第22章 TCP的坚持定时器

22.1 引言

我们已经看到TCP通过让接收方指明希望从发送方接收的数据字节数（即窗口大小）来进行流量控制。如果窗口大小为0会发生什么情况呢？这将有效地阻止发送方传送数据，直到窗口变为非0为止。

可以在图20-3中看到这种情况。当发送方接收到报文段 9时，它打开被报文段 8关闭的窗口并立即开始发送数据。 TCP必须能够处理打开此窗口的ACK（报文段9）丢失的情况。 ACK的传输并不可靠，也就是说， TCP不对ACK报文段进行确认， TCP只确认那些包含有数据的ACK报文段。

如果一个确认丢失了，则双方就有可能因为等待对方而使连接终止：接收方等待接收数据（因为它已经向发送方通告了一个非 0的窗口），而发送方在等待允许它继续发送数据的窗口更新。为防止这种死锁情况的发生，发送方使用一个坚持定时器 (persist timer)来周期性地向接收方查询，以便发现窗口是否已增大。这些从发送方发出的报文段称为窗口探查 (window probe)。在本章中，我们将讨论窗口探查和坚持定时器，还将讨论与坚持定时器有关的糊涂窗口综合症。

22.2 一个例子

为了观察到实际中的坚持定时器，我们启动一个接收进程。它监听来自客户的连接请求，接受该连接请求，然后在从网上读取数据前休眠很长一段时间。

sock程序可以通过指定一个暂停选项 -P使服务器在接受连接和进行第一次读动作之间进入休眠。我们以这种方式调用服务器：

```
svr4 %sock -i -s -P100000 5555
```

该命令在从网络上读数据之前休眠 100 000秒（27.8小时）。客户运行在主机bsdi上，并向服务器的5555端口执行1024字节的写操作。图22-1给出了tcpdump的输出结果（我们已经在结果中去掉了连接的建立过程）。

报文段1~13显示的是从客户到服务器的正常的数据传输过程，有 9216个字节的数据填充了窗口。服务器通告窗口大小为 4096字节，且默认的插口缓存大小为 4096字节。但实际上它一共接收了9216字节的数据，这是在SVR4中TCP代码和流子系统(stream subsystem)之间某种形式交互的结果。

在报文段13中，服务器确认了前面 4个数据报文段，然后通告窗口为 0，从而使客户停止发送任何其他的数据。这就引起客户设置其坚持定时器。如果在该定时器时间到时客户还没有接收到一个窗口更新，它就探查这个空的窗口以决定窗口更新是否丢失。由于服务器进程处于休眠状态，所以TCP缓存9216字节的数据并等待应用进程读取。

请注意客户发出的窗口探查之间的时间间隔。在收到一个大小为 0的窗口通告后的第1个

（报文段14）间隔为4.949秒，下一个（报文段16）间隔是4.996秒，随后的间隔分别约为6, 12, 24, 48和60秒。

```
 1    0.0                bsdi.1027 > svr4.5555: P 1:1025(1024) ack 1 win 4096
 2    0.191961 ( 0.1920) svr4.5555 > bsdi.1027: . ack 1025 win 4096
 3    0.196950 ( 0.0050) bsdi.1027 > svr4.5555: . 1025:2049(1024) ack 1 win 4096
 4    0.200340 ( 0.0034) bsdi.1027 > svr4.5555: . 2049:3073(1024) ack 1 win 4096
 5    0.207506 ( 0.0072) svr4.5555 > bsdi.1027: . ack 3073 win 4096
 6    0.212676 ( 0.0052) bsdi.1027 > svr4.5555: . 3073:4097(1024) ack 1 win 4096
 7    0.216113 ( 0.0034) bsdi.1027 > svr4.5555: P 4097:5121(1024) ack 1 win 4096
 8    0.219997 ( 0.0039) bsdi.1027 > svr4.5555: P 5121:6145(1024) ack 1 win 4096
 9    0.227882 ( 0.0079) svr4.5555 > bsdi.1027: . ack 5121 win 4096
10    0.233012 ( 0.0051) bsdi.1027 > svr4.5555: P 6145:7169(1024) ack 1 win 4096
11    0.237014 ( 0.0040) bsdi.1027 > svr4.5555: P 7169:8193(1024) ack 1 win 4096
12    0.240961 ( 0.0039) bsdi.1027 > svr4.5555: P 8193:9217(1024) ack 1 win 4096
13    0.402143 ( 0.1612) svr4.5555 > bsdi.1027: . ack 9217 win 0

14    5.351561 ( 4.9494) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
15    5.355571 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

16   10.351714 ( 4.9961) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
17   10.355670 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

18   16.351881 ( 5.9962) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
19   16.355849 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

20   28.352213 (11.9964) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
21   28.356178 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

22   52.352874 (23.9967) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
23   52.356839 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

24  100.354224 (47.9974) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
25  100.358207 ( 0.0040) svr4.5555 > bsdi.1027: . ack 9217 win 0

26  160.355914 (59.9977) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
27  160.359835 ( 0.0039) svr4.5555 > bsdi.1027: . ack 9217 win 0

28  220.357575 (59.9977) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
29  220.361668 ( 0.0041) svr4.5555 > bsdi.1027: . ack 9217 win 0

30  280.359254 (59.9976) bsdi.1027 > svr4.5555: . 9217:9218(1) ack 1 win 4096
31  280.363315 ( 0.0041) svr4.5555 > bsdi.1027: . ack 9217 win 0
```

图22-1 坚持定时器探查一个0大小窗口的例子

为什么这些间隔总是比5、6、12、24、48和60小一个零点几秒呢？因为这些探查被TCP的500 ms定时器超时例程所触发。当定时器时间到时，就发送窗口探查，并大约在4 ms之后收到一个应答。接收到应答使得定时器被重新启动，但到下一个时钟滴答之间的时间则约为500减4 ms。

计算坚持定时器时使用了普通的TCP指数退避。对一个典型的局域网连接，首次超时时间算出来是1.5秒，第2次的超时值增加一倍，为3秒，再下次乘以4为6秒，之后再乘以8为12秒等。但是坚持定时器总是在5~60秒之间，这与我们在图22-1中观察到的现象一致。

窗口探查包含一个字节的数据（序号为9217）。TCP总是允许在关闭连接前发送一个字节的数据。请注意，尽管如此，所返回的窗口为0的ACK并不是确认该字节（它们确认了包括9216在内的所有数据），因此这个字节被持续重传。

坚持状态与第21章中介绍的重传超时之间一个不同的特点就是TCP从不放弃发送窗口探查。这些探查每隔60秒发送一次，这个过程将持续到或者窗口被打开，或者应用进程使用的连接被终止。

22.3 糊涂窗口综合症

基于窗口的流量控制方案，如TCP所使用的，会导致一种被称为“糊涂窗口综合症SWS

(Silly Window Syndrome)”的状况。如果发生这种情况，则少量的数据将通过连接进行交换，而不是满长度的报文段[Clark 1982]。

该现象可发生在两端中的任何一端：接收方可以通告一个小的窗口（而不是一直等到有大的窗口时才通告），而发送方也可以发送少量的数据（而不是等待其他的数据以便发送一个大的报文段）。可以在任何一端采取措施避免出现糊涂窗口综合症的现象。

1) 接收方不通告小窗口。通常的算法是接收方不通告一个比当前窗口大的窗口(可以为0)，除非窗口可以增加一个报文段大小（也就是将要接收的 MSS）或者可以增加接收方缓存空间的一半，不论实际有多少。

2) 发送方避免出现糊涂窗口综合症的措施是只有以下条件之一满足时才发送数据： (a)可以发送一个满长度的报文段； (b)可以发送至少是接收方通告窗口大小一半的报文段； (c)可以发送任何数据并且不希望接收 ACK（也就是说，我们没有还未被确认的数据）或者该连接上不能使用Nagle算法（见第19.4节）。

条件(b)主要对付那些总是通告小窗口（也许比 1个报文段还小）的主机，条件 (c)使我们在有尚未被确认的数据（正在等待被确认）以及在不能使用 Nagle算法的情况下，避免发送小的报文段。如果应用进程在进行小数据的写操作（例如比该报文段还小），条件(c)可以避免出现糊涂窗口综合症。

这三个条件也可以让我们回答这样一个问题：在有尚未被确认数据的情况下，如果 Nagle算法阻止我们发送小的报文段，那么多小才算是小呢？从条件 (a)中可以看出所谓“小”就是指字节数小于报文段的大小。条件 (b)仅用来对付较老的、原始的主机。

步骤2中的条件(b)要求发送方始终监视另一方通告的最大窗口大小，这是一种发送方猜测对方接收缓存大小的企图。虽然在连接建立时接收缓存的大小可能会减小，但在实际中这种情况很少见。

一个例子

现在我们通过仔细查看一个详细的例子来观察实际避免出现糊涂窗口综合症的情况，该例子也包括了坚持定时器。我们将在发送主机 sun上运行sock程序，并向网络写 6个1024字节的数据。

```
sun % sock -i -n6 bsdi 7777
```

但是在主机bsdi的接收过程中我们加入一些暂停。在第 1次读数据前暂停4秒，之后每次读之前暂停2秒。而且，接收方进行的是256字节的读操作：

```
bsdi %sock -i -s -P4 -p2 -r256 7777
```

最初的暂停是为了让接收缓存被填满，迫使发送方停止发送。随后由于接收方从网络上进行了一些小数目的读取，我们预期能看到接收方采取的避免糊涂窗口综合症的措施。

图22-2是传输6144字节数据的时间系列（我们去掉了连接建立过程）。

我们还需要跟踪在每个时间点上读取数据时应用程序的运行情况、当前正在接收缓存中的数据的序号以及接收缓存中可用空间的大小。图 22-3显示了所发生的每件事情。

图22-3中的第1列是每个行为的相对时间点。那些带有3位小数点的时间是从tcpdump的输出结果（图22-2）中得到的，而小数点部分为 99的则是在接收服务器上产生行为的估计时间(使这些在接收方的估计时间包含一秒的99%仅与图22-2中的报文段20和22有关，它们是我们能

够从tcpdump的输出结果中看到的由接收主机超时引起的仅有的两个事件。而在主机 bsdi上观察到的其他分组，则是由接收到来自发送方的一个报文段所引起的。这同样是有意义的，因为这就使我们可以将最初的4秒暂停刚好放置在发送方发送第1个数据报文段的时间0前面。这是接收方在连接建立过程中收到它的SYN的ACK之后将要获得控制权的大致时间)。

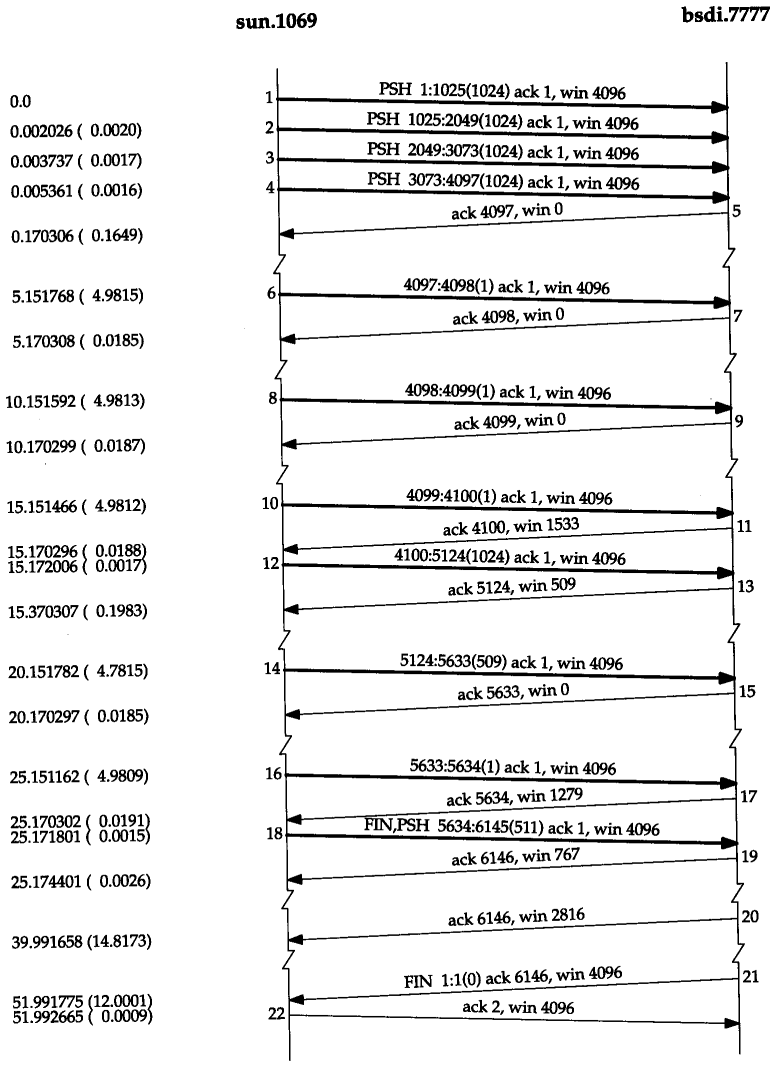


图22-2 显示接收方避免出现糊涂窗口综合症的时间系列

当接收到来自发送方的数据时，接收方缓存中的数据增加，而当应用进程从缓存中读取数据时，数据就减少。接下来我们关注的是接收方发给发送方的窗口通告以及这些窗口通告是什么。这样就可以使我们看到接收方是如何避免糊涂窗口综合症的。

前4个数据报文段及其ACK（报文段1~5）表示发送方正在填充接收方的缓存。在那个时刻发送方停止了发送，但仍然有更多的数据需要发送。它将自己的坚持定时器置为最小值5分钟。

当坚持定时器时间到时，就发送出1个字节的数据（报文段6）。接收的应用进程已经从接收缓存中读取了256字节的数据（在时刻3.99），因此这个字节被接受并被确认（报文段7段）。但是通告窗口仍为0，由于接收方仍然没有足够的空间来接收一个满长度的报文，或者不能腾

出缓存空间的一半。这就是接收方的糊涂窗口避免措施。

时间	报文段号 (图22-2)	行为			接收缓冲区	
		发送TCP	接收TCP	应用	数据	可用的
0.000	1	1:1025(1024)			1024	3072
0.002	2	1025:2049(1024)			2048	2048
0.003	3	2049:3073(1024)			3072	1024
0.005	4	3073:4097(1024)			4096	0
0.170	5		ACK 4097, win 0			
3.99				读取256	3840	256
5.151	6	4097:4098(1)			3841	255
5.17	7		ACK 4098, win 0			
5.99				读取256	3585	511
7.99				读取256	3329	767
9.99				读取256	3073	1023
10.151	8	4098:4099(1)			3074	1022
10.170	9		ACK 4099, win 0			
11.99				读取256	2818	1278
13.99				读取256	2562	1534
15.151	10	4099:4100(1)			2563	1533
15.170	11		ACK 4100, win 1533			
15.172	12	4100:5124(1024)			3587	509
15.370	13		ACK 5124, win 509			
15.99				读取256	3331	765
17.99				读取256	3075	1021
19.99				读取256	2819	1277
20.151	14	5124:5633(509)			3328	768
20.170	15		ACK 5633, win 0			
21.99				读取256	3072	1024
23.99				读取256	2816	1280
25.151	16	5633:5634(1)			2817	1279
25.170	17		ACK 5634, win 1279			
25.171	18	5634:6145(511)			3328	768
25.174	19		ACK 6146, win 767			
25.99				读取256	3072	1024
27.99				读取256	2816	1280
29.99				读取256	2560	1536
31.99				读取256	2304	1792
33.99				读取256	2048	2048
35.99				读取256	1792	2304
37.99				读取256	1536	2560
39.99				读取256	1280	2816
39.99	20		ACK 6146, win 2816			
41.99				读取256	1024	3072
43.99				读取256	768	3328
45.99				读取256	512	3584
47.99				读取256	256	3840
49.99				读取256	0	4096
51.99				读取256(EOF)	0	4096
51.991	21		ACK 6146, win 4096			
51.992	22	ACK 2				

图22-3 接收方避免出现糊涂窗口综合症的事件序列

发送方的坚持定时器被复位，并在5秒后再次到时（在时刻10.151）。然后又发送一个字节并被确认（报文段8和9），而接收方的缓存空间还不够用（1022字节），使得通告窗口为0。

发送方的坚持定时器在时刻 15.151再次时间到，又发送了另一个字节并被确认（报文段10和11）。这一次由于接收方有 1533字节的有效缓存空间，因此通告了一个非 0窗口。发送方立即利用这个窗口发送了1024字节的数据（报文段12）。对这1024字节数据的确认（报文段13）通告其窗口为509字节。这看起来与我们在前面看到的小窗口通告相抵触。

在这里之所以发生这种情况，是因为报文段 11段通告了一个大小为 1533字节的窗口，而发送方只使用了其中的1024字节。如果在报文段13中的ACK通告其窗口为0，就会违反窗口的右边沿不能向左边沿移动而导致窗口收缩的 TCP原则（见第20.3节）。这就是为什么必须通告

一个509字节的窗口的原因。

接下来我们看到发送方没有立即向这个小窗口发送数据。这就是发送方采取的糊涂窗口避免策略。相反，它等待另一个坚持定时器在时刻 20.151到时间，并在该时刻发送 509字节的数据。尽管它最终还是发送了一个长度为 509字节的小数据段，但在发送前它等待了 5秒钟，看是否会有一个ACK到达，以便可以将窗口开得更大。这 509字节的数据使得接收缓存仅剩下768字节的有效空间，因此接收方通告窗口为 0（报文段15）。

坚持定时器在时刻25.151再次到时间，发送方发送 1个字节，于是接收缓存中有 1279字节的可用空间，这就是在报文段 17所通告的窗口大小。

发送方只有另外的511个字节的数据需要发送，因此在收到 1279的窗口通告后立刻发送了这些数据（报文段 18）。这个报文段也带有 FIN标志。接收方确认数据和 FIN，并通告窗口大小为767（见习题22.2）。

由于发送应用进程在执行完 6个1024字节的写操作后发出关闭命令，发送方的连接从ESTABLISHED状态转变到FIN_WAIT_1状态，再到FIN_WAIT_2状态（见图18-12）。它一直处于这个状态，直到收到对方的 FIN。在这个状态上没有设置定时器（回忆我们在 18.6节结束时的讨论），因为它在报文段18中发送的FIN被报文段19确认。这就是为什么我们看到发送方直到接收到FIN（报文段21）为止没有发送其他任何数据的原因。

接收应用进程继续每隔 2秒从接收缓存区中读取 256个字节的数据。为什么在时刻 39.99发送ACK（报文段20）呢？这是因为应用进程在时刻 39.99读取数据时，接收缓存中的可用空间已经从原来通告的767（报文段19）变为2816，这相当于接收缓存中增加了额外的 2049字节的空间。回忆本节开始讲的第 1个规则，因为现在接收缓存已经增加了其空间的一半，因此接收方现在发送窗口更新。这意味着每次当应用进程从 TCP的接收缓存中读取数据时，接收的 TCP将检查是否需要更新发送窗口。

应用进程在时间51.99发出最后一个读操作，然后收到一个文件结束标志，因为缓存已经变空。这就导致了最后两个完成连接终止的报文段（报文段 21和22）的发送。

22.4 小结

在连接的一方需要发送数据但对方已通告窗口大小为0时，就需要设置TCP的坚持定时器。发送方使用与第21章类似的重传间隔时间，不断地探查已关闭的窗口。这个探查过程将一直持续下去。

当运行一个例子来观察坚持定时器时，我们还观察到了 TCP的避免出现糊涂窗口综合症的现象。这就是使 TCP避免通告小的窗口大小或发送小的报文段。在我们的例子中，可以观察到发送方和接收方为避免糊涂窗口综合症所使用的策略。

习题

22.1 在图22-3中注意到所有确认（报文段5、7、9、11、13、15和17）的发送时刻为：0.170, 5.170、10.170、15.170、20.170和25.170，还注意到在接收数据和发送ACK之间的时间差分别为：164.5、18.5、18.7、18.8、198.3、18.5和19.1 ms。试解释可能会发生的情况。

22.2 在图22-3中的时刻25.174，发送出一个767字节的通告窗口，而在接收缓存中有 768字节的可用空间。为什么相差1个字节？

第23章 TCP的保活定时器

23.1 引言

许多TCP/IP的初学者会很惊奇地发现可以没有任何数据流通过一个空闲的 TCP连接。也就是说，如果TCP连接的双方都没有向对方发送数据，则在两个TCP模块之间不交换任何信息。例如，没有可以在其他网络协议中发现的轮询。这意味着我们可以启动一个客户与服务器建立一个连接，然后离去数小时、数天、数个星期或者数月，而连接依然保持。中间路由器可以崩溃和重启，电话线可以被挂断再连通，但是只要两端的主机没有被重启，则连接依然保持建立。

这意味着两个应用进程 —— 客户进程或服务器进程 —— 都没有使用应用级的定时器来检测非活动状态，而这种非活动状态可以导致应用进程中的任何一个终止其活动。回想在第10.7节末尾曾提到过的BGP每隔30秒就向对端发送一个应用的探查，就是独立于 TCP的保活定时器之外的应用定时器。

然而，许多时候一个服务器希望知道客户主机是否崩溃并关机或者崩溃又重新启动。许多实现提供的保活定时器可以提供这种能力。

> 保活并不是TCP规范中的一部分。Host Requirements RFC提供了3个不使用保活定时器的理由：(1) 在出现短暂差错的情况下，这可能会使一个非常好的连接释放掉；(2) 它们耗费不必要的带宽；(3) 在按分组计费的情况下会在互联网上花掉更多的钱。然而，许多实现提供了保活定时器。

保活定时器是一个有争论的功能。许多人认为如果需要，这个功能不应该在 TCP中提供，而应该由应用程序来完成。这是应当认真对待的一些问题之一，因为在这个论题上有些人表达出了很大的热情。

在连接两个端系统的网络出现临时故障的时候，保活选项会引起一个实际上很好的连接终止。例如，如果在一个中间路由器崩溃并重新启动时发送保活探查，那么 TCP会认为客户的主机已经崩溃，而实际上所发生的并非如此。

保活功能主要是为服务器应用程序提供的。服务器应用程序希望知道客户主机是否崩溃，从而可以代表客户使用资源。许多版本的 Rlogin和Telnet服务器默认使用这个选项。

一个说明现在需要使用保活功能的常见例子是当个人计算机用户使用 TCP/IP向一个使用Telnet的主机注册时。如果在一天结束时，他们仅仅关闭了电源而没有注销，那么便会留下一个半开放的连接。在图18-16中，我们看到通过一个半开放连接发送数据会导致返回一个复位，但那是在来自正在发送数据的客户端。如果客户已经消失了，使得在服务器上留下一个半开放连接，而服务器又在等待来自客户的数据，则服务器将永远等待下去。保活功能就是试图在服务器端检测到这种半开放的连接。

23.2 描述

在这个描述中，我们称使用保活选项的一端为服务器，而另一端则为客户。并没有什么使客户不能使用这个选项，但通常都是服务器设置这个功能。如果双方都特别需要了解对方是否已经消失，则双方都可以使用这个选项（在29章我们将看到NFS使用TCP时，客户和服务器都设置了这个选项。但在第26章讲到Telnet和Rlogin时，只有服务器设置了这个选项，而客户则没有）。

如果一个给定的连接在两个小时之内没有任何动作，则服务器就向客户发送一个探查报文段（我们将在随后的例子中看到这个探查报文段看起来像什么）。客户主机必须处于以下4个状态之一。

1) 客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常工作的。服务器在两小时以后将保活定时器复位。如果在两个小时定时器到时间之前有应用程序的通信量通过此连接，则定时器在交换数据后的未来2小时再复位。

2) 客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务器将不能够收到对探查的响应，并在75秒后超时。服务器总共发送10个这样的探查，每个间隔75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。

3) 客户主机崩溃并已经重新启动。这时服务器将收到一个对其保活探查的响应，但是这个响应是一个复位，使得服务器终止这个连接。

4) 客户主机正常运行，但是从服务器不可达。这与状态2相同，因为TCP不能够区分状态4与状态2之间的区别，它所能发现的就是没有收到探查的响应。

服务器不用关注客户主机被关闭和重新启动的情况（这指的是一个操作员的关闭，而不是主机崩溃）。当系统被操作员关闭时，所有的应用进程也被终止（也就是客户进程），这会使客户的TCP在连接上发出一个FIN。接收到FIN将使服务器的TCP向服务器进程报告文件结束，使服务器可以检测到这个情况。

在第1种情况下，服务器的应用程序没有感觉到保活探查的发生。TCP层负责一切。这个过程对应用程序都是透明的，直至第2、3或4种情况发生。在这三种情况下，服务器应用程序将收到来自它的TCP的差错报告（通常服务器已经向网络发出了读操作请求，然后等待来自客户的数据。如果保活功能返回一个差错，则该差错将作为读操作的返回值返回给服务器）。在第2种情况下，差错是诸如“连接超时”之类的信息，而在第3种情况则为“连接被对方复位”。第4种情况看起来像是连接超时，也可根据是否收到与连接有关的ICMP差错来返回其他的差错。在下一节中我们将观察这4种情况。

一个被人们不断讨论的关于保活选项的问题就是两个小时的空闲时间是否可以改变。通常他们希望该数值可以小得多，处在分钟的数量级。正如我们在附录E看到的，这个值通常可以改变，但是在该附录所描述的所有系统中，保活间隔时间是系统级的变量，因此改变它会影响到所有使用该功能的用户。

Host Requirements RFC提到一个实现可提供保活的功能，但是除非应用程序指明要这样，否则就不能使用该功能。而且，保活间隔必须是可配置的，但是其默认值必须不小于两个小时。

23.3 保活举例

现在详细讨论前一节提到的第2、3和4种情况。我们将在使用这个选项的情况下检查所交换的分组。

23.3.1 另一端崩溃

首先观察另一端崩溃且没有重新启动的情况下所发生的现象。为模拟这种情况，我们采用如下步骤：

- 在客户（主机bsdi上运行的sock程序）和主机svr4上的标准回显服务器之间建立一个连接。客户使用-K选项使能保活功能。
- 验证数据可以通过该连接。
- 观察客户TCP每隔2小时发送保活分组，并观察被服务器的TCP确认。
- 将以太网电缆从服务器上拔掉直到这个例子完成，这会使客户认为服务器主机已经崩溃。
- 我们预期服务器在断定连接已中断前发送10个间隔为75秒的保活探查。

这里是客户端的交互输出结果：

```
bsdi %sock -K svr4 echo          -K是保活选项
hello, world                     开始时键入本行以验证连接有效
hello, world                     和看到回显
                                 4小时后断开以太网电缆
read error: Connection timed out这发生在启动后约6小时10分钟
```

图23-1显示的是tcpdump的输出结果（已经去掉了连接建立和窗口通告）。

```
 1      0.0                  bsdi.1055 > svr4.echo: P 1:14(13) ack 1
 2      0.006105 (   0.0061) svr4.echo > bsdi.1055: P 1:14(13) ack 14
 3      0.093140 (   0.0870) bsdi.1055 > svr4.echo: . ack 14

 4   7199.972793 (7199.8797) arp who-has svr4 tell bsdi
 5   7199.974878 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
 6   7199.975741 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
 7   7199.979843 (   0.0041) svr4.echo > bsdi.1055: . ack 14

 8  14400.134330 (7200.1545) arp who-has svr4 tell bsdi
 9  14400.136452 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
10  14400.137391 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
11  14400.141408 (   0.0040) svr4.echo > bsdi.1055: . ack 14

12  21600.318309 (7200.1769) arp who-has svr4 tell bsdi
13  21675.320373 (  75.0021) arp who-has svr4 tell bsdi
14  21750.322407 (  75.0020) arp who-has svr4 tell bsdi
15  21825.324460 (  75.0021) arp who-has svr4 tell bsdi
16  21900.436749 (  75.1123) arp who-has svr4 tell bsdi
17  21975.438787 (  75.0020) arp who-has svr4 tell bsdi
18  22050.440842 (  75.0021) arp who-has svr4 tell bsdi
19  22125.432883 (  74.9920) arp who-has svr4 tell bsdi
20  22200.434697 (  75.0018) arp who-has svr4 tell bsdi
21  22275.436788 (  75.0021) arp who-has svr4 tell bsdi
```

图23-1 决定一个主机已经崩溃的保活分组

客户在第1、2和3行向服务器发送“Hello, world”并得到回显。第4行是第一个保活探查，发生在两个小时以后（7200秒）。在第6行的TCP报文段能够发送之前，首先观察到的是一个ARP请求和一个ARP应答。第6行的保活探查引出来自另一端的响应（第7行）。两个小时以后，在第7和8行发生了同样的分组交换过程。

如果能够观察到第6和第10行的保活探查中的所有字段，我们就会发现序号字段比下一个将要发送的序号字段小1（在本例中，当下一个为14时，它就是13）。但是因为报文段中没有数据，tcpdump不能打印出序号字段（它仅能够打印出设置了SYN、FIN或RST标志的空数据的序号）。正是接收到这个不正确的序号，才导致服务器的TCP对保活探查进行响应。这个响应告诉客户，服务器下一个期望的序号是14。

一些基于4.2BSD的旧的实现不能够对这些保活探查进行响应，除非报文段中包含数据。某些系统可以配置成发送一个字节的无用数据来引出响应。这个无用数据是无害的，因为它不是所期望的数据（这是接收方前一次接收并确认的数据），因此它会被接收方丢弃。其他一些系统在探查的前半部分发送4.3BSD格式的报文段（不包含数据），如果没有收到响应，在后半部分则切换为4.2BSD格式的报文段。

接着我们拔掉电缆，并期望两个小时的再一次探查失败。当这下一个探查发生时，注意到从来没有看到电缆上出现TCP报文段，这是因为主机没有响应ARP请求。在放弃之前，我们仍可以观察到客户每隔75秒发送一个探查，一共发送了10次。从交互式脚本可以看到返回给客户进程的差错码被TCP转换为“连接超时”，这正是实际所发生的。

23.3.2 另一端崩溃并重新启动

在这个例子中，我们可以观察到当客户崩溃并重新启动时发生的情况。最初的环境与前一个例子相似，但是在我们验证连接有效之后，我们将服务器从以太网上断开，重新启动，然后再连接到网络上。我们希望看到下一个保活探查产生一个来自服务器的复位，因为现在服务器不知道关于这个连接的任何信息。这是交互会话的过程：

```
bsdi %sock -K svr4 echo              -K使保活选项有效
hi, there                            键入这行以验证连接有效
hi, there                            这是来自另一端的回显
                                     从以太网断连后，服务器这时重新启动
read error: Connection reset by peer
```

图23-2显示的是tcpdump的输出结果（已经去掉了连接建立和窗口通告）。

```
1     0.0                    bsdi.1057 > svr4.echo: P 1:10(9) ack 1
2     0.006406 (   0.0064)   svr4.echo > bsdi.1057: P 1:10(9) ack 10
3     0.176922 (   0.1705)   bsdi.1057 > svr4.echo: . ack 10

4  7200.067151 (7199.8902)   arp who-has svr4 tell bsdi
5  7200.069751 (   0.0026)   arp reply svr4 is-at 0:0:c0:c2:9b:26
6  7200.070468 (   0.0007)   bsdi.1057 > svr4.echo: . ack 10
7  7200.075050 (   0.0046)   svr4.echo > bsdi.1057: R 1135563275:1135563275(0)
```

图23-2 另一端崩溃并重启时保活的例子

我们建立了连接，并从客户发送9个字节的数据到服务器（第1~3行）。两个小时之后，客户发送第1个保活探查，其响应是一个来自服务器的复位。客户应用进程打印出“连接被对端复位”的差错，这是有意义的。

23.3.3 另一端不可达

在这个例子中，客户没有崩溃，但是在保活探查发送后的10分钟内无法到达，可能是一个中间路由器已经崩溃，或一条电话线临时出现故障，或发生了其他一些类似的情况。

为了仿真这个例子，我们从主机 slip经过一个拨号 SLIP链路与主机 vangogh.cs.berkeley.edu建立一个连接，然后断掉链路。这里是交互输出的结果：

```
bsdi %sock -K vangogh.cs.berkeley.edu echo
testing                                          我们键入这行
testing                                          看到这行的回显
                                                 在某个时刻这条SLIP链路被断开
read error: No route to host
```

图23-3显示了在路由器bsdi上收集到的tcpdump输出结果（已经去掉了连接建立和窗口通告）。

```
 1      0.0                  slip.1056 > vangogh.echo: P 1:9(8) ack 1
 2      0.277669 (   0.2777) vangogh.echo > slip.1056: P 1:9(8) ack 9
 3      0.424423 (   0.1468) slip.1056 > vangogh.echo: . ack 9

 4   7200.818081 (7200.3937) slip.1056 > vangogh.echo: . ack 9
 5   7201.243046 (   0.4250) vangogh.echo > slip.1056: . ack 9

 6  14400.688106 (7199.4451) slip.1056 > vangogh.echo: . ack 9
 7  14400.689261 (   0.0012) sun > slip: icmp: net vangogh unreachable

 8  14475.684360 (  74.9951) slip.1056 > vangogh.echo: . ack 9
 9  14475.685504 (   0.0011) sun > slip: icmp: net vangogh unreachable

                        删除14行

24  15075.759603 (  75.1008) slip.1056 > vangogh.echo: R 9:9(0) ack 9
25  15075.760761 (   0.0012) sun > slip: icmp: net vangogh unreachable
```

图23-3 当另一端不可达时的保活例子

我们与以前一样开始讨论这个例子：第 1~3行证实连接是有效的。两个小时之后的第 1个保活探查是正常的（第4、5行），但是在两个小时后发生下一个探查之前，我们断开在路由器sun和netb之间的SLIP连接（拓扑结构参见封）。

第6行的保活探查引发一个来自路由器 sun的ICMP网络不可达的差错。正如我们在第21.10节描述的那样，对于主机slip上接收的TCP而言，这只是一个软差错。它报告收到了一个ICMP差错，但是差错的接收者并没有终止这个连接。在发送主机最终放弃之前，一共发送了9个保活探查，间隔为 75秒。这时返回给应用进程的差错产生了一个不同的报文：“ 没有到达主机的路由 ”。我们在图6-12看到这对应于ICMP网络不可达的差错。

23.4 小结

正如我们在前面提到的，对保活功能是有争议的。协议专家继续在争论该功能是否应该归入运输层，或者应当完全由应用层来处理。

在连接空闲两个小时后，在一个连接上发送一个探查分组来完成保活功能。可能会发生 4种不同的情况：对端仍然运行正常、对端已经崩溃、对端已经崩溃并重新启动以及对端当前无法到达。我们使用一个例子来观察每一种情况，并观察到在最后三个条件下返回的不同差错。

在前两个例子中，如果没有提供这种功能，并且也没有应用层的定时器，则客户将永远无法知道对端已经崩溃或崩溃并重新启动。可是在最后一个例子中，两端都没有发生差错，只是它们之间的连接临时中断。我们在使用保活时必须关注这个限制。

习题

23.1 列出保活功能的一些优点。

23.2 列出保活功能的一些缺点。

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的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对正在发送的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`traceroute`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。

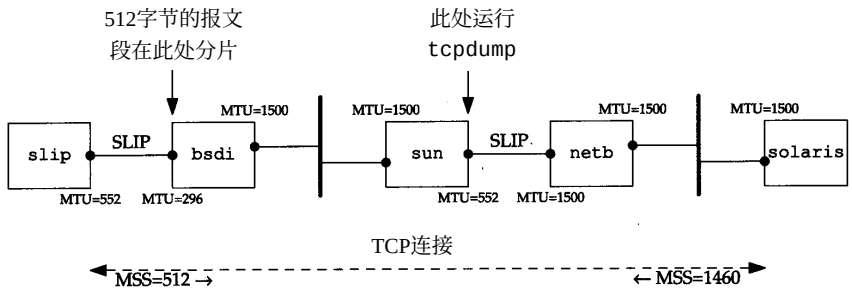


图24-1 路径MTU例子的拓扑结构

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（1 544 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

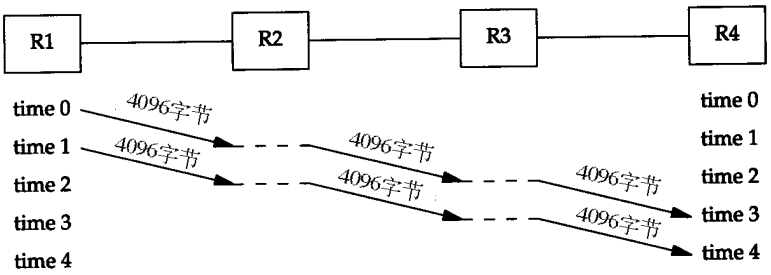


图24-3 通过4个路由器发送两个4096字节的分组

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

(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。

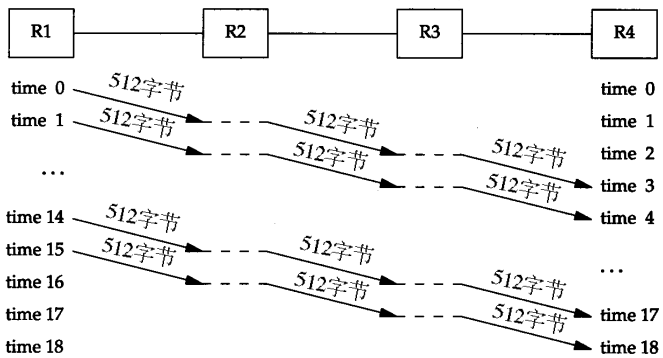


图24-4 通过4个路由器发送16个512字节的分组

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（18×2.9）=52.2 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

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

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)
以太局域网	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迟延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。

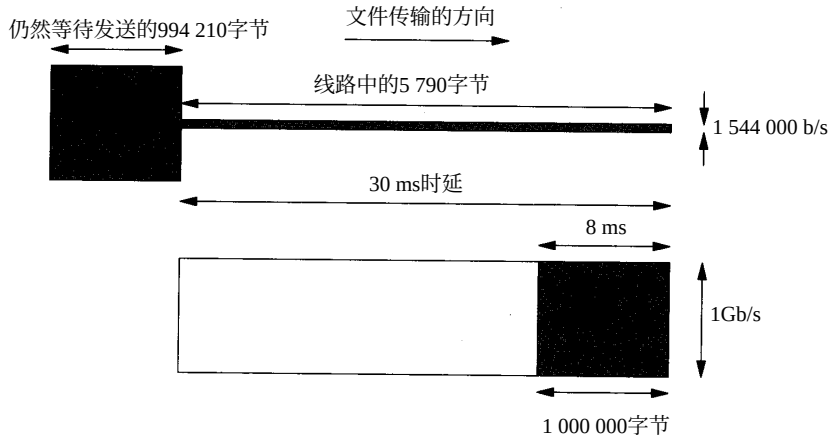


图24-6 以30 ms的延时通过网络发送100万字节的文件

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25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。一个字节的移位记数器取值为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记数器置为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                          我们键入这一行
hello, world                          此处是它的回显
^D                                    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                              我们键入这一行
bye, bye                              此处是它的回显
^D                                    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$2^{16}\times2^{14}$，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比为这个客户缓存的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：

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

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：

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

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。我们观察到中间的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？

第25章 SNMP：简单网络管理协议

25.1 引言

随着网络技术的飞速发展，网络的数量也越来越多。而网络中的设备来自各个不同的厂家，如何管理这些设备就变得十分重要。本章的内容就是介绍管理这些设备的标准。

基于TCP/IP的网络管理包含两个部分：网络管理站（也叫管理进程， manager）和被管的网络单元（也叫被管设备）。被管设备种类繁多，例如：路由器、 X 终端、终端服务器和打印机等。这些被管设备的共同点就是都运行 TCP/IP协议。被管设备端和管理相关的软件叫做代理程序(agent)或代理进程。管理站一般都是带有彩色监视器的工作站，可以显示所有被管设备的状态(例如连接是否掉线、各种连接上的流量状况等)。

管理进程和代理进程之间的通信可以有两种方式。一种是管理进程向代理进程发出请求，询问一个具体的参数值（例如：你产生了多少个不可达的 ICMP端口？）。另外一种方式是代理进程主动向管理进程报告有某些重要的事件发生（例如：一个连接口掉线了）。当然，管理进程除了可以向代理进程询问某些参数值以外，它还可以按要求改变代理进程的参数值（例如：把默认的IP TTL值改为64）。

基于TCP/IP的网络管理包含3个组成部分：

1) 一个管理信息库MIB（Management Information Base）。管理信息库包含所有代理进程的所有可被查询和修改的参数。 RFC 1213 [McCloghrie and Rose 1991]定义了第二版的MIB，叫做MIB-II。

2) 关于MIB的一套公用的结构和表示符号。叫做管理信息结构 SMI（Structure of Management Information）。这个在RFC 1155 [Rose and McCloghrie 1990] 中定义。例如：SMI定义计数器是一个非负整数，它的计数范围是 0~4 294 967 295，当达到最大值时，又从0开始计数。

3) 管理进程和代理进程之间的通信协议，叫做简单网络管理协议 SNMP（Simple Network Management Protocol）。在RFC 1157 [Case et al. 1990]中定义。SNMP包括数据报交换的格式等。尽管可以在运输层采用各种各样的协议，但是在 SNMP中，用得最多的协议还是UDP。

上面提到的RFC所定义的SNMP叫做SNMP v1，或者就叫做SNMP，这也是本章的主要内容。到1993年为止，又有一些新的关于SNMP的 RFC发表。在这些RFC中定义的SNMP叫做第二版SNMP（SNMP v2），这将在25.12章节中讨论。

本章首先介绍管理进程和代理进程之间的协议，然后讨论参数的数据类型。在本章中将用到前面已经出现过的名词，如： IP、UDP和TCP等。我们在叙述中将举一些例子来帮助读者理解，这些例子和前面的某些章节相关。

25.2 协议

关于管理进程和代理进程之间的交互信息， SNMP定义了5种报文：

1) get-request操作：从代理进程处提取一个或多个参数值。

2) get-next-request操作：从代理进程处提取一个或多个参数的下一个参数值（关于“下一个（next）”的含义将在后面的章节中介绍）。

3) set-request操作：设置代理进程的一个或多个参数值。

4) get-response操作：返回的一个或多个参数值。这个操作是由代理进程发出的。它是前面3中操作的响应操作。

5) trap 操作：代理进程主动发出的报文，通知管理进程有某些事情发生。

前面的3个操作是由管理进程向代理进程发出的。后面两个是代理进程发给管理进程的（为简化起见，前面3个操作今后叫做get、get-next和set操作）。图25-1描述了这5种操作。

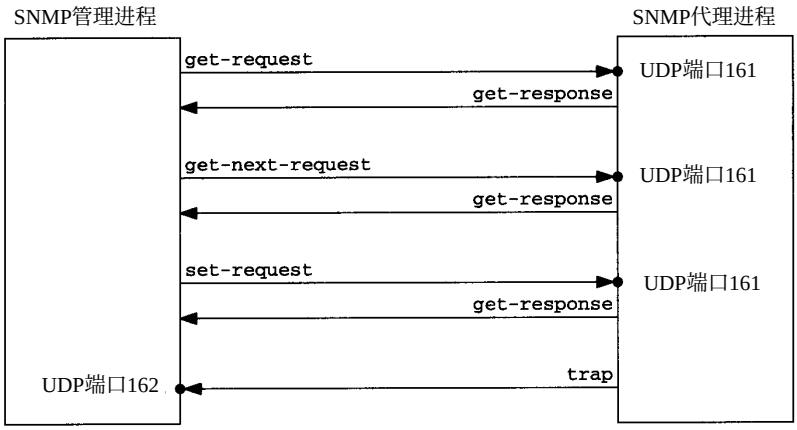


图25-1 SNMP的5种操作

既然这些操作中的前 4种操作是简单的请求 -应答方式（也就是管理进程发出请求，代理进程应答响应），而且在SNMP中往往使用UDP协议，所以可能发生管理进程和代理进程之间数据报丢失的情况。因此一定要有超时和重传机制。

管理进程发出的前面 3种操作采用UDP的161端口。代理进程发出的 Trap操作采用UDP的162端口。由于收发采用了不同的端口号，所以一个系统可以同时为管理进程和代理进程（参见习题25.1）。

图25-2是封装成UDP数据报的5种操作的SNMP报文格式。

在图中，我们仅仅对IP和UDP的首部长度进行了标注。这是由于：SNMP报文的编码采用了ASN.1和BER，这就使得报文的长度取决于变量的类型和值。关于 ASN.1和BER的内容将在后面介绍。在这里介绍各个字段的内容和作用。

版本字段是0。该字段的值是通过SNMP版本号减去1得到的。显然0代表SNMP v1。

图25-3显示各种PDU对应的值（PDU即协议数据单元，也就是分组）。

共同体字段是一个字符串。这是管理进程和代理进程之间的口令，是明文格式。默认的值是public。

对于get、get-next和set操作，请求标识由管理进程设置，然后由代理进程在 get-response中返回。这种类型的字段我们在其他UDP应用中曾经见过（回忆一下在图14-3中DNS的标识字段，或者是图16-2中的事务标识字段）。这个字段的作用是使客户进程（在目前情况下是管理进程）能够将服务器进程（即代理进程）发出的响应和客户进程发出的查询进行匹配。这个

字段允许管理进程对一个或多个代理进程发出多个请求，并且从返回的众多 应答中进行分类。

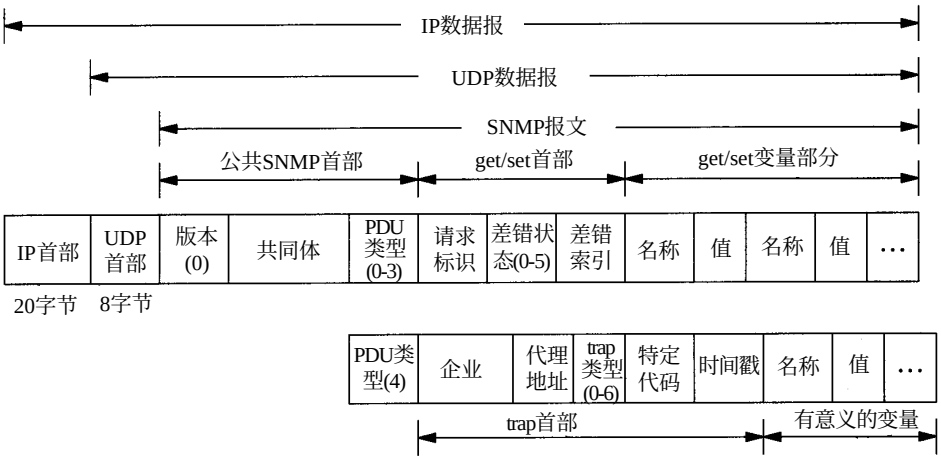


图25-2 SNMP报文的格式

差错状态字段是一个整数，它是由代理进程标注的，指明有差错发生。图25-4是参数值、名称和描述之间的对应关系。

差错索引字段是一个整数偏移量，指明当有差错发生时，差错发生在哪个参数。它是由代理进程标注的，并且只有在发生noSuchName、readOnly和badValue差错时才进行标注。

PDU类型	名　　称
0	get-request
1	get-next-request
2	get-response
3	set-request
4	trap

图25-3 SNMP报文中的PDU类型

差错状态	名　称	描　　述
0	noError	没有错误
1	tooBig	代理进程无法把响应放在一个SNMP消息中发送
2	noSuchName	操作一个不存在的变量
3	badValue	set操作的值或语义有错误
4	readOnly	管理进程试图修改一个只读变量
5	genErr	其他错误

图25-4 SNMP差错状态的值

在get、get-next和set的请求数据报中，包含变量名称和变量值的一张表。对于 get和get-next操作，变量值部分被忽略，也就是不需要填写。

对于trap操作符（PDU类型是4），SNMP报文格式有所变化。我们将在 25.10节中当讨论到trap时再详细讨论。

25.3 管理信息结构

SNMP中，数据类型并不多。在本节，我们就讨论这些数据类型，而不关心这些数据类型在实际中是如何编码的。

- INTEGER。一个变量虽然定义为整型，但也有多种形式。有些整型变量没有范围限制，有些整型变量定义为特定的数值（例如, IP的转发标志就只有允许转发时的1或者不允许转发时的2这两种），有些整型变量定义为一个特定的范围（例如, UDP和TCP的端口号就从0到65535）。
- OCTER STRING 0或多个8 bit字节，每个字节值在0~255之间。对于这种数据类型和

下一种数据类型的BER编码，字符串的字节个数要超过字符串本身的长度。这些字符串不是以NULL结尾的字符串。

- DisplayString。0或多个8 bit字节，但是每个字节必须是ASCII码（26.4中有ASCII字符集）。在MIB-II中，所有该类型的变量不能超过255个字符（0个字符是可以的）。
- OBJECT IDENTIFIER将在下一节中介绍。
- NULL。代表相关的变量没有值。例如，在get或get-next操作中，变量的值就是NULL，因为这些值还有待到代理进程处去取。
- IpAddress。4字节长度的OCTER STRING，以网络序表示的IP地址。每个字节代表IP地址的一个字段。
- PhysAddress。OCTER STRING类型，代表物理地址（例如以太网物理地址为6个字节长度）。
- Counter。非负的整数，可从0递增到 $2^{32}-1$（4 294 976 295）。达到最大值后归0。
- Gauge。非负的整数，取值范围为从0到4 294 976 295(或增或减)。达到最大值后锁定，直到复位。例如，MIB中的tcpCurrEstab就是这种类型的变量的一个例子，它代表目前在ESTABLISHED或CLOSE_WAIT状态的TCP连接数。
- TimeTicks。时间计数器，以0.01秒为单位递增，但是不同的变量可以有不同的递增幅度。所以在定义这种类型的变量的时候，必须指定递增幅度。例如，MIB中的sysUpTime变量就是这种类型的变量，代表代理进程从启动开始的时间长度，以多少个百分之一秒的数目来表示。
- SEQUENCE。这一数据类型与C程序设计语言中的“structure”类似。一个SEQUENCE包括0个或多个元素，每一个元素又是另一个ASN.1数据类型。例如，MIB中的UdpEntry就是这种类型的变量。它代表在代理进程侧目前“激活”的UDP数量（“激活”表示目前被应用程序所用）。在这个变量中包含两个元素：
  1) IpAddress类型中的udpLocalAddress，表示IP地址。
  2) INTEGER类型中的udpLocalPort，从0到65535，表示端口号。
- SEQUENDE OF。这是一个向量的定义，其所有元素具有相同的类型。如果每一个元素都具有简单的数据类型，例如是整数类型，那么我们就得到一个简单的向量（一个一维向量）。但是我们将看到，SNMP在使用这个数据类型时，其向量中的每一个元素是一个SEQUENCE（结构）。因而可以将它看成为一个二维数组或表。

例如，名为udpTable的UDP监听表(listener)就是这种类型的变量。它是一个二元的SEQUENCE变量。每个二元组就是一个UdpEntry。如图25-5所示。

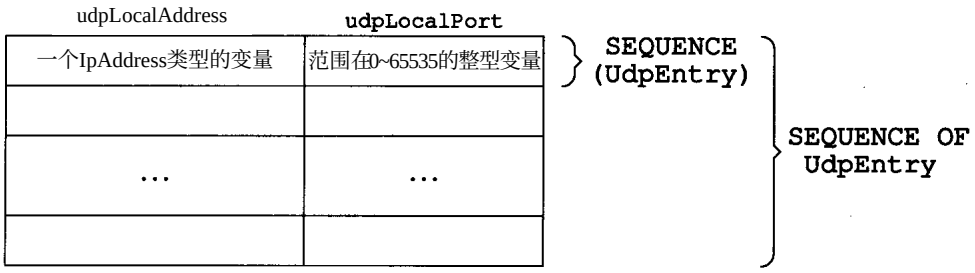


图25-5 表格形式的udpTable 变量

在SNMP中，对于这种类型的表格并没有标注它的列数。但在 25.7节中，我们将看到 get-next操作是如何判断已经操作到最后一列的情况。同时，在 25.6节中，我们还将介绍管理进程如何表示它对某一行数据进行 get或set操作。

25.4 对象标识符

对象标识是一种数据类型，它指明一种“ 授权 ”命名的对象。“ 授权 ”的意思就是这些标识不是随便分配的，它是由一些权威机构进行管理和分配的。

对象标识是一个整数序列，以点（“ . ”）分隔。这些整数构成一个树型结构，类似于 DNS（图14-1）或Unix的文件系统。对象标识从树的顶部开始，顶部没有标识，以 root表示（这和Unix中文件系统的树遍历方向非常类似）。

图25-6显示了在SNMP中用到的这种树型结构。所有的 MIB变量都从1.3.6.1.2.1这个标识开始。

树上的每个结点同时还有一个文字名。例如标识 1.3.6.1.2.1就和`iso.org.dod.internet.memt.mib`对应。这主要是为了人们阅读方便。在实际应用中，也就是说在管理进程和代理进程进行数据报交互时，MIB变量名是以对象标识来标识的，当然都是以1.3.6.1.2.1开头的。

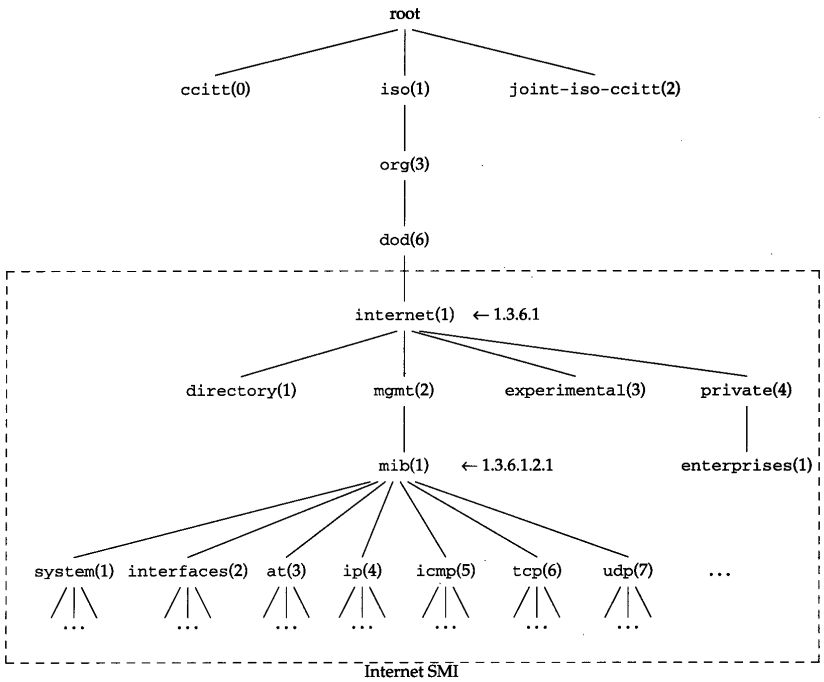


图25-6 管理信息库中的对象标识

在图25-6中，我们除了给出了`mib`对象标识外，还给出了`iso.org.dod.internet.private.enterprises`（1.3.6.1.4.1）这个标识。这是给厂家自定义而预留的。在 Assigned Number RFC中列出了在该结点下大约400个标识。

25.5 管理信息库介绍

所谓管理信息库，或者MIB，就是所有代理进程包含的、并且能够被管理进程进行查询和设

置的信息的集合。我们在前面已经提到了在RFC 1213 [McColghrie 和Rose 1991]中定义的MIB-II。

如图25-6所示，MIB被划分为若干个组，如system、interfaces、at（地址转换）和ip组等。

在本节，我们仅仅讨论UDP组中的变量。这个组比较简单，它包含几个变量和一个表格。在下一节，我们将以UDP组为例，详细讲解什么是实例标识（instance identification），什么是字典式排序(lexicographic ordering)以及和这些概念有关的一些简单例子。在这些例子之后，在25.8节我们继续回到MIB，描述MIB中的其他一些组。

在图25-6中我们画出了udp组在mib的下面。图25-7就显示了UDP组的结构。

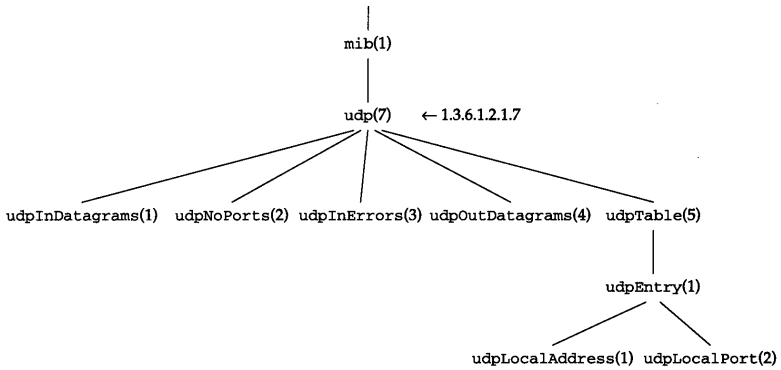


图25-7 UDP组的结构

在该组中，包含4个简单变量和1个由两个简单变量组成的表格。图 25-8描述了这4个简单变量。

名　　称	数据类型	R/W	描　　述
udpInDatagrams	Counter		UDP数据报输入数
udpNoPorts	Counter		没有发送到有效端口的UDP数据报个数
udpInErrors	Counter		接收到的有错误的UDP数据报个数(例如检验错误)
udpOutDatagrams	Counter		UDP数据报输出数

图25-8 UDP组下的简单变量

在本章中，我们就以图 25-8的格式来描述所有的MIB变量。“ R/W ”列如果为空，则代表该变量是只读的；如果变量是可读可写的，则以“ · ”符号来表示。哪怕整个组中的变量都是只读的，我们也将列出“ R/W ”列，以提示读者管理进程只能对这些变量进行查询操作(上图UDP组我们就是这样做的)。同样，如果变量类型是INTEGET类型并且有范围约束，我们也将标明它的下限和上限，就如我们在下图中描述 UDP端口号所做的一样。

图25-9描述了在udpTable中的两个简单变量。

UDP监听表，索引=<udpLocalAddress>.<udpLocalPort>			
名　　称	数据类型	R/W	描　　述
udpLocalAddress	IpAddress		监听进程的本地IP地址。0.0.0.0代表接收任何接口的数据报
udpLocalPort	[0..65535]		监听进程的本地端口号

图25-9 udpTable 中的变量

每次当我们以SNMP表格形式来描述MIB变量时，表格的第1行都表示索引的值，它是表

格中的每一列的参考。在下一节中读者将看到的一些例子也是这样做的。

Case图

在图25-8中，前3个计数器是有相互关系的。Case图真实地描述了一个给出的MIB组中变量之间的相互关系。图25-10就是UDP组的Case图。

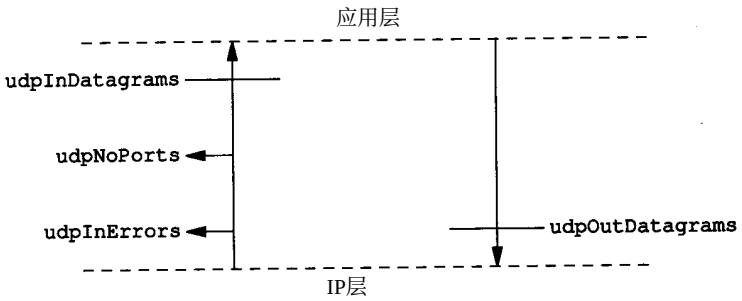


图25-10 UDP组的Case图

这张图表明，发送到应用层的UDP数据报的数量（udpInDatagrams）就是从IP层送到UDP层的UDP数据报的数量，当然udpInError和udpNoPorts也类似。同样，发送到IP层的UDP数据报的数量（udpOutDatagrams）就是从应用层发出的UDP数据报的数量。这表明udpInDatagram不包括udpInError和udpNoPorts。

在深入讲解MIB的时候，这些Case图被用来验证：分组的所有数据路径都是被计数的。[Rose 1994] 中显示了所有MIB组的Case图。

25.6 实例标识

当对MIB变量进行操作，如查询和设置变量的值时，必须对MIB的每个变量进行标识。首先，只有叶子结点是可操作的。SNMP没法处理表格的一整行或一整列。回到图25-7，在图25-8和图25-9中描述过的变量就是叶子结点，而mib、udp、udpTable和udpEntry就不是叶子结点。

