

第14章 在HTTP服务器上找到的分组

14.1 概述

本章我们将通过分析一个繁忙的 HTTP 服务器上所处理的分组，从另外的角度来分析 HTTP 协议，同时还将对 Internet 协议族中的一些特性进行一般性的分析。这样我们就能把卷 1 和卷 2 中描述的 TCP/IP 协议的一些特性与现实世界中的联系起来。从本章也可看到，TCP 协议的行为和实现的变化很多，有时甚至明显不合理。本章有很多主题，我们把它们近似地按照 TCP 连接动作的顺序来安排：连接建立、数据传输和连接终止。

我们是从一个商业的 Internet 服务提供商的系统上收集数据。这个系统为 22 个组织提供 HTTP 服务，同时运行 NCSA httpd 服务器的 22 个副本(我们将在下一节中讨论运行多个服务器程序)。该系统的 CPU 是 Intel 奔腾处理器，运行的操作系统是 BSD / OS V1.1。

我们收集了三种数据：

- 1) 在连续的 5 天当中每小时运行一次 netstat 程序，运行该程序时带 -s 选项，用来收集 Internet 协议维护的所有计数器。这些计数器在卷 2 中我们都有介绍，如第 164 页(IP)、第 639 页(TCP)等。
- 2) 在这 5 天当中 Tcpdump 程序(见卷 1 附录 A)24 小时运行，记录所有发出的和从 80 端口来的带有 SYN、FIN 或 RST 标志的 TCP 分组。这样，我们可以详细考查 TCP 连接的统计结果。在这期间共收集到 686 755 个符合上述条件的分组，它们分属于 147 103 次 TCP 连接尝试。
- 3) 在 5 天的测量中，做了一次为期 2.5 小时的统计，记录所有发出的和从 80 端口来的 TCP 分组。因为我们可以对除了带有 SYN、FIN 或 RST 标记以外的更多的分组进行检查，所以我们可以对少数特殊情况进行更详细的分析。在这次统计中共记录了 1 039 235 个分组，平均每秒 115 个。

收集 24 小时内的 SYS / FIN / RST 分组的命令是：

```
$ tcpdump -p -w data.out 'tcp and port 80 and tcp[13:1] & 0x7 != 0'
```

-p 标志没有把网络接口置于混合模式(promiscuous)，所以只有运行 Tcpdump 程序的主机发出或接收的分组才可能被捕捉，这也正是我们所需要的。这样减少了从本地网络中收集的数据量，同时也使 Tcpdump 程序减少了分组的丢失。

这个标志没有保证非混合模式。也有人可以将网络接口设为混合模式。

在这个主机上多次长时间运行 Tcpdump，报告的分组丢失情况为：每 16 000 个丢失 1 个至每 22 000 个丢失 1 个之间。

-w 标志将收集结果以二进制格式存入文件，而不是以文本方式在终端上输出。这个输出文件的二进制数据随后可以用 -r 标志转换成我们所期望的文本文件。

只有发出的或从 80 端口来的 TCP 分组才被收集。此外还要求：从 TCP 分组首部开始算，取第 13 字节与数字 7 进行逻辑与运算，结果必须为 0。这是用来测试 SYN、FIN 或 RST 标志是否被

置位(见卷1第171页)。通过收集满足上述条件的分组,然后分析 SYN和FIN上的TCP序号,我们能得到连接的每个方向上,传输数据的字节数。Vern Paxson的tcpdump-reduce软件就是采用了这种简化方式(<http://town.hall.org/Archive/pub/ITA>)。

我们给出的第一张图(图14-1)是5天中尝试连接的总数,包括主动和被动建立的连接。图中表示的是两个TCP计数器:tcps_connattempt和tcps_accepts的时间曲线,摘自卷2第639页。当为了打开连接而发送一个SYN分组时,第一个计数器加一;当在侦听端口收到一个SYN分组时,第二个计数器加一。这些计数器对主机上的所有TCP连接进行计数,而不只是HTTP连接。我们期望系统收到的连接请求比它发出的连接请求要多,因为系统主要提供Web服务(当然系统也用作其他用途,但主要的TCP/IP流量是由HTTP分组组成)。

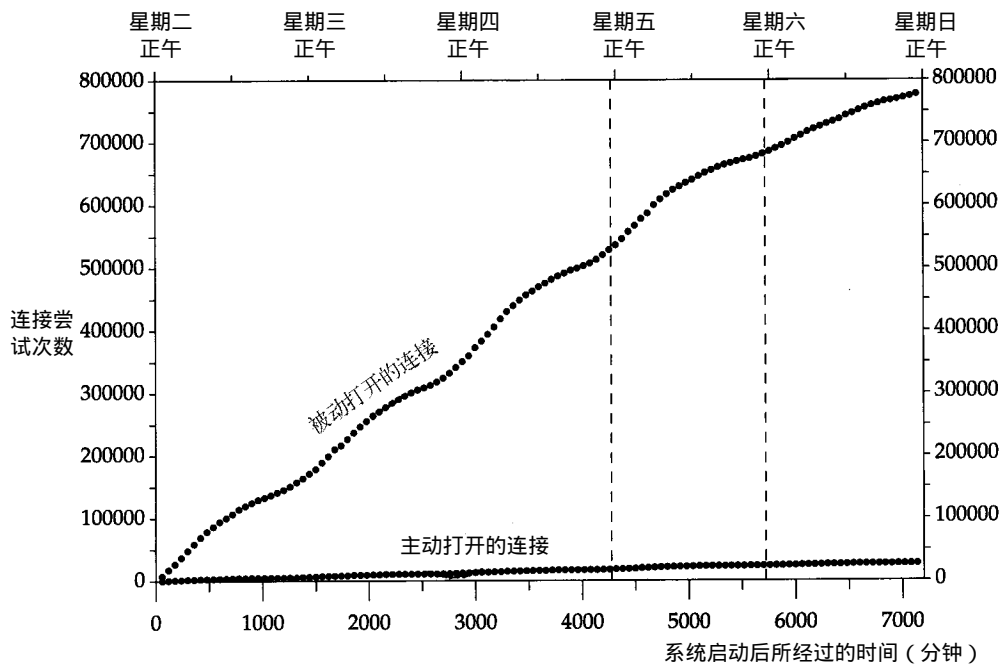


图14-1 主动与被动连接尝试次数累计

图中星期五正午附近和星期六正午附近的虚线描绘了一个24小时周期,在这24小时中对SYN/FIN/RST分组进行了跟踪、收集。注意被动连接尝试的次数曲线,它的斜率像我们所预期的那样在正午后一直到午夜前都比较大。我们也可看出,从星期五的午夜开始到周末这段时间,曲线的斜率一直在减小。我们绘出每小时被动连接尝试次数的曲线,如图14-2所示,从中很容易看出每天的周期性规律。

“繁忙”服务器的定义是什么?我们进行分析的系统每天收到超过150 000个TCP连接请求,这相当于平均每秒1.74个连接请求。[Braun and Claffy 1994]提供了NCSA服务器的详细情况:在1994年9月,平均每天有360 000个TCP连接请求(这个数据每6~8个星期翻一番)。[Mogul 1995b]中描述了两个被作者称为“相对繁忙”的服务器,其中一个每天处理一百万个连接请求,而另一个则是在近3个月时间内平均每天收到40 000个连接请求。1995年6月21日的《华尔街》杂志列出了最繁忙的10个Web服务器,统计了从1995年5月1日至7日之间对它们的点击次数,最高的达每周430万次(www.netscape.com),最低的每天也有30万次。说了这么多,我们还是得提醒读者注

意他们声称的Web服务器的性能和统计数据。如本章中我们所看到的，以下这些提法有很大的区别：每天点击次数、每天连接数、每天客户数和每天会话数。另一个要搞清楚的事实是一个组织的Web服务器程序运行在几台主机上，我们将在下一节讨论这种情况。

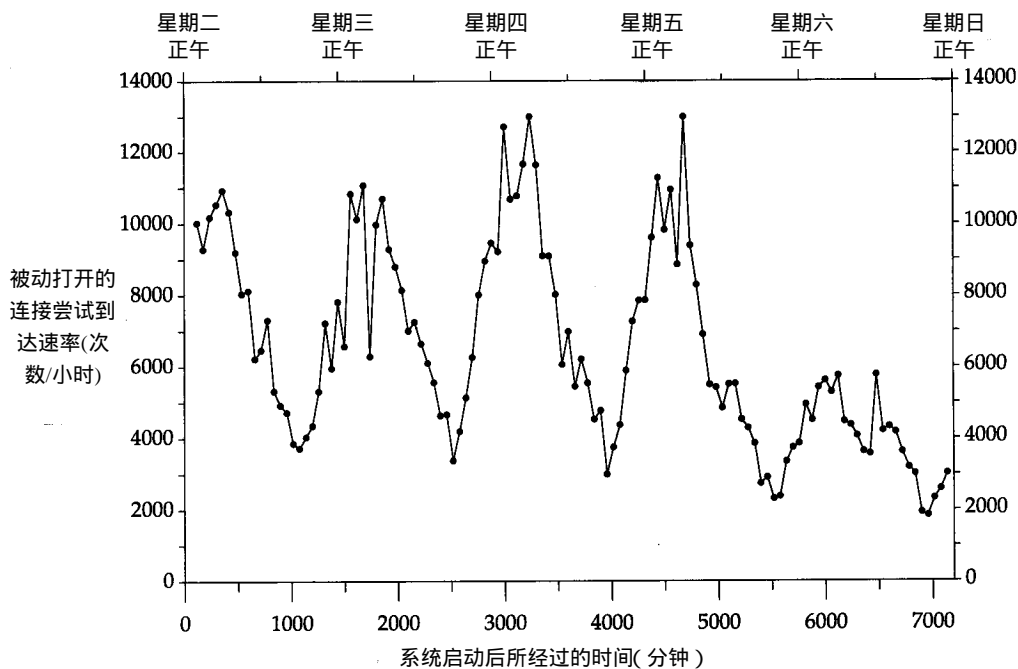


图14-2 每小时被动连接尝试次数

14.2 多个HTTP服务器

最简单的HTTP服务器安排是一台主机上运行一个 HTTP服务器程序。有很多 Web 站点是这样做的，但也有两种较为普遍的变形：

- 1) 一台主机，多个服务器程序。本章中所分析的数据就来源于一台按这种方式运行的主机。单个主机为多个组织提供HTTP服务。每一个组织的WWW域名(`www.organization.com`)映射一个不同的IP地址(都在同一子网上)，单个以太网接口分别对每一个不同的IP地址赋予别名(第6.6节中描述了Net / 3怎样允许单个网络接口上的多个IP地址。在主IP地址之后指派给网络接口的IP地址均称为别名)。这22个httpd服务器实例中的每一个都只使用一个IP地址。当服务器程序启动时，它把本地的IP地址绑定到它的监听TCP插口上，因此它只收到那些目的地址是它的IP地址的连接。
- 2) 多台主机，每台均提供服务器程序的一个副本。这种技术用于繁忙的组织在多个主机上分布输入负载(即负载平衡)。对应组织的WWW域名：`www.organization.com`指派了多个IP地址，每一个提供HTTP服务的主机有不同的IP地址(卷1的第14章，DNS中的多条A记录)。这种组织的DNS服务器响应DNS客户请求时，必须能以不同的顺序返回多个不同的IP地址。DNS中把这个称为循环使用(round-robin)，例如，在通常的DNS服务器程序当前版本中均支持这种功能。

