第一部分　TCP事务协议

第1章　T/TCP　概　述

1.1　概述

本章首先介绍客户-服务器事务概念。我们从使用 UDP的客户-服务器应用开始，这是最简单的情形。接着我们编写使用 TCP的客户和服务器程序，并由此考察两台主机间交互的TCP/IP分组。然后我们使用T/TCP，证明利用T/TCP可以减少分组数，并给出为利用 T/TCP需要对两端的源代码所做的最少改动。

接下来介绍了运行书中示例程序的测试网络，并对分别使用 UDP、TCP和T/TCP的客户-服务器应用程序进行了简单的时间耗费比较。我们考察了一些使用 TCP的典型Internet应用程序，看看如果两端都支持 T/TCP，将需要做哪些修改。紧接着，简要介绍了 Internet协议族中事务协议的发展历史，概略叙述了现有的 T/TCP实现。

本书全文以及有关T/TCP的文献中，事务一词的含义都是指客户向服务器发出一个请求，然后服务器对该请求作出应答。 Internet中最常见的一个例子是，客户向域名服务器 (DNS)发出请求，查询域名对应的 IP地址，然后域名服务器给出响应。本书中的事务这个术语并没有数据库中的事务那样的含义：加锁、两步提交、回退，等等。

1.2　UDP上的客户-服务器

我们先来看一个简单的 UDP客户-服务器应用程序的例子，其客户程序源代码如图 1-1所示。在这个例子中，客户向服务器发出一个请求，服务器处理该请求，然后发回一个应答。

udpcli.c

```
 1 #include    "cliserv.h"

 2 int
 3 main(int argc, char *argv[])
 4 {                                   /* simple UDP client */
 5     struct sockaddr_in serv;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n;

 8     if (argc != 2)
 9         err_quit("usage: udpcli <IP address of server>");

10     if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
11         err_sys("socket error");

12     memset(&serv, 0, sizeof(serv));
13     serv.sin_family = AF_INET;
14     serv.sin_addr.s_addr = inet_addr(argv[1]);
15     serv.sin_port = htons(UDP_SERV_PORT);
```

图1-1　UDP上的简单客户程序

```
16      /* form request[] ... */
17      if (sendto(sockfd, request, REQUEST, 0,
18                 (SA) &serv, sizeof(serv)) != REQUEST)
19          err_sys("sendto error");
20      if ((n = recvfrom(sockfd, reply, REPLY, 0,
21                        (SA) NULL, (int *) NULL)) < 0)
22          err_sys("recvfrom error");
23      /* process "n" bytes of reply[] ... */
24      exit(0);
25  }
```

udpcli.c

图1-1 (续)

本书中所有源代码的格式都是这样。每一非空行前面都标有行号。正文中叙述某段源代码时，这段源代码的起始和结束行号标记于正文段落的左边，如下面的正文所示。有时这些段落前面会有一小段说明，对所描述的源代码进行概要说明。源代码段开头和结尾处的水平线标明源代码段所在的文件名。这些文件名通常都是指我们在1.9节中将介绍的4.4版BSD-Lite中发布的文件。

我们来讨论这个程序的一些有关特性，但不详细描述插口函数，因为我们假设读者对这些函数有一些基本的认识。关于插口函数的细节在参考书 [Stevens 1990]的第6章中可以找到。图1-2给出了头文件`cliserv.h`。

1. 创建UDP插口

10-11 `socket`函数用于创建一个UDP插口，并将一个非负的插口描述符返回给调用进程。出错处理函数`err_sys`参见参考书[Stevens 1992]的附录B.2。这个函数可以接受任意数目的参数，但要用`vsprintf`函数对它们格式化，然后这个函数会打印出系统调用所返回的`errno`值所对应的Unix出错信息，然后终止进程。

2. 填写服务器地址

12-15 首先用`memset`函数将Internet插口地址结构清零，然后填入服务器的IP地址和端口号。为简明起见，我们要求用户在程序运行中通过命令行输入一个点分十进制数形式的IP地址(`argv[1]`)。服务器端口号(`UDP_SERV_PORT`)在头文件`cliserv.h`中用`#define`定义，在本章的所有程序首部中都包含了该头文件。这样做是为了使程序简洁，并避免使调用`gethostbyname`和`getservbyname`函数的源代码复杂化。

3. 构造并向服务器发送请求

16-19 客户程序构造一个请求(只用一行注释来表示)，并用`sendto`函数将其发出，这样就有一个UDP数据报发往服务器。同样是为了简明起见，我们假设请求(`REQUEST`)和应答(`REPLY`)的报文长度为固定值。实用的程序应当按照请求和应答的最大长度来分配缓存空间，但实际的请求和应答报文长度是变化的，而且一般都比较小。

4. 读取和处理服务器的应答

20-23 调用`recvfrom`函数将使进程阻塞(即置为睡眠状态)，直至收到一个数据报。接着客户进程处理应答(用一行注释来表示)，然后进程终止。

由于`recvfrom`函数中没有超时机制，请求报文或应答报文中任何一个丢失都将造成该进程永久挂起。事实上，UDP客户-服务器应用的一个基本问题就是对现实世界中的此类错误缺少健壮性。在本节的末尾将对这个问题做更详细的讨论。

在头文件cliserv.h中，我们将SA定义为struct sockaddr *；即指向一般的插口地址结构的指针。每当有一个插口函数需要一个指向插口地址结构的指针时，该指针必须被置为指向一个一般性插口地址结构的指针。这是由于插口函数先于ANSI C标准出现，在80年代早期开发插口函数的时候，void*(空类型)指针类型尚不可用。问题是，“struct sockaddr *”总共有17个字符，这经常使这一行源代码超出屏幕(或书本页面)的右边界，因此我们将其缩写成为 SA。这个缩写是从BSD内核源代码中借用过来的。

图1-2给出了在本章所有程序中都包含的头文件 cliserv.h。

cliserv.h

```
 1 /* Common includes and defines for UDP, TCP, and T/TCP
 2  * clients and servers */

 3 #include    <sys/types.h>
 4 #include    <sys/socket.h>
 5 #include    <netinet/in.h>
 6 #include    <arpa/inet.h>
 7 #include    <stdio.h>
 8 #include    <stdlib.h>
 9 #include    <string.h>
10 #include    <unistd.h>
11 #define REQUEST 400             /* max size of request, in bytes */
12 #define REPLY   400             /* max size of reply, in bytes */

13 #define UDP_SERV_PORT   7777    /* UDP server's well-known port */
14 #define TCP_SERV_PORT   8888    /* TCP server's well-known port */
15 #define TTCP_SERV_PORT  9999    /* T/TCP server's well-known port */

16     /* Following shortens all the type casts of pointer arguments */
17 #define SA  struct sockaddr *

18 void    err_quit(const char *,...);
19 void    err_sys(const char *,...);
20 int     read_stream(int, char *, int);
```

cliserv.

图1-2 本章各程序中均包含的头文件 cliserv.h

图1-3给出了相应的UDP服务器程序。

udpserv.

```
 1 #include    "cliserv.h"

 2 int
 3 main()
 4 {                               /* simple UDP server */
 5     struct sockaddr_in serv, cli;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n, clilen;

 8     if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
 9         err_sys("socket error");

10     memset(&serv, 0, sizeof(serv));
11     serv.sin_family = AF_INET;
12     serv.sin_addr.s_addr = htonl(INADDR_ANY);
13     serv.sin_port = htons(UDP_SERV_PORT);
```

图1-3 与图1-1的UDP客户程序对应的UDP服务器程序

```
14      if (bind(sockfd, (SA) &serv, sizeof(serv)) < 0)
15          err_sys("bind error");

16      for (;;) {
17          clilen = sizeof(cli);
18          if ((n = recvfrom(sockfd, request, REQUEST, 0,
19                            (SA) &cli, &clilen)) < 0)
20              err_sys("recvfrom error");

21          /* process "n" bytes of request[] and create reply[] ... */

22          if (sendto(sockfd, reply, REPLY, 0,
23                     (SA) &cli, sizeof(cli)) != REPLY)
24              err_sys("sendto error");
25      }
26  }
```

udpserv.c

图1-3 (续)

5. 创建UDP插口和绑定本机地址

8-15 调用socket函数创建一个UDP插口，并在其Internet插口地址结构中填入服务器的本机地址。这里本机地址设置为通配符 (INADDR_ANY)，这意味着服务器可以从任何一个本机接口接收数据报(假设服务器是多宿主的，即可以有多个网络接口)。端口号设为服务器的知名端口(UDP_SERV_PORT)，该常量也在前面讲过的头文件 cliserv.h中定义。本机IP地址和知名端口用bind函数绑定到插口上。