25.6.1 简单变量

对于简单变量的处理方法是通过在其对象标识后面添加“ .0 ”来处理的。例如图25-8中的计数器udpInDatagrams，它的对象标识是1.3.6.1.2.1.7.1，它的实例标识是1.3.6.1.2.1.7.1.0，相对应的文字名称是iso.org.dod.internet.mgmt.mib.udp.udpInDatagrams.0。

虽然这个变量处理后通常可以缩写为 udpInDatagrams.0，但我们还是要提醒读者在SNMP报文中（图25-2）该变量的名称是其对象的标识1.3.6.1.2.1.7.1.0。

25.6.2 表格

表格的实例标识就要复杂得多。回顾一下图25-8中的UDP 监听表。

每个MIB中的表格都指明一个以上的索引。对于 UDP监听表来说， MIB定义了包含两个变量的联合索引，这两个变量是：udpLocalAddress，它是一个IP地址；udpLocalPort，它是一个整数（在图25-9中的第1行就显示了这个索引）。

假设在UDP监听表中有3行具体成员：第1行的IP地址是0.0.0.0，端口号是67；第2行的IP

地址是0.0.0.0，端口号是161；第3行的IP地址是0.0.0.0，端口号是520。如图25-11所示。

这意味着系统将从端口 67（BOOTP服务器）、端口161（SNMP）和端口520（RIP）接受来自任何接口的UDP数据报。表格中的这 3行经过处理后的结果在图 25-12中显示。

udpLocalAddress	udpLocalPort
0.0.0.0	67
0.0.0.0	161
0.0.0.0	520

图25-11 UDP监听表

25.6.3 字典式排序

MIB中按照对象标识进行排序时有一个隐含的排序规则。 MIB表格是根据其对象标识按照字典的顺序进行排序的。这就意味着图 25-12中的6个变量排序后的情况如图 25-13所示。从这种字典式排序中可以得出两个重要的结论。

行	对象标识	简　　称	值
1	1.3.6.1.2.1.7.5.1.1.0.0.0.0.67 1.3.6.1.2.1.7.5.1.2.0.0.0.0.67	udpLocalAddress.0.0.0.0.67 udpLocalPort.0.0.0.0.67	0.0.0.0 67
2	1.3.6.1.2.1.7.5.1.1.0.0.0.0.161 1.3.6.1.2.1.7.5.1.2.0.0.0.0.161	udpLocalAddress.0.0.0.0.161 udpLocalPort.0.0.0.0.161	0.0.0.0 161
3	1.3.6.1.2.1.7.5.1.1.0.0.0.0.520 1.3.6.1.2.1.7.5.1.2.0.0.0.0.520	udpLocalAddress.0.0.0.0.520 udpLocalPort.0.0.0.0.520	0.0.0.0 520

图25-12 UDP监听表中行的实例标识

列	对象标识(字典序)	简　　称	值
1	1.3.6.1.2.1.7.5.1.1.0.0.0.0.67 1.3.6.1.2.1.7.5.1.1.0.0.0.0.161 1.3.6.1.2.1.7.5.1.1.0.0.0.0.520	udpLocalAddress.0.0.0.0.67 udpLocalAddress.0.0.0.0.161 udpLocalAddress.0.0.0.0.520	0.0.0.0 0.0.0.0 0.0.0.0
2	1.3.6.1.2.1.7.5.1.2.0.0.0.0.67 1.3.6.1.2.1.7.5.1.2.0.0.0.0.161 1.3.6.1.2.1.7.5.1.2.0.0.0.0.520	udpLocalPort.0.0.0.0.67 udpLocalPort.0.0.0.0.161 udpLocalPort.0.0.0.0.520	67 161 520

图25-13 UDP监听表的字典式排序

1) 在表格中，一个给定变量（在这里指udpLocalAddress）的所有实例都在下个变量（这里指udpLocalPort）的所有实例之前显示。这暗示表格的操作顺序是“先列后行”的次序。这是由于对对象标识进行字典式排序所得到的，而不是按照人们的阅读习惯而排列的。

2) 表格中对行的排序和表格中索引的值有关。在图25-13中，67的字典序小于161，同样161的字典序小于520。

图25-14描述了例子中UDP监听表的这种“先列后行”的次序。

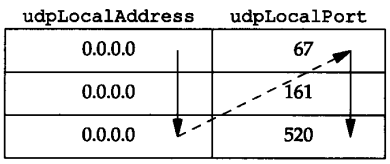


图25-14 按“先列后行”次序显示的UDP监听表

在下节中，讲述到get-next操作时，同样还会遇到这种“先列后行”的次序。

25.7 一些简单的例子

在本节中，我们将介绍如何从SNMP代理进程处获取变量的值。对代理进程进行查询的软件属于ISODE系统，叫做snmpi。两者在[Rose 1994]中有详细的介绍。

25.7.1 简单变量

对一个路由器取两个UDP组的简单变量值：

```
sun % snmpi -a gateway -c secret

snmpi> get udpInDatagrams.0 udpNoPorts.0
udpInDatagrams.0=616168
udpNoPorts.0=33

snmpi> quit
```

其中，-a选项代表要和之通信的代理进程名称，-c选项表示SNMP的共同体名。所谓共同体名，就是客户进程（在这里指snmpi）提供、同时能被服务器进程（这里指代理进程gateway）所识别的一个口令，共同体名称是管理进程请求的权限标志。代理进程允许客户进程用只读共同体名对变量进行读操作，用读写共同体名对变量进行读和写操作。

Snmpi程序的输出提示符是snmpi>，在后面可以键入如get这样的命令，该软件将把它转化为SNMP中的get-request报文。当结束时，键入quit就退出（在后面的例子中，我们将省略掉quit的操作）。

图25-15显示的是对于这个例子tcpdump的两行输出结果。

```
1  0.0                    sun.1024 > gateway.161: GetRequest(42)
                          1.3.6.1.2.1.7.1.0 1.3.6.1.2.1.7.2.0

2  0.348875 (0.3489)      gateway.161 > sun.1024: GetResponse(46)
                          1.3.6.1.2.1.7.1.0=616168
                          1.3.6.1.2.1.7.2.0=33
```

图25-15 简单SNMP查询操作tcpdump 的输出结果

对这两个变量的查询请求是封装在一个UDP数据报中的，而响应也在一个UDP数据报中。

显示的变量是以其对象标识的形式显示的，这是在SNMP报文中实际传输的内容。我们必须指定这两个变量的实例是0。注意，变量的名称（它的对象标识）同样也在响应中返回。在下面我们将看到对于get-next操作这是必需的。

25.7.2 get-next操作

get-next操作是基于MIB的字典式排序的。在下面的例子中，首先向代理进程询问UDP后的下一个对象标识（由于不是一个叶子对象，没有指定任何实例）。代理进程将返回UDP组中的第1个对象，然后我们继续向代理进程取该对象的下一个对象标识，这时候第 2个对象将被返回。重复上面的步骤直到取出所有的对象为止。

```
sun % snmpi -a gateway -c secret

snmpi> next udp
udpInDatagrams.0=616318

snmpi> next udpInDatagrams.0
udpNoPorts.0=33

snmpi> next udpNoPorts.0
udpInErrors.0=0
```

这个例子解释了为什么get-next操作总是返回变量的名称，这是因为我们向代理进程询问下一个变量，代理进程就把变量值和名称一起返回了。

采用这种方式进行get-next操作，我们可以想象管理进程只要做一个简单的循环程序，

就可以从MIB树的顶点开始，对代理进程一步步地进行查询，就可以得出代理进程处所有的变量值和标识。该方式的另外一个用处就是可以对表格进行遍历。

25.7.3 表格的访问

对于“先列后行”次序的UDP监听表，只要采用前面的简单查询程序一步一步地进行操作，就可以遍历整个表格。只要从询问代理进程 udpTable的下一个变量开始就可以了。由于udpTable不是叶子对象，我们不能指定一个实例，但是get-next操作依然能够返回表格中的下一个对象。然后就可以以返回的结果为基础进行下一步的操作，代理进程也会以“先列后行”的次序返回下一个变量，这样就可以遍历整个表格。我们可以看到返回的次序和图25-14相同。

```
sun % snmpi -a gateway -c secret

snmpi> next udpTable
udpLocalAddress.0.0.0.0.67=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.67
udpLocalAddress.0.0.0.0.161=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.161
udpLocalAddress.0.0.0.0.520=0.0.0.0

snmpi> next udpLocalAddress.0.0.0.0.520
udpLocalPort.0.0.0.0.67=67

snmpi> next udpLocalPort.0.0.0.0.67
udpLocalPort.0.0.0.0.161=161

snmpi> next udpLocalPort.0.0.0.0.161
udpLocalPort.0.0.0.0.520=520

snmpi> next udpLocalPort.0.0.0.0.520
snmpInPkts.0=59
```

我们已完成了对UDP监听表的操作

但是管理进程如何知道已经到达表格的最后一行呢？既然get-next操作返回结果中包含表格中的下一个变量的值和名称，当返回的结果是超出表格之外的下一个变量时，管理进程就可以发现变量的名称发生了较大的变化。这样就可以判断出已经到达表格的最后一行。例如在我们的例子中，当返回的是snmpInPkts变量的时候就代表已经到了UDP监听表的最后一个变量了。

25.8 管理信息库（续）

现在继续讨论MIB。我们仅仅介绍下列MIB组：system（系统标识）、if（接口）、at（地址转换）、ip、icmp和tcp。

25.8.1 system组

system组非常简单，它包含7个简单变量（例如，没有表格）。图25-16列出了system组的名称、数据类型和描述。

可以对netb路由器查询一些简单变量：

```
sun % snmpi -a netb -c secret

snmpi> get sysDescr.0 sysObjectID.0 sysUpTime.0 sysServices.0
sysDescr.0="Epilogue Technology SNMP agent for Telebit NetBlazer"
sysObjectID.0=1.3.6.1.4.1.12.42.3.1
sysUpTime.0=22 days, 11 hours, 23 minutes, 2 seconds (194178200 timeticks)
sysServices.0=0xc<internet,transport>
```

名　称	数据类型	R/W	描　　述
sysDescr	DisplayString		系统的文字描述
sysObjectID	ObjectID		在子树1.3.6.1.4.1中的厂商标识
sysUpTime	TimeTicks		从系统的网管部分启动以来运行的时间（以百分之一秒为计算单位）
sysContact	DisplayString	•	联系人的名字及联系方式
sysName	DisplayString	•	结点的完全合格的域名(FQDN)
sysLocation	DisplayString	•	结点的物理位置
sysServices	[0…127]	•	指示结点提供的服务的值。该值是此结点所支持的OSI模型中层次的和。根据所提供的服务，将下面的一些值相加：0x01(物理层)、0x02（数据链路层）、0x04（互联网层）、0x08（端到端的运输层）和 0x40（应用层）

图25-16　system组中的简单变量

回到图25-6中，system的对象标识符在internet.private.enterprises组（1.3.6.1.4.1）中，在Assigned Numbers RFC文档中可以确定下一个对象标识符（12）肯定是指派给了厂家（Epilogue）。

同时还可以看出，sysServices变量的值是4与8的和，它支持网络层（例如选路）和运输层的应用（例如端到端）。

25.8.2　interface组

名　称	数据类型	R/W	描述
ifNumber	INTEGER		系统上的网络接口数

图25-17　if组中的简单变量

在本组中只定义了一个简单变量，那就是系统的接口数量，如图25-17所示。

在该组中，还有一个表格变量，有 22列。表格中的每行定义了接口的一些特征参数。如图25-18所示。

可以向主机sun查询所有这些接口的变量。如3.8节中所示，我们还是希望访问三个接口，如果SLIP接口已经启动：

```
sun % snmpi -a sun                          首先看第一个接口的接口索引

snmpi> next ifTable
ifIndex.1=1

snmpi> get ifDescr.1 ifType.1 ifMtu.1 ifSpeed.1 ifPhysAddress.1
ifDescr.1="le0"
ifType.1=ethernet-csmacd(6)
ifMtu.1=1500
ifSpeed.1=10000000
ifPhysAddress.1=0x08:00:20:03:f6:42

snmpi> next ifDescr.1 ifType.1 ifMtu.1 ifSpeed.1 ifPhysAddress.1
ifDescr.2="sl0"
ifType.2=propPointToPointSerial(22)
ifMtu.2=552
ifSpeed.2=0
ifPhysAddress.2=0x00:00:00:00:00:00

snmpi> next ifDescr.2 ifType.2 ifMtu.2 ifSpeed.2 ifPhysAddress.2
ifDescr.3="lo0"
ifType.3=softwareLoopback(24)
ifMtu.3=1536
ifSpeed.3=0
ifPhysAddress.3=0x00:00:00:00:00:00
```

接口表，索引＝<*IfIndex*>			
名称	数据类型	R/W	描　述
ifIndex	INTEGER		接口索引，介于1和ifNumber之间
ifDescr	DisplayString		接口的文字描述
ifType	INTEGER		类型，例如：6 = 以太网，7 = 802.3以太网，9 = 802.5令牌环，23 = PPP，28 = SLIP,还有其他一些值
ifMtu	INTEGER		接口的MTU
ifSpeed	Gauge		以b/s为单位的速率
ifPhysAddress	PhysAddress		物理地址，对无物理地址的接口，以一串0表示（例如，串行链路）
ifAdminStatus	[1…3]	•	所希望的接口状态：1= 工作，2 = 不工作，3 = 测试
ifOperStatus	[1…3]	•	当前接口的状态：1= 工作，2 = 不工作，3 = 测试
ifLastChange	TimeTicks		当接口进入目前运行状态时sysUpTime的值
ifInOctets	Counter		收到的字节总数，包括组帧字符
ifInUcastPkts	Counter		交付给高层的单播分组数
ifInNUcastPkts	Counter		交付给高层的非单播（例如，广播或多播）分组数
ifInDiscards	Counter		收到的被丢弃的分组数，即使在分组中无差错（例如，无缓存空间）
ifInErrors	Counter		收到的由于差错被丢弃的分组数
ifInUnknownProtos	Counter		收到的由于未知的协议被丢弃的分组数
ifOutOctets	Counter		发送的字节总数，包括组帧字符
ifOutUcastPkts	Counter		从高层接收到的单播分组数
ifOutNUcastPkts	Counter		从高层接收到的非单播（如广播或多播）分组数
ifOutDiscards	Counter		发出的被丢弃的分组数，即使在分组中无差错（如无缓存空间）
ifOutErrors	Counter		发出的由于差错被丢弃的分组数
ifOutQLen	Gauge		在输出队列中的分组数
ifSpecific	ObjectID		对这种特定媒体类型的MIB定义的引用

图25-18　在接口表中的变量：ifTable

对于第1个接口，采用get操作提取5个变量值，然后用get-next操作提取第2个接口的相同的5个参数。对于第3个接口，同样采用get-next操作。

对于SLIP链路的接口类型，所报告的是一个点到点的专用串行链路，而不是　SLIP链路。此外，SLIP链路的速率没有报告。

这个例子对我们理解get-next操作和“先列后行”次序之间的关系十分重要。如果我们键入命令“next ifDescr.1”，则系统返回的是表格中的下一行所对应的变量，而不是同一行中的下个变量。如果表格是按照“先行后列”次序存放，我们就不能通过一个给定变量来读取下一个变量。

25.8.3　at组

地址转换组对于所有的系统都是必需的，但是在 MIB-II中已经没有这个组。从 MIB-II开

始，每个网络协议组（如IP组）都包含它们各自的网络地址转换表。例如对于IP组，网络地址转换表就是ipNetToMediaTable。

在该组中，仅有一个由3列组成的表格变量。如图25-19所示。

我们将用snmpi程序中的一个新命令来转储(dump)整个表格。向一个叫做kinetics的路由器（该路由器连接了一个TCP/IP网络和一个AppleTalk网络）查询其整个ARP高速缓存。命令的输出是字典式排序的整个表格内容。

地址转换表，索引＝<*atIfIndex*>.1.<*atNetAddress*>			
名　称	数据类型	R/W	描　述
atIfIndex	INTEGER	•	接口数：ifIndex
atPhysAddress	PhysAddress	•	物理地址。若设置为长度为0的字符串，则表示无效表项
atNetAddress	NetworkAddress	•	IP地址

图25-19　网络地址转换表：atTable

```
sun % snmpi -a kinetics -c secret dump at

atIfIndex.1.1.140.252.1.4=1
atIfIndex.1.1.140.252.1.22=1
atIfIndex.1.1.140.252.1.183=1
atIfIndex.2.1.140.252.6.4=2
atIfIndex.2.1.140.252.6.6=2

atPhysAddress.1.1.140.252.1.4=0xaa:00:04:00:f4:14
atPhysAddress.1.1.140.252.1.22=0x08:00:20:0f:2d:38
atPhysAddress.1.1.140.252.1.183=0x00:80:ad:03:6a:80
atPhysAddress.2.1.140.252.6.4=0x00:02:16:48
atPhysAddress.2.1.140.252.6.6=0x00:02:3c:48

atNetAddress.1.1.140.252.1.4=140.252.1.4
atNetAddress.1.1.140.252.1.22=140.252.1.22
atNetAddress.1.1.140.252.1.183=140.252.1.183
atNetAddress.2.1.140.252.6.4=140.252.6.4
atNetAddress.2.1.140.252.6.6=140.252.6.6
```

让我们来分析一下用tcpdump命令时的分组交互情况。当snmpi要转储整个表格时，首先发出一条get-next命令以取得表格的名称（在本例中是at），该名称就是要获取的第一个表项。然后在屏幕上显示的同时生成另一条get-next命令。直到遍历完整个表格的内容后才终止。

atIfIndex	atPhysAddress	atNetAddress
1	0xaa:00:04:00:f4:14	140.252.1.4
1	0x08:00:20:0f:2d:38	140.252.1.22
1	0x00:80:ad:03:6a:80	140.252.1.183
2	0x00:02:16:48	140.252.6.4
2	0x00:02:3c:48	140.252.6.6

图25-20　at表举例（ARP高速缓存）

图25-20显示了在路由器中实际表格的内容。

注意图中，接口2的AppleTalk协议的物理地址是32 bit的数值，而不是我们所熟悉的以太网的48 bit物理地址。同时请注意，正如我们所希望的那样，在图中有一条记录和netb路由器（其IP地址是140.252.1.183）有关。这是因为netb路由器和kinetics路由器在同一个以太网中（140.252.1），而且kinetics路由器必需采用ARP来回送SNMP响应。

25.8.4　ip组

ip组定义了很多简单变量和3个表格变量。图25-21显示了所有的简单变量。

名称	数据类型	R/W	描 述
ipForwarding	[1…2]	•	1代表系统正在转发IP数据报，2则代表不在转发
ipDefaultTTL	INTEGER	•	当运输层不提供TTL值时的默认TTL值
ipInReceives	Counter		从所有接口收到的IP数据报的总数
ipInHdrErrors	Counter		由于首部差错被丢弃的数据报数（例如，检验和差错，版本不匹配，TTL超过等）
ipInAddrErrors	Counter		由于不正确的目的地址被丢弃的IP数据报数
ipForwDatagrams	Counter		曾进行过一次转发尝试的IP数据报数
ipInUnknownProtos	Counter		具有无效协议字段的发往本地的IP数据报数
ipInDiscards	Counter		由于缓存空间不足被丢弃的收到的数据报数
ipInDelivers	Counter		交付到适当的协议模块的IP数据报数
ipOutRequests	Counter		传递给IP层来传输的IP数据报总数。不包括已经在ipForwDatagrams中计入的那些
ipOutDiscards	Counter		由于缓存空间不足被丢弃的输出数据报数
ipOutNoRoutes	Counter		由于找不到路由被丢弃的数据报数
ipReasmTimeout	INTEGER		在等待重装时已收到的数据报片被保留的最大秒数
ipReasmReqds	Counter		收到的需要进行重装的IP数据报片的数目
ipReasmOKs	Counter		已成功重装的IP数据报数
ipReasmFails	Counter		IP重装算法失败次数
ipFragOKs	Counter		被成功分片的IP数据报数
ipFragFails	Counter		需要进行分片但由于设置了“不分片”标志而不能分片的IP数据报数
ipFragCreates	Counter		由分片而产生的IP数据报片的数目
ipRoutingDiscards	Counter		所选择的选路表项即使是有效的但也要丢弃的数目

图25-21 ip组中的简单变量

ip组中的第一个表格变量是IP地址表。系统的每个IP地址都对应该表格中的一行。每行中包含了5个变量，如图25-22所示。

IP地址表，索引＝<*ipAdEntAddr*>			
名 称	数据类型	R/W	描 述
ipAdEntAddr	IpAddress		这一行的IP地址
ipAdEntIfIndex	INTEGER		对应的接口数：ifIndex
ipAdEntNetMask	IpAddress		对这个IP地址的子网掩码
ipAdEntBcastAddr	[0…1]		IP广播地址中的最低位的值。通常为1
ipAdEntReasmMaxSize	[0…65535]		在这个接口上收到的、能够进行重装的、最长的IP数据报

图25-22 IP地址表：ipAddrTable

同样可以向主机sun查询整个IP地址表：

```
sun % snmpi -a sun dump ipAddrTable

ipAdEntAddr.127.0.0.1=127.0.0.1
ipAdEntAddr.140.252.1.29=140.252.1.29
ipAdEntAddr.140.252.13.33=140.252.13.33
```

```
ipAdEntIfIndex.127.0.0.1=3              环回接口
ipAdEntIfIndex.140.252.1.29=2           SLIP接口
ipAdEntIfIndex.140.252.13.33=1          以太网接口

ipAdEntNetMask.127.0.0.1=255.0.0.0
ipAdEntNetMask.140.252.1.29=255.255.255.0
ipAdEntNetMask.140.252.13.33=255.255.255.224

ipAdEntBcastAddr.127.0.0.1=1            所有这三个都使用一个比特进行广播
ipAdEntBcastAddr.140.252.1.29=1
ipAdEntBcastAddr.140.252.13.33=1

ipAdEntReasmMaxSize.127.0.0.1=65535
ipAdEntReasmMaxSize.140.252.1.29=65535
ipAdEntReasmMaxSize.140.252.13.33=65535
```

输出的接口号码可以和图25-18中的输出进行比较，同样IP地址和子网掩码可以和3.8节中采用ifconfig命令时的输出进行比较。

ip组中的第二个表是IP路由表（请回忆一下我们在9.2节中讲到的路由表），如图25-23所示。访问该表中每行记录的索引是目的IP地址。

名　称	数据类型	R/W	描　述
ipRouteDest	IpAddress	•	目的IP地址。值0.0.0.0表示一个默认的表项
ipRouteIfIndex	INTEGER	•	接口数：ifIndex
ipRouteMetric1	INTEGER	•	主要的选路度量。这个度量的意义取决于选路协议（ipRouteProto）。-1表示未使用
ipRouteMetric2	INTEGER	•	可选的选路度量
ipRouteMetric3	INTEGER	•	可选的选路度量
ipRouteMetric4	INTEGER	•	可选的选路度量
ipRouteNextHop	IpAddress	•	下一跳路由器的IP地址
ipRouteType	INTEGER	•	路由类型：1 = 其他，2 = 无效路由，3 = 直接，4= 间接
ipRouteProto	INTEGER		选路协议：1 = 其他，4 = ICMP重定向，8 = RIP, 13 = OSPF, 14 = BGP, 以及其他
ipRouteAge	INTEGER	•	自从路由上次被更新或确定是正确的以后所经历的秒数
ipRouteMask	IpAddress	•	在和ipRouteDest相比较之前，掩码要与目的IP地址进行逻辑“与”
ipRouteMetric5	INTEGER	•	其他的选路度量
ipRouteInfo	ObjectID	•	对这种特定选路协议的MIB定义的引用

图25-23　IP路由表：ipRouteTable

图25-24显示的是用snmpi程序采用dump ipRouteTable命令从主机sun得到的路由表。在这张表中，已经删除了所有5个路由度量，那是由于这5条记录的度量都是-1。在列的标题中，对每个变量名称已经删除了ipRoute这样的前缀。

Dest	IfIndex	NextHop	Type	Proto	Mask
0.0.0.0	2	140.252.1.183	间接(4)	其他(1)	0.0.0.0
127.0.0.1	3	127.0.0.1	直接(3)	其他(1)	255.255.255.255
140.252.1.183	2	140.252.1.29	直接(3)	其他(1)	255.255.255.255
140.252.13.32	1	140.252.13.33	直接(3)	其他(1)	255.255.0.0
140.252.13.65	1	140.252.13.35	间接(4)	其他(1)	255.255.255.255

图25-24　路由器sun上的IP路由表

为比较起见，下面的内容是我们用netstat命令（在9.2节中曾经讨论过）格式显示的路由表信息。图25-24是按字典序显示的，这和netstat命令显示格式不同：

```
sun % netstat -rn
Routing tables
Destination          Gateway              Flags    Refcnt Use        Interface
140.252.13.65        140.252.13.35        UGH      0      115        le0
127.0.0.1            127.0.0.1            UH       1      1107       lo0
140.252.1.183        140.252.1.29         UH       0      86         sl0
default              140.252.1.183        UG       2      1628       sl0
140.252.13.32        140.252.13.33        U        8      68359      le0
```

ip组的最后一个表是地址转换表，如图 25-25所示。正如我们前面所说的， at组已经被删除了，在这里已经用IP表来代替了。

IP地址转换表，索引＝<*ipNetToMediaIfIndex*>.<*ipNetToMediaNetAddress*>			
名　　称	数 据 类 型	R/W	描　　述
ipNetToMediaIfIndex	INTEGER	•	对应的接口：ifIndex
ipNetToMediaPhysAddress	PhysAddress	•	物理地址
ipNetToMediaNetAddress	IpAddress	•	IP地址
ipNetToMediaType	[1…4]	•	映射的类型：1 = 其他，2 = 无效的，3 = 动态的，4 = 静态的。

图25-25 IP地址转换表：ipNetToMediaTable

这里显示的是系统sun上的ARP高速缓存信息：

```
sun % arp -a
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi (140.252.13.35) at 0:0:c0:6f:2d:40
```

相应的SNMP输出：

```
sun % snmpi -a sun dump ipNetToMediaTable

ipNetToMediaIfIndex.1.140.252.13.34=1
ipNetToMediaIfIndex.1.140.252.13.35=1
ipNetToMediaPhysAddress.1.140.252.13.34=0x00:00:c0:c2:9b:26
ipNetToMediaPhysAddress.1.140.252.13.35=0x00:00:c0:6f:2d:40
ipNetToMediaNetAddress.1.140.252.13.34=140.252.13.34
ipNetToMediaNetAddress.1.140.252.13.35=140.252.13.35
ipNetToMediaType.1.140.252.13.34=dynamic(3)
ipNetToMediaType.1.140.252.13.35=dynamic(3)
```

25.8.5 icmp组

icmp组包含4个普通计数器变量（ICMP报文的输出和输入数量以及 ICMP差错报文的输入和输出数量）和22个其他ICMP报文数量的计数器： 11个是输出计数器，另外11个是输入计数器。如图25-26所示。

对于有附加代码的 ICMP报文（请回忆一下图 6-3中，有15种报文代表目的不可达），SNMP没有为它们定义专门的计数器。

25.8.6 tcp组

图25-27显示的是tcp组中的简单变量。其中的很多变量和图 18-12描述的TCP状态有关。

名　称	数据类型	R/W	描　述
icmpInMsgs	Counter		收到的ICMP报文总数
icmpInErrors	Counter		收到的有差错的ICMP报文数（例如，无效的ICMP检验和）
icmpInDestUnreachs	Counter		收到的ICMP目的站不可达报文数
icmpInTimeExcds	Counter		收到的ICMP超时报文数
icmpInParmProbs	Counter		收到的ICMP参数问题报文数
icmpInSrcQuenchs	Counter		收到的ICMP源站抑制报文数
icmpInRedirects	Counter		收到的ICMP重定向报文数
icmpInEchos	Counter		收到的ICMP回显请求报文数
icmpInEchosReps	Counter		收到的ICMP回显应答报文数
icmpInTimeStamps	Counter		收到的ICMP时间戳请求报文数
icmpInTimeStampReps	Counter		收到的ICMP时间戳应答报文数
icmpInAddrMasks	Counter		收到的ICMP地址掩码请求报文数
icmpInAddrMaskReps	Counter		收到的ICMP地址掩码应答报文数
icmpOutMsgs	Counter		输出的ICMP报文总数
icmpOutErrors	Counter		由于在ICMP报文中有一个问题（例如，缓存空间不足）而未发送的ICMP报文数
icmpOutDestUnreachs	Counter		发送的ICMP目的站不可达报文数
icmpOutTimeExcds	Counter		发送的ICMP超时报文数
icmpOutParmProbs	Counter		发送的ICMP参数问题报文数
icmpOutSrcQuenchs	Counter		发送的ICMP源站抑制报文数
icmpOutRedirects	Counter		发送的ICMP重定向报文数
icmpOutEchos	Counter		发送的ICMP回显请求报文数
icmpOutEchosReps	Counter		发送的ICMP回显应答报文数
icmpOutTimeStamps	Counter		发送的ICMP时间戳请求报文数
icmpOutTimeStampReps	Counter		发送的ICMP时间戳应答报文数
icmpOutAddrMasks	Counter		发送的ICMP地址掩码请求报文数
icmpOutAddrMaskReps	Counter		发送的ICMP地址掩码应答报文数

图25-26　icmp组中的简单变量

名　称	数据类型	R/W	描　述
tcpRtoAlgorithm	INTEGER		用来计算重传超时值的算法：1 =除下列值以外，2 = 固定的RTO, 3 = MIL-STD-1778附件B, 4 = Van Jacobson算法
tcpRtoMin	INTEGER		以毫秒计的最小重传超时值
tcpRtoMax	INTEGER		以毫秒计的最大重传超时值
tcpMaxConn	INTEGER		最大的TCP连接数。若为动态的，则值为- 1
tcpActiveOpens	Counter		从CLOSED到SYN_SENT的状态变迁数
tcpPassiveOpens	Counter		从LISTEN到SYN_RCVD的状态变迁数
tcpAttempFails	Counter		从SYN_SENT或SYN_RCVD到CLOSED的状态变迁数,加上从SYN_RCVD到LISTEN的状态变迁数
tcpEstabResets	Counter		从ESTABLISHED或CLOSE_WAIT状态到CLOSED的状态变迁数
tcpCurrEstab	Gauge		当前在ESTABLISHED或CLOSE_WAIT状态的连接数
tcpInSegs	Counter		收到的报文段的总数
tcpOutSegs	Counter		发送的报文段的总数，但将仅包含重传字节的除外
tcpRetransSegs			重传的报文段的总数
tcpInErrs	Counter		收到的具有一个差错(如无效的检验和)的报文段总数
tcpOutRsts	Counter		具有RST标志置位的报文段的总数

图25-27　tcp组中的简单变量

现在向系统sun查询一些tcp组变量：

```
sun % snmpi -a sun

snmpi> get tcpRtoAlgorithm.0 tcpRtoMin.0 tcpRtoMax.0 tcpMaxConn.0
tcpRtoAlgorithm.0=vanj(4)
tcpRtoMin.0=200
tcpRtoMax.0=12800
tcpMaxConn.0=-1
```

本系统（指SunOS4.1.3）使用的是Van Jacobson超时重传算法，超时定时器的范围在200 ms~12.8 s之间，并且对TCP连接数量没有特定的限制（这里的超时上限12.8 s恐怕有错，因为我们在21章中曾经介绍大多数应用的超时上限是64 s）。

tcp组还包括一个表格变量，即TCP连接表，如图25-28所示。对于每个TCP连接，都对应表格中的一条记录。每条记录包含5个变量：连接状态、本地IP地址、本地端口号、远端IP地址以及远端端口号。

索引＝<tcpConnLocalAddress>.<tcpConnLocalPort>.<tcpConnRemAddress>.<tcpConnRemPort>			
名　称	数据类型	R/W	描　述
tcpConnState	[1…12]	•	连接状态：1 = CLOSED, 2 = LISTEN, 3 = SYN_SENT, 4 = SYN_RCVD, 5 = ESTABLISHED, 6 = FIN_WAIT, 7 = FIN_WAIT, 8 = CLOSE_WAIT, 9 = LAST_ACK, 10 = CLOSING, 11 = TIME_WAIT, 12 = 删除TCB。管理进程对此变量可以设置的唯一值就是12（例如，立即终止此连接）
tcpConnLocalAddress	IpAddress		本地IP地址。0.0.0.0代表监听进程愿意在任何接口接受连接
tcpConnLocalPort	[1…65535]		本地端口号
tcpConnRemAddress	IpAddress		远程IP地址
tcpConnRemPort	[1…65535]		远程端口号

图25-28 TCP连接表：tcpConnTable

让我们看一看在系统sun上的这个表。由于有许多服务器进程在监听这些连接，所以我们只显示该表的一部分内容。在转储全部表格的变量之前，我们必需先建立两条TCP连接：

```
sun % rlogin gemini              gemini的IP地址是140.252.1.11
```

和

```
sun % rlogin localhost           IP地址应该是127.0.0.1
```

在所有的监听服务器进程中，我们仅仅列出了FTP服务器进程的情况，它使用21号端口。

```
sun % snmpi -a sun dump tcpConnTable

tcpConnState.0.0.0.0.21.0.0.0.0.0=listen(2)
tcpConnState.127.0.0.1.23.127.0.0.1.1415=established(5)
tcpConnState.127.0.0.1.1415.127.0.0.1.23=established(5)
tcpConnState.140.252.1.29.1023.140.252.1.11.513=established(5)

tcpConnLocalAddress.0.0.0.0.21.0.0.0.0.0=0.0.0.0
tcpConnLocalAddress.127.0.0.1.23.127.0.0.1.1415=127.0.0.1
tcpConnLocalAddress.127.0.0.1.1415.127.0.0.1.23=127.0.0.1
tcpConnLocalAddress.140.252.1.29.1023.140.252.1.11.513=140.252.1.29
```

```
tcpConnLocalPort.0.0.0.0.21.0.0.0.0.0=21
tcpConnLocalPort.127.0.0.1.23.127.0.0.1.1415=23
tcpConnLocalPort.127.0.0.1.1415.127.0.0.1.23=1415
tcpConnLocalPort.140.252.1.29.1023.140.252.1.11.513=1023

tcpConnRemAddress.0.0.0.0.21.0.0.0.0.0=0.0.0.0
tcpConnRemAddress.127.0.0.1.23.127.0.0.1.1415=127.0.0.1
tcpConnRemAddress.127.0.0.1.1415.127.0.0.1.23=127.0.0.1
tcpConnRemAddress.140.252.1.29.1023.140.252.1.11.513=140.252.1.11

tcpConnRemPort.0.0.0.0.21.0.0.0.0.0=0
tcpConnRemPort.127.0.0.1.23.127.0.0.1.1415=1415
tcpConnRemPort.127.0.0.1.1415.127.0.0.1.23=23
tcpConnRemPort.140.252.1.29.1023.140.252.1.11.513=513
```

对于rlogin到gemini，只显示一条记录，这是因为gemini是另外一个主机。而且我们仅仅能够看到连接的客户端信息（端口号是1023）。但是Telnet连接，客户端和服务器端都显示（客户端口号是1415，服务器端口号是23），这是因为这种连接通过环回接口。同时我们还可以看到，FTP监听服务器程序的本地IP地址是0.0.0.0。这表明它可以接受通过任何接口的连接。

25.9 其他一些例子

现在开始回答前面一些没有回答的问题，我们将用SNMP的知识进行解释。

25.9.1 接口MTU

回忆一下在11.6节的实验中，我们试图得出一条从netb到sun的SLIP连接的MTU。现在可以采用SNMP得到这个MTU。首先从IP路由表中取到SLIP连接（140.252.1.29）的接口号（ipRouteIfIndex），然后就可以用这个数值进入接口表并且取得想要的SLIP连接的MTU（通过SLIP的描述和数据类型）。

```
sun % snmpi -a netb -c secret

snmpi> get ipRouteIfIndex.140.252.1.29
ipRouteIfIndex.140.252.1.29=12
snmpi> get ifDescr.12 ifType.12 ifMtu.12
ifDescr.12="Telebit NetBlazer dynamic dial virtual interface"
ifType.12=other(1)
ifMtu.12=1500
```

可以看到，即使连接的类型是SLIP连接，但是MTU仍设置为以太网，其值为1500，目的可能是为了避免分片。

25.9.2 路由表

回忆一下在14.4节中，我们讨论了DNS如何进行地址排序的问题。当时我们介绍了从域名服务器返回的第1个IP地址是和客户有相同子网掩码的情况。还介绍了用其他的IP地址也会正常工作，但是效率比较低。现在我们从SNMP的角度来查阅路由表的入口，在这里将用到前面章节中和IP路由有关的很多相关知识。

路由器gemini是一个多接口主机，有两个以太网接口。首先确认一下两个接口都可以Telnet登录：

```
sun % telnet 140.252.1.11 daytime
Trying 140.252.1.11 ...
Connected to 140.252.1.11.
```

```
Escape character is '^]'.
Sat Mar 27 09:37:24 1993
Connection closed by foreign host.

sun % telnet 140.252.3.54 daytime
Trying 140.252.3.54 ...
Connected to 140.252.3.54.
Escape character is '^]'.
Sat Mar 27 09:37:35 1993
Connection closed by foreign host.
```

可以看出这两个地址的连接没有什么区别。现在我们采用 traceroute命令来看一下对于每个地址，是否有选路方面的不同：

```
sun % traceroute 140.252.1.11
traceroute to 140.252.1.11 (140.252.1.11), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  299 ms  234 ms  233 ms
 2  gemini (140.252.1.11)  233 ms  228 ms  234 ms

sun % traceroute 140.252.3.54
traceroute to 140.252.3.54 (140.252.3.54), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  245 ms  212 ms  234 ms
 2  swnrt (140.252.1.6)  233 ms  229 ms  234 ms
 3  gemini (140.252.3.54)  234 ms  233 ms  234 ms
```

可以看到：如果采用属于140.252.3子网的地址，就多了额外的一跳。下面解释造成这个额外一跳的原因。

图25-29是系统的连接关系图。从traceroute命令的输出结果可以看出主机gemini和路由器swnrt都连接了两个网段：140.252.3子网和140.252.1子网。

回忆一下在图4-6中，我们解释了路由器netb采用ARP代理进程，使得sun工作站好象是直接连接到140.252.1子网上的情况。我们忽略了sun和netb之间SLIP连接的调制解调器，因为这和我们这里的讨论不相关。

在图25-29中，我们用虚线箭头画出了当Telnet到140.252.3.54时的路径。返回的数据报怎么知道直接从gemini到netb，而不是从原路返回呢？我们采用在8.5节中介绍过的，带有宽松选路特性的traceroute版本来解释：

```
sun % traceroute -g 140.252.3.54 sun
traceroute to sun (140.252.13.33), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  244 ms  256 ms  234 ms
 2  * * *
 3  gemini (140.252.3.54)  285 ms  227 ms  234 ms
 4  netb (140.252.1.183)  263 ms  259 ms  294 ms
 5  sun (140.252.13.33)  534 ms  498 ms  504 ms
```

当在命令中指明是宽松源站选路时，swnrt路由器就不再有响应。看一下前面没有指明源站选路的traceroute命令输出，可以看出swnrt路由器是事实上的第2跳。超时数据必须这样设置的原因是：当数据报指定了宽松源站选路选项时，该路由器没有发生ICMP超时差错。所以在traceroute命令的输出中可以得出，返回路径是从gemini(TTL 3, 4和5)路由器直接到达netb路由器，而不通过swnrt路由器。

还剩下一个需要用SNMP来解释的问题就是：在netb路由器的路由表中，哪条信息代表寻径到140.252.3？该信息表示netb路由器把分组发送给swnrt而不是直接发送给gemini？用get命令来取下一跳路由器的值。

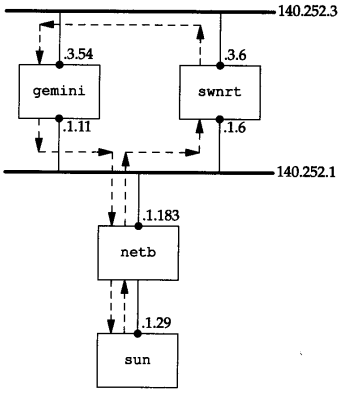


图25-29 例子中的网络拓扑结构

```
sun % snmpi -a netb -c secret get ipRouteNextHop.140.252.3.0
ipRouteNextHop.140.252.3.0=140.252.1.6
```

正如我们所看到发生的那样，路由表设置使得 netb路由器把分组发送到swnrt路由器。

为什么gemini路由器直接把分组回送给netb路由器？那是因为在gemini路由器端，它要回送的分组目的地址是140.252.1.29，而子网140.252.1是直接连接到gemini路由器上的。

从上面这个例子可以看出选路的策略。由于gemini是打算作一个多接口主机而不是路由器，所以默认的到140.253.3子网的路由器是swnrt。这是多接口主机和路由器之间差异的一个典型例子。

25.10 Trap

本章我们看到的例子都是从管理进程到代理进程的。当然代理进程也可以主动发送 trap到管理进程，以告诉管理进程在代理进程侧有某些管理进程所关心的事件发生，如图 25-1所示。trap发送到管理进程的162号端口。

在图25-2中，我们已经描述了 trap PDU的格式。在下面关于 tcpdump输出内容中我们将再一次用到这些字段。

现在已经定义了6种特定的trap类型，第7种trap类型是由供应商自己定义的特定类型。图25-30给出了trap报文中trap类型字段的内容。

trap类型	名　称	描　述
0	coldStart	代理进程对自己初始化
1	warmStart	代理进程对自己重新初始化
2	linkDown	一个接口已经从工作状态改变为故障状态（图 25-18）。报文中的第一个变量标识此接口
3	linkUp	一个接口已经从故障状态改变为工作状态（图 25-18）。报文中的第一个变量标识此接口
4	authenticationFailure	从SNMP管理进程收到无效共同体的报文
5	egpNeighborLoss	一个EGP邻站已变为故障状态。报文中的第一个变量包含此邻站的IP地址
6	enterpriseSpecific	在这个特定的代码字段中查找trap信息

图25-30　trap的类型

用tcpdump命令来看看trap的情况。我们在系统sun上启动SNMP代理进程，然后让它产生coldStart类型的trap（我们告诉代理进程把trap信息发送到bsdi主机。虽然在该主机上并没有运行处理trap的管理进程，但是可以用tcpdump来查看产生了什么样的分组。回忆一下在图25-1中，trap是从代理进程发送到管理进程的，而管理进程不需要给代理进程发送确认。所以我们不需要trap的处理程序）。然后我们用snmpi程序发送一个请求，但该请求的共同体名称是无效的。这将产生一个authenticationFailure类型的trap。图25-31显示了命令的输出结果。

```
1  0.0              sun.snmp > bsdi.snmp-trap: C=traps Trap(28)
                    E:unix.1.2.5 [140.252.13.33] coldStart 20

2  18.86 (18.86)    sun.snmp > bsdi.snmp-trap: C=traps Trap(29)
                    E:unix.1.2.5 [140.252.13.33] authenticationFailure 1907
```

图25-31　tcpdump 输出的由SNMP代理进程产生的trap

首先注意一下两个 UDP数据报都是从 SNMP代理进程（端口是 161，图中显示的名称是 snmp）发送到目的端口号是162的服务器进程上的（图中显示的名称是 snmp-trap）。

再注意一下C=traps是trap报文的共同体名称。这是ISODE SNMP代理进程的配置选项。

下一个要注意的是：第1行中的Trap（28）和第2行中的Trap（29）是PDU类型和长度。

两个输出行的第一项内容都是相同的 E:unix.1.2.5。这代表企业名字段和代理进程的sysObjiectID。它是图25-6中1.3.6.1.4.1（iso.org.dod.internet.private.enterparses）结点下面的某个结点，所以代理进程的对象标识是1.3.6.1.4.1.4.1.2.5。它的简称是：unix.agents.fourBSD-isode.5。最后一个数字“5”代表ISODE代理进程软件的版本号。这些企业名的值代表了产生trap的代理进程软件信息。

输出的下一项是代理进程的IP地址（140.252.13.33）。

在第1行中，trap的类型显示的是coldStart，第2行中，显示的是authenticationFailure。与之相对应的trap类型值是0和4（如图25-30所示）。由于这些都不是厂家自定义的trap，所以特定代码必须是0，在图中没有显示。

输出的最后分别是20和1907，这是时间戳字段。它是TimeTicks类型的值，表示从代理进程初始化开始到trap发生所经历了多少个百分之一秒。在冷启动 trap的情况下，在代理进程初始化后到trap的产生共经历了200 ms。同样，tcpdump的输出表明第2个trap在第1个trap产生的18.86s后出现，这对应于打印出的1907个百分之一秒减去200 ms所得到的值。

图25-2表明trap报文还包含很多代理进程发送给管理进程的变量，但在这些在例子中没有再讨论。

25.11 ASN.1和BER

在正式的SNMP规范中都是采用ASN.1（Abstract Syntax Notation 1）语法，并且在SNMP报文中比特的编码采用BER（Basic Encoding Rule）。和其他介绍SNMP的书不同，我们有目的地把ASN.1和BER的讨论放到最后。因为如果放在前面讨论，有可能使读者产生混淆而忽略了SNMP的真正目的是进行网络管理。在这里我们也只是对这两个概念简单地进行解释，[Rose 1990]的第8章详细讨论了ASN.1和BER。

ASN.1是一种描述数据和数据特征的正式语言。它和数据的存储及编码无关。 MIB和SNMP报文中的所有的字段都是用 ASN.1描述的。例如：对于 SMI中的ipAddress数据类型，ASN.1是这样描述的：

```
IpAddress ::=
    [APPLICATION 0]              -- in network-byte order
        IMPLICIT OCTET STRING (SIZE (4))
```

同样，在MIB中，简单变量的定义是这样描述的：

```
udpNoPorts OBJECT-TYPE
    SYNTAX  Counter
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "The total number of received UDP datagrams for which
            there was no application at the destination port."
    ::= { udp 2 }
```

用SEQUENCE和SEQUENCE OF来定义表格的描述更加复杂。

当有了这样的ASN.1定义，可以有多种编码方法把数据编码为传输的比特流。 SNMP使用的编码方法是BER。例如，对于一个简单的整数如64，在BER中需要用3个字节来表示。第一个字节说明类型是一个整数，下个字节说明用了多少个字节来存储该整数（在这里是 1），最后一个字节才是该整数的值。

幸运的是，ASN.1和BER这两个繁琐的概念仅仅在实现SNMP的时候才重要，对我们理解

网络管理的概念和流程并没有太大的关系。

25.12 SNMPv2

在1993年，发表了定义新版本SNMP的11个RFC。RFC 1441 [Case et al. 1993]是其中的第一个，它系统地介绍了SNMPv2。同样，有两本书 [Stallings 1993; Rose 1994]也对SNMPv2进行了介绍。现在已经有两个SNMPv2的基本模型（参见附录 B.3中的[Rose 1994]），但是厂家的实现到1994年才能广泛使用。

在本节中，我们主要介绍SNMPv1和SNMPv2之间的重要区别。

1) 在SNMPv2中定义了一个新的分组类型`get-bulk-request`，它高效率地从代理进程读取大块数据。

2) 另的一个新的分组类型是`inform-request`，它使一个管理进程可以向另一个管理进程发送信息。

3) 定义了两个新的MIB，它们是：SNMPv2 MIB和SNMPv2-M2M MIB（管理进程到管理进程的MIB）。

4) SNMPv2的安全性比SNMPv1大有提高。在SNMPv1中，从管理进程到代理进程的共同体名称是以明文方式传送的。而SNMP v2可以提供鉴别和加密。

厂家提供的设备支持SNMPv2的会越来越多，管理站将对两个版本的 SNMP代理进程进行管理。[Routhier 1993]中描述了如何将SNMPv1的实现扩展到支持SNMPv2。

25.13 小结

SNMP是一种简单的、SNMP管理进程和SNMP代理进程之间的请求-应答协议。MIB定义了所有代理进程所包含的、能够被管理进程查询和设置的变量，这些变量的数据类型并不多。

所有这些变量都以对象标识符进行标识，这些对象标识符构成了一个层次命名结构，由一长串的数字组成，但通常缩写成人们阅读方便的简单名字。一个变量的特定实例可以用附加在这个对象标识符后面的一个实例来标识。

很多SNMP变量是以表格形式体现的。它们有固定的栏目，但有多少条记录并不固定。对于SNMP来讲，重要的是对表格中的每一行如何进行标识（尤其当我们不知道表格中有多少条记录时）以及如何按字典方式进行排序（“先列后行”的次序）。最后要说明的一点是：SNMP的`get-next`操作符对任何SNMP管理进程来讲都是最基本的操作。

然后我们介绍了下列的SNMP变量组：system、interface、address translation、IP、ICMP、TCP和UDP。接着是两个例子，一个介绍如何确定一个接口的MTU，另外一个介绍如何获取路由器的路由信息。

在本章的后面介绍了SNMP的trap操作，它是当代理进程发生了某些重大事件后主动向管理进程报告的。最后我们简单介绍了 ASN.1和BER，这两个概念比较繁琐，但所幸的是，它对我们了解SNMP并不十分重要，仅仅在实现SNMP的时候才要用到。

习题

25.1 我们说过采用两个不同的UDP端口（161和162）可以使得一个系统既可以是管理进程，也可以是代理进程。如果对管理进程和代理进程采用一个同样的端口，会出现什么情况？

25.2 用`get-next`操作，如何列出一张路由表的完整信息？

第26章　Telnet和Rlogin：远程登录

26.1　引言

远程登录（Remote Login）是Internet上最广泛的应用之一。我们可以先登录（即注册）到一台主机然后再通过网络远程登录到任何其他一台网络主机上去，而不需要为每一台主机连接一个硬件终端（当然必须有登录帐号）。

在TCP/IP网络上，有两种应用提供远程登录功能。

1) Telnet是标准的提供远程登录功能的应用，几乎每个TCP/IP的实现都提供这个功能。它能够运行在不同操作系统的主机之间。Telnet通过客户进程和服务器进程之间的选项协商机制，从而确定通信双方可以提供的功能特性。

2) Rlogin起源于伯克利Unix，开始它只能工作在Unix系统之间，现在已经可以在其他操作系统上运行。

在本章中，我们将介绍Telnet和Rlogin。首先介绍Rlogin，因为Rlogin比较简单。

Telnet是一种最老的Internet应用，起源于1969年的ARPANET。它的名字是“电信网络协议（telecommunication network protocol）”的缩写词。

远程登录采用客户-服务器模式。图26-1显示的是一个Telnet客户和服务器的典型连接图（对于Rlogin的客户和服务器连接图，我们可以画得更加简单）。

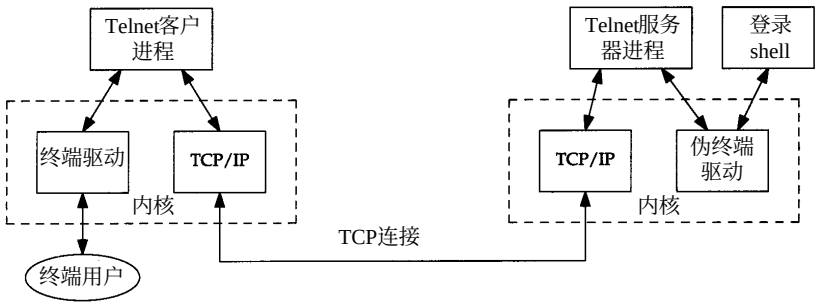


图26-1　客户-服务器模式的Telnet简图

在这张图中，有以下要点需要注意：

1) Telnet客户进程同时和终端用户和TCP/IP协议模块进行交互。通常我们所键入的任何信息的传输是通过TCP连接，连接的任何返回信息都输出到终端上。

2) Telnet服务器进程经常要和一种叫做“伪终端设备”（pseudo-terminal device）打交道，至少在Unix系统下是这样的。这就使得对于登录外壳(shell)进程来讲，它是被Telnet服务器进程直接调用的，而且任何运行在登录外壳进程处的程序都感觉是直接和一个终端进行交互。对于像满屏编辑器这样的应用来讲，就像直接在和终端打交道一样。实际上，如何对服务器进程的登录外壳进程进行处理，使得它好像在直接和终端交互，往往是编写远程登录服务器

进程程序中最困难的方面之一。

3) 仅仅使用了一条TCP连接。由于客户进程必须多次和服务器进程进行通信（反之亦然），这就必然需要某些方法，来描绘在连接上传输的命令和用户数据。我们在后面的内容中会介绍Telnet和Rlogin是如何处理这个问题的。

4) 注意在图 26-1中，我们用虚线框把终端驱动进程和伪终端驱动进程框了起来。在TCP/IP实现中，虚线框的内容一般是操作系统内核的一部分。 Telnet客户进程和服务器进程一般只是属于用户应用程序。

5) 把服务器进程的登录外壳进程画出来的目的是为了说明：当我们想登录到系统的时候，必须要有一个帐号， Telnet和Rlogin都是如此。

对于Telnet和Rlogin，如果比较一下它们客户进程和服务器进程源代码的数量，就可以知道这两者的复杂程度。图 26-2显示了伯克利不同版本的 Telnet和Rlogin客户进程和服务器进程源代码的数量。

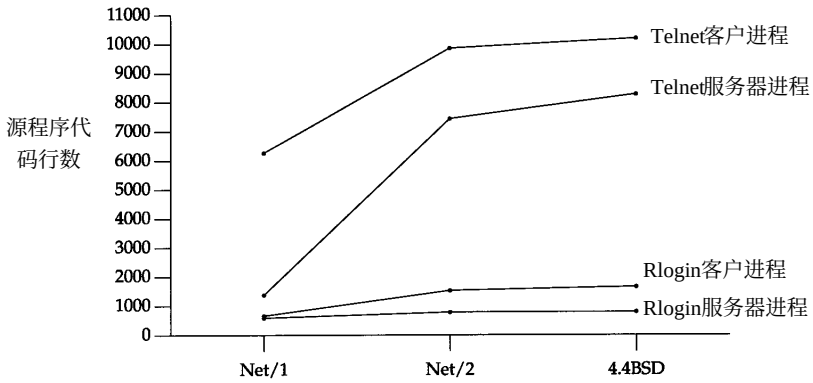


图26-2 Telnet/ Rlogin/客户进程/服务器进程的源代码数量比较

现在，不断有新的Telnet选项被添加到Telnet中去，这就使得Telnet实现的源代码数量大大增加，而Rlogin依然变化不大，还是比较简单。

远程登录不是那种有大量数据报传输的应用。正如我们前面讲到的一样，客户进程和服务器进程交互的分组大多比较小。 [Paxson 1993]发现客户进程发出的字节数（用户在终端上键入的信息）和服务器进程端发出的字节数的数量之比是 1:20。这是因为我们在终端上键入的一条短命令往往令服务器进程端产生很多输出。

26.2 Rlogin协议

Rlogin的第一次发布是在 4.2BSD中，当时它仅能实现 Unix主机之间的远程登录。这就使得Rlogin比Telnet简单。由于客户进程和服务器进程的操作系统预先都知道对方的操作系统类型，所以就不需要选项协商机制。在过去的几年中， Rlogin协议也派生出几种非 Unix环境的版本。

RFC 1282 [Kantor 1991]详细说明了Rlogin协议。类似于选路信息协议（RIP）的RFC，它是Rlogin用了许多年后才发布的。 [Stevens 1990]的第15章介绍了远程登录的客户进程及服务器进程端的编程，并且给出了 Rlogin的客户进程及服务器进程的完整源代码。 [Comer和Stevens 1993]的第25章和第26章给出了Telnet的客户进程的实现细节和源代码。

26.2.1 应用进程的启动

Rlogin的客户进程和服务器进程使用一个TCP连接。当普通的TCP连接建立完毕之后，客户进程和服务器进程之间将发生下面所述的动作。

1) 客户进程给服务器进程发送4个字符串：(a) 一个字节的0；(b) 用户登录进客户进程主机的登录名，以一个字节的0结束；(c) 登录服务器进程端主机的登录名，以一个字节的0结束；(d) 用户终端类型名，紧跟一个正斜杠“/”，然后是终端速率，以一个字节的0结束。在这里需要两个登录名字，这是因为用户登录客户和服务器的名称有可能不一样。
   由于大多满屏应用程序需要知道终端类型，所以终端类型也必须发送到服务器进程。发送终端速率的原因是因为有些应用随着速率的改变，它的操作也有所变化。例如vi编辑器，当速率比较小的时候，它的工作窗口也变小。所以它不能永远保持同样大小的窗口。
2) 服务器进程返回一个字节的0。
3) 服务器进程可以选择是否要求用户输入口令。这个步骤的数据交互没有什么特别的协议，而被当作是普通的数据进行传输。服务器进程给客户进程发送一个字符串（显示在客户进程的屏幕上），通常是password: 。如果在一定的限定时间内（通常是60秒）客户进程没有输入口令，服务器进程将关闭该连接。
   通常可以在服务器进程的主目录(home directory)下生成一个文件（通常叫 .rhosts），该文件的某些行记录了一个主机名和用户名。如果从该文件中已经记录的主机上用已经记录的用户名进行登录，服务器进程将不提示我们输入口令。但是很多关于安全性的文献，如[Curry 1992]，强烈建议不要采用这种方法，因为这存在安全漏洞。
   如果提示输入口令，那么我们输入的口令将以明文的形式发送到服务器进程。我们所键入的每个字符都是以明文的格式传输的。所以某人只要能够截取网络上的原始传输的分组，他就可以截获用户口令。针对这个问题，新版本的 Rlogin客户程序，例如4.4BSD版本的客户程序，第一次采用了Kerberos安全模型。Kerberos安全模型可以避免用户口令以明文的形式在网络上传输。当然，这要求服务器进程也支持 Kerberos ([Curry 1992]详细描述了Kerberos安全模型)。
4) 服务器进程通常要给客户进程发送请求，询问终端的窗口大小（将在后面解释）。

客户进程每次给服务器进程发送一个字节的内容，并且接收服务器进程的所有返回信息。这在19.2节中已经介绍过了。同样我们也采用了Nagle算法（在19.4节中曾经介绍），该算法可以保证在速率较低的网络上，若干输入字节以单个 TCP报文段传输。操作其实很简单：用户键入的所有东西被发送到服务器，服务器发送给客户的任何信息返回到用户的屏幕上。

另外，服务器和客户之间还可以互相发送命令。在介绍这些命令之前，先介绍需要用到这些命令的场合。

26.2.2 流量控制

默认情况下，流量控制是由 Rlogin的客户进程完成的。客户进程能够识别用户键入的STOP和START的ASCII字符（Control_S和Control_Q），并且终止或启动终端的输出。

如果不是这样，每次我们为终止终端输出而键入的 Control_S字符将沿网络传输到服务器进程，这时服务器进程将停止往网络上写数据。但是在写操作终止之前，服务器进程可能已经往网络上写了一窗口的输出数据。也就是说，在输出停止之前，成千上万的数据字节还将

在屏幕上显示。图26-3显示了这个情况。

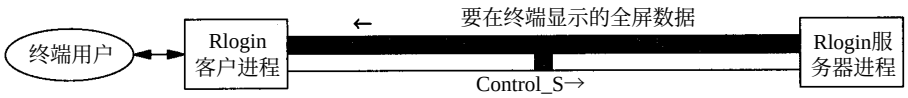


图26-3 服务器进程执行STOP/START的情况

对于一个交互式用户来讲，Control_S字符的响应延时是较大的。

有时候，服务器的应用程序需要解释输入的每个字节，但又不想让客户对它的输入内容进行处理，例如对控制字符如Control_S和Control_Q进行特殊处理（emacs编辑器就是这样的一个例子，它把Control_S和Control_C作为自己的命令）。解决这个问题的办法就是由服务器告诉客户是否要进行流量控制。