例如，NCSA提供9个HTTP服务器。我们第一次查询它们的域名服务器时，返回如下：

```
$ host -t a www.ncsa.uiuc.edu newton.ncsa.uiuc.edu
Server: newton.ncsa.uiuc.edu
Address: 141.142.6.6 141.142.2.2
www.ncsa.uiuc.edu      A      141.142.3.129
www.ncsa.uiuc.edu      A      141.142.3.131
www.ncsa.uiuc.edu      A      141.142.3.132
www.ncsa.uiuc.edu      A      141.142.3.134
www.ncsa.uiuc.edu      A      141.142.3.76
www.ncsa.uiuc.edu      A      141.142.3.70
www.ncsa.uiuc.edu      A      141.142.3.74
www.ncsa.uiuc.edu      A      141.142.3.30
www.ncsa.uiuc.edu      A      141.142.3.130
```

(host程序在卷1第14章有描述并用到了它。)上例命令中的最后一个参数是我们要查询的NCSA的DNS服务器的名字,使用该参数的原因是:在缺省情况下, host程序将使用本地DNS服务器,而本地域名服务器的缓存中可能有这9个记录,而且可能每次返回同一个IP地址。

第二次我们再运行上例中的程序时,得到了不同次序。

```
$ host -t a www.ncsa.uiuc.edu newton.ncsa.uiuc.edu
Server: newton.ncsa.uiuc.edu
Address: 141.142.6.6 141.142.2.2
www.ncsa.uiuc.edu      A      141.142.3.132
www.ncsa.uiuc.edu      A      141.142.3.134
www.ncsa.uiuc.edu      A      141.142.3.76
www.ncsa.uiuc.edu      A      141.142.3.70
www.ncsa.uiuc.edu      A      141.142.3.74
www.ncsa.uiuc.edu      A      141.142.3.30
www.ncsa.uiuc.edu      A      141.142.3.130
www.ncsa.uiuc.edu      A      141.142.3.129
www.ncsa.uiuc.edu      A      141.142.3.131
```

14.3 客户端SYN的到达间隔时间

下面我们来做一件有趣的事情:通过观察客户端 SYN的到达,我们来看平均请求速率和最大请求速率之间的区别。服务器应有能力应付峰值负载,而不是平均负载。

通过对SYN / FIN / RST进行24小时跟踪,我们可以分析客户端SYN的到达时间间隔。在这个24小时的跟踪期间共有160 948个SYN到达(在本章的开头我们曾提到,在这期间有147 103次连接尝试。这中间的不同是因为SYN的重传。注意到,大约有10%的SYN须重传)。最小的到达间隔时间是0.1 ms,最大值是44.5秒,平均值是538 ms,中间值是222 ms。91%的到达间隔时间小于1.5秒。图14-3给出了到达间隔时间的柱状图。

这张图虽然有趣,但它不能提供峰值到达速率。为了测定峰值速率,我们把一天的24小时划分为1秒的时间间隔,并计算每秒的SYN到达个数(实际测量了86 622秒,比24小时长几分钟)。图14-4列出了前20个时间间隔内计数器的值。图中第二列给出了所有到达的SYN数,第三列的计数器表示的是忽略重传后的SYN到达数。在本节的最后,我们将用到第三列的数据。

例如,考虑所有到达的SYN,一天有27 868秒(一天中的32%)内没有SYN到达,22 471秒(一天中的26%)内只有一次SYN到达,等等。一秒中最大的SYN到达数为73次,一天中共有两次这种情况。我们观察所有SYN到达次数超过50次的“秒”,将发现它们都在一个3分钟的时间段内,这就是我们要找的峰值。

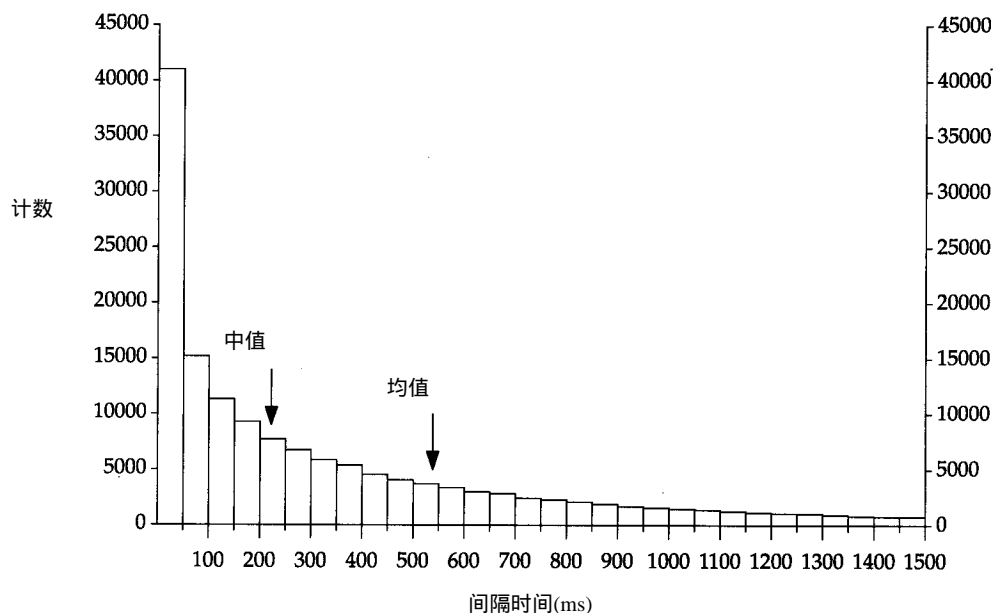


图14-3 客户端SYN的到达间隔时间分布

1秒钟内到达的SYN个数	所有SYN的累计数	新SYN的累计数
0	27 868	30 565
1	22 471	22 695
2	13 036	12 374
3	7 906	7 316
4	5 499	5 125
5	3 752	3 441
6	2 525	2 197
7	1 456	1 240
8	823	693
9	536	437
10	323	266
11	163	130
12	90	66
13	50	32
14	22	18
15	14	10
16	12	9
17	4	3
18	5	2
19	2	1
20	3	0
	86 560	86 620

图14-4 给定秒数内到达的SYN数

图14-6是含有峰值的那个小时的情况。在这个图中，我们把每 30秒到达的SYN数取平均值，y轴表示的是每秒到达的SYN数，平均到达速率约为每秒 3.5个，因此，这个小时处理的到达的SYN几乎为平均值的两倍。

图14-7给出了包含峰值的那个3分钟的更详细的情况。

在这3分钟中的变化有违人们的直觉，也表明某些客户有反常行为。如果我们检查这3分钟Tcpdump程序的输出会发现，问题果然来自一个特别的客户。在包含图14-7最左边尖峰的30秒中，那个客户在两个不同的端口发送1024个SYN，平均每秒30个。有少数几秒还在60~65次之间，再加上其他客户发送的，在图中的峰值就接近70个。图14-7中中间的尖峰也是由这个客户引起的。

图14-5列出了与这个客户相关的部分Tcndump输出。

```
1 0.0 client.1537 > server.80: S 1317079:1317079(0)
                                     win 2048 <mss 1460>
2 0.001650 (0.0016) server.80 > client.1537: S 2104019969:2104019969(0)
                                     ack 1317080 win 4096 <mss 512>
3 0.020060 (0.0184) client.1537 > server.80: S 1317092:1317092(0)
                                     win 2048 <mss 1460>
4 0.020332 (0.0003) server.80 > client.1537: R 2104019970:2104019970(0)
                                     ack 1317080 win 4096
5 0.020702 (0.0004) server.80 > client.1537: R 0:0(0)
                                     ack 1317093 win 0
6 1.938627 (1.9179) client.1537 > server.80: R 1317080:1317080(0) win 2048
7 1.958848 (0.0202) client.1537 > server.80: S 1319042:1319042(0)
                                     win 2048 <mss 1460>
8 1.959802 (0.0010) server.80 > client.1537: S 2105107969:2105107969(0)
                                     ack 1319043 win 4096 <mss 512>
9 2.026194 (0.0664) client.1537 > server.80: S 1319083:1319083(0)
                                     win 2048 <mss 1460>
10 2.027382 (0.0012) server.80 > client.1537: R 2105107970:2105107970(0)
                                     ack 1319043 win 4096
11 2.027998 (0.0006) server.80 > client.1537: R 0:0(0)
                                     ack 1319084 win 0
```