6. 处理客户请求

16-25 接下来，服务器程序就进入一个无限循环：等待客户程序的请求到达 (recvfrom)，处理该请求(我们只用一行注释来表示处理动作)，然后发出应答(sendto)。

这只是最简单的UDP客户 - 服务器应用。实际中常见的例子是域名服务系统 (DNS)。DNS客户(称作解析器)通常是一般客户应用程序 (例如，Telnet客户、FTP客户或WWW浏览器)的一个部分。解析器向 DNS服务器发出一个UDP数据报，查询某一域名对应的 IP地址。服务器发回的应答通常也是一个UDP数据报。

如果观察客户向服务器发送请求时双方交换的分组，我们就会得到图 1-4这样的时间系列，页面上时间自上而下递增。服务器程序先启动，其行为过程给在图 1-4的右半部，客户程序稍后启动。

我们分别来看客户和服务器程序中调用的函数及其相应内核执行的动作。在对 socket函数的两次调用中，上下紧挨着的两个箭头表示内核执行请求的动作并立即返回。在调用sendto函数时，尽管内核也立即返回，但实际上已经发出了一个 UDP数据报。为简明起见，我们假设客户程序的请求和服务器程序的应答所生成的 IP数据报的长度都小于网络的最大传输单元(MTU)，IP数据报不必分段。

在这个图中，有两次调用 recvfrom函数使进程睡眠，直到有数据报到达才被唤醒。我们把内核中相应的例程记为sleep和wakeup。

最后，我们还在图中标出了事务所耗费的时间。图 1-4的左侧标示的是客户端测得的事务时间：从客户发出请求到收到服务器的应答所经历的时间。组成这段事务时间的数值标在图的右侧：RTT + SPT，其中RTT是网络往返时间，SPT是服务器处理客户请求的时间。UDP客户 - 服务器事务的最短时间就是 RTT + SPT。

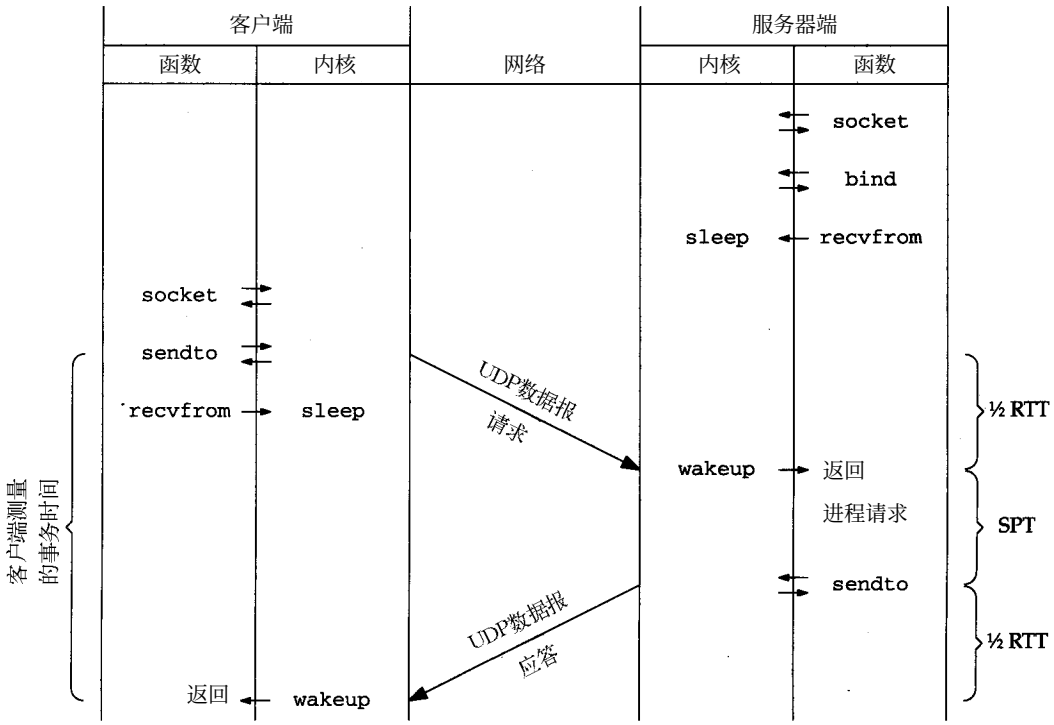

图1-4 UDP客户-服务器事务的时序图

尽管没有明确说明，但我们已经假设从客户到服务器的路径需要 ½ RTT时间，返回的路径又需½ RTT时间。但实际情况并非总是如此。据对大约 600条Internet路径的研究[Paxson 1995b]发现：30%的路径呈现明显的不对称性，说明两个方向上的路由经过了不同的站点。

我们的UDP客户-服务器看起来非常简捷 (每个程序只有大约30行有关网络的源代码)，但在实际环境中应用还不够健壮。由于 UDP是不保证可靠的协议，数据报可能会丢失、失序或重复，因此实用的应用程序必须处理这些问题。这通常是在客户程序调用 `recvfrom`时设置一个超时定时器，用以检测数据报的丢失，并重传请求。如果要使用超时定时器，客户程序就要测量RTT并动态更新，这是因为互连网上的RTT会在很大范围内变化，并且变化很快。但如果是服务器的应答丢失，而不是请求，那么服务器就要再次处理同一个请求，这可能会给某些服务带来问题。解决这个问题的办法之一是让服务器将每个客户最近一次请求的响应暂存起来，必要时重传这个应答即可，而不需要再次处理这个请求。最后，典型的情况是，客户向服务器发送的每个请求中都有一个不同的标识，服务器把这个标识在响应中传回来，使客户能把请求和响应匹配起来。在参考书 [Stevens 1990]的 8.4节中给出了UDP上的客户-服务器处理这些问题的源代码细节，但这将在程序中增加大约 500行源代码。

一方面，许多UDP应用程序都通过执行所有这些额外步骤 (超时机制、RTT值测量、请求标识，等等)来增加可靠性；另一方面，随着新的 UDP应用程序不断出现，这些步骤也在不断地推陈出新。参考书 [Patridge 1990b]中指出，“为了开发‘可靠的UDP应用程序’，你要有状态信息(序列号、重传计数器和往返时间估计器)，原则上你要用到当前TCP连接块中的全部信

息。因此，构筑一个‘可靠的UDP’，本质上和开发TCP一样难”。

有些应用程序并不实现上面所述的所有步骤：例如在接收时使用超时机制，但并不测量RTT值，当然更不会动态地更新RTT值。这样，当应用程序从一个环境(比如局域网)移植到另一个环境(比如广域网)中应用时，就可能会引发一些问题。比较好的解决办法是用TCP而不是用UDP，这样就可以利用TCP提供的所有可靠传输特性。但是这种办法会使客户端测得的事务时间由RTT + SPT增加到2×RTT + SPT(见下一节)，而且还会大大增加两个系统之间交换的分组数目。对付这些新的问题也有一个办法，即用T/TCP取代TCP，我们将在1.4节中对此进行讨论。

1.3 TCP上的客户-服务器

下一个例子是TCP上的客户-服务器事务应用。图1-5给出了客户程序。

tcpcli.c

```
 1 #include    "cliserv.h"
 2 int
 3 main(int argc, char *argv[])
 4 {                                   /* simple TCP client */
 5     struct sockaddr_in serv;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n;
 8     if (argc != 2)
 9         err_quit("usage: tcpcli <IP address of server>");
10     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");
12     memset(&serv, 0, sizeof(serv));
13     serv.sin_family = AF_INET;
14     serv.sin_addr.s_addr = inet_addr(argv[1]);
15     serv.sin_port = htons(TCP_SERV_PORT);
16     if (connect(sockfd, (SA) &serv, sizeof(serv)) < 0)
17         err_sys("connect error");
18     /* form request[] ... */
19     if (write(sockfd, request, REQUEST) != REQUEST)
20         err_sys("write error");
21     if (shutdown(sockfd, 1) < 0)
22         err_sys("shutdown error");
23     if ((n = read_stream(sockfd, reply, REPLY)) < 0)
24         err_sys("read error");
25     /* process "n" bytes of reply[] ... */
26     exit(0);
27 }
```

tcpcli.c

图1-5 TCP事务的客户

1. 创建TCP插口和连接到服务器

10-17 调用socket函数创建一个TCP插口，然后在Internet插口地址结构中填入服务器的IP地址和端口号。对connect函数的调用启动TCP的三次握手过程，在客户和服务器之间建立起连接。卷1的第18章给出了TCP连接建立和释放过程中交换分组的详细情况。

2. 发送请求和半关闭连接

19-22 客户的请求是用write函数发给服务器的。之后客户调用shutdown函数(函数的第2

个参数为1)关闭连接的一半，即数据流从客户向服务器的方向。这就告知服务器客户的数据已经发完了：从客户端向服务器传递了一个文件结束的通知。这时有一个设置了 FIN标志的TCP报文段发给服务器。客户此时仍然能够从连接中读取数据——只关闭了一个方向的数据流。这就叫做TCP的半关闭(half-close)。卷1的第18.5节给出了有关细节。

