

第24章 TCP的未来和性能

24.1 引言

TCP已经在从1200 b/s的拨号SLIP链路到以太网数据链路上运行了许多年。在80年代和90年代初期，以太网是运行TCP/IP最主要的数据链路方式。虽然TCP在比以太网速率高的环境（如T2电话线、FDDI及千兆比网络）中也能够正确运行，但在这些高速率环境下，TCP的某些限制就会暴露出来。

本章讨论TCP的一些修改建议，这些建议可以使TCP在高速率环境中获得最大的吞吐量。首先要讨论前面已经碰到过的路径MTU发现机制，本章主要关注它如何与TCP协同工作。这个机制通常可以使TCP为非本地的连接使用大于536字节的MTU，从而增加吞吐量。

接着介绍长肥管道(long fat pipe)，也就是那些具有很大的带宽时延乘积的网络，以及TCP在这些网络上所具有的局限性。为处理长肥管道，我们描述两个新的TCP选项：窗口扩大选项（用来增加TCP的最大窗口，使之超过65535字节）和时间戳选项。后面这个选项可以使TCP对报文段进行更加精确的RTT测量，还可以在高速率下对可能发生的序号回绕提供保护。这两个选项在RFC 1323 [Jacobson, Braden, and Borman 1992]中进行定义。

我们还将介绍建议的T/TCP，这是为增加事务功能而对TCP进行的修改。通信的事务模式以客户请求将被服务器应答的响应为主要特征。这是客户服务器计算的常见模型。T/TCP的目的就是减少两端交换的报文段数量，避免三次握手和使用4个报文段进行连接的关闭，从而使客户可以在一个RTT和处理请求所必需的时间内收到服务器的应答。

这些新选项（路径MTU发现、窗口扩大选项、时间戳选项和T/TCP）中令人印象最深刻的就是它们与现有的TCP实现能够向后兼容，即包括这些新选项的系统仍然可以与原有的旧系统进行交互。除了在一个ICMP报文中为路径MTU发现增加了一个额外字段之外，这些新的选项只需要在那些需要使用它们的端系统中进行实现。

我们以介绍近来发表的有关TCP性能的图例作为本章的结束。

24.2 路径MTU发现

在2.9节我们描述了路径MTU的概念。这是当前在两个主机之间的路径上任何网络上的最小MTU。路径MTU发现在IP首部中继承并设置“不要分片(DF)”比特，来发现当前路径上的路由器是否需要正在发送的IP数据报进行分片。在11.6节我们观察到如果一个待转发的IP数据报被设置DF比特，而其长度又超过了MTU，那么路由器将返回ICMP不可达的差错。在11.7节我们显示了某版本的tracert程序使用该机制来决定目的地的路径MTU。在11.8节我们看到UDP是怎样处理路径MTU发现的。在本节我们将讨论这个机制是如何按照RFC 1191 [Mogul and Deering 1990]中规定的那样在TCP中进行使用的。

在本书的多种系统（参看序言）中只有Solaris 2.x支持路径MTU发现。

TCP的路径MTU发现按如下方式进行：在连接建立时，TCP使用输出接口或对端声明的MSS中的最小MTU作为起始的报文段大小。路径MTU发现不允许TCP超过对端声明的MSS。如果对端没有指定一个MSS，则默认为536。一个实现也可以按21.9节中讲的那样为每个路由单独保存路径MTU信息。

一旦选定了起始的报文段大小，在该连接上的所有被TCP发送的IP数据报都将被设置DF比特。如果某个中间路由器需要对一个设置了DF标志的数据报进行分片，它就丢弃这个数据报，并产生一个我们在11.6节介绍的ICMP的“不能分片”差错。

如果收到这个ICMP差错，TCP就减少段大小并进行重传。如果路由器产生的是一个较新的该类ICMP差错，则报文段大小被设置为下一跳的MTU减去IP和TCP的首部长度。如果是一个较旧的该类ICMP差错，则必须尝试下一个可能的最小MTU（见图2-5）。当由这个ICMP差错引起的重传发生时，拥塞窗口不需要变化，但要启动慢启动。

由于路由可以动态变化，因此在最后一次减少路径MTU的一段时间以后，可以尝试使用一个较大的值（直到等于对端声明的MSS或输出接口MTU的最小值）。RFC 1191推荐这个时间间隔为10分钟（我们在11.8节看到Solaris 2.2使用一个30分钟的时间间隔）。

在对非本地目的地，默认的MSS通常为536字节，路径MTU发现可以避免在通过MTU小于576（这非常罕见）的中间链路时进行分片。对于本地目的主机，也可以避免在中间链路（如以太网）的MTU小于端点网络（如令牌环网）的情况下进行分片。但为了能使路径MTU更加有用和充分利用MTU大于576的广域网，一个实现必须停止使用为非本地目的制定的536的MTU默认值。MSS的一个较好的选择是输出接口的MTU（当然要减去IP和TCP的首部大小）（在附录E中，我们将看到大多数的实现都允许系统管理员改变这个默认的MSS值）。

24.2.1 一个例子

在某个中间路由器的MTU比任一个端点接口MTU小的情况下，我们能够观察路径MTU发现是如何工作的。图24-1显示了这个例子的拓扑结构。

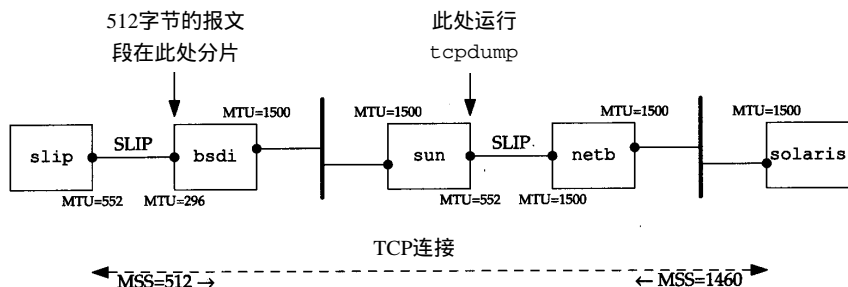


图24-1 路径MTU例子的拓扑结构

我们从主机solaris（支持路径MTU发现机制）到主机slip建立一个连接。这个建立过程与UDP的路径MTU发现（图11-13）中的一个例子相同，但在这里我们已经把slip接口的MTU设置为552，而不是通常的296。这使得slip通告一个512的MSS。但是在bsdi上的SLIP链路上的MTU为296，这就引起超过256的TCP报文段被分片。于是就可以观察在solaris上的路径MTU发现是如何进行处理的。

