启动为发送方的TCP增加了另一个窗口：拥塞窗口(congestion window)，记为cwnd。当

与另一个网络的主机建立TCP连接时，拥塞窗口被初始化为1个报文段（即另一端通告的报文段大小）。每收到一个ACK，拥塞窗口就增加一个报文段（cwnd以字节为单位，但是慢启动以报文段大小为单位进行增加）。发送方取拥塞窗口与通告窗口中的最小值作为发送上限。拥塞窗口是发送方使用的流量控制，而通告窗口则是接收方使用的流量控制。

发送方开始时发送一个报文段，然后等待ACK。当收到该ACK时，拥塞窗口从1增加为2，即可以发送两个报文段。当收到这两个报文段的ACK时，拥塞窗口就增加为4。这是一种指数增加的关系。

在某些点上可能达到了互联网的容量，于是中间路由器开始丢弃分组。这就通知发送方它的拥塞窗口开得过大。当我们在下一章讨论TCP的超时和重传机制时，将会看到它们是怎样对拥塞窗口起作用的。现在，我们来观察一个实际中的慢启动。

一个例子

图20-8表示的是将从主机sun发送到主机vangogh.cs.berkeley.edu的数据。这些数据将通过一个慢的SLIP链路，该链路是TCP连接上的瓶颈（我们已经在时间系列上去掉了连接建立的过程）。

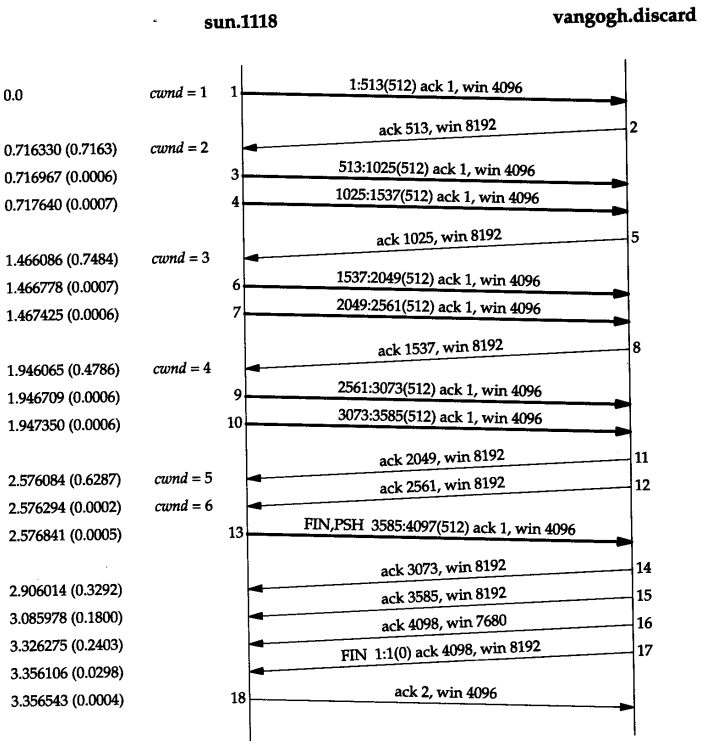

图20-8 慢启动的例子

我们观察到发送方发送一个长度为512字节的报文段，然后等待ACK。该ACK在716 ms后收到。这个时间是一个往返时间的指示。于是拥塞窗口增加了2个报文段，且又发送了两个报文段。当收到报文段5的ACK后，拥塞窗口增加为3。此时尽管可发送多达3个报文段，可是在下一个ACK收到之前，只发送了2个报文段。

在21.6节中我们将再次讨论慢启动，并介绍怎样采用另一种被称为“拥塞避免”的技术来

作为通常的实现。

20.7 成块数据的吞吐量

让我们看一看窗口大小、窗口流量控制以及慢启动对传输成块数据的 TCP连接的吞吐量的相互作用。

图20-9显示了左边的发送方和右边的接收方之间的一个 TCP连接上的时间系列，共显示了16个时间单元。为简单起见，本图只显示离散的时间单元。每个粗箭头线的上半部分显示的是从左到右的携带数据的报文段，标记为 1, 2, 3, 等等。在粗线箭头下面表示的是反向传输的ACK。我们把ACK用细箭头线表示，并标注了被确认的报文段号。