图14-5 违规的客户以高速率发送无效的SYN

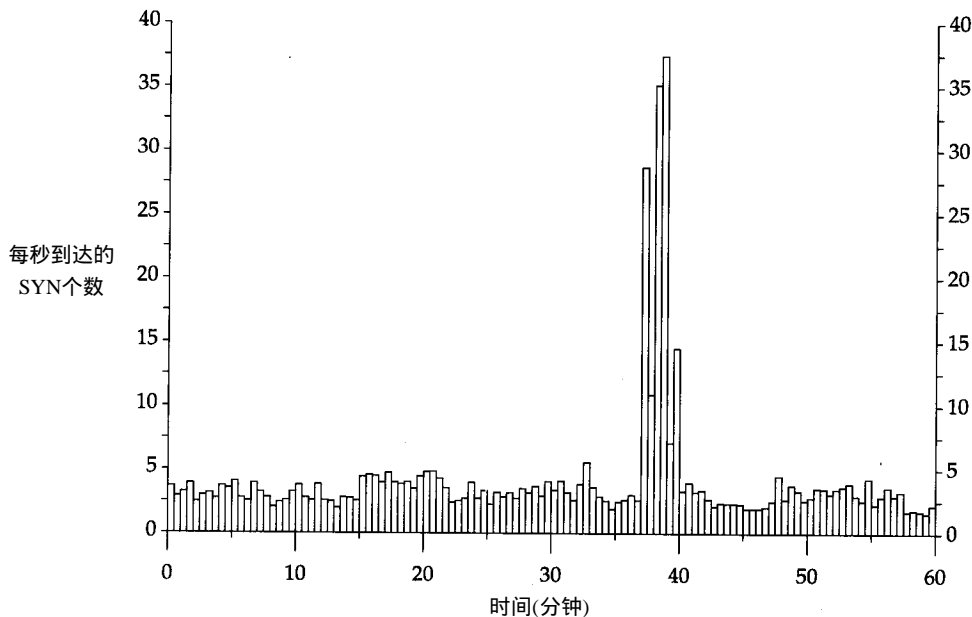


图14-6 60分钟时间内每秒到达的SYN数

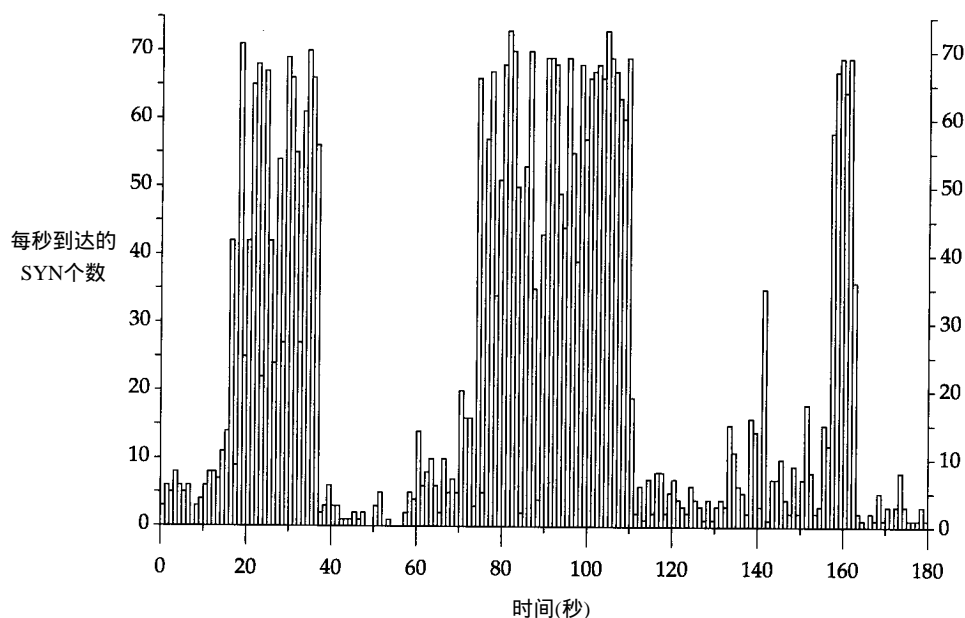


图14-7 3分钟的峰值时间内每秒到达的SYN数

第一行是表示客户的SYN，第二行是服务器的SYN / ACK。第三行是从同一个客户的同一个端口来的另一个SYN，但它的起始序列号是13，比第一行的高。第四行是服务器发送一个RST，第五行发送另一个RST，第六行是客户发送的RST。从第7行开始又重复这个情况。

为什么服务器要在一行内给客户发送两个RST(第五行和第六行)？可能是由于设有打印出来的某些数据段引起，因为遗憾的是Tcpdump跟踪程序仅包含有SYN、FIN或RST标志的报文段。然而，这个客户显然违规了，在同一个端口如此高速率地发送SYN，并且从一个SYN到下一个的序列号增加很小。

忽略重传的SYN后的计算结果

我们需要忽略重传的SYN，重新分析客户SYN的到达间隔时间。因为从上面我们可以看出，一个违反常规的客户就可以将数据拉出显著的峰值来。正如我们在本节的前面所提到的，忽略重传可以减少约10%的SYN。同样，通过考察有效的SYN，我们可以来分析连接到达服务器的速率。所有到达的SYN均影响TCP/IP协议的处理(因为每一个SYN要经过设备驱动程序、IP输入，然后才是TCP输入)，连接的到达速率影响HTTP服务器(服务器程序为每一个连接处理新的客户请求)。

在忽略重传SYN后，图14-3中的平均值由538 ms增加至600 ms，中间值由222 ms增加至251 ms。在图14-4中我们已给出每秒到达的SYN的分布图。峰值也像图14-6中表示的那样，不过要小得多。一天中到达的SYN数最大的3秒内分别为有19、21、33个SYN到达。这就给我们一个范围，从每秒4个(由到达时间间隔中值251 ms得来)到33个SYN，约为8倍的关系。这就意味着，当我们设计一个Web服务器时，应使它能适应的峰值在这种平均值之上。在14.5节中我们将看到这种入连接请求队列中的峰值到达速率的作用。

14.4 RTT的测量

下一个我们感兴趣的内容是各种客户与服务器之间的往返时间。不幸的是，我们不能通过在服务器上跟踪SYN/FIN/RST来测量它。图14-8描述了TCP三次握手和用四个报文段来终止一个连接的情况(第一个FIN由服务器发出)。加粗的线表示在跟踪SYN / FIN / RST时可以被跟踪到。

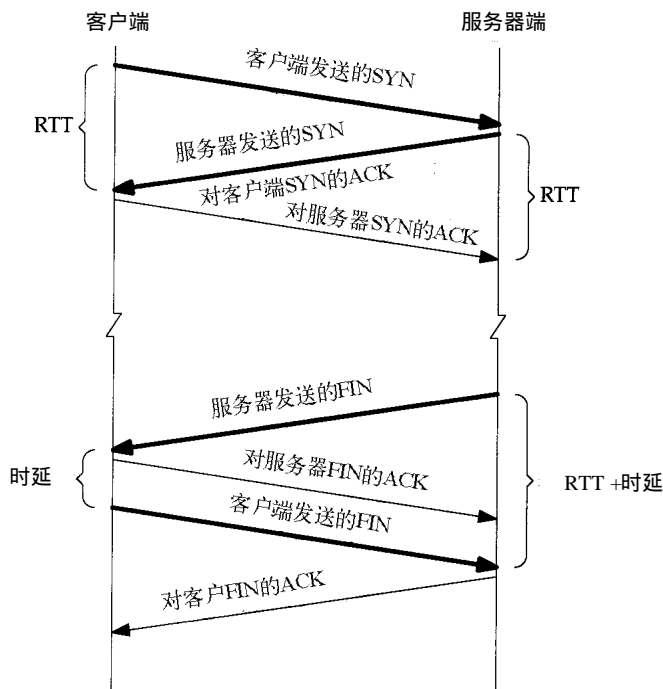


图14-8 TCP的三次握手和连接终止

在客户端可以测量 RTT，即发送SYN与接收服务器发来的 SYN 之间的时间间隔，但我们的测量均在服务器端。我们可以通过测量服务器发送 FIN 与接收客户发来的 FIN 之间的时间间隔来测量 RTT，但是这种测量包含一个不确定的时延：客户应用程序收到文件结束标志与关闭连接之间的时间。

我们需要跟踪服务器上的所有分组来测量 RTT，因此我们使用前面提到的 2.5 小时的跟踪，并测量服务器发送 SYN / ACK 与收到客户的 ACK 之间的时间间隔。客户发送的、用来确认服务器 SYN 的 ACK 报文通常不会被延迟（卷 2 第 758 页），因此这种测量不会包含一个时延的 ACK。这些报文通常都是尽可能的小（服务器的 SYN 为 44 字节，通常包括一个服务器上使用的 MSS 选项，客户的 ACK 为 40 字节），因此在较慢的 SLIP 或 PPP 链路上也不会产生明显的时延。

在 2.5 小时内，进行了 19 195 次 RTT 的测量，涉及 810 个不同的 IP 地址。最小的 RTT 等于 0（从同一主机的客户程序），最大的 RTT 是 12.3 秒，平均值是 445 ms，中间值是 187 ms。图 14-9 给出了 3 秒以内的 RTT 的分布。98.5% 的 RTT 在 3 秒以内。这些测量表明，由大西洋岸至太平洋岸的 RTT 最好的情况在 60 ms 左右，典型情况下的 RTT 值至少是这个值的三倍。

为什么中间值(178 ms)比由大西洋岸至太平洋岸的RTT(60 ms)小这么多?一种可能是目前情况下,大量的用户仍使用拨号线访问 Internet,即使是最快的调制解调器(28 800 bps),也给每个RTT增加100~200 ms的时延。另外一个原因是,有些客户实现在处理三次握手的第三个报文段(客户发送的、用来确认服务器SYN的ACK报文)时产生了时延。

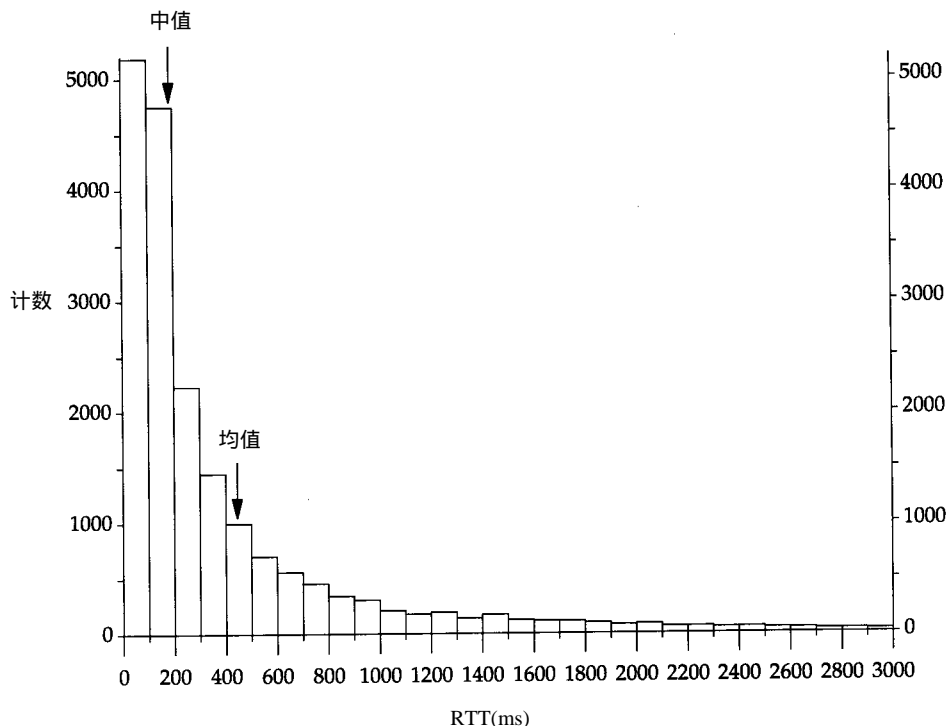


图14-9 客户往返时间的分布

14.5 用listen设置入连接队列的容量

为了准备一个接收入连接请求的插口,服务器通常执行下面的调用:

```
listen(sockfd, 5);
```

第二个参数称为 *backlog*, 指示listen调用的入连接队列的容量。BSD内核因为历史的原因,通过在<sys/socket.h>头文件中定义SOMAXCONN常量,将入连接队列的容量的上限设为5。如果应用程序指定了一个大于5的值,内核将不作任何提示地把它置为SOMAXCONN。新的内核将SOMAXCONN的值增加至10或更高,增加的原因我们马上要介绍。

在插口数据结构中, *so_qlimit*值就等于backlog参数值(卷2第365页)。当一个TCP入连接请求到达时(客户端的SYN), TCP程序执行sonewconn调用,紧接着进行如下测试(卷2第370页的第130~131行):

```
if (head->so_qlen + head->so_q0len > 3 * head->so_qlimit / 2)
    return ((struct socket *)0);
```

