第九章 网络编程概要

本章概要阐述网络应用程序的编程技术，包括客户/服务器系统的概念、网络编程的标准接口函数 socket API、高性能服务器设计技术，并给出一些实际程序的例子。本章的目的是为读者建立起网络程序的结构框架，并概要指出网络编程需要考虑的主要因素和技术。然而，网络编程是一个博大精深的领域，需要长期认真训练并积累大量细节材料才能熟练掌握，不是这样一本基础性教程用一两章就可以做到的，为此我们在进一步学习指南一节给出了供读者深入学习的材料。

9.1 客户/服务器系统

客户/服务器(client/server)是一种分布式系统的模式，按照这一模式，组成一个分布式软件的所有程序在功能上划分为两类：客户程序和服务器程序。客户程序在运行期间根据其需要向某个服务器程序(今后简称为服务器)请求特定的服务，服务器响应该请求、提供所要求的服务。服务器根据其所提供的服务的特点专门设计，这类服务一般都较复杂，因此每个服务器都专门提供一种最主要的服务。在大多数分布式系统中，极少数服务器集中管理和控制系统的特定资源，共大多数客户程序使用，因此服务器要求有同时接受和处理多个客户程序请求的能力，以提高对资源访问的吞吐量[1]。另外，由于服务器在分布式系统中的特殊角色，高可靠性也常常是对服务器的关键性要求。

常见的服务器的例子有 Web 服务器、FTP 服务器、NFS 文件服务器、消息服务器(如 MS Exchange Server)、数据库服务器等。

在行为方式上，客户是主动的，服务器的被动的。客户根据需要向特定的服务器提出服务请求，服务器被动地响应到达的请求，在没有任何请求到达时即处于空闲状态，如图 9-1。这种工作模式实际上也并不限于网络程序，现代软件系统甚至硬件系统有大量这类工作模式的例子，但不同的应用环境中影响客户/服务器设计的因素及具体技术有本质的不同。我们在本章当然是就网络环境来阐述客户/服务器软件的设计与实现技术，这些技术与其它环境中的客户/服务器软件设计与实现，例如操作系统内核中的客户/服务器软件很少有相似之处。

[1] 服务器吞吐量常以单位时间内所能处理的客户请求的数量来度量。

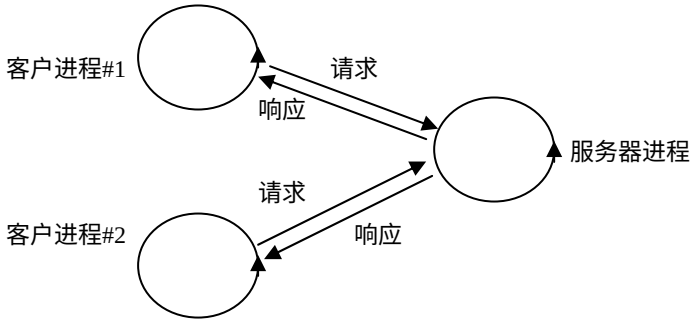


图 9-1 客户/服务器系统

最后说明一点，关于软件系统中客户/服务器角色的划分虽然不是绝对的，即某些服务器为完全响应客户请求可以去请求另一个服务器，以此类推可以形成一个客户/服务器请求链，其中除最后一个是纯粹服务器外，中间环节都是客户-服务器双重角色，然而出于性能及可靠性的考虑一般应该避免这种设计，让服务器尽可能地不依赖其它程序来完成服务响应。

9.2 Socket API

9.2.1 socket 模型

TCP 或 UDP 实现传输服务，它们位于操作系统内核。从应用程序的观点看，这类服务是操作系统可被调用的功能之一，问题是应用程序如何调用这类功能？

和操作系统提供功能其它调用的方式类似，首先需要将传输服务做一合理的抽象，然后依据这一抽象模型定义相应的接口函数，即API。针对传输服务，一个普遍适用的、统一的抽象模型是基于socket[2]的、端-端的、字节流。具体地讲，这一模型有以下涵义[3]：