26.2.3 客户的中断键

当我们为中断服务器正在运行的进程而键入一个中断字符时（通常是DELETE或Control_C），会发生和流量控制相同的问题。这个情况和图26-3所示的类似，在一条TCP连接的管道上，从服务器进程向客户进程正在发送大量的数据，而客户进程同时在向服务器进程传输中断字符。而我们的本意是要中断字符尽快终止某个进程，使屏幕上不再有任何响应输出。

在流量控制和中断键这两种情况中，流量控制机制很少终止客户进程到服务器进程的数据流。这个方向仅仅包含我们键入的字符。所以对于从客户输出到服务器的特殊输入字符（Control_S和中断字符）不需要采用TCP的紧急方式（urgent mode）。

26.2.4 窗口大小的改变

如果是窗口风格的显示方式，当应用程序在运行的时候，我们还可以动态地改变窗口的大小。一些应用程序（典型的如那些操作整个窗口的应用程序，如全屏编辑器）需要知道窗口大小的变化。目前大多数Unix系统提供这种功能，可以告诉应用程序关于窗口大小的变化。

对于远程登录这种情况，窗口大小的变化发生在客户端，而运行在服务器端的应用程序需要知道窗口大小变化。所以Rlogin的客户需要采用某些方法来通知服务器窗口大小变化的情况以及新窗口的大小。

26.2.5 服务器到客户的命令

现在我们介绍通过TCP连接，Rlogin服务器进程可以发送给客户进程的4条命令。问题是只有一条TCP连接可供使用，所以服务器进程必须给这些命令字节做标记，使得客户进程可以从数据流中识别出这些是命令，而不是显示在终端上。所以我们将使用TCP的紧急方式（在20.8节中曾经介绍）。

当服务器要给客户发送命令时，服务器就进入紧急方式，并且把命令放在紧急数据的最后一个字节中。当客户进程收到这个紧急方式通知时，它从连接上读取数据并且保存起来，直到读到命令字节（即紧急数据的最后一个字节）。这时候客户进程根据读到的命令，再决定对于所读到并保存起来的数据是显示在终端上还是丢弃它。图26-4介绍了这4个命令。

采用TCP紧急方式发送这些命令的一个原因是第一个命令（“清仓输出(flush output)”）需要立即发送给客户，即使服务器到客户的数据流被窗口流量控制所终止。这种情况下，即服

务器到客户的输出被流量控制所终止的情况是经常发生的，这是因为运行在服务器的进程的输出速率通常大于客户终端的显示速率。另一方面，客户到服务器的数据流很少被流量控制所终止，因为这个方向的数据流仅仅包含用户所键入的字符。

字节	描　述
0x02	清仓输出。客户丢弃所有从服务器收到的数据，直到命令字节（紧急数据的最后一个字节）。客户还丢弃任何有可能被缓存的挂起输出 (pending output)。当服务器收到客户发出的中断命令时，就发送此命令
0x10	客户停止执行流量控制
0x20	客户继续进行流量控制处理
0x80	客户立即响应，将当前窗口大小发送给服务器，并在今后当窗口大小变化时通知服务器。通常，当连接建立后，服务器就立即发送这个命令

图26-4 服务器到客户的Rlogin命令

回忆一下图20-14中的例子，在那里我们介绍了即使窗口大小是 0时，紧急通知通过网络进行传输的情况（在下节中，我们还将介绍一个类似的例子）。其他的3个命令实时性并不特别强，但为了简单起见，也采用了和第一个命令相同的技术。

26.2.6 客户到服务器的命令

对于客户到服务器的命令，只定义了一条命令，那就是：将当前窗口大小发送给服务器。当客户的窗口大小发生变化时，客户并不立即向服务器报告，除非收到了服务器发来的 0x80命令（图26-4中有介绍）。

同样，由于只存在一条 TCP连接，客户必须对在连接上传输的该命令字节进行标注，使得服务器可以从数据流中识别出命令，而不是把它发送到上层的应用程序中去。处理的方法就是在两个字节的0xff后面紧跟着发送两个特殊的标志字节。

对于窗口大小命令，两个标志字节是 ASCII码的字符‘ s ’。之后是4个16 bit长的数据（按网络字节顺序），分别是：行数（例如， 25），每列的字符数（例如， 80），X方向的像素数量，Y方向的像素数量。通常情况下，后两个 16bit是0，因为在Rlogin服务器进程调用的应用程序中，通常是以字符为单位来度量屏幕的，而不是像素点。

上面我们介绍的从客户进程到服务器进程的命令采用带内信令（ in-band signaling），这是因为命令字节和其他的普通数据一起传输。选择 0xff字节来表示这个带内信令的原因是：一般用户的操作不会产生0xff这个字节。所以说Rlogin是不完备的，如果我们采用某种方法，使得通过键盘就可以产生两个连续的 0xff字节，而且正好在这之前是两个 ASCII的‘ s ’字符，那么下面的8个字节就会被误认为是窗口大小了。

图26-4中介绍的是从服务器到客户的 Rlogin命令，由于大多数的 API采用的技术叫做“ 带外数据（out-of-band data）”，所以我们就称它为带外信令 (out-of-band signaling)。但是回忆一下在20.8节中对TCP紧急方式的讨论，在那里我们说紧急方式数据不是带外数据，命令字节是按照普通数据流进行传输的，特殊之处是采用了紧急指针。

既然带内信令被用来传输从客户到服务器的命令，那么服务器进程必须检查从客户进程收到的每个字节，看看是否有两个连续的 0xff字节。但是对于采用带外信令的、从服务器传输到到客户的命令，客户进程不需要检查收到的每个字节，除非服务器进程进入了紧急方

式。即使在紧急方式下，客户进程也仅仅需要留意紧急指针所指向的字节。而且由于从客户进程到服务器的数据流量和相反方向的数据流量之比是 1:20，这就暗示带内信令适合于数据量比较小的情况（从客户到服务器），而带外信令适合于数据量比较大的情况（从服务器到客户）。

26.2.7 客户的转义符

通常情况下，我们向Rlogin客户进程键入的信息将传输到服务器进程。但是有些时候，我们并不需要把键入的信息传输到服务器，而是要和 Rlogin客户进程直接通信。方法是在一行的开头键入代字符(tilde)“ ~ ”，紧跟着是下列4个字符之一：

1) 以一个句号结束客户进程。

2) 以文件结束符（通常是Control_D）结束客户进程。

3) 以任务控制挂起符（通常是Control_Z）挂起客户进程。

4) 以任务控制延迟挂起符（通常是Control_Y）来挂起仅仅是客户进程的输入。这时，不管客户运行什么程序，键入的任何信息将由该程序进行解释，但是从服务器发送到客户的信息还是输出到终端上。这非常适合当我们需要在服务器上运行一个长时间程序的场合，我们既想知道该程序的输出结果，同时还想在客户上运行其他程序。

只有当客户进程的Unix系统支持任务控制时，后两个命令才有效。

26.3 Rlogin的例子

在这里举两个例子：第一个是当Rlogin会话建立的时候，客户和服务器的协议交互；从第二个例子可以看到，当用户键入中断键以取消正在服务器运行的程序时，服务器将产生很多输出。在图19-2中，我们给出了通常情况下， Rlogin会话上的数据流交互情况。

26.3.1 初始的客户-服务器协议

图26-5显示的是从主机bsdi到服务器svr4的Rlogin建立一个连接时的时间系列(在图中，去掉了通常的TCP连接的建立过程，窗口通告以及服务类型信息)。

上节介绍的协议对应图中的报文段 1~9。客户发送一个字节的 0（报文段1）之后发送3个字符串（报文段 3）。在本例中，这 3个字符串分别是： rstevens（客户的登录名）、rstevens（服务器的登录名）和 ibmpc3/9600（终端类型和速率）。当服务器确认了这些信息后回送一个字节的0（报文段5）。

然后服务器发送窗口请求命令（报文段 7）。这是采用TCP紧急方式发送的，我们又一次看到一个实现(SVR4)采用较老的但更普通的解释，即紧急指针指明的序号是紧急数据的最后一个字节加1。客户回送12字节的数据：2字节的0xff，2字节的‘ s ’，4个16 bit长度的窗口数据。

下面的4个报文段（ 10，12，14和16）是由服务器发送的，是从服务器操作系统的问候(greeting)。之后报文段18是一个7字节长度的外壳进程提示符“ svr4% ”。

客户输入的信息如图 19-2所示，每次发送一个字节。客户和服务器都可以主动中断该连接。如果我们输入一个命令，让服务器的外壳程序终止运行，那么服务器将中断该连接。如果我们给Rlogin客户键入一个转移符（通常是一个“ ~ ”），紧跟着一个句点或者是一个文件结束符号，那么客户将主动关闭该连接。

图26-5中，客户进程的端口号是1023，这是由IANA分配的（在1.9节中介绍）。Rlogin协议要求客户进程用小于1024的端口号，术语叫做保留端口。在Unix系统中，客户进程一般不能使用保留端口号，除非客户进程具有超级用户权限。这是客户进程和服务器进程相互鉴别的一部分，这种鉴别可以使得用户不需要口令而可以登录。[Stevens 1990]详细讨论了客户进程和服务器进程相互鉴别的过程和有关保留端口号的问题。

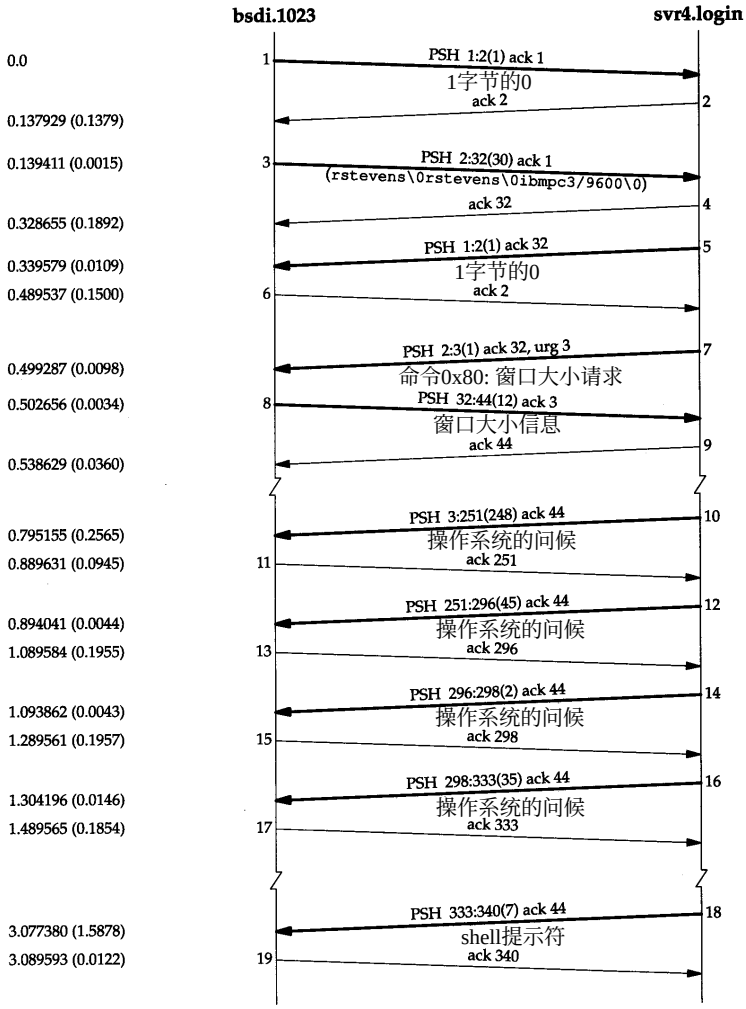


图26-5 Rlogin连接时间次序

26.3.2 客户中断键

让我们看一下另外一个例子，这个例子涉及到TCP的紧急方式。当数据流已经终止时，我们键入中断键。这个例子要用到前面讲到的很多TCP算法如：紧急方式、糊涂窗口避免技术、窗口流量控制和坚持计时器。在主机sun上运行客户进程。我们登录到主机bsdi，向终端输出一个大文本文件，然后键入Control_S中断输出。当输出停止时，我们键入中断键（DELETE）以异常方式中止该进程。

```
sun % rlogin bsdi
                                          所有操作系统的问候
bsdi % cat /usr/share/misc/termcap        向终端输出大文件
                                          大量的终端输出
                                          键入Control_S以中断输出,
                                          然后等待直到输出停止
^?                                        键入中断键，而这被回显了,
bsdi %                                    然后输出了提示符
```

下面这些要点是关于客户、服务器和连接的状态的概述：

1) 键入Control_S以停止终端的输出。

2) 用户终端的输出缓存很快被填满，所以Rlogin的客户向终端的写操作被阻塞。

3) 此时客户也不能从网络连接上读取数据，所以客户的TCP接收缓存也将被填满。

4) 当接收缓存已满时，客户进程的TCP会向服务器进程的TCP通告现在的接收窗口是0。

5) 当服务器收到客户的窗口为0时，将停止向客户发送数据，这样，服务器的发送缓存也将被填满。

6) 由于发送缓存已满，所以Rlogin服务器进程将停止。这样，Rlogin服务器将不能从服务器运行的应用程序（cat）处读取数据。

7) 当cat程序的输出缓存也被填满时，cat也将停止。

8) 然后我们用中断键来终止服务器上的cat程序。这个命令从客户的TCP传输到服务器的TCP，这是因为该方向的数据传输没有被流量控制所终止。

9) cat应用程序收到中断命令并且终止。这使得它的输出缓存（也就是Rlogin服务器进程读取数据的地方）被清空，这将唤醒Rlogin服务器进程。然后Rlogin服务器进程进入紧急方式，向客户进程发送“清仓输出”命令（0x02）。

图26-6概括了从服务器到客户的数据流（图中的序号就是下面将介绍的图中的时间系列）。

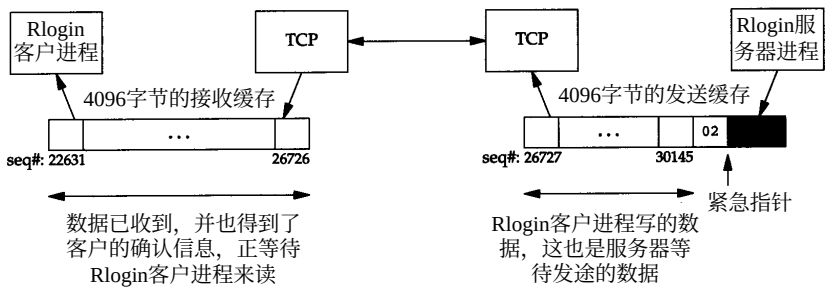


图26-6 Rlogin例子中，服务器进程到客户进程的数据流概述

发送缓存的阴影部分是4096字节的缓存中没有被使用的部分。图26-7是该例子的时间系列。

在报文段1~3，服务器进程向客户进程发送满长度（即1024字节）的TCP报文段。由于此时客户进程不能向终端写信息，客户进程也不能从网络上读数据，所以在报文段4中，客户进程向服务器进程发送ACK确认时，告诉服务器进程此时接收窗口是1024个字节。在报文段5中，服务器进程发送的数据长度就不再是满长度的了。同样，报文段6中客户进程的确认信号所带的接收窗口大小是此时接收缓存的空余字节长度。那么在报文段5中，客户进程ACK信号中为什么接收窗口大小是349而不是0呢？这是因为如果发送的是0（糊涂窗口避免技术），那么窗口指针将右边界移动到了左边界，而这是绝对不能发生的（见20.3节）。当服务器进程收到报

文段6的ACK信号后，它就不能再发送全长的数据报了，这时候它就采用糊涂窗口症避免技术，不发送任何东西，同时置一个5秒的坚持计时器。当计时器超时，服务器进程就发送一个349字节大小的数据（如报文段7）。由于此时客户进程依然不能输出接收缓存的信息，所以接收缓存将被填满，客户进程将发送ACK信号，此时接收窗口大小为0（如报文段8）。

这时候我们键入中断键并且以报文段9显示的那样传输。此时的接收窗口大小依然为0。当服务器进程接收到该中断键后，服务器进程把它发送给应用程序（cat），应用程序就终止。由于应用程序被终端中断键所终止，应用程序就清空它的输出缓存。服务器进程发现该变化后就通过TCP紧急方式向客户进程发送“清仓输出”命令，这如报文段10所示。注意命令字节0x02放在第30146字节中（紧急指针减1）。报文段10告诉客户进程在命令字节前还有3419个字节（从26727到30145）在服务器进程的发送缓存中等待发送。

报文段10采用紧急通知方式发送，包含了服务器进程向客户进程发送的下一个字节（序号是26727）。它不包含“清仓输出”命令字节。记得在22.2节中曾经介绍过，发送进程可以发送一个字节的数据来试探对方的接收窗口是否关闭。报文段10就是采用了这个原理。然后客户进程TCP就立即发送如报文段11所示的数据。虽然此时接收窗口还是0，但是在客户进程内部，由于客户进程的TCP收到了对方的紧急通知，它把该通知告诉客户进程，客户进程就知道服务器进程已经进入了紧急方式了。

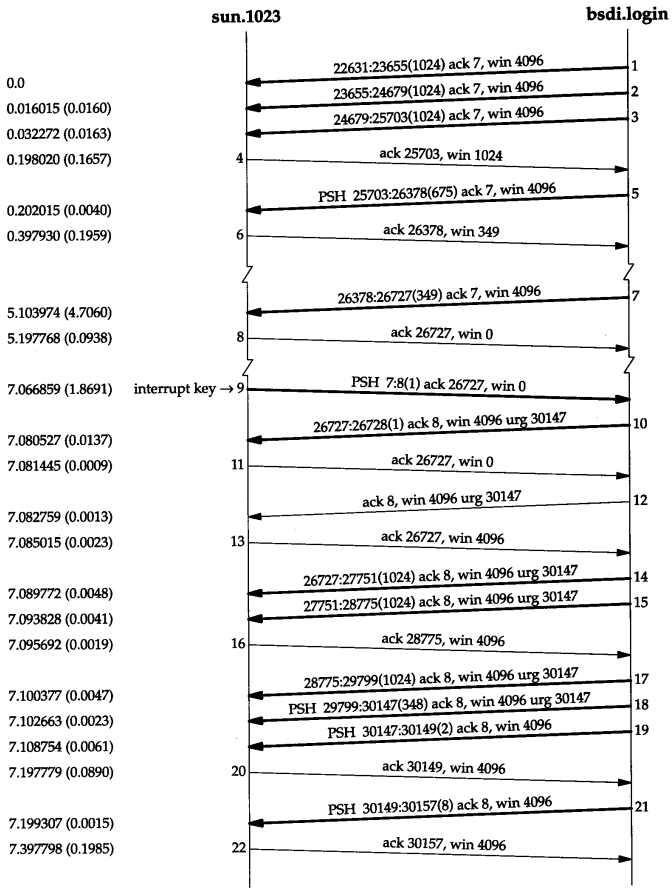


图26-7 Rlogin举例：当客户程停止输出然后终止服务器的应用程序的情况

当Rlogin客户进程从它的TCP收到了紧急通知，并且客户进程开始读取已经在输入缓存中等待被读取的数据时，接收窗口就会重新打开（报文段 13）。然后服务器进程就开始正常发送数据（报文段14, 15, 17和18）。注意报文段18的数据报中包含紧急数据的最后一个字节的数据（序号30146），该字节包含服务器进程发送给客户进程的命令字节。当客户进程收到该命令后，它就丢弃报文段14、15、17和18所收到的数据，并且清空终端的输出缓存。在报文段 19中的下两个字节是中断键的回显“ ^? ”。最后一个报文段（21）包含了客户进程的外壳提示符。

这个例子描述了当用户键入中断键后，连接的双方数据如何被存储的情况。如果这些动作仅仅丢弃在服务器的 3419个字节数据，而不丢弃已经在客户的 4096个字节的数据，那么这些已经在客户的终端输出缓存中的 4096字节数据将输出到终端上。

26.4 Telnet协议

Telnet协议可以工作在任何主机（例如，任何操作系统）或任何终端之间。 RFC 854 [Postel 和Reynolds 1983a]定义了该协议的规范，其中还定义了一种通用字符终端叫做网络虚拟终端NVT（Network Virtual Terminal）。NVT是虚拟设备，连接的双方，即客户机和服务器，都必须把它们的物理终端和 NVT进行相互转换。也就是说，不管客户进程终端是什么类型，操作系统必须把它转换为 NVT格式。同时，不管服务器进程的终端是什么类型，操作系统必须能够把NVT格式转换为终端所能够支持的格式。

NVT是带有键盘和打印机的字符设备。用户击键产生的数据被发送到服务器进程，服务器进程回送的响应则输出到打印机上。默认情况下，用户击键产生的数据是发送到打印机上的，但是我们可以看到这个选项是可以改变的。

26.4.1 NVT ASCII

术语NVT ASCII代表7比特的ASCII字符集，网间网协议族都使用 NVT ASCII。每个7比特的字符都以8比特格式发送，最高位比特为 0。

行结束符以两个字符 CR（回车）和紧接着的 LF（换行）这样的序列表示。以 \r\n来表示。单独的一个 CR也是以两个字符序列来表示，它们是 CR和紧接着的 NUL（字节 0），以 \r\0表示。

在下面的章节中可以看到， FTP, SMTP, Finger和Whois协议都以NVT ASCII来描述客户命令和服务器的响应。

26.4.2 Telnet命令

Telnet通信的两个方向都采用带内信令方式。字节 0xff（十进制的 255）叫做 IAC（interpret as command，意思是“ 作为命令来解释 ”）。该字节后面的一个字节才是命令字节。如果要发送数据255，就必须发送两个连续的字节 255（在前面一节中我们讲到数据流是 NVT ASCII，它们都是 7bit的格式，这就暗示着 255这个数据字节不能在 Telnet上传输。其实在Telnet中有一个二进制选项，在 RFC856[Postel和Reynolds 1983b]中有定义，关于这点我们没有讨论，该选项允许数据以 8bit进行传输）。图26-8列出了所有的Telnet命令。

由于这些命令中很多命令很少用到，所以对于一些重要的命令，如果在下面章节的例子或叙述中遇到，我们再做解释。

名称	代码(十进制)	描 述
EOF	236	文件结束符
SUSP	237	挂起当前进程（作业控制）
ABORT	238	异常中止进程
EOR	239	记录结束符
SE	240	子选项结束
NOP	241	无操作
DM	242	数据标记
BRK	243	中断
IP	244	中断进程
AO	245	异常中止输出
AYT	246	对方是否还在运行？
EC	247	转义字符
EL	248	删除行
GA	249	继续进行
SB	250	子选项开始
WILL	251	选项协商（图26-9）
WONT	252	选项协商
DO	253	选项协商
DONT	254	选项协商
IAC	255	数据字节255

图26-8 当前一个字节是IAC（255）时的Telnet命令集

26.4.3 选项协商

虽然我们可以认为Telnet连接的双方都是NVT，但是实际上Telnet连接双方首先进行交互的信息是选项协商数据。选项协商是对称的，也就是说任何一方都可以主动发送选项协商请求给对方。

对于任何给定的选项，连接的任何一方都可以发送下面4种请求的任意一个请求。

1) WILL：发送方本身将激活(enable)选项。

2) DO：发送方想叫接收端激活选项。

3) WONT：发送方本身想禁止选项。

4) DON'T：发送方想让接收端去禁止选项。

	发送方		接收方	描 述
1.	WILL	→		发送方想激活选项
		←	DO	接收方说同意
2.	WILL	→		发送方想激活选项
		←	DONT	接收方说不同意
3.	DO	→		发送方想让接收方激活选项
		←	WILL	接收方说同意
4.	DO	→		发送方想让接收方激活选项
	WONT	←		接收方说不同意
5.	WONT	→		发送方想禁止选项
		←	DONT	接收方必须说同意
6.	DONT	→		发送方想让接收方禁止选项
		←	WONT	接收方必须说同意

图26-9 Telnet选项协商的6种情况

由于Telnet规则规定，对于激活选项请求（如1和2），有权同意或者不同意。而对于使选项失效请求（如3和4），必须同意。这样，4种请求就会组合出6种情况，如图26-9所示。

选项协商需要3个字节：一个IAC字节，接着一个字节是WILL, DO, WONT和DONT这四

者之一，最后一个ID字节指明激活或禁止选项。现在，有40多个选项是可以协商的。Assigned Number RFC文档中指明选项字节的值，并且一些相关的RFC文档描述了这些选项。图26-10显示了在本章中会出现的选项代码。

Telnet的选项协商机制和Telnet协议的大部分内容一样，是对称的。连接的双方都可以发起选项协商请求。但我们知道，远程登录不是对称的应用。客户进程完成某些任务，而服务器进程则完成其他一些任务。下面我们将看到，某些Telnet选项仅仅适合于客户进程（例如要求激活行模式方式），某些选项则仅仅适合于服务器进程。

选项标识(十进制)	名　称	RFC
1	回显	857
3	抑制继续进行	858
5	状态	859
6	定时标记	860
24	终端类型	1091
31	窗口大小	1073
32	终端速率	1079
33	远程流量控制	1372
34	行方式	1184
36	环境变量	1408

图26-10　本章中将讨论的Telnet选项代码

26.4.4　子选项协商

有些选项不是仅仅用“激活”或“禁止”就能够表达的。指定终端类型就是一个例子，客户进程必须发送用一个ASCII字符串来表示终端类型。为了处理这种选项，我们必须定义子选项协商机制。

在RFC1091[VanBokkelen 1989]中定义了如何表示终端类型这样的子选项协商机制。首先连接的某一方（通常是客户进程）发送3个字节的字符序列来请求激活该选项。

<IAC, WILL, 24>

这里的24（十进制）是终端类型选项的ID号。如果收端（通常是服务器进程）同意，那么响应数据是：

<IAC, DO, 24>

然后服务器进程再发送如下的字符串：

<IAC, SB, 24, 1, IAC, SE>

该字符串询问客户进程的终端类型。其中SB是子选项协商的起始命令标志。下一个字节的“24”代表这是终端类型选项的子选项（通常SB后面的选项值就是子选项所要提交的内容）。下一个字节的“1”表示“发送你的终端类型”。子选项协商的结束命令标志也是IAC，就像SB是起始命令标志一样。如果终端类型是ibmpc，客户进程的响应命令将是：

＜IAC, SB, 24, 0‘I’, ‘B’, ‘M’, ‘P’, ‘C’, IAC, SE＞

第4个字节“0”代表“我的终端类型是”（在Assigned Numbers RFC文档中有正式的关于终端类型的数值定义，但是最起码在Unix系统之间，终端类型可以用任何对方可理解的数据进行表示。只要这些数据在termcap或terminfo数据库中有定义）。在Telnet子选项协商过程中，终端类型用大写表示，当服务器收到该字符串后会自动转换为小写字符。

26.4.5　半双工、一次一字符、一次一行或行方式

对于大多数Telnet的服务器进程和客户进程，共有4种操作方式。

1. 半双工

这是Telnet的默认方式，但现在却很少使用。NVT默认是一个半双工设备，在接收用户输入之前，它必须从服务器进程获得GO AHEAD（GA）命令。用户的输入在本地回显，方向是从NVT键盘到NVT打印机，所以客户进程到服务器进程只能发送整行的数据。

虽然该方式适用于所有类型的终端设备，但是它不能充分发挥目前大量使用的支持全双工通信的终端功能。RFC 857 [Postel 和Reynolds 1983c]定义了ECHO选项，RFC 858 [Postel和Reynolds 1983d]定义了SUPPRESS GO AHEAD（抑制继续进行）选项。如果联合使用这两个选项，就可以支持下面将讨论的方式：带远程回显的一次一个字符的方式。

2. 一次一个字符方式

这和前面的Rlogin工作方式类似。我们所键入的每个字符都单独发送到服务器进程。服务器进程回显大多数的字符，除非服务器进程端的应用程序去掉了回显功能。

该方式的缺点也是显而易见的。当网络速度很慢，而且网络流量比较大的时候，那么回显的速度也会很慢。虽然如此，但目前大多数Telnet实现都把这种方式作为默认方式。

我们将看到，如果要进入这种方式，只要激活服务器进程的SUPPRESS GO AHEAD选项即可。这可以通过由客户进程发送DO SUPPRESS GO AHEAD（请求激活服务器进程的选项）请求完成，也可以通过服务器进程给客户进程发送WILL SUPPRESS GO AHEAD（服务器进程激活选项）请求来完成。服务器进程通常还会跟着发送WILL ECHO，以使回显功能有效。

3. 一次一行方式

该方式通常叫做准行方式（kludge line mode），该方式的实现是遵照RFC 858的。该RFC规定：如果要实现带远程回显的一次一个字符方式，ECHO选项和SUPPRESS GO AHEAD选项必须同时有效。准行方式采用这种方式来表示当两个选项的其中之一无效时，Telnet就是工作在一次一行方式。在下节中我们将介绍一个例子，可以看到如何协商进入该方式，并且当程序需要接收每个击键时如何使该方式失效。

4. 行方式

我们用这个术语代表实行方式选项，这是在RFC 1184[Borman 1990]中定义的。这个选项也是通过客户进程和服务器进程进行协商而确定的，它纠正了准行方式的所有缺陷。目前比较新的Telnet实现支持这种方式。

图26-11是不同的Telnet客户进程和服务器进程之间默认的操作方式。“char”表示一次一个字符方式，“kludge”表示准行方式，“linemode”表示如RFC 1184定义的实行方式。

客户端	服务器端					
	SunOS 4.1.3	Solaris 2.2	SVR4	AIX 3.2.2	BSD/386	4.4BSD
SunOS 4.1.3	char	char	char	char	kludge	kludge
Solaris 2.2	char	char	char	char	kludge	kludge
SVR4	char	char	char	char	kludge	kludge
AIX 3.2.2	char	char	char	char	kludge	kludge
BSD/386	char	char	char	char	linemode	linemode
4.4BSD	char	char	char	char	linemode	linemode

图26-11 不同的Telnet客户进程和服务器进程之间默认的操作方式

从图中可以看出，只有当客户进程和服务器进程都是BSD/386或4.4BSD的时候才支持实行方式。当服务器进程的操作系统是这两者之一时，如果客户进程不支持实行方式，才会协商进入准行方式。从图中还可以看出，其实任何类型的客户进程和服务器进程都支持准行方式，但是一般都不把它作为默认方式，除非服务器进程指定。

26.4.6 同步信号

Telnet以Data Mark命令（即图26-8中的DM）作为同步信号，该同步信号是以TCP紧急数据形式发送的。DM命令是随数据流传输的同步标志，它告诉收端回到正常的处理过程上来。Telnet的双方都可以发送该命令。

当一端收到对方已经进入了紧急方式的通知后，它将开始读数据流，一边读一边丢弃所读的数据，直到读到Telnet命令为止。紧急数据的最后一个字节就是DM字节。采用TCP紧急方式的原因就是：即使TCP数据流已经被TCP流量控制所终止，Telnet命令也可以在连接上传输。

在下节中我们将看到Telnet同步信号的使用情况。

26.4.7 客户的转义符

和Rlogin的客户进程一样，Telnet客户进程也可以使用户直接和客户进程进行交互，而不是被发送到服务器进程。通常客户的转义字符是Control_]（control键和右中括号键，通常以“^]”表示）。这使得客户进程显示它的提示符，通常是“telnet>”。这时候有很多命令可供用户选择，以改变连接的特性或打印某些信息。大多数的Unix客户进程提供“help”命令，该命令将显示所有可用的命令。

在下节中我们将看到客户进程转义的例子，以及此时可以输入的命令。

26.5 Telnet举例

在这里我们将介绍在三种不同的操作方式下Telnet选项协商的情况。这些方式包括：单字符方式、实行方式和准行方式。同样我们还将讨论当用户在服务器端按了中断键退出了一个正在运行的进程后，系统的运行情况。

26.5.1 单字符方式

首先介绍基本的单字符方式，该方式类似于Rlogin。用户在终端输入的每个字符都将由终端发送到服务器进程，服务器进程的响应也将以字符方式回显到终端上。在这里运行的是一个新的客户进程BSD/386，它试图激活很多新的选项，服务器进程还是运行老的SVR4，我们将看到很多选项被服务器拒绝。

为了看到服务器和客户机之间选项协商的内容，我们将激活客户进程的一个选项来显示所有的选项协商。同样我们运行tcpdump来获得数据报交换的时间次序。图26-12显示了这个交互会话。

在图中，我们已经对由SENT或RCVD开头的选项协商的每一步都进行了标注。关于每一步的解释如下：

1) 客户发起SUPPRESS GO AHEAD选项协商。由于GO AHEAD命令通常是由服务器发送给客户的，而且客户希望服务器激活该选项，因此该选项的请求方式是DO（由于激活这一选项将会禁止GA命令的发送，上述过程很容易让人混淆）。在第10行可以看到服务器进程同意该选项。

2) 客户进程要按照在RFC 1091[VanBokkelen 1989]中的定义发送终端类型。这对Unix类型的客户进程来讲是很普通的。因为客户进程要激活本地的选项，所以该选项的请求方式是WILL。

```
bsdi % telnet                                   调用客户进程，不带任何命令行选项

telnet> toggle options                          告诉客户进程显示所有的选项协商过程
Will show option processing.

telnet> open svr4                               现在和服务器建立连接
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.

SENT DO SUPPRESS GO AHEAD                       1. (后面将讨论的行序号)
SENT WILL TERMINAL TYPE                         2.
SENT WILL NAWS                                  3.
SENT WILL TSPEED                                4.
SENT WILL LFLOW                                 5.
SENT WILL LINEMODE                              6.
SENT WILL ENVIRON                               7.
SENT DO STATUS                                  8.
RCVD DO TERMINAL TYPE                           9.
RCVD WILL SUPPRESS GO AHEAD                     10.
RCVD DONT NAWS                                  11.
RCVD DONT TSPEED                                12.
RCVD DONT LFLOW                                 13.
RCVD DONT LINEMODE                              14.
RCVD DONT ENVIRON                               15.
RCVD WONT STATUS                                16.
RCVD IAC SB TERMINAL-TYPE SEND                  17.
SENT IAC SB TERMINAL-TYPE IS "IBMPC3"           18.
RCVD WILL ECHO                                  19.
SENT DO ECHO                                    20.
RCVD DO ECHO                                    21.
SENT WONT ECHO                                  22.


UNIX(r) System V Release 4.0 (svr4)

RCVD DONT ECHO                                  23.
login: rstevens                                 我们键入用户和口令，服务进程不回显这些数据，
Password:                                       然后操作系统问候输出，然后是外壳提示符
```

图26-12 Telnet双方选项协商的初始化过程

3) NAWS的意思是“协商窗口大小”，它在RFC 1073 [Waitzman]中有定义。如果服务器进程同意该选项（实际上不同意，见11行），客户进程就要发送终端窗口的行、列大小的子选项。而且只要窗口大小发生变化，客户进程随时都将向服务器进程发送这一子选项（这和图26-4中Rlogin的0x80命令类似）。

4) TSPEED选项允许发送方（通常是客户进程）发送它的终端速率，这在 RFC 1079 [Hedrick 1988b]中有定义。如果服务器进程同意（实际上不同意，见12行），客户进程将发送其发送速率和接收速率的子选项。

5) LFLOW代表“本地流量控制”，这在RFC1371 [Hedrick 和Borman 1992]中定义。客户进程给服务器进程发送该选项，表示客户进程希望用命令方式激活或禁止流量控制。如果服务器进程同意（实际上不同意，见13行），只要Control_S和Control_Q进程需要在客户进程和服务器进程进行切换，客户进程都要向服务器进程发送子选项（这类似于图 26-4中Rlogin的0x10和0x20命令）。正如在关于Rlogin的讨论中我们所提到的那样，由客户进程进行流量控制的效果比由服务器进程来完成要好。

6) LINEMODE代表在26.4中所说的实行方式。所有终端字符的处理由 Telnet客户进程完成（例如回格，删除行等），然后整行发送给服务器进程。在本节后面，我们将介绍一个例子。该选项同样被服务器进程拒绝，如14行所示。

7) ENVIRON选项允许客户进程把环境变量发送给服务器进程，这在 RFC 1408 [Borman

1993a]中有定义。这样就可以把客户进程的用户环境变量自动传播到服务器进程。在 15行，服务器进程拒绝该选项（Unix中的环境变量通常是大写字母，紧跟一个等号，然后是一个字符串值，当然这只是一个惯例而已）。默认情况下，BSD/386 Telnet客户进程发送两个环境变量：DISPLAY和PRINTER，前提是这两个变量已经定义并且有效。Telnet用户可以定义其他一些要发送的环境变量。

8) STATUS选项（RFC 859 [Postel 和Reynolds 1983e]中定义）允许连接的一方询问对方对Telnet选项目前状态的理解。在这个例子中，客户进程要求对方激活选项（DO）。如果服务器进程同意（实际上不同意，见16行），客户进程就可以要求服务器进程以子选项的形式发送它的状态值。

9) 这是服务器进程的第一个响应。服务器进程同意激活终端类型选项（几乎所有的 Unix类型的服务器进程都支持该选项）。但现在客户进程还不能立即发送它的终端类型。它必须要等到服务器进程用子选项的形式询问终端类型的时候才能够发送（17行）。

10) 服务器进程同意抑制发送GO AHEAD命令。

11) 服务器进程不同意客户进程发送它的窗口大小。

12) 服务器进程不同意客户进程发送它的终端速率。

13) 服务器进程不同意客户进程实施流量控制。

14) 服务器进程不同意客户进程激活行方式选项。

15) 服务器进程不同意客户进程发送环境变量。

16) 服务器进程不发送状态信息。

17) 这是服务器进程要求客户进程发送终端类型的子选项。

18) 客户进程把终端类型“IBMPC3”以6字节的字符串形式发送给服务器进程。

19) 服务器进程要求客户进程发起请求，要求服务器进程激活回显选项。这是本例中服务器进程第一次主动发起选项协商。

20) 客户进程同意由服务器进程实现回显功能。

21) 服务器进程要求客户进程实现回显功能。这个命令是多余的，它只是将前两行进行了交换。这是目前大多数Unix的Telnet服务器进程判断客户进程是否运行 4.2BSD或更新的BSD版本时的一个方法。如果客户进程回送 WILL ECHO，就表明客户进程运行的是老版本的4.2BSD，不支持TCP的紧急方式（在这种情况下就不能采用TCP紧急方式）。

22) 客户进程回送WONT ECHO，表示它不是一台4.2BSD主机。

23) 对于客户进程回送的WONT ECHO，服务器进程以DONT ECHO作为响应。

图26-13显示的是本例中服务器进程和客户进程交互的时间系列（去掉了连接建立部分）。

报文段1包含了图26-12中的1~8行。该报文段中包含24个字节数据，每个选项占3个字节。这是客户进程发起的选项协商。该报文段显示多个Telnet选项可以打在一个TCP段中发送。

报文段3是图26-12中的第9行，即DO TERMINAL TYPE命令。报文段5包含下面的8个选项协商中服务器进程的响应，即图 26-12中的10~17行。该报文段的长度是27个字节，因为10~16行是常规选项，每个占3个字节，而17行的子选项部分占6个字节。报文段6包含12个字节，和18行对应，这是客户发送它的终端类型的子选项。

报文段8（53个字节）包含两个Telnet命令和47字节的输出数据。前面的两个服务器进程发送Telnet命令占6字节，：WILL ECHO和DO ECHO（19和21行）。后47个字节的数据是：

```
\r\n\r\nUNIX(r) System V Release 4.0 (svr4) \r\n\r\0\r\n\r\0
```

前面4个字节数据在字符串输出之前产生两个空行。两字节的字符序列“ \r\n ”在Telnet中被认为是换行命令，而两字节的字符序列“ \r\0 ”则被认为是回车命令。这表明数据和命令可以在一个数据段中传输。 Telnet服务器进程和客户进程必须扫描接收到的每个字符，寻找 IAC 字节并执行它后续的命令。

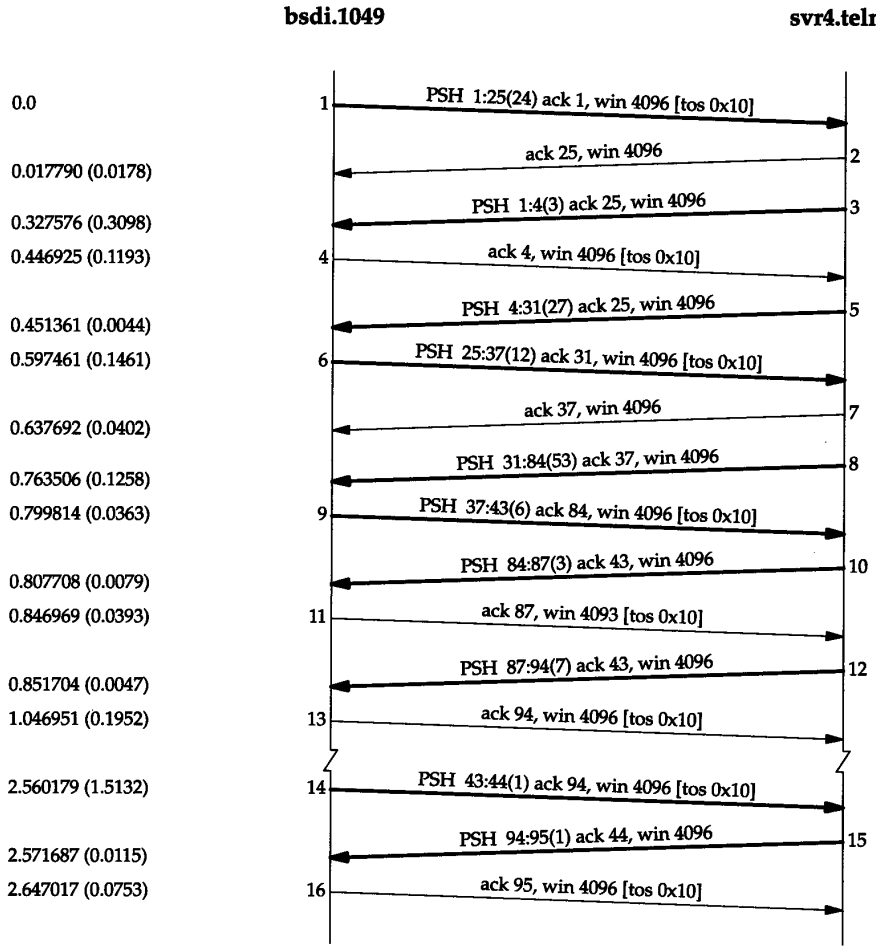


图26-13 Telnet服务器进程和客户进程选项协商初始化过程

报文段9包含客户进程发送的最后两个选项： 20和22行。23行是报文段10的响应，也是服务器进程发送的最后一个选项数据。

从现在开始，双方就可以交互数据了。当然在交互数据的过程中还可以进行选项协商，我们在该例子中就不多介绍了。报文段 12是服务器进程发送的提示符“ login: ”。报文段14是用户输入的登录用户名的第一个字符，它的回显在报文段 15中。这和我们在 19.2节中介绍的Rlogin交互类似：客户进程每次发送一个字符，服务器进程完成回显工作。

图26-12中的选项协商由客户进程初始化的，但是在本书中我们已经介绍了用Telnet客户进程连接某些标准服务器进程如：日间（daytime）服务器、回显(echo)服务器等情况。当然我们介绍这些的目的是为了描述 TCP的各种特性。但考察这些例子中

的分组交换，如图18-1，我们并没有看到客户进程发起的选项协商。为什么？这是因为在Unix系统中，除非使用标准的Telnet端口号23，否则客户进程不进行选项协商。这个特性使得Telnet客户进程可以使用标准的NVT同其他一些非Telnet服务器进程交换数据。我们已经在日间服务器、回显服务器和丢弃(discard)服务器中使用了这个特性，在后面章节介绍FTP和SMTP服务器的时候我们还将使用该特性。

26.5.2 行方式

为了描述Telnet的行方式选项协商过程，我们在主机 bsdi运行客户进程，服务器是位于`vangogh.vs.berkeley.edu`节点运行4.4BSD操作系统的一台主机。BSD/386和4.4BSD都支持这个选项。

我们不详细讨论所有的报文、选项和子选项协商过程，因为这个过程和前面的例子类似，而且对于行方式选项我们已经论述得比较清楚。下面我们仅仅讨论在选项协商中的一些区别。

1) 对于BSD/386希望协商的选项例如：窗口大小、本地流量控制、状态、环境变量和终端速率等，4.4BSD服务器进程都支持。

2) 4.4BSD服务器进程将协商一个 BSD/386客户进程不支持的新选项：鉴别（为避免以明文形式在网络上传输用户口令）。

3) 和上个例子一样，客户进程发送 WILL LINEMODE选项，由于服务器进程支持该选项，所以服务器进程发送DO LINEMODE。此时客户进程以子选项形式给服务器进程发送16个特定字符。这些字符是能影响客户进程的特定终端字符值：如中断字符，文件结束符等。服务器进程给客户进程发送一个子选项，让客户进程处理所有的输入，执行所有的编辑功能（删除字符，删除行等）。客户进程把除控制字符以外的字符以行的形式发送给服务器进程。服务器进程还要求客户进程把所有中断键和信号键转换为相应的 Telnet字符。例如中断键是Control_C，我们可以按Control_C来中断服务器端的某个进程。客户进程必须把Control_C转换为Telnet的IP命令（<IAC, IP>）传输给服务器进程。

4) 当用户输入口令时情况也有所不同。在 Rlogin和一次一字符方式的 Telnet中，都是由服务器进程负责回显，所以当服务器进程读到口令时，它并不回显这些字符。但在行方式中由客户进程负责回显。下面这些交互过程将处理这种情况：

   (a) 服务器进程发送WILL ECHO，以告诉客户进程：服务器进程将处理回显。

   (b) 客户进程回送DO ECHO。

   (c) 服务器进程向客户进程发送字符串`Password:`，客户进程把它发送到终端上。

   (d) 然后用户输入口令，当用户按下 RETURN键的时候，客户进程把口令发送给服务器进程。此时口令不回显，因为客户进程认为服务器进程将回显它。

   (e) 由于口令的结束符 RETURN没有回显，服务器进程发送两字节字符序列 CR和LF以移动光标。

   (f) 服务器进程发送WONT ECHO。

   (g) 客户进程回送DONT ECHO。然后继续由客户进程负责回显。

一旦登录完成，客户进程将把数据以整行的方式发送给服务器进程。这就是行方式选项的目的。行方式大大地减少了客户进程和服务器进程之间的数据交互数量，而且对于用户的

击键（也就是回显和编辑）提供更快的响应。图 26-14显示的是当我们输入命令时，在行方式连接下分组交换的情况。

```
Vangogh %date
```

（去掉了业务种类信息和窗口通告信息）。

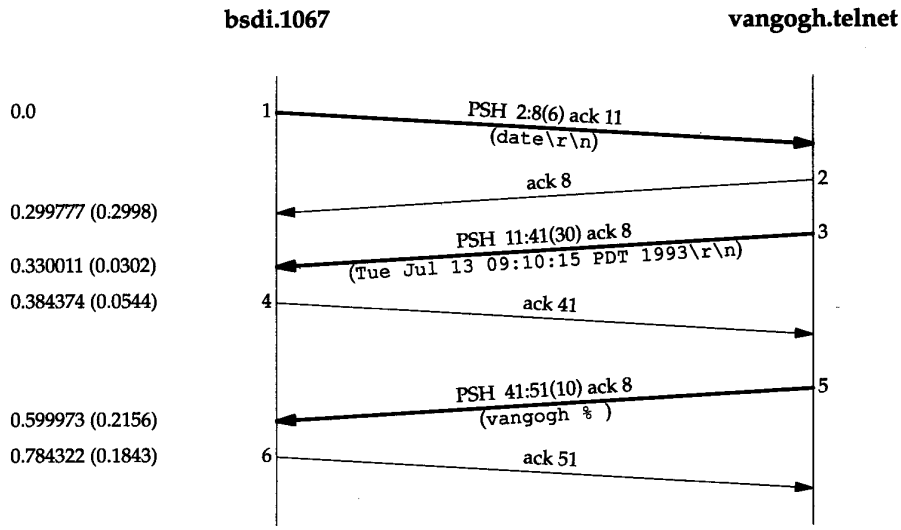


图26-14 Telnet行方式下客户进程向服务器进程发送命令的情况

把它和在Rlogin中输入同样命令（图 19-2）时的情况进行一下比较。我们看到在 Telnet行方式下只需要2个报文段（一个包含数据，另一个用于 ACK，连同IP和TCP首部共86字节），而在Rlogin中要发送15个报文段（5个有键入的数据， 5个有回显的数据， 5个是ACK，共611字节）。可见节省的数据量是非常可观的。

如果在服务器端运行一个需要进入单个字符方式的应用程序（例如 vi编辑器）会怎么样呢？实际上将发生如下的一些交互：

1) 当服务器的应用程序启动了，并改变其伪终端方式时， Telnet服务器进程被通告需要进入单个字符方式。然后服务器发送 WILL ECHO命令和行方式子选项，以告知客户不要再以行方式工作，转而进入单个字符方式。

2) 客户响应以DO ECHO，并确认行方式子选项。

3) 应用程序在服务器上运行。我们键入的每个字符将发送到服务器（当然要强制使用Nagle算法），此时服务器将处理必要的回显工作。

4) 当应用程序终止时，就恢复其伪终端方式，并通告 Telnet服务器。服务器将向客户发送WONT ECHO命令，同时发送行方式子选项，告诉客户恢复进入行方式。

5) 客户响应DONT ECHO，确认进入行方式。

上述情况同我们键入口令之间的区别表明：回显功能和单个字符方式与一次一行方式没有依赖关系。当我们键入口令时，回显功能必须失效，但一次一行方式有效。对于一个全屏应用来讲，例如编辑器，回显必须失效而单个字符方式必须有效。

图26-15概括了Rlogin和Telnet不同方式之间的差异。

应用程序	客户进程发送		客户进程回显？	例　子
	一次一字符	一次一行		
Rlogin	•		否	
Telnet	•		否	
Telnet，行方式		•	是	正常命令
Telnet，行方式		•	否	键入我们的口令
Telnet，行方式	•		否	vi编辑器

图26-15　Rlogin和不同方式的Telnet之间的比较

26.5.3　一次一行方式（准行方式）

从图26-11可以看出，如果客户不支持行方式，那么较新的服务器支持行方式选项，它也将转入准行方式(Kludge line mode)。我们同时指出所有的客户进程和服务器进程都支持准行方式，但它不是默认方式，必须由客户进程或用户特地激活它。让我们来看看如何用 Telnet选项激活准行方式。

首先介绍当客户进程不支持行方式时， BSD/386服务器进程如何协商进入该方式。

1) 当客户进程不同意服务器进程激活行方式的请求时，服务器进程发送　DO TIMING MARK选项。RFC 860 [Postel 和Reynolds 1983f]定义了这个Telnet选项。它的作用是让收发双方同步，关于这个问题将在本节的后面讲到用户键入中断键时讨论。该选项只是用来判断客户进程是否支持准行方式。

2) 客户响应WILL TIMING MARK，表明支持准行方式。

3) 服务器发送WONT SUPPRESS GO AHEAD和WONT ECHO选项，告诉客户它希望禁止这两个选项。我们在前面已经强调：单个字符方式下是假定　SUPPRESS GO AHEAD和ECHO选项同时有效的，所以禁止两个选项就进入了准行方式。

4) 客户响应DONT SUPPRESS GO AHEAD和DONT ECHO命令。

5) 服务器发送`login:`提示符，然后用户键入用户名。用户名是以整行的方式发送给服务器，回显由客户进程在本地处理。

6) 服务器发送`Password:`提示符和WILL ECHO命令。这将使客户进程的回显失效，因为此时客户进程认为服务器进程将处理回显工作，所以用户键入的口令就不回显到屏幕上。客户响应DO ECHO命令。

7) 我们键入口令。客户以整行方式发送到服务器。

8) 服务器发送WONT ECHO命令，使得客户重新激活回显功能，客户响应 DONT ECHO。

从此以后的普通命令处理过程就和行方式相似了。客户进程负责所有的编辑和回显，并以整行的方式发送给服务器进程。

在图26-11中，我们已经强调：所有标注为“ char ”的记录都支持准行方式，只不过默认是单个字符方式罢了。如果要客户进入行方式，我们就能很容易看到选项协商的过程：

```
svr4 %                                          客户是sun，服务器是svr 4
                                                键入Control]以和Telnet客户进程通信(无回显)
telnet> status                                  检验现在是否在一次一字符方式下
Connected to svr4.tuc.noao.edu
Operating in character-at-a-time mode.
Escape character is '^]'.
```

```
telnet> toggle options                    注意选项协商过程
Will show option processing.

telnet> mode line                         切换到准行方式
SENT dont SUPPRESS GO AHEAD               客户发送这两个选项
SENT dont ECHO
RCVD wont SUPPRESS GO AHEAD               服务器把WONT做为上述两个选项协商的响应
RCVD wont ECHO
```

这将使Telnet会话进入准行方式，此时SUPPRESS GO AHEAD和ECHO选项都是失效的。

如果在服务器端运行如vi编辑器这样的应用程序，同样会有行方式下遇到的问题。当要运行这样的应用程序时，服务器进程必须告诉客户进程从准行方式切换到单字符方式。当应用程序结束时，必须告诉客户进程返回到准行方式。下面是这个过程需要用到的技术要点。

1) 当应用程序改变其伪终端方式并通知服务器进程时，服务器进程将进入单字符方式。服务器进程向客户进程发送WILL SUPPRESS GO AHEAD和WILL ECHO，这将使客户进程进入单字符方式。

2) 客户进程回送DO SUPPRESS GO AHEAD和WILL ECHO。

3) 应用程序开始在服务器端运行。

4) 当应用程序结束并改变其伪终端方式时，服务器进程发送 WONT SUPPRESS GO AHEAD和WONT ECHO命令，使得客户进程返回准行方式。

5) 客户进程回送DONT SUPPRESS GO AHEAD和DONT ECHO命令，告诉服务器进程它已经回到了准行方式。

图26-16概括了单个字符方式及准行方式中不同的SUPPRESS GO AHEAD和ECHO选项设置。

方 式	SUPPRESS GO AHEAD	ECHO	举 例
一次一字符	√	√	准行方式下的vi编辑器
准行方式	×	×	正常命令
准行方式	×	√	键入我们的口令

图26-16 准行方式下Telnet选项的设置

26.5.4 行方式：客户中断键

看一下当用户键入中断键时 Telnet将发生什么情况。假定在客户主机 bsdi和服务器cangogh.cs.berkeley.edu之间建立了一个Telnet会话。图26-17显示了当用户键入中断键后的时间系列（去掉了窗口通告和服务类型）。

报文段1中显示的是中断键（通常是Control_C或DELETE）已经转换为Telnet的IP（中断进程）命令：<IAC, IP>。下面的3个字节：<IAC, DO, TM>，组成了Telnet的DO TIMING MARK选项。这个标志由客户进程发送，必须使用WILL或WONT响应。所有在响应前收到的数据都要丢弃（除非是Telnet命令）。这是服务器进程和客户机端的同步过程。报文段1没有采用TCP紧急方式。

Host Requirements RFC叙述了IP命令不能使用Telnet的同步信号来发送。如果可以的话，那么<IAC, IP>的后面将跟随<IAC, DM>，同时紧急指针指向DM字节。大多数的Unix Telnet 客户有一个选项来使用同步信号发送IP命令，但是这个选项默认是不用的（正如我们这里看到的）。

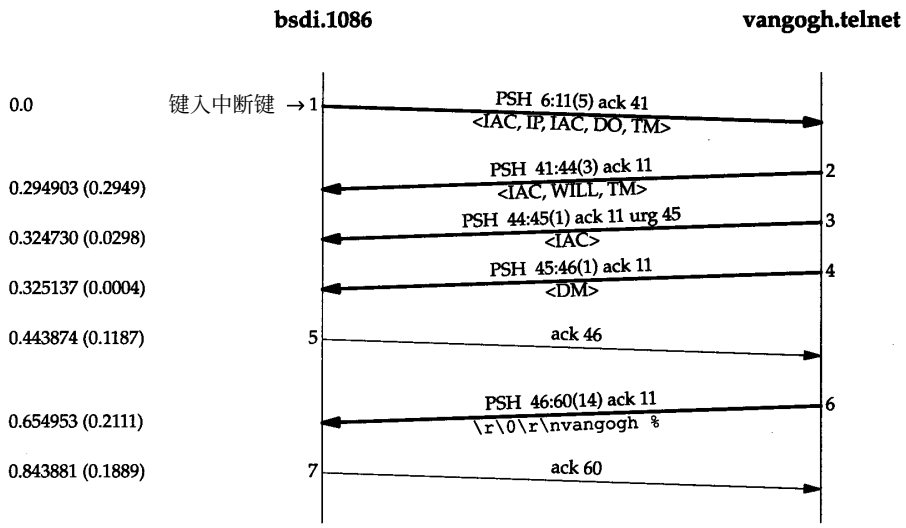


图26-17 行方式下键入中断键后的情况

报文段2是服务器进程对 DO TIMING MARK选项的响应。紧随其后的是报文段 3和4中Telnet的同步信号：<IAC, DM>。报文段3中的紧急指针指向将在报文段4中发送的DM字节。

如果服务器进程到客户进程的窗口已满，那么客户进程发送了如报文段 1中的IP命令后就丢弃收到的所有数据。即使服务器进程被 TCP流量控制所终止而不能发送如报文段 2、3和4中的数据，紧急指针仍然可以发送。这和图 26-7中的Rlogin类似。

为什么同步信号要分为两个数据段发送（3和4）？原因就是我们在20.8节中详细讨论TCP紧急指针时提到的情况。有关主机需求的 RFC中提到紧急指针应指向紧急数据的最后一个字节，而很多衍生于伯克利的系统中，紧急指针指向紧急数据的倒数第 2个字节（回忆一下在图26-6中，紧急指针指向命令字节的前一个字节）。Telnet服务器进程故意把同步信号的第 1个字节作为紧急数据，它知道紧急指针将指向下 1个字节（即DM字节），而IAC字节和紧急指针必须立即发送，在下一步才发送DM字节。

最后一个报文段6发送的是数据，它是服务器进程发生的提示符。

26.6 小结

本章我们介绍了 Rlogin和Telnet操作。两者都提供了从客户进程远程登录到服务器进程，是我们能够在服务器端运行程序的方法。

这两个应用是不同的。 Rlogin假定连接的双方都是 Unix系统，所以只提供一个选项，它是1个简单的协议。Telnet则不同，它用于在不同类型的主机之间建立连接。

为了支持这种多机环境， Telnet提供客户进程和服务器进程的选项协商机制。如果连接的双方都支持这些选项，则可以增强一些功能。对于比较简单的客户进程和服务器进程，它可以提供Telnet的基本功能，而当双方都支持某些选项时，它又可以充分利用双方的新特性。

我们介绍了Telnet的选项协商机制，也介绍了 3种数据传输的方式：单字符方式、准行方式和实行方式。现在的趋势是只要有可能，就尽量工作在准行方式下。这样可以减少网络上的数据量，同时为交互用户提供更好的行编辑和回显的响应。

图26-18概括并比较了Rlogin和Telnet的不同特性。

特　征	Rlogin	Telnet
运输协议	一个TCP连接。使用紧急方式	一个TCP连接。使用紧急方式。
分组方式	总是一次一字符，远程回显	通常的默认是一次一字符，远程回显。带客户回显的准行方式也支持带回显的实行模式。当服务器上的应用进程请求时，总是一次一字符的方式
流量控制	通常由客户完成，可以被服务器禁止	通常由服务器完成，选项允许客户来完成
终端类型	总是提供	选项，通常被支持
终端速率	总是提供	选项
窗口大小	大多数服务器支持此选项	选项
环境变量	不支持	选项
自动登录	默认。提示用户键入口令，口令以明文发送。较新的版本支持Kerberos方式的登录	默认是键入登录名和口令。口令以明文发送。较新的版本支持鉴别选项