我们在solaris上运行sock程序并向slip上的丢弃服务器进行一个512字节的写操作：

```
solaris %sock -i -n1 -w512 slip discard
```

图24-2是在主机sun的SLIP接口上收集的tcpdump的输出结果。

```

1  0.0                solaris.33016 > slip.discard: S 1171660288:1171660288(0)
                               win 8760 <mss 1460> (DF)
2  0.101597 (0.1016)  slip.discard > solaris.33016: S 137984001:137984001(0)
                               ack 1171660289 win 4096
                               <mss 512>
3  0.630609 (0.5290)  solaris.33016 > slip.discard: P 1:513(512)
                               ack 1 win 9216 (DF)
4  0.634433 (0.0038)  bsdi > solaris: icmp:
                               slip unreachable - need to frag, mtu = 296 (DF)
5  0.660331 (0.0259)  solaris.33016 > slip.discard: F 513:513(0)
                               ack 1 win 9216 (DF)
6  0.752664 (0.0923)  slip.discard > solaris.33016: . ack 1 win 4096
7  1.110342 (0.3577)  solaris.33016 > slip.discard: P 1:257(256)
                               ack 1 win 9216 (DF)
8  1.439330 (0.3290)  slip.discard > solaris.33016: . ack 257 win 3840
9  1.770154 (0.3308)  solaris.33016 > slip.discard: FP 257:513(256)
                               ack 1 win 9216 (DF)
10 2.095987 (0.3258)  slip.discard > solaris.33016: . ack 514 win 3840
11 2.138193 (0.0422)  slip.discard > solaris.33016: F 1:1(0) ack 514 win 4096
12 2.310103 (0.1719)  solaris.33016 > slip.discard: . ack 2 win 9216 (DF)

```

图24-2 路径MTU发现的tcpdump 输出结果

在第1和第2行的MSS值是我们所期望的。接着我们观察到 solaris发送一个包含512字节的数据和对SYN的确认报文段（第3行）（在习题18.9中可以看到这种把SYN的确认与第一个包含数据的报文段合并的情况）。这就在第4行产生了一个ICMP差错，我们看到路由器 bsdi产生较新的、包含输出接口 MTU的ICMP差错。

看来在这个差错回到 solaris之前，就发送了FIN（第5行）。由于slip从没有收到被路由器bsdi丢弃的512字节的数据，因此并不期望接收这个序号（513），所以在第6行用它期望的序号（1）进行了响应。

在这个时候，ICMP差错返回到了 solaris，solaris用两个256字节的报文段（第7和第9行）重传了512字节的数据。因为在 bsdi后面可能还有具有更小的 MTU的路由器，因此这两个报文段都设置了DF比特。

接着是一个较长的传输过程（持续了大约 15分钟），在最初的512字节变为256字节以后，solaris没有再尝试使用更大的报文段。

24.2.2 大分组还是小分组

常规知识告诉我们较大的分组比较好 [Mogul 1993, 15.2.8节]，因为发送较少的大分组比发送较多的小分组“花费”要少（假定分组的大小不足以引起分片，否则会引起其他方面的问题）。这些减少的花费与网络（分组首部负荷）、路由器（选路的决定）和主机（协议处理和设备中断）等有关。但并非所有的人都同意这种观点 [Bellovin 1993]。

考虑下面的例子。我们通过4个路由器发送8192个字节，每个路由器与一个T1电话线（1544 000b/s）相连。首先我们使用两个4096字节的分组，如图24-3所示。

基本问题在于路由器是存储转发设备。它们通常接收整个输入分组，检验包含IP检验和的IP首部，进行选路判决，然后开始发送输出分组。在这个图中，我们可以假定在理想情况下这些在路由器内部进行的操作不花费时间（水平点状线）。然而，从R1到R4它需要花费4个

单位时间来发送所有的8192字节。每一跳的时间为

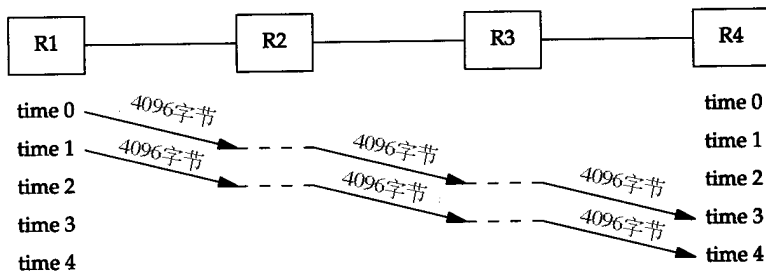


图24-3 通过4个路由器发送两个4096字节的分组

$$\frac{(4096 \text{ 字节} + 40 \text{ 字节}) \times 8 \text{ b/字节}}{1\,544\,000 \text{ b/s}} = 21.4 \text{ ms/跳}$$

(将TCP和IP的首部算为40字节)。发送数据的整个时间为分组个数加上跳数减1，从图中可以看到是4个单位时间，或85.6秒。每个链路空闲2个单位时间，或42.8秒。

图24-4显示了当我们发送16个512字节的分组时所发生的情况。

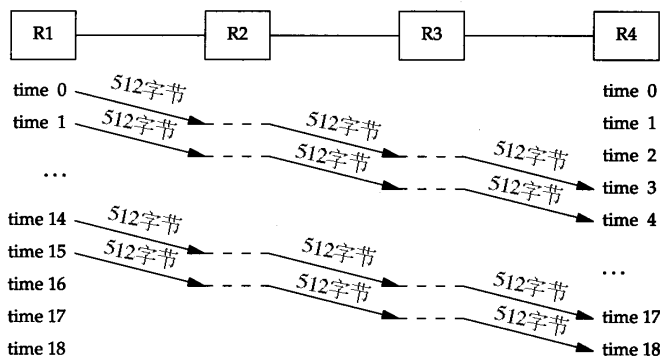


图24-4 通过4个路由器发送16个512字节的分组

这将花费更多的单位时间，但是由于发送的分组较短，因此每个单位时间较小。

$$\frac{(512 \text{ 字节} + 40 \text{ 字节}) \times 8 \text{ b/字节}}{1\,544\,000 \text{ b/s}} = 2.9 \text{ ms/跳}$$