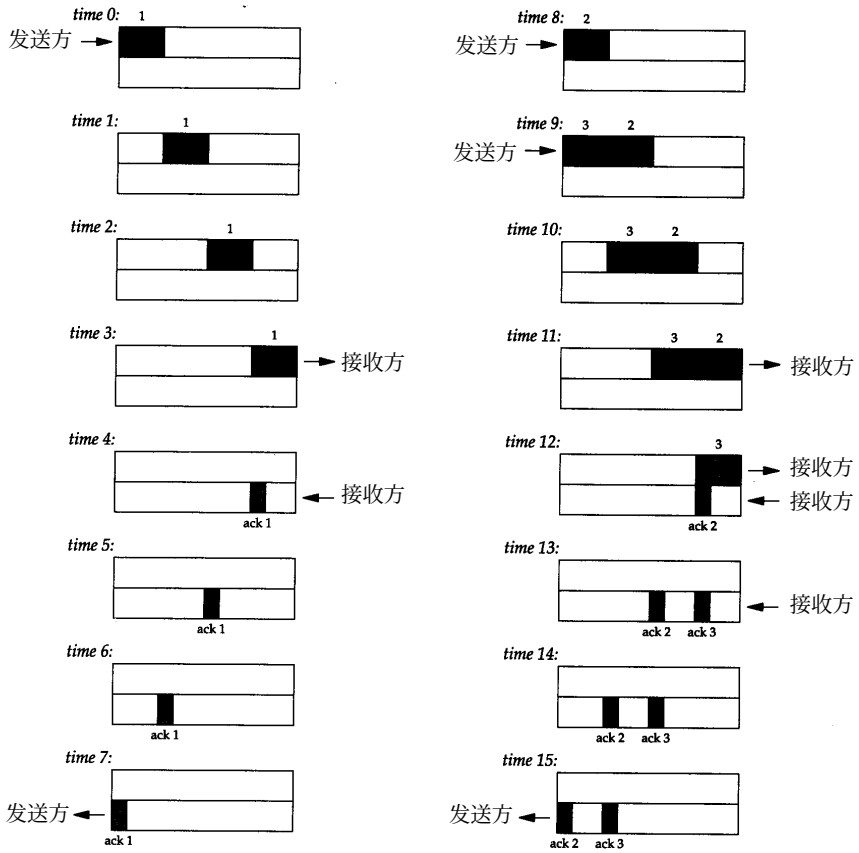

图20-9 时间0~15的成块数据吞吐量举例

在时间0，发送方发送了一个报文段。由于发送方处于慢启动中（其拥塞窗口为 1个报文段），因此在继续发送以前它必须等待该数据段的确认。

在时间1, 2和3，报文段从左向右移动一个时间单元。在时间 4接收方读取这个报文段并产生确认。经过时间 5、6和7，ACK移动到左边的发送方。我们有了一个 8个时间单元的往返时间RTT（Round-Trip Time）。

我们有意把 ACK报文段画得比数据报文段小，这是因为它通常只有一个 IP首部和一个TCP首部。这里显示仅仅是一个单向的数据流动，并且假定 ACK的移动速率与数据报文段的移动速率相等。实际上并不总是这样。

通常发送一个分组的时间取决于两个因素：传播时延（由光的有限速率、传输设备的等待时间等引起）和一个取决于媒体速率（即媒体每秒可传输的比特数）的发送时延。对于一个给定的两个接点之间的通路，传播时延一般是固定的，而发送时延则取决于分组的大小。在速率较慢的情况下发送时延起主要作用（例如，在习题7.2中我们甚至没有考虑传播时延），而在千兆比特速率下传播时延则占主要地位（见图24-6）。

当发送方收到ACK后，在时间8和9发送两个报文段（我们标记为2和3）。此时它的拥塞窗口为2个报文段。这两个报文段向右传送到接收方，在时间12和13接收方产生两个ACK。这两个返回到发送方的ACK之间的间隔与报文段之间的间隔一致，被称为TCP的自计时(self-clocking)行为。由于接收方只有在数据到达时才产生ACK，因此发送方接收到的ACK之间的间隔与数据到达接收方的间隔是一致的（然而在实际中，返回路径上的排队会改变ACK的到达率）。

图20-10表示的是后面16个时间单位。2个ACK的到达使得拥塞窗口从2个报文段增加为4个，而这4个报文段在时间16~19时被发送。第1个ACK在时间23到达。4个ACK的到达使得拥塞窗口从4个报文段增加为8个，并在时间24~31发送8个报文段。