第一，两个通讯进程各自创建一个 socket 对象，通过适当定义的参数告知 socket 对象通讯对方是谁、以什么协议(例如 TCP 还是 UDP)进行通讯、通讯过程中的某些输入/输出问题如何处理(例如以 TCP 传输数据时，是有任何字节到达时即通知应用程序，还是到达指定的临界字节数后才这样做)，等。

第二，一旦通讯开始，socket 对象负责处理传输过程中一切事务，例如当以 TCP 通讯时，对到达数据的确认、对丢失数据的重传、对 TCP 流量的窗口调节等等均由 socket 对象控制，所有这些对应用程序都是透明的。socket 对象向应用程序所呈现的是一个完全可靠的通讯过

[2] 中文文献常译为套接字。
[3] 请读者回顾面向对象分析与设计的概念, socket模型的特点实质上都是面向对象设计所具备的特点。

程，无须应用程序处理任何传输细节。

第三，socket 对象在网络上传输的仅仅是字节流，即 socket 不对被传输信息的结构做任何假设，所有关于被传输信息的结构和含义的解释都由应用程序完成。

第四，socket-对象为应用程序提供通讯功能的方式与具体协议族无关。事实上，这一模型既适合 TCP/IP，也同样适合其它如 OSI 协议族。

在操作系统内核，socket 对象被实现为操作系统的文件对象。事实上，从应用程序的观点看，socket 对象与一切文件对象所提供的服务都一样，都是输入/输出服务，只不过具体输入/输出的操作性涵义不同，但从面向对象系统设计的角度，这些不同的操作性涵义正是应该向应用程序隐藏而非显露的部分。作为这一实现方式的结果，每个 socket 对象在创建后(和文件对象一样)以一个唯一的文件描述符来标识，在这之后所有的调用都带以该文件描述符做参数。在这些调用—它们本质上都是 socket 对象上的方法(即 method，面向对象的术语)—中，直接实现输入/输出功能的就是 read 和 write，这两个函数的形式与文件对象上的 read 和 write 函数完全一样。

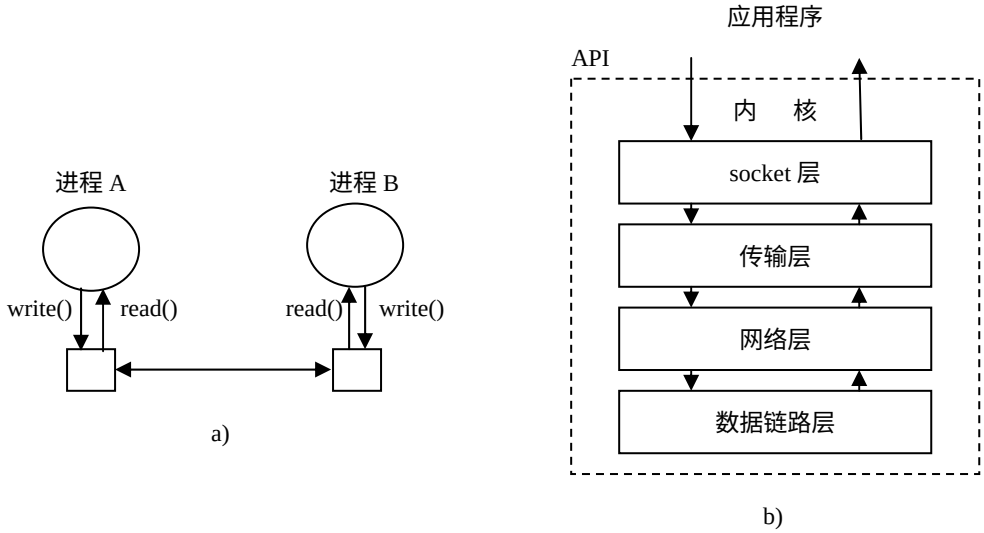


图 9-2 socket 模型

9.2.1 socket API

这一小节概要解释几个最常用的 socket API-函数的主要功能。在具体应用时，读者需要参考更详细的材料，特别是每个函数的错误返回代码，这对编写实际应用程序是

int socket(int afam, int type, int protocol)