3. 读取应答

23-24 读取应答是由函数read_stream完成的，如图1-6所示。由于TCP是一个面向字节流的协议，没有任何形式的记录定界符，因而从服务器端 TCP传回的应答可能会包含在多个TCP报文段中。这也就可能会需要多次调用 read函数才能传递给客户进程。而且我们知道，当服务器发送完应答后就会关闭连接，使得 TCP向客户端发送一个带FIN的报文段，在read函数中返回一个文件结束标志 (返回值为0)。为了处理这些细节问题，在 read_stream函数中不断调用read函数直到接收缓存满或者read函数返回一个文件结束标志。 read_stream函数的返回值就是读取到的字节数。

tcpcli.c

```
 1 #include    "cliserv.h"
 2 int
 3 main(int argc, char *argv[])
 4 {                                   /* simple TCP client */
 5     struct sockaddr_in serv;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n;
 8     if (argc != 2)
 9         err_quit("usage: tcpcli <IP address of server>");
10     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");
12     memset(&serv, 0, sizeof(serv));
13     serv.sin_family = AF_INET;
14     serv.sin_addr.s_addr = inet_addr(argv[1]);
15     serv.sin_port = htons(TCP_SERV_PORT);
```

图1-6 read_stream 函数

还有一些别的方法可以在类似 TCP这样的流协议中用来给记录定界。许多Internet应用程序(FTP、SMTP、HTTP和NNTP)使用回车和换行符来标记记录的结束。其他一些应用程序(DNS，RPC)则在每个记录的前面加上一个定长的记录长度字段。在我们的例子中，利用了TCP的文件结束标志(FIN)，因为在每次事务中客户只向服务器发送一个请求，而服务器也只发回一个应答。 FTP也在其数据连接中采用了这项技术，用以告知对方文件已经结束。

图1-7给出的是TCP的服务器程序。

tcpserv.c

```
 1 #include    "cliserv.h"
 2 int
 3 main()
 4 {                                   /* simple TCP server */
 5     struct sockaddr_in serv, cli;
 6     char    request[REQUEST], reply[REPLY];
```

图1-7 TCP事务的服务器程序

```
 7      int     listenfd, sockfd, n, clilen;

 8      if ((listenfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
 9          err_sys("socket error");

10      memset(&serv, 0, sizeof(serv));
11      serv.sin_family = AF_INET;
12      serv.sin_addr.s_addr = htonl(INADDR_ANY);
13      serv.sin_port = htons(TCP_SERV_PORT);

14      if (bind(listenfd, (SA) &serv, sizeof(serv)) < 0)
15          err_sys("bind error");

16      if (listen(listenfd, SOMAXCONN) < 0)
17          err_sys("listen error");

18      for (;;) {
19          clilen = sizeof(cli);
20          if ((sockfd = accept(listenfd, (SA) &cli, &clilen)) < 0)
21              err_sys("accept error");

22          if ((n = read_stream(sockfd, request, REQUEST)) < 0)
23              err_sys("read error");

24          /* process "n" bytes of request[] and create reply[] ... */

25          if (write(sockfd, reply, REPLY) != REPLY)
26              err_sys("write error");

27          close(sockfd);
28      }
29  }
```

tcpserv.c

图1-7 (续)

4. 创建监听用TCP插口

8-17 用于创建一个TCP插口，并将服务器的知名端口绑定到该插口上。与UDP服务器一样，TCP服务器也将通配符作为其IP地址。调用listen函数将新创建的插口作为监听插口，用于等待客户端发起的连接。listen函数的第二个参数规定了允许的最大挂起连接数，内核要为该插口将这些连接进行排队处理。

SOMAXCONN在头文件<sys/socket.h>中定义。其数值过去一直都取5，但现在有一些比较新的系统将其定为10。对于一些很繁忙的服务器(例如：Web服务器)，已经发现需要取更大的值，比如256或1024。在14.5节中我们还将对此问题进行更多的讨论。

5. 接受连接和处理请求

18-28 服务器进程调用accept函数后就进入阻塞状态，直到有客户进程调用connect函数而建立起一个连接。函数accept返回一个新的插口描述符sockfd，代表与客户和服务器之间所建立的连接。服务器调用函数read_stream读取客户的请求(图1-6)，再调用write函数向客户发送应答。

这是一个反复循环的服务器：把当前的客户请求处理完毕后才又调用accept去接受另一个客户的连接。并发服务器可以并行地处理多个客户请求(即：同时处理)。在Unix的主机上实现并发服务器的常用技术是：在accept函数返回后，调用Unix的fork函数创建一个子进程，由子进程处理客户的请求，父进程则紧接着又调用accept去接受别的客户连接。实现并发服务器的另一项技术是为每个新建立的连接

创建一个线程(叫做轻量进程)。为了避免那些与网络无关的进程控制函数把我们的例子搞复杂，我们只给出了反复循环的服务器。参考书 [Stevens 1992]的第4章讨论比较了循环服务器和并发服务器。

还有第三个选择是采用预分支服务器。即服务器启动时连续调用 fork函数数次，并让每个子进程都在同一个监听插口描述符上调用 accept函数。这种办法节省了为每个客户的连接请求临时创建子进程的时间开销，这对于繁忙的服务器来说，是很大的节省。有些HTTP服务器就采用了这项技术。

图1-8给出了TCP上客户 - 服务器事务的时间系列。我们首先注意到，与图 1-4中UDP上的

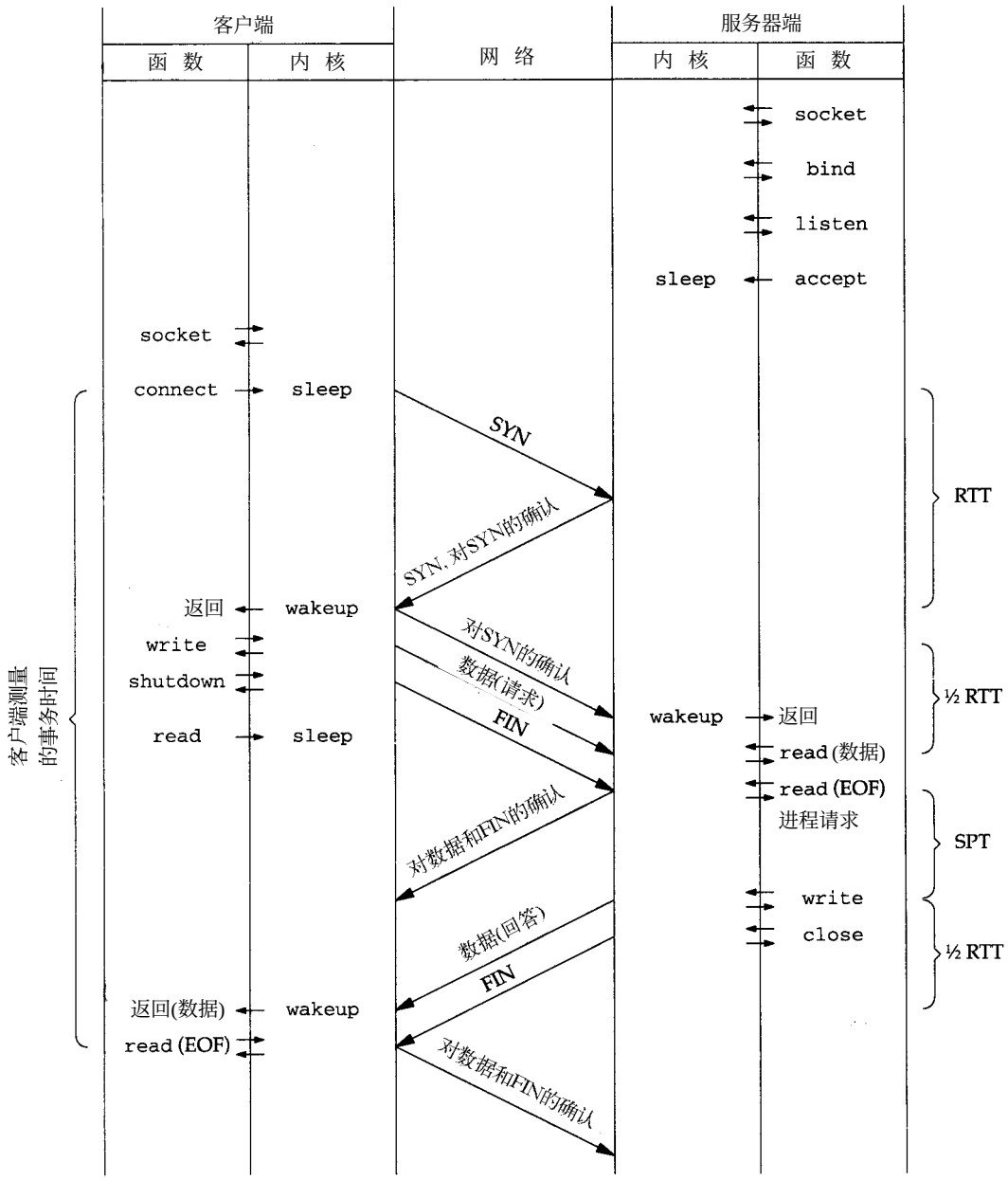

图1-8 TCP上客户 - 服务器事务的时序

事务相比，网络上交换的分组数增加了： TCP上事务的分组数是9，而UDP上的则是2。采用TCP后，客户端测量的事务时间是不少于2 × RTT + SPT。通常，中间三个从客户到服务器的报文段(对服务器SYN的ACK、请求以及客户的FIN)是紧密相连的；后面两个从服务器到客户的报文段(服务器的应答和FIN)也是紧密相连的。这使实际事务时间比从图1-8中看到的更接近2×RTT + SPT。