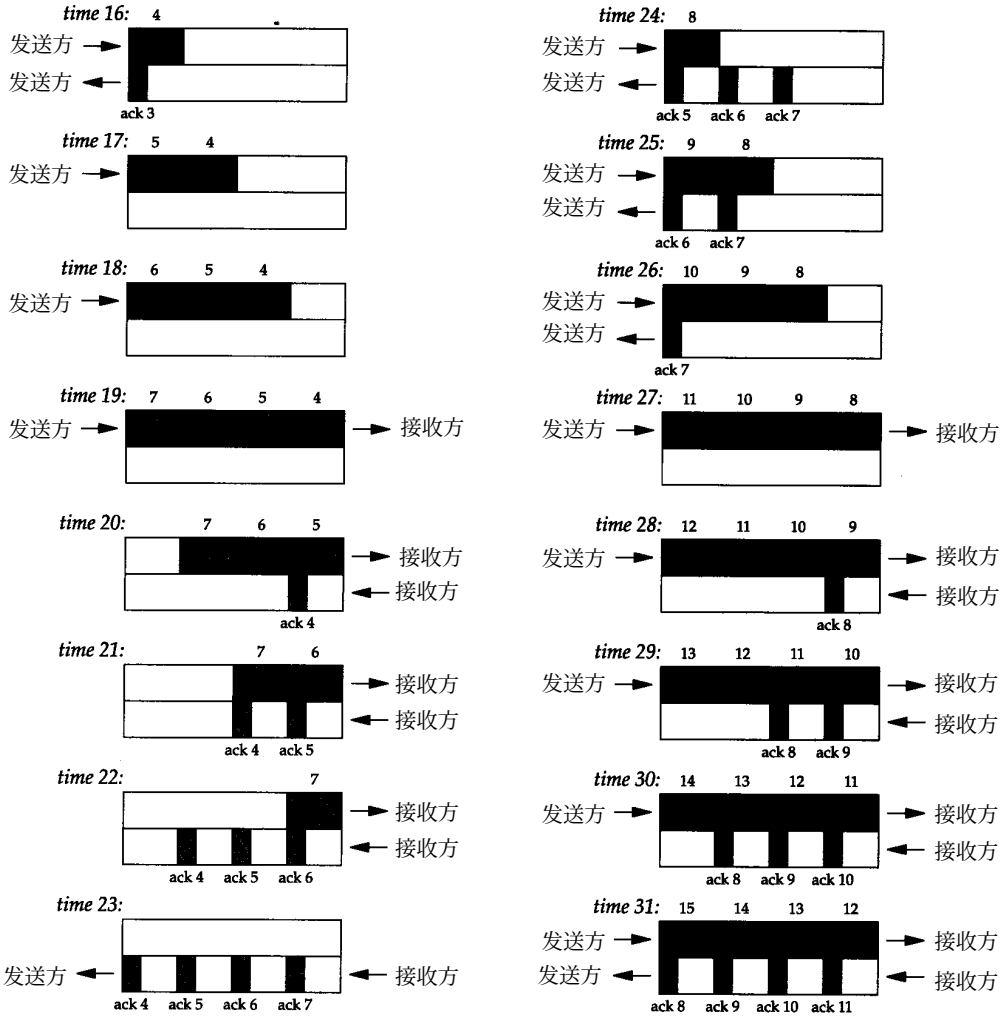

图20-10 时间16~31的成块数据吞吐量举例

在时间31及其后续时间，发送方和接收方之间的管道 (pipe)被填满。此时不论拥塞窗口和通告窗口是多少，它都不能再容纳更多的数据。每当接收方在某一个时间单位从网络上移去一个报文段，发送方就再发送一个报文段到网络上。但是不管有多少报文段填充了这个管道，返回路径上总是具有相同数目的 ACK。这就是连接的理想稳定状态。

20.7.1 带宽时延乘积

现在来回答窗口应该设置为多大的问题。在我们的例子中，作为最大的吞吐量，发送方在任何时候有8个已发送的报文段未被确认。接收方的通告窗口必须不小于这个数目，因为通告窗口限制了发送方能够发送的段的数目。

可以计算通道的容量为：

capacity (bit) = *bandwidth* (b/s) × *round-trip time* (s)