现在总时间为 $(18 \times 2.9) = 52.2 \text{ ms}$ 。每个链路也空闲2个单位的时间，即5.8 ms。

在这个例子中，我们忽略了确认返回所需要的时间、连接建立和终止以及链路可能被其他流量共享等的影响。然而，在 [Bellovin 1993] 中的测量表明，分组并不一定是越大越好。我们需要在更多的网络上对该领域进行更多的研究。

24.3 长肥管道

在20.7节，我们把一个连接的容量表示为

$$\text{capacity (b)} = \text{bandwidth (b/s)} \times \text{round-trip time (s)}$$

并称之为带宽时延乘积。也可称它为两端的管道大小。

当这个乘积变得越来越大时，TCP的某些局限性就会暴露出来。图24-5显示了多种类型的网络的某些数值。

网 络	带宽(b/s)	RTT(ms)	带宽时延乘积 (字节)
以太网	10 000 000	3	3 750
横跨大陆的T1电话线	1 544 000	60	11 580
卫星T1电话线	1 544 000	500	96 500
横跨大陆的T3电话线	45 000 000	60	337 500
横跨大陆的gigabit线路	1 000 000 000	60	7 500 000

图24-5 多种网络的带宽时延乘积

可以看到带宽时延乘积的单位是字节，这是因为我们用这个单位来测量每一端的缓存大小和窗口大小。

具有大的带宽时延乘积的网络被称为长肥网络 (Long Fat Network, 即 LFN, 发音为 “ elefan(t)s ”), 而一个运行在 LFN 上的 TCP 连接被称为长肥管道。回顾图 20-11 和图 20-12, 管道可以被水平拉长 (一个长的 RTT), 或被垂直拉高 (较高的带宽), 或向两个方向拉伸。使用长肥管道会遇到多种问题。

1) TCP 首部中窗口大小为 16 bit, 从而将窗口限制在 65535 个字节内。但是从图 24-5 的最后一列可以看到, 现有的网络需要一个更大的窗口来提供最大的吞吐量。在 24.4 节介绍的窗口扩大选项可以解决这个问题。

2) 在一个长肥网络 LFN 内的分组丢失会使吞吐量急剧减少。如果只有一个报文段丢失, 我们需要利用 21.7 节介绍的快速重传和快速恢复算法来使管道避免耗尽。但是即使使用这些算法, 在一个窗口内发生的多个分组丢失也会典型地使管道耗尽 (如果管道耗尽了, 慢启动会使它渐渐填满, 但这个过程将需要经过多个 RTT)。

在 RFC 1072 [Jacobson and Braden 1988] 中建议使用有选择的确认 (SACK) 来处理在一个窗口发生的多个分组丢失。但是这个功能在 RFC 1323 中被忽略了, 因为作者觉得在把它们纳入 TCP 之前需要先解决一些技术上的问题。

3) 我们在第 21.4 节看到许多 TCP 实现对每个窗口的 RTT 仅进行一次测量。它们并不对每个报文段进行 RTT 测量。在一个长肥网络 LFN 上需要更好的 RTT 测量机制。我们将在 24.5 节介绍时间戳选项, 它允许更多的报文段被计时, 包括重传。

4) TCP 对每个字节数据使用一个 32 bit 无符号的序号来进行标识。如果在网络中有一个被延迟一段时间的报文段, 它所在的连接已被释放, 而一个新的连接在这两个主机之间又建立了, 怎样才能防止这样的报文段再次出现呢? 首先回想起 IP 首部中的 TTL 为每个 IP 段规定了一个生存时间的上限——255 跳或 255 秒, 看哪一个上限先达到。在 18.6 节我们定义了最大的报文段生存时间 (MSL) 作为一个实现的参数来阻止这种情况的发生。推荐的 MSL 的值为 2 分钟 (给出一个 240 秒的 2MSL), 但是我们在 18.6 节看到许多实现使用的 MSL 为 30 秒。

在长肥网络 LFN 上, TCP 的序号会碰到一个不同的问题。由于序号空间是有限的, 在已经传输了 4 294 967 296 个字节以后序号会被重用。如果一个包含序号 N 字节数据的报文段在网络上被延迟并在连接仍然有效时又出现, 会发生什么情况呢? 这仅仅是一个相同序号 N 在 MSL 期间是否被重用的问题, 也就是说, 网络是否足够快以至于在不到一个 MSL 的时候序号就发生了回绕。在一个以太网上要发送如此多的数据通常需要 60 分钟左右, 因此不会发生这种情况。但是在带宽增加时, 这个时间将会减少: 一个 T3 的电话线 (45 Mb/s) 在 12 分钟内会发生回绕, FDDI (100 Mb/s) 为 5 分钟, 而一个千兆比网络 (1000 Mb/s) 则为 34 秒。这时问题不再是带宽时延乘积, 而在于带宽本身。

在24.6节,我们将介绍一种对付这种情况的办法:使用 TCP的时间戳选项的 PAWS (Protection Against Wrapped Sequence numbers)算法(保护回绕的序号)。

4.4BSD包含了我们将要在下面介绍的所有选项和算法:窗口扩大选项、时间戳选项和保护回绕的序号。许多供应商也正在开始支持这些选项。

千兆比网络

当网络的速率达到千兆比的时候,情况就会发生变化。[Partridge 1994]详细介绍了千兆比网络。在这里我们看一下在时延和带宽之间的差别 [Kleinrock 1992]。

考虑通过美国发送一个 100 万字节的文件的情况,假定时延为 30 ms。图 24-6 显示了两种情况:上图显示了使用一个 T1 电话线 (1 544 000 b/s) 的情况,而下图则是使用一个 1 Gb/s 网络的情况。 x 轴显示的是时间,发送方在图的左侧,而接收方则在图的右侧, y 轴为网络容量。两幅图中的阴影区域表示发送的 100 万字节。