本例中多出来的一个RTT源于TCP连接建立的时间开销：图1-8中前两个报文段所花的时间。如果TCP可以把建连和发送客户数据以及客户 FIN(图中客户端发出的前四个报文段)合起来，再把服务器的应答和FIN合起来，事务时间就又可以回到RTT + SPT了，这与UDP的一样。事实上，这就是T/TCP中采用的基本技巧。

6. TCP的TIME_WAIT状态

TCP要求，首先发出FIN的一端(我们的例子中是客户)，在通信双方都完全关闭连接之后，仍然要保持在TIME_WAIT状态直至两倍的报文段最大生存时间 (MSL)。MSL的建议值是120秒，也即处于TIME_WATE状态要达到4分钟。当连接处于TIME_WAIT状态时，同一连接(即客户IP地址和端口号，以及服务器IP地址和端口号这4个值相同)不能重复打开(我们在第4章中还要更多地讨论TIME_WAIT状态)。

> 许多基于伯克利代码的TCP实现，在TIME_WAIT状态的保持时间仅仅为60秒，而不是RFC 1122 [Braden 1989]中指定的240秒。在本书的所有计算中，我们还是假定正确的等待周期为240秒。

在我们的例子中，客户端首先发出 FIN，这称为主动关闭，因而 TIME_WAIT状态出现在客户端。在这个状态延续期内， TCP要为这个已经关闭的连接保留一定的状态信息，以便能正确处理那些在网络中延迟一段时间、在连接关闭之后到达的报文段。同样，如果最后一个ACK丢失了，服务器将重传FIN，使客户端重传最后的ACK。

其他一些应用程序，特别是WWW中的HTTP，要求客户程序发送一个专门的命令来指示已经将请求发送完毕(而不是像我们的客户程序那样采用半关闭连接的办法)；接着服务器就发回应答，紧接着就是服务器的 FIN。然后客户程序再发出FIN。这样做与前面所述的不同之处在于，现在的TIME_WAIT状态出现在服务器端而不是客户端。对许多客户访问的繁忙服务器来说，需要保留的状态信息会占用服务器的大量内存。因此，当设计一个事务性客户服务器应用程序时，让连接的哪一端关闭后进入 TIME_WAIT状态值得仔细斟酌。我们还将看到，T/TCP可以让TIME_WAIT状态的延续时间从240秒减少到大约12秒。

7. 减少TCP中的报文段数

像图1-9所示的那样，把数据和控制报文段合并起来可以减少图1-8中所示的TCP报文段数。请注意，这里的第一个报文段中包含有 SYN、数据和FIN，而不像图1-8中那样仅仅是SYN。类似地，服务器的应答和服务器的 FIN也可以合并。虽然这样的分组序列也符合 TCP的规定，但是作者无法在应用程序中利用现有的插口 API使TCP产生这样的报文段序列(因此才在图1-9中客户端产生第一个报文段时和服务器端产生最后一个报文段时标上了问号)；而且据作者所知，也没有哪一个应用程序确实生成了这样的报文段序列。

值得一提的是，尽管我们把报文段的数目由9减少到了5，但客户端观测的事务依然是2×RTT + SPT。这是因为TCP中规定，服务器端的TCP在三次握手结束之前不能向服务器进程提交数据(卷2的第27.9节说明了TCP是如何在连接建立之前将到达的数据进行排队缓存的)。加

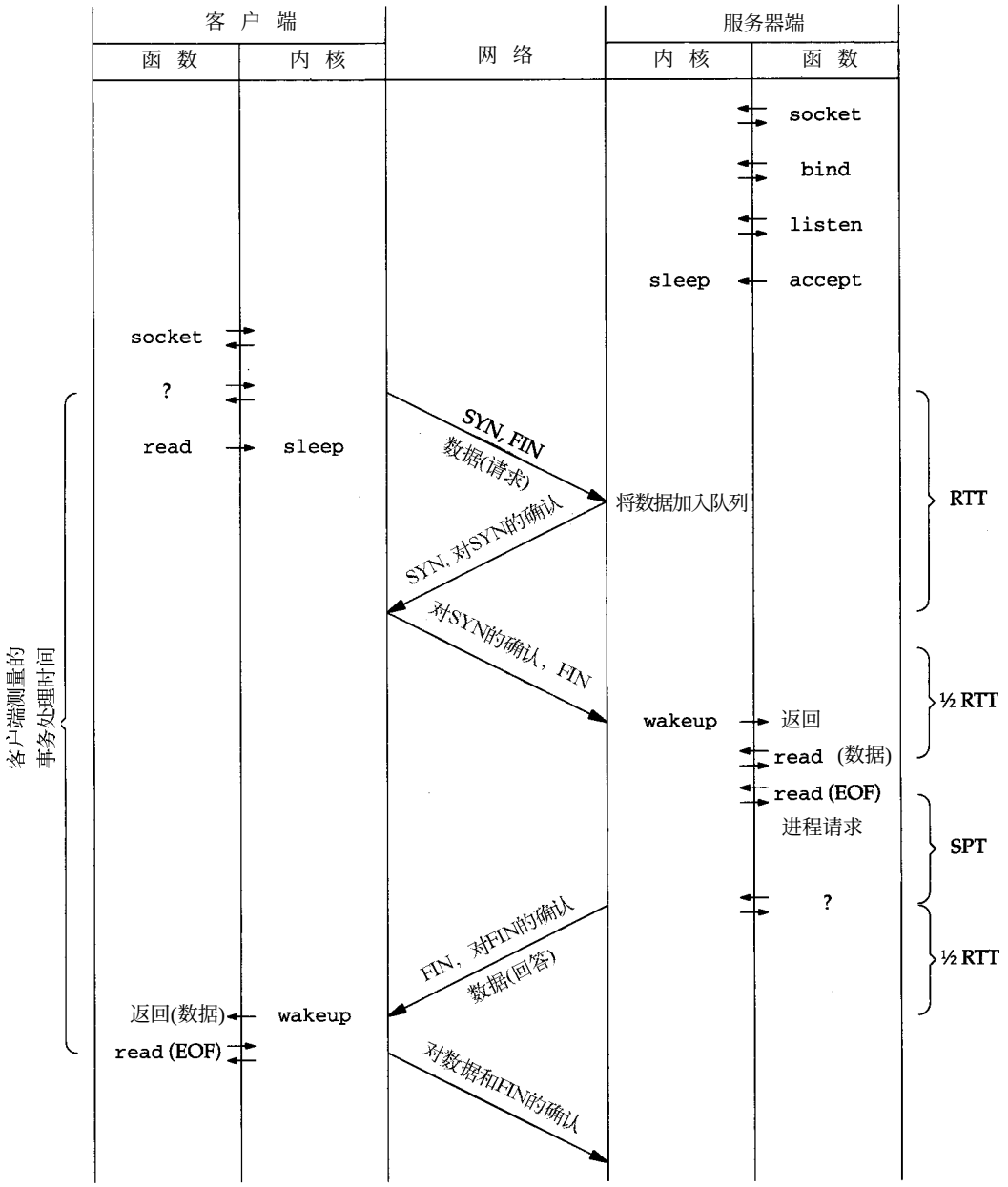

图1-9 最少TCP事务的时序

上这种限制的原因是服务器必须确信来自客户的 SYN是“ 新的 ”，即不是以前某次连接的SYN在网络中延迟一段时间后到达服务器端的。确认过程是这样的：服务器对客户发送的 SYN发送确认，再发出自己的 SYN，然后等待客户对该 SYN的确认。当三次握手完成之后，通信双方就都知道对方的 SYN是新的。由于在三次握手结束之前服务器无法开始处理客户的请求，故分组数的减少并没有缩短客户端测得的事务时间。