一般称之为带宽时延乘积。这个值依赖于网络速度和两端的 RTT，可以有很大的变动。例如，一条穿越美国（RTT约为60 ms）的T1的电话线路（1 544 000 b/s）的带宽时延乘积为11 580字节。对于20.4节中讨论的缓存大小而言，这个结果是合理的。但是一条穿越美国的 T3电话线路（45 000 000 b/s）的带宽时延乘积则为337 500字节，这个数值超过了最大所允许的TCP通告窗口的大小（65535字节）。在24.4节我们将讨论能够避免当前TCP限制的新的TCP窗口大小选项。

> T1电话线的1 544 000 b/s是原始比特率。由于每193个bit使用1个作为帧同步，因此实际数据率为1 536 000 b/s。一个T3电话线的原始比特率实际上是44 736 000 b/s，其数据率可达到44 210 000 b/s。在讨论中我们使用1.544 Mb/s和45 Mb/s。

不论是带宽还是时延均会影响发送方和接收方之间通路的容量。在图 20-11中我们显示了一个增加了一倍的RTT会使通路容量也增加一倍。

在图20-11底下的说明部分，通过使用一个较长的 RTT，这个管道能够容纳8个报文段而不是4个。

类似地，图20-12表示了增加一倍的带宽也可使该管道的容量增加一倍。

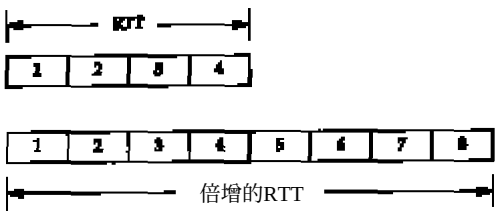

图20-11 RTT加倍可使管道容量增加一倍

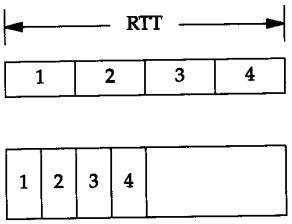

图20-12 带宽加倍可使管道容量增加一倍

在图20-12的下部，假定网络速率已经加倍，使得我们能够只使用上面一半的时间来发送4个报文段。这样，该管道的容量再次加倍（假定该图的上半部分与下半部分中的报文段具有同样大小，即具有相同的比特数）。

20.7.2 拥塞

当数据到达一个大的管道（如一个快速局域网）并向一个较小的管道（如一个较慢的广

域网）发送时便会发生拥塞。当多个输入流到达一个路由器，而路由器的输出流小于这些输入流的总和时也会发生拥塞。

图20-13显示了一个典型的大管道向小管道发送报文的情况。之所以说它典型，是因为大多数的主机都连接在局域网上，并通过一个路由器与速率相对较低的广域网相连（我们再次假定图中上半部分的报文段（9~20）都是相同的，而图中下半部分的ACK也都是相同的）。

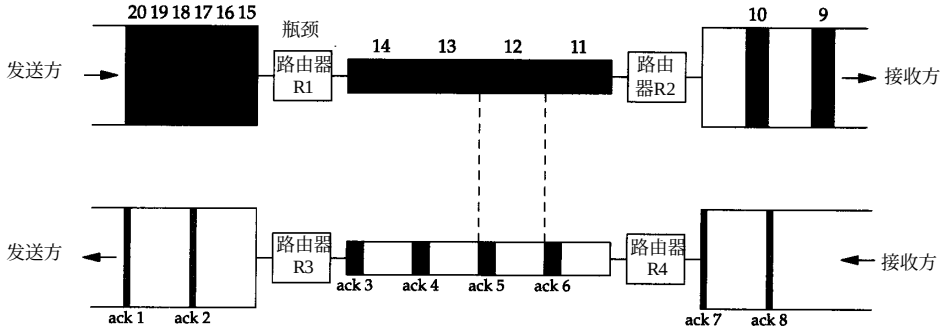

图20-13 从较大管道向较小管道发送分组引起的拥塞

在该图中，我们已经标记路由器 R1为“瓶颈”，因为它是拥塞发生的地方。它从左侧速率较高的局域网接收数据并向右侧速率较低的广域网发送（通常 R1与R3是同样的路由器，如同R2与R4一样。但这并不是必需的，有时也会使用不对称的路径）。当路由器R2将所接收到的分组发送到右侧的局域网时，这些分组之间维持与其左侧广域网上同样的间隔，尽管局域网具有更高的带宽。类似地，返回的确认之间的间隔也与其在路径中最慢的链路上的间隔一致。