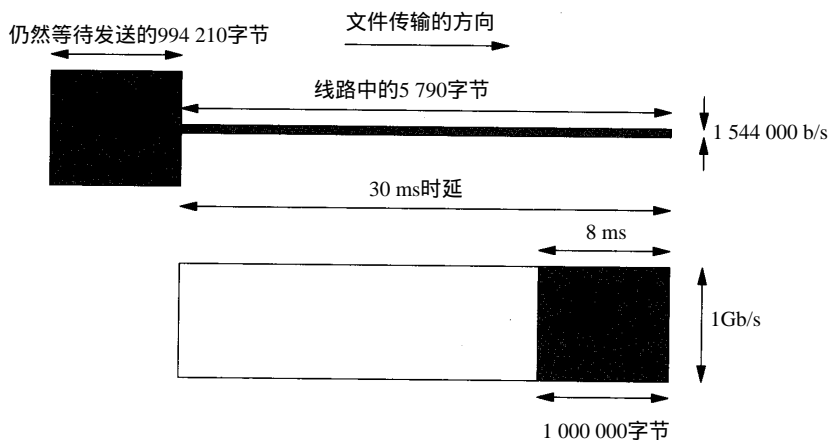


图24-6 以30 ms的延时通过网络发送100万字节的文件

图24-6显示了30 ms后这两个网络的状态。经过30 ms(延时)以后数据的第1个比特都已到达对端。但对T1网络而言,由于管道容量仅为5 790字节,因此发送方仍然有994 210个字节等待发送。而千兆比网络的容量则为3 750 000字节,因此,整个文件仅使用了25%左右的带宽,此时文件的最后一个比特已经到达第1个字节后8 ms处。

经过T1网络传输文件的总时间为5.211秒。如果增加更多的带宽,使用一个T3网络(45 000 000 b/s),则总时间减少到0.208秒。增加约29倍的带宽可以将总时间减小到约5分之一。

使用千兆比网络传输文件的总时间为0.038秒:30 ms的时延加上8 ms的真正传输文件的时间。假定能够将带宽增加为2000 Mb/s,我们只能够将总时间减小为0.304 ms:同样30 ms的时延和4ms的真正传输时间。现在使带宽加倍仅能够将时间减少约10%。在千兆比速率下,时延限制占据了主要地位,而带宽不再成为限制。

时延主要是由光速引起的,而且不能够被减小(除非爱因斯坦是错误的)。当我们考虑到分组需要建立和终止一个连接时,这个固定时延起的作用就更糟糕了。千兆比网络会引起一些需要不同看待的连网观点。

24.4 窗口扩大选项

窗口扩大选项使TCP的窗口定义从16 bit增加为32 bit。这并不是通过修改TCP首部来实现的, TCP首部仍然使用16 bit, 而是通过定义一个选项实现对16 bit的扩大操作 (scaling operation)来完成的。于是TCP在内部将实际的窗口大小维持为32 bit的值。

在图18-20可以看到关于这个选项的例子。一个字节的移位计数器取值为0 (没有扩大窗口的操作) 和14。这个最大值14表示窗口大小为1 073 725 440字节 (65535×2^{14})。

这个选项只能够出现在一个SYN报文段中, 因此当连接建立起来后, 在每个方向的扩大因子是固定的。为了使用窗口扩大, 两端必须在它们的SYN报文段中发送这个选项。主动建立连接的一方在其SYN中发送这个选项, 但是被动建立连接的一方只能够在收到带有这个选项的SYN之后才可以发送这个选项。每个方向上的扩大因子可以不同。

如果主动连接的一方发送一个非零的扩大因子, 但是没有从另一端收到一个窗口扩大选项, 它就将发送和接收的移位计数器置为0。这就允许较新的系统能够与较旧的、不理解新选项的系统进行互操作。

Host Requirements RFC要求TCP接受在任何报文段中的一个选项 (只有前面定义的一个选项, 即最大报文段大小, 仅在SYN报文段中出现)。它还进一步要求TCP忽略任何它不理解的选项。这就使事情变得容易, 因为所有新的选项都有一个长度字段 (图8-20)。

假定我们正在使用窗口扩大选项, 发送移位记数为 S , 而接收移位记数则为 R 。于是我们从另一端收到的每一个16 bit的通告窗口将被左移 R 位以获得实际的通告窗口大小。每次当我们向对方发送一个窗口通告的时候, 我们将实际的32 bit窗口大小右移 S 比特, 然后用它来替换TCP首部中的16 bit的值。

TCP根据接收缓存的大小自动选择移位计数。这个大小是由系统设置的, 但是通常向应用进程提供了修改途径 (我们在20.4节中讨论了这个缓存)。

一个例子

如果在4.4BSD的主机vangogh.cs.berkeley.edu上使用sock程序来初始化一个连接, 我们可以观察到它的TCP计算窗口扩大因子的情况。下面的交互输出显示的是两个连续运行的程序, 第1个指定接收缓存为128 000字节, 而第2个的缓存则为220 000字节。

```
vangogh % sock -v -R128000 bsdi.tuc.noao.edu echo
SO_RCVBUF = 128000
connected on 128.32.130.2.4107 to 140.252.13.35.7
TCP_MAXSEG = 512
hello, world
hello, world
^D
```

我们键入这一行
此处是它的回显
键入文件结束字符以终止

```
vangogh % sock -v -R220000 bsdi.tuc.noao.edu echo
SO_RCVBUF = 220000
connected on 128.32.130.2.4108 to 140.252.13.35.7
TCP_MAXSEG = 512
bye, bye
bye, bye
^D
```