下面这段话引自 RFC 1185 [Jacobson, Braden, and Zhang 1990]的附录：“ 注意：使连接能够尽快重复利用是早期 TCP开发的重要目标。之所以有这样的要求是因为当

时人们希望TCP既是应用层事务协议的基础，同时也是面向连接协议的基础。当时讨论中甚至把既包含有SYN和FIN比特，同时又包含数据的报文段叫做‘圣诞树’报文段和‘Kamikaze(敢死队)’报文段。但这种热情很快被泼了冷水，因为人们发现，三次SYN握手和FIN握手意味着一次数据交换至少需要5个分组。而且，TIME_WAIT状态的延续说明同一个连接不可能马上再次打开。于是，再没有人在这个领域做进一步的研究，尽管现在的某些应用程序(比如，简单邮件传送协议，SMTP)经常会产生很短的会话。人们一般都可以采用为每个连接选用不同的端口对的办法来避开重用问题”。

RFC 1379 [Braden 1992b]中写到：“这些‘Kamikaze(敢死队)’报文段不是作为一种支持的服务来提供，而主要用来搞垮其他实验性的TCP！”

作为一个实验，作者编写了一个测试程序，这个程序把SYN与数据和FIN在一个报文段中发出去，即图1-9中的第一个报文段。该报文段发给8个不同版本Unix的标准echo服务器(卷1的第1.12节)，再用Tcpdump观察所交换的数据。其中的7个(4.4BSD、AIX 3.2.2、BSD/OS 2.0、HP-UX 9.01、IRIX System V.3、SunOS 4.1.3和System V Release 4.0)都能正确处理该报文段，另外一个(Solaris 2.4)则把随SYN一起传送的数据扔掉，迫使客户程序重传数据。

那7个系统中的报文段序列与图1-9所描绘的不尽相同。当三次握手结束后，服务器立刻就对客户的数据和FIN发出确认。另外，由于echo服务器无法把数据和FIN捆绑在一起(图1-9中的第四个报文段)发送，结果是发了两个报文段而不只是一个：应答和紧接其后的FIN。因此，报文段的总数是7而不是图1-9中所示的5。我们在3.7节中会进一步讨论与非T/TCP实现的兼容性问题，并给出一些Tcpdump的输出结果。

许多从伯克利演变而来的系统中，服务器无法处理接收到的报文段中只有SYN、FIN，而没有数据、ACK的情况。这个bug使得新创建的插口保持在CLOSE_WAIT状态直到主机重新启动。但这却是一个合法的T/TCP报文段：客户建立起了一个连接，没有发送任何数据，然后就关闭连接。

1.4 T/TCP上的客户-服务器

我们的T/TCP客户-服务器的源代码和上一节的TCP客户-服务器的源代码略有不同，以便能够利用T/TCP的优势。图1-10给出了T/TCP上的客户程序。

ttcpcli.c

```
 1 #include    "cliserv.h"

 2 int
 3 main(int argc, char *argv[])
 4 {                                /* T/TCP client */
 5     struct sockaddr_in serv;
 6     char    request[REQUEST], reply[REPLY];
 7     int     sockfd, n;

 8     if (argc != 2)
 9         err_quit("usage: ttcpcli <IP address of server>");

10     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");
```

图1-10 T/TCP上的事务客户程序

```
12      memset(&serv, 0, sizeof(serv));
13      serv.sin_family = AF_INET;
14      serv.sin_addr.s_addr = inet_addr(argv[1]);
15      serv.sin_port = htons(TCP_SERV_PORT);

16      /* form request[] ... */

17      if (sendto(sockfd, request, REQUEST, MSG_EOF,
18                 (SA) &serv, sizeof(serv)) != REQUEST)
19          err_sys("sendto error");

20      if ((n = read_stream(sockfd, reply, REPLY)) < 0)
21          err_sys("read error");

22      /* process "n" bytes of reply[] ... */

23      exit(0);
24  }
```

ttcpcli.c

图1-10 (续)

1. 创建TCP插口

10-15 对socket函数的调用与TCP上的客户程序一样，在Internet插口地址结构中同样也填入服务器的IP地址和端口号。

2. 向服务器发送请求

17-19 T/TCP上的客户程序不调用connect函数。而是直接调用标准的sendto函数，该函数向服务器发送请求，同时与服务器建立起连接。此外，我们还用 sendto函数的第4个参数指定了一个新的标志MSG_EOF，用以告诉系统内核数据已经发送完毕。这样做就相当于图 1-5中调用shutdown函数，向服务器发送一个FIN。MSG_EOF标志是T/TCP实现中新加入的，不要把它与MSG_EOR标志混淆，后者是基于记录的协议(比如OSI的运输层协议)中用来标志记录结束的。我们将在图1-12中看到，调用sendto函数的结果是客户端的SYN、客户的请求以及FIN都包含在一个报文段中发送出去。换言之，调用一个 sendto函数就实现了connect、write和shutdown三个函数的功能。

3. 读服务器的应答

20-21 读服务器的应答还是用 read_stream函数，与前文讨论过的TCP上的客户程序一样。

图1-11所示的是T/TCP上的服务器程序。

ttcpserv.c

```
1 #include    "cliserv.h"

2 int
3 main()
4 {                                   /* T/TCP server */
5     struct sockaddr_in serv, cli;
6     char    request[REQUEST], reply[REPLY];
7     int     listenfd, sockfd, n, clilen;

8     if ((listenfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
9         err_sys("socket error");
```

图1-11 T/TCP上的事务服务器程序

```
10      memset(&serv, 0, sizeof(serv));
11      serv.sin_family = AF_INET;
12      serv.sin_addr.s_addr = htonl(INADDR_ANY);
13      serv.sin_port = htons(TCP_SERV_PORT);

14      if (bind(listenfd, (SA) &serv, sizeof(serv)) < 0)
15          err_sys("bind error");

16      if (listen(listenfd, SOMAXCONN) < 0)
17          err_sys("listen error");

18      for (;;) {
19          clilen = sizeof(cli);
20          if ((sockfd = accept(listenfd, (SA) &cli, &clilen)) < 0)
21              err_sys("accept error");

22          if ((n = read_stream(sockfd, request, REQUEST)) < 0)
23              err_sys("read error");

24          /* process "n" bytes of request[] and create reply[] ... */

25          if (send(sockfd, reply, REPLY, MSG_EOF) != REPLY)
26              err_sys("send error");

27          close(sockfd);
28      }
29  }
```

ttcpserv.c

图1-11 (续)

这个程序与图1-7中TCP上的服务器程序几乎完全一样：对 socket函数、bind函数、listen函数、accept函数和read_stream函数的调用都一模一样。唯一的不同在于T/TCP上的服务器发送应答时调用的是 send函数，而不是write函数。这样就可以设置MSG_EOF标志，从而可以将服务器的应答和服务器的FIN合并在一起发送。

图1-12所示的是T/TCP上客户-服务器事务的时序图。

T/TCP上的客户测量到的事务时间和UDP上的几乎一样(图1-4)：RTT + SPT。我们估计T/TCP上的时间会比UDP上的时间稍长一点，这是因为TCP协议需要处理的事情比UDP要多一些，而且通信双方都要执行两次 read操作分别读数据和文件结束标志(而UDP环境下双方都只要调用一次recvfrom函数即可)。但是双方主机上这一段额外的处理时间比一次网络往返时间RTT要小得多(我们在1.6节中给出了一些测试数据，用来比较UDP、TCP和T/TCP上的客户-服务器事务的差别)。由此我们可以得出结论：T/TCP上的事务时间要比TCP上的事务小大约一次网络往返时间 RTT。T/TCP中省下来的这个RTT来自于TAO，即TCP加速打开(TCP Accelerated Open)。这种方式跳过了三次握手的过程。下面两章中我们将说明其实现方法；在4.5节中我们还将证明这样做的正确性。

UDP上的事务需要两个分组来传送，T/TCP上的事务需要3个分组，而TCP上的事务则需要9个分组(这些数字的前提是没有分组丢失)。因此，T/TCP不仅缩短了客户端的事务处理时间，而且也减少了网络上传送的分组数。我们希望减少网络上的分组数，因为路由器往往受限于它们可以转发的分组数，而不是每个分组的长度。