在图20-13中已经假定发送方不使用慢启动，它按照局域网的带宽尽可能快地发送编号为1~20的报文段（假定接收方的通告窗口至少为 20个报文段）。正如我们看到的那样， ACK之间的间隔与在最慢链路上的一致。假定瓶颈路由器具有足够的容纳这 20个分组的缓存。如果这个不能保证，就会引起路由器丢弃分组。在 21.6节讨论避免拥塞时会看到怎样避免这种情况。

20.8 紧急方式

TCP提供了“紧急方式 (urgent mode)”，它使一端可以告诉另一端有些具有某种方式的“紧急数据”已经放置在普通的数据流中。另一端被通知这个紧急数据已被放置在普通数据流中，由接收方决定如何处理。

可以通过设置TCP首部（图17-2）中的两个字段来发出这种从一端到另一端的紧急数据已经被放置在数据流中的通知。 URG比特被置1，并且一个16bit的紧急指针被置为一个正的偏移量，该偏移量必须与 TCP首部中的序号字段相加，以便得出紧急数据的最后一个字节的序号。

仍有许多关于紧急指针是指向紧急数据的最后一个字节还是指向紧急数据最后一个字节的下一个字节的争论。最初的TCP规范给出了两种解释，但Host Requirements RFC确定指向最后一个字节是正确的。

然而，问题在于大多数的实现（包括源自伯克利的实现）继续使用错误的解释。

所有符合Host Requirements RFC的实现都是可兼容的，但很有可能无法与其他大多数主机正确通信。

TCP必须通知接收进程，何时已接收到一个紧急数据指针以及何时某个紧急数据指针还不在此连接上，或者紧急指针是否在数据流中向前移动。接着接收进程可以读取数据流，并必须能够被告知何时碰到了紧急数据指针。只要从接收方当前读取位置到紧急数据指针之间有数据存在，就认为应用程序处于“紧急方式”。在紧急指针通过之后，应用程序便转回到正常方式。

TCP本身对紧急数据知之甚少。没有办法指明紧急数据从数据流的何处开始。TCP通过连接传送的唯一信息就是紧急方式已经开始（TCP首部中的URG比特）和指向紧急数据最后一个字节的指针。其他的事情留给应用程序去处理。

不幸的是，许多实现不正确地称TCP的紧急方式为带外数据(out-of-band data)。如果一个应用程序确实需要一个独立的带外信道，第二个TCP连接是达到这个目的的最简单的方法(许多运输层确实提供许多人认为的那种真正的带外数据：使用同一个连接的独立的逻辑数据通道作为正常的数据通道。这是TCP所没有提供的)。

TCP的紧急方式与带外数据之间的混淆，也是因为主要的编程接口（插口API）将TCP的紧急方式映射为称为带外数据的插口。

紧急方式有什么作用呢？两个最常见的例子是Telnet和Rlogin。当交互用户键入中断键时，我们在第26章将看到使用紧急方式来完成这个功能的例子。另一个例子是FTP，当交互用户放弃一个文件的传输时，我们将在第27章看到这样的一个例子。

Telnet和Rlogin从服务器到客户使用紧急方式是因为在这个方向上的数据流很可能要被客户的TCP停止（也即，它通告了一个大小为0的窗口）。但是如果服务器进程进入了紧急方式，尽管它不能够发送任何数据，服务器TCP也会立即发送紧急指针和URG标志。当客户TCP接收到这个通知时就会通知客户进程，于是客户可以从服务器读取其输入、打开窗口并使数据流动。

如果在接收方处理第一个紧急指针之前，发送方多次进入紧急方式会发生什么情况呢？在数据流中的紧急指针会向前移动，而其在接收方的前一个位置将丢失。接收方只有一个紧急指针，每当对方有新的值到达时它将被覆盖。这意味着如果发送方进入紧急方式时所写的内容对接收方非常重要，那么这些字节数据必须被发送方用某种方式特别标记。我们将看到Telnet通过在数据流中加入一个值为255的字节作为前缀来标记它所有的命令。

一个例子

让我们观察一下即使是在接收方窗口关闭的情况下，TCP是如何发送紧急数据的。在主机bsdi上启动sock程序，并使之在连接建立后和从网络读取前暂停10秒种（通过使用-P选项），这将使另一端填满发送窗口：

```
bsdi %sock -i -s -P10 5555
```