创建一个socket对象，返回socket对象的文件描述符。参数afam指定协议族，例如

AF_INET[4]表示因特网协议；参数type表示协议类型，例如 SOCK_STREAM表示协议族中面向连接的协议，SOCK_DGRAM 表示协议族中的非连接性协议；参数protocol一般为 0，表示前两个参数所界定的标准协议。因此，要创建一个TCP socket对象，调用socket(AF_INET, SOCK_STREAM, 0)；创建UDP对象，调用socket(AF_INET, SOCK_DGRAM, 0)。

int bind(int sock, struct sockaddr* localaddr, int addrlen)

对 socket 对象指定地址及传输层端口号，例如对 AF_INET 协议族上的 socket 对象指定就是指定 IP 地址和 TCP 或 UDP 端口号。注意即使程序不调用该函数，操作系统也会自动分配 IP 地址和端口号到新创建的每个 socket 对象，因此这一函数对服务器程序最有用。

int connect(int sock, struct sockaddr* remoteaddr, int addrlen)

从描述符sock所标识的本地socket对象到remoteaddr所指示的远程对象建立连接。如果sock所标识的本地socket对象由socket(AF_INET, SOCK_STREAM, 0)所创建，则这一函数所请求建立的就是TCP连接[5]。参数remoteaddr中含对方进程的IP地址和TCP端口号。读者不难想象，函数connect主要是被客户进程调用。

int listen(int sock, int queuelen)

listen 使 TCP-socket 对象 sock 准备接受到达的连接请求。参数 queuelen 指示能在该 socket 对象上同时排队等待连接的到达请求的最大数量。

int accept(int sock, struct sockaddr* remoteaddr, int* paddrlen)

这是任何一个基于 TCP 的服务器进程都必须调用的函数，sock 是准备接受连接请求的 socket 对象的描述符(在这一对象上已调用过 listen 函数)。调用 accept 函数的进程或线程进入睡眠，直到有 TCP 连接请求到达并且该连接被正确建立后才被唤醒，这时建立 TCP 连接的三握手过程已经完成，TCP 连接已被建立并且被赋予一个新的(由操作系统自动创建的) socket 对象，accept 的返回值正是该对象的文件描述符。结构*remoteaddr 中含有连接对方的 IP 地址和 TCP 端口号，*paddrlen 是这一结构的实际长度，因此读者不难理解，指针参数 remoteaddr 和 paddrlen 所指向的空间必须在调用 accept 之前已经被分配。

对服务器进程，随后的输入/输出以及关闭连接这些调用都是在 accept 函数返回的那个

[4] 在某些操作系统上该常数名字为PF_INET, 请读者注意具体的编程手册。AF_xxx表示Address Family, PF_xxx 表示Protocol Family。

[5] connect()同样可以在UDP-socket上被调用, 但意义不同, 这里不详细阐述了。

socket 对象上进行的。

int read(int sock, char* buff, int size)

int write(int sock, char* buff, int size)

这两个函数并非socket API特有，它们就是文件API，其参数涵义与文件输入/输出时完全相同，只不过现在描述符sock所表示的内核对象是socket对象。和accept一样，这两个函数是阻塞式的，即它们使调用进程或线程睡眠直到其操作完成后返回[6]。

请读者注意，在 UDP 和 TCP-socket 对象上这两个函数都可以应用，但其操作性涵义有微妙的区别，例如在 UDP-socket 对象上，操作系统在数据发送后便使 write 正常返回，但在 TCP-socket 对象上是在数据被对方确认后才使 write 正常返回，否则使 write 返回错误(write 的返回值为负数)。另外，read/write 函数的行为受 socket 对象配置选项(socket option)控制，例如可以人为设置接收临界水平，使得仅当到达的 TCP 字节总数超过该临界水平时 read 函数才返回，否则一直阻塞；同样，可以设置发送临界水平，使发送缓冲区内积累的字节数超过该临界水平时才真正通过 TCP 发送数据，否则 write 将数据写入发送缓冲区后便返回。