概括地讲，T/TCP以一个额外的分组和可以忽略的延续时间为代价，同时具有了可靠性和适应性这两个对网络应用至关重要的特性。

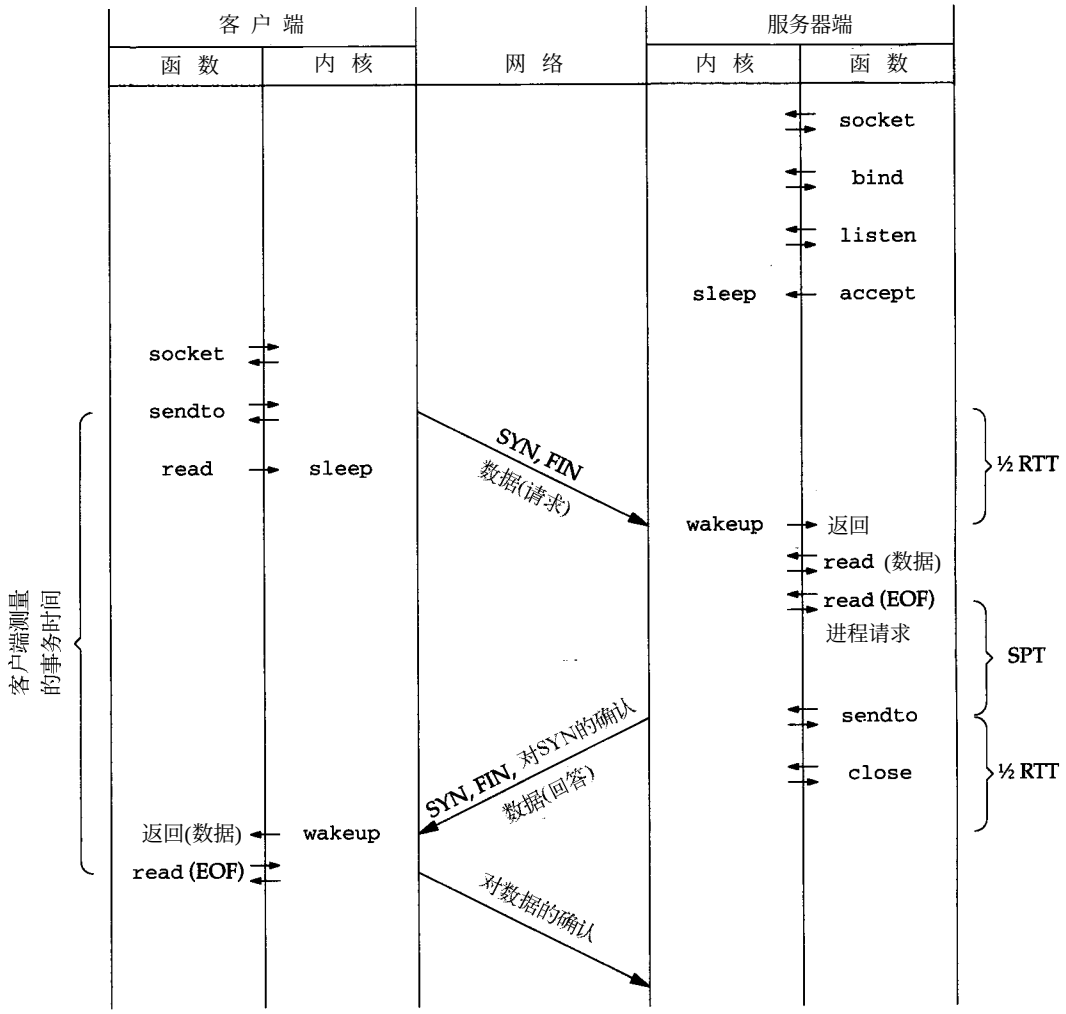

图1-12 T/TCP上客户-服务器事务的时序

1.5 测试网络

图1-13画出了用于验证本书所有例子的测试网络。

书中大多数的示例程序都运行在laptop和bsdi这两个系统上，它们都支持T/TCP协议。图1-13中所有的IP地址都属于B类子网140.252.0.0。所有主机的名字都属于tuc.noao.edu域。noao表示“国家光学空间观测站”，tuc表示Tucson。图中每个方框上部的记号表示在该系统运行的操作系统。

1.6 时间测量程序

我们可以分别测量三种客户-服务器事务的时间，并比较其测量结果。我们要对客户程序作如下改动：

• 在图1-1所示的UDP上客户程序中，我们在即将调用sendto函数前和recvfrom函数刚

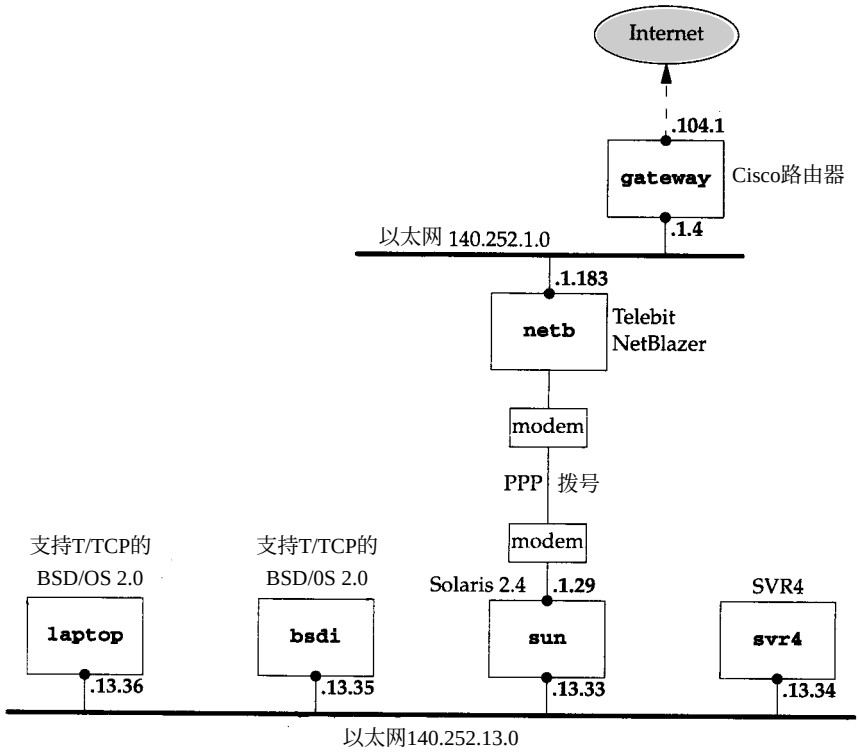

图1-13用于验证本书所有例子的测试网络，所有 IP地址都以140.252打头

刚返回后分别读取当前系统时间。这两个时间的差值即为客户端测得的事务时间。

- 在图1-5所示的TCP上客户程序中，我们在即将调用 `connect`函数前和`read_stream`函数刚刚返回后分别读取当前系统时间。
- 在图1-10所示的T/TCP上客户程序中，我们取当前的系统时间为即将调用 `sendto`函数前和`read_stream`函数刚刚返回后。

图1-14给出了以14种不同长度的请求和应答分别测得的结果。客户和服务器分别为图 1-13中的`bsdi`和`laptop`。附录A中给出了这些测量的细节，并分析了影响结果的因素。

T/TCP上的事务时间总是比同样条件下的 UDP上的事务时间要长几个毫秒 (由于这个时间差是软件造成的，因此这个时间差会随着计算机速度的提高而缩短)。T/TCP协议栈比UDP协议栈所做的操作要多 (图A-8)，而且T/TCP上的客户和服务器要分别调用两次 `read`函数，而UDP上的客户和服务器则只需分别调用一次 `recvfrom`函数。

TCP上的事务时间总是比相同条件下 T/TCP上的事务要长大约 20 ms。其中部分原因是由于TCP建立连接时的三次握手。两个 SYN报文段的长度是 44字节(20字节的IP首部、20字节的标准TCP首部和4字节的TCP MSS选项)。这相当于用户数据为 16字节的Ping；从图A-3可知，其网络往返时间 RTT大约为 10 ms。另外10 ms的差值可能是因为 TCP协议需要处理额外 6个TCP报文段造成的。

因此我们可以得出结论：T/TCP上的事务时间接近、但比 UDP上的事务时间略大，比TCP上的事务时间短至少相当于一个 44字节报文段的网络往返时间。

就客户段测量的事务时间而言，用 T/TCP取代TCP带来的好处依赖于 RTT和SPT之间的关

系。比如，在一个局域网上的RTT为3 ms(如图A-2)，服务器的平均处理时间为500 ms，那么TCP上的事务时间大约为506 ms(2×RTT+SPT)，而T/TCP的事务时间则大约为503 ms。但如果是一个网络往返时间RTT为200ms的广域网(见第14.4节)，服务器处理时间SPT的平均值为100 ms，那么TCP上和T/TCP上的事务时间就分别为大约500 ms和300 ms。我们已经看到，使用T/TCP所需传送的网络分组数少(从图1-8和图1-12的比较中看分别是3个和9个)，因此，不管客户端所测得的事务时间减少了多少，使用T/TCP总是能减少网络分组数。减少了网络分组数就可以减少分组丢失的概率，而在Internet中，分组丢失对整个网络的稳定性有很大影响。