正如卷2中所描述的,把应用程序指定的backlog乘以一个毫无根据的因子: $3/2$,

确实能在内核指定 backlog 为 5 时将等待的连接数增加至 8。这个毫无根据的因子只在基于伯克利的实现中有作用(卷1第195页)。

这个队列长度的上限限制以下两项的和：

- 1) 未完成连接队列(so_q0len, 一个SYN已经到达、但三次握手还没有完成的连接)中的项数。
- 2) 已完成连接队列(so_q1len, 三次握手已完成、内核正等待进程执行 accept调用)中的项数。

卷2第369页详细描述了当一个TCP连接请求到达时, 服务器端处理的步骤。

当已完成连接队列被填满(例如, 服务器进程或服务器主机非常繁忙时, 进程执行 accept调用不够快, 不能及时清空队列), 或未完成连接队列被填满时, 将达到 backlog 的上限。当服务器主机与客户主机的往返时间较长, 而相比较而言, 新的连接请求到达较快, 那么服务器就要面对上述的后一个问题, 因为一个新的 SYN 占用队列中的一个记录项的时间是一次往返时间。图 14-10 描述了未完成连接队列的这部分时间。

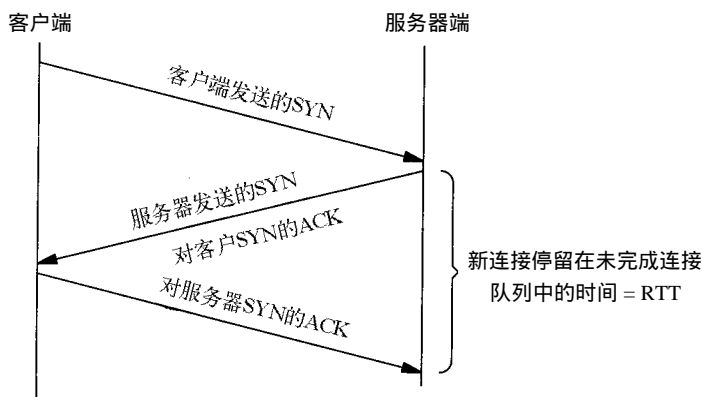


图14-10 用分组表示的未完成连接队列中一个记录项的占用时间。

为了检验未完成连接队列是否已满(不是已完成连接队列), 我们使用一个被修改过的 netstat 程序, 在最繁忙的 HTTP 监听服务器上连续打印 so_q0len 和 so_q1len 这两个变量的值。这个程序共运行了 2 个小时, 进行了 379 076 次采样, 或者说约每 19 ms 进行一次采样。图 14-11 给出了结果。

前面曾经提到, 将 backlog 设为 5 时, 实际上可以有 8 条连接在排队。已完成连接队列绝大部分时间是空的, 因为当有连接进入这个队列时, 只要服务器程序的 accept 调用一返回, 这条连接便会马上从该队列中被取走。

当队列已满时, TCP 丢弃入连接请求(卷2第743页), 并且假定客户程序会发生超时, 重传它的 SYN, 希望在几秒钟以后在队列中找到空闲位置。但是 Net/3 的内核并不提供有关丢失的 SYN 的统计数据, 因此系统管理员无法知道这种情况发生的频度。我们把系统中这一段代码作了如下修改：

队列长度	未完成连接队列计数	已完成连接队列计数
0	167 123	379 075
1	116 175	1
2	42 185	
3	18 842	
4	12 871	
5	14 581	
6	6 346	
7	708	
8	245	
	379 076	379 076

图14-11 繁忙的HTTP服务器的连接队列长度分布

```
if (so->so_options & SO_ACCEPTCONN) {
    so = sonewconn(so, 0);
    if (so == 0) {
        tcpstat.tcp_listendrop++; /* new counter */
        goto drop;
    }
}
```

所作的修改就是增加了一个计数器。

图14-12中列出了为期5天、一小时采集一次得到的该计数器的值。这个计数器是对主机上的所有服务器程序进行统计的，但是我们假定所监视的主机主要是作为一台 Web服务器，实际上绝大多数的溢出也是发生在httpd的侦听插口上。从平均上来说，这台主机每分钟的呼入连接请求溢出刚好超过三个(22 918次溢出除以7 139分钟)，但是这里也有几个值得注意的连接丢失数量的跳跃点。大约在第4500分钟(星期五下午4 00)左右，一个小时内丢弃了1964个入连接请求，约为每分钟32个(每两秒钟一个)。其他两次值得注意的跳跃发生在星期二下午的早些时候。

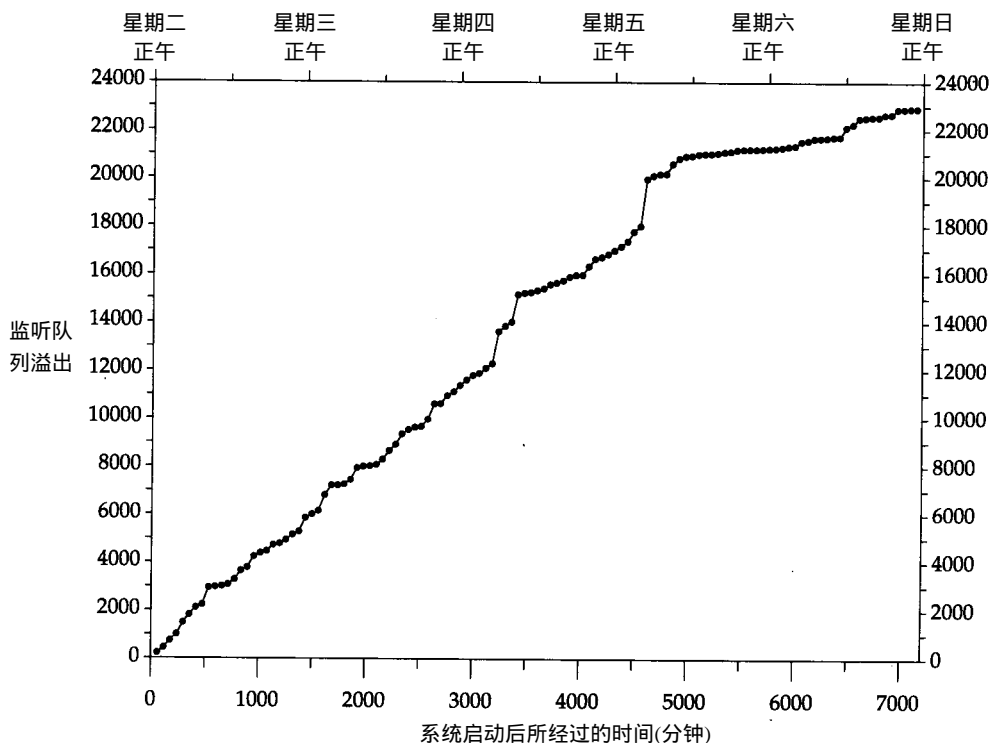


图14-12 服务器监听队列的溢出

必须增加支持繁忙服务器的内核的 backlog 参数的上限，同时必须修改繁忙服务器应用程序(例如httpd)，使之设置一个更大的 backlog。例如，httpd的1.3版就存在这个问题，因为它用下面的语句将backlog强制设置为5：

```
listen(10, 5);
```

1.4版将backlog增加到了35，但这对于某些繁忙的服务器来说还是不够。

不同的厂商采用不同的方法来增加内核的 backlog 的上限。例如，BSD / OS V2.0内核将somaxconn全局变量指定为16，但系统管理员可以将它调整至更大的值。Solaris 2.4允许系统管理员使用nnd程序改变TCP参数：tcp_conn_req_max，这个参数的默认值为5，最

大可以到32。Solaris 2.5将默认值增加到32，而最大可以到1024。不幸的是，应用程序使用listen调用时，没有一个简单的办法来确定当前操作系统内核所允许的队列最大值，所以最好的办法是应用程序的代码中给这个参数赋一个很大的值（因为使用listen调用时不会因为这个值太大而返回错误），或者让用户可以在命令行中指定这个参数。在[Mogul 1995c]提出一种思想，认为在listen调用中应忽略这个参数，而由系统内核直接把它设为最大值。

有些应用程序特意将backlog参数设为一个较低的值来限制服务器的负载，因此，在这种情况下我们要避免增加这些应用程序中的这个参数值。

SYN_RCVD错误

当我们检查netstat的输出时发现，插口在SYN_RCVD状态下保持了几分钟。Net / 3用它的连接建立定时器限制这个状态保持的时间为75秒(卷2第664页和755页)，为什么还会出现这种现象？图14-13列出了Tcpdump的输出。

```

1    0.0                client.4821 > server.80: S 32320000:32320000(0)
                                win 61440 <mss 512>
2    0.001045 ( 0.0010) server.80 > client.4821: S 365777409:365777409(0)
                                ack 32320001 win 4096 <mss 512>
3    5.791575 ( 5.7905) server.80 > client.4821: S 365777409:365777409(0)
                                ack 32320001 win 4096 <mss 512>
4    5.827420 ( 0.0358) client.4821 > server.80: S 32320000:32320000(0)
                                win 61440 <mss 512>
5    5.827730 ( 0.0003) server.80 > client.4821: S 365777409:365777409(0)
                                ack 32320001 win 4096 <mss 512>
6    29.801493 (23.9738) server.80 > client.4821: S 365777409:365777409(0)
                                ack 32320001 win 4096 <mss 512>
7    29.828256 ( 0.0268) client.4821 > server.80: S 32320000:32320000(0)
                                win 61440 <mss 512>
8    29.828600 ( 0.0003) server.80 > client.4821: S 365777409:365777409(0)
                                ack 32320001 win 4096 <mss 512>
9    77.811791 (47.9832) server.80 > client.4821: S 365777409:365777409(0)
                                ack 32320001 win 4096 <mss 512>
10   141.821740 (64.0099) server.80 > client.4821: S 365777409:365777409(0)
                                ack 32320001 win 4096 <mss 512>

                                服务器每64秒重传ACK / SYN
18   654.197350 (64.1911) server.80 > client.4821: S 365777409:365777409(0)
                                ack 32320001 win 4096 <mss 512>

```

图14-13 服务器插口在SYN_RCVD状态被阻塞近11分钟

客户发送的SYN在第一个报文段中到达，服务器的SYN / ACK在第二个报文段发出。同时服务器设置连接建立定时器为75秒，重传定时器为6秒。上图的第3行中，重传定时器溢出，服务器重传SYN / ACK。这正是我们所期望的。

第4行中可以看到客户端的响应，但这个响应是重传第1行中的最初的那个SYN，而不是我们所期望的对服务器SYN的响应ACK。客户端好像是被中断了。服务器给出了正确的响应：重传SYN / ACK。收到第4个报文段后，服务器端的TCP程序将这条连接的保活定时器(keepalive timer)的超时间隔设为2小时(卷2第745页)。但是，保活定时器与连接建立定时器使用连接控制块中的同一个计数器(卷2的图25-2)，因此，程序清除该计数器的当前值69秒，而