我们键入这一行
此处是它的回显
键入文件结束字符以终止

图24-7显示了这两个连接的tcpdump输出结果 (去掉了第2个连接的最后8行, 因为没有

什么新内容)。

```

1   0.0                vangogh.4107 > bsdi.echo: S 462402561:462402561(0)
                                win 65535
                                <mss 512,nop,wscale 1,nop,nop,timestamp 995351 0>
2   0.003078 ( 0.0031) bsdi.echo > vangogh.4107: S 177032705:177032705(0)
                                ack 462402562 win 4096 <mss 512>
3   0.300255 ( 0.2972) vangogh.4107 > bsdi.echo: . ack 1 win 65535
4   16.920087 (16.6198) vangogh.4107 > bsdi.echo: P 1:14(13) ack 1 win 65535
5   16.923063 ( 0.0030) bsdi.echo > vangogh.4107: P 1:14(13) ack 14 win 4096
6   17.220114 ( 0.2971) vangogh.4107 > bsdi.echo: . ack 14 win 65535
7   26.640335 ( 9.4202) vangogh.4107 > bsdi.echo: F 14:14(0) ack 14 win 65535
8   26.642688 ( 0.0024) bsdi.echo > vangogh.4107: . ack 15 win 4096
9   26.643964 ( 0.0013) bsdi.echo > vangogh.4107: F 14:14(0) ack 15 win 4096
10  26.880274 ( 0.2363) vangogh.4107 > bsdi.echo: . ack 15 win 65535

11  44.400239 (17.5200) vangogh.4108 > bsdi.echo: S 468226561:468226561(0)
                                win 65535
                                <mss 512,nop,wscale 2,nop,nop,timestamp 995440 0>
12  44.403358 ( 0.0031) bsdi.echo > vangogh.4108: S 182792705:182792705(0)
                                ack 468226562 win 4096 <mss 512>
13  44.700027 ( 0.2967) vangogh.4108 > bsdi.echo: . ack 1 win 65535

```

该连接的其余部分被删除

图24-7 窗口扩大选项的例子

在第1行, vangogh通告一个65535的窗口, 并通过设置移位计数为1来指明窗口扩大选项。这个通告的窗口是比接收窗口(128 000)还小的一个最大可能取值, 因为在一个SYN报文段中的窗口字段从不进行扩大运算。

扩大因子为1表示vangogh发送窗口通告一直到131 070 (65535×2^1)。这将调节我们的接收缓存的大小(12 8000)。因为bsdi在它的SYN(第2行)中没有发送窗口扩大选项, 因此这个选项没有被使用。注意到vangogh在随后的连接阶段继续使用最大可能的窗口(65535)。

对于第2个连接vangogh请求的移位计数为2, 表明它希望发送窗口通告一直为262 140 (65535×2^2), 这比我们的接收缓存(220 000)大。

24.5 时间戳选项

时间戳选项使发送方在每个报文段中放置一个时间戳值。接收方在确认中返回这个数值, 从而允许发送方为每一个收到的ACK计算RTT(我们必须说“每一个收到的ACK”而不是“每一个报文段”, 是因为TCP通常用一个ACK来确认多个报文段)。我们提到过目前许多实现为每一个窗口只计算一个RTT, 对于包含8个报文段的窗口而言这是正确的。然而, 较大的窗口大小则需要进行更好的RTT计算。

RFC 1323的3.1节给出了需要为较大窗口进行更好的RTT计算的信号处理的理由。通常RTT通过对一个数据信号(包含数据的报文段)以较低的频率(每个窗口一次)进行采样来进行计算, 这就将别名引入了被估计的RTT中。当每个窗口中有8个报文段时, 采样速率为数据率的1/8, 这还是可以忍受的。但是如果每个窗口中有100个报文段时, 采样速率则为数据速率的1/100, 这将导致被估计的RTT不精确, 从而引起不必要的重传。如果一个报文段被丢失, 则会使情况变得更糟。

图18-20显示了时间戳选项的格式。发送方在第1个字段中放置一个32 bit的值, 接收方在应答字段中回显这个数值。包含这个选项的TCP首部长度将从正常的20字节增加为32字节。

时间戳是一个单调递增的值。由于接收方只需要回显收到的内容, 因此不需要关注时间戳单元是什么。这个选项不需要在两个主机之间进行任何形式的时钟同步。RFC 1323推荐在1毫秒和1秒之间将时间戳的值加1。

4.4BSD在启动时将时间戳始终设置为0, 然后每隔500 ms将时间戳时钟加1。

在图24-7中, 如果观察在报文段1和报文段11的时间戳, 它们之间的差(89个单元)对应于每个单元500 ms的规定, 因为实际时间差为44.4秒。

在连接建立阶段, 对这个选项的规定与前一节讲的窗口扩大选项类似。主动发起连接的一方在它的SYN中指定选项。只有在它从另一方的SYN中收到了这个选项之后, 该选项才会在以后的报文段中进行设置。

我们已经看到接收方TCP不需要对每个包含数据的报文段进行确认, 许多实现每两个报文段发送一个ACK。如果接收方发送一个确认了两个报文段的ACK, 那么哪一个收到的时间戳应当放入回显应答字段中来发回去呢?

为了减少任一端所维持的状态数量, 对于每个连接只保持一个时间戳的数值。选择何时更新这个数值的算法非常简单:

1) TCP跟踪下一个ACK中将要发送的时间戳的值(一个名为`tsrecent`的变量)以及最后发送的ACK中的确认序号(一个名为`lastack`的变量)。这个序号就是接收方期望的序号。

2) 当一个包含有字节号`lastack`的报文段到达时, 则该报文段中的时间戳被保存在`tsrecent`中。

3) 无论何时发送一个时间戳选项, `tsrecent`就作为时间戳回显应答字段被发送, 而序号字段被保存在`lastack`中。

这个算法能够处理下面两种情况:

1) 如果ACK被接收方迟延, 则作为回显值的时间戳值应该对应于最早被确认的报文段。例如, 如果两个包含1~1024和1025~2048字节的报文段到达, 每一个都带有一个时间戳选项, 接收方产生一个ACK 2049来对它们进行确认。此时, ACK中的时间戳应该是包含字节1~1024的第1个报文段中的时间戳。这种处理是正确的, 因为发送方在进行重传超时时间的计算时, 必须将迟延的ACK也考虑在内。