图26-18　Rlogin和Telnet的不同特性

Rlogin服务器和Telnet服务器通常都将设置TCP的保活选项以检测客户主机是否崩溃（如果服务器的TCP实现支持，见第23章）。这两种应用都采用了TCP紧急方式，以便即使从服务器到客户的数据传输被流量控制所终止，服务器仍然可以向客户发送命令。

习题

26.1　在图26-5中，标出所有延迟的ACK。

26.2　在图26-7中，为什么要发送报文段12？

26.3　我们说过Rlogin客户进程必须使用保留端口号（见　1.9节）（通常Rlogin客户使用512~1023之间的保留端口）。这会给主机带来什么限制？有没有解决的办法？

26.4　阅读RFC 1097，它描述了Telnet的阈下报文(subliminal-message)选项。

第27章 FTP：文件传送协议

27.1 引言

FTP是另一个常见的应用程序。它是用于文件传输的 Internet标准。我们必须分清文件传送（file transfer）和文件存取 (file access) 之间的区别，前者是 FTP提供的，后者是如 NFS（Sun的网络文件系统，第 29章）等应用系统提供的。由 FTP提供的文件传送是将一个完整的文件从一个系统复制到另一个系统中。要使用 FTP，就需要有登录服务器的注册帐号，或者通过允许匿名FTP的服务器来使用（本章我们将给出这样的一个例子）。

与Telnet类似，FTP最早的设计是用于两台不同的主机，这两个主机可能运行在不同的操作系统下、使用不同的文件结构、并可能使用不同字符集。但不同的是，Telnet获得异构性是强制两端都采用同一个标准：使用7比特ASCII码的NVT。而FTP是采用另一种方法来处理不同系统间的差异。FTP支持有限数量的文件类型（ASCII，二进制，等等）和文件结构（面向字节流或记录）。

参考文献959 [Postel 和 Reynolds 1985] 是FTP的正式规范。该文献叙述了近年来文件传输的历史演变。

27.2 FTP协议

FTP与我们已描述的另一种应用不同，它采用两个 TCP连接来传输一个文件。

1) 控制连接以通常的客户服务器方式建立。服务器以被动方式打开众所周知的用于FTP的端口（21），等待客户的连接。客户则以主动方式打开 TCP端口21，来建立连接。控制连接始终等待客户与服务器之间的通信。该连接将命令从客户传给服务器，并传回服务器的应答。

   由于命令通常是由用户键入的，所以IP对控制连接的服务类型就是“最大限度地减小迟延”。

2) 每当一个文件在客户与服务器之间传输时，就创建一个数据连接。（其他时间也可以创建，后面我们将说到）。

   由于该连接用于传输目的，所以IP对数据连接的服务特点就是“最大限度提高吞吐量”。

图27-1描述了客户与服务器以及它们之间的连接情况

从图中可以看出，交互式用户通常不处理在控制连接中转换的命令和应答。这些细节均由两个协议解释器来完成。标有“用户接口”的方框功能是按用户所需提供各种交互界面（全屏幕菜单选择，逐行输入命令，等等），并把它们转换成在控制连接上发送的 FTP命令。类似地，从控制连接上传回的服务器应答也被转换成用户所需的交互格式。

从图中还可以看出，正是这两个协议解释器根据需要激活文件传送功能。

27.2.1 数据表示

FTP协议规范提供了控制文件传送与存储的多种选择。在以下四个方面中每一个方面都必须作出一个选择。

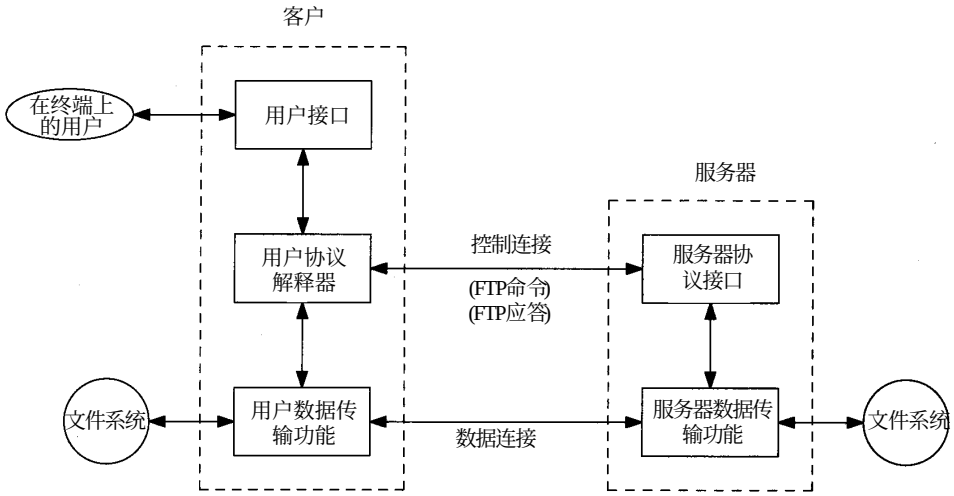


图27-1 文件传输中的处理过程

1. 文件类型

(a) ASCII码文件类型 （默认选择）文本文件以NVT ASCII码形式在数据连接中传输。这要求发方将本地文本文件转换成NVT ASCII码形式，而收方则将NVT ASCII码再还原成本地文本文件。其中，用NVT ASCII码传输的每行都带有一个回车，而后是一个换行。这意味着收方必须扫描每个字节，查找CR、LF对（我们在第15.2节见过的关于TFIP的ASCII码文件传输情况与此相同）。

(b) EBCDIC文件类型 该文本文件传输方式要求两端都是EBCDIC系统。

(c) 图像文件类型（也称为二进制文件类型） 数据发送呈现为一个连续的比特流。通常用于传输二进制文件。

(d) 本地文件类型 该方式在具有不同字节大小的主机间传输二进制文件。每一字节的比特数由发方规定。对使用8 bit字节的系统来说，本地文件以8 bit字节传输就等同于图像文件传输。

2. 格式控制

该选项只对ASCII和EBCDIC文件类型有效。

(a) 非打印 （默认选择）文件中不含有垂直格式信息。

(b) 远程登录格式控制 文件含有向打印机解释的远程登录垂直格式控制。

(c) Fortran 回车控制 每行首字符是Fortran格式控制符。

3. 结构

（a） 文件结构 （默认选择）文件被认为是一个连续的字节流。不存在内部的文件结构。

（b） 记录结构 该结构只用于文本文件（ASCII或EBCDIC）。

（c） 页结构 每页都带有页号发送，以便收方能随机地存储各页。该结构由 TOPS-20操作系统提供（主机需求RFC不提倡采用该结构）。

4. 传输方式

它规定文件在数据连接中如何传输。

（a） 流方式 （默认选择）文件以字节流的形式传输。对于文件结构，发方在文件尾提示关闭数据连接。对于记录结构，有专用的两字节序列码标志记录结束和文件结束。

（b） 块方式 文件以一系列块来传输，每块前面都带有一个或多个首部字节。

（c） 压缩方式 一个简单的全长编码压缩方法，压缩连续出现的相同字节。在文本文件

中常用来压缩空白串，在二进制文件中常用来压缩 0字节（这种方式很少使用，也不受支持。现在有一些更好的文件压缩方法来支持 FTP）。

如果算一下所有这些选择的排列组合数，那么对传输和存储一个文件来说就有 72种不同的方式。幸运的是，其中很多选择不是废弃了，就是不为多数实现环境所支持，所以我们可以忽略掉它们。

通常由Unix实现的FTP 客户和服务器把我们的选择限制如下：

- 类型：ASCII或图像。
- 格式控制：只允许非打印。
- 结构：只允许文件结构。
- 传输方式：只允许流方式。

这就限制我们只能取一、两种方式： ASCII或图像（二进制）。

该实现满足主机需求 RFC的最小需求（该 RFC也要求能支持记录结构，但只有操作系统支持它才行，而Unix不行）。

很多非Unix的实现提供了处理它们自己文件格式的 FTP功能。主机需求RFC指出“ FTP协议有很多特征，虽然其中一些通常不实现，但对 FTP中的每一个特征来说，都存在着至少一种实现 ”。

27.2.2 FTP命令

命令和应答在客户和服务器的控制连接上以 NVT ASCII码形式传送。这就要求在每行结尾都要返回CR、LF对（也就是每个命令或每个应答）。

从客户发向服务器的 Telnet命令（以IAC打头）只有中断进程（ <IAC,IP>）和Telnet的同步信号（紧急方式下 <IAC,DM>）。我们将看到这两条 Telnet命令被用来中止正在进行的文件传输，或在传输过程中查询服务器。另外，如果服务器接受了客户端的一个带选项的 Telnet命令（WILL，WONT，DO或DONT），它将以DONT 或WONT响应。

这些命令都是3或4个字节的大写ASCII字符，其中一些带选项参数。从客户向服务器发送的FTP命令超过30种。图27-2给出了一些常用命令，其中大部分将在本章再次遇到。

命　令	说　明
ABOR	放弃先前的FTP命令和数据传输
LIST *filelist*	列表显示文件或目录
PASS *password*	服务器上的口令
PORT *n1,n2,n3,n4,n5,n6*	客户端IP地址（*n1.n2.n3.n4*）和端口（*n5*×256+*n6*）
QUIT	从服务器注销
RETR *filename*	检索（取）一个文件
STOR *filename*	存储（放）一个文件
SYST	服务器返回系统类型
TYPE *type*	说明文件类型：A表示ASCII码，I表示图像
USER *username*	服务器上用户名

图27-2　常用的FTP命令

下节我们将通过一些例子看到，在用户交互类型和控制连接上传送的 FTP命令之间有时是一对一的。但也有些操作下，一个用户命令产生控制连接上多个 FTP命令。

27.2.3 FTP应答

应答都是ASCII码形式的3位数字，并跟有报文选项。其原因是软件系统需要根据数字代码来决定如何应答，而选项串是面向人工处理的。由于客户通常都要输出数字应答和报文串，一个可交互的用户可以通过阅读报文串（而不必记忆所有数字回答代码的含义）来确定应答的含义。

应答3位码中每一位数字都有不同的含义（我们将在第 28章看到简单邮件传送输协议，SMTP，使用相同的命令和应答约定）。

图27-3给出了应答代码第1位和第2位的含义。

应答	说　　明
1yz	肯定预备应答。它仅仅是在发送另一个命令前期待另一个应答时启动
2yz	肯定完成应答。一个新命令可以发送
3yz	肯定中介应答。该命令已被接受，但另一个命令必须被发送
4yz	暂态否定完成应答。请求的动作没有发生，但差错状态是暂时的，所以命令可以过后再发
5yz	永久性否定完成应答。命令不被接受，并且不再重试
x0z	语法错误
x1z	信息
x2z	连接。应答指控制或数据连接
x3z	鉴别和记帐。应答用于注册或记帐命令
x4z	未指明
x5z	文件系统状态

图27-3 应答代码3位数字中第1位和第2位的含义

第3位数字给出差错报文的附加含义。例如，这里是一些典型的应答，都带有一个可能的报文串。

- 125 数据连接已经打开；传输开始。
- 200 就绪命令。
- 214 帮助报文（面向用户）。
- 331 用户名就绪，要求输入口令。
- 425 不能打开数据连接。
- 452 错写文件。
- 500 语法错误（未认可的命令）。
- 501 语法错误（无效参数）。
- 502 未实现的MODE(方式命令)类型。

通常每个FTP命令都产生一行回答。例如， QUIT命令可以产生如下应答：

```
221 Goodbye.
```

如果需要产生一条多行应答，第1行在3位数字应答代码之后包含一个连字号，而不是空格，最后一行包含相同的3位数字应答代码，后跟一个空格符。例如，HELP命令可以产生如下应答：

```
214- The following commands are recognized (* =>'s unimplemented).
   USER    PORT    STOR    MSAM*   RNTO    NLST    MKD     CDUP
   PASS    PASV    APPE    MRSQ*   ABOR    SITE    XMKD    XCUP
   ACCT*   TYPE    MLFL*   MRCP*   DELE    SYST    RMD     STOU
   SMNT*   STRU    MAIL*   ALLO    CWD     STAT    XRMD    SIZE
```

```
   REIN*    MODE     MSND*    REST     XCWD     HELP     PWD      MDTM
   QUIT     RETR     MSOM*    RNFR     LIST     NOOP     XPWD
214 Direct comments to ftp-bugs@bsdi.tuc.noao.edu.
```

27.2.4 连接管理

数据连接有以下三大用途：

1) 从客户向服务器发送一个文件。

2) 从服务器向客户发送一个文件。

3) 从服务器向客户发送文件或目录列表。

FTP服务器把文件列表从数据连接上发回，而不是控制连接上的多行应答。这就避免了行的有限性对目录大小的限制，而且更易于客户将目录列表以文件形式保存，而不是把列表显示在终端上。

我们已说过，控制连接一直保持到客户-服务器连接的全过程，但数据连接可以根据需要随时来，随时走。那么需要怎样为数据连接选端口号，以及谁来负责主动打开和被动打开？

首先，我们前面说过通用传输方式（Unix环境下唯一的传输方式）是流方式，并且文件结尾是以关闭数据连接为标志。这意味着对每一个文件传输或目录列表来说都要建立一个全新的数据连接。其一般过程如下：

1) 正由于是客户发出命令要求建立数据连接，所以数据连接是在客户的控制下建立的。

2) 客户通常在客户端主机上为所在数据连接端选择一个临时端口号。客户从该端口发布一个被动的打开。

3) 客户使用PORT命令从控制连接上把端口号发向服务器。

4) 服务器在控制连接上接收端口号，并向客户端主机上的端口发布一个主动的打开。服务器的数据连接端一直使用端口20。

图27-4给出了第3步执行时的连接状态。假设客户用于控制连接的临时端口是1173，客户用于数据连接的临时端口是1174。客户发出的命令是PORT命令，其参数是6个ASCII中的十进制数字，它们之间由逗点隔开。前面4个数字指明客户上的IP地址，服务器将向它发出主动打开（本例中是140.252.13.34），而后两位指明16 bit端口地址。由于16 bit端口地址是从这两个数字中得来，所以其值在本例中就是4×256+150 = 1174。

图27-5给出了服务器向客户所在数据连接端发布主动打开时的连接状态。服务器的端点是端口20。

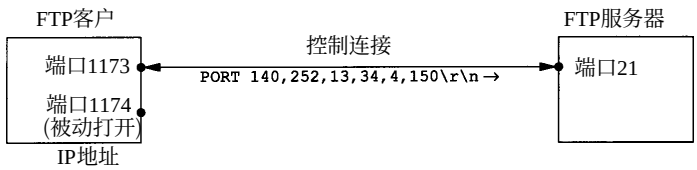


图27-4 在FTP控制连接上通过的PORT命令

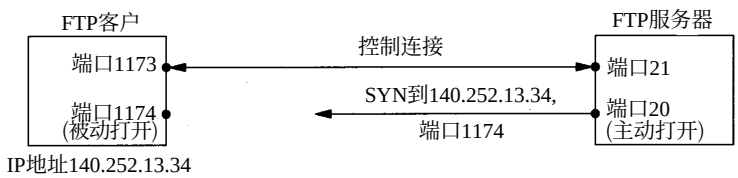


图27-5 主动打开数据连接的FTP服务器

服务器总是执行数据连接的主动打开。通常服务器也执行数据连接的主动关闭，除非当客户向服务器发送流形式的文件时，需要客户来关闭连接（它给服务器一个文件结束的通知）。

客户也有可能不发出PORT命令，而由服务器向正被客户使用的同一个端口号发出主动打开，来结束控制连接。这是可行的，因为服务器面向这两个连接的端口号是不同的：一个是20，另一个是21。不过，下节我们将看到为什么现有实现通常不这样做。

27.3 FTP的例子

现在看一些使用FTP的例子：它对数据连接的管理，采用NVT ASCII码的文本文件如何发送，FTP使用Telnet同步信号来中止进行中的文件传输，最后是常用的“匿名FTP”。

27.3.1 连接管理：临时数据端口

先看一下FTP的连接管理，它只在服务器上用简单FTP会话显示一个文件。我们用-d标志（debug）来运行svr4主机上的客户。这告诉它要打印控制连接上变换的命令和应答。所有前面冠以--->的行是从客户上发向服务器的，所有以3位数字开头的行都是服务器的应答。客户的交互提示是ftp>。

```
svr4 %ftp -d bsdi                              -d 选项用作排错输出
Connected to bsdi.                             客户执行控制连接的主动打开
220 bsdi FTP server (Version 5.60) ready       服务器响应就绪
Name (bsdi:rstevens):                          客户提示我们输入
---> USER rstevens                             键入RETURN，客户发送默认信息
331 Password required for rstevens.
Password:                                      键入口令；它不需要回显
---> PASS XXXXXXXX                                  客户以明文发送它
230 User rstevens logged in.
ftp> dir hello.c                               要求列出一个文件的目录
---> PORT 140,252,13,34,4,150                       见图27-4
200 PORT Command successful.
---> LIST hello.c
150 Opening ASCII mode data connection for /bin/ls.
-rw-r--r--   1 rstevens  staff  38 Jul 17 12:47 hello.c
226 Transfer complete.
remote: hello.c                                客户输出
56 bytes received in 0.03 seconds (1.8 Kbytes/s)
ftp> quit                                      我们已完成
---> QUIT
221 Goodbye
```

当FTP客户提示我们注册姓名时，它打印了默认值（我们在客户上的注册名）。当我们敲RETURN键时，默认值被发送出去。

对一个文件列出目录的要求引发一个数据连接的建立和使用。本例体现了我们在图27-4和图27-5中给出的程序。客户要求TCP为其数据连接的终端提供一个临时端口号，并用PORT命令发送这个端口号（1174）给服务器。我们也看到一个交互用户命令（dir）成为两个FTP命令（PORT和LIST）。

图27-6是控制连接上分组交换的时间系列（已除去了控制连接的建立和结束，以及所有窗口大小的通知）。我们关注该图中数据连接在哪儿被打开、使用和过后的关闭。

图27-7是数据连接的时间系列。图中的起始时间与图 27-6中的相同。已除去了所有窗口大小通知，但留下服务类型字段，以说明数据连接使用另一个服务类型（最大吞吐量），而不同于控制连接（最小时延）（服务类型(TOS)值在图3-2中）。

在时间系列上，FTP服务器执行数据连接的主动打开，从端口 20（称为ftp-data）到来自PORT命令的端口号（1174）。本例中还可以看到服务器在哪儿向数据连接上执行写操作，服务器对数据连接执行主动的关闭，这就告诉客户列表已完成。

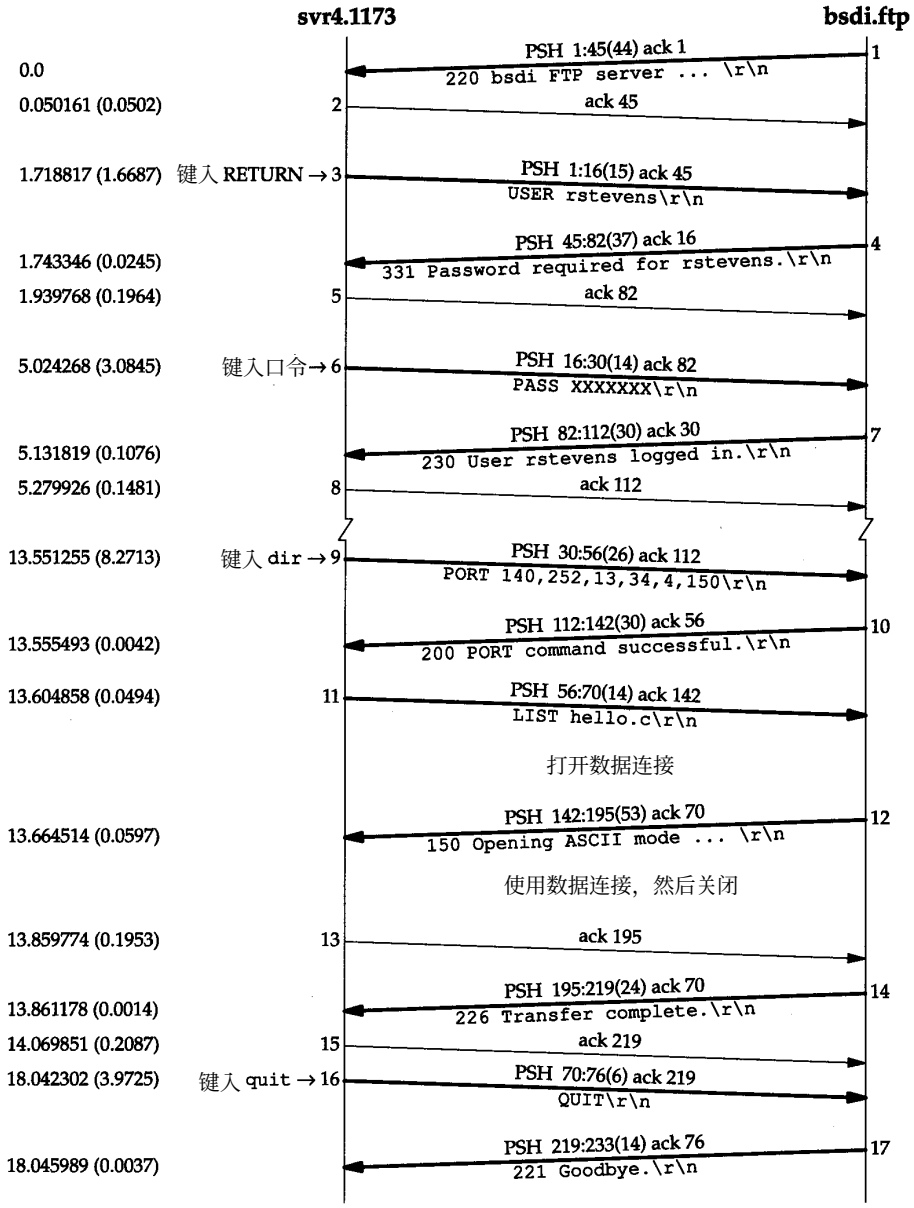


图27-6 FTP控制连接示例

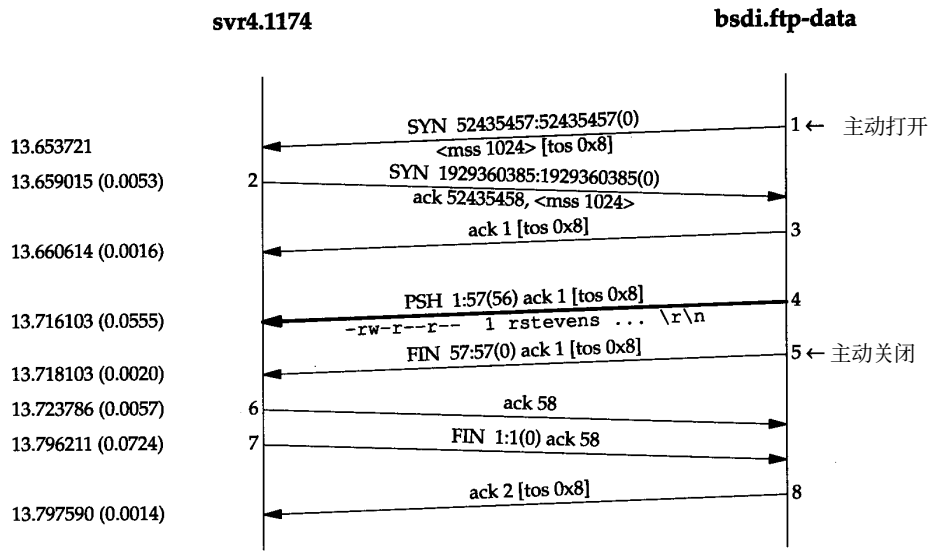


图27-7 FTP数据连接示例

27.3.2 连接管理：默认数据端口

如果客户没有向服务器发出PORT命令，来指明客户数据连接端的端口号，服务器就用与控制连接正在用的相同的端口号给数据连接。这会给使用流方式（ Unix FTP客户和服务器一直使用）的客户带来一些问题。正如下面所示：

> Host Requirements RFC建议使用流方式的FTP客户在每次使用数据连接前发一个PORT命令来启用一个非默认的端口号。

回到先前的例子（图27-6），如果我们要求在列出第1个目录后几秒钟再列出另一个目录，那该怎么办？客户将要求其内核选择另一个临时端口号（可能是 1175），下一个数据连接将建立在svr4端口1175和bsdi端口20之间。但在图27-7中服务器执行数据连接的主动打开，我们在18.6节说明了服务器将不把端口 20分配给新的数据连接，这是因为本地端口号已被更早的连接使用，而且还处于2MSL等待状态。

服务器通过指明我们在18.6节中提到的SO_REUSEADDR选项，来解决这个问题。这让它把端口20分配给新连接，而新连接将从处于 2MSL等待状态的端口（1174）处得到一个不一样的外部端口号（1175），这样一切都解决了。

如果客户不发送PORT命令，而在客户上指明一个临时端口号，那么情况将改变。我们可以通过执行用户命令sendport给FTP来使之发生。Unix FTP客户用这个命令在每个数据连接使用之前关闭向服务器发送PORT命令。

图27-8给出了用于两个连续LIST命令的数据连接时间系列。控制连接起自主机 svr4上的端口1176，所以在没有PORT命令的情况下，客户和服务器给数据连接使用相同的端口号（除去了窗口通知和服务类型值）。

事件序列如下：

1) 控制连接是建立在客户端口1176到服务器端口21上的（这里我们不展示）。

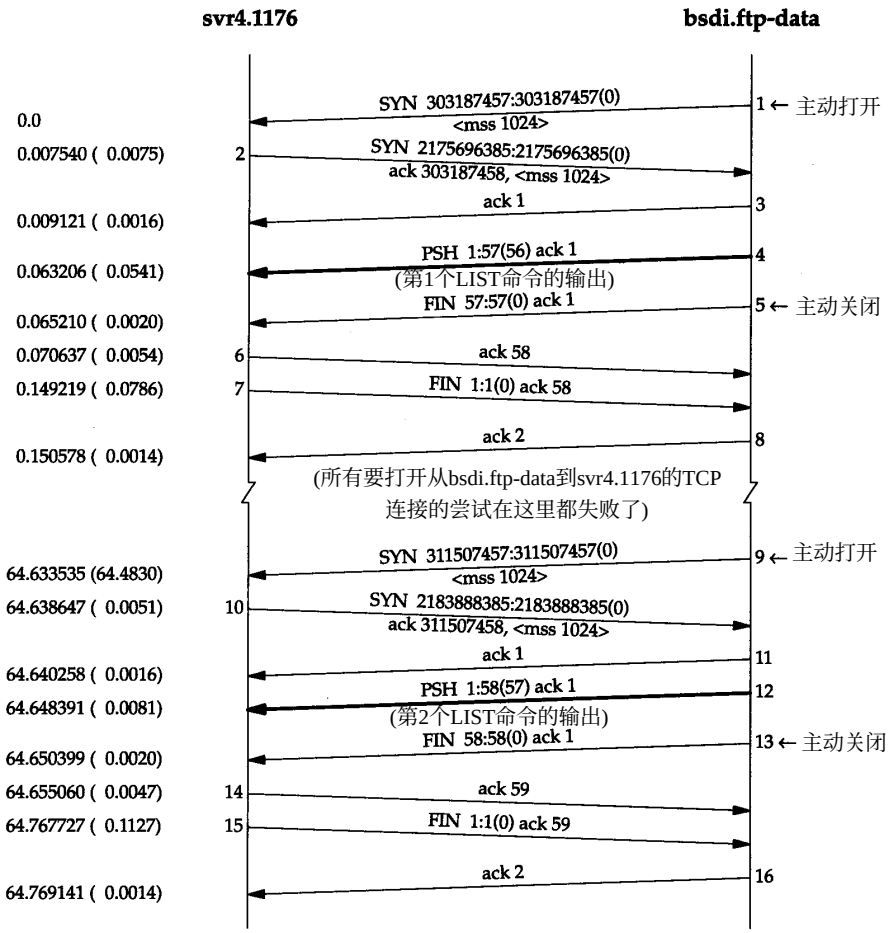


图27-8 两个连续LIST命令的数据连接

2) 当客户为端口1176上的数据连接做被动打开时，由于该端口已被客户上的控制连接使用，所以必须确定SO_REUSEADDR选项。

3) 服务器给端口20到端口1176的数据连接（报文段1）做主动打开。即便端口1176已在客户上被使用，客户仍会接受它（报文段2），这是因为下面这一对插口是不同的：

```
<svr4, 1176, bsdi, 21>
<svr4, 1176, bsdi, 20>
```

（在bsdi上的端口号是不同的）。TCP通过查看源IP地址、源端口号、目的IP地址、目的端口号分用各呼入报文段，只要这4个元素中的一个不同，就行。

4) 服务器对数据连接（报文段 5）做主动的关闭，即把这对插口置入服务器上的一个2MSL等待。

```
<svr4, 1176, bsdi, 20>
```

5) 客户在控制连接上发送另一个 LIST命令（这里我们不展示）。在此之前，客户在端口1176上为其数据连接端做一个被动打开。客户必须再一次指明 SO_REUSEADDR，这是因为端口号1176已在使用。

6) 服务器给从端口20到端口1176的数据连接发出一个主动打开。在此之前，服务器必须指明SO_REUSEADDR，这是因为本地端口（20）与处于2MSL等待状态的连接是相关联的，但从18.6节所示可知，该连接将不成功。其原由是这个连接用插口对 (socket pair)与步骤4中的仍处于2MSL等待状态的插口对相同。TCP规定禁止服务器发送同步信息（SYN）。这样就没办法让服务器跨过插口对的2MSL等待状态来重用相同的插口对。

在这一步伯克利软件分发（BSD）服务器每隔5秒就重试一次连接请求，直到满18次，总共90秒。我们看到报文段9将在大约1分钟后成功（我们在第18章提到过，SVR4使用一个30秒的MSL，以两个MSL来达到持续1分钟的等待）。我们没看到在这个时间系列上的这些失败有任何同步（SYN）信息，这是因为主动打开失败，服务器的TCP不再发送一个SYN。

Host Requirements RFC建议使用PORT命令的原因是在两个相继使用数据连接之间避免出现这个2MSL。通过不停地改变某一端的端口号，我们所说的这个问题就不会出现。

27.3.3 文本文件传输：NVT ASCII表示还是图像表示

让我们查证一下默认的文本文件传输使用 NVT ASCII码。这次不指定 -d标志，所以不看客户命令，但注意到客户还将打印服务器的响应：

```
sun % ftp bsdi
Connected to bsdi.
220 bsdi FTP server (Version 5.60) ready.
Name (bsdi:retevens);                                    键入RETURN
331 Password required for rstevens.
Passord :                                                键入口令
230 User rstevens logged in.
ftp> get hello.c                                         取一个文件
200 PORT command successful.
150 Opening ASCII mode data connection for hello.c (38 bytes).
226 Transfer complete.                                   服务器说明文件含有38字节
local: hello.c remote: hello.c                           由客户输出
42 bytes received in 0.0037 seconds (11 Kbytes/s)        字节传过数据连接
ftp> quit
221 Goodbye.
Sun % ls -l hello.c
-rw-rw-r--   1 rstevens     38 Jul 18   08:48 hello.c   但文件还含有38字节
sun % wc -l hello.c                                      在文件中记行数
    4 hello.c
```

因为文件有4行，所以从数据连接上传输42个字节。Unix下的每一新行符（\n）被服务器转换成NVT ASCII码的2字节行结尾序列（\r\n）来传输，然后再由客户转换成原先形式来存储。

新客户试图确定服务器是否是相同类型的系统，一旦相同，就可以用二进制码（图像文件类型）来传输文件，而不是ASCII码。这可以获得两个方面的好处：

1) 发方和收方不必查看每一字节（很大的节约）。

2) 如果主机操作系统使用比2字节的NVT ASCII码序列更少的字节来作行尾，就会传输更少的字节数（很小的节约）。

我们可以看到使用一个BSD/386客户和服务器的最优效果。启动排错（ debug）方式来看

客户FTP命令：

```
bsdi &ftp -d slip                                   指明 -d来看客户命令
Connected to slip.
220 slip FTP server (Version 5.60) ready.
Name (slip:rstevens):                               我们键入RETURN
---> USER rstevns
331 Password required for rstevens.
Password :                                          我们键入自己的口令
---> PASS XXXX
230 User rstevns logged in .
---> SYST                                           这由客户服务器的应答自动发送
215 UNIX Type: L8 Version : BSD-199103
Remote system type is UNIX.                         由客户发出的信息
Using binary mode to transfer files.                由客户发出的信息
ftp> get hello.c                                    取一个文件
---> TYPE I                                         由客户自动发送
200 Type set to I.
---> PORT  140,252,13,66,4,84                       端口号=4×256+84=1108
200 PORT command successful.
---> RETR hello.c
150 Opening BINARY mode data connection for hello.c (38 bytes) .
226 Transefer complete .
38 bytes received in 0.035 seconds (1.1 Kbytes/这时只有38个字节
ftp> quit
---> QUIT
221 Goodbye.
```

注册到服务器后，客户FTP自动发出SYST命令，服务器将用自己的系统类型来响应。如果应答起自字符串“ 215 UNIX Type : L8 ”，并且如果客户在每字节为8 bit的Unix系统上运行，那么二进制方式（图像）将被所有文件传输所使用，除非被用户改变。

当我们取文件hello.c时，客户自动发出命令TYPE I把文件类型定成图像。这样在数据连接上只有38字节被传输。

> Host Requirements RFC指出一个FTP服务器必须支持SYST命令（这曾是RFC 959中的一个选项）。但支持它的使用文本的系统（见封 2）仅仅是BSD/386和AIX 3.2.2。SunOS 4.1.3和Solaris 2.x 用500（不能理解的命令）来应答。SVR4采用极不大众化的应答行为500，并关闭控制连接！

27.3.4 异常中止一个文件的传输：Telnet 同步信号

现在看一下FTP客户是怎样异常中止一个来自服务器的文件传输。异常中止从客户传向服务器的文件很容易 —— 只要客户停止在数据连接上发送数据，并发出 ABOR命令到控制连接上的服务器即可。而异常中止接收就复杂多了，这是因为客户要告知服务器立即停止发送数据。我们前面提到要使用Telnet同步信号，下面的例子就是这样。

我们先发起一个接收，并在它开始后键入中断键。这里是交互会话，其中初始注册被略去：

```
ftp> get a.out                                  取一个大文件
---> TYPE I                                     客户和服务器都是8 bit字节的Unix系统
200 Type set to I.
---> PORT 140,252,13,66,4,99
200 PORT command successful.
---> RETR a.out
150 Opening BINARY mode data connection for a.out (28672 bytes).
^?                                              键入的中断键
receive aborted                                 由客户输出
waiting for remote to finish abort              由客户输出
426 Transfer aborted. Data connection closed.
226 Abort successful
1536 bytes received in 1.7 seconds (0.89 Kbytes/s)
```

在我们键入中断键之后，客户立即告知我们它将发起异常中止，并正在等待服务器完成。服务器发出两个应答：426和226。这两个应答都是由Unix 服务器在收到来自客户的紧急数据和ABOR命令时发出的。

图27-9和图27-10展示了会话时间系列。我们已把控制连接（实线）和数据连接（虚线）合在一起来说明它们之间的关系。

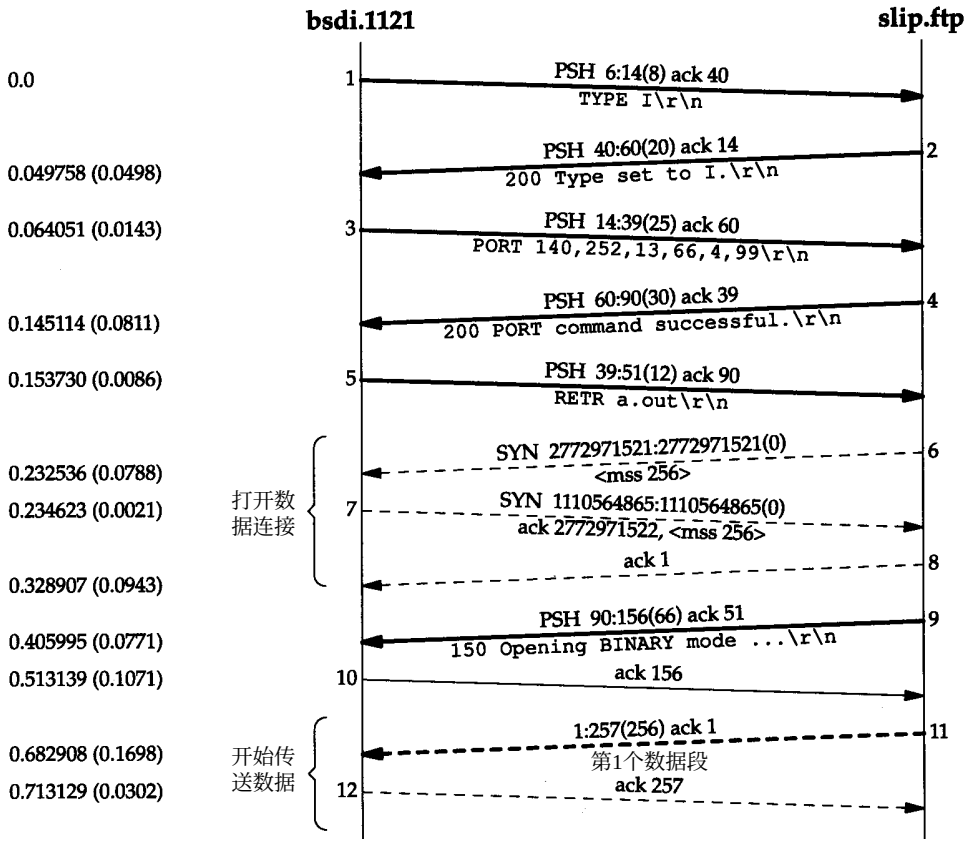


图27-9 异常中止一个文件的传输（前半部）

图27-9的前面12个报文段是我们所期望的。通过控制连接的命令和应答建立起文件传输，数据连接被打开，第1个报文段的数据从服务器发往客户。

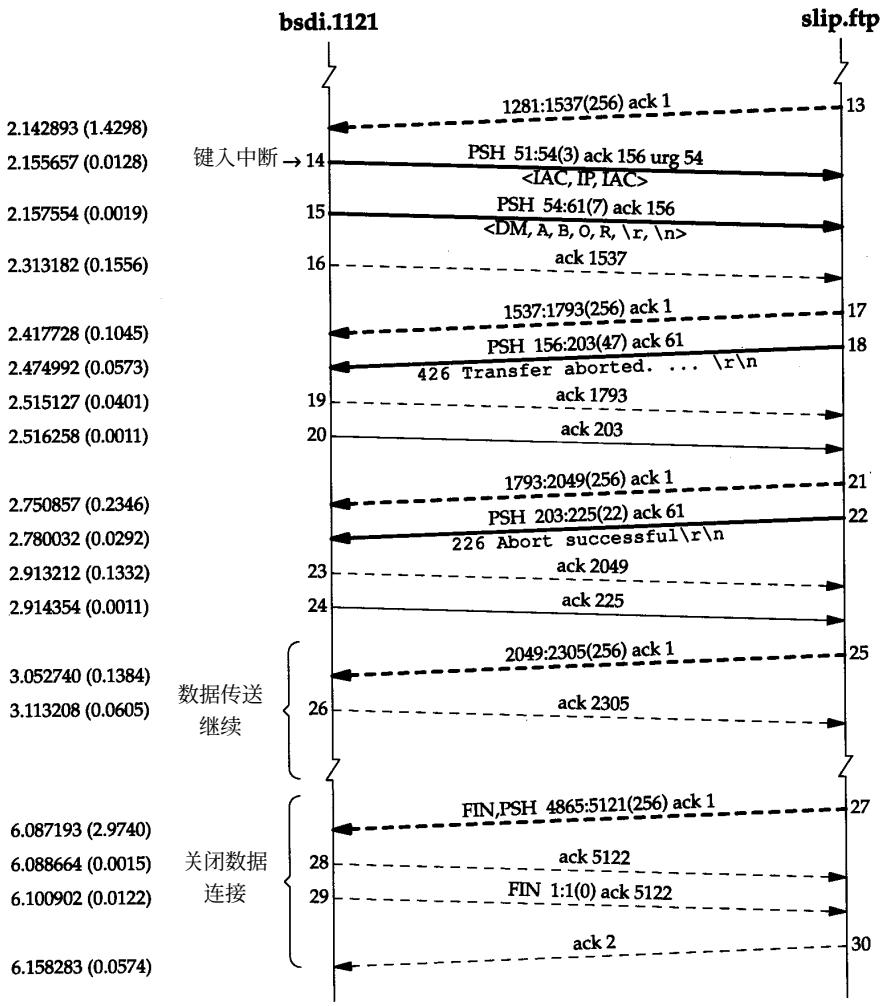


图27-10 异常中止一个文件的传输（后半部）

在图27-10中，报文段13是数据连接上来自服务器的第 6个数据报文段，后跟由我们键入的中断键产生的报文段14。客户发出10个字节来异常中止传输：

```
<IAC,IP,IAC,DM,A,B,O,R,\r,\n>
```

由于20.8节中详细讨论过这个问题，我们看到有两个报文段（ 14和15）涉及到TCP的紧急指针（我们在图26-17中看过对Telnet问题也做相同的处理）。Host Requirements RFC指出紧急指针应指向紧急数据的最后一个字节，而多数伯克利的派生实现使之指向紧急数据最后一个字节后面的第一个字节。了解到紧急指针将（错误地）指向下一个要写的字节（数据标志，DM。在序号为54处），FTP客户进程特意写前3个字节作为紧急数据。首先写下的 3字节紧急数据与紧急指针一起被立即发送，紧接着是后面 7个字节（BSD FTP 服务器不会出现由客户使用的紧急指针的解释问题。当服务器收到控制连接上的紧急数据时，它读下一个 FTP命令，寻找ABOR或STAT，忽略嵌入的Telnet命令）。

注意到尽管服务器指出传输被异常中止（报文段 18，在控制连接上），客户进程还要在数据连接上再接收14个报文段的数据（序列号是 1537~5120）。这些报文段可能在收到异常中止

时，还在服务器上的网络设备驱动器中排队，但客户打印“收到 1536字节”，意思是在发出异常中止后（报文段14和15），略去收到的所有数据报文段。

一旦Telnet用户键入中断键，我们在图 26-17中看到Unix客户在默认情况下不发出中断进程命令作为紧急数据。因为几乎没有机会用流控制来中止从客户进程到服务器进程的数据流，所以我们说这样就行了。FTP的客户进程也通过控制连接发送一个中断进程命令，因为两个连接正在被使用，因此没有机会用流控制来中止控制连接。为什么 FTP发送中断进程命令作为紧急数据而Telnet不呢？答案在于FTP使用两个连接，而 Telnet只使用一个，在某些操作系统上要求一个进程同时监控两个连接的输入是困难的。 FTP假设这些临界的操作系统至少提供紧急数据在控制连接上已到达的通知，而后让服务器从处理数据连接切换到控制连接上来。

27.3.5 匿名FTP

FTP的一种形式很常用，我们下面给出它的例子。它被称为匿名 FTP，当有服务器支持时，允许任何人注册并使用FTP来传输文件。使用这个技术可以提供大量的自由信息。

怎样找出你正在搜寻的站点是一个完全不同的问题。我们将在 30.4节简要介绍。

我们将把匿名FTP用在站点`ftp.uu.net`上（一个常用的匿名FTP站点）来取本书的勘误表文件。要使用匿名FTP，须使用“anonymous”（复习数遍就能正确地拼写）用户名来注册。当提示输入口令时，我们键入自己的电子邮箱地址。

```
sun % ftp ftp.uu.net
Connected to ftp.uu.net
220 ftp.UU.NET FTP server (Version 2.0WU(13) Fri Apr 9 20:44:32 EDT 1993) ready
Name (ftp.uu.net:rstevens): anonymous
331 Guest login ok, send your complete e-mail addraess as password.
Password :                                  键入rstevens@noao.edu；它没有回显
230-
230-                    Welcome to the UUNET archive.
230-    A service of UUNET Technologies Inc, Falls Church, Virginia
230-    For information about UUNET, call +1 703 204 8000, or see the files
230-    in /uunet-info
                                            还有一些问候行
230 Guest login ok, access restrictions apply.
ftp> cd published/books                     换成需要的目录
250 CWD command successful.
ftp> binary                                 我们将传送一个二进制文件
200 Type set to I.
ftp> get stevens.tcpipivl.errata.Z          取文件
200 PORT command successful.
150 Opening BINARY mode data connection for stevens.tcpipivl.errata.Z (150 bytes).
226 Transfer complete.                      (你可能得到一个不同的文件大小)
local: stevens.tcpipivl.errata.Z remote: stevens.tcpipivl.errata.Z
105 bytes received in 4.1 seconds (0.83 Kbytes/s)
ftp> quit
221 Goodbye.
sun % uncompress stevens.tcpipivl.errata.Z
sun % more stevens.tcpipivl.errata
```

不压缩是因为很多现行匿名FTP文件是用Unix compress(1)程序压缩的，这样导致文件带有.Z的扩展名。这些文件必须使用二进制文件类型来传输，而不是ASCII码文件类型。

27.3.6 来自一个未知IP地址的匿名FTP

可以把一些使用匿名FTP的域名系统（DNS）特征和选路特征结合在一起。在14.5节中我们谈到DNS中指针查询现象——取一个IP地址并返回其主机名。不幸的是并非所有系统管理员都能正确地创立涉及指针查询的名服务器。他们经常记得把新主机加入名字到地址匹配的文件中，却忘了把他们加入到地址到名字匹配的文件中。对此，可用traceroute经常看到这种现象，即它打印一个IP地址，而不是主机名。

有些匿名FTP服务器要求客户有一个有效域名。这就允许服务器来记录正在执行传输的主机域名。由于服务器在来自客户IP数据报中收到的关于客户的唯一标识是客户的IP地址，所以服务器能叫DNS来做指针查询，并获得客户的域名。如果负责客户主机的名服务器没有正确地创立，指针查询将失败。

要看清这个错误，我们来做以下诸步骤：

1) 把主机slip（见封2的图）的IP地址换成140.252.13.67。这是给作者子网的一个有效IP地址，但没有涉及到noao.edu域的域名服务器。

2) 把在bsdi上SLIP连接的目的IP地址换成140.252.13.67。

3) 把将数据报引向140.252.13.67的sun上的路由表入口加入路由器bsdi（回忆一下我们在9.2节中关于这个选路表的讨论）。

从Internet上仍然可以访问我们的主机slip，这是因为在10.4节中路由器gateway和netb正好把所有目的是子网140.252.13的所有数据报都发送给路由器sun。路由器sun知道利用我们在上述第3步建立的路由表入口来如何处理这些数据报。我们所创建的是拥有完整Internet连接性的主机，但没有有效的域名。结果，指针查询IP地址140.252.13.67将失败。

现在给一个我们所知的服务器使用匿名FTP，需要一个有效的域名：

```
slip % ftp ftp.uu.net
Connected to ftp.uu.net.

220 ftp.UU.NET FTP server (Version 2.0WU(13) Fri Apr 9 20:44:32 EDT 1993) ready.
Name (ftp.uu.net:rstevens): anonymous

530- Sorry, we're unable to map your IP address 140.252.13.67 to a hostname
530- in the DNS.  This is probably because your nameserver does not have a
530- PTR record for your address in its tables, or because your reverse
530- nameservers are not registered.  We refuse service to hosts whose
530- names we cannot resolve.  If this is simply because your nameserver is
530- hard to reach or slow to respond then try again in a minute or so, and
530- perhaps our nameserver will have your hostname in its cache by then.
530- If not, try reaching us from a host that is in the DNS or have your
530- system administrator fix your servers.
530 User anonymous access denied..

Login failed.
Remote system type is UNIX.
Using binary mode to transfer files.

ftp> quit
221 Goodbye.
```

来自服务器的出错应答是无需加以说明的。

27.4 小结

FTP是文件传输的Internet标准。与多数其他TCP应用不同，它在客户进程和服务器进程之间使用两个TCP连接——一个控制连接，它一直持续到客户进程与服务器进程之间的会话完成为止；另一个按需可以随时创建和撤消的数据连接。

FTP使用的关于数据连接的连接管理让我们更详细地了解TCP连接管理需求。我们看到TCP在不发出PORT命令的客户进程上对2MSL等待状态的作用。

FTP使用NVT ASCII码做跨越控制连接的所有远程登录命令和应答。数据传输的默认方式通常也是NVT ASCII码。我们看到较新的Unix客户进程会自动发送命令来查看服务器是否是8bit字节的Unix主机，并且如果是，那么就使用二进制方式来传输所有文件，那将带来更高的效率。

我们也展示了匿名FTP的一个例子，它是在Internet上分发软件的常用形式。

习题

27.1 图27-8中，如果客户对第2个数据连接做一次主动打开，而不是由服务器来做，那将发生什么变化？

27.2 在本章FTP客户例子中，我们加入诸如由客户输出行的行注释。如果不看源代码，我们如何确定这些不是来自服务器？

```
local: hello.c remote: hello.c
42 bytes received in 0.0037 seconds (11 Kbytes/s)
```

第28章 SMTP: 简单邮件传送协议

28.1 引言

电子邮件（e-mail）无疑是最流行的应用程序。[Caceres et al.1991]说明，所有TCP连接中大约一半是用于简单邮件传送协议SMTP (Simple Mail Transfer Protocol)的（以比特计算为基础，FTP连接传送更多的数据）。[Paxson 1993] 发现，平均每个邮件中包含大约1500字节的数据，但有的邮件中包含兆比特的数据，因为有时电子邮件也用于发送文件。

图28-1显示了一个用TCP/IP交换电子邮件的示意图。

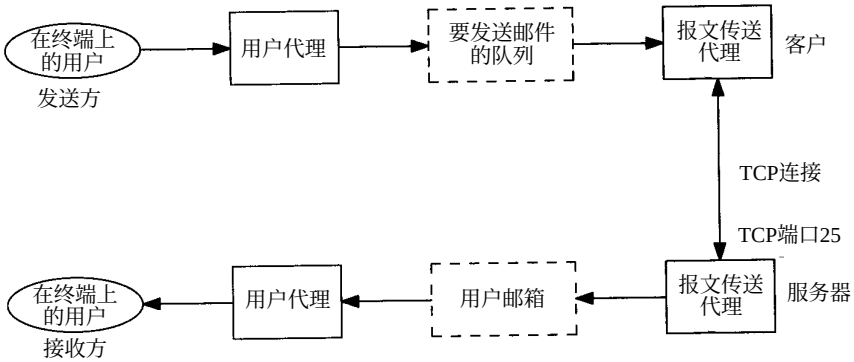


图28-1 Internet电子邮件示意图

用户与用户代理（user agent）打交道，可能会有多个用户代理可供选择。常用的Unix上的用户代理包括MH，Berkeley Mail, Elm和Mush。

用TCP进行的邮件交换是由报文传送代理MTA（Message Transfer Agent）完成的。最普通的Unix系统中的MTA是Sendmail。用户通常不和MTA打交道，由系统管理员负责设置本地的MTA。通常，用户可以选择它们自己的用户代理。

本章研究在两个MTA之间如何用TCP交换邮件。我们不考虑用户代理的运行或实现。

RFC 821 [Postel 1982] 规范了SMTP协议，指定了在一个简单TCP连接上，两个MTA如何进行通信。RFC 822 [Crocker 1982] 指定了在两个MTA之间用RFC 821发送的电子邮件报文的格式。

28.2 SMTP协议

两个MTA之间用NVT ASCII进行通信。客户向服务器发出命令，服务器用数字应答码和可选的人可读字符串进行响应。这与上一章的FTP类似。

客户只能向服务器发送很少的命令：不到12个（相比较而言，FTP超过40个）。我们用简单的例子说明发送邮件的工作过程，并不仔细描述每个命令。

28.2.1 简单例子

我们将发送一个只有一行的简单邮件，并观察SMTP连接。我们用 -v 标志调用用户代理，

它被传送给邮件传送代理（本例中是 Sendmail）。当设置该标志时，该 MTA显示在SMTP连接上发送和接收的内容。以 >>>开始的行是SMTP客户发出的命令，以 3位数字的应答码开始的行是从SMTP服务器来的。以下就是交互会话：

```
sun % mail -v rstevens@noao.edu                    调用我们的代理
To: rstevens@noao.edu                              这是用户代理的输出
Subject: testing                                   然后指示我们键入主题
                                                   用户代理在首部和正文之间加上一行空行
1, 2, 3.                                           这是我们键入的正文
.                                                  我们在一行上输入一个句点，说明完成了
Sending letter ... rstevens@noao.edu...            用户代理上详细的输出

Connecting to mailhost via ether...                以下是MTA(Sendmail)的输出
Trying 140.252.1.54...  connected.
220 noao.edu Sendmail 4.1/SAG-Noao.G89 ready at Mon, 19 Jul 93 12:47:34 MST

>>> HELO sun.tuc.noao.edu.
250 noao.edu Hello sun.tuc.noao.edu., pleased to meet you

>>> MAIL From:<rstevens@sun.tuc.noao.edu>
250 <rstevens@sun.tuc.noao.edu>... Sender ok
>>> RCPT To:<rstevens@noao.edu>
250 <rstevens@noao.edu>... Recipient ok

>>> DATA
354 Enter mail, end with "." on a line by itself

>>> .
250 Mail accepted

>>> QUIT
221 noao.edu delivering mail

rstevens@noao.edu... Sent
sent.                                              这是用户代理的输出
```

只有5个SMTP命令用于发送邮件：HELO，MAIL，RCTP，DATA和QUIT。

我们键入mail启动用户代理，然后键入主题（subject）的提示；键入后，再键入报文的正文。在一行上键入一个句点结束报文，用户代理把邮件传给 MTA，由MTA进行交付。

客户主动打开TCP端口25。返回时，客户等待从服务器来的问候报文（应答代码为 220）。该服务器的应答必须以服务器的完全合格的域名开始：本例中为 noao.edu（通常，跟在数字应答后面的文字是可选的。这里需要域名。以 Sendmail打头的文字是可选的）。

下一步客户用 HELO命令标识自己。参数必须是完全合格的的客户主机名： sun.tuc.noao.edu。

MAIL命令标识出报文的发起人。下一个命令， RCPT，标识接收方。如果有多个接收方，可以发多个RCPT命令。

邮件报文的内容由客户通过 DATA命令发送。报文的末尾由客户指定，是只有一个句点的一行。最后的命令QUIT，结束邮件的交换。

图28-2是在发送方SMTP（客户端）与接收方SMTP（服务器）之间的一个SMTP连接。

我们键入到用户代理的数据是一行报文（“ 1，2，3 ”），但在报文段12中共发送了393字节的数据。下面的12行组成了客户发送的393字节数据：

```
Received: by sun.tuc.noao.edu. (4.1/SMI-4.1)
        id AA00502; Mon, 19 Jul 93 12:47:32 MST
Message-Id: <9307191947.AA00502@sun.tuc.noao.edu.>
From: rstevens@sun.tuc.noao.edu (Richard Stevens)
Date: Mon, 19 Jul 1993 12:47:31 -0700
```

```
Reply-To: rstevens@noao.edu
X-Phone: +1 602 676 1676
X-Mailer: Mail User's Shell (7.2.5 10/14/92)
To: rstevens@noao.edu
Subject: testing

1, 2, 3.
```

前三行，`Received:`和`Message-Id:`由MTA加上；下一行由用户代理生成。

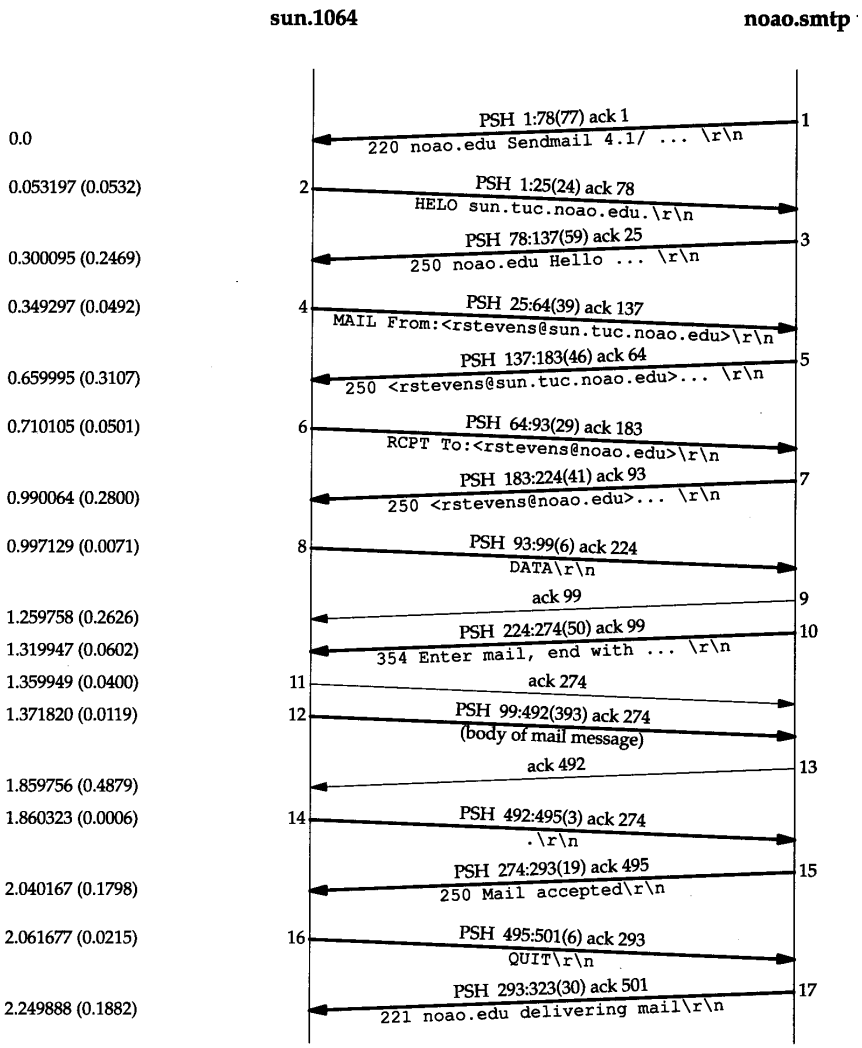


图28-2 基本SMTP邮件交付

28.2.2 SMTP命令

最小SMTP实现支持8种命令。我们在前面的例子中遇到5个：HELO，MAIL，RCPT，DATA和QUIT。

RSET命令异常中止当前的邮件事务并使两端复位。丢掉所有有关发送方、接收方或邮件的存储信息。

VRFY命令使客户能够询问发送方以验证接收方地址，而无需向接收方发送邮件。通常是系统管理员在查找邮件交付差错时手工使用的。我们将在下一节中给出这方面的例子。

NOOP命令除了强迫服务器响应一个OK应答码（200）外，不做任何事情。

还有附加和可选命令。EXPN扩充邮件表，与VRFY类似，通常是由系统管理员使用的。事实上，许多Sendmail的版本都把这两者等价地处理。

4.4BSD 中的Sendmail版本8不再将两者等同处理。VRFY不扩充别名也不接受.forward文件。

TURN命令使客户和服务器交换角色，无需拆除 TCP连接并建立新的连接就能以相反方向发送邮件（Sendmail不支持这个命令）。其他还有三个很少被实现的命令（SEND、SOML和SAML）取代MAIL命令。这三个命令允许邮件直接发送到客户终端（如果已注册）或发送到接收方的邮箱。

28.2.3 信封、首部和正文

电子邮件由三部分组成：

1) 信封（envelope）是MTA用来交付的。在我们的例子中信封由两个 SMTP命令指明：

```
MAIL From: <rstevens@sun.tuc.noao.edu>
RCPT To: <estevens@noao.edu>
```

RFC 821指明了信封的内容及其解释，以及在一个 TCP连接上用于交换邮件的协议。

2) 首部由用户代理使用。在我们的例子中可以看到 9个首部字段：`Received`、`Message-Id`、`From`、`Data`、`Reply-To`、`X-Phone`、`X-Mailer`、`To`和`Subject`。每个首部字段都包含一个名，紧跟一个冒号，接着是字段值。RFC 822指明了首部字段的格式的解释（以X-开始的首部字段是用户定义的字段，其他是由 RFC 822定义的）。长首部字段，如例子中的Received，被折在几行中，多余行以空格开头。