习惯上常对 TCP 通讯使用这两个函数，对 UDP 则使用下面两个函数。

recvfrom(int sock, char* buff, int size, int flags, struct sockaddr* from, int* fromlen)

sendto(int sock, char* buff, int size, int flags, struct sockaddr* to, int tolen)

这两个函数在 UDP-socket 对象上接收和发送 UDP 数据报，也都是是阻塞式函数。

int close(int sock)

在调用这一函数后，socket 对象不再可用，即不能再在该 socket 对象上调用 read/write 这类函数,否则将返回错误。如果是 TCP-socket 对象，从本地 socket 对象到对方 socket 对象的 TCP 连接将被关闭。注意这时并未关闭从对方 socket 对象到本地 socket 对象的 TCP 连接，但本地 socket 对象的内部状态已被置为不可用！这一点和下面的 shutdown 函数截然不同。

如果调用 close 时该 socket 对象上尚有数据等待被读出，这时会发生什么？这时操作系统简单地释放这些数据了事，然后关闭 TCP 连接并置 socket 对象内部状态为不可用。

int shutdown(int sock, int flags)

这是一个较 close 更灵活的函数，它关闭本地 socket 对象到对方 socket 对象的 TCP 连接，

[6] 读者不难想象, write引起进程阻塞的机率远小于read, 除非因为某些原因需要反复重发。

这意味着在调用 shutdown 之后不能在本地对象上调用 write，但还可以继续调用 read 正常接受对方发送的数据。

int setsockopt(int sock, int level, int option, char* optval, optlen)

这是一个非常有用的函数，尽管本身非常简单，它的作用在于可以设置大量的 socket 对象配置选项，这些选项用来控制 socket 对象的行为，实际上是 TCP/IP 协议的行为和各种参数。我们不能在此详细列举这些选项的涵义和用法，感兴趣的读者可以查阅本章后面列举的 Stevens 关于网络编程的著作。与这一函数配对的是获取 socket 对象当前配置选项值的函数 getsockopt。

其他辅助函数和宏包括 gethostbyaddr、gethostbyname、gethostname、getpeername、getprotobyname、getprotobynumber、getservbyname、getservbyport、getsockname、htonl、htons、ntohl、ntohs。

9.3 基于 UDP 的客户/服务器系统

图 9-3 是基于 UDP 的客户/服务器进程相互作用的一般性过程，图中不仅画出应用程序(外侧的两条垂直线)的函数调用过程，而且特别画出了内核中的 UDP 过程(内侧的两条垂直线)，