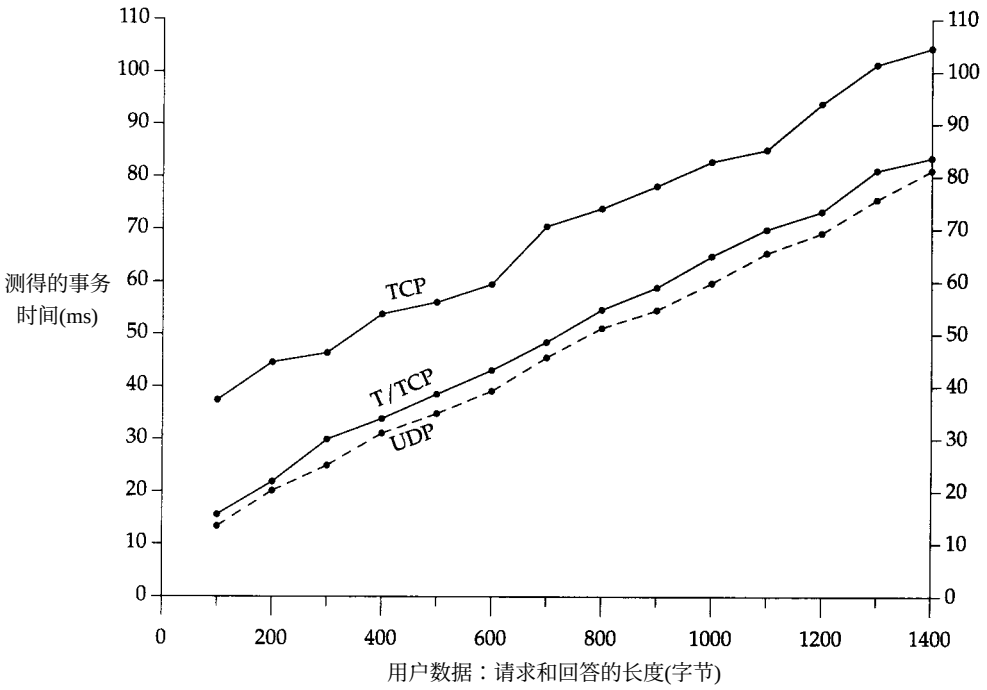

图1-14 UDP、T/TCP和TCP上客户-服务器事务的时间系列

在A.3节里，我们介绍了传播时迟和带宽的差异。这两者对RTT都有影响；但是当网络变快以后，传播时迟的影响也就变大了。此外，传播时延是我们几乎无法控制的，因为它的大小取决于客户和服务器之间的信号传播距离和光在介质中的传播速度。于是，在网络速率越来越快的条件下，省下一个RTT的时间就显得尤为可贵，使用T/TCP的相对好处也就越发明显。

现在可以公开获得并支持T/TCP的用于测量网络性能的工具：

```
http://www.cup.hp.com/netperf/netperfpage.html
```

1.7 应用

T/TCP给所有TCP上的应用程序带来的第一个好处就是可以缩短TIME_WAIT状态的持续时间。这样，一般情况下协议必须处理的控制块也跟着少了。4.4节详细介绍了T/TCP协议的这个特性。现在我们可以这样说：对于连接时间很短(典型值为小于2分钟)的所有TCP应用程

序，如果通信双方的主机都支持T/TCP的话，它们都将因使用该协议而获益。

使用T/TCP的最大好处或许在于避免了三次握手过程，对于那些交换的数据量比较小的应用程序，T/TCP减少的时延将给它们带来好处。我们将给出几个例子来说明这一点（附录B谈到了利用T/TCP来避免三次握手过程要对应用程序做怎样的修改）。

1. WWW：超文本传输协议

WWW及其所依赖的HTTP协议(将在第13章介绍该协议)将可能大大地受益于T/TCP协议。参考书[Mogul 1995b]中指出：“然而，构成Web应用传输时延的主要因素是网络通信……即便我们无法提高光的传播速度，但我们至少应该想办法减少一次交互过程中的往返传输次数。当前Web网中使用的超文本传输协议(HTTP)实际上造成了大量不必要的往返传输”。

比如，[Mogul 1995b]中对随机抽取的200 000个HTTP请求的统计发现，应答长度的中值为1770字节(通常使用中值而不使用均值，这是因为很少出现的大文件会使均值变大)。Mogul还引用了另一个例子。该例随机抽样了大约150万个请求，其应答的长度中值为958字节。客户的请求一般很短：在100~300字节之间。

典型的HTTP客户-服务器事务和图1-8所示的很相似。客户端主动打开，向服务器发出很短的请求，服务器收到请求后发出应答，然后服务器关闭连接。这种情况非常适于使用T/TCP协议，把客户端的SYN和客户的请求合并在一起传送省去三次握手中的往返时间。这也还减少了网络上的分组数，而这对于已经非常巨大的Web通信量来说也是很有意义的。

2. FTP数据连接

FTP数据连接也会从使用T/TCP协议中获益。从一项对Internet通信量的统计调查中，[Paxson 1994b]发现平均每个FTP数据连接所传输的数据量约为3 000字节。卷1的第323页给出了FTP数据连接的一个例子。虽然例子中的数据流是单向的，但其传输过程还是与图1-12所示的十分相似。采用T/TCP后，图中的8个报文段减少到了3个。

3. 域名服务系统(DNS)

DNS客户的查询请求是用UDP传送到DNS服务器的。服务器仍然用UDP发送给客户的应答。但如果应答超过512字节，那么只有前512字节会在应答中返回给客户，同时在应答中有“truncated(截断)”标志，表示还有信息要传给客户。于是客户用TCP向服务器重新发送查询请求，而后服务器用TCP向客户传送完整的应答。

采用这项技术的原因是不能保证特定的主机能够重组长度超过576字节的IP数据报(实际上，许多UDP应用程序都把用户数据的长度限定在512字节以内，以保证不超过576字节的限制)。由于TCP是一个字节流协议，应答数据量再大也不会有问题。发送方TCP会根据连接建立时对等端声明的报文段最大长度(MSS)限制，把应用程序的应答数据分割成适当长度的报文段发给对方。接收方TCP会把这些报文段拼接起来，并以应用程序读取时指定的数据长度交给接收的应用程序。

DNS的客户和服务器可以利用T/TCP，既达到UDP的请求-应答速度，又具有TCP的所有好处。

4. 远程过程调用(RPC)

在所有论述将传输协议用于事务的论文中，无不将RPC作为一个候选的应用协议。RPC中客户要向载有待执行程序的服务器发送请求，请求中带有客户给定的参数；服务器的应答中包括过程执行后所返回的结果。参考书[Stevens 1994]的第29.2节中讨论了Sun RPC。

RPC的数据包往往会非常大，必须给 RPC协议增加可靠性，使其能在像 UDP这样不保证可靠性的协议上运行，同时还要避免 TCP的三次握手。使用T/TCP协议就能实现这一目标，既有TCP的可靠性，又没有三次握手的开销。

所有建立在RPC基础上的应用程序，比如网络文件系统 (NFS)等都可以采用T/TCP协议。

1.8 历史

RFC 938 [Miller 1985]是较早讲述事务的 RFC文档之一。该文档中规定了 IRTP，即：Internet 可靠的事务协议，能保证数据分组的可靠、按顺序提交。该文档中把事务定义为一个短小的、自包含的报文；而 IRTF定义了任意两台主机 (即IP地址)之间持续存在的优选连接，当其中任何一台主机重新启动后，该连接都重新同步。 IRTF协议位于IP协议之上，并定义了专门的8字节首部。

RFC 955 [Braden 1985]本质上并未规定任何协议，而只是给出了事务协议的一些设计准则。它认为UDP和TCP这两个主流的运输层协议所提供的业务相差太大，而事务协议正好填补TCP和UDP之间的空档。该RFC文档把事务定义为一次简单的报文交换：一个请求发给服务器，然后一个应答发回到客户。它还认为各种事务都有如下特征：不对称的模式 (一端是服务器，另一端是客户)、单工数据传递(任一时刻都只有一个方向有数据传输)、持续时间短(可能延续几十秒，但绝不可能几小时)、时延小、数据分组少以及面向报文 (不是字节流)。

该RFC中列举了域名服务系统 DNS的例子。它认为，在考虑是用 UDP还是用TCP作为域名服务系统的运输层协议时，设计者往往陷入两难的境地。一个理想的解决方案应该既能提供可靠的数据传输，又不需要专门地建立和释放连接，不需要报文的分段和重组 (从而应用程序不再需要知道像576这类的神秘数字)，同时还能使两端的空闲状态所处时间最短。 TCP什么都好，只可惜它需要建立和释放连接。