3) 正文（body）是发送用户发给接收用户报文的内容。RFC 822 指定正文为NVT ASCII文字行。当用DATA命令发送时，先发送首部，紧跟一个空行，然后是正文。用 DATA命令发送的各行都必须小于1000字节。

用户接收我们指定为正文的部分，加上一些首部字段，并把结果传到 MTA。MTA加上一些首部字段，加上信封，并把结果发送到另一个 MTA。

内容（content）通常用于描述首部和正文的结合。内容是客户用 DATA命令发送的。

28.2.4 中继代理

在我们的例子中本地MTA的信息输出的第1行是：“ Connecting to mailhost via ether ”（即“ 通过以太网连接到邮件主机 ”）。这是因为作者的系统已被配置成把所有非本地的向外的邮件发送到一台中继机上进行转发。

这样做的原因有两个。首先，简化了除中继系统 MTA外的其他所有MTA的配置（所有曾使用过Sendmail的人都能证明，配置一个MTA并不简单）。第二，它允许某个机构中的一个系统作为邮件集线器，从而可能把其他所有系统隐藏起来。

在这个例子中，中继系统在本地域（.tuc.noao.edu）中有一个mailhost的主机名，而其他所有系统都被配置成把它们的邮件发往该主机。我们可以执行 host命令来看看在DNS中这个名

是如何定义的：

```
sun % host mailhost
mailhost.tuc.noao.edu     CNAME     noao.edu          规范名
noao.edu                    A       140.252.1.54      它的真实IP地址
```

如果将来用于中继的主机改变了，只需改变它的 DNS名 —— 其他所有单个系统的邮箱配置都无需改变。

目前许多机构都采用中继系统。图 28-3是修改后的Internet邮件图（图28-2），考虑发送主机和最后的接收主机都可能使用中继主机。

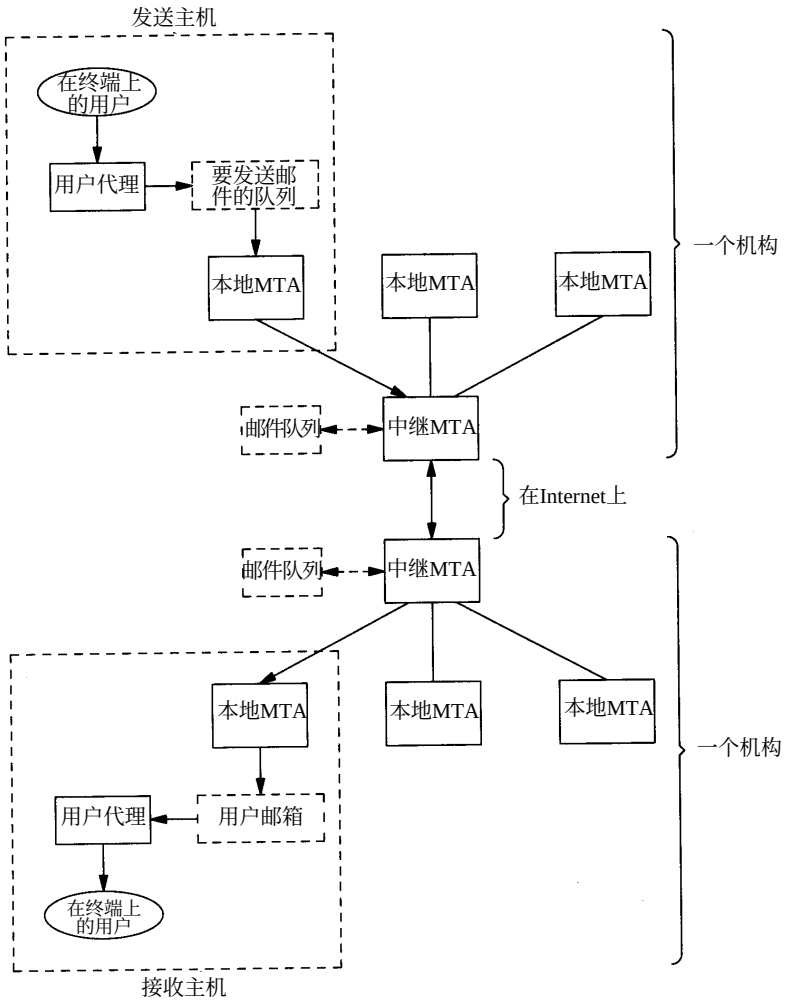


图28-3 在两端都有一个中继系统的Internet电子邮件

在这种情况下，在发送方和接收方之间有 4个MTA。发送方主机上的本地 MTA只把邮件交给它自己的中继 MTA（该中继 MTA可能在该机构的域中有一个 mailhost的主机名）。这个通信就在该机构的本地互联网上用 SMTP。然后，发送方机构的中继 MTA就在Internet上把邮件发送到接收方机构的中继 MTA上，而这个中继 MTA就通过与接收方主机上的本地 MTA通信，把邮件交给接收方主机。尽管可能存在其他协议，但这个例子中所有 MTA均使用SMTP协议。

28.2.5 NVT ASCII

SMTP的一个特色是它用NVT ASCII表示一切：信封、首部和正文。正如我们在 26.4节中谈到的，这是一个7 bit的字符码，以8 bit字节发送，高位比特被置为0。

在28.4节中，我们讨论了Internet邮件的一些新特性、允许发送和接收诸如音频和视频数据的扩充SMTP和多媒体邮件（MIME）。我们将看到，MIME和NVT ASCII一起表示信封、首部和正文，只需对用户代理作一些改变。

28.2.6 重试间隔

当用户把一个新的邮件报文传给它的 MTA时，通常立即试图交付。如果交付失败， MTA必须把该报文放入队列中以后再重试。

Host Requirements RFC推荐初始时间间隔至少为30分钟。发送方至少4~5天内不能放弃。而且，因为交付失败通常是透明的（接收方崩溃或临时网络连接中断），所以当报文在队列中等待的第1个小时内，尝试两次连接是有意义的。

28.3 SMTP的例子

上面我们说明了普通邮件发送，在这里我们将说明 MX记录如何用于邮件发送，以及VRFY和EXPN命令的用法。

28.3.1 MX记录：主机非直接连到Internet

在14.6节中我们提到DNS中的一种资源记录类型是邮件交换记录，称为 MX记录。在下面的例子中我们将说明如何用 MX记录向不直接连到 Internet的主机发送邮件。 RFC 974 [Partridge 1986] 描述了MTA对MX记录的处理。

主机mlfarm.com不是直接连到Internet的，但是有一个MX记录指向Internet上的一个邮件转发器。

```
sun % host -a -v -t mx mlfarm.com
The following answer is not authoritative:
mlfarm.com              86388    IN    MX    10 mercury.hsi.com
mlfarm.com              86388    IN    MX    15 hsi86.hsi.com
Additional information:
mercury.hsi.com         86388    IN     A    143.122.1.91
hsi86.hsi.com          172762    IN     A    143.122.1.6
```

有两个MX记录，各有不同的优先级。我们希望MTA从优先级数值低的开始。

```
sun % mail -v ron@mlfarm.com                    -v标志看MTA在做什么
To: ron@mlfarm.com
Subject: MX test message

                                                在这里键入报文的正文(没显示出来)
.                                               一行中的一个句号，结束报文

Sending letter ... ron@mlfarm.com...
Connecting to mlfarm.com via tcp...
mail exchanger is mercury.hsi.com               找到MX记录
Trying 143.122.1.91...  connected.             先试优先级低的那一个
220 mercury.hsi.com ...

                                                下面是正常的SMTP邮件传送
```

从输出中我们看到，MTA发现目的主机有一个MX记录，并使用具有低优先级数值的MX记录。

在主机sun运行这个例子之前，它被配置成不使用本地中继主机，所以我们会看到与目的主机的邮件交换。主机sun还被配置成可使用主机noao.edu（通过拨号SLIP链路）上的域名服务器，所以我们能用tcpdump捕获在SLIP链路上进行的邮件发送和DNS通信。图28-4显示了tcpdump输出的开始部分。

```
1  0.0                 sun.1624 > noao.edu.53: 2+ MX? mlfarm.com. (28)
2  0.445572 (0.4456)   noao.edu.53 > sun.1624: 2* 2/0/2 MX
                                                mercury.hsi.com. 10 (113)

3  0.505739 (0.0602)   sun.1143 > mercury.hsi.com.25: S 1617536000:1617536000(0)
                                                       win 4096
4  0.985428 (0.4797)   mercury.hsi.com.25 > sun.1143: S 1832064000:1832064000(0)
                                                       ack 1617536001 win 16384
5  0.986003 (0.0006)   sun.1143 > mercury.hsi.com.25: . ack 1 win 4096
6  1.735360 (0.7494)   mercury.hsi.com.25 > sun.1143: P 1:90(89) ack 1 win 16384
```

图28-4 向一个使用MX记录的主机发送邮件

在第1行，MTA向它的域名服务器查询mlfarm.com的MX记录。跟在2后面的加号“+”意思是设置要求递归的标志位。第2行的响应置位授权比特（跟在2后面的星号“*”），并包含两个回答RR（两个MX主机名），0个授权RR，以及两个附加的RR（两个主机的IP地址）。

第3~5行与主机mercury.hsi.com上的SMTP建立了一个TCP连接。服务器的初始响应220显示在第6行。

由于某种原因，主机mercury.hsi.com必须把这个邮件报文交付给目的地，mlfarm.com。对于没有连接到Internet上与它的MX站点交换邮件的系统，UUCP协议是一种常用的办法。

在这个例子中，MTA要求一个MX记录，得到一个肯定的结果，然后发送邮件。但不幸的是，MTA与DNS之间的交互随不同的实现而不同。RFC 974指定MTA必须首先要求MX记录，如果没有，就尝试提交给目的主机（也就是说，向DNS要主机的记录和IP地址）。MTA也必须处理DNS中的CNAM记录（规范的名）。

作为一个例子，如果我们从一个BSD/386主机上向rstevens@mailhost.tuc.noao.edu发送邮件，则MTA（Sendmail）执行以下步骤：

1) Sendmail向DNS询问主机mailhost.tuc.noao.edu的CNAME记录。我们看到存在一个CNAME记录：

```
sun % host -t cname mailhost.tuc.noao.edu
mailhost.tuc.noao.edu     CNAME    noao.edu
```

2) 发布一个要求noao.edu的CNAME记录的DNS查询，回答是不存在。

3) Sendmail向DNS寻求noao.edu的MX记录并得到一个记录：

```
sun % host -t mx noao.edu
noao.edu                  MX       noao.edu
```

4) Sendmail向DNS查询noao.edu的A记录（IP地址），并得到返回值140.252.1.54（这个A记录大概是由域名服务器为noao.edu返回的，作为第3步中MX应答的一个附加的RR）。

5) 启动一个到140.252.1.54的SMTP连接并发送邮件。

CNAME查询不是为MX记录（noao.edu）中返回的数据做的。MX记录中的数据不能是

别名——必须是具有一个A记录的主机名。

与只用DNS 的SunOS 4.1.3一起发布的Sendmail版本查询MX记录，并且如果没有找到MX记录就放弃。

28.3.2 MX记录：主机出故障

MX记录的另一个用途是在目的主机出故障时可提供另一个邮件接收器。如果看一下主机sun的DNS入口，我们就会看到它有两个MX记录：

```
sun % host -a -v -t mx sun.tuc.noao.edu
sun.tuc.noao.edu        86400     IN     MX       0 sun.tuc.noao.edu
sun.tuc.noao.edu        86400     IN     MX      10 noao.edu
Additional information:
sun.tuc.noao.edu        86400     IN      A      140.252.1.29
sun.tuc.noao.edu        86400     IN      A      140.252.13.33
noao.edu                86400     IN      A      140.252.1.54
```

最低优先级的MX记录表明应该首先尝试直接发送到主机本身，下一个优先级是把邮件发送到主机noao.edu。

在下面的描述中，在关掉目的SMTP服务器后，我们从主机vangogh.cs.berkeley.edu向位于主机sun.tuc.noao.edu的我们自己发送邮件。当端口25上的连接请求到达时，TCP应该响应一个RST，因为没有被动打开的进程为等待该端口而挂起。

```
vangogh % mail -v rstevens@sun.tuc.noao.edu
A test to a host that's down.
.
EOT
rstevens@sun.tuc.noao.edu... Connecting to sun.tuc.noao.edu. (smtp)...
rstevens@sun.tuc.noao.edu... Connecting to noao.edu. (smtp)...
220 noao.edu ...
```

下面是正常的*SMTP*邮件传送

我们看到MTA尝试联系sun.tuc.noao.edu，然后放弃，并转而联系noao.edu。

图28-5显示了TCP用一个RST向到来的SYN响应的tcpdump输出。

```
1 0.0                vangogh.3873 > 140.252.1.29.25: S 2358303745:2358303745(0) ...
2 0.000621 (0.0006) 140.252.1.29.25 > vangogh.3873: R 0:0(0) ack 2358303746 win 0
3 0.300203 (0.2996) vangogh.3874 > 140.252.13.33.25: S 2358367745:2358367745(0) ...
4 0.300620 (0.0004) 140.252.13.33.25 > vangogh.3874: R 0:0(0) ack 2358367746 win 0
```

图28-5 尝试连接一个不在运行的SMTP服务器

第1行vangogh向sun的第1个IP地址140.252.1.29 的端口25 发送一个SYN。在第2行它被拒绝。然后，vangogh上的SMTP客户尝试sun的第2个IP地址140.252.13.33（第3行），也产生一个RST的返回（第4行）。

SMTP客户不区分第1行它主动打开时所返回的不同差错，而这是导致它在第 2行尝试其他IP地址的原因。如果第1次的差错是类似“ host unreachable(主机不可达)”，那么第2次尝试或许可行。

如果SMTP客户的主动打开失败的原因是因为服务器主机出故障了，我们将看到客户会向IP地址140.252.1.29重传SYN总共75秒（类似于图18-6）。然后客户向IP地址140.252.13.33发送另一个75秒的其他3个SYN。150秒后客户会移到下一个具有更高优先级的 MX记录。

28.3.3 VRFY和EXPN命令

VRFY命令无需发送邮件而验证某个接收方地址是否OK。EXPN的目的是无需向邮件表发送邮件就可以扩充该表。许多SMTP实现（如Sendmail）把两者看成一个，但我们提到新的Sendmail区分这两者。

作为一个简单测试，我们可以连到一个新的Sendmail版本，并看到不同之处（已经删除了无关的Telnet客户输出）。

```
sun % telnet vangogh.cs.berkeley.edu 25
220-vangogh.CS.Berkeley.EDU Sendmail 8.1C/6.32 ready at Tue, 3 Aug 1993 14:
59:12 -0700
220 ESMTP spoken here
helo bsdi.tuc.noao.edu
250 vangogh.CS.Berkeley.EDU Hello sun.tuc.noao.edu [140.252.1.29], pleased
to meet you
vrfy nosuchname
550 nosuchname... User unknown
vrfy rstevens
250 Richard Stevens <rstevens@vangogh.CS.Berkeley.EDU>
expn rstevens
250 Richard Stevens <rstevens@noao.edu>
```

首先注意到我们故意在HELO命令中键入错误的主机名：bsdi，而不是sun。许多SMTP服务器得到客户的IP地址，完成一个DNS指针查询（14.5节）并比较主机名。这样允许服务器基于IP地址注册到客户的连接，而不是基于用户可能错误键入的名。某些服务器会用幽默的报文回答，如“你是一个骗子”，或“为什么叫你自己……”。在这个例子中我们看到，这个服务器通过指针查询只打印出我们的真实域名以及我们的IP地址。

然后我们用一个无效的名字键入VRFY命令，服务器就响应550差错。下一步我们键入一个有效的名字，服务器用本地主机上的用户名回答。然后我们试试EXPN命令，并得到一个不同的回答。EXPN命令决定到该用户的邮件是否被转发，并打印出转发的地址。

许多站点禁止VRFY和EXPN命令，有时是因为隐私，有时因为相信这是安全漏洞。例如，我们可以向白宫的SMTP服务器试试下面的命令：

```
sun % telnet whitehouse.gov 25
220 whitehouse.gov SMTP/smap Ready.
helo sun.tuc.noao.edu
250 (sun.tuc.noao.edu) pleased to meet you.
vrfy clinton
500 Command unrecognized
expn clinton
500 Command unrecognized
```

28.4 SMTP的未来

Internet邮件发生了很多改变。应当记得Internet邮件的三个组成部分：信封、首部和正文。新加入的SMTP命令影响了信封，首部中可以使用非ASCII字母，正文（MIME）中也加入了结构。本节中我们依次对这三部分的扩充进行讨论。

28.4.1 信封的变化：扩充的SMTP

RFC 1425 [Klensin等，1993a] 定义了扩充的SMTP的框架，其结果被称为扩充的 SMTP（ESMTP）。与其他我们已经讨论过的新特性一样，这些变化以向后兼容的方式被加入，所以不影响已有的实现。

如果客户想使用新的特性，首先通过发布一个 EHLO而不是HELO命令启动一个与服务器的会话。相兼容的服务器用 250应答码响应。这个应答通常有好几行，每行都包含一个关键字和一个可选的参数。这些关键字指定了该服务器支持的 SMTP扩充。新的扩充将在一个RFC中描述并以IANA注册（在一个多行应答中，各行数字应答码的后面都要有一个连字符。最后一行的数字应答码后面跟一个空行）。

我们将给出到4个SMTP服务器的初始连接，其中 3个支持扩充的SMTP。我们用Telnet和它们连接，但删掉了不必要的Telnet客户输出。

```
sun % telnet vangogh.cs.berkeley.edu 25
220-vangogh.CS.Berkeley.EDU Sendmail 8.1C/6.32 ready at Mon, 2 Aug 1993 15:
47:48 -0700
220 ESMTP spoken here

ehlo sun.tuc.noao.edu
250-vangogh.CS.Berkeley.EDU Hello sun.tuc.noao.edu [140.252.1.29], pleased
to meet you
250-EXPN
250-SIZE
250 HELP
```

这个服务器用一个多行220应答作为它的欢迎报文。对EHLO命令的250应答中列出的扩充命令是EXPN、SIZE和HELP。第一个和最后一个来自原来的 RFC 821规范，但它们是可选命令。ESMTP服务器说明除了新命令外，它们还支持哪些可选的 RFC 821命令。

这个服务器支持的SIZE关键字是在RFC 1427 [Klensin, Freed和Moore 1993] 中定义的。它让客户在MAIL FROM命令行中以字节的多少指定报文的大小，这样服务器就可以在客户开始发送该报文之前，验证它是否接收该长度的报文。增加这个命令的原因在于，随着对非 ASCII码（如图像、音频等）内容的支持， Internet邮件报文的长度在不断增大。

下一个主机也支持 ESMTP，注意250应答指明支持包含一个可选参数的 SIZE关键字。这表明该服务器将接受长度不超过461兆字节的报文。

```
sun % telnet ymir.claremont.edu 25
220 ymir.claremont.edu -- Server SMTP (PMDF V4.2-13 #4220)

ehlo sun.tuc.noao.edu
250-ymir.claremont.edu
250-8BITMIME
250-EXPN
250-HELP
250-XADR
250 SIZE 461544960
```

关键字8BITMIME来自于RFC 1426 [Klensin等，1993a]。它允许客户把关键字BODY加到MAIL FROM命令中，指定正文中是否包含NVT ASCII字符（默认的）或8 bit数据。除非客户收到服务器应答EHLO命令发来的8BITMIME关键字，否则禁止客户发送任何非 NVT ASCII字符（当我们在本节中谈到MIME时，我们将看到MIME不要求8 bit传送）。

该服务器也通告了XADR关键字。任何以X开头的关键字都指的是本地SMTP扩充。

另一个服务器也支持ESMTP，通知了我们已经看到的HELP和SIZE关键字。它也支持三个以X开头的本地扩充。

```
sun % telnet dbc.mtview.ca.us 25
220 dbc.mtview.ca.us Sendmail 5.65/3.1.090690, it's Mon, 2 Aug 93 15:48:50
-0700
ehlo sun.tuc.noao.edu
250-Hello sun.tuc.noao.edu, pleased to meet you
250-HELP
250-SIZE
250-XONE
250-XVRB
250 XQUE
```

最后，我们将看到当客户试图通过向一个不支持 EHLO的服务器发布EHLO命令来使用ESMTP时将发生什么。

```
sun % telnet relay1.uu.net 25
220 relay1.UU.NET Sendmail 5.61/UUNET-internet-primary ready at Mon, 2 Aug
93 18:50:27 -0400
ehlo sun.tuc.noao.edu
500 Command unrecognized
rset
250 Reset state
```

对EHLO命令，客户收到一个500应答而不是250应答。客户应发布RSET命令，并跟着一个HELO命令。

28.4.2 首部变化：非ASCII字符

RFC 1522 [Moore 1993] 指明了一个在RFC 822报文首部中如何发送非ASCII字符的方法。这样做的主要用途是为了允许在发送方名、接收方名以及主题中使用其他的字符。

首部字段中可以包含编码字(coded word)。它们具有以下格式：

=?*charset*?*encoding*?*encoded-text*?=

charset是字符集规范。有效值是两个字符串us-ascii和iso-8859-x，其中x是一个单个数字，例如在iso-8859-1中的数字“1”。

encoding是一个单个字符用来指定编码方法，支持两个值。

1) Q编码意思是引号中可打印的（quoted-printable），目的是用于拉丁字符集。大多数字符是作为NVT ASCII（当然最高位比特置0）发送的。任何要发送的字符若其第8比特置1则被作为3个字符发送：第1个是字符是“=”，跟着两个十六进制数。例如，字符é（它的二进制8 bit值为0xe9）作为三个字符发送：=E9。空格通常作为下划线或三个字符=20发送。这种编码的目的在于，某些文本中除了大多数ASCII字符外，还有几个特殊字符。

2) B意思是以64为基数的编码。文本中的3个连续字节（24bit）被编码成4个6 bit值。用于表示所有可能的6bit值的64个NVT ASCII字符如图28-6所示。当要编码的个数不是3的倍数时，等号符“=”被用作填充符。

下面两种编码方式的例子取自RFC 1522：

```
From: =?US-ASCII?Q?Keith_Moore?= <moore@cs.utk.edu>
To: =?ISO-8859-1?Q?Keld_J=F8rn_Simonsen?= <keld@dkuug.dk>
CC: =?ISO-8859-1?Q?Andr=E9_?= Pirard <PIRARD@vm1.ulg.ac.be>
Subject: =?ISO-8859-1?B?SWYgeW91IGNhbiByZWFkIHRoaXMgeW8=?=
 =?ISO-8859-2?B?dSB1bmRlcnN0YW5kIHRoZSBleGFtcGxlLg==?=
```

6 bit 值	ASCII 字符	6 bit 值	ASCII 字符	6 bit 值	ASCII 字符	6 bit 值	ASCII 字符
0	A	10	Q	20	g	30	w
1	B	11	R	21	h	31	x
2	C	12	S	22	i	32	y
3	D	13	T	23	j	33	z
4	E	14	U	24	k	34	0
5	F	15	V	25	l	35	1
6	G	16	W	26	m	36	2
7	H	17	X	27	n	37	3
8	I	18	Y	28	o	38	4
9	J	19	Z	29	p	39	5
a	K	1a	a	2a	q	3a	6
b	L	1b	b	2b	r	3b	7
c	M	1c	c	2c	s	3c	8
d	N	1d	d	2d	t	3d	9
e	O	1e	e	2e	u	3e	+
f	P	1f	f	2f	v	3f	/

图28-6 6 bit值的编码（以64为基数编码）

能处理这些首部的用户代理将输出：

```
From: Keith Moore <moore@cs.utk.edu>
To: Keld Jørn Simonsen <keld@dkuug.dk>
CC: André  Pirard <PIRARD@vm1.ulg.ac.be>
Subject: If you can read this you understand the example.
```

为说明以64为基数的编码方法是如何工作的，我们看一下主题行中前面 4个编码的字符：SWYg。按照图28-6写出这4个字符的6 bit值（S=0x12,W=0x16,Y=0x18以及g=0x20）的二进制码：

```
010010  010110  011000  100000
```

然后把这24 bit重新分组成3个8 bit字节：

```
01001001  01100110  00100000
 =0x49     =0x66     =0x20
```

它们是I、f和空格的ASCII表示。

28.4.3 正文变化：通用Internet邮件扩充

我们已经提到RFC 822指定正文是NVT ASCII文本行，没有结构。RFC 1521 [Borenstein和Freed 1993] 把扩充定义为允许把结构置入正文。这被称为MIME，即通用Internet邮件扩充。

MIME不要求任何扩充，我们在本节前面已作了说明（扩充的 SMTP或非ASCII标题）。MIME正好加入了一些告知收件者正文结构的新标题（与 RFC 822相一致）。正文仍可以用NVT ASCII码来发送，而不考虑邮件内容。虽然我们前面所述的一些扩充可能会和 MIME合在一起产生好的效果——扩充的SMTP SIZE命令，因为MIME报文能变得很长，以及非ASCII标题——这些扩充并不是MIME所要求的。与另一方交换MIME报文所需的一切，就是双方都要有一个能够理解MIME的用户代理。在任何一个MTA中不需要做任何改变。

MIME定义这5个新标题字段如下：

```
Mime-Version:
Content-Type:
Content-Transfer-Encoding:
```

```
Content-ID:
Content-Description:
```

作为例子，下面两个标题行可以出现在一个 Internet邮件报文中：

```
Mime-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
```

当前MIME版本是1.0，内容类型是无格式 ASCII码文本，即Internet邮件的默认选择。PLAIN这个字被认为是内容类型（ TEXT）的一个子类型，字符串 charset=US-ASCII是一个参数。

Text是MIME的7个被定义的内容类型之一。图 28-7总结了RFC 1521中定义的16个不同的内容类型和子类型。对具体的内容类型和子类型来说都有指定的很多参数。

内容类型	子类型	描　　述
text	plain richtext enriched	无格式文本 简单格式文本，如粗体、斜体或下划线等 richtext的简化和改进
multipart	mixed parallel digest altenative	多个正文部分，串行处理 多个正文部分，可并行处理 一个电子邮件的摘要 多个正文部分，具有相同的语义内容
message	rfc822 partial external-body	内容是另一个RFC 822邮件报文 内容是一个邮件报文的片断 内容是指向实际报文的指针
application	octet-stream postscript	任意二进制数据 一个PostScript程序
image	jpeg gif	ISO 10918格式 CompuServe的图形交换格式
audio	basic	用8 bit ISDN μ律格式编码
video	mpeg	ISO 11172格式

图28-7　MIME 内容类型和子类型

内容类型和用于内容的传送编码是相互独立的。前者由首部字段 Content-Type指明，后者由首部字段Content-Transfer-Encoding指明。在RFC 1521中定义了5种不同的编码格式。

1) 7bit，是默认的NVT ASCII；

2) quoted-printable，我们在前面的一个例子中看到有非 ASCII首部。当字符中只有很少一部分的第8 bit置1时非常有用；

3) base64，如图28-6所示；

4) 8bit，包含字符行，其中某些为非ASCII字符且第8bit置1；

5) binary编码，无需包含多行的8 bit数据。

对RFC 821 MTA，以上5种编码格式中只有前 3种是有效的。因为这 3种产生只包含NVT ASCII字符的正文。使用有8BITMIME支持的扩充SMTP允许使用8bit编码。

尽管内容类型和编码是独立的， RFC1521推荐有非ASCII数据的text使用quoted-printable，而image、audio、video和octet-stream application使用base64。这样允许与符合 RFC 821的MTA保持最大的互操作性。而且， multipart和message内容类型必须以7bit编码。

作为一个multipart内容类型的例子，图28-8显示了一个来自RFC发布清单的邮件报文。子类型是mixed，意思是各部分是顺序处理的，各部分的边界是字符串 NextPart，其前面是行首的两个连字符。

每个边界上可跟一行用于指明下一部分首部字段。忽略报文中第 1个边界之前和最后一个边界之后的所有内容。

因为在第一个边界后面跟着一个空行，而不是首部，所以在第 1个和第2个边界之间的数据的内容类型被假定为具有us-ascii字符集的text/plain。这是新RFC的文字描述。

但是第2个边界后面跟着首部字段。它指定了另一个multipart报文，具有边界OtherAccess。子类型为alternative，有两种不同的选择。第1种OtherAccess选项是用电子邮件获取RFC，第2种选项是用匿名FTP获取。MIME用户代理将列出这两种选项，允许我们选择一个，然后自动地用电子邮件或匿名FTP获取一份复制的RFC。

```
To: rfc-dist@nic.ddn.mil
Subject: RFC1479 on IDPR Protocol
Mime-Version: 1.0
Content-Type: Multipart/Mixed; Boundary="NextPart"
Date: Fri, 23 Jul 93 12:17:43 PDT
From: "Joyce K. Reynolds" <jkrey@isi.edu>


--NextPart                                  第1个边界

A new Request for Comments is now available in online RFC libraries.

. . .                                       这里的细节在新的RFC中

Below is the data which will enable a MIME compliant Mail Reader
implementation to automatically retrieve the ASCII version
of the RFCs.

--NextPart                                  第2个边界
Content-Type: Multipart/Alternative; Boundary="OtherAccess"
                                            一个具有新边界的嵌套的多部分报文

--OtherAccess
Content-Type:  Message/External-body;
        access-type="mail-server";
        server="mail-server@nisc.sri.com"

Content-Type: text/plain

SEND rfc1479.txt

--OtherAccess
Content-Type:   Message/External-body;
        name="rfc1479.txt";
        site="ds.internic.net";
        access-type="anon-ftp";
        directory="rfc"

Content-Type: text/plain


--OtherAccess--
--NextPart--                                最后的边界
```

图28-8 MIME multipart报文的例子

这一部分是MIME的一个简要概述。MIME的详细细节和例子，见RFC 1521和[Rose 1993]。

28.5 小结

电子邮件包括在两端（发送方和接收方）都有的一个用户代理以及两个或多个报文传送代理。可以把一个邮件报文分成三个部分：信封、首部和正文。我们已经看到这三个部分用SMTP和Internet标准是如何进行交换的。所有都作为NVT ASCII字符进行交换。

我们也看到了一些新的扩充：用于信封和非ASCII首部的扩充SMTP，以及使用MIME的正文增加了结构。MIME的结构和编码允许使用已有的7bit SMTP MTA交换任意二进制数据。

习题

28.1 读RFC 822，找到域文字（domain literal）的意思。试试用其中一个给自己发送邮件。

28.2 除了连接建立和终止外，要发送一个小的邮件报文的最小网络往返次数是多少？

28.3 TCP是一个全双工协议，但是SMTP用半双工的形式使用TCP。客户发送一个命令后停止等待应答。为什么客户不一次发送多个命令，如一行中包括HELO、MAIL、RCPT、DATA和QUIT命令（假定正文不是太大）？

28.4 当网络在接近其容量运行时，SMTP的这种半双工操作如何欺骗缓慢的启动机制？

28.5 当存在多个具有相同优先值的MX记录时，名服务器是否总能以相同的顺序返回它们？

第29章　网络文件系统

29.1　引言

本章中我们要讨论另一个常用的应用程序： NFS（网络文件系统），它为客户程序提供透明的文件访问。NFS的基础是Sun RPC：远程过程调用。我们首先必须描述一下 RPC。

客户程序使用 NFS不需要做什么特别的工作，当 NFS内核检测到被访问的文件位于一个NFS服务器时，就会自动产生一个访问该文件的 RPC调用。

我们对NFS如何访问文件的细节并不感兴趣，只对它如何使用 Internet的协议，尤其是UDP协议，感兴趣。

29.2　Sun远程过程调用

大多数的网络程序设计都是编写一些调用系统提供的函数来完成特定的网络操作的应用程序。例如，一个函数完成 TCP的主动打开，另一个完成 TCP的被动打开，一个函数在一个TCP连接上发送数据，另一个设置特定的协议选项（如激活 TCP的keepalive定时器）。在1.15节我们提到过两个常用的用于网络编程的函数集（ API）：插口(socket)和TLI。正像客户端和服务器端运行的操作系统可能会不相同一样，双方使用的 API也可能会不相同。由通信协议和应用协议决定一对客户和服务器是否可以彼此通信。如果两台主机连接在一个网络上，并且都有一个TCP/IP的实现，那么一台主机上的一个使用 C语言编写的、使用插口和TCP的Unix客户程序可以和另一台主机上的一个使用 COBOL语言编写的、使用其他 API和TCP的大型机服务器进行通信。

一般来说，客户发送命令给服务器，服务器向客户发送应答。目前为止，我们讨论过的所有应用程序 —— Ping，Traceroute，选路守护程序、以及 DNS、TFTP、BOOTP、SNMP、Telnet、FTP和SMTP的客户和服务器 —— 都是采用这种方式实现的。

远程过程调用RPC (Remote Procedure Call)是一种不同的网络程序设计方法。客户程序编写时只是调用了服务器程序提供的函数。这只是程序员所感觉到的，实际上发生了下面一些动作。

1) 当客户程序调用远程的过程时，它实际上只是调用了一个位于本机上的、由 RPC程序包生成的函数。这个函数被称为客户残桩（ stub）。客户残桩将过程的参数封装成一个网络报文，并且将这个报文发送给服务器程序。

2) 服务器主机上的一个服务器残桩负责接收这个网络报文。它从网络报文中提取参数，然后调用应用程序员编写的服务器过程。

3) 当服务器函数返回时，它返回到服务器残桩。服务器残桩提取返回值，把返回值封装成一个网络报文，然后将报文发送给客户残桩。

4) 客户残桩从接收到的网络报文中取出返回值，将其返回给客户程序。

网络程序设计是通过残桩和使用诸如插口或 TLI的某个API的RPC库例程来实现的，但是

用户程序——客户程序和被客户程序调用的服务器过程——不会和这个API打交道。客户应用程序只是调用服务器的过程，所有网络程序设计的细节都被 RPC程序包、客户残桩和服务器残桩所隐藏。

一个RPC程序包提供了很多好处。

1) 程序设计更加容易，因为很少或几乎没有涉及网络编程。应用程序设计员只需要编写一个客户程序和客户程序调用的服务器过程。

2) 如果使用了一个不可靠的协议，如 UDP，像超时和重传等细节就由 RPC程序包来处理。这就简化了用户应用程序。

3) RPC库为参数和返回值的传输提供任何需要的数据转换。例如，如果参数是由整数和浮点数组成的， RPC程序包处理整数和浮点数在客户机和服务器主机上存储的不同形式。这个功能简化了在异构环境中的客户和服务器的编码问题。

RPC程序设计的细节可以参看参考文献 [Stevens 1990]的第18章。两个常用的RPC程序包是Sun RPC和开放软件基金（OSF）分布式计算环境（DCE）的RPC程序包。我们对于RPC的兴趣在于想了解Sun RPC中过程调用和过程返回报文的形式，因为本章中讨论的网络文件系统使用了它们。Sun RPC的第2版定义在RFC 1057 [Sun Microsystems 1988a]中。

Sun RPC

Sun RPC有两个版本。一个版本建立在插口API基础上，和TCP和UDP打交道。另一个称为 TI-RPC的（独立于运输层），建立在TLI API基础上，可以和内核提供的任何运输层协议打交道。尽管本章中我们只讨论 TCP和UDP，从讨论的观点来看，两者是一样的。

图29-1显示的是使用 UDP时，一个RPC过程调用报文的格式。 IP首部和UDP首部是标准的首部，我们已经在图 3-1和图11-2中显示过。 UDP首部以下是 RPC程序包定义的部分。

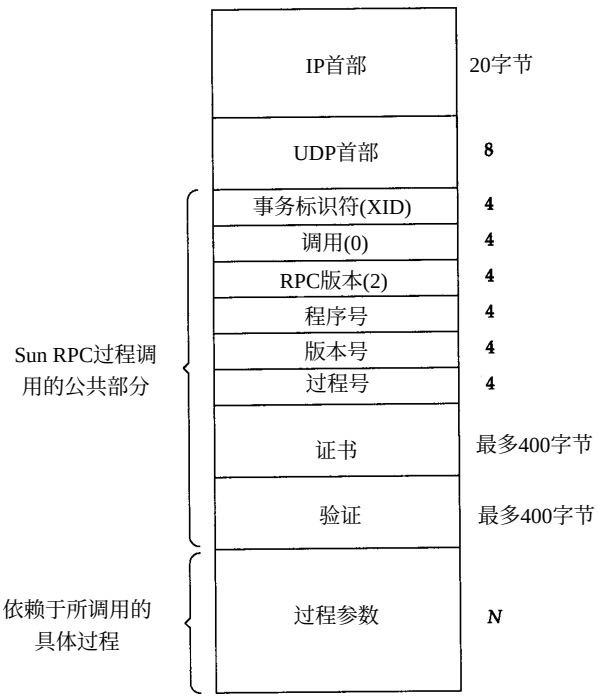


图29-1 RPC过程调用报文作为一个UDP数据报的格式

事务标识符（XID）由客户程序设置，由服务器程序返回。当客户收到一个应答，它将服务器返回的 XID与它发送的请求的XID相比较。如果不匹配，客户就放弃这个报文，等待从服务器返回的下一个报文。每次客户发出一个新的 RPC，它就会改变报文的XID。但是如果客户重传一个以前发送过的RPC（因为它没有收到服务器的一个应答），重传报文的XID不会修改。

调用(call)变量在过程调用报文中设置为 0，在应答报文中设置为 1。当前的RPC版本是2。接下来三个变量：程序号、版本号和过程号，标识了服务器上被调用的特定过程。

证书(credential)字段标识了客户。有些情况下，证书字段设置为空值；另外一些情况下，证书字段设置为数字形式的客户的用户号和组号。服务器可以查看证书字段以决定是否执行请求的过程。验证 (verifier)字段用于使用了DES加密的安全RPC。尽管证书字段和验证字段是可变长度的字段，它们的长度也作为字段的一部分被编码。

接下来是过程参数(procedure parameter)字段。参数的格式依赖于远程过程的定义。接收者（服务器残桩）如何知道参数字段的大小呢？既然使用的是UDP协议，UDP数据报的大小减去验证字段以上所有字段的长度就是参数的大小。如果使用的不是UDP而是TCP，因为TCP是一个字节流协议，没有记录边界，所以没有固定的长度。为了解决这个问题，在TCP首部和XID之间增加了一个4字节的长度字段，告诉接收者这个RPC调用由多少字节组成。这也使得一个RPC调用报文在必要时可以用多个TCP段来传输（DNS使用了类似的技术，参见习题14-4)。

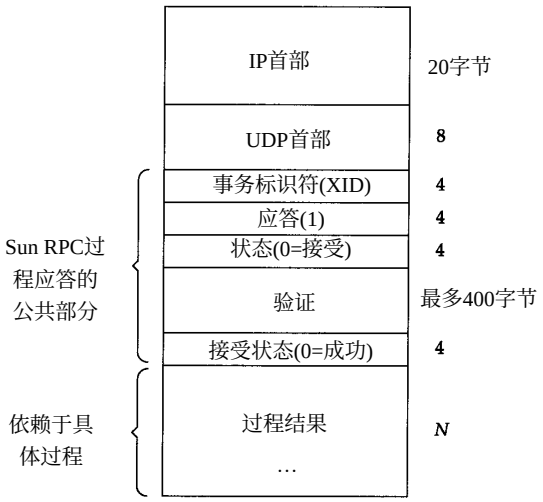


图29-2 RPC应答报文作为一个UDP数据报的格式

图29-2显示了一个RPC应答报文的格式。当远程过程返回时，服务器残桩将这个报文发送给客户残桩。

应答报文中的XID字段是从调用报文的XID字段复制而来。应答字段设置为 1，以区别于调用报文。如果调用报文被接受，状态字段设置为 0（如果RPC的版本号不为2，或者服务器不能鉴别客户的身份，调用报文可能被拒绝)。安全的RPC使用验证字段来标识服务器。

如果远程过程调用成功，接受状态字段置为 0。一个非零的值可能表示一个不合法的版本号或者一个不合法的过程号。如果使用的不是 UDP而是TCP，如同RPC调用报文一样，在TCP首部和XID字段之间插入一个4字节的长度字段。

29.3 XDR: 外部数据表示

外部数据表示XDR (eXternal Data Representation)是一个标准，用来对RPC调用报文和应答报文中的值进行编码。这些值包括 RPC首部字段（XID、程序号、接受状态等)、过程参数和过程结果。采用标准化的方法对这些值进行编码使得一个系统中的客户可以调用另一个不同架构的系统中的一个过程。 XDR在RFC 1014中定义[Sun Microsystems 1987]。

XDR定义了很多数据类型以及它们如何在一个 RPC报文中传输的具体形式（如比特顺序，字节顺序等)。发送者必须采用XDR格式构造一个RPC报文，然后接收者将XDR格式的报文转换为本机的表示形式。例如，在图29-1和图29-2中，我们显示的所有整数值（XID、调用字段、程序号等）都是4字节的整数。在XDR中，所有的整数的确占据 4个字节。XDR支持的其他数据类型包括无符号整数、布尔类型、浮点数、定长数组、可变长数组和结构。

29.4 端口映射器

包含远程过程的RPC服务器程序使用的是临时端口，而不是知名端口。这就需要某种形

式的“注册”程序来跟踪哪一个RPC程序使用了哪一个临时端口。在Sun RPC中，这个注册程序被称为端口映射器(port mapper)。

> “端口”这个词作为Internet协议族的一个特征，来自于TCP和UDP端口号。既然TI-RPC可以工作在任何运输层协议之上，而不仅仅是TCP和UDP，所以使用TI-RPC的系统中（如SVR4和Solaris 2.2），端口映射器的名字变成了rpcbind。下面我们继续使用更为常见的端口映射器的名字。

很自然地，端口映射器本身必须有一个知名端口：UDP端口111和TCP端口111。端口映射器也就是一个RPC服务器程序。它有一个程序号（100000）、一个版本号（2）、一个TCP端口111和一个UDP端口111。服务器程序使用RPC调用向端口映射器注册自身，客户程序使用RPC调用向端口映射器查询。端口映射器提供四个服务过程：

1) PMAPPROC_SET。一个RPC服务器启动时调用这个过程，注册一个程序号、版本号和带有一个端口号的协议。

2) PMAPPROC_UNSET。RPC服务器调用此过程来删除一个已经注册的映射。

3) PMAPPROC_GETPORT。一个RPC客户启动时调用此过程。根据一个给定的程序号、版本号和协议来获得注册的端口号。

4) PMAPPROC_DUMP。返回端口映射器数据库中所有的记录（每个记录包括程序号、版本号、协议和端口号）：

在一个RPC服务器程序启动，接着被一个RPC客户程序调用的过程中，进行了以下一些步骤：

1) 一般情况下，当系统引导时，端口映射器必须首先启动。它创建一个TCP端点，并且被动打开TCP端口111。它也创建一个UDP端点，并且在UDP端口111等待着UDP数据报的到来。

2) 当RPC服务器程序启动时，它为它所支持的程序的每一个版本创建一个TCP端点和一个UDP端点（一个给定的RPC程序可以支持多个版本。客户调用一个服务器过程时，说明它想要哪一个版本）。两个端点各自绑定一个临时端口（TCP端口号和UDP端口号是否一致无关紧要）。服务器通过RPC调用端口映射器的PMAPPROC_SET过程，注册每一个程序、版本、协议和端口号。

3) 当RPC客户程序启动时，它调用端口映射器的PMAPPROC_GETPORT过程来获得一个指定程序、版本和协议的临时端口号。

4) 客户发送一个RPC调用报文给第3步返回的端口号。如果使用的是UDP，客户只是发送一个包含RPC调用报文（见图29-1）的UDP数据报到服务器相应的UDP端口。服务器发送一个包含RPC应答报文（见图29-2）的UDP数据报到客户作为响应。

如果使用的是TCP，客户对服务器的TCP端口号做一个主动打开，然后在建立的TCP连接上发送一个RPC调用报文。服务器作为响应，在连接上发送一个RPC应答报文。

程序`rpcinfo(8)`打印了端口映射器中当前的映射记录（它调用了端口映射器的PMAPPROC_DUMP过程）。这里给出的是典型的输出：

```
sun % /usr/etc/rpcinfo -p
   program vers proto   port
    100005    1   tcp    702  mountd      NFS的安装守护程序
    100005    1   udp    699  mountd
    100005    2   tcp    702  mountd
    100005    2   udp    699  mountd
```

```
100003    2   udp   2049  nfs          NFS本身
100021    1   tcp    709  nlockmgr     NFS的加锁管理程序
100021    1   udp   1036  nlockmgr
100021    2   tcp    721  nlockmgr
100021    2   udp   1039  nlockmgr
100021    3   tcp    713  nlockmgr
100021    3   udp   1037  nlockmgr
```

可以看出一些程序确实支持多个版本。在端口映射器中，每一个程序号、版本号和协议的组合都有自己的端口号映射。

安装守护程序（mount daemon）的两个版本可以通过同样的TCP端口号（702）和同样的UDP端口号（699）来访问，而加锁管理程序（lock manager）的每个版本都有各自不同的端口号。

29.5 NFS协议

使用NFS，客户可以透明地访问服务器上的文件和文件系统。这不同于提供文件传输的FTP（第27章）。FTP会产生文件一个完整的副本。NFS只访问一个进程引用文件的那一部分，并且NFS的一个目的就是使得这种访问透明。这就意味着任何能够访问一个本地文件的客户程序不需要做任何修改，就应该能够访问一个NFS文件。

NFS是一个使用Sun RPC构造的客户服务器应用程序。NFS客户通过向一个NFS服务器发送RPC请求来访问其上的文件。尽管这一工作可以使用一般的用户进程来实现——即NFS客户可以是一个用户进程，对服务器进行显式调用。而服务器也可以是一个用户进程——因为两个理由，NFS一般不这样实现。首先，访问一个NFS文件必须对客户透明。因此，NFS的客户调用是由客户操作系统代表用户进程来完成的。第二，出于效率的考虑，NFS服务器在服务器操作系统中实现。如果NFS服务器是一个用户进程，每个客户请求和服务器应答（包括读和写的数据）将不得不在内核和用户进程之间进行切换，这个代价太大。

本节中，我们考察在RFC1094中说明的第2版的NFS [Sun Microsystems 1988b]。[X/Open 1991] 中给出了Sun RPC、XDR和NFS的一个更好的描述。[Stern 1991] 给出了使用和管理NFS的细节。第3版的NFS协议在1993年发布，我们在29.7节中对它做一个简单的描述。

图29-3显示了一个NFS客户和一个NFS服务器的典型配置，图中有很多地方需要注意。

1) 访问的是一个本地文件还是一个NFS文件对于客户来说是透明的。当文件被打开时，由内核决定这一点。文件被打开之后，内核将本地文件的所有引用传递给名为“本地文件访问”的框中，而将一个NFS文件的所有引用传递给名为“NFS客户”的框中。

2) NFS客户通过它的TCP/IP模块向NFS服务器发送RPC请求。NFS主要使用UDP，最新的实现也可以使用TCP。

3) NFS服务器在端口2049接收作为UDP数据报的客户请求。尽管NFS可以被实现成使用端口映射器，允许服务器使用一个临时端口，但是大多数的实现都是直接指定UDP端口2049。

4) 当NFS服务器收到一个客户请求时，它将这个请求传递给本地文件访问例程，后者访问服务器主机上的一个本地的磁盘文件。

5) NFS服务器需要花一定的时间来处理一个客户的请求。访问本地文件系统一般也需要一部分时间。在这段时间间隔内，服务器不应该阻止其他的客户请求得到服务。为了实现这一功能，大多数的NFS服务器都是多线程的——即服务器的内核中实际上有多个NFS服务器在

运行。具体怎么实现依赖于不同的操作系统。既然大多数的 Unix内核不是多线程的，一个共同的技术就是启动一个用户进程（常被称为 nfsd）的多个实例。这个实例执行一个系统调用，使自己作为一个内核进程保留在操作系统的内核中。

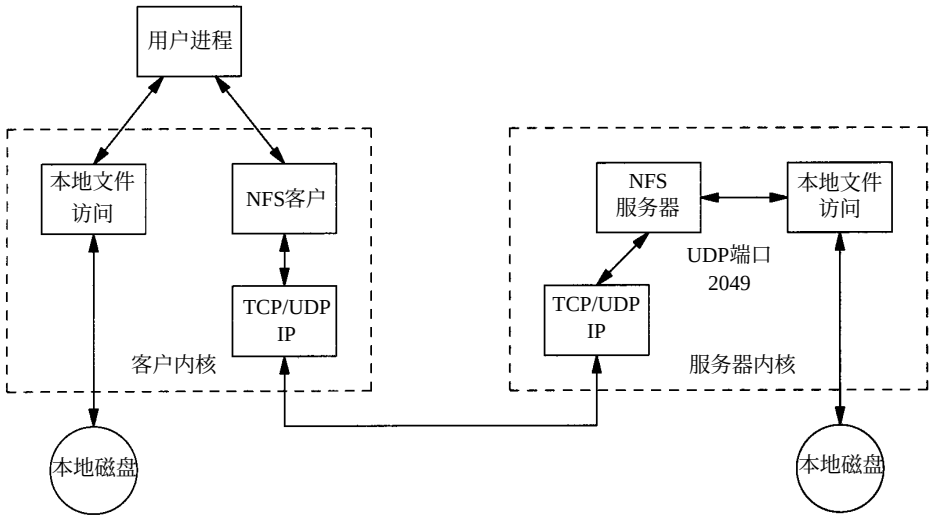


图29-3 NFS客户和NFS服务器的典型配置

6) 同样，在客户主机上， NFS客户需要花一定的时间来处理一个用户进程的请求。 NFS客户向服务器主机发出一个 RPC调用，然后等待服务器的应答。为了给使用 NFS的客户主机上的用户进程提供更多的并发性，在客户内核中一般运行着多个 NFS客户。同样，具体实现也依赖于操作系统。Unix系统经常使用类似于NFS服务器的技术：一个叫作biod的用户进程执行一个系统调用，作为一个内核进程保留在操作系统的内核中。

大多数的Unix主机可以作为一个NFS客户，一个NFS服务器，或者两者都是。大多数PC机的实现（MS-DOS）只提供了NFS客户实现。大多数的IBM大型机只提供了NFS服务器功能。

NFS实际上不仅仅由NFS协议组成。图29-4显示了NFS使用的不同RPC程序。

应用程序	程序号	版本号	过程数
端口映射器	100000	2	4
NFS	100003	2	15
安装程序	100005	1	5
加锁管理程序	100021	1, 2, 3	19
状态监视器	100024	1	6

图29-4 NFS使用的不同RPC程序

在这个图中，程序的版本是在SunOS 4.1.3中使用的。更新的实现提供了其中一些程序更新的版本。例如，Solaris 2.2还支持端口映射器的第3版和第4版，以及安装守护程序的第2版。SVR4支持第3版的端口映射器。

在客户能够访问服务器上的文件系统之前， NFS客户主机必须调用安装守护程序。我们在下面讨论安装守护程序。

加锁管理程序和状态监视器允许客户锁定一个 NFS服务器上文件的部分区域。这两个程

序独立于NFS协议，因为加锁需要知道客户和服务器的状态，而 NFS本身在服务器上是无状态的（下面我们对 NFS的无状态会介绍得更多）。[X/Open 1991] 的第9, 10和11章说明了使用加锁管理程序和状态监视器进行 NFS文件锁定的过程。

29.5.1 文件句柄

NFS中一个基本概念是文件句柄(file handle)。它是一个不透明(opaque)的对象，用来引用服务器上的一个文件或目录。不透明指的是服务器创建文件句柄，把它传递给客户，然后客户访问文件时，使用对应的文件句柄。客户不会查看文件句柄的内容 —— 它的内容只对服务器有意义。

每次一个客户进程打开一个实际上位于一个 NFS服务器上的文件时，NFS客户就会从NFS服务器那里获得该文件的一个文件句柄。每次 NFS客户为用户进程读或写文件时，文件句柄就会传给服务器以指定被访问的文件。

一般情况下，用户进程不会和文件句柄打交道 —— 只有NFS客户和NFS服务器将文件句柄传来传去。在第2版的NFS中，一个文件句柄占据32个字节，第3版中增加为64个字节。

> Unix服务器一般在文件句柄中存储下面的信息：文件系统标识符（文件系统最大和最小的设备号），i-node号（在一个文件系统中唯一的数值）和一个 i-node的生成码（每当一个i-node被一个不同的文件重用时就改变的数值）。

29.5.2 安装协议

客户必须在访问服务器上一个文件系统中的文件之前，使用安装协议安装那个文件系统。一般情况下，这是在客户主机引导时完成的。最后的结果就是客户获得服务器文件系统的一个文件句柄。

图29-5显示了一个Unix客户发出mount(8)命令所发生的情况，它说明一个NFS的安装过程。

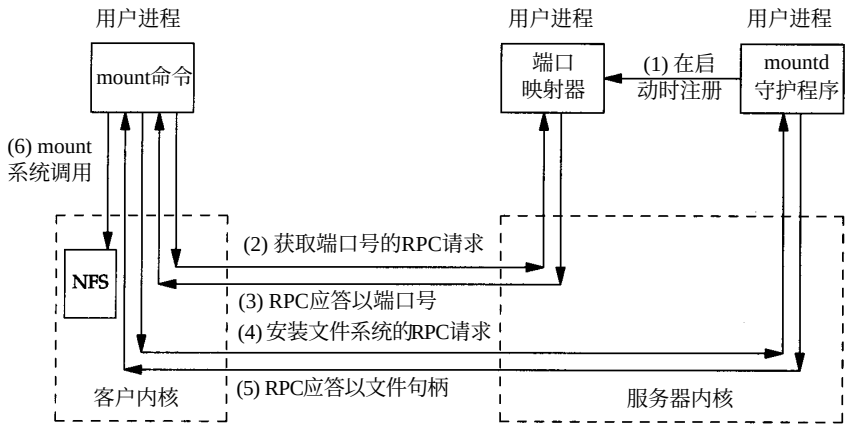


图29-5 使用Unix mount命令的安装协议

依次发生了下面的动作。

1) 服务器上的端口映射器一般在服务器主机引导时被启动。

2) 安装守护程序（mountd）在端口映射器之后被启动。它创建了一个 TCP端点和一个

UDP端点，并分别赋予一个临时的端口号。然后它在端口映射器中注册这些端口号。

3) 在客户机上执行mount命令，它向服务器上的端口映射器发出一个RPC调用来获得服务器上安装守护程序的端口号。客户和端口映射器交互既可以使用TCP也可以使用UDP，但一般使用UDP。

4) 端口映射器应答以安装守护程序的端口号。

5) mount命令向安装守护程序发出一个RPC调用来安装服务器上的一个文件系统。同样，既可以使用TCP也可以使用UDP，但一般使用UDP。服务器现在可以验证客户，使用客户的IP地址和端口号来判别是否允许客户安装指定的文件系统。

6) 安装守护程序应答以指定文件系统的文件句柄。

7) 客户机上的mount命令发出mount系统调用将第5步返回的文件句柄与客户机上的一个本地安装点联系起来。文件句柄被存储在NFS客户代码中，从现在开始，用户进程对于那个服务器文件系统的任何引用都将从使用这个文件句柄开始。

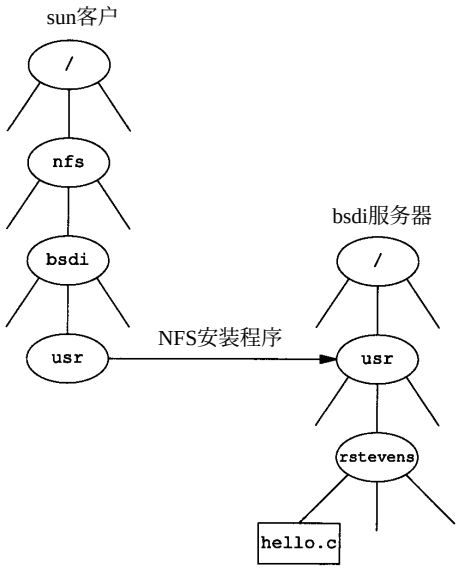


图29-6 将bsdi:/usr 目录安装成主机sun上的/nfs/bsdi/usr 目录

上述实现技术将所有的安装处理，除了客户机上的mount系统调用，都放在用户进程中，而不是放在内核中。我们显示的三个程序——mount命令、端口映射器和安装守护程序——都是用户进程。

作为一个例子，在我们的主机sun（一个NFS客户机）上执行：

```
sun  # mount -t nfs bsdi:/usr /nfs/bsdi/usr
```

这个命令将主机bsdi（一个NFS服务器）上的/usr目录安装成为本地文件系统/nfs/bsdi/usr。图29-6显示了结果。

当我们引用客户机sun上的/nfs/bsdi/usr/rstevens/hello.c文件时，实际上引用的是服务器bsdi上的文件/usr/rstevens/hello.c。

29.5.3 NFS过程

现在我们描述NFS服务器提供的15个过程（使用的个数与NFS过程的实际个数不一样，因为我们把它们按照功能分了组）。尽管NFS被设计成可以在不同的操作系统上工作，而不仅仅是Unix系统，但是一些提供Unix功能的过程可能不被其他操作系统支持（例如硬链接、符号链接、组的属主和执行权等）。[Stevens 1992]的第4章包含了Unix文件系统其他的一些信息，其中有些被NFS采用。

1) GETATTR。返回一个文件的属性：文件类型（一般文件，目录等）、访问权限、文件大小、文件的属主者及上次访问时间等信息。

2) SETATTR。设置一个文件的属性。只允许设置文件属性的一个子集：访问权限、文件

的属主、组的属主、文件大小、上次访问时间和上次修改时间。

3) STATFS。返回一个文件系统的状态：可用空间的大小、最佳传送大小等。例如 Unix的df命令使用此过程。

4) LOOKUP。查找一个文件。每当一个用户进程打开一个 NFS服务器上的一个文件时，NFS客户调用此过程。

5) READ。从一个文件中读数据。客户说明文件的句柄、读操作的开始位置和读数据的最大字节数（最多8192个字节）。

6) WRITE。对一个文件进行写操作。客户说明文件的句柄、开始位置、写数据的字节数和要写的数据。

7) CREATE。创建一个文件。

8) REMOVE。删除一个文件。

9) RENAME。重命名一个文件。

10) LINK。为一个文件构造一个硬链接。硬链接是一个 Unix的概念，指的是磁盘中的一个文件可以有任意多个目录项（即名字，也叫作硬链接）指向它。

11) SYMLINK。为一个文件创建一个符号链接。符号链接是一个包含另一个文件名字的文件。大多数引用符号链接的操作（例如，打开）实际上引用的是符号链接所指的文件。

12) READLINK。读一个符号链接。即返回符号链接所指的文件的名字。

13) MKDIR。创建一个目录。

14) RMDIR。删除一个目录。

15) READDIR。读一个目录。例如， Unix的ls命令使用此过程。

这些过程实际上有一个前缀NFSPROC_，我们把它省略了。

29.5.4 UDP还是TCP

NFS最初是用UDP写的，所有的厂商都提供了这种实现。最新的一些实现也支持 TCP。TCP支持主要用于广域网，它可以使文件操作更快。 NFS已经不再局限于局域网的使用。

当从LAN转换到WAN时，网络的动态特征变化得非常大。往返时间（round-trip time）变动范围大，拥塞经常发生。 WAN的这些特征使得我们考虑使用具有 TCP属性的算法 —— 慢启动，但是可以避免拥塞。既然 UDP没有提供任何类似的东西，那么在 NFS客户和服务器上加进同样的算法或者使用TCP。

29.5.5 TCP上的NFS

伯克利实现的Net/2 NFS支持UDP或者TCP。[Macklem 1991]描述了这个实现。让我们看一下使用TCP有什么不同。

1) 当服务器主机进行引导时，它启动一个 NFS服务器，后者被动打开 TCP端口2049，等待着客户的连接请求。这通常是另一个 NFS服务器，正常的NFS UDP服务器在UDP端口2049等待着进入的UDP数据报。