接着我们在主机sun上启动客户，使之使用一个8192字节的发送缓存（使用-S选项）并进行6个向网络写1024字节数据的操作（使用-n选项）。还指明-U5选项，告知它向网络写第5个缓存之前要写1个字节的数据，并进入紧急数据方式。我们指明详细标志来观察写的顺序：

```
sun % sock -v -i -n6 -S8192 -U5 bsdi 5555
connected on 140.252.13.33.1305 to 140.252.13.35.5555
SO_SNDBUF = 8192
TCP_MAXSEG = 1024
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1 byte of urgent data
wrote 1024 bytes
wrote 1024 bytes
```

我们设置发送缓存为8192个字节，以便让发送应用程序能够立即写所有的数据。图 20-14显示了tcpdump输出的这个交换过程的结果（删去了连接建立的过程）。第1~5行表示发送方用4个1024字节的报文段去填充接收方的窗口。然后由于接收方的窗口被填满（第 4行的ACK确认了数据，但并没有移动窗口的右边沿），所以发送方停止发送。

在写了第4个正常数据之后，应用进程写了1个字节并进入紧急方式。第6行是该应用进程写的结果，紧急指针被设置为4098。尽管发送方不能发送任何数据，但紧急指针和URG标志一起被发送。

5个这样的ACK在13 ms内被发送（第6~10行）。第1个ACK在应用进程写1个字节并进入紧急方式时被发送，后面两个在应用进程写最后两个 1024字节的数据时被发送（尽管 TCP不能发送这2048个字节的数据，可每次当应用程序执行写操作的时候， TCP的输出功能被调用。当TCP看到正处于紧急方式时，它会发送其他的紧急通知）。第4个ACK在应用进程关闭其TCP连接时被发送（TCP的输出功能再次被调用）。发送应用程序在启动几毫秒后终止 —— 在接收方应用进程已经发出其第一个写操作之前。 TCP将所有的数据进行排队，并在可能时发送出去（这就是为何指明发送缓存为 8192字节的原因，因此只有这样才能够把所有的数据都放置在缓存中）。第5个ACK很可能是在接收第 4行的ACK时产生的。发送 TCP很可能在这个ACK到达前便已将其第 4个报文段放入队列以便输出（第 5行）。另一端接收到这个 ACK也会引起TCP输出例程被调用。

```
 1  0.0                  sun.1305 > bsdi.5555: P 1:1025(1024) ack 1 win 4096
 2  0.073743 (0.0737)    sun.1305 > bsdi.5555: P 1025:2049(1024) ack 1 win 4096
 3  0.096969 (0.0232)    sun.1305 > bsdi.5555: P 2049:3073(1024) ack 1 win 4096
 4  0.157514 (0.0605)    bsdi.5555 > sun.1305: . ack 3073 win 1024
 5  0.164267 (0.0068)    sun.1305 > bsdi.5555: P 3073:4097(1024) ack 1 win 4096

 6  0.167961 (0.0037)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 7  0.171969 (0.0040)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 8  0.176196 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 9  0.180373 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
10  0.180768 (0.0004)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

11  0.367533 (0.1868)    bsdi.5555 > sun.1305: . ack 4097 win 0
12  0.368478 (0.0009)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

13  9.829712 (9.4612)    bsdi.5555 > sun.1305: . ack 4097 win 2048
14  9.831578 (0.0019)    sun.1305 > bsdi.5555: . 4097:5121(1024) ack 1 win 4096
                                                 urg 4098
15  9.833303 (0.0017)    sun.1305 > bsdi.5555: . 5121:6145(1024) ack 1 win 4096

16  9.835089 (0.0018)    bsdi.5555 > sun.1305: . ack 4097 win 4096
17  9.835913 (0.0008)    sun.1305 > bsdi.5555: FP 6145:6146(1) ack 1 win 4096
18  9.840264 (0.0044)    bsdi.5555 > sun.1305: . ack 6147 win 2048
19  9.842386 (0.0021)    bsdi.5555 > sun.1305: . ack 6147 win 4096
20  9.843622 (0.0012)    bsdi.5555 > sun.1305: F 1:1(0) ack 6147 win 4096
21  9.844320 (0.0007)    sun.1305 > bsdi.5555: . ack 2 win 4096
```