另一个相关的协议是RDP，即可靠数据协议。该协议在RFC 908 [Velten，inden，and Sax 1984]中定义，后来又更新为RFC 1151 [Patridge and Hinden 1990]。与RDP实现有关的经验在参考文献[Patridge 1987] 中可以找到。参考文献[Patridge 1990a]中对RDP有如下评价：“ 当人们寻求一个可靠的数据报协议时，他们通常是想要一个事务协议，一个能够让他们与多个远端系统可靠地交换数据单元的协议，一个类似于可靠 UDP的协议。RDP应该看作是一个面向记录的TCP协议，它利用连接可靠地传输有格式数据块流。 RDP并不是一个事务协议。” (RDP不是一个事务协议的理由是因为它和 TCP一样采用了三次握手技术)。

RDP使用通常的插口应用编程接口 (API)。与TCP类似， RDP提供流插口接口 (SOCK_STREAM)。另外， RDP还提供SOCK_RDM插口类型(可靠的报文提交)和SOCK_SEQPACKET插口类型(有序的分组)。

VMTP，即通用报文事务协议，是在 RFC 1045 [Cheriton 1998]中规定的，是一个专门用于事务的协议，就像远程过程调用一样。像 IRTP和RDP那样， VMTP也是IP之上的运输层协议，但VMTP还支持多播通信，这个特性是 T/TCP以及本节提到过的其他协议所不具备的 (参考文献[Floyd et al. 1995]中有不同意见，他们认为提供可靠的多播通信是应用层的任务，而不是运输层的任务)。

VMTP还为应用程序提供不一样的应用编程接口 API，其插口类型为SOCK_TRANSACT。

具体定义详见RFC 1045。

虽然T/TCP的许多概念早在RFC 955中就已经出现，但直到RFC 1379 [Braden 1992b]发布才正式有了T/TCP的第一个规范。该RFC文档定义了T/TCP的概念，接下来的RFC 1644 [Braden 1994]给出了更多的细节，并讨论了一些实现问题。

图1-15比较了实现各种运输层协议分别都需要多少行C源代码。

协　议	源代码行数
UDP(卷2)	800
RDP	2 700
TCP(卷2)	4 500
T/TCP模式的TCP	5 700
VMTP	21 000

图1-15　实现各种运输层协议所需要的源代码行数

为支持T/TCP所需增加的源代码行数(大约1200行)是UDP协议源代码行数的1.5倍。为使4.4BSD支持多播通信，需要增加大约2000行源代码(设备驱动程序的改变和支持多播路由所需要的代码行数尚未计算在内)。

VMTP可以从`ftp://gregorio.stanford.edu/vmtp-ip`得到。RDP通常还得不到。

1.9 实现

第一个T/TCP实现是由Bob Braden和Liming Wei在南加州大学的信息科学学院(USC ISI)完成的。该项工作得到了国家科学基金NSF的部分资助，批准号为NCR-8 922 231。该实现是为SunOS 4.1.3(从伯克利演变而来的内核)做的，1994年9月就可以用匿名的FTP得到了。SunOS 4.1.3的源代码补丁可以从`ftp://ftp.isi.edu/pub/braden/TTCP.tar.Z`得到，但你必须有SunOS内核的源代码才能应用这些补丁。

Twente大学(荷兰)的Andras Olah修改了USC ISI的实现，并于1995年3月将其在FreeBSD 2.0版中发布。FreeBSD 2.0中的网络代码是基于4.4BSD-Lite版的(卷2中有介绍)。图1-16给出了各种BSD版本的演变历程。与路由表(我们将在第6章中讨论)有关的所有工作都是由麻省理工学院(Massachusetts Institute of Technology)的Garrett Wollman完成的。FreeBSD实现的有关信息可以从`http://www.freebsd.org`得到。

本书作者把FreeBSD实现移植到了BSD/OS 2.0内核(该内核也基于4.4BSD-Lite中的网络代码)中，也就是运行在主机`bsdi`和`laptop`(图1-13中)中的代码，本书从头至尾都用它们。为了支持T/TCP而对BSD/OS所做的修改,可以从作者的个人主页里找到：`http://www.noao.edu/~rstevens`。

图1-16给出了各个BSD版本的演变历程，其中还标出了重要的TCP/IP特性。图中左边显示的是可以公开得到源代码的版本，其中有所有网络代码：协议本身、网络接口的内核例程以及许多应用程序和实用工具(比如Telnet和FTP)。

本书中所描述的T/TCP实现的基础软件的正式名称是4.4BSD-Lite，但我们一般简称其为Net/3。还要说明的是，可以公开得到的Net/3版本中不包括本书所述为支持T/TCP而做的修改。

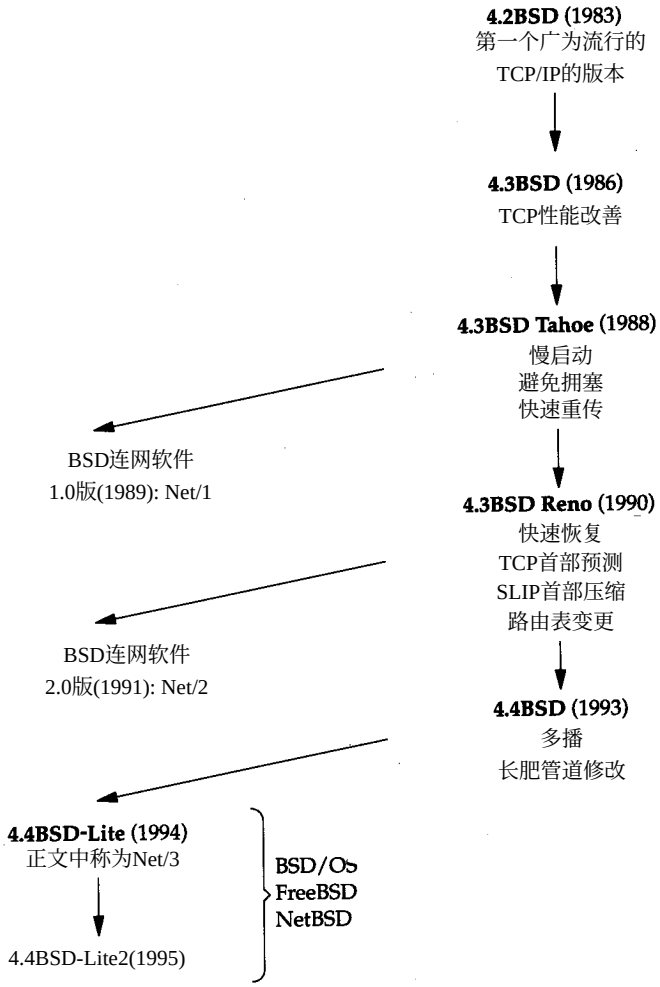

图1-16 带有重要TCP/IP特性的各种BSD发行版

当提到Net/3这个术语时，实际所指的就是这个不包含 T/TCP的、可公开得到的版本。

4.4BSD-Lite2是1995年对4.4BSD-Lite的升级。从网络部分来看，从 Lite到Lite2仅仅是解决了一些bug，以及少量的改进(比如我们将在14.9节中介绍的坚持探测的超时)。我们给出了3个基于Lite代码的系统：BSD/OS、FreeBSD和NetBSD。本书所述全部都是基于 Lite代码的，但所有以上的3个版本都应该在下一个主要版本中升级到 Lite2。可以从下面的 Walnut Creek CDROM站点得到含有Lite2版本的光盘：`http://www.cdrom.com`。

本书全书都将用“从伯克利演变而来的实现”这个术语指称厂商的实现，比如 SunOS、SVR4(System V Release 4)和AIX，因为所有这些实现的TCP/IP代码最初都来自于伯克利源代码，它们之间有许多共同点，甚至连程序中的差错都相同！

1.10 小结

本章的目的是要让读者相信 T/TCP的确为许多实际中的网络应用问题提供了一个解决方案。我们从比较一个分别用 UDP、TCP和T/TCP编写的、简单的客户-服务器程序开始。用

UDP协议需要交换两个分组，用TCP需要9个，而用T/TCP需要3个。我们还发现，用T/TCP和用UDP时在客户端测得的事务时间相差无几。图1-14所示的时间测量结果证明了我们的结论。除了达到UDP的性能之外，T/TCP还具有可靠性和适应性，这两点都是对UDP的重大改进。

T/TCP因为避免了常规TCP中的三次握手而获得上述各种优点。为了利用这些优点，客户和服务器程序在应用T/TCP时必须对源代码做一些简单的改动，主要是在客户端用`sendto`函数代替`connect`、`write`和`shutdown`三个函数。

在后面的3章中，我们将研究协议是如何工作的，同时还会研究更多的T/TCP应用例子。