2) 如果一个收到的报文段虽然在窗口范围内但同时又是失序, 这就表明前面的报文段已经丢失。当那个丢失的报文段到达时, 它的时间戳(而不是失序的报文段的时间戳)将被回显。例如, 假定有3个各包含1024字节数据的报文段, 按如下顺序接收: 包含字节1~1024的报文段1, 包含字节2049~4072的报文段3和包含字节1025~2048的报文段2。返回的ACK应该是带有报文段1的时间戳的ACK 1025(一个正常的所期望的对数据的ACK)、带有报文段1的时间戳的ACK 1025(一个重复的、响应位于窗口内但却是失序的报文段的ACK), 然后是带有报文段2的时间戳的ACK 3073(不是报文段3中的较后的时间戳)。这与当报文段丢失时的对RTT估计过高具有同样的效果, 但这比估计过低要好些。而且, 如果最后的ACK含有来自报文段3的时间戳, 它可以包括重复的ACK返回和报文段2被重传所需要的时间, 或者可以包括发送方的报文段2的重传超时定时器到期的时间。无论在哪一种情况下, 回显报文段3的时间戳将引起发送方的RTT计算出现偏差。

尽管时间戳选项能够更好地计算RTT, 它还还为发送方提供了一种方法, 以避免接收到旧的报文段, 并认为它们是现在的数据的一部分。下一节将对此进行描述。

24.6 PAWS : 防止回绕的序号

考虑一个使用窗口扩大选项的 TCP 连接, 其最大可能的窗口大小为 1 千兆字节 (2^{30}) (最大的窗口是 65535×2^{14} , 而不是 $2^{16} \times 2^{14}$, 但只比这个数值小一点点, 并不影响这里的讨论)。还假定使用了时间戳选项, 并且由发送方指定的时间戳对每个将要发送的窗口加 1 (这是保守的方法。通常时间戳比这种方式增加得快)。图 24-8 显示了在传输 6 千兆字节的数据时, 在两个主机之间可能的数据流。为了避免使用许多 10 位的数字, 我们使用 G 来表示 1 073 741 824 的倍数。我们还使用了 tcpdump 的记号, 即用 $J:K$ 来表示通过了 J 字节的数据, 且包括字节 $K-1$ 。

时间	发送字节	发送序号	发送时间戳	接 收
A	0G:1G	0G:1G	1	正确
B	1G:2G	1G:2G	2	正确, 但有一个段丢失并重发
C	2G:3G	2G:3G	3	正确
D	3G:4G	3G:4G	4	正确
E	4G:5G	0G:1G	5	正确
F	5G:6G	1G:2G	6	正确, 但重发的段又出现了

图24-8 在6个1千兆字节的窗口中传输6千兆字节的数据

32 bit 的序号在时间 D 和时间 E 之间发生了回绕。假定一个报文段在时间 B 丢失并被重传。还假定这个丢失的报文段在时间 E 重新出现。

这假定了在报文段丢失和重新出现之间的时间差小于 MSL, 否则这个报文段在它的 TTL 到期时会被某个路由器丢弃。正如我们前面提到的, 这种情况只有在高速连接上才会发生, 此时旧的报文段重新出现, 并带有当前要传输的序号。

我们还可以从图 24-8 中观察到使用时间戳可以避免这种情况。接收方将时间戳视为序列号的一个 32 bit 的扩展。由于在时间 E 重新出现的报文段的时间戳为 2, 这比最近有效的时间戳小 (5 或 6), 因此 PAWS 算法将其丢弃。

PAWS 算法不需要在发送方和接收方之间进行任何形式的时间同步。接收方所需要的就是时间戳的值应该单调递增, 并且每个窗口至少增加 1。

24.7 T/TCP : 为事务用的 TCP 扩展

TCP 提供的是一种虚电路方式的运输服务。一个连接的生存时间包括三个不同的阶段: 建立、数据传输和终止。这种虚电路服务非常适合诸如远程注册和文件传输之类的应用。

但是, 还有出现其他的应用进程被设计成使用事务服务。一个事务 (transaction) 就是符合下面这些特征的一个客户请求及其随后的服务器响应。

1) 应该避免连接建立和连接终止的开销, 在可能的时候, 发送一个请求分组并接收一个应答分组。

2) 等待时间应当减少到等于 RTT 与 SPT 之和。其中 RTT (Round-Trip Time) 为往返时间, 而 SPT (Server Processing Time) 则是服务器处理请求的时间。

3) 服务器应当能够检测出重复的请求, 并且当收到一个重复的请求时不重新处理事务 (避免重新处理意味着服务器不必再次处理请求, 而是返回保存的、与该请求对应的应答)。

我们已经看到的一个使用这种类型服务的应用就是域名服务 (第 14 章), 尽管 DNS 与服务器重新处理重复的请求无关。

如今一个应用程序设计人员面对的一种选择是使用 TCP 还是 UDP。TCP 提供了过多的事务特征, 而 UDP 提供的则不够。通常应用程序使用 UDP 来构造(避免 TCP 连接的开销), 而许多需要的特征(如动态超时和重传、拥塞避免等)被放置在应用层, 一遍又一遍的重新设计和实现。

一个较好的解决方法是提供一个能够提供足够多的事务处理功能的运输层。我们在本节所介绍的事务协议被称为 T/TCP。我们从它的定义, 即 RFC 1379 [Braden 1992b] 和 [Braden 1992c], 开始介绍。

大多数的 TCP 需要使用 7 个报文段来打开和关闭一个连接(见图 18-13)。现在增加三个报文段: 一个对应于请求, 一个对应于应答和对请求的确认, 第三个对应于对应答的确认。如果额外的控制比特被追加到报文段上——也就是, 第 1 个报文段带有 SYN、客户请求和一个 FIN——客户仍然能够看到一个 2 倍的 RTT 与 SPT 之和的最小开销(与数据一起发送一个 SYN 和 FIN 是合法的; 当前的 TCP 是否能够正确处理它们是另外一个问题)。

另一个与 TCP 有关的问题是 TIME_WAIT 状态和它需要的 2MSL 的等待时间。正如在习题 18.14 中看到的, 这使两个主机之间的事务率降低到每秒 268 个。

TCP 为处理事务而需要进行的两个改动是避免三次握手和缩短 WAIT_TIME 状态。T/TCP 通过使用加速打开来避免三次握手:

- 1) 它为打开的连接指定一个 32 bit 的连接计数 CC (Connection Count), 无论主动打开还是被动打开。一个主机的 CC 值从一个全局计数器中获得, 该计数器每次被使用时加 1。
- 2) 在两个使用 T/TCP 的主机之间的每一个报文段都包括一个新的 TCP 选项 CC。这个选项的长度为 6 个字节, 包含发送方在该连接上的 32 bit 的 CC 值。
- 3) 一个主机维持一个缓存, 该缓存保留每个主机上一次的 CC 值, 这些值从来自这个主机的一个可接受的 SYN 报文段中获得。
- 4) 当在一个开始的 SYN 中收到一个 CC 选项的时候, 接收方比较收到的值与为该发送方缓存的 CC 值。如果接收到的 CC 比缓存的大, 则该 SYN 是新的, 报文段中的任何数据被传递给接收应用进程(服务器)。这个连接被称为半同步。
如果接收的 CC 比缓存的小, 或者接收主机上没有对应这个客户的缓存 CC, 则执行正常的 TCP 三次握手过程。
- 5) 为响应一个开始的 SYN, 带有 SYN 和 ACK 的报文段在另一个被称为 CCECHO 的选项中回显所接收到的 CC 值。
- 6) 在一个非 SYN 报文段中的 CC 值检测和拒绝来自同一个连接的前一个替身的任何重复的报文段。

这种“加速打开”避免了使用三次握手的要求, 除非客户或者服务器已经崩溃并重新启动。这样做的代价是服务器必须记住从每个客户接收的最近的 CC 值。

基于在两个主机之间测量 RTT 来动态计算 TIME_WAIT 的延时, 可以缩短 TIME_WAIT 状态。TIME_WAIT 时延被设置为 8 倍的重传超时值 RTO (见 21.3 节)。

通过使用这些特征, 最小的事务序列是交换三个报文段:

- 1) 由一个主动打开引起的客户到服务器: 客户的 SYN、客户的数据(请求)、客户的 FIN 以及客户的 CC。当被动的服务器 TCP 接收到这个报文段的时候, 如果客户的 CC 比这个客户缓存的 CC 要大, 则客户的数据被传送给服务器应用程序进行处理。
- 2) 服务器到客户: 服务器的 SYN、服务器的数据(应答)、服务器的 FIN、对客户的 FIN

的ACK、服务器的CC以及客户的CC的CCECHO。由于TCP的确认是累积的，这个对客户的FIN的ACK也对客户的SYN、数据及FIN进行了确认。

当客户TCP接收到这个报文段，就将其传送给客户应用进程。

3) 客户到服务器：对服务器的FIN的ACK，它也确认了服务器的SYN、数据和FIN。

客户对它的请求的响应时间为RTT与SPT的和。

在参考资料中有许多关于实现这个TCP选项的很好的地方。我们在这里将它们归纳如下：

- 服务器的SYN和ACK（第2个报文段）必须被迟延，从而允许应答与它一起捎带发送（通常对SYN的ACK是不迟延的）。但它也不能迟延得太多，否则客户将超时并引起重传。
- 请求可以需要多个报文段，但是服务器必须对它们可能失序达到的情况进行处理（通常当数据在SYN之前到达时，该数据被丢弃并产生一个复位。通过使用T/TCP，这些失序的数据将放入队列中处理）。
- API必须使服务器进程用一个单一的操作来发送数据和关闭连接，从而允许第二个报文段中的FIN与应答一起捎带发送（通常应用进程先写应答，从而引起发送一个数据报文段，然后关闭连接，引起发送FIN）。
- 在收到来自服务器的MSS通告之前，客户在第1个报文段中正在发送数据。为避免限制客户的MSS为536，一个给定主机的MSS应该与它的CC值一起缓存。
- 客户在没有接收到来自服务器的窗口通告之前也可以向服务器发送数据。T/TCP建议默认的窗口为4096，并且也为服务器缓存拥塞门限。
- 使用最小3个报文段交换，在每个方向上只能计算一个RTT。加上包括了服务器处理时间的客户测量RTT。这意味着被平滑的RTT及其方差的值也必须为服务器缓存起来，这与我们在21.9节描述的类似。

T/TCP的特征中吸引人的地方在于它对现有协议进行了最小的修改，同时又兼容了现有的实现。它还利用了TCP中现有的工程特征（动态超时和重传、拥塞避免等），而不是迫使应用进程来处理这些问题。

一个可作为替换的事务协议是通用报文事务协议 VMTP (Versatile Message Transaction Protocol)，该协议在RFC 1045 [Cheriton 1988]中进行了描述。与T/TCP是现有协议的一个小的扩充不同，VMTP是使用IP的一个完整的运输层。VMTP处理差错检测、重传和重复压缩。它还支持多播通信。

24.8 TCP的性能

在80年代中期出版的数值显示出TCP在一个以太网上的吞吐量在每秒100 000~200 000字节之间 ([Stevens 1990]的17.5节给出了参考文献)。从那时起事情已经发生了许多改变。现在通常使用的硬件（工作站和更快的个人电脑）每秒可以传输800 000字节或者更快。

在10 Mb/s的以太网上计算我们能够观察到的理论上的TCP最大吞吐量是一件值得做的练习 [Warnock

字 段	数据 (字节)	ACK (字节)
以太网前导	8	8
以太网目的地址	6	6
以太网源地址	6	6
以太网类型字段	2	2
IP首部	20	20
TCP首部	20	20
用户数据	1460	0
填充字符	0	6
以太网CRC检验	4	4
分组间隙(9.6ms)	12	12
总计	1538	84

图24-9 计算以太网理论上最大吞吐量的字段大小

1991]。我们可以在图 24-9 中看到这个计算的基础。这个图显示了满长度的数据报文段和一个 ACK 交换的全部的字节。

我们必须计及所有的开销：前同步码、加到确认上的填充字节、循环冗余检验 CRC 以及分组之间的最小间隔（9.6ms，相当在 10 Mb/s 速率下的 12 个字节）。

首先假定发送方传输两个背对背、满长度的数据报文段，然后接收方为这两个报文段发送一个 ACK。于是最大的吞吐量（用户数据）为：

$$\text{throughput} = \frac{2 \times 1460 \text{ B}}{22 \times 1538 \text{ B} + 84 \text{ B}} \times \frac{10\,000\,000 \text{ b/s}}{8 \text{ b/B}} = 1\,555\,063 \text{ B/S}$$