把它设成2小时。通常客户端用一个响应服务器 SYN的ACK来完成三次握手，建立 TCP连接。当这个ACK报文被处理后，保活定时器被设为 2小时，重传定时器则被关闭。

第6、7、8 行的情况类似。服务器的重传定时器在 24秒后超时，重传它的 SYN / ACK，但是客户端的响应(它又一次重传了最初的SYN)不正确，因此服务器再次正确地重传 SYN / ACK。在第9行可以看到，服务器的重传定时器在 48秒后再次超时，同样重传它的 SYN / ACK。这样，重传定时器到了它的最大值：64秒，在连接被丢弃之前共发生了 12次重传(12是卷2第674页中的常量TCP_MAXRXTSHIFT的值)。

因为保活定时器、连接建立定时器共用 TCPT_KEEP计数器，所以修补这个故障的方法是：连接还没有完全建立好时(卷2第745页)，不将保活定时器的超时间隔设为 2小时。当然，作了上述修改后，就要求当连接转移到已建立的状态后把保活定时器设置成初始值 2小时。

14.6 客户端的SYN选项

我们在为期 24小时的跟踪中收集了所有的 SYN报文段，从中我们可以看到伴随 SYN的一些不同的参数和选项。

客户端口号

基于伯克利的系统分配的客户临时使用的端口号的范围是 1024~5000(卷2第588页)。正如我们所期望的那样，超过 160 000个客户中有 93.5%使用的端口在这个范围内。有 14个客户连接请求使用的端口号小于 1024(端口号小于 1024的在Net / 3中通常作为保留端口)，其余的6.5%都在5001~65535之间。有些系统，特别是 Solaris 2.x，分配的客户端口号都大于 32768。

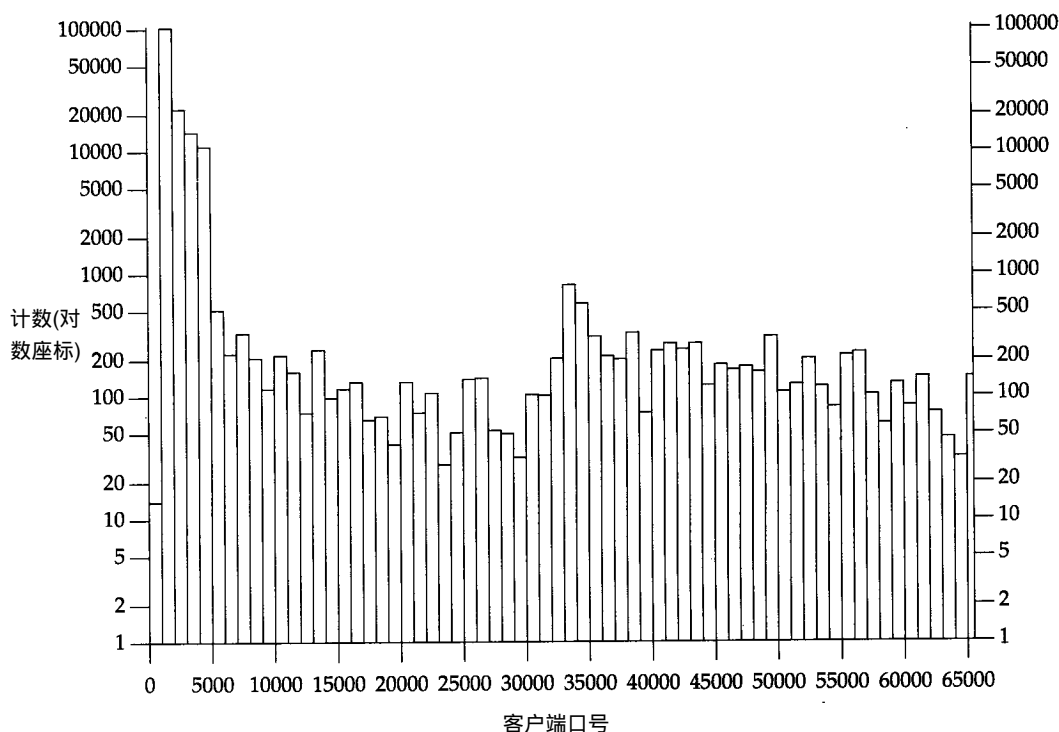


图14-14 客户端口号的范围

图14-14是一个客户使用端口号的分布图，每 1000个端口(如1001~2000、2001~3000)作为一个统计范围。请注意 y轴是对数座标。同时我们也看到，绝大部分客户使用的端口在 1024~5000之间，而且 2/3 的端口在 1024~2000之间。

最大报文长度

可以基于选用网络的MTU(见前面我们对图 10-9的讨论)或直接使用固定值(非本地的同层之间使用512或536，较老的BSD系统使1024，等等)来设置MSS。RFC 1191[Mogul and Deering 1990]列出了典型的16种不同的MTU。因此，在实验中我们希望能找到 Web客户所发出的不同的MSS的值有十几种或更多。事实上我们找到了117种不同的值，范围在128~17 520之间。

图14-15列出了最常见的13种客户通告的MSS的值。这5071个连到Web服务器的客户占总客户数5386的94%。第一栏中标记“none”的意思是客户的SYN中没有通告MSS。

MSS	计 数	注 释
无	703	RFC 1122指出如不使用选项，则设为536
212	53	
216	47	256-40
256	516	MTU为296的PPP或SLIP链路
408	24	
472	21	512-40
512	465	非本地主机的常用默认值
536	1097	非本地主机的常用默认值
966	123	ARPANET MTU (1006) - 40
1024	31	老版本BSD中本地主机的默认值
1396	117	
1440	248	以太网MTU (1500) - 40
1460	1626	
	5071	

图14-15 客户所通告的MSS值的分布

初始窗口宽度通告

客户的SYN中也包含了客户端的初始窗口宽度的通告。这里共有117种不同的值，跨越了整个允许值的范围：0~65 535。图14-16列出了最常见的14种值的使用统计数。这4990个值占5386个不同客户的93%。有些值有特殊的意义，但有些值让人感到迷惑，例如：22 099。

好像有些PC平台上的Web浏览器允许用户指定MSS和初始窗口尺寸。这就是我们看到一些奇怪值的一个原因，用户设置这些值时可能并没有理解它们的作用。

不管怎么说，我们共找到117种不同的MSS值和117种不同的初始窗口尺寸，并检查了267种不同的MSS和初始窗口尺寸的组合，并没有发现它们之间有明显的相关性。

窗 口	计 数	注 释
0	317	
512	94	
848	66	
1024	67	
2048	254	
2920	296	2 × 1460
4096	2062	接收缓存大小的默认值
8192	683	小于常用的默认值
8760	179	6 × 1460 (以太网上常用)
16384	175	
22099	486	7 × 7 × 11 × 41 ?
22792	128	7 × 8 × 11 × 37 ?
32768	94	
61440	89	60 × 1024
	4 990	

图14-16 客户所通告的初始窗口尺寸分布

窗口比例和时戳选项

RFC 1323指定了窗口比例和时戳选项(图2-1)。在5 386个不同的客户中共有78个只发送了窗口比例选项, 23个既发送了窗口比例选项, 又发送了时戳选项, 没有一个只发送时戳选项。在所有的窗口比例选项中都通告了偏移因子 0(意味着比例因子是 1, 或就是通告的TCP窗口的宽度)。

利用SYN发送数据

五个客户在发送的SYN中捎带数据, 但这些SYN并不包含新的T / TCP的选项。检查这些分组, 发现这些连接都是同一个模式。客户发送一个普通的 SYN, 不含任何数据。三次握手的第二个报文段是服务器的响应, 但是响应好像丢失了, 因此客户重传了它的 SYN。但是每一个客户重传的SYN中都包含有数据(在200~300字节之间, 一个常见的HTTP客户请求)。

路径MTU发现

在RFC 1191[Mogul and Deering 1990]和卷1的第24.2节均描述了路径MTU发现。通过检查客户所发送的SYN报文段中的DF 位(不分段), 可以判断客户是否支持这个选项。在我们的例子中, 共有679个客户(占12.6%)支持路径MTU发现。

客户初始序列号

有大量的客户(超过10%)使用0作为初始序列号, 用0作为初始序列号明显违反了TCP规范。这些客户的TCP / IP实现中对所有的主动连接都使用 0作为初始序列号, 在跟踪中我们发现, 同一个客户在几秒钟内在不同端口发出的多个连接请求均使用 0作为初始序列号。图 14-19 列出一个这样的客户。

14.7 客户端的SYN重传

伯克利派生系统是在初始 SYN发出6秒后重传SYN(如果需要), 如果在24秒内仍收不到响应, 就再重传(卷2第664页)。因为在24小时的跟踪中我们记录下了所有的 SYN报文(包括那些没有被网络和Tcpdump丢弃的), 所以我们能从中看出客户重传 SYN有多频繁和每一次重传之间的时间。

在这24小时的跟踪中共有 160 948个SYN到达(见第14.3节), 其中17 680个(占11%)是重复的(真正的重传数量要小一些, 因为如果指定 IP地址和端口号的连续两个SYN报文之间的时间非常长, 那么第二个SYN就不是重传, 而是后来发起的另一条连接。我们没有试图去减掉这部分重传, 因为它只占11%中的一小部分)。

SYN只重传一次(最通常的情况), 重传时间典型值是在发出初始 SYN以后3、4或5秒。如果要重传多次, 许多客户使用BSD的算法: 第一次重传是在6秒以后, 接着的下一是24秒后。我们用{6, 24}来表示这种序列。其他观察到的序列是:

- {3, 6, 12, 14};
- {5, 10, 20, 40, 60, 60};
- {4, 4, 4, 4}(违反了RFC 1122中指数增长的要求);

- {0.7, 1.3} (20跳以外的主机过分频繁的重传；实际上，在这个主机上有 20个连接重传 SYN，所有的重传间隔都小于 500 ms！)；
- {3, 6.5, 13, 26, 3, 6.5, 13, 26, 3, 6.5, 13, 26} (这个主机每4次重传后重新按指数退避方法重传)；
- {2.75, 5.5, 11, 22, 44}；
- {21, 17, 106}；
- {5, 0.1, 0.2, 0.4, 0.8, 1.4, 3.2, 6.4} (第一次超时后太主动地重传)；
- {0.4, 0.9, 2, 4} (另一个19跳以外的主机过分频繁的重传)；
- {3, 18, 168, 120, 120, 240}。

就像我们所看到的，上面有些奇怪的序列。有些 SYN被重传很多次，可能是因为发送它的客户有路由问题：它能发送数据到服务器，但收不到服务器的任何响应。同样，也有可能是前一个连接请求的新的实例 (卷2第765~766页描述了BSD服务器是如何处理这种情况的：当新的SYN的序列号比处在TIME_WAIT状态的连接的最后一个SYN的序列号还大时，服务器将接受这个新的连接请求)，但是这个时间 (例如，明显的是3秒或6秒的倍数) 又让人看起来不太像。