2) 当客户使用TCP安装服务器上的文件系统时，它对服务器上的 TCP端口2049做一个主动打开。这样就为这个文件系统在客户和服务器之间形成了一个 TCP连接。如果同样的客户安装同样服务器上的另一个文件系统，就会创建另一个 TCP连接。

3) 客户和服务器在它们连接的两端都要设置 TCP的keepalive选项，这样双方都能检测到对方主机崩溃，或者崩溃然后重启动。

4) 客户方所有使用这个服务器文件系统的应用程序共享这个 TCP连接。例如，在图 29-6中，如果在bsdi的/usr目录下还有另一个目录 smith，那么对两个目录 /nfs/bsdi/usr/rstevens和/nfs/bsdi/usr/smith下所有文件的引用将共享同样的 TCP连接。

5) 如果客户检测到服务器已经崩溃，或者崩溃然后重启动（通过收到一个 TCP差错“连接超时”或者“对方复位连接”），它尝试与服务器重新建立连接。客户做另一个主动打开，为同一个文件系统请求重新建立 TCP连接。在以前连接上超时的所有客户请求在新的连接上都会重新发出。

6) 如果客户机崩溃，那么当它崩溃时正在运行的应用程序也要崩溃。当客户机重新启动时，它很可能使用 TCP重新安装服务器的文件系统，这将导致和服务器的另一个连接。客户和服务器之间针对同一个文件系统的前一个连接现在打开了一半（服务器方认为它还开着），但是既然服务器设置了 keepalive选项，当服务器发出下一个 keepalive探查报文时，这个半开着的TCP连接就会被中止。

随着时间的流逝，另外一些厂商也计划支持 TCP上的NFS。

29.6 NFS实例

我们使用tcpdump来看一下在典型的文件操作中，客户调用了哪些 NFS过程。当tcpdump检测到一个包含 RPC调用（在图 29-1中调用字段等于 0）、目的端口是 2049的UDP数据报时，它把数据报按照一个 NFS请求进行解码。类似地，如果一个 UDP数据报是一个RPC应答（在图29-2中应答字段为1），源端口是2049，tcpdump就把此数据报作为一个NFS应答来解码。

29.6.1 简单的例子：读一个文件

第一个例子是使用cat(1)命令将位于一个NFS服务器上的一个文件复制到终端上：

```
sun % cat /nfs/bsdi/usr/rstevens/hello.c        把文件复制到终端
main()
{
    printf("hello, world\n");
}
```

如同图29-6所示，主机sun（NFS客户机）上的文件系统 /nfs/bsdi/usr 实际上是主机bsdi（NFS服务器）上的 /usr 文件系统。当cat打开这个文件时，sun上的内核检测到这一点，然后使用NFS去访问文件。图29-7显示了tcpdump的输出。

当tcpdump解析一个NFS请求或应答报文时，它打印客户的 XID字段，而不是端口号。第1行和第2行中的XID字段值是0x7aa6。

客户内核中的打开函数一次处理文件名 /nfs/bsdi/usr/rstevens/hello.c 中的一个成员。当处理到/nfs/bsdi/usr时，它发现这是指向一个已安装的 NFS文件系统的一个安装点。

在第1行中，客户调用 GETATTR过程取得客户已经安装的服务器目录的属性（ /usr）。这个RPC请求，除IP首部和UDP首部之外，包含 104个字节的数据。第 2行中的应答返回了一个OK值，除了IP首部和UDP首部之外，包含了 96个字节的数据。在这个图中，我们可以看出最小的NFS报文包含大约100个字节的数据。

```
1   0.0                 sun.7aa6 > bsdi.nfs: 104 getattr
2   0.003587 (0.0036)   bsdi.nfs > sun.7aa6: reply ok 96

3   0.005390 (0.0018)   sun.7aa7 > bsdi.nfs: 116 lookup "rstevens"
4   0.009570 (0.0042)   bsdi.nfs > sun.7aa7: reply ok 128

5   0.011413 (0.0018)   sun.7aa8 > bsdi.nfs: 116 lookup "hello.c"
6   0.015512 (0.0041)   bsdi.nfs > sun.7aa8: reply ok 128

7   0.018843 (0.0033)   sun.7aa9 > bsdi.nfs: 104 getattr
8   0.022377 (0.0035)   bsdi.nfs > sun.7aa9: reply ok 96

9   0.027621 (0.0052)   sun.7aaa > bsdi.nfs: 116 read 1024 bytes @ 0
10  0.032170 (0.0045)   bsdi.nfs > sun.7aaa: reply ok 140
```

图29-7 读一个文件的NFS操作

在第3行中，客户调用 LOOKUP过程来查看 rstevens文件。在第 4行中收到一个 OK应答。LOOKUP过程说明了文件名 rstevens和远程文件系统被安装时由内核保存的文件句柄。应答中包含了下一步要使用的一个新的文件句柄。

在第5行中，客户使用第4行中返回的文件句柄对 hello.c调用LOOKUP过程。在第6行返回了另一个文件句柄。新的文件句柄就是客户在第7行和第9行中引用文件/nfs/bsdi/usr/rstevens/hello.c所使用的文件句柄。我们看到客户对于正在打开的路径名的每个成员都调用了一次 LOOKUP过程。

在第7行中，客户又调用了一次 GETATTR过程，接着在第 9行中调用了 READ过程。客户请求从偏移 0开始的 1024个字节，但是接收到的没有这么多（减去 RPC字段和其他由 READ过程返回的值的大小，在第 10行中返回了 38个字节的数据。这是文件 hello.c的实际大小）。

在这个例子中，应用进程对于内核所做的这些 RPC请求和应答一点儿也不知道。应用进程只是调用了内核的 open函数，后者引起了 3个RPC请求和3个应答（1~6行），然后应用进程又调用了内核的 read函数，它引起了两个请求和两个应答（ 7~10行）。该文件位于一个 NFS文件服务器，这一点对客户应用进程来说是透明的。

29.6.2 简单的例子：创建一个目录

作为另一个简单的例子，我们将当前工作目录改变为一个 NFS服务器上的一个目录，然后创建一个新的目录：

```
sun  % cd  /nfs/bsdi/usr/rstevens          改变当前工作目录
sun  % mkdir  Mail                         并且创建一个目录
```

图29-8显示了tcpdump的输出。

```
1   0.0                 sun.7ad2 > bsdi.nfs: 104 getattr
2   0.004912 ( 0.0049)  bsdi.nfs > sun.7ad2: reply ok 96

3   0.007266 ( 0.0024)  sun.7ad3 > bsdi.nfs: 104 getattr
4   0.010846 ( 0.0036)  bsdi.nfs > sun.7ad3: reply ok 96

5   35.769875 (35.7590) sun.7ad4 > bsdi.nfs: 104 getattr
6   35.773432 ( 0.0036) bsdi.nfs > sun.7ad4: reply ok 96

7   35.775236 ( 0.0018) sun.7ad5 > bsdi.nfs: 112 lookup "Mail"
8   35.780914 ( 0.0057) bsdi.nfs > sun.7ad5: reply ok 28

9   35.782339 ( 0.0014) sun.7ad6 > bsdi.nfs: 144 mkdir "Mail"
10  35.992354 ( 0.2100) bsdi.nfs > sun.7ad6: reply ok 128
```

图29-8 NFS的操作：cd到NFS目录，然后mkdir

改变目录引起客户调用了两次GETATTR过程（1~4行）。当我们创建新的目录时，客户调用了GETATTR过程（5~6行），接着调用LOOKUP过程（7~8行，用来验证将创建的目录不存在），跟着调用了MKDIR过程来创建目录（9-10行）。在第8行中，应答OK并不表示目录存在。它只是表示过程返回了。 tcpdump并不理解NFS过程的返回值。它一般打印 OK和应答报文中数据的字节数。

29.6.3 无状态

NFS的一个特征（NFS的批评者称之为NFS的一个瑕疵，而不是一个特征）是 NFS服务器是无状态的(stateless)。服务器并不记录哪个客户正在访问哪个文件。请注意一下在前面给出的NFS过程中，没有一个 open操作和一个close操作。LOOKUP过程的功能与open操作有些类似，但是服务器永远也不会知道客户对一个文件调用了 LOOKUP过程之后是否会引用该文件。

无状态设计的理由是为了在服务器崩溃并且重启动时，简化服务器的崩溃恢复操作。

29.6.4 例子：服务器崩溃

在下面的例子中我们从一个崩溃然后重启动的 NFS服务器上读一个文件。这个例子演示了无状态的服务器是如何使得客户不知道服务器的崩溃。除了在服务器崩溃然后重启动时一个时间上的暂停外，客户并不知道发生的问题，客户应用进程没有受到影响。

在客户机sun上，我们对一个长文件（NFS服务器主机svr4上的文件/usr/share/lib/termcap）执行cat命令。在传送过程中把以太网的网线拔掉，关闭然后重启动服务器主机，再重新将网线连上。客户被配置成每个 NFS read过程读1024个字节。图29-9显示了tcpdump的输出。

1~10行对应于客户打开文件，操作类似于图 29-7所示。在第11行我们看到对文件的第一个READ操作，在12行返回了1024个字节的数据。这个操作一直继续到 129行（读1024个字节的数据，跟着一个OK应答）。

在第130行和第131行我们看到两个请求超时，并且分别在 132行和133行重传。第一个问题是这里为什么会有两个读请求，一个从偏移 65536开始读，另一个从偏移 73728开始读？答案是客户内核检测到客户应用进程正在进行顺序地读操作，所以试图预先取得数据块（大多数的Unix内核都采用了这种预读技术）。客户内核也正在运行多个 NFS块I/O守护程序，后者试图代表客户产生多个RPC请求。一个守护程序正在从偏移 65536处读8192个字节（以1024字节为一组数据块），而另一个正在从73728处预读8192个字节。

客户重传发生在130~168行。在第169行我们看到服务器已经重启动，在它对第 168行的客户NFS请求做出应答之前，它发送了一个 ARP请求。对168行的响应被发送在 171行。客户的READ操作继续进行下去。

除了从129行到171行5分钟的暂停，客户应用进程并不知道服务器崩溃然后又重启动了。这个服务器的崩溃对于客户是透明的。

为了研究这个例子中的超时和重传时间间隔，首先要意识到这儿有两个客户守护程序，分别有它们各自的超时。第 1个守护程序（在偏移65536处开始读）的间隔，四舍五入到两个十进制小数点，为0.68, 0.87, 1.74, 3.48, 6.96, 13.92, 20.0, 20.0, 20.0等等。第2个守护程序（在偏移73728处开始读）的间隔也是一样的（精确到两个小数点）。可以看出这些NFS客户使用了一个这样的超时定时器：间隔为0.875秒的倍数，上限为20秒。每次超时后，重传间隔翻倍：

0.875, 1.75, 3.5, 7.0和14.0。

```
 1    0.0                sun.7ade > svr4.nfs: 104 getattr
 2    0.007653 ( 0.0077) svr4.nfs > sun.7ade: reply ok 96

 3    0.009041 ( 0.0014) sun.7adf > svr4.nfs: 116 lookup "share"
 4    0.017237 ( 0.0082) svr4.nfs > sun.7adf: reply ok 128

 5    0.018518 ( 0.0013) sun.7ae0 > svr4.nfs: 112 lookup "lib"
 6    0.026802 ( 0.0083) svr4.nfs > sun.7ae0: reply ok 128

 7    0.028096 ( 0.0013) sun.7ae1 > svr4.nfs: 116 lookup "termcap"
 8    0.036434 ( 0.0083) svr4.nfs > sun.7ae1: reply ok 128

 9    0.038060 ( 0.0016) sun.7ae2 > svr4.nfs: 104 getattr
10    0.045821 ( 0.0078) svr4.nfs > sun.7ae2: reply ok 96

11    0.050984 ( 0.0052) sun.7ae3 > svr4.nfs: 116 read 1024 bytes @ 0
12    0.084995 ( 0.0340) svr4.nfs > sun.7ae3: reply ok 1124
```

连续地读

```
128    3.430313 ( 0.0013) sun.7b22 > svr4.nfs: 116 read 1024 bytes @ 64512
129    3.441828 ( 0.0115) svr4.nfs > sun.7b22: reply ok 1124

130    4.125031 ( 0.6832) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
131    4.868593 ( 0.7436) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

132    4.993021 ( 0.1244) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
133    5.732217 ( 0.7392) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

134    6.732084 ( 0.9999) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
135    7.472098 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

136   10.211964 ( 2.7399) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
137   10.951960 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

138   17.171767 ( 6.2198) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
139   17.911762 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

140   31.092136 (13.1804) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
141   31.831432 ( 0.7393) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

142   51.090854 (19.2594) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
143   51.830939 ( 0.7401) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

144   71.090305 (19.2594) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
145   71.830155 ( 0.7398) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
```

连续重传

```
167 291.824285 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
168 311.083676 (19.2594) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
```

服务器重启动

```
169 311.149476 ( 0.0658) arp who-has sun tell svr4
170 311.150004 ( 0.0005) arp reply sun is-at 8:0:20:3:f6:42

171 311.154852 ( 0.0048) svr4.nfs > sun.7b23: reply ok 1124

172 311.156671 ( 0.0018) sun.7b25 > svr4.nfs: 116 read 1024 bytes @ 66560
173 311.168926 ( 0.0123) svr4.nfs > sun.7b25: reply ok 1124
```

连续读

图29-9 当一个NFS服务器崩溃然后重启动时，客户正在读一个文件的过程

客户要重传多久呢？客户有两个与此有关的选项。首先，如果服务器文件系统是“硬”安装的，客户就会永远重传下去。但是如果服务器文件系统是“软”安装的，客户重传了固定数目的次数之后就会放弃。在“硬”安装的情况下，客户还有一个选项决定是否允许用户中断无限制的重传。如果客户主机安装服务器文件系统时说明了中断能力，并且如果我们不想在服务器崩溃之后等5分钟，等着服务器重启动，就可以键入一个中断键以终止客户应用

程序。

29.6.5 等幂过程

如果一个RPC过程被服务器执行多次仍然返回同样的结果，那么就把它叫作等幂过程（Idempotent Procedure）。例如，NFS的读过程是等幂的。正像我们在图29-9中看到的，客户只是重发一个特定的READ调用直到它得到一个响应。在我们的例子中，重传的原因是服务器崩溃了。如果服务器没有崩溃，而是RPC应答报文丢失了（既然UDP是不可靠的），客户只是重传请求，服务器再一次执行同样的READ过程。同一个文件的同一部分被重读一次，发送给客户。

这种方法行得通的原因在于每个READ请求指出了读操作开始的偏移位置。如果有一个NFS过程要求服务器读一个文件的下*N*个字节，这种方法就不行了。除非服务器被做成是有状态的（与无状态相反），如果一个应答丢失了，客户重发读下*N*个字节的READ请求，结果将是不一样的。这就是为什么NFS的READ和WRITE过程要求客户说明开始的偏移位置的原因。客户维护着状态（每个文件当前的偏移位置），而不是服务器。

不幸的是并不是所有的文件系统操作都是等幂的。例如，考虑下面的动作：客户NFS发出REMOVE请求来删除一个文件；服务器NFS删除了文件，并回答OK；服务器的回答丢失了；客户NFS超时，然后重传请求；服务器NFS找不到指定的文件，回答指出一个错误；客户应用程序接收到一个错误表示文件不存在。这个返回给客户应用程序的错误是不对的——该文件的确存在并且被删除了。

等幂的NFS过程是：GETATTR、STATES、LOOKUP、READ、WRITE、READLINK和READDIR。不是等幂的过程是：CREATE、REMOVE、RENAME、LINK、SYMLINK、MKDIR和RMDIR。SETATTR过程如果不用来截断文件，一般是等幂的。

既然使用UDP总会发生响应报文丢失的现象，NFS服务器需要一种方法来处理非等幂的操作。大多数的服务器实现了一个最近应答的高速缓存，用于存放非等幂操作最近的应答。每当服务器收到一个请求，它首先检查这个高速缓存，如果找到了一个匹配，就返回以前的应答而不再调用相应的NFS过程。[Juszczak 1989]提供了这种高速缓存的实现细节。

等幂服务器过程的概念可以应用于任何基于UDP的应用程序，而不仅仅是NFS。例如，DNS也提供了一个等幂服务。一个DNS的服务器可以任意多次地执行一个解析者的请求而没有任何不良的后果（如果不考虑网络资源浪费的话）。

29.7 第3版的NFS

1993年发布了第3版的NFS协议规范[Sun Microsystem 1994]。其实现有望在1994年成为可能。

我们总结一下第2版和第3版的主要区别。下面把两者分别称为V2和V3。

1) V2中的文件句柄是32字节的固定大小的数组。在V3中，它变成了一个最多为64个字节的可变长度的数组。在XDR中，一个可变长度的数组被编码为一个4字节的数组成员个数跟着实际的数组成员字节。这样在实现时减少了文件句柄的长度，例如Unix只需要12个字节，但又允许非Unix实现维护另外的信息。

2) V2将每个READ和WRITE RPC过程可以读写的数据限制为8192个字节。这个限制在V3

中取消了，这就意味着一个UDP上的实现只受到IP数据报大小的限制（65535字节）。这样允许在更快的网络上读写更大的分组。

3) 文件大小以及READ和WRITE过程开始偏移的字节从32字节扩充到64字节，允许读写更大的文件。

4) 每个影响文件属性值的调用都返回文件的属性。这样减少了客户调用GETATTR过程的次数。

5) WRITE过程可以是异步的，而在V2中要求同步的WRITE过程。这样可以提高WRITE过程的性能。

6) V3中删去了一个过程（STATFS），增加了七个过程：ACCESS（检查文件访问权限）、MKNOD（创建一个Unix特殊文件）、READDIRPLUS（返回一个目录中的文件名字和它们的属性）、FSINFO（返回一个文件系统的静态信息）、FSSTAT（返回一个文件系统的动态信息）、PATHCONF（返回一个文件的POSIX.1信息）和COMMIT（将以前的异步写操作提交到外存中）。

29.8 小结

RPC是构造客户-服务器应用程序的一种方式，使得看起来客户只是调用了服务器的过程。所有的网络操作细节都被隐藏在RPC程序包为一个应用程序生成的客户和服务器残桩以及RPC库的例程中。我们显示了RPC调用和应答报文的格式，并且提到了使用XDR对传输的值进行编码，使得RPC客户和服务器可以运行在不同架构的机器上。

最广泛使用的RPC应用之一就是Sun的NFS，一个在各种大小的主机上广泛实现的异构的文件访问协议。我们浏览了NFS和它使用UDP和TCP的方式。第2版的NFS协议定义了15个过程。

一个客户对一个NFS服务器的访问开始于安装协议，返回给客户一个文件句柄。客户接着可以使用那个文件句柄来访问服务器文件系统中的文件。在服务器上，一次检查文件名的一个成员，返回每个成员的一个新的文件句柄。最后的结果就是要引用的文件的一个文件句柄，它可以在随后的读写操作中被使用。

NFS试图把它的所用过程都做成等幂的，使得如果响应报文丢失了，客户只需要重发一个请求。我们看到了服务器崩溃然后又重启动时，一个客户读服务器上的一个文件的例子。

习题

29.1 在图29-7中，我们看到tcpdump将分组理解为NFS的请求和应答，打印了XID。tcpdump可以为任何的RPC请求或者应答这样做吗？

29.2 在一个Unix系统中，你认为为什么RPC服务器程序使用的是临时端口，而不是知名端口？

29.3 一个RPC客户调用了两个服务器过程。第1个服务器过程执行花了5秒钟的时间，第二个过程花了1秒钟。客户有一个4秒钟的超时。画出客户与服务器之间在时间轴上交互的信息（假定信息从客户传到服务器或者相反都不花时间）。

29.4 在图29-9的例子中，如果NFS服务器关机时，把它的以太网卡给换掉了，将会发生什么事情？

29.5 在图29-9中，当服务器重启动后，它处理了从偏移 65536开始的请求（168行和171行），然后处理了从偏移66560开始的下一个请求（172行和173行）。对于从偏移73728开始的请求怎么处理的呢？（167行）

29.6 当描述等幂NFS过程时，我们给出了一个REMOVE应答在网络中丢失的例子。在这种情况下，如果使用的是TCP而不是UDP会怎么样呢？

29.7 如果NFS服务器使用的是一个临时端口而不是 2049，那么当服务器崩溃然后又重启动时，一个NFS客户会发生什么情况呢？

29.8 每个主机最多只有1023个保留端口，所以保留端口是很缺乏的（ 1.9节）。如果一个NFS服务器要求它的客户拥有保留端口（公共的端口），一个NFS客户使用TCP安装了*N*个不同的服务器上的*N*个文件系统，那么客户对每个连接都需要一个不同的保留端口号吗？

第30章　其他的TCP/IP应用程序

30.1　引言

本章中我们描述了另外一些很多实现都支持的 TCP/IP应用程序。有些很简单，易于全面了解（Finger和Whois），而另一个则相当复杂（ X窗口系统）。我们只提供了这个复杂应用程序的一个简短的概述，集中介绍其对 TCP/IP协议的使用。

另外，我们提供一些 Internet上资源发现工具的概述。包括一组在 Internet上导航的工具，可以帮助寻找一些我们不知道确切位置和名字的信息。

30.2　Finger协议

Finger协议返回一个指定主机上一个或多个用户的信息。它常被用来检查某个人是否登录了，或者搞清一个人的登录名以便给他发送邮件。 RFC1288 [Zimmerman 1991] 指明了这个协议。

由于两个原因，很多站点不支持一个 Finger服务器。第一， Finger服务器的一个早期版本中的一个编程错误被 1988年声名狼藉的 Internet蠕虫病毒利用，作为进入点之一（ RFC1135 [Reynolds 1989] 和 [Curry 1992] 更详细地描述了蠕虫）。第二， Finger协议有可能会泄露一些很多管理员认为是有关用户的私有信息（登录名、电话号码，他们上次的登录时间，等等）。RFC1288的第3节给出了这个有关服务安全方面的细节。

从一个协议的角度来看， Finger服务器有一个知名的端口 79。客户对这个端口做一个主动打开，然后发送一个在线的请求。服务器处理这个请求，把输出发送回去，然后关闭连接。查询和响应都是采用NVT ASCII，类似于我们在FTP和SMTP协议中所看到的。

尽管大多数的Unix用户都是使用finger (1)客户来访问Finger服务器，我们将从使用 Telnet客户与Finger服务器直接相连开始，看看客户发出的每一条在线命令。如果客户的查询是一个空行（在NVT ASCII中，空行以一个回车符 CR跟着一个换行符 LF来传输），它就是一个请求查询所有在线用户信息的命令。

```
sun % telnet slip finger
Trying 140.252.13.65 ...
Connected to slip.
Escape character is '^]'.

Login    Name               Tty  Idle  Login Time   Office     Office Phone
rstevens Richard Stevens    *co    45  Jul 31 09:13
rstevens Richard Stevens    *c2    45  Aug  5 09:41
Connection closed by foreign host.
```

Telnet客户输出前三行

这儿我们键入回车作为Finger客户的命令

Telnet客户的输出

office和office phone的空白输出字段是从用户的口令 (password)文件记录的选项字段中取出的（在这个例子中，这两个字段的值没有提供）。

服务器必须在最后做一个主动的关闭操作，因为服务器返回的是一个可变长度的信息。

当客户收到文件结束字符时，就知道服务器的输出结束了。

当客户的请求由一个用户名组成时，服务器只以该用户的信息作为响应。下面是另一个例子，这个例子中删去了Telnet客户的输出：

```
sun % telnet vangogh.cs.berkeley.edu finger
rstevens                              这是我们键入的客户请求
Login: rstevens                       Name: Richard Stevens
Directory: /a/guest/rstevens          Shell: /bin/csh
Last login Thu Aug  5 09:55 (PDT) on ttyq2 from sun.tuc.noao.edu
Mail forwarded to: rstevens@noao.edu
No Plan.
```

当一个系统完全禁止了Finger服务时，因为没有进程被动打开端口79，所以客户的主动打开将从服务器接收到一个RST。

```
sun % finger @svr4
[svr4.tuc.noao.edu] connect: Connection refused
```

一些站点在端口79提供了一个服务器，但服务器只是向客户输出信息，而不理睬客户的任何请求：

```
sun % finger @att.com
[att.com]                  这一行是Finger客户输出的；其余行是服务器输出的
-------------------------------------------------------------
There are no user accounts on the AT&T Internet gateway.
To send email to an AT&T employee, send email to their name
separated by periods at att.com. If the employee has an email
address registered in the employee database, they will receive
email - otherwise, you'll receive a non-delivery notice.
For example: John.Q.Public@att.com
sun % finger clinton@whitehouse.gov
[whitehouse.gov]

        Finger service for arbitrary addresses on whitehouse.gov is not
supported. If you wish to send electronic mail, valid addresses are
"PRESIDENT@WHITEHOUSE.GOV", and "VICE-PRESIDENT@WHITEHOUSE.GOV".
```

对一个组织来说，另一种可能就是实现一个防火墙网关：在组织内部和Internet之间的一个路由器，负责过滤（也就是扔掉）特定的IP数据报（[Cheswick and Bellovin 1994]详细讨论了防火墙网关）。防火墙网关可以被配置成扔掉从Internet进来的这样一些数据报，这些数据报是目的端口为79的TCP报文段。

对于Finger的服务器和Unix的Finger客户还有其他的实现。欲知详情，请参考RFC1288和有关finger(1)的手册。

RFC1288指出提供了Finger服务器的、具有TCP/IP连接的自动售货机应该对客户的空行请求响应以现有产品的列表。对于由一个名字组成的客户请求，它们应该响应以一个数目或者与这个产品有关的可用项的列表。

30.3 Whois协议

Whois协议是另一种信息服务。尽管任何站点都可以提供一个Whois服务器，在InterNIC站点（rs.internic.net）的服务器是最常使用的。这个服务器维护着所有的DNS域和很多连接在Internet上的系统的系统管理员的信息（另一个可用的服务器在nic.ddn.mil，不过只包含了有关MILNET的信息）。不幸的是信息有可能是过期的或不完整的。RFC954 [Harrenstein, Stahl,

and Feinler 1985] 说明了Whois服务。

从协议的角度来看，Whois服务器有一个知名的TCP端口43。它接受客户的连接请求，客户向服务器发送一个在线的查询。服务器响应以任何可用的信息，然后关闭连接。请求和应答都以NVT ASCII来传输。除了请求和应答所包含的信息不一样，Whois服务器和Finger服务器几乎是一样的。

最常用的Unix客户程序是whois(1) 程序，尽管我们可以使用Telent自己手工键入命令。开始的命令是只包含一个问号的请求，服务器会返回所支持的客户请求的具体信息。

当NIC在1993年改变为InterNIC时，Whois服务器的站点也从 nic.ddn.mil移到了rs.internic.net。很多厂商仍然装载了采用nic.ddn.mil版本的whois客户程序。为了和正确的服务器联系上，你可能需要指明命令行参数 -h rs.internic.net。

另外，我们可以使用Telnet登录rs.internic.net站点，登录名采用whois。

我们将使用Whois服务器来查询一下本书的作者（已经删去了无关的 Telnet客户输出）。第一个请求是查询所有匹配“ stevens ”的名字。

```
sun % telnet rs.internic.net whois
stevens                                    这是我们键入的客户命令

                                           我们省略了其他25个“ stevens ”的信息
Stevens, W. Richard (WRS28)    stevens@kohala.com      +1 602 297 9416

The InterNIC Registration Services Host ONLY contains Internet
Information (Networks, ASN's, Domains, and POC's).
Please use the whois server at nic.ddn.mil for MILNET Information.
```

名字后面的括号中的三个大写字母跟着一个数字，（WRS28），是个人的NI句柄。下一个查询包含一个感叹号和一个NIC句柄，用于获得有关这个人的进一步信息。

```
sun % telnet rs.internic.net whois
!wrs28                                     我们键入的客户请求
Stevens, W. Richard (WRS28)                   stevens@kohala.com
   Kohala Software
   1202 E. Paseo del Zorro
   Tucson, AZ 85718
   +1 602 297 9416

   Record last updated on 11-Jan-91.
```

很多有关Internet变量的其他信息也可以查找。例如，请求 net 140.252将返回有关B类地址140.252的信息。

白页

使用SMTP的VRFY命令、Finger协议以及Whois协议在Internet上查找用户类似于使用电话号码簿的白页查找一个人的电话号码。在目前阶段，诸如上述的工具已经广泛可用了，为了提高这种服务的研究正在进行当中。

[Schwartz and Tsirigotis 1991] 包含了正在Internet上试验的不同白页服务的其他信息。一个叫作Netfind的特别工具可以通过使用 Telent，以netfind登录到bruno.cs.colorado或者ds.internic.net站点来访问。

RFC1309 [Weider, Reynolds, and Heker 1992]提供了对OSI目录服务X.500的概述，并且比较了它与当前的Internet技术（Finger和Whois）的相同点和不同点。

30.4 Archie、WAIS、Gopher、Veronica和WWW

前两节我们讨论的工具——Finger、Whois和一个白页服务——是用来查找人的信息的。还有一些工具是用来定位文件和文档的，本节中对这些工具给出了一个概述。我们只提供了一个概述，因为对每一个工具的细节的研究超出了本书的范围。我们给出了在Internet上找到这些工具的方法，鼓励你去试一试，找找看哪些工具可以帮助你。还有一些其他的工具正在被开发。[Obraczka, Danzig, and Li1993] 概述了在Internet上的资源发现服务。

30.4.1 Archie

本书中使用的很多资源都是使用匿名FTP得到的。问题是如何找到有我们想要的程序的FTP站点。有时候我们甚至不知道精确的文件名，但知道几个很可能在文件名中出现的关键字。

Archie提供了Internet上几千个FTP服务器的目录。我们可以通过登录进一个Archie服务器，搜索那些名字中包含了一个指定的常规表达式的文件。输出是一个与文件名匹配的FTP服务器的列表。然后我们可以使用匿名FTP去那个站点取得想要的文件。

全世界有很多Archie服务器。一个比较好的开始点是使用Telnet以archie名字登录进ds.internic.net，然后执行命令servers。这个命令的输出提供了所有Archie服务器以及它们的地址的一个列表。

30.4.2 WAIS

Archie帮助我们查找名字中包含关键字的文件，但有时候我们需要查找包含一个关键字的文件或数据库。即，想查找一个内容中包含一个关键字的文件，而不是文件名字中包含关键字。

WAIS (Wide Area Information Servers广域信息服务系统)知道几百个包含了有关计算机主题的和其他一般性主题信息的数据库。为了使用WAIS，我们要选择需要查找的数据库，指明关键字。尝试WAIS服务请使用Telnet，以wais名字登录quake.think.com站点。

30.4.3 Gopher

Gopher是其他Internet资源服务如Archie、WAIS和匿名FTP的一个菜单驱动的前端程序。Gopher是最容易使用的工具之一，因为不管它使用了哪个资源服务，它的用户界面都是一样的。

为了尝试Gopher，请使用Telnet，以gopher名字登录is.internic.net站点。

30.4.4 Veronica

就像Archie是一个匿名FTP服务器的索引一样，Veronica(Very Easy Rodent-Oriented Netwide Index to Computerized Archives)是一个Gopher标题的索引。一次Veronica搜索一般要查找几百个Gopher服务器。

我们必须通过一个Gopher客户来访问Veronica服务。选择Gopher的菜单项“Beyond InterNIC: Virtual Treasures of the Internet”，然后在下一个菜单中选择Veronica。

30.4.5 万维网WWW

万维网使用一个称为超文本的工具，使得我们可以浏览一个大的 /全球范围的服务和文档。信息和关键字一起显示，不过关键字被突出显示[⊖]。我们可以通过选择关键字得到更多的信息。

为了访问WWW，请使用Telnet登录info.cern.ch站点。

30.5 X窗口系统

X窗口系统(X Window System)，或简称为X，是一种客户-服务器应用程序。它可以使得多个客户（应用）使用由一个服务器管理的位映射显示器。服务器是一个软件，用来管理显示器、键盘和鼠标。客户是一个应用程序，它与服务器在同一台主机上或者在不同的主机上。在后一种情况下，客户与服务器之间通信的通用形式是 TCP，尽管也可以使用诸如 DECNET 的其他协议。在有些场合，服务器是与其他主机上客户通信的一个专门的硬件（一个 X终端）。在另一种场合，一个独立的工作站，客户与服务器位于同一台主机，使用那台主机上的进程间通信机制进行通信，而根本不涉及任何网络操作。在这两种极端情况之间，是一台既支持同一台主机上的客户又支持不同主机上的客户的工作站。

X需要一个诸如TCP的、可靠的、双向的流协议（ X不是为不可靠协议，如 UDP，而设计的）。客户与服务器的通信是由在连接上交换的 8 bit字节组成的。[Nye 1992] 给出了客户与服务器在它们的TCP连接上交换的150多个报文的格式。

在一个Unix系统中，当X客户和X服务器在同一台主机上时，一般使用 Unix系统的本地协议，而不使用 TCP协议，因为这样比使用 TCP的情况减少了协议处理时间。 Unix系统的本地协议是同一台主机上的客户和服务器之间可以使用的一种进程间通信的形式。回忆一下在图 2-4中，当使用TCP作为同一台主机上进程间的通信方式时，在 IP层以下发生了这个数据的环回(loopback)，隐含着所有的TCP和IP处理都发生了。

图30-1显示了三个客户使用一个显示器的可能的脚本。一个客户与服务器在同一台主机上，使用Unix系统的本地协议。另外两个位于不同的主机上，使用 TCP。一般来说，其中一个客户是一个窗口管理程序 (window manager)，它有权限管理显示器上窗口的布局。例如，窗口管理程序允许我们在屏幕上移动窗口，或者改变窗口的大小。

在这里客户和服务器这两个词猛一看含义相反了。对于 Telnet和FTP的应用，我们把客户看作是在键盘和显示器上的交互式用户。但是对于 X，键盘和显示器是属于服务器的。服务器被认为是提供服务的一方。 X提供的服务是对窗口、键盘和鼠标的访问。对于 Telnet，服务是登录远程的主机。对于FTP，服务是服务器上的文件系统。

当X终端或工作站引导时，一般启动 X服务器。服务器创建一个 TCP端点，在端口6000 + *n*上做一个被动打开，其中 *n*是显示器号（一般是 0）。大多数的 Unix服务器也使用名字 /tmp/.X11-unix/X*n*创建一个Unix系统的插口，其中*n*还是显示器的号。

当一个客户在另一台主机上启动时，它创建一个 TCP端点，对服务器上的端口 6000+*n*做一个主动打开。每个客户都得到了一个自己与服务器的连接。服务器负责对所有的客户请求进行复用。从这点开始，客户通过 TCP连接向服务器发送请求（例如，创建一个窗口），服务

⊖ 例如通过使用不同的颜色 —— 译者注。

器返回应答，服务器也发送事件给客户（鼠标按钮按下，键盘键按下，窗口暴露，窗口大小改变，等等）。

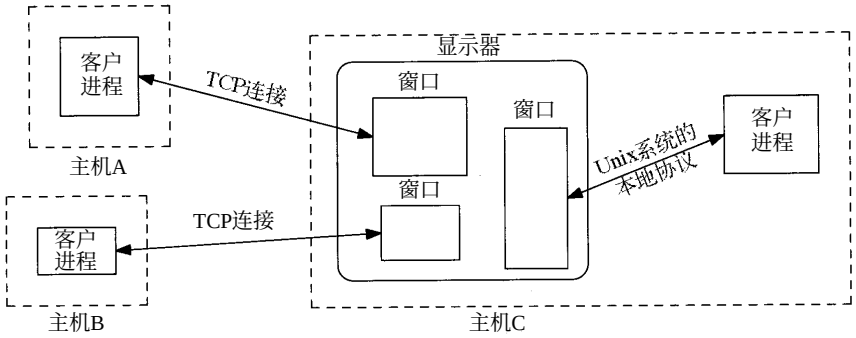


图30-1 使用一个显示器的三个X客户

图30-2将图30-1重新画，但强调了客户与X服务器进程间的通信，X服务器进程轮流管理着每个窗口。图中没有显示的是X服务器管理键盘和鼠标。

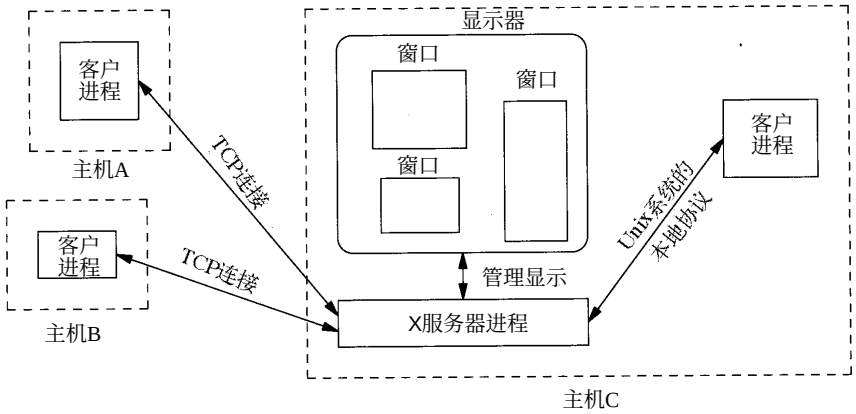


图30-2 使用一个显示器的三个客户

单个服务器处理多个客户请求的这种设计与我们在18.11节描述的正常的TCP并发服务器设计不同。例如，每次一个新的TCP连接请求到达，FTP和Telnet服务器都会产生一个新的进程，因此，每个客户都和一个不同的服务器进程通信。然而，对于X，运行在同一台主机或者在不同主机上的所有客户都和同一个服务器通信。

通过X客户和它的服务器之间的TCP连接可以交换很多数据。传输数据的数目依赖于特定的应用程序设计。例如，如果我们运行Xclock客户，Xclock在服务器的一个窗口中显示客户机当前的时间和日期。如果我们指定每隔1秒修改一次时间，那么每隔1秒，就会有一个X报文通过TCP连接从客户传输到服务器。如果我们运行X终端模拟程序，Xterm，我们敲的每一个键都会变成一个32字节的X报文（加上标准的IP和TCP首部就是72字节），在相反方向上的回送字符将是一个更大的X报文。[Droms and Dyksen 1990]检查了不同的X客户与一个特定的服务器之间的TCP流量。

30.5.1 Xscope程序

Xscope是检查X客户与它的服务器之间交换的信息的一个方便的程序。大多数的X窗口实

现都提供这个程序。它处在客户与服务器之间，双向传输所有的数据，同时解析所有的客户请求和服务器应答。图30-3显示了这种设置。

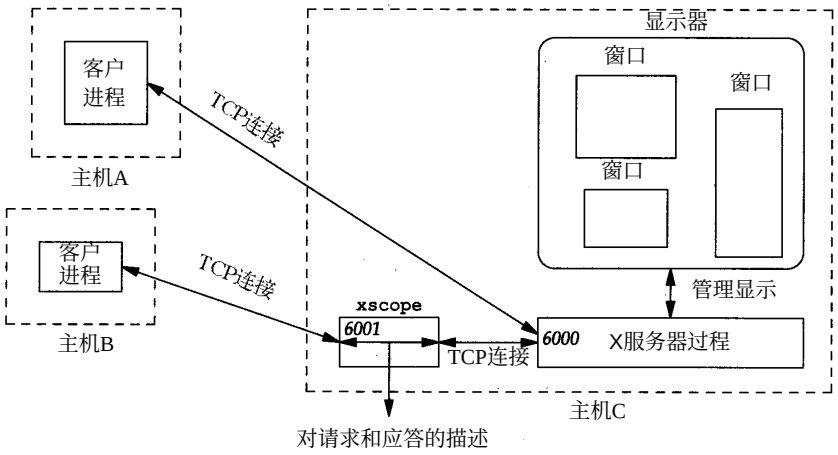


图30-3 使用xscope 监视一个X连接

首先，我们在服务器所在的主机上启动 xscope进程，但是xscope不是在端口6000而是在端口6001上监听TCP的连接请求。然后我们在另一台主机上启动一个客户，指明显示器号为1，而不是0，使得客户与xscope相连，而不直接与服务器相连。当客户的连接请求到达时，xscope创建与端口6000上的真正的服务器的一个 TCP连接，在客户与服务器之间复制所有的数据，同时生成请求与应答的一个可读的描述。

我们将在sun主机上启动xscope，然后在主机svr4上运行xclock客户。

```
svr4  %  DISPLAY=sun:1 xclock  -digital  -update 5
```

这条命令在主机sun的一个窗口中以数字形式显示时间和日期。我们指明了一个每 5秒的更新时间。

```
Thu  Sep  9  10:32:55  1993
```

我们对xscope指明一个 - q选项以产生最小的输出。为了看到每个报文的所有字段，可以使用不同的冗长级别。下面的输出显示了前三个请求和应答。

```
sun % xscope -q
0.00: Client --> 12 bytes
0.02:                                152 bytes <-- X11 Server
0.03: Client --> 48 bytes
        ............REQUEST: CreateGC
        ............REQUEST: GetProperty
0.20:                                396 bytes <-- X11 Server
                                     ..............REPLY: GetProperty
0.30: Client -->  8 bytes
0.38: Client --> 20 bytes
        ............REQUEST: InternAtom
0.43:                                 32 bytes <-- X11 Server
                                     ..............REPLY: InternAtom
```

客户的第1个在时刻0.00的报文和服务器在时刻 0.02的响应是客户与服务器之间标准的连接建立过程。客户标识它的字节顺序以及它希望的服务器版本。服务器响应以有关自己的不

同的信息。

下一个在时刻0.03的报文包含了两个客户请求。第1个请求在服务器上创建一个客户可以在其中画的图形上下文。第2个请求从服务器上得到一个属性（RESOURCE_MANAGER属性）。属性可以用于客户之间的通信，经常是在一个应用程序和窗口管理程序之间。服务器在时刻0.20的应答包含了这个属性。

下面两个在时刻0.30和0.38的客户报文形成了返回一个原子的单个请求（每个属性具有一个唯一的整型标识符称为原子）。服务器在时刻0.43的应答包含了这个原子。

如果不提供有关X窗口系统更多的细节是不可能进一步理解这个例子的，但这又不是本节的目的。在这个例子中，在窗口被显示之前，客户总共发送了1668个字节组成的12个报文段，服务器总共发送了1120个字节组成的10个报文段。耗费的时间为3.17秒。从这以后客户每5秒发送一个平均44个字节的小请求，请求更新窗口。这样一直持续到客户被终止。

30.5.2 LBX：低带宽X

为了将X用于局域网，对X协议使用的编码进行了优化，因为在局域网中花在对数据进行编码和解码的时间比最小化传输的数据更重要。尽管这种推断对以太网是适用的，但对于低速的串行线，如SLIP和PPP链路，就存在问题了（2.4节和2.6节）。

定义一个称为低带宽X（LBX）的标准的工作正在进行当中，它使用了下面的技术来减少网络流量的数目：快速缓存、只发送与前面分组的不同部分以及压缩技术。标准的规范和在第6版的X窗口系统中的一个样本实现应该会在1994年的早些时候完成。

30.6 小结

我们介绍的前两个应用，Finger和Whois，是用来获得用户信息的。Finger客户查询一个服务器，经常是为了找到某个人的登录名（以便给他们发电子邮件），或者去看一下某个人是否登录了。Whois客户一般与InterNIC运行的服务器联系，查找关于一个人、机构、域或网络号的信息。

我们简单描述了其他一些Internet资源发现服务：Archie、WAIS、Gopher、Veronica和WWW，帮助我们在Internet上定位文件和文档。还有一些资源发现工具正在被开发。

本章的最后简单浏览了另一个TCP/IP的重要客户程序，X窗口系统。我们看到X服务器管理一个显示器上的多个窗口，处理客户与其窗口的通信。每个客户都有它自己的与服务器的TCP连接，一个单个的服务器为一个给定的显示器管理着所有的客户。通过Xscope程序，我们看到怎样把一个程序放在一个客户与服务器之间，输出有关两者之间交换的报文的信息。

习题

30.1 试用Whois找到网络号为88的A类网络的拥有者。

30.2 试用Whois找到管理`whitehouse.gov`域的DNS服务器。这个应答与DNS给出的答案相匹配吗？

30.3 在图30-3中，你认为`xscope`进程必须和X服务器运行在同一台主机上吗？

附录A　tcpdump程序

tcpdump程序是由Van Jacobson、Craig Leres和Steven McCanne编写的，他们都来自加利福尼亚大学伯克利分校的劳伦斯伯克利实验室。本书中使用的是 2.2.1版（1992年6月）。

tcpdump通过将网络接口卡设置为混杂模式（ promiscuous mode）来截获经过网络接口的每一个分组。正常情况下，用于诸如以太网媒体的接口卡只截获送往特定接口地址或广播地址的链路层的帧（2.2节）。

底层的操作系统必须允许将一个接口设置成混杂模式，并且允许一个用户进程截获帧。下列的操作系统可以支持 tcpdump，或者可以加入对 tcpdump的支持：4.4BSD、BSD/386、SunOS、Ultrix和HP-UX。参考一下随着 tcpdump发布的README文件，了解它支持哪些操作系统以及哪些版本。

除了tcpdump还有其他一些选择。在图10-8中，我们使用了Solaris 2.2的程序snoop来查看一些分组。AIX 3.2.2提供了iptrace程序，该程序也提供了类似的功能。

A.1　BSD 分组过滤器

当前由BSD演变而来的 Unix内核提供了 BSD 分组过滤器 BPF (BSD Packet Filter)，tcpdump用它来截获和过滤来自一个被置为混杂模式的网络接口卡的分组。 BPF也可以工作在点对点的链路上，如 SLIP（2.4节），不需要什么特别的处理就可以截获所有通过接口的分组。BPF还可以工作在环回接口上（2.7节）。

> BPF有一个很长的历史。1980年卡耐基梅隆大学的Mike Accetta和Rick Rashid创造了Enet分组过滤程序。斯坦福的Jeffrey Mogul将代码移植到BSD，从1983年开始继续开发。从那以后，它演变为DEC的Ultrix分组过滤器、SunOS 4.1下的一个STREAMS NIT模块和BPF。劳伦斯伯克利实验室的Steven McCanne在1990年的夏天实现了BPF。其中很多设计来自于Van Jacobson。[McCanne and Jacobson 1993] 给出了最新版本的细节以及与Sun的NIT的一个比较。

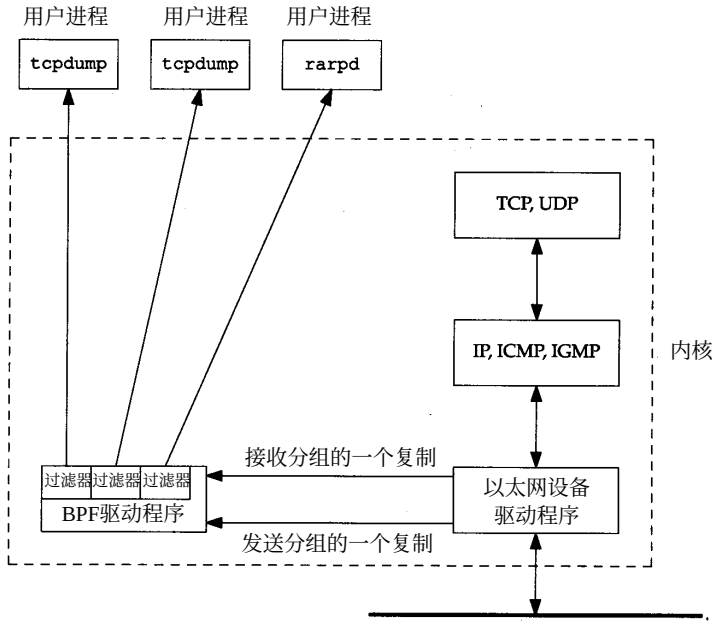
图A-1显示了用于以太网的BPF的特征。

BPF将以太网设备驱动程序设置为混杂模式，然后从驱动程序那里接收每一个收到的分组和传输的分组。这些分组要通过一个用户指明的过滤器，使得只有那些用户进程感兴趣的分组才会传递给用户进程。

多个进程可以同时监视一个接口，每个进程指明了一个自己的过滤器。图　A-1显示了tcpdump的两个实例进程和一个RARP守护进程（5.4节）监视同样的以太网接口。 tcpdump的每个实例指明了一个自己的过滤器。 tcpdump的过滤器可以由用户在命令行指明，而rarpd总是使用只截获RARP请求的过滤器。

除了指明一个过滤器， BPF的每个用户还指明了一个超时定时器的值。因为网络的数据传输率可以很容易地超过 CPU的处理能力，而且一个用户进程从内核中只读小块数据的代价昂

贵，因此，BPF试图将多个帧装载进一个读缓存，只有缓存满了或者用户指明的超时到期才将读缓存保存的帧返回。tcpdump将超时定时器置为1秒，因为它一般从BPF收到很多数据。而RARP守护进程收到的帧很少，所以rarpd将超时置为0（收到一个帧就返回）。



图A-1 BSD分组过滤器

用户指明的过滤器告诉BPF用户进程对什么帧感兴趣，过滤器是对一个假想机器的一组指令。这些指令被内核中的BPF过滤器解释。在内核中过滤，而不在用户进程中，减少了必须从内核传递到用户进程的数据量。RARP守护进程总是使用绑定在程序里的、同样的过滤程序。另一方面，tcpdump在每次运行时，让用户在命令行指明一个过滤表达式。tcpdump将用户指明的表达式转换为相应的BPF的指令序列。tcpdump表达式的例子如下：

```
% tcpdump tcp port 25
% tcpdump'icmp [0] != 8 and icmp[0] != 0
```

第一个只打印源端口和目的端口为25的TCP报文段。第二个只打印不是回送请求和回送应答的ICMP报文（也就是非ping的分组）。这个表达式指明了ICMP报文的第一个字节，图6-2中的type字段，不等于8或0，即图6-3中的回送请求和回送应答。正像你所看到的，设计过滤器需要有底层分组结构的知识。第二个例子中的表达式被放在一对单引号中，防止Unix外壳程序解释特殊字符。

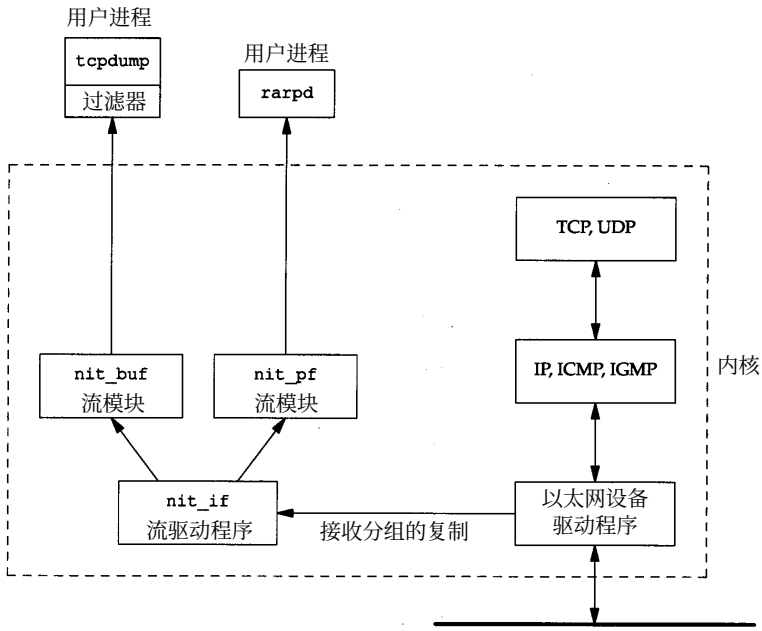
参考tcpdump (1)的手册，了解用户可以指明的表达式的全部细节。bpf(4)的手册详细描述了BPF使用的假想机器指令。[McCanne and Jacobson 1993]比较了这个假想机器方法与其他方法的设计与性能。

A.2 SunOS的网络接口分接头

SunOS 4.1.x提供了一个STREAMS伪设备驱动程序(pseudo-device driver)，称为网络接口分接头(Network Interface Tap)或者NIT（[Rago 1993]包含了流设备驱动程序的其他细节。我

们把这种特征叫作“流”)。NIT类似于BSD分组过滤器，但不如后者功能强大和效率高。图A-2显示了使用NIT所用到的流模块。这个图与图A-1之间的一个不同点在于BPF可以截获网络接口收到的和传送的分组，而NIT只能截获接口收到的分组。将tcpdump与NIT结合起来意味着我们只能看见由网络中其他主机发送来的分组——即根本不可能看见我们自己主机发送的分组（尽管BPF可以工作在SunOS 4.1.x上，但它需要对以太网设备驱动程序的源代码进行改变，大多数的用户没有权限访问源代码，因而这是不可能的)。

当设备/dev/nit被打开时，流驱动程序nit_if就会被打开。既然NIT是使用流来构造的，处理模块可以放在nit_if驱动程序之上。tcpdump将模块nit_buf放在STREAM之上。这个模块将多个网络帧聚集在一个读缓存中，允许用户进程指明一个超时的值。这种情况类似于我们在BPF中所描述的。RARP守护进程没有把这个模块放在它的流之上，因为它只处理了一小部分分组。



图A-2 SunOS的网络接口分接头

用户指明的过滤由流模块nit_pf处理。在图A-2中，我们注意到这个模块被RARP守护进程所用，但没有被tcpdump使用。在SunOS操作系统中，tcpdump代之以在用户进程中完成自己的过滤操作。这么做的理由是nit_pf使用的假想机器的指令与BPF所支持的指令不同(不如BPF所支持的功能强大)。这就意味着当用户对tcpdump指明了一个过滤表达式时，与BPF相比较，使用NIT就会有更多的数据在内核与用户进程之间交换。

A.3 SVR4数据链路提供者接口

SVR4支持数据链路提供者接口DLPI (Data Link Provider Interface)，它是OSI数据链路服务定义的一个流实现。SVR4的大多数版本支持第1版的DLPI，SVR4.2同时支持第1版和第2版，Sun的Solaris 2.x支持第2版，但是增强了一些功能。

像tcpdump的网络监视程序必须使用DLPI来直接访问数据链路设备驱动程序。在Solaris 2.x中，分组过滤的流模块被改名为pfmod，缓存模块被改名为bufmod。

尽管Solaris 2.x还很新，tcpdump在其上的一个实现有一天也会出现。Sun还提供了一个叫作snoop的程序，完成的功能类似于tcpdump（snoop代替了SunOS 4.x的程序etherfind）。作者还不知道tcpdump到vanilla SVR4上的任何端口实现。

A.4 tcpdump的输出

tcpdump的输出是“原始的”。在本书中包含它的输出时，我们对它进行了修改以便阅读。

首先，它总是输出它正在监听的网络接口的名字。我们把这一行给删去了。

其次，tcpdump输出的时间戳在一个微秒精度的系统中采用如同09:11:22.642008的格式，在一个10ms时钟精度的系统中则如同09:11:22.64一样（在附录B中，我们更多地讨论了计算机时钟的精度）。在任何一种情况下，HH:MM:SS的格式都不是我们想要的。我们感兴趣的是每个分组与开始监听的相对时间以及与下面分组的时间差。我们修改了输出以显示这两个时间差。第1个差值在微秒精度的系统中打印到十进制小数点后面6位（对于只有10 ms精度的系统打印到小数点后面2位），第2个差值打印到十进制小数点后面4位或2位（依赖于时钟精度）。

本书中大多数tcpdump的输出都是在sun主机上收集的，它提供了微秒精度。一些输出来自于运行0.9.4版BSD/386操作系统的主机bsdi，它只提供了10 ms的精度（如图5-1所示）。一些输出收集于当bsdi主机运行1.0版BSD/386时，后者提供了微秒级的精度。

tcpdump总是打印发送主机的名字，接着一个大于号，然后是目的主机的名字。这样显示很难追踪两个主机之间的分组流。尽管tcpdump输出仍然显示了数据流的方向，但我们经常把这条输出删掉，代替以产生一条时间线（在本书中的第一次出现是在图6-11）。在我们的时间线上，一个主机在左边，另一个在右边。这样很容易看出哪一边发送分组，哪一边接收分组。

我们给tcpdump的每条输出增加了行号，使得我们可以在书中引用特定的行。还在某些行之间增加了额外的空白，以区别一些不同的分组交换。

最后，tcpdump的输出可能会超出一页的宽度。我们在太长行的适当地方进行了换行。

作为一个例子，相应于图4-4的tcpdump的原始输出显示在图A-3中。这里假设了一个80列的终端窗口。

没有显示我们键入的中断键（用于中止tcpdump），也没有显示接收到的和漏掉的分组的个数（漏掉的分组是那些到达得太快，tcpdump来不及处理的分组。因为本文中的例子经常运行在另外一个空闲网络上，所以漏掉的分组个数总是0）。

```
sun % tcpdump -e
tcpdump: listening on le0
09:11:22.642008 0:0:c0:6f:2d:40 ff:ff:f f:ff:ff:ff arp 60: arp who-has svr4 tell
bsdi
09:11:22 .644182 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 arp 60: arp reply svr4 is-at 0:0
:c0:c2:9b:26
09:11:22.644839 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26  ip 60: bsdi.1030 > svr4.discard:
 S 596459521:596459521(0) win 4096 <mss 1024> [tos 0x10]
09:11:22. 649842 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 ip 60: svr4.discard > bsdi.1030:
 S 3562228225:3562228225(0) ack 596459522 win 4096 <mss 1024>
09:1 1:22.651623 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60: bsdi.1030 > svr4.discard:
 . ack 1 win 4096 [tos 0x10]
                                        我们没有显示其他4个分组
^?                                      键入中断字符来中断显示
9 packets received by filter
0 packets dropped by kernel
```

图A-3 图4-4的tcpdump 的输出

A.5 安全性考虑

很明显，截获网络中传输的数据流使我们可以看到很多不应该看到的东西。例如，Telnet和FTP用户输入的口令在网络中传输的内容和用户输入的一样（与口令的加密表示相比，这称为口令的明文表示。在Unix口令文件中，一般是/etc/passwd或/etc/shadow，存储的是加密的表示）。然而，很多时候一个网络管理员需要使用一个类似于 tcpdump的工具来分析网络中出现的问题。

我们是把tcpdump作为一个学习的工具，用来查看网络中实际传输的东西。对 tcpdump以及其他厂商提供的类似工具的访问权限依赖于具体系统。例如，在 SunOS，对NIT设备的访问只限于超级用户。 BSD的分组过滤器使用了一种不同的技术：通过对 /dev/bpfXX设备的授权来控制访问。一般来说，只有属主才能读写这些设备（属主应该是超级用户），对于同组用户是可读的（经常是系统管理组）。这就是说如果系统管理员不对程序设置用户的 ID，一般的用户是不能运行类似于tcpdump的程序的。

A.6 插口排错选项

查看一个TCP连接上发生的事情的另一种方法是使能插口排错选项，当然是在支持这一特征的系统中。这个特征只能工作在 TCP上（其他协议都不行），并且需要应用程序支持（当应用程序启动时，使能一个插口排错选项）。

大多数伯克利演变的实现都支持这个特征，包括SunOS、4.4BSD和SVR4。

程序使能了一个插口选项，内核就会保留在那个连接上发生的事情的一个痕迹记录。在这之后，所有记录的信息都可以使用 trpt(8)程序打印出来。使能一个插口排错选项不需要特别的许可，但是因为trpt程序访问了内核的内存，所以运行 trpt需要特别的权限。

sock程序（附录C）的-D选项支持这个特征，但是输出的信息比相应的 tcpdump的输出更难解析和理解。然而，我们在 21.4节确实使用它查看了TCP连接上tcpdump不能访问的内核变量。

附录B　计算机时钟

既然本书中的大多数的例子都需要测量一个时间间隔，我们需要更仔细地介绍一下当前Unix系统所采用的记录时间的方法。下面的描述适用于本书中例子所使用的系统，也适用于大多数的Unix系统。[Leffler et al. 1989]的3.4节和3.5节给出了另外的细节。