如果 TCP 窗口开到它的最大值（65535，不使用窗口扩大选项），这就允许一个窗口容纳 44 个 1460 字节的报文段。如果接收方每个报文段发送一个 ACK，则计算变为：

$$\text{throughput} = \frac{22 \times 1460 \text{ B}}{22 \times 1538 \text{ B} + 84 \text{ B}} \times \frac{10\,000\,000 \text{ b/s}}{8 \text{ b/B}} = 1\,183\,667 \text{ B/S}$$

这就是理论上的限制，并做出某些假定：接收方发送的一个 ACK 没有和发送方的报文段之一在以太网上发生冲突；发送方可按以太网的最小间隔时间来发送两个报文段；接收方可以在最小的以太网间隔时间内产生一个 ACK。不论在这些数字上多么乐观，[Warnock 1991] 在一个以太网上使用标准的多用户工作站（即使是快的工作站）测量到了一个连续的 1 075 000 字节/秒的速率，这个值在理论值的 90% 之内。

当移到更快的网络上时，如 FDDI（100 Mb/s），[Schryver 1993] 指出三个商业厂家已经演示了在 FDDI 上的 TCP 在 80 Mb/s~90 Mb/s 之间。即使在有更多带宽的环境下，[Borman 1992] 报告说两个 Gray Y-MP 计算机在一个 800 Mb/s 的 HIPPI 通道上最大值为 781 Mb/s，而运行在一个 Gray Y-MP 上的使用环回接口的两个进程间的速率为 907 Mb/s。

下面这些实际限制适用于任何的实际情况 [Borman 1991]。

- 1) 不能比最慢的链路运行得更快。
- 2) 不能比最慢的机器的内存运行得更快。这假定实现是只使用一遍数据。如果不是这样（也就是说，实现使用一遍数据是将它从用户空间复制到内核中，而使用另一遍数据是计算 TCP 的检验和），那么将运行得更慢。[Dalton et al. 1993] 描述了将数据复制数目减少从而使一个标准伯克利源程序的性能得到改进。[Partridge and Pink 1993] 将类似的“复制与检验和”的改变与其他性能改进措施一道应用于 UDP，从而将 UDP 的性能提高了约 30%。
- 3) 不能够比由接收方提供的窗口大小除以往返时间所得结果运行得更快（这就是带宽时延乘积公式，使用窗口大小作为带宽时延乘积，并解出带宽）。如果使用 24.4 节的最大窗口扩大因子 14，则窗口大小为 1.073 千兆字节，所以这除以 RTT 的结果就是带宽的极限。

所有这些数字的重要意义就是 TCP 的最高运行速率的真正上限是由 TCP 的窗口大小和光速决定的。正如 [Partridge and Pink 1993] 中计算的那样，许多协议性能问题在于实现中的缺陷而不是协议所固有的一些限制。

24.9 小结

本章已经讨论了五个新的 TCP 特征：路径 MTU 发现、窗口扩大选项、时间戳选项、序号回绕保护以及使用改进的 TCP 事务处理。我们观察到中间三个特征是为在长肥管道——具有大的带宽时延乘积的网络——上优化性能所需要的。

路径MTU发现在MTU较大时，对于非本地连接，允许 TCP使用比默认的 536大的窗口。这样可以提高性能。

窗口扩大选项使最大的 TCP窗口从65535增加到1千兆字节以上。时间戳选项允许多个报文段被精确计时，并允许接收方提供序号回绕保护（PAWS）。这对于高速连接是必须的。这些新的TCP选项在连接时进行协商，并被不理解它们的旧系统忽略，从而允许较新的系统与旧的系统进行交互。

为事务用的TCP扩展，即T/TCP，允许一个客户/服务器的请求-应答序列在通常的情况下只使用三个报文段来完成。它避免使用三次握手，并缩短了 TIME_WAIT状态，其方法是为每个主机高速缓存少量的信息，这些信息曾用来建立过一个连接。它还在包含数据报文段中使用SYN和FIN标志。

由于还有许多关于TCP能够运行多快的不精确的传闻，因此我们以对 TCP性能的分析来结束本章。对于一个使用本章介绍的较新特征、协调得非常好的实现而言，TCP的性能仅受最大的1千兆字节窗口和光速（也就是往返时间）的限制。

习题

- 24.1 当一个系统发送一个开始的SYN报文段，其窗口扩大因子为0，这是什么含义？
- 24.2 如果在图24-7中的主机bsdi支持窗口扩大选项，则来自 vangogh的报文段3的16 bit窗口大小字段中的期望值是多少？类似地，如果在该图的第 2个连接中也使用这个选项，那么报文段13中的窗口通告应该是多少？
- 24.3 与在建立连接时的固定窗口扩大因子不同，已经定义过的窗口扩大因子能否在扩大因子变化时也出现呢？
- 24.4 假定MSL为2分钟，那么在什么速率下序号回绕会成为一个问题呢？
- 24.5 PAWS被定义为只在一个单独的连接中进行。为了使 TCP将PAWS来替换2MSL等待(即TIME_WAIT状态)，需要进行什么改动？
- 24.6 在24.4节最后的例子中，为什么 sock程序在紧接着（具有IP地址和端口）后面的一行之前，将接收缓存的大小来输出呢？
- 24.7 假定MSS为1024，重新计算24.8节中的吞吐量。
- 24.8 时间戳选项是如何影响Karn算法（见21.3节）的？
- 24.9 如果主动建立连接的TCP发送带有SYN标志的报文段（没有使用我们在24.7节介绍的扩展），那么接收TCP应该怎样处理这些数据呢？
- 24.10 在24.7节我们提到如果没有使用T/TCP扩展，即使主动开启方发送带有FIN的数据，客户在接收服务器的响应的时延仍然是两倍的RTT再加上SPT。给出符合这种情况的报文段。
- 24.11 假定支持T/TCP，且源自伯克利系统的最小RTO为0.5秒，重做习题18.14。
- 24.12 如果我们实现了T/TCP，并测量两个主机之间的事务时间，那么可以通过比较什么指标来确定它的有效性？