14.8 域名

在24小时期间共有5386个不同IP地址的客户连接到Web服务器。因为Tcpdump (带-w标志) 只记录带IP地址的分组首部，因此我们必须再来找相应的域名。

我们第一轮用DNS查找名字，试图把这些IP地址映射到它们的域名，只找到了4052个 (占75%)。然后我们在DNS上运行了一天，查找剩下的1334个IP地址，又找到了62个域名。这意味着有23.6%的客户的IP地址到域名的逆映射不正确 (卷1的第14.5节讨论了这些指针查询)。虽然这些客户中的大部分都是通过拨号上网，而且大部分时间是离线的，但他们也应该有他们的名字服务器来提供名字服务，而且名字服务器应是任何时候都接入Internet的。

在DNS查找名字失败后，我们马上对剩下的1272个客户运行Ping程序，验证这些没有地址-名字映射的客户是不是会临时不可达。结果是Ping测试成功了520台主机 (占41%)。

分析这些没有映射到一个域名的IP地址的顶级域名的分布，发现它们来自57个不同的顶级域名。其中50个是除美国以外其他国家的两个字母的域名，这也说明用“世界范围内 (world wide)”这个词来形容Web是恰当的。

14.9 超时的持续探测

Net / 3从没有放弃过发送持续探测 (persist probe)。那就是，当Net / 3收到对等端发送的窗口为0的窗口通告后，它不管是否曾经收到过对方的任何报文，都不断地发送持续探测。当对等端完全消失时 (例如，在SLIP或PPP连接时挂断电话)，这样做就会产生问题。回忆一下卷2第723页提到的，当客户端消失时，有些中间路由器会发送一个主机不可达错误的ICMP报文，一旦连接建立，TCP将忽略这些错误。

如果连接没有被丢弃，TCP会每60秒往已经消失了的主机发送一个持续探测报文 (浪费Internet资源)，同时每一条连接还继续占用主机上的内存和TCP访问控制块。

图14-17列出的4.4BSD-Lite2中的代码修补了这个问题，用它来替代卷2第662页的代码。

```

case TCPT_PERSIST:
    tcpstat.tcps_persisttimeo++;
    /*
     * Hack: if the peer is dead/unreachable, we do not
     * time out if the window is closed. After a full
     * backoff, drop the connection if the idle time
     * (no responses to probes) reaches the maximum
     * backoff that we would use if retransmitting.
     */
    if (tp->t_rxtshift == TCP_MAXRXTSHIFT &&
        (tp->t_idle >= tcp_maxpersistidle ||
         tp->t_idle >= TCP_REXMTVAL(tp) * tcp_totbackoff)) {
        tcpstat.tcps_persistdrop++;
        tp = tcp_drop(tp, ETIMEDOUT);
        break;
    }
    tcp_setpersist(tp);
    tp->t_force = 1;
    (void) tcp_output(tp);
    tp->t_force = 0;
    break;

```

tcp_timer.c

tcp_timer.c

图14-17 正确处理持续超时的代码

图中的if语句是新代码。变量tcp_maxpersistidle是一个新定义的变量，它的初值是TCPTV_KEEP_IDLE(14 400个500 ms的时钟嘀哒 (clock tick)，或2小时)。变量tcp_totbackoff也是一个新变量，它的值是511，是tcp_backoff数组(卷2第669页)中所有元素之和。最后，tcps_persistdrop是tcpstat结构(卷2第638页)中的一个新的计数器，它统计被丢弃的连接。

TCP_MAXRXTSHIFT指定了TCP在等待ACK时的最大重传次数，它的值是12。如果在2小时或对等端的当前RTO的511倍(取两个中较小的)内没有收到对方任何报文，在12次重传后将丢弃连接。例如，RTO是2.5秒(5个时钟嘀哒，一个合理的值)，在22分钟(即2640个时钟嘀哒)后，OR测试条件中的后一个将引起丢弃连接，因为2640大于2555(即 5×511)。

代码中的“Hack”注释不是必需的。RFC 1122中规定：即使提供的窗口宽度为0，但只要接收TCP继续给探测报文发送响应，TCP就必须无限期地保持一个连接在打开状态。如果长时间内探测没有响应，最好还是丢弃连接。

在系统中加入的代码可以看出这种情况的发生有多频繁。图14-18给出了为期5天的这个新计数器的值。这个系统平均每天丢弃90个连接，每小时约4个。

让我们详细看一下其中一条连接。图14-19给出了Tcpdump分组跟踪的详细情况。

第1~3行中除了初始序列号错误(0)、MSS值有些奇怪以外，是比较常见的TCP三次握手过程。在第4行，客户发送了一个182字节的请求报文。第5行中服务器对请求进行了响应，在响应报文中包含了应答数据的前512字节，第6行是包含后512字节数据的应答。

在第7行客户发送了一个FIN，第8行中服务器对FIN进行了响应：ACK，紧接着在第9行服务器发送了1024字节的应答。客户在第10行确认了服务器的前512字节的应答，并重传它的FIN。第11行、第12行是服务器的后1024字节的应答。第13~15行中延续了这种情况。

注意，当服务器发送数据时，客户在第7、10、13和16行通告了窗口的减小，直到第17行

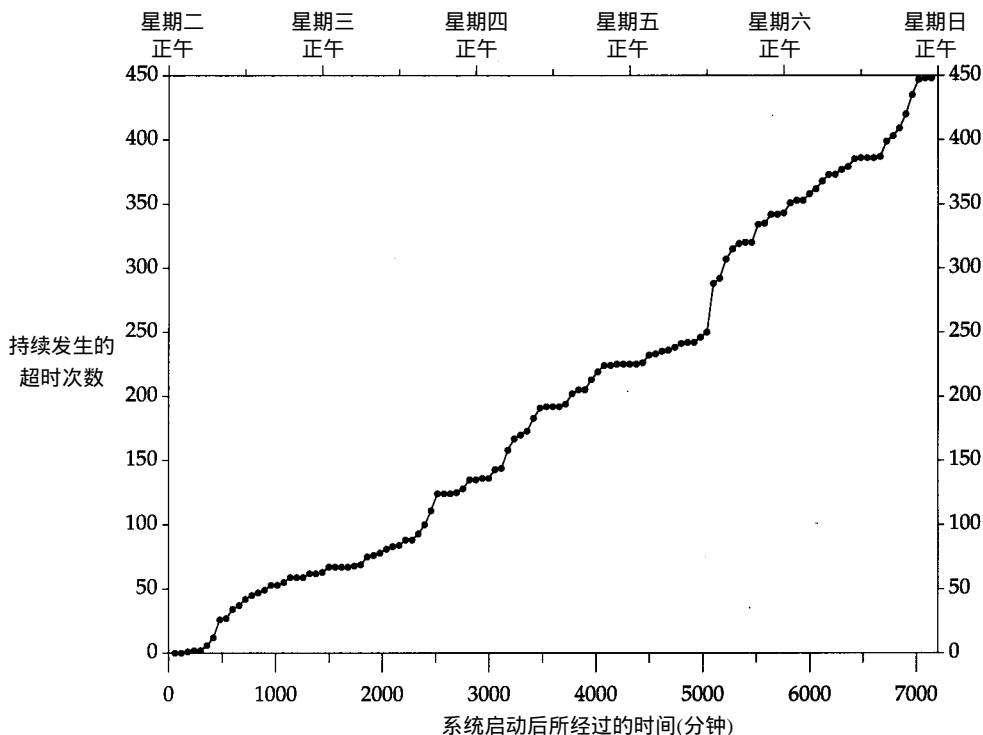


图14-18 持续探测超时后丢弃连接的数量

```

1  0.0 client.1464 > serv.80: B 0:0(0) win 4096 <www 1396>
2  0.001212 [0.0012] serv.80 > client.1464: B 323930113:323930113(0)
   ack 1 win 4096 <www 512>
3  0.354041 [0.3536] client.1464 > serv.80: P ack 1 win 4096
4  0.481275 [0.1164] client.1464 > serv.80: P 1:183(183) ack 1 win 4096
5  0.546304 [0.0650] serv.80 > client.1464: . 1:512(512) ack 183 win 4096
6  0.546761 [0.0005] serv.80 > client.1464: P 512:1025(512) ack 183 win 4096
7  1.393139 [0.8464] client.1464 > serv.80: FP 183:183(0) ack 512 win 3584
8  1.394103 [0.0010] serv.80 > client.1464: . 1025:1537(512) ack 184 win 4096
9  1.394587 [0.0005] serv.80 > client.1464: . 1537:2049(512) ack 184 win 4096
10 1.582501 [0.1879] client.1464 > serv.80: FP 183:183(0) ack 1025 win 3073
11 1.583139 [0.0006] serv.80 > client.1464: . 2049:2561(512) ack 184 win 4096
12 1.583600 [0.0005] serv.80 > client.1464: . 2561:3073(512) ack 184 win 4096
13 2.851548 (1.2679) client.1464 > serv.80: P ack 2049 win 2048
14 2.852214 (0.0007) serv.80 > client.1464: . 3073:3585(512) ack 184 win 4096
15 2.852672 (0.0005) serv.80 > client.1464: . 3585:4097(512) ack 184 win 4096
16 3.812675 (0.9600) client.1464 > serv.80: P ack 3073 win 1024
17 5.257997 (1.4453) client.1464 > serv.80: P ack 4097 win 0
18 10.024936 (4.7669) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
19 16.035379 (6.0104) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
20 28.055130 (12.0198) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
21 52.086026 (24.0309) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
22 100.135380 (48.0494) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
23 160.195529 (60.0601) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
24 220.255059 (60.0595) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096

```

图14-19 Tcpdump对持续超时的跟踪

持续探测连续进行

```
140 7187.603975 (60.0501) serv.80 > client.1464: . 4097:4098(1) ack 184 win 4096
141 7247.643905 (60.0399) serv.80 > client.1464: R 4098:4098(0) ack 184 win 4096
```

图14-19 (续)

窗口变为0。到17行为止，客户已接收了从服务器发来的 4096字节的数据，4096字节的接收缓存已满了，所以客户通告窗口为 0。客户端应用程序没有从接收缓存区中读任何数据。

第18行中服务器发出了它的第一个持续探测报文，它是收到客户窗口为 0的通告约5秒钟后发出的。持续探测报文的间隔时间按照卷 2图25-14的典型情况进行。在第 17行和18行之间的时间内，客户离开了 Internet。在接下来的2小时内，服务器共发送了 124个持续探测报文，最后服务器丢弃了连接，并在第 141行发送了一个RST报文(RST是由tcp_drop调用发送的，见卷2第713页)。