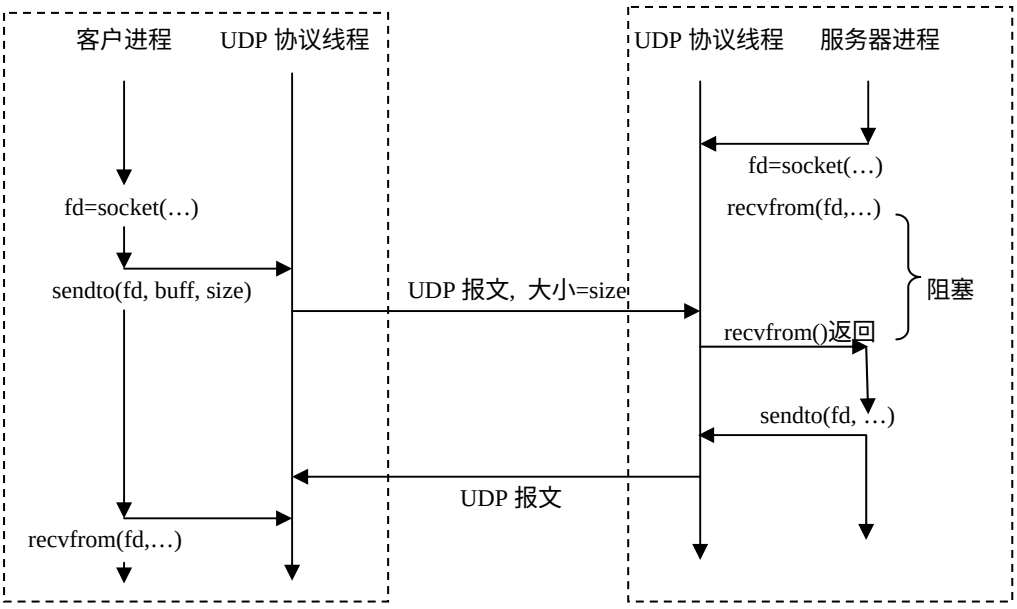


图 9-3 基于 UDP 的客户/服务器过程

目的是使读者了解 socket API 是如何引发真正的 UDP 通讯过程的。

UDP-服务器常见的程序结构如下面这样：

```
int    fd=socket(AF_INET, SOCK_DGRM, 0);

bind(fd, …);
while(1){
      recvfrom(fd, buff, size, ….);
      request_proc(buff, size, …);
      sendto(fd,…);
}
```

注意为了解释主要概念，以上程序没有检查任何 API 的错误返回代码，这在实际编程时是不允许的：必须检查 API 可能返回的错误并最适当处理。实际上，这尤其是服务器编程的重要方面。

正如读者在前面的章节已经理解的，UDP 的最大特点是简单，因此使用 UDP 实现客户/服务器的突出优点是响应速度快，但 UDP 的另一个突出特点是不可靠，因此某些要求高性能同时也要求一定传输可靠性的服务器常基于UDP实现，同时加入一些简单实用的可靠性机制，例如可以仿照 TCP 来实现顺序检查、消息丢失-重传机制。当然，这类补偿机制不应过于复杂，否则不如基于 TCP 来实现服务器。另外一些情形，如视频会议这类要求支持组播的应用，UDP 则是唯一的选择。

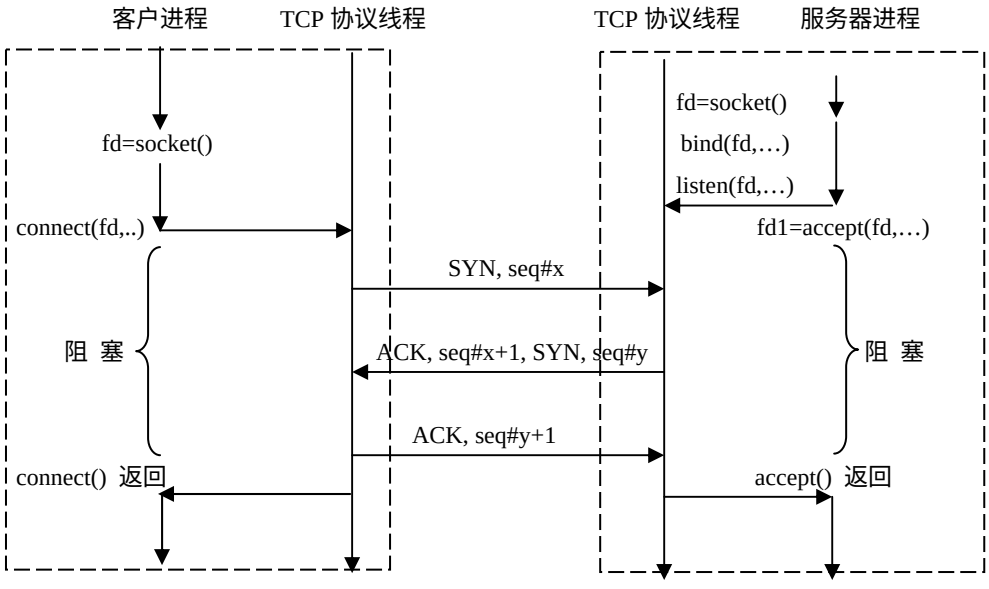
9.4 基于 TCP 的客户/服务器系统

图 9-4 是基于 TCP 的客户/服务器进程相互作用的一般性过程，与图 9-3 一样，图中画出了应用程序(外侧的两条垂直线)的函数调用过程和内核中的 TCP 过程(内侧的两条垂直线)之间的关系。TCP-客户进程一般比 TCP 服务器简单，典型的结构如下：