硬件按照一定的频率产生一个时钟中断。对于Sun SPARC和Intel 80386，时钟中断每10 ms产生一次。

应该注意到大多数的计算机使用一种无补偿的晶体振荡器来生成这些时钟中断。正如RFC1305 [Mills 1992] 的表7指出的，你不要想知道这种振荡器一天的偏差有多少。这就意味着几乎没有计算机能维持精确的时间（即，中断并不是精确地每 10 ms发生一次）。一个0.01%的误差就会产生一个每天 8.64秒的差错。为了得到更好的时间测量需要：（1）一个更好的振荡器；（2）一个外部的更精确的时间资源（如，全球定位卫星提供的时间资源）；或者（ 3）通过因特网访问一个具有更精确时钟的系统。后者通过 RFC1305定义的网络时间协议实现，对该协议的描述超出了本书的范围。

Unix系统中引起时间差错的另一个公共的原因是 10 ms的中断只是引起内核给一个记录时间的变量增 1。如果内核丢失了一个中断（也就是说两个连续中断之间间隔 10 ms对于内核来说太快了），时钟将失去 10 ms。丢失这种类型的中断经常引起 Unix系统丢失时间。

尽管时间中断近似于每10 ms到达一次，更新的系统，如SPARC，提供了一个更高精度的定时器来测量时间差异。通过NIT驱动程序，`tcpdump`（在附录A中描述）已经访问了这个更高精度的定时器。在 SPARC上，这个定时器提供了微秒级的精度。用户进程也可以通过`gettimeofday`(2)函数来访问这个更高精度的定时器。

作者做了下面的试验。我们运行了一个程序，这个程序在一个循环里调用了10 000次`gettimeofday`函数，并将每次的返回值保存在一个数组中。在循环结束后，打印了9999个时间差。对于一个SPARC ELC，时间差的分布如图B-1所示。

微　秒	次数
36	4 914
37	4 831
38	167
39	8
其他	79

图B-1　在SPARC ELC上调用`gettimeofday`函数10 000次所需要的时间分布

在一个空闲的系统中，运行这个程序所花的时钟时间为 `0.38`秒。根据这一点，我们可以说进程调用`gettimeofday`所花的时间大约`37`微秒。既然`ELC`的速度是`21 MIPS`（`MIPS`表示每秒`100`万指令），`37`微秒相应于大约`800`个指令。这些指令对于内核处理一个用户进程的调用、执行系统调用、复制 8个字节的结果及返回给用户进程看起来是合理的（ `MIPS`速度是不可靠的，很难测量当前系统的指令时间。我们试图做的只是得到一个粗略的估计来评价一下上面的值是否有意义）。

从这个简单的试验，我们可以说`gettimeofday`返回的值确实包含了微秒级的精度。

如果我们在SVR4/386上进行类似的测试，结果是不同的。这是因为很多 386 Unix系统，如SVR4，只计数10 ms的时钟中断，而没有提供更高的精度。图 B-2是运行在 25 MHz 80386上的SVR4中9999个时间差的分布。

微 秒	次 数
0	9 871
10 000	128

图B-2 在SVR4/386上调用gettimeofday函数10000次所需要的时间

这些值是无意义的，因为时间差一般小于 10 ms，都被认为是0了。在这些系统中，我们所能做的就是在一个空闲的系统上测量时钟时间，除以循环的次数。这个结果提供了一个上界，因为它包含了调用 `printf`函数9999次的时间和将结果写入一个文件的时间（在 SPARC的情况，图B-1，时间差没有包括`printf`的时间，因为所有10 000个值都是首先获得的，然后才打印结果）。在SVR4的时钟时间为3.15秒，每个系统调用消耗了315微秒。这个大约比SPARC慢8.5倍的系统调用时间看来是正确的。

BSD/386 1.0版提供了类似于SPARC的微秒级的精度。它读8253时钟寄存器，计算从上次时钟中断以来的微秒次数。调用`gettimeofday`的进程和内核模块，如BSD 分组过滤器，可以使用这个精度。

和`tcpdump`联系起来，这些数字意味着我们可以相信在 SPARC和BSD/386系统上打印的毫秒和亚毫秒(submillisecond)的值。而在SVR4/386上，`tcpdump`打印的值总是10 ms的倍数。对于其他打印往返时间的程序，如 `ping`（第7章）和`traceroute`（第8章），在SPARC和BSD/386系统上，我们可以相信它们输出的毫秒值，但在 SVR4/386上，打印的值总是10的倍数。为了在LAN上测量某个`ping`的时间，在第 7章中我们显示的时间是 3 ms，所以需要在SPARC和BSD/386系统上运行`ping`程序。

本书中的一些例子是运行在BSD/386 0.9.4版上，它类似于SVR4，只提供了10 ms的时钟精度。在我们显示这个系统的`tcpdump`输出时，只显示到小数点后面两位，因为这就是所提供的精度。

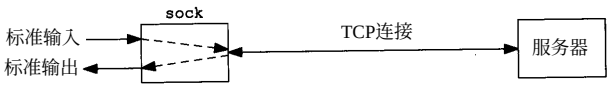
附录C　sock程序

在本书中一直使用一个称为sock的小测试程序，用来生成TCP和UDP数据。它既可以用作一个客户进程，也可以用作一个服务器进程。有这样一个可以从外壳程序执行的测试程序，使我们避免了为每一个我们想要研究的特征编写新的客户和服务器　C程序。因为本书的目的是了解网络互联协议，而不是网络编程，所以在这个附录中我们只描述这个程序和它不同的选项。

有很多与sock功能类似的程序。Juergen Nickelsen写了一个称为socket的程序，Dave Yost写了一个称为sockio的程序。两者都包含了很多类似的特征。sock程序的某些部分也受到了Mike Muuss和Terry Slattery所写的公开域ttcp程序的启发。

sock程序运行在以下四种模式之一：

1) 交互式客户：默认模式。程序和一个服务器相连，然后将标准输入的数据传给服务器，再将从服务器那里接收到的数据复制到标准输出。如图C-1所示。



图C-1　sock 程序作为交互式客户的默认操作

我们必须指明服务器主机的名字和想要连接的服务的名字。主机可指明为点分十进制数，服务可指明为一个整数的端口号。从sun到bsdi与标准的echo服务器（1.12节）相连，回显我们键入的每一个字符：

```
sun % sock bsdi echo
    a test line                    我们键入这一行
    a test line                    echo服务器返回一个复制行
    ^D                             键入文件结束符来中止
```

2) 交互式服务器：指明-s选项。需要指明服务名字（或端口号）：

```
sun % sock -s 5555                 作为一个在端口5555监听的服务器
```

程序等待一个客户的连接请求，然后将标准输入复制给客户，将从客户接收到的东西复制到标准输出。在命令行中，端口号之前可以有一个因特网地址，用来指明接收哪一个本地接口上的连接：

```
sun % sock -s 140.252.13.33 5555        只接受来自以太网的连接
```

默认的模式是接受任何一个本地接口上的连接请求。

3) 源客户：指明-i选项。在默认情况下，将一个1024字节的缓存写到网络中，写1024次。-n选项和-w选项可以改变默认值。例如，

```
sun % sock -i -n12 -w4096 bsdi discard
```

把12个缓存，每个包含4096字节的数据，送给主机bsdi上的discard服务器。

4) 接收器服务器：指明-i选项和-s选项。从网络中读数据然后扔掉。

这些例子都使用了TCP（默认情况），-u选项指明使用UDP。

sock程序有许多选项，用于对程序的运行提供更好的控制。我们需要使用这些选项来产

生本书中用到的所有测试条件。

-b	*n*	将*n*绑定为客户的本地端口号（在默认情况下，系统给客户分配一个临时的端口号）。
-c		将从标准输入读入的新行字符转换为一个回车符和一个换行符。类似地，当从网络中读数据时，将〈回车，换行〉序列转换为新行字符。很多因特网应用需要NVT ASCII（26.4节），它使用回车和换行来终止每一行。
-f	*a.b.c.d.p*	为一个UDP端点指明远端的IP地址（*a.b.c.d*）和远端的端口号（*p*）。
-h		实现TCP的半关闭机制（18.5节）。即，当在标准输入中读到一个文件结束符时并不终止。而是在TCP连接上发送一个半关闭报文，继续从网络中读报文直到对方关闭连接。
-i		源客户或接收器服务器。向网络写数据（默认），或者如果和 -s 选项一起用，从网络读数据。 -n选项可以指明写（或读）的缓存的数目， -w选项可以指明每次写的大小， -r 选项可以指明每次读的大小。
-n	*n*	当和-i 选项一起使用时， *n*指明了读或写的缓存的数目。 *n*的默认值是1024。
-p	*n*	指明每个读或写之间暂停的秒数。这个选项可以和源客户（- i）或接收器服务器（-is）一起使用作为每次对网络读写时的延迟。参考- P选项，实现在第1次读或写之前暂停。
-q	*n*	为TCP服务器指明挂起的连接队列的大小：TCP将为之进行排队的、已经接受的连接的数目（图18-23）。默认值是5。
-r	*n*	和-is选项一起使用， *n*指明每次从网络中读数据的大小。默认是每次读1024字节。
-s		作为一个服务器，而不是一个客户。
-u		使用UDP，而不是TCP。
-v		详细模式。在标准差错上打印附加的细节信息（如客户和服务器的临时端口号）。
-w	*n*	和-i选项一起使用， *n*指明每次从网络中写数据的大小。默认值是每次写1024字节。
-A		使能 SO_REUSEADDR插口选项。对于TCP，这个选项允许进程给自己分配一个处于2MSL等待的连接的端口号。对于UDP，这个选项支持多播，它允许多个进程使用同一个本地端口来接收广播或多播的数据报。
-B		使能SO_BROADCAST插口选项，允许向一个广播 IP地址发送UDP数据报。
-D		使能SO_DEBUG插口选项。这个选项使得内核为这个 TCP连接维护另外的调试信息（A.6节）。以后可以运行trpt(8)程序输出这个信息。
-E		如果实现支持，使能 IP_RECVDSTADDR插口选项。这个选项用于

UDP服务器，用来打印接收到的UDP数据报的目的IP地址。

-F 指明一个并发的TCP服务器。即，服务器使用fork函数为每一个客户连接创建一个新的进程。

-K 使能TCP的SO_KEEPALIVE插口选项（第23章）。

-L *n* 把一个TCP端点的拖延时间(linger time)（SO_LINGER）设置为*n*。一个为0的拖延时间意味着当网络连接关闭时，正在排队等着发送的任何数据都被丢弃，向对方发送一个重置报文（ 18.7节）。一个正的拖延时间（百分之一秒）是关闭网络连接必须等待的将所有正在排队等着发送的数据发送完并收到确认的时间。关闭网络连接时，如果这个拖延定时器超时，挂起的数据没有全部发送完并收到确认，关闭操作将返回一个差错信息。

-N 设置TCP_NODELAY插口选项来禁止Nagle算法（19.4节）。

-O *n* 指明一个TCP服务器在接受第一个客户连接之前暂停的秒数。

-P *n* 指明在第一次对网络进行读或写之前暂停的秒数。这个选项可以和接收器服务器（- is）一起使用，完成在接受了客户的连接请求之后但在执行从网络中第一次读之前的延迟。和接收源（- i）一起使用时，完成连接建立之后但第一次向网络写之前的延迟。参看- p选项，实现在接下来的每一次读或写之间进行暂停。

-Q *n* 指明当一个TCP客户或服务器收到了另一端发来的一个文件结束符，在它关闭自己这一端的连接之前需要暂停的秒数。

-R *n* 把插口的接收缓存（SO_RCVBUF插口选项）设置为*n*。这可以直接影响TCP通告的接收窗口的大小。对于 UDP，这个选项指明了可以接收的最大的UDP数据报。

-S *n* 把插口的发送缓存（SO_SNDBUF插口选项）设置为*n*。对于UDP，这个选项指明了可以发送的最大的UDP数据报。

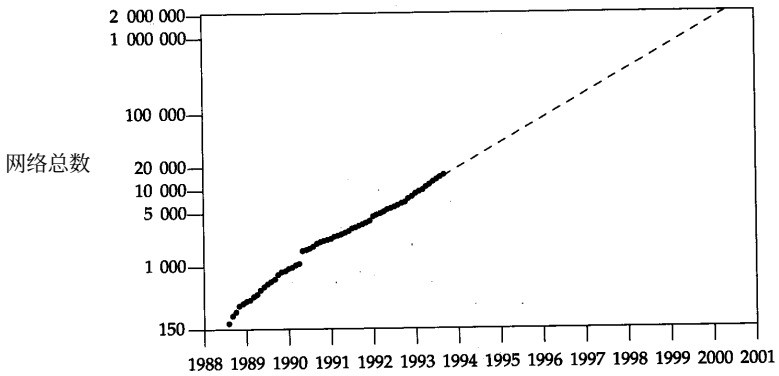
-U *n* 在向网络写了数字*n*后进入TCP的紧急模式。写一个字节的数据以启动紧急模式（20.8节）。

附录D　部分习题的解答

第1章

1.1　答案是：2^7-2（126）＋$2^{14}-2$（16 382）＋$2^{21}-2$（2 097 150）= 2 113 658。每一部分都减去2是因为全0或全1网络ID是非法的。

1.2　图D-1显示了直到1993年8月的有关数据。



图D-1　宣布加入NSFNET的网络数

如果网络数继续呈指数增长的话，虚线估计了2000年可能达到的最大的网络数。

1.3　“自由地接收，保守地发送。”

第3章

3.1　不，任何网络ID为127的A类地址都是可行的，尽管大多数系统使用了127.0.0.1。

3.2　kpno有5个接口：3个点对点链路和2个以太网接口。R10有4个以太网接口。gateway有3个接口：2个点对点链路和1个以太网接口。最后，netb有1个以太网接口和2个点对点链路。

3.3　没有区别：作为一个没有再区分子网的C类地址，它们都有一个255.255.255.0的子网掩码。

3.5　它是合法的，被称为非连续的子网掩码，因为其用于子网掩码的16位是不连续的。但是RFC建议反对使用非连续的子网掩码。

3.6　这是一个历史遗留问题。值是1024＋512，但是打印的MTU值包含了所有需要的首部字节数。Solaris 2.2将回环接口的MTU设置为8232（8192＋40），其中包含了8192字节的用户数据加上20字节的IP首部和20字节的TCP首部。

3.7　第一，数据报降低了路由器中对于连接状态的需求。第二，数据报提供了基本的构件，在它的上面可以构造不可靠的（UDP）和可靠的（TCP）的运输层。第三，数据报代表

了最小的网络层假定，使得可以使用很大范围的数据链路层服务。

第4章

4.1 发出一条rsh命令与另一台主机建立一个TCP连接。这样做引起在两个主机之间交换IP数据报。为此，在那台主机的ARP缓存中必须有我们这台主机的登记项。因此，即使在执行rsh命令之前，ARP缓存是空的，当rsh服务器执行arp命令时，必须保证ARP缓存中登记有我们这台主机。

4.2 保证你的主机上的ARP缓存中没有登记以太网上的某个叫作foo的主机。保证foo引导时发送一个免费ARP请求，也许是在foo引导时，在那台主机上运行tcpdump。然后关闭主机foo，使用说明了temp选项的arp命令，在你的系统的ARP缓存中为foo输入一个不正确的登记项。引导foo并在它启动好之后，察看主机的ARP缓存，看看不正确的登记项是不是已经被更正了。

4.3 阅读主机需求（Host Requirement）RFC的2.3.2.2节和本书中的11.9节。

4.4 假设当服务器关闭时，客户机保存了关于服务器的一个完整的ARP登记项。如果我们继续试图与（已关闭的）服务器联系，过了20分钟以后，ARP将超时。最后，当服务器以另一个新的硬件地址重启动。如果它没有发出一个免费ARP，旧的、不再正确的ARP登记项仍然存在于客户机上。我们将无法和在新硬件地址上的服务器联系直到我们手工删除这个ARP登记项，或者在20分钟内停止与服务器联系的尝试。

第5章

5.1 一个单独的帧类型并不是必需的，因为图4-3中的op字段对于所有的四个操作（ARP请求、ARP应答、RARP请求和RARP应答）都有一个不同的值。但是实现一个RARP服务器，独立于内核中的ARP服务器，更容易处理不同的帧类型字段。

5.2 每个RARP服务器在发送一个响应之前可以延迟一个小的随机时间。

作为一个优化，可以指定一个RARP服务器为主服务器，其他的为次服务器。主服务器发出响应不需要延迟，而次服务器发出响应则需要一个随机的延迟。

作为另一个优化，也是指定一个主RARP服务器，其他为次服务器。次服务器只对在一个短时间段内发生的重复请求进行响应。这里假设出现重复请求的原因是由于主服务器停机了。

第6章

6.1 如果在局域网线上有一百个主机，每个都可能在同一时刻发送一个ICMP端口不可达的报文。很多报文的传输都可能发生冲突（如果使用的是以太网），这将导致1秒或2秒的时间里网络不可用。

6.2 它是一个“should”。

6.3 如我们在图3-2所指出的，发送一个ICMP差错总是将TOS置为0。发送一个ICMP查询请求可以将TOS置为任何值，但是发送相应的应答必须将TOS置为相同的值。

6.4 netstat -s是查看每个协议统计数据的常用方法。在一台收到了4800万个IP数据报的SunOS 4.1.1主机（gemini）上，ICMP的统计为：

```
Output histogram:
        echo reply: 1757
        destination unreachable: 700
        time stamp reply: 1
Input histogram:
        echo reply: 211
        destination unreachable: 3071
        source quench: 249
        routing redirect: 2789
        echo: 1757
        #10: 21
        time exceeded: 56
        time stamp: 1
```

21个类型为10的报文是SunOS 4.1.1不支持的路由器的请求。

也可以使用 SNMP（图25-26），有些系统，如 Solaris 2.2，可以生成使用 SNMP变量名的netstat-s的输出。

第7章

7.2 86字节除以960字节/秒，乘以2得到179.2 ms。当以这个速率运行ping时，打印的值为180 ms。

7.3 （86＋48）除以960字节/秒，乘以2得到279.2 ms。另外的48字节是因为56字节的数据部分的最后48字节必须忽略：0xc0是SLIP END字符。

7.4 CSLIP只压缩了TCP报文段的TCP首部和IP首部。它对ping使用的ICMP报文没有作用。

7.5 在一个SPARC工作站 ELC上，对回环地址的ping操作产生一个1.310 ms的RTT，而对一个主机的以太网地址的ping操作产生一个1.460 ms的RTT。这个差值是由于以太网驱动程序需要时间来判定这个数据报的目的地址是一个本地的主机。需要一个产生微秒级输出的ping来验证这一点。

第8章

8.1 如果一个输入数据报的 TTL为0，做减一操作然后测试会将把 TTL设置为255，并且让数据报继续传输。尽管一个路由器永远不会收到一个 TTL为0的数据报，但这种情况确实会发生。

8.2 我们注意到traceroute在UDP数据报的数据部分存储了 12个字节，其中包含了数据报发送的时间。然而，从图6-9可以看出ICMP只返回了出错的IP数据报的头8个字节，实际上这是8个字节的UDP首部。因此，ICMP的差错报文没有返回traceroute存储的时间值。traceroute保存了它发送分组的时间，当收到一个ICMP应答时，取出当时的时间，把两个值相减就可以得出RTT。

回忆一下第7章中，ping在输出的ICMP回显请求中存储了时间，这个值被服务器回显了回来。这样即使分组返回时失序，ping也能打印出正确的RTT。

8.3 第1行输出是正确的，并且标识了 R1。下一个探测分组启动时将 TTL置为2，并且这个值被R1减1。当R2收到这个分组时，把 TTL从1减为0，但是错误地将它传递给了 R3。R3看见进入的TTL是0就将超时的分组发送回来。这就意味着第2行输出（TTL为2）标识了R3，而不是R2。第3行输出正确地标识了 R3。这个错误所表现出来的线索就是两个连续的输

出行标识了同一个路由器。

8.4 在这种情况下，TTL为1标识了R1，TTL为2标识了R2，TTL为3标识了R3，但是当TTL为4时，UDP数据报到达了目的地，其输入的TTL为1。ICMP端口不可达报文生成了，但它的TTL是1（错误地从进入的TTL复制而来）。这个ICMP报文到了R3，在那儿TTL被减1，报文被丢弃。没有生成一个ICMP超时报文，因为被丢弃的数据报是一个ICMP的差错报文（端口不可达）。类似的现象也出现在TTL为5的探测分组，但这次输出的端口不可达报文以TTL为2开始（进入的TTL），这个报文被传给R2，在那儿被丢弃。对应于TTL为6探测分组的端口不可达报文被传递给R1，在那儿被丢弃。最后，对应于TTL为7探测分组的端口不可达报文被送回了原地，到达时它的TTL为1（traceroute认为一个TTL为0或1的到达ICMP报文是有问题的，因此它在RTT后面打印了一个惊叹号）。总之，TTL为1、2和3的行正确地标识了R1、R2和R3，接下来的三行每个都包含三个超时，再接下来的TTL为7的行标识了目的地。

8.5 它表明这些路由器都将一个ICMP报文的输出TTL设置为255，这是共同的。从netb输入的255的TTL值是我们想要的，而从butch输入的253的TTL值表明在butch和netb之间可能有一个未觉察的路由器。否则，在这个点上我们应该看到一个TTL值为254的输入报文。类似地，我们希望看到一个值为252而不是249的、来自enss142.UT.westnet.net的报文。这表明这些未觉察的路由器没有正确地处理向外输出的UDP数据报，但它们都对返回的ICMP报文正确地进行了TTL减1操作。

我们必须在查看输入的TTL时非常小心，因为有时候一个和我们想要的不同的值可能是由于返回的ICMP报文采用了一条与输出UDP数据报不同的路径。但是，在这个例子中证实了我们所怀疑的——当使用松源路由选项时，确实存在traceroute没有发现的未察觉的路由器。

8.7 ping的客户把ICMP回显请求报文（图7-1）的标识符字段设置为它的进程ID。ICMP回显应答报文包含同样值的标识符字段。每个客户都要查看这个返回的标识符字段，并且只处理那些它发送过的报文。

traceroute客户将它的UDP源端口号设置为它的进程ID和32768的逻辑或。因为返回的ICMP报文总是包含产生错误的IP数据报的前8个字节（图6-9），这8个字节包括了完整的UDP首部，所以这个源端口号在ICMP差错报文中被返回。

8.8 ping客户将ICMP回显请求报文的可选数据部分设置为分组发送的时间。这个可选的数据必须在ICMP回显应答中返回。这样使得即使分组返回时失序，客户也能计算出精确的回环时间。

traceroute客户不能这样操作，因为在ICMP差错报文中返回的只是UDP首部（图6-9），没有UDP数据。因此，traceroute必须记住它发送一个请求的时间，等待应答，然后计算两者的时间差。

这里显示了ping和traceroute的另一个不同点：ping每秒发送一个分组，而不管是否收到任何应答；traceroute发送一个请求，然后在发送下一个请求前等待一个应答或者一个超时。

8.9 因为默认情况下Solaris 2.2从32768开始使用临时的UDP端口，所以目的主机上的目的端口已经被使用的机会更大。

第9章

9.1 当ICMP标准第1次发布时，RFC 792 [Postel 1981b]所述的划分子网技术还没有使用。另外，使用一个网络重定向而不是N个主机重定向（对于目的网络中的所有*N*个主机）也节省了路由表的空间。

9.2 这一项并不需要，但是如果把它删除了，所有到slip的IP数据报将被发送到默认的路由器（sun），后者又将把它们送到路由器bsdi。既然sun将数据报从与接收数据报相同的接口转发出去，它把一个ICMP重定向到svr4。这样在svr4中又创建了我们删除过的同样的路由表项，尽管这一次是因为重定向而创建的，而不是在引导时增加的。

9.3 当那个4.2BSD主机收到目的地址是140.1.255.255的数据报，发现它有一个通往该网络（140.1）的路由，因此就试图转发数据报。它发送一个ARP广播来寻找140.1.255.255。这个ARP请求没有收到任何应答，所以这个数据报最终被丢弃。如果在网线上有很多这样的4.2BSD主机，每一个都在差不多同一时刻发送ARP这个广播，将会暂时地阻塞网络。

9.4 这次，每一个ARP请求都收到一个应答，告诉每个4.2BSD主机向一个指定的硬件地址（以太网广播）发送数据报。如果网线上有*k*个这样的4.2BSD主机，全部收到了它们自己的ARP应答，使得每一个生成了另一个广播。每个主机都收到了每一个目的地址为140.1.255.255的广播IP数据报，既然现在每个主机都有一个ARP缓存项，这个数据报又被转发给了广播地址。这个过程继续下去，就会产生一次以太网的熔毁（Ethernet meltdown）。[Manber 1990] 描述了网络中另一种形式的链式反应。

第10章

10.1 路由表中有13条来自于kpno：除了140.252.101.0和140.252.104.0之外的所有gateway直接相连的其他网络。

10.2 丢失的数据报中通告的25条路由需要60秒才能得到更新。这不成问题，因为一般来说一条路由如果连续3分钟没有得到更新，RIP才会声明它失效。

10.3 RIP运行在UDP上，而UDP提供了UDP数据报中数据部分的一个可选的检验和（11.3节）。然而，OSPF运行在IP上，IP的检验和只覆盖了IP首部，所以OSPF必须增加它自己的检验和字段。

10.4 负载平衡增加了分组被失序交付的机会，并且很可能使得运输层计算的环回时间出错。

10.5 这叫作简单的分裂范围（split horizon）。

10.6 在图12-1中，我们显示了100个主机的每一个都通过设备驱动程序、IP层和UDP层来处理这个广播的UDP数据报。当它们发现UDP端口520没有被使用时，这个广播数据报才最终被丢弃。

第11章

11.1 因为使用IEEE 802封装时，存在8个额外的首部字节，所以1465个字节的用户数据是引起分片的最小长度。

11.3 对于IP来说有8200字节的数据需要发送，8192字节的用户数据和8个字节的UDP首部。

采用tcpdump记号，第1个分片是1480@0+（1480字节的数据，偏移为0，将“更多片”比特置1）。第2个是1480@1480+，第3个是1480@2960+，第4个是1480@4440+，第5个是1480@5920+，第6个是800@7400。1480×5 + 800 = 8200，正好是要发送的字节。

11.4 每个1480字节的数据报片被分成三小片：两个528字节和一个424字节。小于532（552-20）的8的最大倍数是528。800字节的数据报片被分成两小片：一个528字节和一个272字节。这样，原来8192字节的数据报变成了SLIP链路上的17个帧。

11.5 不。问题是当应用程序超时重传时，重传产生的IP数据报有一个新的标识字段。而重新装配只针对那些具有相同标识字段的分段。

11.6 IP首部中的标识字段（47942）是一样的。

11.7 第一，从图11-4我们看到gemini没有使能输出UDP的检验和。如果输出UDP的检验和没有被使能，这个主机上的操作系统（SunOS 4.1.1）就不会验证一个进入UDP的检验和。第二，大多数的UDP通信量都是本地的，而不是WAN的，因此没有服从所有的WAN特征。

11.8 不严格的和严格的源站选路选项被复制到每一个数据报片中。时间戳选项和记录路由选项没有被复制到每一个数据报片中——它们只出现在第1个数据报片中。

11.9 不。在11.12节中，我们看到很多实现可以根据目的IP地址、源IP地址和源端口号来过滤送往一个给定UDP端口号的输入数据报。

第12章

12.1 广播本身不会增加网络通信量，但它增加了额外的主机处理时间。如果接收主机不正确地响应了诸如ICMP端口不可达之类的差错，那么广播也可能导致额外的网络通信量。路由器一般不转发广播分组，而网桥一般转发，所以在一个桥接网络上的广播分组可能比在一个路由网络上走得更远。

12.2 每个主机都收到了所有广播分组的一个副本。接口层收到了帧，把它传递给设备驱动程序。如果类型字段指的是其他协议，设备驱动程序就会丢弃该帧。

12.3 首先执行netstat -r来看一下路由表，结果显示了所有接口的名字。然后对每个接口执行ifconfig（3.8节）：标志指出了一个接口是否支持广播，如果支持，相应的广播地址也会被输出。

12.4 伯克利演变的实现不允许对一个广播数据报进行分片。当我们说明了1472字节的长度，产生的IP数据报将是1500字节，正好是以太网的MTU。不允许分片一个广播数据报是一个策略上的决定——没有技术上的原因（并不是想要减少广播分组的数目）。

12.5 依赖于100个主机上不同的以太网接口卡的多播支持，多播数据报可能被接口卡忽略，或者被设备驱动程序丢弃。

第13章

13.1 生成随机数时要使用对于主机唯一的值。IP地址和链路层地址是每个主机都应该不一样的两个值。日期时间是一个不好的选择，尤其是在所有的主机都运行了一个类似于NTP的协议来同步它们的时钟的情况下。

13.2 他们增加了一个包括一个序号和一个时间戳的应用协议首部。

第14章

14.1 一个解析器总是一个客户，但一个名字服务器既是一个客户又是一个服务器。

14.2 问题被返回，它占用了前44个字节。一个回答占用了剩下来的31个字节：2个字节指向域名的指针（即，指向问题中域名的一个指针），10字节固定长度的字段（类型、种类、TTL和资源长度），19字节的资源数据（一个域名）。注意到资源数据中的域名（svr4.tuc.noao.edu.）没有共享问题（34.13.252.140.in-addr.arpa.）中域名的后缀，所以不能使用一个指针。

14.3 将顺序颠倒意味着首先使用DNS，如果使用DNS失败，然后才将参数翻转过来作为一个点分十进制数。这就是说每次说明一个点分十进制数，都要使用DNS，涉及一个名字服务器。这是对资源的一种浪费。

14.4 RFC 1035的4.2.2节说明了在实际的DNS报文之前的两个字节长度的字段。

14.5 当一个名字服务器启动时，它一般从一个磁盘文件中读出一个根服务器列表（可能已经过时了）。然后尝试和这些根服务器中的一个联系，请求根域的名字服务器记录（一个NS的查询类型）。这个请求返回了当前最新的根服务器列表。启动磁盘文件中根服务器项中至少需要一个是有效的。

14.6 InterNIC的注册服务每一周更新三次根服务器。

14.7 就像应用是不定的一样，解析器也是不定的。如果系统配置成使用多个名字服务器，而且解析器是无状态的，那么解析器就不能记住不同的名字服务器的往返时间。这样定时太短的解析器将会超时，引起不必要的重传。

14.8 对A记录的排序应该由解析器来执行，而不是名字服务器，因为解析器一般比服务器了解更多的客户的网络拓扑（更新版本的BIND提供了解析器对A记录排序的功能）。

第15章

15.1 送往广播地址的TFTP请求应该被忽略。正像Host Requirements RFC所描述的，对一个广播请求的响应可能产生一个非常严重的安全漏洞。但是，问题是并不是所有的实现和API都对接收一个UDP数据报的进程提供了该数据报的目的地址（11.12节）。因为这个原因，很多TFTP服务器没有严格遵守这个限制。

15.2 不幸的是，RFC没有提到这个块数目环绕问题。具体实现时应该能够传输最大为33 553 920（65535×512）字节的文件。但是当文件的长度超过16 776 704（32767×512）时，很多实现都会失败，因为它们将块数目错误地表示为一个有符号的16位整数，而不是一个无符号的整数。

15.3 这样简化了编写一个适合于只读内存的TFTP客户的工作，因为服务器是引导文件的发送者，所以服务器必须实现超时和重传机制。

15.4 利用它的停止等待协议，TFTP可以在每一次客户与服务器的往返过程中最多传输512字节的数据。TFTP的最大吞吐量就是512字节除以客户与服务器之间的往返时间。在以太网上，假设一个往返时间为3 ms，那么最大的吞吐量就是大约170 000字节/秒。

第16章

16.1 一个路由器可以转发一个RARP请求到路由器连接的其他网络上的任何一台主机上。但

是发送应答就成问题了，路由器还必须转发RARP应答。

BOOTP没有这个应答问题，因为应答的地址是路由器知道如何转发的一般IP地址。问题是RARP只使用了链路层地址，路由器一般不知道在其他的、没有连接在路由器的网络上主机的链路层地址。

16.2 它可能使用了自己的硬件地址。该地址应该是唯一的，在请求报文中设置，在应答中返回。

第17章

17.1 除了UDP的检验和，其他都是必需的。IP检验和只覆盖了IP首部，而其他字段都紧接着IP首部开始。

17.2 源IP地址、源端口号或者协议字段可能被破坏了。

17.3 很多Internet应用使用一个回车和换行来标记每个应用记录的结束。这是NVT ASCII采用的编码（26.4节）。另外一种技术是在每个记录之前加上一个记录的字节计数，DNS（习题14.4）和Sun RPC（29.2节）采用了这种技术。

17.4 就像我们在6.5节所看到的，一个ICMP差错报文必须至少返回引起差错的IP数据报中除了IP首部的前8个字节。当TCP收到一个ICMP差错报文时，它需要检查两个端口号以决定差错对应于哪个连接。因此，端口号必须包含在TCP首部的前8个字节里。

17.5 TCP首部的最后有一些选项，但UDP首部中没有选项。

第18章

18.1 ISN是一个32 bit的计数器，它在系统引导大约9.5小时后从4 294 912 000环绕到8704。再过大约9.5小时它将环绕到17 408，然后再过9.5小时是26 112，如此继续下去。因为系统引导时ISN从1开始，并且因为最低次序的数字在4、8、2、6和0之间循环，所以ISN应该总是一个奇数。

18.2 在第1种情况下，我们使用了sock程序。默认情况下它把Unix的新行字符不作改变地进行传输——单个ASCII字符012（八进制）。在第2种情况下，我们使用了Telnet客户，它把Unix的新行字符转变为两个ASCII字符——一个回车符（八进制015）跟着一个换行符（八进制012）。

18.3 在一个半关闭的连接上，一个端点已经发送了一个FIN，正等待另一端的数据或者一个FIN。一个半打开的连接是当一个端点崩溃了，而另一端还不知道的情况。

18.4 一个连接只有经过了已建立状态才能进入2MSL等待状态。

18.5 首先，日期服务器在将时间和日期写给客户之后对TCP连接做一个主动关闭。这可以通过sock程序打印的消息："connection closed by peer."表现出来。连接的客户端经历了被动关闭的状态。这样就把一对插口置于服务器端的TIME_WAIT状态，而不是在客户端。

其次，正如在18.6节所示，大多数伯克利演变的实现都允许一个新的连接请求到达一个正处于TIME_WAIT状态的一对插口，这也就是这里所发生的情况。

18.6 因为在一个已经关闭的连接上到达了一个FIN，所以相应于这个FIN发送了一个复位。

18.7 拨号的一方做主动打开，电话振铃的一方做被动打开。不允许同时打开，但允许同时关

闭。

18.8 我们将只看到ARP请求，而不是TCP SYN报文段，但ARP请求将和图中具有相同的计时。

18.9 客户在主机solaris上，服务器在主机bsdi上。客户对服务器SYN的确认（ACK）和客户的第一个数据段结合在一起（第3行）。这种处理非常符合TCP的规则，尽管大多数的实现都没有这么做。接着，客户在等待它的数据的确认之前发送了它的 FIN（第4行）。这样使得服务器可以在第5行同时确认客户数据和它的FIN。

这次交互（将一个报文段从客户发送到服务器）需要 7个报文段，而正常的连接建立和终止（图18-13），以及一个数据段和它的确认，需要9个报文段。

18.10 首先，服务器对客户的 FIN的确认一般不会被延迟（我们在 19.3节讨论延迟的确认），而是在FIN到达后立即发送。应用进程需要一些时间来接收 EOF，告诉它的TCP关闭它这一端的连接。第二，服务器收到客户的FIN后，并不一定要关闭它这一端的连接。就像我们在18.5节中看到的，仍然可以发送数据。

18.11 如果一个产生RST的到达报文段有一个 ACK字段，那么RST的序号就是到达的ACK字段。第6行中值为1的ACK是相对于第2行中的26368001的ISN。

18.12 参见 [Crowcroft et al. 1992]中对分层的评论。

18.13 发出了5个查询。假设有3个分组用于建立连接， 1个用于查询，1个用于确认查询，1个用于响应， 1个用于确认响应， 4个用于终止连接。这就是说每次查询需要 11个分组，总共需要55个分组。使用UDP则可以减少到10个分组。

如果对查询的确认和响应结合在一起，每个查询需要的分组可以减少到 10个（19.3节）。

18.14 限制是每秒268个连接：最大数目的TCP端口号（65536 - 1024 = 64512，忽略知名端口）除以TIME_WAIT状态的2MSL。

18.15 重复的FIN会得到确认， 2MSL定时器重新开始。

18.16 在TIME_WAIT状态中收到一个RST引起状态过早地终止。这就叫作 TIME_WAIT断开(assassination)。RFC1337 [Braden 1992a]仔细讨论了这个现象，并显示了潜在的问题。这个RFC提出的简单的修改就是在TIME_WAIT状态时忽略RST段。

18.17 它是在具体实现不支持半关闭连接的时候。一旦应用进程引起发送一个 FIN，应用进程就不能再从这个连接读数据了。

18.18 不。输入的数据段通过源 IP地址、源端口号、目的 IP地址和目的端口号进行区分。在18.11节中我们看到对于输入的连接请求，一个 TCP服务器一般可以通过目的 IP地址来拒绝接收。

第19章

19.1 应用程序的两个写操作，跟着一个读操作，引起了迟延，因为 Nagle算法很可能被激活。第一个报文段（包含8个字节的数据）被发送后，在发送后面 12个字节的数据之前必须等待第一个报文段的确认。如果服务器实现了延迟的确认，在收到这个确认之前，可能会有一个达到200 ms的时延（加上RTT）。

19.2 假设5个字节的CSLIP首部（IP和TCP）和两个字节的数据，这些段通过 SLIP链路的RTT大约是14.5 ms。我们要加上通过以太网的 RTT（一般是5~10 ms），加上在sun和bsdi

上的选路时间。因此，观察到的时间看起来是正确的。

19.3 在图19-6中，第6和第9报文段之间的时间是533ms。在图19-8中，第8和第12报文段之间的时间是272ms（我们测量了F2键的时间，而不是F1键，因为F1键的第1个回显在第2个图中丢掉了）。

第20章

20.1 字节号0是SYN，字节号8193是FIN。SYN和FIN在序号空间里各占用了一个字节。

20.2 应用程序的第1个写操作引起置位 PUSH标志发送第1个报文段。因为 BSD/386总是使用慢启动，它在发送更多数据之前等待第1个确认。在这段时间里，下面三个写操作发生了，发送TCP把要发送的数据缓存了起来。因为在缓存中有更多的数据要发送，下面的三个报文段没有包含PUSH标志。最后，慢启动跟上了应用程序的写操作，每个应用程序的写操作引起了发送一个报文段，并且因为那个报文段是缓存中最后的一个，所以设置PUSH标志。

20.3 为容量求解带宽迟延方程式，第一种情况是 1920字节，卫星的情况是2062字节。看起来TCP只声明了一个2048字节的窗口。

一个大于16000字节的窗口应该能够使卫星链路饱和。

20.4 不，因为TCP超时之后可能重新对数据进行分组，就像我们将在 21.11节中看到的。

20.5 作为应用进程读数据的结果，第 15报文段是TCP模块自动发送的窗口更新，它引起窗口的打开。这类似于图中的第 9报文段。但是，第 16报文段是应用进程关闭它这一端连接的结果。

20.6 这可能引起发送者以比网络实际能够处理的更快的速率向网络发送分组。这叫作确认压缩(ACK compression)或确认粉碎(ACK smashing) [Mogul 1993, 15.8.13节]。这个引用显示了在Internet上发生的ACK压缩，尽管它很少会导致拥塞。

第21章

21.1 下一个超时设定是48秒：0＋4×12。因子4是指数退避的下一个乘数。

21.2 看起来SVR4在计算*RTO*时仍然使用了因子2*D*，而不是4*D*。

21.3 TFTP使用的停止等待协议被限制为每个往返过程传送 512字节的数据。32768/512×1.5是96秒。

21.4 显示了4个报文段，编号为1、2、3和4。假设接收的顺序是1、3、2和4，接收者产生的确认将是ACK 1（一个正常的确认），ACK 1（当收到了报文段3，失序后一个重复的确认），当收到了报文段2后的ACK 3（对报文段2和报文段3的确认），然后是ACK 4。这儿产生了一个重复的确认。如果接收的顺序是1、3、4和2，将会产生两个重复的确认。

21.5 不，因为斜率仍然是向上和向右，不是向下。

21.6 参见图E-1。

21.7 在图21-2中，报文段包含256个字节的数据，在`slip`和`bsdi`之间的9600 b/s的CSLIP链路传输大约需要250 ms。假定数据段没有在`bsdi`和`vangogh`之间的某个地方排队，它们分别需要大约250 ms到达`vangogh`。因为这个时间超过了延迟确认定时器的 200 ms时间，当下一个延迟确认定时器超时时，每个报文段得到确认。

第22章

22.1 主机bsdi上的确认很可能都要被延迟，因为没有理由立即发送它们。这就是为什么相对次数有一个0.170和0.370的小数部分。看起来bsdi上200 ms的定时器在sun上同样的定时器之后18 ms才开始计时。

22.2 FIN标志，和SYN标志一样，在序号空间占据了 1个字节。通知的窗口看起来小了一个字节，因为TCP允许FIN标志在序号空间占用1个字节的空间。

第23章

23.1 通常激活keepalive选项比显式地编写应用程序探测报文更容易； keepalive探测报文比应用程序探测报文占用更少的网络带宽（因为 keepalive探测报文和应答不包含任何数据)；如果连接不是空闲的，就不会发送探测报文。

23.2 keepalive选项可能会由于一个临时性的网络中断而引起一个非常好的连接断开；发送探测报文的间隔（2小时）一般不可以根据应用程序进行配置。

第24章

24.1 它意味着发送TCP支持窗口扩缩选项，但这个连接并不需要扩缩它的窗口。另一端（接收这个SYN的）可以说明一个窗口扩缩因子（可以是 0或非0）。

24.2 64 000：接收缓存大小（128 000）向右移1位。55 000：接收缓存（220 000）向右移2位。

24.3 不。问题是确认没有可靠地交付（除非它们被数据捎带在一起发送），因此，一个确认上的扩缩改变可能会丢失。

24.4 $2^{32} \times 8/120$等于286 Mb/s，2.86倍于FDDI数据率。

24.5 每个TCP将不得不记住从每个主机的任何一个连接上收到的上一个时间戳。阅读 RFC 1323的附录B.2以了解进一步的细节。

24.6 应用程序必须在和另一端建立连接之前设置接收缓存的大小，因为窗口扩缩选项在初始的SYN段中发送。

24.7 如果接收者每次确认第2个数据报文段，吞吐量是 1 118 881字节/秒。若使用一个62个报文段的窗口，每31个报文段确认一次，则数值是1 158 675。

24.8 使用这个选项，确认中回显的时间戳总是来自于引起确认的报文段。对哪个重传的报文段进行确认没有疑问，但仍然需要Karn算法的另一部分，即处理重传的指数退避。

24.9 接收TCP对数据进行排队，但只有完成了三次握手后，数据才能传递给应用程序：当接收TCP进入了ESTABLISH状态。

24.10 交换了5个报文段：

1) 客户到服务器：SYN，数据（请求）和FIN。服务器必须像上个习题所述的一样对数据进行排队。

2) 服务器到客户：SYN和对客户SYN的确认。

3) 客户到服务器：对服务器的 SYN的确认和客户的FIN（再次）。这样使得服务器进入已建立状态，并且来自报文段1的排队数据被传递给服务器应用程序。

4) 服务器到客户：客户 FIN的确认（它也确认了客户的数据）、数据（服务器的应答）

和服务器的FIN。这里假定了SPT足够短以允许这个延迟的确认。当客户 TCP收到这个报文段，就将应答传递给客户端的应用程序，但是整个时间是 RTT加上SPT的两倍。

5) 客户到服务器：对服务器FIN的确认。

24.11 每秒16 128次交互（64 512除以4）。

24.12 使用T/TCP的交互时间不可能比两个主机之间交换一个 UDP数据报所需的时间短。因为T/TCP涉及了UDP没有做的状态处理，所以T/TCP总是要花更多的时间。

第25章

25.1 如果一个系统运行了一个管理进程和一个代理进程，它们很可能是不同的进程。管理进程在UDP端口162监听trap告警，代理进程在 UDP端口161等待请求。如果 trap告警和SNMP请求使用了同样的端口，将很难区分管理进程和代理进程。

25.2 参考25.7节中的“ 表访问（Table Access）”部分。

第26章

26.1 我们预期从服务器来的第 2、4和9报文段将被延迟。第 2和第4报文段之间的时间差是190.7 ms，第2和第9报文段之间的时间差是400.7 ms。

从客户到服务器的所有确认看起来都被延迟：第 6、11、13、15、17和19报文段。从第6报文段开始的最后5个时间差是400.0、600.0、800.0、1000.0和2.600 ms。

26.2 如果连接的一个端点处于TCP的紧急模式，每次收到一个报文段，就会发送一个报文段。这个报文段没有告诉接收者任何新的东西（例如，它不是对新数据的确认），它也不包含数据，它只是重申进入了紧急模式。

26.3 只有512个这样的保留端口（ 512~1023），限制了一个主机只能有 512个远程登录的（Rlogin）客户。在实际生活中，这个限制一般小于 512个，因为在这个范围中的一些端口号被不同的服务器，如远程登录服务器，用作了知名端口。

TCP的限制是一对插口定义的一个连接（ 4元组）必须是唯一的。因为 Rlogin服务器总是使用了同样的知名端口（513），一台主机上多个Rlogin客户只有在它们和不同的服务器主机相连接时才可以使用相同的保留端口。然而， Rlogin客户没有采用重用保留端口的技术。如果使用了这种技术，理论限制就是在同一时刻最多有 512个Rlogin客户和同一个的服务器主机相连。

第27章

27.1 当一对插口处于 2MSL等待另一端状态时，理论上不能建立连接。然而，实际上，在18.6节中我们看到大多数伯克利演变的实现确实为一个处于 TIME_WAIT状态的连接接受了一个新的SYN。

27.2 这些行不是以3个数字作为应答代码开始的服务器应答的一部分，因此它们不可能来自于服务器。

第28章

28.1 一个域文字 (domain literal) 是在一对方括号里的点分十进制 IP地址。例如： `mail`

rstevens@[140.252.1.54]。

28.2 6个来回：HELO命令、MAIL、RCPT、DATA报文主体和QUIT。

28.3 这是合法的，称为流水线技术 (pipelining) [Rose 1993，4.4.4节]。不幸的是，有一些脑子坏了（brain-damaged）的SMTP接收者实现，每处理完一条命令就要清除它们的输入缓存，使得这种技术不可用。如果使用了这种技术，客户自然不能丢弃报文直到所有的应答都已检查过，确信报文已被服务器接受了。

28.4 考虑习题28.2的前5个网络上的往返。每个都是一个小命令（很可能是只有一个报文段），对网络几乎没有负载。如果所有 5个都没有重传地送到了服务器，当报文主体发送时，拥塞窗口可能是6个报文段。如果报文主体很大，客户可能一次发送前 6个报文段，造成网络可能来不及处理。

28.5 更新版本的BIND使用同样的值来正移MX记录，作为平衡负载的一种方式。

第29章

29.1 不，因为tcpdump不能从其他UDP数据报中区别RPC请求或应答。它只有在源端口号或目的端口号为2049时，才将UDP数据报的内容理解为NFS分组。随机的RPC请求和应答可以使用两个端点上的一个临时的端口号。

29.2 回忆一下1.9节中，一个进程必须有超级用户权限才能给自己分配一个小于 1024的端口号（一个知名端口）。尽管这对于系统提供的服务器没问题，如 Telnet服务器、FTP服务器、和端口映射器，但我们不想给所有的 RPC服务器提供这个权限。

29.3 这个例子中的两个概念是客户忽略那些服务器应答，如果这些应答不具有客户正在等待的XID；UDP对收到的数据报进行排队（队列长度有一个上限），直到应用进程读取了数据报。另外，XID不会在超时和重传时改变，它只在调用另一个服务器过程时改变。

客户stub执行的事件为：时刻 0：发送请求1；时刻4：超时并重传请求 1；时刻5：接收服务器应答1，将应答返回给应用程序；时刻 5：发送请求2；时刻9：超时并重传请求2；时刻10：收到服务器的应答 1，但因为我们在等待应答 2，所以忽略它；时刻 11：收到服务器的应答2，将应答返回给应用程序。

服务器的事件如下：时刻0：收到请求1，启动操作；时刻5：发送应答1；时刻5：收到请求1（来自于客户在时刻 4的重传），启动操作；时刻 10：发送应答1；时刻10：收到请求2（来自于客户在时刻 5的重传），启动操作；时刻 11：发送应答2；时刻11：收到请求2（来自于客户在时刻 9的重传），启动操作；时刻 12：发送应答2。这个最后的服务器应答仅仅被客户的 UDP排队，用于客户的下一次接收操作。当客户读了这个应答时，XID将是错误的，客户将忽略它。

29.4 改变服务器的以太网卡就改变了它的物理地址。即使我们注意到在 4.7节中SVR4没有在引导时发送一个免费 ARP，它仍然必须在能够应答 sun的NFS请求之前，向sun发送一个请求sun的物理地址的ARP请求。因为sun已经有了svr4的一个ARP登记项，它从这个ARP请求中根据发送者的（新）硬件地址更新这个登记项。

29.5 客户块I/O守护进程的第2个（在偏移 73728处读的）与第1个失去同步大约 0.74秒。即在行131~145，第2个守护进程在第 1个之后超时0.74秒。看来服务器没有看到在 167行的请求，但它看到了 168行的请求。第2个块 I/O守护进程只会在 168行之后0.74秒才会重传。

同时，第1个块I/O守护进程继续发送请求。

29.6 如果使用的是TCP，包含着服务器应答的TCP报文段在网络中丢失了，当服务器的TCP模块没有从客户的TCP模块收到一个确认时，它将超时，并重传应答。最终，这个报文段将到达客户的TCP。这里不同的是两个TCP模块完成超时和重传，而不是NFS客户和服务器（当使用UDP时，NFS客户代码完成超时和重传）。因此，NFS客户并不知道应答丢失了，需要被重传。

29.7 NFS服务器在重启动之后获得一个不同的端口号是可能的。这将使客户变得很复杂，因为它需要知道服务器崩溃了，并且在服务器重启动之后与服务器主机的端口映射器联系以找到NFS服务器的新的端口号。

这种情况，即服务器主机崩溃然后重启动时，一个服务器的RPC应用程序获得一个新的临时性端口，可能发生在任何一个没有使用知名端口的RPC应用程序上。

29.8 不。NFS客户可以为不同的服务器主机使用相同的本地的保留端口号。TCP要求由本地IP地址、本地端口、远端IP地址和远端端口组成的4元组是唯一的，对于每个服务器主机来说，远端的IP地址是不同的。

第30章

30.1 键入`whois "net 88"`。A类网络号64~95是保留的。

30.2 键入`whois whitehouse.gov`可以使用`host`命令或者`nslookup`查询DNS。

30.3 不，`xscope`可以与服务器运行在不同的主机上。如果主机不同，`xscope`也可以使用TCP端口6000作为它的呼入连接。

附录E 配置选项

我们已经看到了许多冠以“依赖于具体配置”的TCP/IP特征。典型的例子包括是否使能UDP的检验和（11.3节），具有同样的网络号但不同的子网号的目的IP地址是本地的还是非本地的（18.4节）以及是否转发直接的广播（12.3节）。实际上，一个特定的TCP/IP实现的许多操作特征都可以被系统管理员修改。

这个附录列举了本书中用到的一些不同的TCP/IP实现可以配置的选项。就像你可能想到的，每个厂商都提供了与其他实现不同的方案。不过，这个附录给出的是不同的实现可以修改的参数类型。一些与实现联系紧密的选项，如内存缓存池的低水平线，没有描述。

这些描述的变量只用于报告的目的。在不同的实现版本中，它们的名字、默认值、或含义都可以改变。所以你必须检查你的厂商的文档（或向他们要更充分的文档）来了解这些变量实际使用的单词。

这个附录没有覆盖每次系统引导时发生的初始化工作：对每个网络接口使用`ifconfig`进行初始化（设置IP地址、子网掩码等等）、往路由表中输入静态路由等等。这个附录集中描述了影响TCP/IP操作的那些配置选项。

E.1 BSD/386版本1.0

这个系统是自从4.2BSD以来使用的“经典”BSD配置的一个例子。因为源代码是和系统一起发布的，所以管理员可以指明配置选项，内核也可重编译。存在两种类型的选项：在内核配置文件中定义的常量（参见`config`(8)手册）和在不同的C源文件中的变量初始化。大胆而又经验丰富的管理员也可以使用排错工具修改正在运行的内核或者内核的磁盘映像中这些变量的值，以避免重新构造内核。

下面列出的是在内核配置文件中可以修改的常量。

IPFORWARDING

这个常量的值初始化内核变量`ipforwarding`。如果值为0（默认），就不转发IP数据报。如果是1，就总是使能转发功能。

GATEWAY

如果定义了这个常量，就使得IPFORWARDING的值被置为1。另外，定义这个常量还使得特定的系统表格（ARP快速缓存表和路由表）更大。

SUBNETSARELOCAL

这个常量的值初始化内核变量`subnetsarelocal`。如果值为1（默认），一个和发送主机具有同样网络号、但不同子网号的目的IP地址被认为是本地的。如果是0，只有在同一个子

网的目的IP地址才认为是本地的。图E-1总结了上述规律。

网络标识符	子网标识符	subnetsarelocal		注释
		1	0	
相同	相同	本地	本地	总是本地的
相同	不同	本地	非本地	依赖于配置
不同		非本地	非本地	总是非本地的

图E-1 对subnetarelocal内核变量的理解

这个变量的值影响了TCP选择的MSS。当给一个本地的目的地址发送报文时，TCP选择的是基于输出接口的MTU的MSS。而发送给一个非本地的地址时，TCP使用变量 `tcp_mssdflt` 作为MSS。

IPSENDREDIRECTS

这个常量的值初始化内核变量 `ipsendredirects`。如果值为1（默认），主机在转发IP数据报时，将发送ICMP重定向。如果是0，不发送ICMP重定向。

DIRECTED_BROADCAST

如果值为1（默认），如果收到的数据报的目的地址是主机的一个接口的直接广播地址，就将它作为一个链路层的广播来转发。如果是0，这些数据报就会被丢弃。

下面的变量也可以改变，它们在目录 /usr/src/sys/netinet中的不同文件中定义。

`tcprexmtthresh`

引起快速重传和快速恢复算法的连续ACK的数目。默认值是3。

`tcp_ttl`

TCP段的TTL字段的默认值。默认值是60。

`tcp_mssdflt`

用于非本地目的地址的默认的TCP MSS。默认值是512。

tcp_keepidle

在发送一个`keepalive`探测报文之前必须等待的500 ms时钟间隔的次数。默认值是14 400（2个小时）。

`tcp_keepintvl`

如果没有收到响应，在两个连续的`keepalive`探测报文之间等待的500 ms时钟间隔的次数。默认值是150（75秒）。

`tcp_sendspace`

TCP发送缓存的默认大小。默认值是4096。

`tcp_recvspace`

TCP接收缓存的默认大小。这个值影响了提供的窗口大小。默认值是4096。

`udpcksum`

如果非0，对输出的UDP数据报计算UDP检验和，并且对于包含了非0检验和的输入UDP数据报要验证它们的检验和。如果值为0，不计算输出的UDP数据报的检验和，也不验证输入UDP数据报的检验和，即使发送者计算了一个检验和。默认值是1。

udp_ttl
UDP数据报TTL字段的默认值。默认值是30。
udp_sendspace
UDP发送缓存的默认大小。定义了可以发送最大的UDP数据报。默认值是9126。
udp_recvspace
UDP接收缓存的默认大小。默认值是41 600，允许40个1024字节的数据报。

E.2 SunOS 4.1.3

SunOS 4.1.3使用的方法类似于我们在BSD/386中看到的。因为大部分的内核源代码都没有发布，所以所有的C变量初始化都包含在一个提供的C源文件中。

管理员的内核配置文件（参见config(8)手册）可以定义下面的变量。修改了配置文件之后，需要构造一个新的内核，然后重启动。

IPFORWARDING

这个常量的值初始化内核变量ip_forwarding。如果值为-1，就不转发IP数据报，而且变量的值不能再改变。如果是0（默认），不转发IP数据报，但是如果多个接口都工作，变量的值可以修改为1。如果是1，就总是能转发IP数据报。

SUBNETSARELOCAL

这个常量的值初始化内核变量ip_subnetsarelocal。如果值为1（默认），一个和发送主机具有同样网络号，但不同子网号的目的IP地址被认为是本地的。如果是0，只有在同一个子网的目的IP地址才认为是本地的。图E-1总结了上述规律。当给一个本地的目的地址发送报文时，TCP选择的是基于输出接口的MTU的MSS，而发送给一个非本地的地址时，TCP使用变量tcp_default_mss作为MSS。

IPSENDREDIRECTS

这个常量的值初始化内核变量ip_sendredirects。如果值为1（默认），主机在转发IP数据报时，将发送ICMP重定向。如果是0，不发送ICMP重定向。

DIRECTED_BROADCAST

这个常量的值初始化内核变量ip_dirbroadcast。如果值为1（默认），如果收到的数据报的目的地址是主机的一个接口的直接广播地址，就将它作为一个链路层的广播来转发。如果是0，这些数据报就会被丢弃。

文件/usr/kvm/sys/netinet/in_proto.c定义了下面一些可以修改的变量。一旦修改了这些变量，必须构造一个新的内核，然后重启动。

tcp_default_mss
用于非本地地址的默认TCP MSS。默认值是512。
tcp_sendspace
TCP发送缓存的默认大小。默认值是4096。

`tcp_recvspace`

TCP接收缓存的默认大小。这个值影响了提供的窗口大小。默认值是 4096。

`tcp_keeplen`

一个发往4.2BSD主机的keepalive探测报文必须包含一个字节的数据来得到一个响应。把这个变量的值设置为1是为了兼容于以前的实现。默认值是 1。

`tcp_ttl`

TCP段的TTL字段的默认值。默认值是60。

`tcp_nodelack`

如果非0，对ACK不做延迟。默认值是0。

`tcp_keepidle`

在发送一个keepalive探测报文之前必须等待的 500 ms时钟间隔的次数。默认值是 14 400（2个小时）。

`tcp_keepintvl`

如果没有收到响应，在两个连续的 keepalive探测报文之间等待的500 ms时钟间隔的次数。默认值是150（75秒）。

`udp_cksum`

如果非0，对输出的UDP数据报计算UDP检验和，并且对于包含了非 0检验和的输入UDP数据报要验证它们的检验和。如果值为 0，不计算输出UDP数据报的检验和，也不验证输入UDP数据报的检验和，即使发送者计算了一个检验和。默认值是 0。

`udp_ttl`

UDP数据报TTL字段的默认值。默认值是60。

`udp_sendspace`

UDP发送缓存的默认大小。定义了可以发送最大的 UDP数据报。默认值是9000。

`udp_recvspace`

UDP接收缓存的默认大小。默认值是 18 000，允许两个9000字节的数据报。

E.3 SRV4

SVR4的TCP/IP配置类似于前两个系统，但可用的选项更少。在文件`/etc/conf/pack.d/ip/space.c`5可以定义两个常量，然后必须重新构造内核并且重启动。

IPFORWARDING

这个常量的值初始化内核变量`ipforwarding`。如果是0（默认），不转发IP数据报。如果是1，就总是能转发IP数据报。

IPSENDREDIRECTS

这个常量的值初始化内核变量`ipsendredirects`。如果值为1（默认），主机在转发IP数据报时，将发送ICMP重定向。如果是0，不发送ICMP重定向。

前两节中，我们描述的许多变量在内核中都有定义，但必须修补内核来改变它们。例如，存在一个名为`tcp_keepidle`的变量，它的值是14 400。

E.4 Solaris 2.2

Solaris 2.2是较新的Unix系统的典型代表，它为管理员提供了一个可以改变 TCP/IP系统配置选项的程序。这样可以不必通过修改源文件和重新构造内核来进行配置。