图20-14 tcpdump 对TCP紧急方式的输出结果

接着，接收方确认最后的1024字节的数据（第11行），但同时通告窗口为0。发送方用一个包含紧急通知的报文段进行了响应。

在第13行，当应用进程被唤醒、并从接收缓存读取一些数据时，接收方通告窗口为 2048字节。于是后面又发送了两个1024字节的报文段（第14和15行）。其中，由于紧急指针在第1个报文段的范围内，因此这个报文段被设置了紧急通知标志，而第 2个报文段则关闭了该标志。

当接收方再次打开窗口（第16行）时，发送方传输最后的数据（序号为6145）并发起正常的连接关闭。

图20-15显示了发送的6145个字节数据的序号。可以看到当进入紧急方式时所发送的字节的序号是4097，但在图20-14中紧急指针指向4098，这证明了该实现（SunOS 4.1.3）将紧急指针设置为紧急数据最后字节的下一个字节。

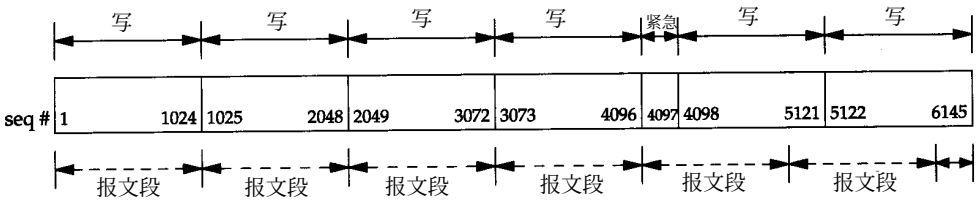

图20-15 紧急方式例子中，应用进程的写操作和TCP的一些报文段

该图还可以让我们观察TCP是如何对应用进程写的数据进行重新分组化的。当进入紧急方式时待输出的1个字节是与在缓存中的后面1023个字节一同发送的。下一个报文段也包含1024字节的数据，而最后一个报文段则只包含一个字节。

20.9 小结

正如我们在本章一开始时讲的那样，没有一种单一的方法可以使用 TCP进行成块数据的交换。这是一个依赖于许多因素的动态处理过程，有些因素我们可以控制（如发送和接收缓存的大小），而另一些我们则没有办法控制（如网络拥塞、与实现有关的特性等）。在本章，我们已经考察了许多TCP的传输过程，介绍了所有我们能够看到的特点和算法。

进行成块数据有效传输的最重要的方法是TCP的滑动窗口协议。我们考察了TCP为使发送方和接收方之间的管道充满来获得最可能快的传输速度而采用的方法。我们用带宽时延乘积衡量管道的容量，并分析了该乘积与窗口大小之间的关系。在24.8节介绍TCP性能的时候将再次涉及这个概念。

我们还介绍了TCP的PUSH标志，因为在跟踪结果中总是观察到它，但我们无法对它的设置与否进行控制。本章最后一个主题是TCP的紧急数据，人们常常错误地称其为“带外数据”。TCP的紧急方式只是一个从发送方到接收方的通知，该通知告诉接收方紧急数据已被发送，并提供该数据最后一个字节的序号。应用程序使用的有关紧急数据部分的编程接口常常都不是最佳的，从而导致更多的混乱。

习题

20.1 在图20-6中，我们可以看到一个序号为0的字节和一个序号为8193的字节，试问这两个

字节的含义是什么？

20.2 提前观察图22-1，并解释主机bsdi设置PUSH标志的含义。

20.3 在一个Usenet记录中，有人抱怨说美国和日本之间的一个128 ms时延、速率为256 000 b/s的链路吞吐量为120 000 b/s（利用率为47%），而当链路通过卫星时其吞吐量则为33 000 b/s（利用率为13%）。试问在这两种情况下窗口大小各为多少（假定卫星链路的时延为500 ms）？卫星链路的窗口大小应该如何调整？

20.4 如果API提供一种方法，使得发送方可以告诉其TCP打开PUSH标志，而接收方可以查询一个接收的报文段是否被设置了PUSH标志，试问该标志能否被用作一个记录标记？

20.5 在图20-3中为什么没有合并报文段15和16？

20.6 在图20-13中，我们假定对应数据报文段之间的间隔，返回的ACK之间的间隔被分隔得很好。如果在链路某处进行缓存并使许多ACK同时到达发送方，试问会发生什么情况？