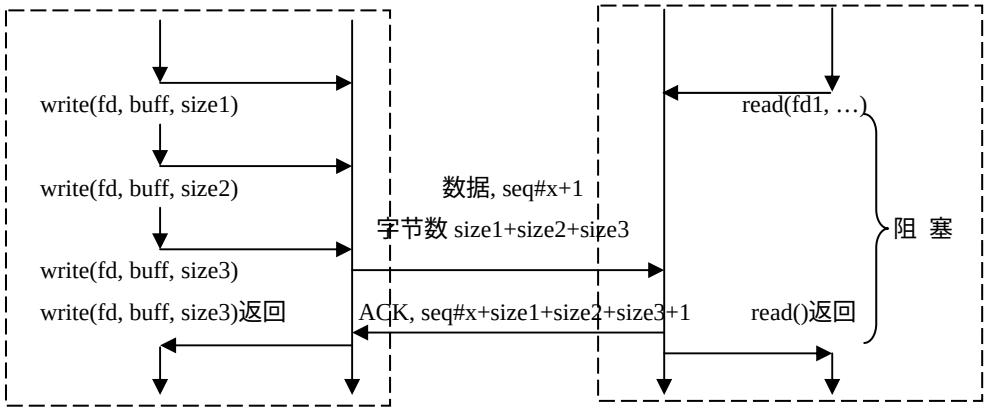
```
int    fd=socket(AF_INET, SOCK_STREAM, 0);

connect(fd, …);
write(fd, …);
read(fd, buff, size);
close(fd);
respond_proc(buff, size, …); /*解析服务器的响应*/
```

最简单的 TCP-服务器进程的结构如下：

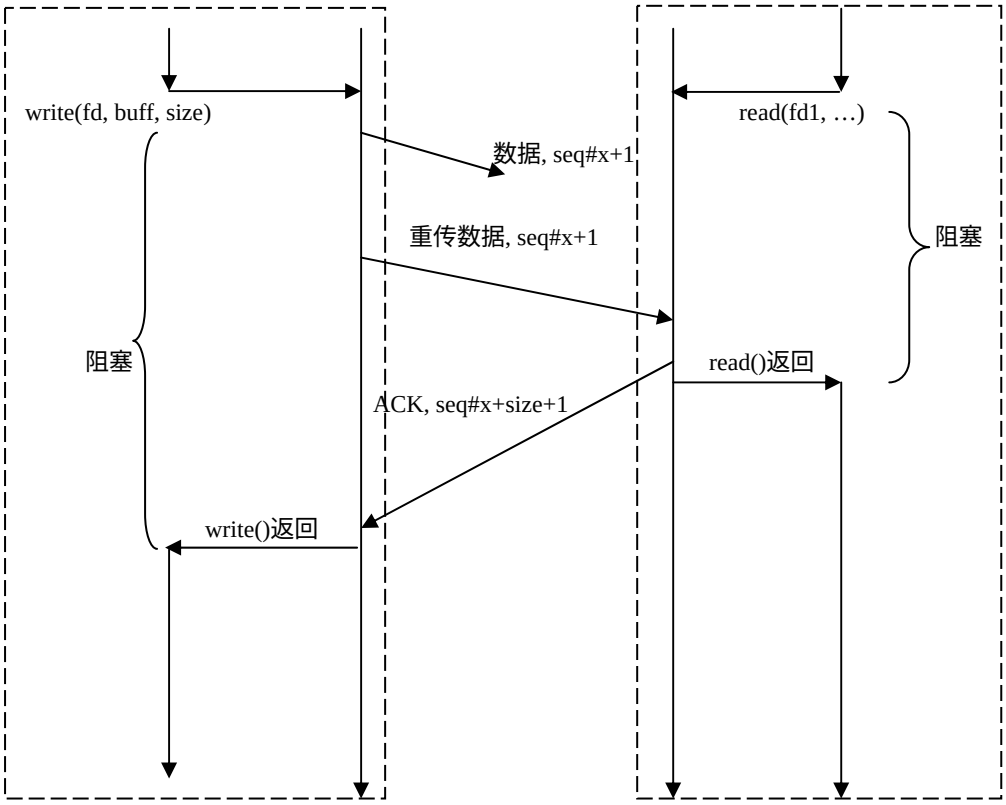
```
int    fd=socket(AF_INET, SOCK_STREAM, 0);
int fd1;

bind(fd, …);
listen(fd,…);
while(1){
      fd1=accept(fd, ….);
      read(fd1, buff, size);
       /*解析客户请求并响应*/
      ret = request_proc(buff,size,…);
      if(ret>=0){ /*ret<0 表示有错误，例如非法请求*/
          write(fd1,…);
      }
      close(fd1);
 }
```