配置程序是ndd(1)。我们可以运行程序，看看在UDP模块中可以检验和修改的参数：

```
solaris % ndd /dev/udp \?
udp_wroff_extra                 读、写
udp_def_ttl                     读、写
udp_first_anon_port             读、写
udp_trust_optlen                读、写
udp_do_checksum                 读、写
udp_status                      只读
```

我们可以指明 5个模块：/dev/ip、/dev/icmp、/dev/arp、/dev/udp和/dev/tcp。问号参数（为了防止外壳程序解释问号，我们在它前面加了一个反斜线）告诉ndd程序列出那个模块的所有参数。查询一个变量的值的例子是：

```
solaris %ndd /dev/tcp tcp_mss_def
536
```

为了修改一个变量的值，我们需要有超级用户的权限，输入：

```
solaris #ndd -set /dev/ip ip_forwarding 0
```

这些变量可以划分为三种类型：

1) 系统管理员可以修改的配置变量（如， ip_forwarding）。

2) 只能显示的状态变量（如， ARP快速缓存）。这个信息一般通过命令 ifconfig，netstat和arp以一种更好理解的格式提供。

3) 用于内核源代码的排错变量。使能一些这种变量可以在运行时产生内核的排错输出，当然这会降低系统的性能。

现在我们可以描述每个模块的参数了。所有的参数如果没有注明“（只读）”，就是可读写的。只读的参数是上面第 2种情况的状态变量。我们对于第 3种情况的变量注明了“（排错）”。如果不另外说明，所有的计时变量都以毫秒指明，这和其他系统不同，其他系统一般以 500 ms时钟间隔的次数来指明时间。

/dev/ip

ip_cksum_choice

（排错）在IP检验和算法的两个独立实现之中选择一个。

ip_debug

（排错）如果大于0，使能内核打印排错信息功能。值越大输出的信息越多。默认为 0。

ip_def_ttl

如果运输层没有指明，指定输出IP数据报默认的TTL。默认值是255。

ip_forward_directed_broadcasts

如果值为1（默认），如果收到的数据报的目的地址是主机的一个接口的直接广播地址，就将它作为一个链路层的广播来转发。如果是 0，这些数据报就会被丢弃。

ip_forward_src_routed

如果为1（默认），就转发包含一个源路由选项的接收数据报。如果为 0，这些数据报将被

丢弃。

ip_forwarding

指明系统是否转发进入的IP数据报：0表示不转发，1表示总是转发，2（默认）表示只有当两个或两个以上接口都工作时才转发。

ip_icmp_return_data_bytes

一个ICMP差错返回的除了IP首部以外的数据字节的数目，默认是64。

ip_ignore_delete_time

（排错）一个IP路由表项（IRE）最小的生命期。默认是30秒（这个参数以秒记，不是毫秒）。

ip_ill_status

（只读）显示每个IP下层数据结构的状态。每个接口存在一个下层数据结构。

ip_ipif_status

（只读）显示每个IP接口数据结构的状态（IP地址、子网掩码等等）。每个接口存在一个这种结构。

ip_ire_cleanup_interval

（排错）扫描IP路由表，删除过时表项的时间间隔。默认是30 000 ms（30秒）。

ip_ire_flush_interval

从IP路由表中无条件地刷新ARP信息的间隔。默认是1200 000 ms（20分钟）。

ip_ire_pathmtu_interval

路径MTU发现算法尝试增加MTU的间隔。默认是30 000 ms（30秒）。

ip_ire_redirect_interval

来自ICMP重定向的IP路由表项被删除的间隔。默认是60 000 ms（60秒）。

ip_ire_status

（只读）显示所有的IP路由表项。

ip_local_cksum

如果为0（默认），IP不为通过环回接口发送和接收的数据报计算IP检验和或者更高层的检验和（即TCP、UDP、ICMP或IGMP）。如果为1，就要计算这些检验和。

ip_mrtdebug

（排错）如果为1，使能内核打印多播路由的排错输出。默认是0。

ip_path_mtu_discovery

如果为1（默认），IP执行路径MTU发现。如果是0，IP不会在输出的数据报中设置“不分片”比特。

ip_respond_to_address_mask

如果为0（默认），主机不响应ICMP的地址掩码请求。如果为1，主机则响应。

ip_respond_to_echo_broadcast

如果为1（默认），主机响应发往一个广播地址的ICMP回显请求。如果为0，则不响应。

ip_respond_to_timestamp

如果为0（默认），主机不响应ICMP的时间戳请求。如果为1，则响应。

ip_respond_to_timestamp_broadcast

如果为0（默认），主机不响应发往一个广播地址的ICMP时间戳请求。如果为1，则响应。

ip_rput_pullups

（排错）来自于网络接口驱动程序的缓存数目的计数，它需要增长以访问整个 IP首部。引导时它被初始化为0，并且可以被复位为0。

ip_send_redirects

如果为1（默认），当主机作为一个路由器时，它发送ICMP重定向。如果为0，则不发送。

ip_send_source_quench

如果为1（默认），当输入的数据报被丢弃时，主机生成 ICMP源抑制差错。如果为 0，则不生成这种差错。

ip_wroff_extra

（排错）在缓存中为IP首部分配的额外空间的字节数。默认是 32。

/dev/icmp

icmp_bsd_compat

（排错）如果为1（默认），收到的数据报的IP首部的长度字段的值被调整为不包括 IP首部的长度。这和伯克利演变的实现是一致的，用于读原始的 IP或原始的ICMP分组的应用程序。如果为0，则不改变长度字段的值。

icmp_def_ttl

输出ICMP报文的默认的TTL。默认值为255。

icmp_wroff_extra

（排错）在缓存中为IP选项和数据链路首部所分配的额外空间的字节数。默认是 32。

/dev/arp

arp_cache_report

（只读）ARP的快速缓存。

arp_cleanup_interval

ARP登记项从ARP快速缓存中被删除的时间间隔。默认是 300 000 ms（5分钟）（IP为完成的ARP传输维护着它自己的快速缓存；参见 ip_ire_flush_interval）。

arp_debug

（排错）如果为1，使能打印ARP驱动程序的排错输出。默认是 0。

/dev/udp

udp_def_ttl

输出UDP数据报的默认的TTL。默认值是255。

udp_do_checksum

如果为1（默认），为输出的UDP数据报计算UDP检验和。如果为0，输出的UDP数据报不包含一个检验和（和其他大多数的实现不一样，这个 UDP检验和标志并不影响进入的数据报。如果一个接收到的数据报有一个非 0的检验和，它总是要被验证）。

udp_largest_anon_port

可以为UDP临时端口分配的最大端口号。默认是 65535。

`udp_smallest_anon_port`
可以为UDP临时端口分配的最小端口号。默认是32768。

`udp_smallest_nonpriv_port`
一个进程需要超级用户的权限才能给自己分配一个小于这个值的端口号。默认是 1024。

`udp_status`
（只读）所有本地的UDP端点的状态：本地IP地址和端口，远端IP地址和端口。

`udp_trust_optlen`
（排错）不再使用。

`udp_wroff_extra`
（排错）在缓存中为IP选项和数据链路首部所分配的额外空间的字节数。默认是 32。

/dev/tcp

`tcp_close_wait_interval`
2MSL的值：在TIME_WAIT状态花费的时间。默认是240 000 ms（4分钟）。

`tcp_conn_grace_period`
（排错）当发送一个SYN时，在定时器间隔上附加的时间。默认是 500 ms。

`tcp_conn_req_max`
在一个监听的端口上挂起的连接请求的最大数目。默认是 5。

`tcp_cwnd_max`
拥塞窗口的最大值。默认是32768。

`tcp_debug`
（排错）如果为1，使能打印TCP的排错输出。默认是0。

`tcp_deferred_ack_interval`
在发送一个延迟的ACK之前等待的时间。默认是50 ms。

`tcp_dupack_fast_retransmit`
引起快速重传、快速恢复算法的连续的重复 ACK的数目。默认是3。

`tcp_eager_listeners`
（排错）如果为1（默认），TCP在将一个新的连接返回给一个挂起的被动打开的应用程序之前需要进行三次握手。这是大多数的 TCP实现采用的方式。如果为 0，TCP将呼入连接请求（收到的SYN）传递给应用程序，并不完成三次握手直到该应用程序接受了这个连接（把这个值置为0可能引起很多已经存在的应用程序不能用）。

`tcp_ignore_path_mtu`
（排错）如果为1，路径MTU发现算法忽略接收到的需要 ICMP分段的报文。如果为0（默认），使能TCP的路径MTU发现。

`tcp_ip_abort_cinterval`
当TCP进行一个主动打开时，整个重传超时的值。默认是 240 000 ms（4分钟）。

`tcp_ip_abort_interval`
一个TCP连接建立以后，整个重传超时的值。默认是 120 000 ms（2分钟）。

`tcp_ip_notify_cinterval`

当TCP正在进行一个主动打开时，TCP通知IP去寻找一条新路由超时的值。默认是10 000 ms（10秒）。

`tcp_ip_notify_interval`

TCP为一个已经建立的连接通知IP去寻找一条新路由超时的值。默认是10 000 ms（10秒）。

`tcp_ip_ttl`

用于输出TCP段的TTL。默认为255。

`tcp_keepalive_interval`

在发出一个keepalive探测报文之前，一个连接保持空闲状态的时间。默认为 7200000 ms（2小时）。

`tcp_largest_anon_port`

为TCP临时端口分配的最大端口号。默认为 65535。

`tcp_maxpsz_multiplier`

（排错）指明了报文流首部将应用程序写的数据分装成几个 MSS。默认是1。

`tcp_mss_def`

非本地的目的地址的默认的MSS。默认是536。

`tcp_mss_max`

最大的MSS。默认为65495。

`tcp_mss_min`

最小的MSS。默认为1。

`tcp_naglim_def`

（排错）每个连接的Nagle算法阈值的最大值。默认是 65535。每个连接的值以MSS的最小值或这个值开始。TCP_NODELAY插口选项将每个连接的值设置为 1，以禁止Nagle算法。

`tcp_old_urp_interpretation`

（排错）如果为1（默认），采用紧急指针的一个以前的（但更常见的） BSD的理解：它指向紧急数据最后一个字节后的一个字节。如果为 0，采用主机需求RFC理解：它指向紧急数据的最后一个字节。

`tcp_rcv_push_wait`

（排错）在把接收数据传递给应用程序之前，可以缓存的没有设置 PUSH标志的数据的最大字节数。默认是16384。

`tcp_rexmit_interval_initial`

（排错）初始的重传超时间隔。默认是 500 ms。

`tcp_rexmit_interval_max`

（排错）最大的重传超时间隔。默认是 60 000 ms（60秒）。

`tcp_rexmit_interval_min`

（排错）最小的重传超时间隔。默认是 200 ms。

`tcp_rwin_credit_pct`

（排错）在对每个接收的段进行流量控制检查之前，必须达到的接收缓存窗口的百分比。默认是50%。

`tcp_smallest_anon_port`

分配给TCP临时端口的开始端口号。默认是32768。

`tcp_smallest_nonpriv_port`

一个进程需要有超级用户的权限才能给自己分配一个小于这个值的端口号。默认是1024。

`tcp_snd_lowat_fraction`

（排错）如果非0，发送缓存的低水平线是发送缓存的大小除以这个值。默认是0（禁止）。

`tcp_status`

（只读）所有TCP连接的信息。

`tcp_sth_rcv_hiwat`

（排错）如果非0，把报文流首部的高水平线设置为这个值。默认为0。

`tcp_sth_rcv_lowat`

（排错）如果非0，把报文流首部的低水平线设置为这个值。默认为0。

`tcp_wroff_xtra`

（排错）在缓存中为IP选项和数据链路首部所分配的额外空间的字节数。默认是32。

E.5 AIX 3.2.2

AIX3.2.2允许在运行时使用no命令设置网络选项。它可以显示一个选项的值，设置一个选项的值，或者将一个选项的值设置为默认。例如，显示一个选项，我们键入：

```
aix % no -o udp_ttl
udp_ttl = 30
```

下面的选项可以被修改。

`arpt_killc`

在删除一个不活动的、完成的ARP项之前等待的时间（以分钟计）。默认是20。

`ipforwarding`

如果为1（默认），总是转发IP数据报。如果为0，则禁止转发。

`ipfragttl`

等待重新装配的IP数据报片的生存时间（单位为秒）。默认是60。

`ipsendredirects`

如果为1（默认），当转发IP数据报时，主机将发送ICMP重定向。如果为0，则不发送ICMP重定向。

`loop_check_sum`

如果为1（默认），对通过环回接口发送的数据报计算IP检验和。如果为0，则不计算这个检验和。

`nonlocsrcroute`

如果为1（默认），就转发包含一个源路由选项的接收数据报。如果为0，就丢弃这些数据报。

`subnetsarelocal`

如果值为1（默认），一个和发送主机具有同样网络号，但不同子网号的目的IP地址被认为是本地的。如果是0，只有在同一个子网的目的IP地址才认为是本地的。图E-1总结了上述规律。当给一个本地的目的地址发送报文时，TCP选择的是基于输出接口的MTU的MSS。当发送给一个非本地的地址时，TCP使用默认（536）作为MSS。

`tcp_keepidle`

在发送一个keepalive探测报文之前等待的500 ms时间段的倍数。默认值是14 400（2小时）。

`tcp_keepintvl`

如果没有收到响应，在发送下一个 keepalive探测报文之前等待的 500 ms时间段的倍数。默认值是150（75秒）。

`tcp_recvspace`

TCP接收缓存的默认大小。它影响了提供的窗口大小。默认值是 16 384。

`tcp_sendspace`

TCP发送缓存的默认大小。默认是 16 384。

`tcp_ttl`

TCP报文段TTL字段的默认值。默认值是60。

`udp_recvspace`

UDP接收缓存的默认大小。默认值是 41 600，允许40个1024字节数据报。

`udp_sendspace`

UDP发送缓存的默认大小。定义了可以发送的最大的 UDP数据报。默认是9216。

`udp_ttl`

UDP数据报TTL字段的默认值。默认值是30。

E.6 4.4BSD

4.4BSD是第一个为多个内核参数提供动态配置的伯克利版本。使用 sysctl（8）命令。参数的名字看起来就像SNMP的MIB变量的名字。查看一个参数，我们键入：

```
vangogh %sysctl net.inet.ip.forwarding
net.inet.ip.forwarding = 1
```

要修改一个参数的值需要有超级用户的权限，键入：

```
vangogh #sysctl -w net.inet.ip.ttl=128
```

可以修改下面的参数。

`net.inet.ip.forwarding`

如果为0（默认），就不转发IP数据报。如果为1，则使能转发功能。

`net.inet.ip.redirect`

如果为1（默认），当转发 IP数据报时，主机将发送 ICMP重定向。如果为 0，则不发送ICMP重定向。

`net.inet.ip.ttl`

TCP和UDP默认的TTL。默认值是64。

`net.inet.icmp.maskrepl`

如果为0（默认），主机不响应ICMP地址掩码请求。如果为1，则响应。

`net.inet.udp.checksum`

如果为1（默认），对输出的UDP数据报计算UDP检验和，并且对于包含了非0检验和的呼入UDP数据报要校验它们的检验和。如果值为 0，不计算输出UDP数据报的检验和，也不验证输入UDP数据报的检验和，即使发送者计算了一个检验和。

另外，在本附录前面部分描述的许多变量分散在几个不同的源文件中（`tcp_keepidle`、`subnetarelocal`等等），它们也可以被修改。

附录F　可以免费获得的源代码

本书中使用了很多共享软件包。本附录提供了一些如何获得这些软件的细节。

用来获得共享软件的技术称为匿名 FTP，FTP是标准的Internet文件传输协议（第 27章）。27.3节显示了一个匿名FTP的例子。如果想了解Internet资源的一般背景和匿名FTP的知识，请参考任何一本关于Internet的书，如[LaQuey 1993] 或 [Krol 1992]。

这里列出的主机被认为是提供共享软件的主要站点。其他许多站点也提供这些软件。可以通过Internet的Archie服务查找其他的软件版本。下面列出的软件版本是本书中使用到的版本。

当你读到这本书时，可能已经发布了更新版本的软件。

你应该使用FTP的dir命令来看一下在指定的主机上是否存在更新的版本。

本附录按照资源在本书中出现的章节号进行排序。

RFC（1.11节）

1.11节提供了请求得到RFC站点的电子邮件地址。应答中包含了许多可以使用电子邮件或匿名FTP获得RFC的站点。

记住你要先得到一个当前的索引，在索引中查找想要的 RFC。这个RFC项中还告诉你这个RFC是否废弃了或是被一个更新的RFC替代了。

BSD Net/2 源代码（1.14节）

BSD Net/2 源代码，其中包括了TCP/IP协议的内核实现和标准的应用程序（Telnet客户和服务器、FTP客户和服务器等），从主机`ftp.uu.net`的目录树中以`system/unix/bsd-source`开始的位置可以得到。

SLIP（2.4节）

本书中使用的SLIP的版本来自于`ftp.ee.lbl.gov`。文件名以`cslip`开始，因为它提供了压缩的SLIP（2.5节）。

icmpaddrmask程序（6.3节）

参见本附录的最后一项。

icmptime程序（6.4节）

参见本附录的最后一项。

ping程序（第7章）

BSD版本的`ping`程序一般比其他厂商提供的版本具有更多的选项和特征。主机`ftp.uu.net`

的文件system/unix/bsd-sources/sbin/ping中包含了最新版本的ping程序。

traceroute程序（第8章）

traceroute程序来自于主机ftp.ee.lbl.gov。本附录的最后一项提供了8.5节使用的允许不严格的和严格的源站选路的版本。

路由器发现守护程序（9.6节）

可用的一个程序为路由器发现报文提供了主机支持和路由器支持。主机是gregorio.stanford.edu，文件是gw-discovery/nordmark-rdisc.tar。这个程序是Sun微系统公司开发的一个共享软件。

gated守护程序（10.3节）

在10.3节提到的gated路由选择守护程序在主机gated.cornell.edu上可以得到。

traceroute.pmtu程序（11.7节）

参见本附录的最后一项。

IP多播软件（第13章）

为SunOS 4.x和Ultrix提供IP多播的修改程序在主机gregorio.stanford.edu的目录vmtp-ip中可以得到。这个目录中还包含了为伯克利Unix系统提供IP多播的修改源程序。

BIND名字服务器（第14章）

BIND名字服务器，即named守护程序，在主机ftp.uu.net的文件networking/ip/dns/bind/bind.4.8.3.tar.z中。

一个更新的版本，即4.9版，从主机gatekeeper.dec.com的目录pub/BSD/bind/4.9中可以得到。

host程序（第14章）

host程序在主机nikhefh.nikhef.nl的文件host.tar.z中提供。

dig和doc程序（第14章）

第14章中提到的dig程序和doc程序在主机isi.edu的文件dig.2.0.tar.Z和doc.2.0.tar.Z中提供。

BOOTP服务器（第26章）

常用的Unix BOOTP服务器的不同版本在主机lancaster.andrew.cmu.edu的pub目录中提供。

TCP快速扩充（第24章）

TCP窗口扩缩选项、时间戳选项和PAWS算法的一个共享源代码作为BSD Net/2版的一组

修改程序在主机uxc.cso.uiuc.edu的文件pub/tcplw.shar.z中提供。

ISODE SNMP管理进程和代理进程（第25章）

25.7节中描述的SNMP管理者进程和代理进程是ISODE 8.0版本的一部分。很多站点都可以得到它，如ftp.uu.net上的目录networking/osi/isode中。

MIME软件和实例（28.4节）

一个名为MetaMail的程序为很多不同的用户代理提供了MIME的能力。这个程序可以从主机thumper.bellcore.com的pub/nsb目录中得到。在这个目录中还有MIME的其他信息。

Sun RPC（29.2节）

RPC 4.0版本的源代码（使用插口API）在主机ftp.uu.net的目录systems/sun/sextape/rpc4.0中提供。TI-RPC版本的源代码（使用了TLI API）在主机ftp.uu.net的目录networking/rpc中提供。

Sun NFS（第29章）

一个NFS客户和服务器的共享软件的实现作为BSD Net/2源代码的一部分提供，在本附录的前面部分介绍了。

tcpdump程序（附录A）

本书中使用的tcpdump的版本来自于主机ftp.ee.lbl.gov上的文件tcpdump-2.2.1.tar.z。

BSD分组过滤器（A.1节）

BSD分组过滤器是tcpdump发布的一部分。

sock程序（附录C）

参见本附录的最后一项。

ttcp程序

（这个程序本书中没有用到，但它是一个读者应该注意的常用的工具）。ttcp是测量两个系统之间TCP和UDP性能的基准工具。它是美国军事弹道研究实验室开发的，是一个共享软件。它的副本可以从很多匿名FTP站点获得，但一个增强的版本可以从主机ftp.sgi.com的目录sgi/src/ttcp中获得。

作者编写的软件

本书中用到的作者编写的软件在主机ftp.uu.net的文件published/books/stevens.tcpipiv1.tar.z中提供。

参 考 文 献

所有的RFC都可以如同在1.11节描述的那样在Internet上通过电子邮件或使用匿名 FTP免费获得。

Albitz, P., and Liu, C. 1992. *DNS and BIND*. O'Reilly & Associates, Sebastopol, Calif.
配置和运行一个名字服务器所需要的管理任务的许多细节。

Alexander, S., and Droms, R. 1993. "DHCP Operations and BOOTP Vendor Extensions," RFC1533, 30 pages(Oct.).

Almquist, P. 1992. "Type of Service in the Internet Protocol Suite," RFC1349, 28 pages (July).
怎样使用IP首部的服务类型字段。

Almquist, P., ed. 1993. "Requirements for IP Routers," Internet Draft (Mar.).
这是代替RFC 1009[Braden and Postel 1987]的一个RFC的草稿。新的RFC将很可能以四卷出现。卷1：Internet的结构、术语和一般性内容。卷 2：链路层、互联网层、运输层和应用层。卷3：转发和选路协议。卷4：操作、维护和网络管理。
这个草稿可以通过匿名 FTP从主机jessica.stanford.edu的目录rreq中获得。一旦最终的RFC发布就要忽略这个草稿。

Bellovin, S. M. 1993. Private Communication.

Bhide, A., Elnozahy, E. N., and Morgan, S. P. 1991. "A Highly Available Network File Server," *Proceeding of the 1991 Winter USENIX Conference*, pp. 199-205, Dallas, Tex.
描述了一个对免费ARP的使用（4.7节）。

Borenstein, N., and Freed, N. 1993. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," RFC 1521, 81 pages (Sept.).
这个RFC废止了早期的RFC 1341。这个RFC的附录H列出了与RFC 1341的不同点。

Borman, D. A., ed. 1990. "Telnet Linemode Option," RFC 1184, 23 pages (Oct.).

Borman, D. A. 1991. "IP Bandwidth Limits," Message-ID 91011437.AA17276@berserkly.cray.com>, Usenet, comp.protocols.tcp-ip Newsgroup (Jan.).
说明了我们在24.8节的最后部分列出的有关 TCP性能的三个限制。

Borman, D. A. 1992. "TCP/IP Performance at Cray Research," *Proceeding of the Twenty-third Internet Engineering Task Force*, pp. 492 - 493 (Mar.), San Diego Supercomputer Center, San Diego, Calif.

Borman, D. A., ed. 1993a. "Telnet Environment Option," RFC 1408, 7 pages (Jan.).
从客户向服务器传递环境变量的 Telnet选项。

Borman, D. A. 1993b. " A Practical Perspective on Host Networking," in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 309-367. Addison-Wesley, Reading, Mass.
对于Host Requirements（主机需求）RFC（1122和1123）的一个实际的审视。

Braden, R. T., ed. 1989a. “Requirements for Internet Hosts —— Communication Layers,” RFC1122, 116 pages (Oct.).
主机需求RFC的前半部分。这部分覆盖了链路层， IP，TCP和UDP。
Braden, R. T., ed. 1989b. “Requirements for Internet Hosts —— Application and Support,” RFC1123, 98 pages (Oct.).
主机需求RFC的后半部分。这部分覆盖了Telnet，FTP，TFTP，SMTP和DNS。
Braden, R. T. 1989c. “Perspective on the Host Requirements RFCs,” RFC 1127, 20 pages (Oct.).
对于开发主机需求RFC的IETF工作组的讨论和结果的一个非正式的总结。
Braden, R. T. 1992a. “TIME-WAIT Assassination Hazards in TCP,” RFC1337, 11 pages (May).
显示处于TIME_WAIT状态时，接收一个RST如何导致问题。
Braden, R. T. 1993b. “Extending TCP for Transactions —— Concepts,” RFC 1379, 38 pages (Nov.).
开发T/TCP背后的概念和历史。
Braden, R. T. 1992c. “Extending TCP for Transactions —— Functional Specification,” Internet Draft, 32 pages (Dec.).
T/TCP的功能说明和有关实现的讨论。
Braden, R. T., Borman, D. A., and Partridge, C. 1988. “Computing the Internet Checksum,” RFC 1071, 24 pages (Sept.).
提供计算IP、ICMP、IGMP、UDP和TCP的检验和的技术和算法。
Braden, R. T., and Postel, J. B. 1987. “Requirements for Internet Gateways,” RFC1009, 55 pages (June).
等效于路由器的主机需求RFC。这个RFC已经被代替了；见 [Almquist 1993]。
Caceres, R., Danzig, P. B., Jamin, S., and Mitzel, D. J. 1991. “Characteristics of Wide-Area TCP/IP Conversations,” *Computer Communication Review*, vol. 21, no. 4, pp. 101-112 (Sept.).
Callon, R. 1992. “TCP and UDP with Bigger Addresses (TUBA), A Simple Proposal for Internet Addressing and Routing,” RFC 1347, 9 pages (June).
Case, J. D., Fedor, M. S., Schoffstall, M. L., and Davin, C. 1990. “Simple Network Management (SNMP),” RFC 1157, 36 pages (May).
SNMP的协议规范。
Case, J. D., McCloghrie, K., Rose, M. T., and Waldbusser, S. 1993. “An Introduction to Version 2 of the Internet-Standard Network Management Framework,” RFC 1441, 13 pages (Apr.).
一个SNMPv2的介绍和其他11个定义SNMPv2的RFC的引用。
Case, J. D., and Partridge, C. 1989. “Case Diagrams: A First Step to Diagrammed Management Information Bases,” *Computer Communication Review*, vol. 19, no. 1, pp. 13-16 (Jan.).
定义了用来可视化一个给定模块中SNMP变量之间关系的图形表示。
Casner, S., and Deering, S. E. 1992. “First IETF Internet Audiocast,” *Computer Communication Review*, vol. 22, no. 3, pp. 92-97 (July).

描述了怎样使用Internet的多播技术实时传递一个IETF会议的声音信息。这篇文章的一个PostScript的 副本可以通过匿名FTP从主机venera.isi.edu上的文件pub/ietf-autocast-article.ps中获得。另外，那台主机上的文件 mbone/faq.txt包含了有关Internet多播骨干（MBONE）常问的问题。

Cheriton, D. P. 1988. “VMTP: Versatile Message Transaction Protocol,” RFC 1045, 123 pages (Feb.).

Cheswick, W. R., and Bellovin, S. M. 1994. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, Reading, Mass.

描述了怎样建立和管理一个防火墙网关以及涉及的安全问题。

Clark, D. D. 1982. “Window and Acknowledgement Strategy in TCP,” RFC 813, 22 pages (July).

标识了糊涂窗口综合症以及如何避免它的原始RFC。

Clark, D. D. 1988. “ The Design Philosophy of the DARPA Internet Protocols,” *Computer Communication Review*, vol. 18, no. 4, pp. 106-114 (Aug.).

描述了影响Internet协议的早期的推理技术。

Comer, D. E., and Stevens, D. L. 1993. *Internetworking with TCP/IP: Vol. III: Client-Server Programming and Applications, BSD Socket Version*. Prentice-Hall, Englewood Cliffs, N.J.

Cooper, A. W., and Postel, J. B. 1993. “ The US Doman,” RFC 1480, 47 pages (June).

描述了DNS的.us域。

Crocker, D. H. 1982. “Standard for the Format of ARPA Internet Text Messages,” RFC 822, 47 pages (Aug.).

定义了使用SMTP传输的电子邮件的格式。

Crocker, D. H. 1993. “Evolving the System,” in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 41-76. Addison-Wesley, Reading, Mass.

讲述了开发ARPANET标准的一些历史和Internet技术共同体当前结构的细节，还定义了当前Internet标准形成的过程。

Croft, W., and Gilmore, J. 1985. “Bootstrap Protocol (BOOTP),” RFC 951, 12 pages (Sept.).

Crowcroft, J., Wakeman, I., Wang, Z., and Sirovica, D. 1992. “Is Layering Harmful?,” *IEEE Network*, vol. 6, no. 1, pp. 20-24 (Jan.).

这篇文章中漏掉的7张图刊登在下一期，卷6，第2期（三月）。

Curry, D. A. 1992. *UN LX System Security: A Guide for Users and System Administrators*. Addison-Wesley, Reading, Mass.

一本关于Unix安全性的书。第4章和第5章讲述网络安全。

Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., and Lumley, J. 1993. “After-burner,” *IEEE Network*, vol. 7, no. 4, pp. 36-43 (July).

描述了怎样通过减少数据复制的次数来加快TCP，以及一个支持这种设计的专用接口卡。

Danzig, P. B., Obraczka, K., and Kumar, A. 1992. “An Analysis of Wide-Area Name Server Traffic,” *Computer Communication Review*, vol. 22, no. 4, pp. 281-292 (Oct.).

对根名字服务器之一进行的两个 24小时流量监视的分析。显示来自于一个故障实现的

DNS流量如何消耗了20倍于正常的WAN带宽。这篇文章的一个PostScript副本通过匿名FTP在主机**caldera.usc.edu**的文件**pub/danzig/dns.ps.z**中提供。

Deering, S. E. 1989. “Host Extensions for IP Multicasting,” RFC 1112, 17 pages (Aug.).
IP多播和IGMP的规范。

Deering, S. E., ed. 1991. “ICMP Router Discovery Messages,” RFC 1256, 19 pages (Sept.).

Deering, S. E., and Cheriton, D. P. 1990. “Multicast Routing in Datagram Internetworks and Extended LANs,” *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85-110 (May).
对于常用路由技术的支持多播传输的扩充。

Dixon, T. 1993. “Comparison of Proposals for Next Version of IP,” RFC 1454, 15 pages (May).
SIP, PIP 和TUBA的比较和总结。

Droms, R. 1993. “Dynamic Host Configuration Protocol,” RFC 1541, 39 pages (Oct.).

Droms, R., and Dyksen, W. R. 1990. “Performance Measurements of the X Window System Communication Protocol,” *Software Practice & Experience*, vol. 20, pp. 119-136 (Oct.).
使用不同的X客户时涉及到的TCP通信的测量。

Fedor, M. S. 1988. “GATED: A Multi-routing Protocol Daemon for Unix,” *Proceeding of the 1988 Summer USENIX Conference*, pp. 365-376, San Francisco, Calif.

Finlayson, R. 1984. “Bootstrap Loading using TFTP,” RFC 906, 4 pages (June).

Finlayson, R., Mann, T., Mogul, J. C., and Theimer, M. 1984. “ A Reverse Address Resolution Protocol,” RFC 903, 4 pages (June).

Floyd, S. 1994. Private Communication.

Ford, P. S., Rekhter, Y., and Braun, H-W. 1993. “Improving the Routing and Addressing of IP,” *IEEE Network*, vol. 7, no. 3, pp. 10-15 (May).
对于CIDR（无类型域间路由）的描述。

Fuller, V., Li, T., Yu, J.Y., and Varadhan, K. 1993. “Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy,” RFC 1519, 24 pages (Sept.).
CIDR（无类型域间选路）的规范。

Gerich, E. 1993. “Guidelines for Management of IP Address Space,” RFC 1466, 10 pages (May).
将来怎样分配IP地址的说明（即B类地址将很难获得，作为替代手段，一般将分配一组C类地址）。

Gurwitz, R., and Hinden, R. 1982. “IP —— Local Area Network Addressing Issues,” IEN 212, 11 pages (Sept.)。
对IP广播地址最早的引用之一。

Harrenstein, K., Stahl, M. K., and Feinler, E. J. 1985. “NICNAME/WHOIS,” RFC 954, 4 pages (OCT.).

Hedrick, C. L. 1988a. “Routing Information Protocol,” RFC 1058, 33 pages (June).

Hedrick, C. L. 1988b. “Telnet Terminal Speed Option,” RFC 1079, 3 pages (Dec.).

Hedrick, C. L., and Borman, D. A. 1992. “Telnet Remote Flow Control Option,” RFC 1372, 6 pages (Oct.).

Hornig, C. 1984. “Standard for the Transmission of IP Datagrams over Ethernet Networks,” RFC 894, 3 pages (Apr.).

Huitema, C. 1993. “IAB Recommendation for an Intermediate Strategy to Address the Issue of Scaling,” RFC 1481, 2 pages (July).
实现CIDR的IAB建议。

Jacobson, V. 1988. “Congrestion Avoidance and Control,” *Computer Communication Review*, vol. 18, no. 4, pp. 314-329 (Aug.).
描述TCP的慢启动和拥塞避免算法的一篇经典文章。这篇文章的一个 PostScript副本通过匿名FTP在主机`ftp.ee.lbl.gov`的文件`congavoid.ps.z`中提供。

Jacobson, V. 1990a. “Compressing TCP/IP Headers for Low-Speed Serial Links,” RFC 1144, 43 pages (Feb.).
描述CSLIP, 一个压缩TCP和IP首部的SLIP版本。

Jacobson, V. 199b. “Modified TCP Congestion Avoidance Algorithm,” April 30, 1990, end2end-interest mailing list (Apr.).
描述了快速重传和快速恢复算法。

Jacobson, V. 1990c. “Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno,” *Proceeding of the Eighteenth Internet Engineering Task Force*, p. 365 (Sept.), University of British Columbia, Vancouver, B.C.

Jacobson, V., and Braden, R. T. 1988. “TCP Extensions for Long-Delay Paths,” RFC 1072, 16 pages (Oct.).
描述了TCP的选择确认选项，在后来的RFC 1323中这个选项已经被删去了。

Jacobson, V. Braden, R. T., and Borman, D. A. 1992. “TCP Extensions for High Performance,” RFC 1323, 37 pages (May).
描述了窗口缩放选项、时间戳选项和PAWS算法，以及需要这些改变的理由。

Jacobson, V., Braden, R. T., and Zhang, L. 1990. “TCP Extensions for High-Speed Paths,” RFC 1185, 21 pages (Oct.).
尽管这个RFC已经被RFC 1323废止了，但有关防止TCP旧的重复的报文段的附录仍然值得一读。

Juszczak, C. 1989. “Improving the Performance and Correctness of an NFS Server,” *Proceedings of the 1989 Winter USENIX Conference*, pp. 53-63, San Diego, Calif.
提供了NFS服务器快速缓存的实现细节。

Kantor, B. 1991. “BSD Rlogin,” RFC 1282, 5 pages (Dec.).
Rlogin协议的规范。

Karn, P., and Partridge, C. 1987. “Improving Round-Trip Time Estimates in Reliable Transport Protocols,” *Computer Communication Review*, vol. 17, no. 5, pp. 2-7 (Aug.).
处理已经重传报文段的重传超时的 Karn算法的细节。这篇文章的一个 PostScript副本通过匿名FTP在主机sics.se的文件pub/craig/karn-partridge.ps中提供。

Katz, D. 1990. “Proposed Standard for the Transmission of IP Datagrams Over FDDI Networks,” RFC 1188, 11 pages (Oct.).

说明了在FDDI网络上如何封装IP数据报和ARP请求和应答，包括多播技术。

Kent, C. A., and Mogul, J. C. 1987. “Fragmentation Considered Harmful,” *Computer Communication Review*, vol. 17, no. 5, pp. 390-401 (Aug.).

Kent, S. T. 1991. “ U.S. Department of Defense Security Options for the Internet Protocol,” RFC 1108, 17 pages (Nov.).

Kleinrock, L. 1992. “The Latency / Bandwidth Tradeoff in Gigabit Networks,” *IEEE Communications Magazine*, vol. 30, no. 4, pp. 36-40 (Apr.).

Klensin, J., Freed, N., and Moore, K. 1993. “SMTP Service Extension for Message Size Declaration,” RFC 1427, 8 pages (Feb.).

Klensin, J., Freed, N., Rose, M. T., Stefferud, E. A., and Crocker, D. 1993a. “SMTP Service Extensions,” RFC 1425, 10 pages (Feb.).

Klensin, J., Freed, N., Rose, M. T., Stefferud, E. A., and Crocker, D. 1993b. “SMTP Service Extension for 8bit-MIME Transport,” RFC 1426, 6 pages (Feb.).

Krol, E. 1992. *The Whole Internet*. O’Reilly & Associates, Sebastopol, Calif.

关于Internet介绍的入门书。

LaQuey, T. 1993. *The Internet Companion: A Beginner ’s Guide to Global Networking*. Addison-Wesley, Reading, Mass.

关于Internet介绍的入门书。

Leffler, S. J., and Karels, M. J. 1984. “Trailer Encapsulations,” RFC 893, 3 pages (Apr.).

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Mass.

一本完整介绍4.3BSD Unix系统的书。这本书描述了Tahoe版的4.3BSD。

Lougheed, K., and Rekhter, Y. 1991. “A Border Gateway Protocol 3 (BGP-3),” RFC 1267, 35 pages (Oct.).

Lynch, D. C. 1993. “Historical Perspective,” *in Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 3-14. Addison-Wesley, Reading, Mass.

描述了Internet的前身：ARPANET。

Macklem, R. 1991. “Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol,” *Proceedings of the 1991 Winter USENIX Conference*, pp. 53-64, Dallas, Tex.

描述了一个同时使用TCP和UDP的NFS实现。

Malkin, G. S. 1993a. “RIP Version 2: Carrying Additional Information.” RFC 1388, 7 pages (Jan.).

Malkin, G. S. 1993b. “Traceroute Using an IP Option,” RFC 1393, 7 pages (Jan.).

建议用于traceroute新版本的ICMP修改。

Mallory, T., and Kullberg, A. 1990. “Incremental Updating of the Internet Checksum,” RFC 1141, 2 pages (Jan.).

描述了递增改变Internet检验和的实现技术。

Manber, U. 1990. “Chain Reactions in Networks,” *IEEE Computer*, vol. 23, no. 10, pp. 57-63 (Oct.).

描述了广播风暴和网络崩溃的类型，类似于习题 9.3和9.4所显示的。

McCanne, S., and Jacobson, V. 1993. “The BSD Packet Filter: A New Architecture for User-Level Packet Capture,” *Proceeding of the 1993 Winter USENIX Conference*. Pp. 259-269, San Diego, Calif.

BSD分组过滤器（BPF）的具体描述以及与 Sun的网络接口栓（NIT）的比较。这篇文章的一个PostScript副本通过匿名FTP在主机ftp.ee.lbl.gov的文件papers/bpf-usenix93.ps.Z中提供。

McCloghrie, K., and Rose, M. T. 1991. “Management Information Base for Network Management of TCP/IP-based Internets: MIB-II,” RFC 1213 (Mar.).

McGregor, G. 1992. “PPP Internet Protocol Control Protocol (IPCP),” RFC 1332, 12 pages (May).

用于TCP/IP的PPP的NCP的描述。

Mills, D. L. 1992. “Network Time Protocol (Version 3): Specification, Implementation, and Analysis,” RFC 1305, 113 pages (Mar.).

Mockapetris, P. V. 1987a. “Domain Names: Concepts and Facilities,” RFC 1034, 55 pages (Nov.).

一个DNS的介绍。

Mockapetris, P. V. 1987b. “Domain Names: Implementation and Specification,” RFC 1035, 55 pages (Nov.).

DNS规范。

Mogul, J. C. 1990. “Efficient Use of Workstations for Passive Monitoring of Local Area Networks,” *Computer Communication Review*, vol. 20, no. 4, pp. 253-263 (Sept.).

描述了如何使用工作站监视局域网，而不购买专门的网络分析硬件产品。

Mogul, J. C. 1992. “Holy Turbocharger Batman, (evil cheating), NFS async writes,” Message-ID 1992Mar2.191711.9935@PA.dec.com, Usenet, comp.protocols.nfs Newsgroup (Mar.).

在一个繁忙的NFS服务器上收集的，40天内Internet检验和错误的一些有趣的统计数据。

Mogul, J. C. 1993. “IP Network Performance,” in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 575-675. Addison-Wesley, Reading, Mass.

包含了许多TCP/IP协议栈的主题，它们可以用来获得最优的性能。

Mogul, J. C., and Deering, S. E. 1990. “Path MTU Discovery,” RFC 1191, 19 pages (Apr.).

Mogul, J. C., and Postel, J. B. 1985. “Internet Standard Subnetting Procedure,” RFC 950, 18 pages (Aug.).

Moore, K. 1993. “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text,” RFC 1522, 10 pages (Sept.).

描述了使用7 bit ASCII在RFC 822邮件首部发送非ASCII字符的一种方法。

Moy, J. 1991. “OSPF Version 2,” RFC 1247, 189 pages (July).

Nagle, J. 1984. “Congestion Control in IP/TCP Internetworks,” RFC 896, 9 pages(Jan.).

Nagle算法的描述。

Nye, A., ed. 1992. *The X Window System, Volume 0: X Protocol Reference Manual, Third*

Edition. O'Reilly & Associates, Sebastopol, Calif.

Obracezka, K., Danzig, P. B., and Li, S. 1993. "Internet Resource Discovery Services," *IEEE Computer*, vol. 26, no. 9, pp. 8-22 (Sept.).

对当前Internet上资源发现工具的一个综述：Alex、Archie、Gopher、Indie、Knowbot信息服务、Netfind、Prospero、WAIS、WWW和X.500。这篇文章的一个PostScript副本通过匿名FTP在主机`caldera.usc.edu`的文件`/pub/kobraczk/ieeecomputer.ps.z`中提供。

Papadopoulos, C., and Parulkar, G. M. 1993. "Experimental Evaluation of SunOS IPC and TCP/IP Protocol Implementation," *IEEE/ACM Transactions on Networking*, vol. 1, no. 2, pp. 199-216 (Apr.).

测量当发送数据和接收数据时在协议族的不同层中引入的开销。

Partridge, C. 1986. "Mail Routing and the Domain System," RFC 974, 7 pages (Jan.).

怎样使用DNS的MX记录进行邮件的选路。

Partridge, C. 1994. *Gigabit Networking*. Addison-Wesley, Reading, Mass.

描述当网络速率超过1 Gb/s时产生的问题。

Partridge, C., and Pink, S. 1993. "A Faster UDP." *IEEE/ACM Transactions on Networking*, Vol. 1, no. 4, pp. 429-440 (Aug.).

描述了对伯克利实现的改进，可以将UDP的性能提高30%。

Paxson, V. 1993. "Empirically-Derived Analytic Models of Wide-Area TCP Connections: Extended Report," LBL-34086, Lawrence Berkeley Laboratory and EECS Division, University of California, Berkeley (June).

包含了在14个广域流量跟踪中250万个TCP连接的分析。这个报告的一个PostScript副本通过匿名FTP在主机`ftp.ee.lbl.gov`的文件`WAN-TCP-models.1.ps.Z`和`WAN-TCP-models.2.ps.Z`中提供。

Perlman, R. 1992. *Interconnections: Bridges and Routers*. Addison-Wesley, Reading, Mass.

一本包含了许多详细的网络互连方法（桥和路由器）和不同的路由算法的书。

Plummer, D. C. 1982. "An Ethernet Address Resolution Protocol," RFC 826, 10 pages (Nov.).

Postel, J. B. 1980. "User Datagram Protocol," RFC 768, 3 pages (Aug.).

Postel, J. B., ed. 1981a. "Internet Protocol," RFC 791, 45 pages (Sept.).

Postel, J. B. 1981b. "Internet Control Message Protocol," RFC 792, 21 pages (Sept.).

Postel, J. B., ed. 1981c. "Transmission Control Protocol," RFC 793, 85 pages (Sept.).

Postel, J. B. 1982. "Simple Mail Transfer Protocol," RFC 821, 68 pages (Aug.).

Postel, J. B. 1987. "TCP and IP Bake Off," RFC 1025, 6 pages (Sept.).

描述了在早期开发过程中，为了测试互连性在TCP/IP的不同实现之间进行的一些测试过程和记录。

Postel, J. B., ed. 1994. "Internet Official Protocol Standards," RFC 1600, 36 pages (Mar.).

所有的Internet协议的状态。这个RFC定期改变——查看最近的RFC索引以了解当前的版本。

Postel, J. B., and Reynolds, J. K. 1983a. "Telnet Protocol Specification," RFC 854, 15 pages (May).

基本的Telnet协议规范。很多以后的RFC描述了特定的Telnet选项。

Postel, J. B., and Reynolds, J. K. 1983b. “Telnet Binary Transmission,” RFC 856, 4 pages (May).

Postel, J. B., and Reynolds, J. K. 1983c. “Telnet Echo Option,” RFC 857, 5 pages (May).

Postel, J. B., and Reynolds, J. K. 1983d. “Telnet Suppress Go Ahead Option,” RFC 858, 3 pages (May).

Postel, J. B., and Reynolds, J. K. 1983e. “Telnet Status Option,” RFC 859, 3 pages (May).

Postel, J. B., and Reynolds, J. K. 1983f. “Telnet Timing Mark Option,” RFC 860, 4 pages (May).

Postel, J. B., and Reynolds, J. K. 1985. “File Transfer Protocol (FTP),” RFC 959, 69 pages (Oct.).

Postel, J. B., and Reynolds, J. K. 1988. “Standard for the Transmission of IP Datagrams over IEEE 802 Networks,” RFC 1042, 15 pages (Apr.).

在IEEE 802网络上封装IP数据报和ARP请求和应答的规范。

Pusateri, T. 1993. “IP Multicast Over Token-Ring Local Area Networks,” RFC 1469, 4 pages (June).

Rago, S. A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, Mass.

一本关于TLI和流子系统的书。

Rekhter, Y. and Gross, P. 1991. “Application of the Border Gateway Protocol in the Internet,” RFC 1268, 13 pages (Oct.).

Rekhter, Y., and Li, T. 1993. “An Architecture for IP Address Allocation with CIDR,” RFC 1518, 27 pages (Sept.).

Reynolds, J. K. 1989. “The Helminthiasis of the Internet,” RFC 1135, 33 pages (Dec.).

包含了1988年Internet蠕虫的详细讨论。

Reynolds, J. K., and Postel, J. B. 1992. “Assigned Numbers,” RFC 1340, 138 pages (July).

Internet协议族所有的编码。这个RFC定期改变——查看最近的RFC索引以了解当前的版本。

Romkey, J. L. 1988. “A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP,” RFC 1055, 6 pages (June).

Rose, M. T. 1990. *The Open Book: A Practical Perspective on OSI*. Prentice-Hall, Englewood Cliffs, N.J.

一本关于OSI协议的书。第8章提供了ASN.1和BER的细节。

Rose, M. T. 1993. *The Internet Message: Closing the Book with Electronic Mail*. Prentice-Hall, Englewood Cliffs, N.J.

一本关于Internet邮件的书，包含MIME的细节。

Rose, M. T. 1994. *The Simple Book: An Introduction to Internet Management, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.

一本关于SNMPv2的书。这本书的第1版介绍了SNMPv1。

Rose, M. T., and McCloghrie, K. 1990. “Structure and Identification of Management

Information for TCP/IP-based Internets,” RFC 1155, 22 pages (May).
定义了SNMPv1的SMI。

Rosenberg, W., Kenney, D., and Fisher, G. 1992. *Understanding DCE*. O’Reilly & Associates, Sebastopol, Calif.
提供了OSF的分布式计算环境的概述。

Routhier, S. A. 1993.“Implementation Experience for SNMPv2,” *The Simple Times*, vol. 2, no. 4, pp. 1-4 (July-Aug.).
描述了对SNMPv1的修改以支持SNMPv2。
这个期刊是免费电子发行的。以主题“ help ”向st-subscriptions@simple-times.org发送一个电子邮件以获得订阅信息。

Schryver, V. J. 1993.“Info on High Speed Transport Protocols Requested,” Message-ID i0imr8g@rhyolite.wpd.sgi.com, Usenet, comp.protocols.tcp-ip Newsgroup (May).
提供了一些FDDI实现上的TCP性能数字。

Schwartz, M. F., and Tsirigotis, P. G. 1991.“Experience with a Semantically Cognizant Internet White Pages Directory Tool,” *Journal of Internetworking Research and Experience*, vol. 2, no. 1, pp. 23-50 (Mar.).
也可以通过匿名 FTP在主机 ftp.cs.colorado.edu的文件 pub/cs/techreports/schwartz/PostScript/ White.Pages.ps.Z中得到。

Simpson, W. A. 1993.“The Point-to-Point Protocol (PPP),” RFC 1548, 53 pages (Dec.).
定义了PPP和它的链路控制协议。

Sollins, K. R. 1992.“The TFTP Protocol (Revision 2),” RFC 1350, 11 pages (July).

Stallings, W. 1987. *Handbook of Computer-Communications Standards, Volume* 2: *Local Network Standards*. Macmillan, New York.
包含了IEEE 802 局域网标准的细节。

Stallings, W. 1993. *SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards*. Addison-Wesley, Reading, Mass.
描述了SNMPv1和SNMPv2的差别。

Stern, H. 1991. *Managing NFS and NIS*. O’Reilly & Associates, Sebastopol, Calif.
包含了许多安装、使用和管理NFS的细节。

Stevens, W. R. 1990. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, N.J.
一本详细介绍在Unix上使用插口和TLI进行网络程序设计的书。

Stevens, W. R. 1992. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, Mass.
一本详细介绍Unix程序设计的书。

Sun Microsystems. 1987.“XDR: External Data Representation Standard,” RFC 1014, 20 pages (June).

Sun Microsystems. 1988a.“RFC: Remote Procedure Call, Protocol Specification, Version 2,” RFC 1057, 25 pages (June).

Sun Microsystems. 1988b.“NFS: Network File System Protocol Specification,” RFC 1094, 27 pages

(Mar.).
第2版Sun NFS的规范。

Sun Microsystems. 1994. *NFS: Network File System Version 3 Protocol Specification*. Sun Microsystems, Mountain View, Calif.
这个文档的一个 PostScript副本通过匿名 FTP在主机 ftp.uu.net的文件 networking/ip/nfs/NFS3.spec.ps.Z中提供。

Tanenbaum, A. S. 1989. *Computer Networks, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.
一本关于计算机网络的通用书。

Topolcic, C. 1993. “Status of CIDR Deployment in the Internet,” RFC 1467, 9 pages (Aug.).

Tsuchiya, P. F. 1991. “On the Assignment of Subnet Numbers,” RFC 1219, 13 pages (Apr.).
建议从最高位往下分配子网 ID，从最低位往上分配主机 ID。这样使得以后在某个点上改变子网掩码时不需要对所有的系统重新编号。

Ullmann, R. 1993. “TP/IX: The Next Internet,” RFC 1475, 35 pages (June).
下一代Internet协议的另一个建议。

VanBokkelen, J. 1989. “Telnet Terminal-Type Option,” RFC 1091, 7 pages (Feb.).

Waitzman, D. 1988. “Telnet Window Size Option,” RFC 1073, 4 pages (Oct.).

Waitzman, D., Partridge, C., and Deering, S. E. 1988. “Distance Vector Multicast Routing Protocol,” RFC 1075, 24 pages (Nov.).

Warnock, R. P. 1991. “Need Help Selecting Ethernet Cards for Very High Performance Throughput Rates,” Message-ID< lbhal10@sgi.sgi.com> Usenet, comp.protocols.tcp-ip Newsgroup (Sept.).
提供了我们从图24-9计算的TCP的性能数据。

Weider, C., Reynolds, J. K., and Heker, S. 1992. “Technical Overview of Directory Services Using the X.500 Protocol,” RFC 1309, 16 pages (Mar.).

Wimer, W. 1993. “Clarifications and Extensions for the Bootstrap Protocol,” RFC 1542, 23 pages (Oct.).

X/Open. 1991. *Protocols for X/Open Internetworking: XNFS*. X/Open, Reading, Berkshire, U.K.
对Sun RPC、XDR和NFS的一个更好的描述。还包含了 NFS锁定管理程序和状态监视协议的描述。附录中详细讨论了使用 NFS和使用本地文件访问之间的语义差别。 X/Open文档编号为XO/CAE/91/030。

Zimmerman, D. P. 1991. “Finger User Information Protocol,” RFC 1288, 12 pages (Dec.)

缩 略 语

ACK (ACKnowledgment) TCP首部中的确认标志
API (Application Programming Interface) 应用编程接口
ARP (Address Resolution Protocol) 地址解析协议
ARPANET
(Defense Advanced Research Project Agency NETwork) (美国)国防部远景研究规划局
AS (Autonomous System) 自治系统
ASCII (American Standard Code for Information Interchange) 美国信息交换标准码
ASN.1 (Abstract Syntax Notation One) 抽象语法记法1
BER (Basic Encoding Rule) 基本编码规则
BGP (Border Gateway Protocol) 边界网关协议
BIND (Berkeley Internet Name Domain) 伯克利Internet域名
BOOTP (BOOTstrap Protocol) 引导程序协议
BPF (BSD Packet Filter) BSD 分组过滤器
CIDR (Classless InterDomain Routing) 无类型域间选路
CIX (Commercial Internet Exchange) 商业互联网交换
CLNP (ConnectionLess Network Protocol) 无连接网络协议
CRC (Cyclic Redundancy Check) 循环冗余检验
CSLIP (Compressed SLIP) 压缩的SLIP
CSMA (Carrier Sense Multiple Access) 载波侦听多路存取
DCE (Data Circuit-terminating Equipment) 数据电路端接设备
DDN (Defense Data Network) 国防数据网
DF (Don't Fragment) IP首部中的不分片标志
DHCP (Dynamic Host Configuration Protocol) 动态主机配置协议
DLPI (Data Link Provider Interface) 数据链路提供者接口
DNS (Domain Name System) 域名系统
DSAP (Destination Service Access Point) 目的服务访问点
DSLAM (DSL Access Multiplexer) 数字用户线接入复用器
DSSS (Direct Sequence Spread Spectrum) 直接序列扩频
DTS (Distributed Time Service) 分布式时间服务
DVMRP (Distance Vector Multicast Routing Protocol) 距离向量多播选路协议
EBONE (European IP BackbONE) 欧洲IP主干网
EOL (End of Option List) 选项清单结束
EGP (External Gateway Protocol) 外部网关协议
EIA (Electronic Industries Association) 美国电子工业协会

FCS (Frame Check Sequence) 帧检验序列
FDDI (Fiber Distributed Data Interface) 光纤分布式数据接口
FIFO (First In, First Out) 先进先出
FIN (FINish) TCP首部中的结束标志
FQDN (Full Qualified Domain Name) 完全合格的域名
FTP (File Transfer Protocol) 文件传送协议
HDLC (High-level Data Link Control) 高级数据链路控制
HELLO 选路协议
IAB (Internet Architecture Board) Internet体系结构委员会
IANA (Internet Assigned Numbers Authority) Internet号分配机构
ICMP (Internet Control Message Protocol) Internet控制报文协议
IDRP (InterDomain Routing Protocol) 域间选路协议
IEEE (Institute of Electrical and Electronics Engineering) （美国）电气与电子工程师协会
IEN (Internet Experiment Notes) 互联网试验注释
IESG (Internet Engineering Steering Group) Internet工程指导小组
IETF (Internet Engineering Task Force) Internet工程专门小组
IGMP (Internet Group Management Protocol) Internet组管理协议
IGP (Interior Gateway Protocol) 内部网关协议
IMAP (Internet Message Access Protocol) Internet报文存取协议
IP (Internet Protocol) 网际协议
IRTF (Internet Research Task Force) Internet研究专门小组
IS-IS (Intermediate System to Intermediate System Protocol) 中间系统到中间系统协议
ISN (Initial Sequence Number) 初始序号
ISO (International Organization for Standardization) 国际标准化组织
ISOC (Internet SOCiety) Internet协会
LAN (Local Area Network) 局域网
LBX (Low Bandwidth X) 低带宽X
LCP (Link Control Protocol) 链路控制协议
LFN (Long Fat Net) 长肥网络
LIFO (Last In, First Out) 后进先出
LLC (Logical Link Control) 逻辑链路控制
LSRR (Loose Source and Record Route) 宽松的源站及记录路由
MBONE (Multicast Backbone On the InterNEt) Internet上的多播主干网
MIB (Management Information Base) 管理信息库
MILNET (MILitary NETwork) 军用网
MIME (Multipurpose Internet Mail Extensions) 通用Internet邮件扩充
MSL (Maximum Segment Lifetime) 报文段最大生存时间
MSS (Maximum Segment Size) 最大报文段长度
MTA (Message Transfer Agent) 报文传送代理

MTU (Maximum Transmission Unit) 最大传输单元
NCP (Network Control Protocol) 网络控制协议
NFS (Network File System) 网络文件系统
NIC (Network Information Center) 网络信息中心
NIT (Network Interface Tap) 网络接口栓（Sun公司的一个程序）
NNTP (Network News Transfer Protocol) 网络新闻传送协议
NOAO (National Optical Astronomy Observatories) 国家光学天文台
NOP (No Operation) 无操作
NSFNET (National Science Foundation NETwork) 国家科学基金网络
NSI (NASA Science Internet) （美国）国家宇航局Internet
NTP (Network Time Protocol) 网络时间协议
NVT (Network Virtual Terminal) 网络虚拟终端
OSF (Open Software Foudation) 开放软件基金
OSI (Open Systems Interconnection) 开放系统互连
OSPF (Open Shortest Path First) 开放最短通路优先
PAWS (Protection Against Wrapped Sequence number) 防止回绕的序号
PDU (Protocol Data Unit) 协议数据单元
POSIX (Portable Operating System Interface) 可移植操作系统接口
PPP (Point-to-Point Protocol) 点对点协议
PSH (PuSH) TCP首部中的急迫标志
RARP (Reverse Address Resolution Protocol) 逆地址解析协议
RFC (Request For Comments) Internet的文档，其中的少部分成为标准文档
RIP (Routing Information Protocol) 路由信息协议
RPC (Remote Procedure Call) 远程过程调用
RR (Resource Record) 资源记录
RST (ReSeT) TCP首部中的复位标志
RTO (Retransmission Time Out) 重传超时
RTT (Round-Trip Time) 往返时间
SACK (Selective ACKnowledgment) 有选择的确认
SLIP (Serial Line Internet Protocol) 串行线路Internet协议
SMI (Structure of Management Information) 管理信息结构
SMTP (Simple Mail Transfer Protocol) 简单邮件传送协议
SNMP (Simple Network Management Protocol) 简单网络管理协议
SSAP (Source Service Access Point) 源服务访问点
SSRR (Strict Source and Record Route) 严格的源站及记录路由
SWS (Silly Window Syndrome) 糊涂窗口综合症
SYN (SYNchronous) TCP首部中的同步序号标志
TCP (Transmission Control Protocol) 传输控制协议
TFTP (Trivial File Transfer Protocol) 简单文件传送协议

TLI (Transport Layer Interface) 运输层接口
TTL (Time-To-Live) 生存时间或寿命
TUBA (TCP and UDP with Bigger Addresses) 具有更长地址的TCP和UDP
Telnet 远程终端协议
UA (User Agent) 用户代理
UDP (User Datagram Protocol) 用户数据报协议
URG (URGent) TCP首部中的紧急指针标志
UTC (Coordinated Universal Time) 协调的统一时间
UUCP (Unix-to-Unix CoPy) Unix到Unix的复制
WAN (Wide Area Network) 广域网
WWW (World Wide Web) 万维网
XDR (eXternal Data Representation) 外部数据表示
XID (transaction ID) 事务标识符
XTI (X/Open Transport Layer Interface) X/Open运输层接口