为什么这个例子中服务器发送持续探测报文时间长达 2小时，为什么没有按我们在本节前面讨论过的 4.4BSD-Lite2源代码中的OR测试的后半个条件来执行？我们所监视的系统使用的是 BSD / OS V2.0，其中持续超时测试代码只测试 t_idle是否大于或等于tcp_maxpersistidle。OR条件测试的后半部分是在 4.4BSD-Lite2中加入的新代码。在上面的例子中，我们也可以看出加入这段代码的原因：当通信的另一端显然已离开了Internet时，就不再需要进行2小时的持续探测了。

我们在上面提到系统平均每天有 90个这种持续超时的连接，这就意味着如果系统内核不终止这些连接，4天以后系统中将有360个这样的“保留”连接，这将引起每秒发送 6个无用的TCP报文。另外，因为HTTP服务器还将试图给这些客户发送数据，所以还会产生一些 mbuf在连接发送等待队列中等待发送。[Mogul 1995a]中提到：“当客户过早地终止TCP连接时，会引发服务器程序中隐藏的故障，从而真正地影响性能”。

图14-19的第 7行中，服务器收到客户发来的一个 FIN。这使服务器把连接置为 CLOSE_WAIT状态。但在跟踪过程中，有时服务器调用close调用，而转至LAST_ACK状态，我们并不能从Tcpdump的输出中区别出来。的确，绝大多数这种连接均在 LAST_ACK状态持续发送探测报文。

在1995年早期，最初开始对插口阻塞在 LAST_ACK状态的问题进行讨论时，有人建议设置 SO_KEEPA_LIVE选项来检测客户退出的时间，然后终止连接(卷1的第23章讨论了这个选项是怎么工作的，卷 2的第25.6节提供了使用它的细节)。不幸的是，这样做还是解决不了问题。注意卷 2第663页，KEEPA_LIVE选项在FIN_WAIT_1、FIN_WAIT_2、CLOSING和LAST_ACK状态并不终止连接。据报导，有些厂商对此作了改变。

14.10 T/TCP路由表大小的模拟

实现T/TCP的主机为每一个与它通信的主机保留一条路由表的表项(第6章)。如今的大部分主机维护的路由表只有一条缺省路由和少数显式指定的路由，所以实现 T/TCP可能要建立一个比通常使用的路由表大得多的路由表。我们将使用 HTTP服务器发出的数据来模拟 T/TCP的路由表，看它的空间大小是怎么变化的。

我们只进行简单的模拟。我们通过对这个主机进行 24 小时的分组跟踪来建立一个路由表，其中包含每一个与 HTTP 服务器通信的主机（共有 5386 个不同的 IP 地址）的路由。路由表保留的每一条路由信息都设有最后一次更新后的失效时间。我们把失效时间分别设为 30 分钟、60 分钟和 2 小时来进行仿真。每 10 分钟扫描一次路由表，把所有超过失效时间的路由信息删除（模仿 6.10 节中的 `in_rtqtime` 的动作），用一个计数器来记录表中剩下的条目。这些计数器都列在图 14-20 中。

在卷 2 的习题 18.2 中我们注意到，每一条 Net/3 的路由表表项要占用 152 字节。在 T/TCP 中，这个数字变成了 168 字节，增加的 16 字节是 `rt_metrics` 结构，用作 TAO 缓存，不过在 BSD 的内存分配策略中，实际分配的是 256 字节。如果取最大的失效时间：2 小时，路由表的表项数将达到 1000 个，即需要 256 000 字节。将失效时间减半，可以使内存的占用量减小一半。

当有 5 386 个不同的 IP 地址访问这个服务器时，如果失效时间设为 30 分钟，路由表最大可到约 300 条表项。这样的空间对路由表而言并不是很不切实际的。

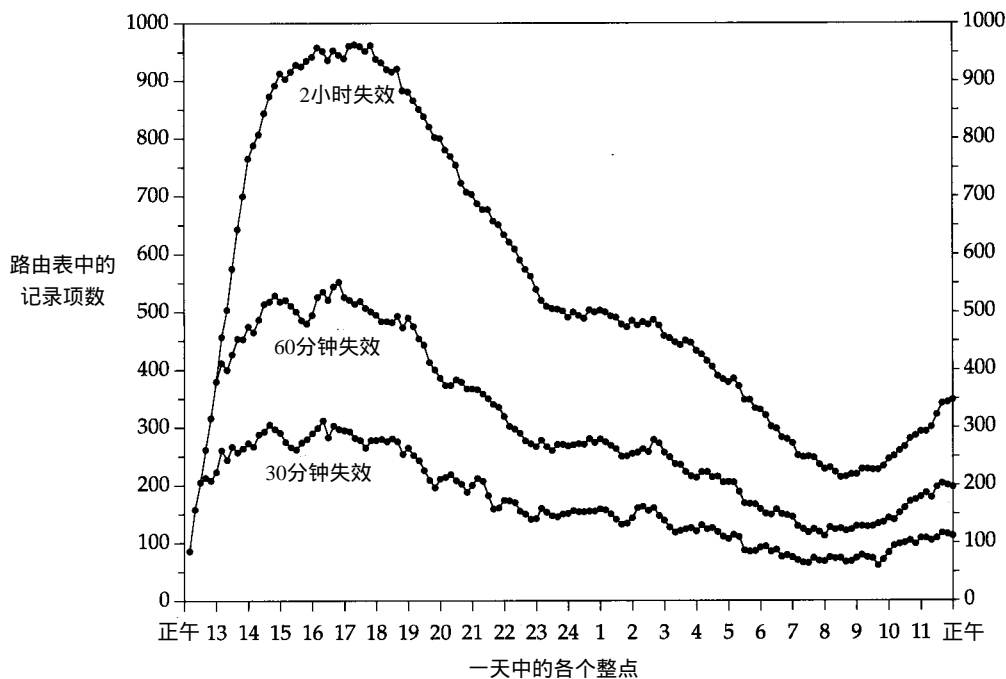


图14-20 T/TCP路由表模拟：每次不同的表项数

路由表的重用

图14-20告诉我们当使用不同的失效时间时路由表会变得多大。但是另一个我们关心的问题是：路由表中保留的这些路由信息中有多少被重用。没有必要保留那些很少用第二次的路由信息。

为了考察这一点，我们检查从 24 小时跟踪中得来的 686 755 个分组，并从中找寻在客户发出最后一个分组至少 10 分钟以后发出的 SYN。图 14-21 给出了主机数与静默时间（分钟）的相对关系图。例如，在 5386 个不同的客户所在的主机中，有 683 台主机在 10 分钟或超过 10 分钟的静

默时间后发送了另一个 SYN。在 11 分钟或超过 11 分钟的静默时间后发送了另一个 SYN 的主机减少至 669 台，静默时间超过 120 分钟发送了另一个 SYN 的主机为 367 台。

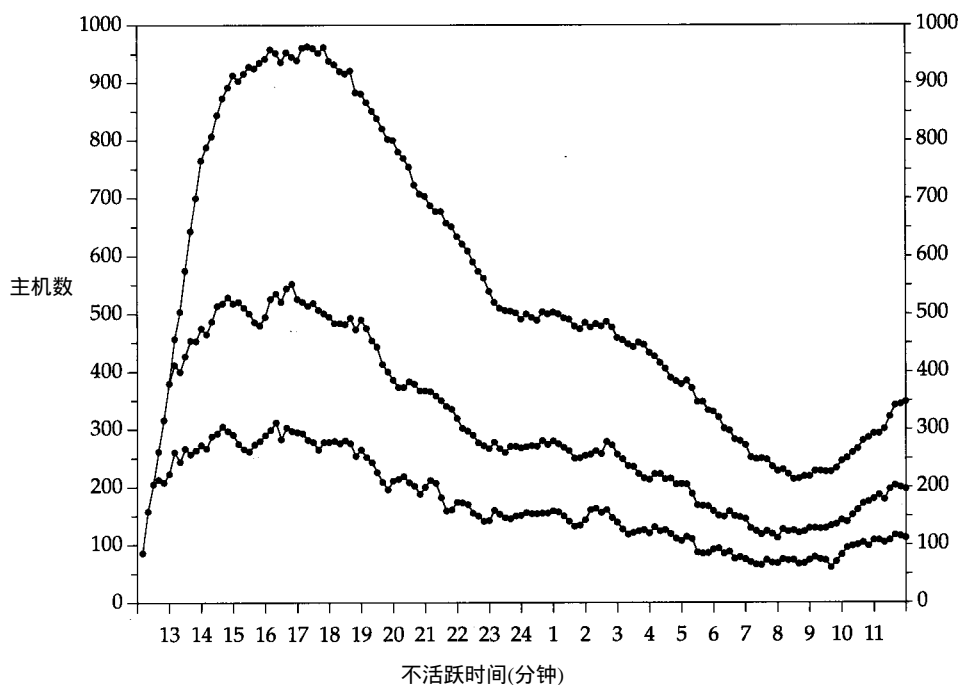


图14-21 在一段时间的静默后发送一个SYN的主机数

如果我们留意一下静默一段时间后又重现的主机，它们的 IP 地址所对应的主机名都是一些 `wwwproxy1`、`webgate1`、`proxy`、`gateway` 和类似于这样的名字，也就是说，多数是一些组织的代理服务器。

14.11 mbuf的交互

在用 `Tcpdump` 对 HTTP 数据交换进行监视时，我们发现了一个有趣的现象。尽管 `MSS` 的值大于 208（通常都是这样），可是当应用程序写数据的字节数在 101~208 之间时，4.4BSD 系统还是把它分成了两个 `mbuf`（一个用来存放前 100 字节，另一个存放剩余的 1~108 字节），这样成了两个 TCP 报文段。这个反常现象的原因是：`so_send` 函数（卷 2 第 399 页和第 400 页）。因为 TCP 不是一个原子协议，所以每填充一个 `mbuf`，协议的输出函数就被调用一次。

使事情变得更糟的是：因为现在客户的请求被分成多个报文，慢启动现象就产生了。客户只有在收到服务器对第一个报文的确认后才发送第二个报文，这样就增加了一个 RTT 时延。

大量的 HTTP 请求的长度在 101~208 字节间。的确，在 13.4 节中我们讨论的 17 个请求的长度均在 152~197 字节间。这是因为客户的请求基本上都是一个固定的格式，从一个请求转换到另一个请求只是改变 URL。

要修补这个问题很简单（如果你有系统内核的源代码）。常量 `MINCLSIZE` 的值应从 208 改为 101。这就使得要写 101~208 字节时，不再使用两个 `mbuf`，而是把超过 100 字节的数据放入一个或多个 `mbuf` 串中。作了这个改变后，还可以摆脱在图 A-6 和 A-7 中 200 字节数据附近的尖峰现象。

图14-22(后面给出)中Tcpdump跟踪的客户就已经作了这个修补。如果没有进行这个修补,客户的第一个报文将只含有100字节,客户将为了等待这个报文的确认而花去一个RTT(慢启动),然后客户才发送剩余的52字节。只有在收到剩余的字节后,服务器才会发出第一个应答报文。