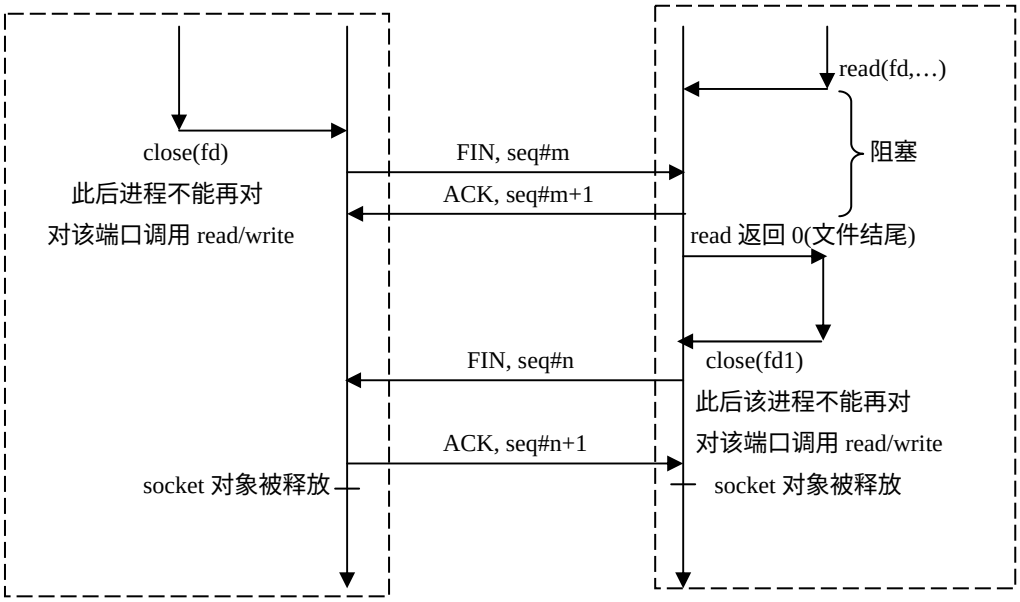
a)



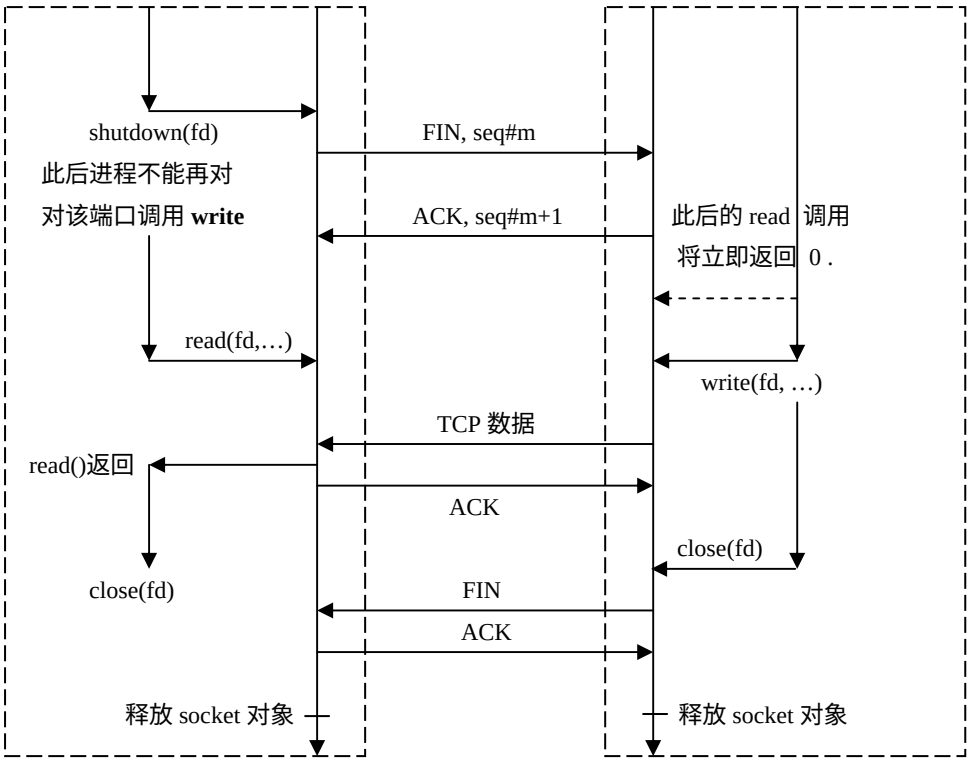
b)