这里有一些其他的修补方法。第一种方法是:一个mbuf的大小可以由128字节增加至256字节。有些基于伯克利源码的系统已经作了这种修改(例如,AIX)。第二种:对sosend作修改,当使用多个mbuf时,避免多次调用TCP输出。

14.12 TCP的PCB高速缓存和首部预测

当Net/3收到一个报文时,它把指针保存在相应的Internet PCB(指向inpcb结构的tcp_last_inpcb指针,见卷2图28-5)中,并希望下个到达的报文还是属于同一条连接。这样做避免了查找TCP的PCB链表,而这样的查找的代价是昂贵的。每一次缓存比较失败,计数器tcps_pcbcachemiss就增加。在卷2图24-5的抽样统计中缓存的命中率接近80%,但被统计的系统不是一个HTTP服务器而是一个普通的分时系统。

当给定连接所接收的下一个报文不是下一个希望的ACK(在数据发送方),就是下一个希望的数据报文(在数据接收方)时,TCP的输入也执行一些首部预测(卷2第28.4节)。

在本章讨论的HTTP服务器上,我们观察到了下面的一些百分数:

- 20%的PCB缓存命中率(18~20%);
- 对下一个ACK报文的15%的首部预测率(14~15%);
- 对下一个数据报文的30%的首部预测率(20~35%)。

所有这些比率都是比较低的。两天中每个小时对这些百分数进行测量,发现它们的变化都很小:上面括号中列出了高低值的范围。

作为一个在同一时刻有大量不同客户使用TCP的HTTP服务器,PCB缓存命中率比较低并不让人感到奇怪。这种低的比率与HTTP是一个传输协议相适应,[McKenney and Dove 1992]中表明了Net/3的PCB缓存机制对事务协议不太有效。

通常一个HTTP服务器发送的数据报文比它接收的要多。图14-22是图13-5中客户的第一个HTTP请求的时间线(客户端口号为1114)。客户的请求是第4段报文,服务器的应答是第5、6、8、9、11、13和14段报文。这里,服务器只有一个可能的数据报文预测,那就是第4段报文。服务器的下一个可能的ACK报文预测是第7、10、12、15和16段报文(当第3段报文到达时,连接还没有完全建立好,第17段报文中FIN标志使程序不再对首部进行预测)。这些ACK报文究竟会不会限制依赖于窗口通告的首部预测,取决于客户端发送ACK时它读取了多少服务器返回的数据。例如在第7段报文,TCP确认了收到1024字节数据,但是HTTP客户应用程序只从插口缓存中读取了260字节数据(1024-8192+7428)。

当TCP的200 ms定时器超时,会发送一个延迟的ACK,它带有一个可笑的窗口通告。同样都是延迟的ACK,第7段与第12段报文时间上的差距是799 ms:4个TCP的200 ms时钟中断。这就暗示它们都是延迟的ACK,发送它们是因为时钟中断,而不是因为进程执行了新的、从插口缓存读数据的调用。第10段报文看上去好像也是延迟的ACK,因为它与第7段报文之间的时间为603 ms。

带有小的窗口通告的 ACK 报文的发送也会使首部预测失效，因为只有当窗口通告的值等于当前发送窗口的值时，才会执行首部预测。

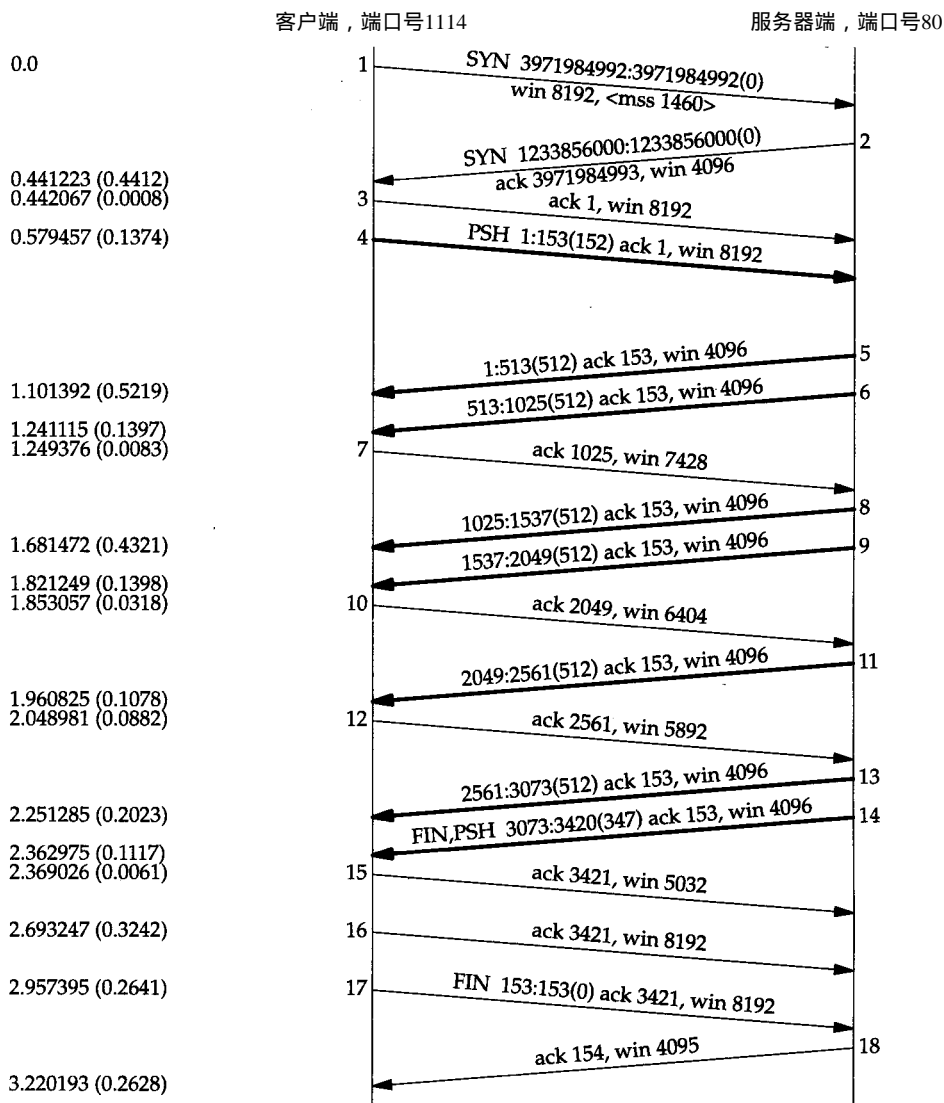


图14-22 HTTP客户-服务器事务

总的来说，我们对 HTTP 服务器上首部预测成功率低并不感到惊讶。在 TCP 连接上交换大量的数据时首部预测工作得最好。因为系统内核首部预测的统计是计算所有的 TCP 连接，我们只能猜测这台主机上对下一个数据报文的首部预测的高百分比（与对下一个 ACK 的预测相比）是来自非常长时间的 NNTP 连接（图 15-3），这种 NNTP 连接平均每条 TCP 连接接收约 1300 万字节。

慢启动错误

注意到图 14-22 中当服务器发送它的应答时没有像预期的那样发生慢启动。我们预期的是服务器先发送 512 字节的报文，等待客户的 ACK，然后发送下一个 512 字节的报文。而服务器

不是这样做的，它没有等待客户的 ACK 而是立即发送了两个 512 字节的报文 (第 5 段和第 6 段报文)。事实上这种现象在绝大多数伯克利的派生系统是很少见和异常的，因为许多应用程序都是由客户发送大多数数据给服务器。甚至对于 FTP 也是这样，例如，从一个 FTP 服务器上获取一个文件时，FTP 服务器打开一个数据传输连接，实际上成为了数据传输的客户端 (卷 1 图 27-7 给出了一个这样的例子)。

这个错误出在 `tcp_input` 函数上。新的连接启动时拥塞窗口为一个报文。当客户完成连接建立后 (卷 2 图 28-21)，代码执行转移到 `step6`，跳过了 ACK 的处理。当客户发送第一个数据段时，它的拥塞窗口是一个报文，这是不正确的。但是，当服务器完成连接建立后 (卷 2 图 29-2)，紧接着执行处理 ACK 的代码，收到 ACK 后拥塞窗口增加 1 个报文 (卷 2 图 29-7)。这就是为什么服务器一开始就连续发送两个报文。解决这个问题的办法是把图 11-16 中的代码加进去，不管这样是不是支持 T/TCP。

当服务器在第 7 段报文中收到 ACK 时，它的拥塞窗口增加至 3 个报文段，但接着服务器却只发送 2 个报文段 (第 8 和第 9 段报文)。我们不能从图 14-22 中找出原因来，因为我们只在连接的一端记录报文 (在客户端运行 `Tcpdump`)，第 10 段和第 11 段报文可能在网络中客户端与服务器端中间的什么地方。如果真是这样，那么服务器就的确像我们所预期的那样：拥塞窗口的宽度为 3 个报文段。

这些报文交互的线索是从对分组跟踪得来的 RTT 值。在客户端测量出来的第 1 段与第 2 段报文之间的 RTT 是 441 ms，第 4 段与第 5 段之间是 521 ms，第 7 段与第 8 段之间是 432 ms。这些都是可能的值，在客户端使用 Ping 程序 (指定分组长度为 300 字节) 也表明到这个服务器之间的 RTT 大约是 461 ms。但第 10 段与第 11 段报文之间的 RTT 非常小，只有 107 ms。

14.13 小结

通过运行一个繁忙的 Web 服务器来重点考察 TCP / IP 的实现。我们可以看到，服务器会收到 Internet 上各种各样的客户发来的一些奇怪的分组。

在本章中，我们对一个繁忙的 Web 服务器的分组进行跟踪，并对跟踪结果进行分析，着眼于各种实现中的特性。我们得到了如下结论：

- 客户端 SYN 的峰值到达速率约为平均到达速率的 8 倍 (忽略不正常的客户)。
- 客户到服务器之间的 RTT 平均值是 445 ms，中间值是 187 ms。
- 采用典型的 backlog 极限值 5 或 10 时，未完成连接队列很容易溢出。这个问题不是因为服务器进程太忙，而是因为客户的 SYN 至少要在队列中停留一个 RTT 时间。一个繁忙的 Web 服务器的这个队列需要比这大得多的容量。同时内核也提供一个计数器对这个队列的溢出次数进行统计，这样系统管理员就可以知道这种溢出发生的频度。
- 对阻塞在 LAST_ACK 状态的连接不断进行持续探测，因为这种情况经常出现，所以系统必须提供一种办法能让这种连接超时。
- 许多伯克利派生系统在客户请求报文的长度为 101~208 字节时 (通常许多客户均为这样) 使用 `mubf` 的效率比较低。
- 许多伯克利派生系统的实现提供 TCP PCB 高速缓存，同时绝大多数的系统也提供首部预测，但是它们对一个繁忙的 Web 服务器的帮助却很小。

[Mogul 1995d] 中提供了对另一个繁忙的 Web 服务器进行了相似的分析。