write(fd, buff, size)
数据, seq#x+1
重传数据, seq#x+1
阻塞
ACK, seq#x+size+1
write()返回
read(fd1, …)
阻塞
read()返回

c)


close(fd)
此后进程不能再对
对该端口调用 read/write
FIN, seq#m
ACK, seq#m+1
FIN, seq#n
ACK, seq#n+1
socket 对象被释放
read(fd,…)
阻塞
read 返回 0(文件结尾)
close(fd1)
此后该进程不能再对
对该端口调用 read/write
socket 对象被释放

d)



e)

图 9-4 基于 TCP 的客户/服务器过程

a)建立 TCP 连接 b)缓冲-发送 c)数据成批发送 d)终止连接: 简单终止 e)终止连接: 单向终止

9.5 高性能服务器程序设计

阅读上一节的 TCP 服务器程序，读者会立刻意识到这是一个典型的串行服务器，它按照到达的顺序依此建立 TCP 连接，一个接一个地处理通过这些连接到来的服务请求。当一个连接上的请求没有处理完时，不会去处理其它连接上的服务请求，哪怕其它连接上的服务要比当前连接上的服务简单得多。如果当前连接上的服务处理需要服务器阻塞一段时间，例如检索很大的磁盘文件或访问大型数据库，服务器也只能经过睡眠-唤醒过程直到当前服务请求被处理完，而不能利用这段时间去处理其它连接上请求。很明显，这样一种服务器的响应时间和吞吐量都是很不理想的。

从服务器在分布式系统中扮演的角色来看，当然是响应时间越短越好、吞吐量越大越好，特别是吞吐量（对基于 TCP 的服务器而言，可以定义为单位时间内能处理的 TCP 连接数量）常常是度量服务器性能的最重要的指标，吞吐量小也常意味着响应时间长(因为小吞吐量服务

器常导致很长的等待服务的排队时间)，因此提高吞吐量常常是服务器程序设计中的重要目标。

提高服务器吞吐量，实质上是要提高服务器支持并发服务的程度，并发性越高，服务器能同时为之服务的请求数量也就越多。既然现代操作系统都支持多进程并发，特别是在SMP平台上甚至能实现真正的物理并发，因此对以上串行服务器方案可以自然地改进如下[7]：

```
bind(fd, …);
listen(fd,…);
while(1){
    fd1=accept(fd, ….);
    if(fork()==0){
        /*子进程*/
        read(fd1, buff, size);
        /*解析客户请求并响应*/
        ret = request_proc(buff,size,…);
        if(ret>=0){ /*ret<0 表示有错误，例如非法请求*/
            write(fd1,…);
        }
        close(fd1);
    }
  }
}
```

这一改进的结果是对每个请求都创建一个单独的进程为之服务，既然这些进程被操作系统并发调度，服务的并发程度也自然随之提高。图 9-5 是这种方案的运行机制。这种方案也同样适合于实现高吞吐量 UDP 服务器。

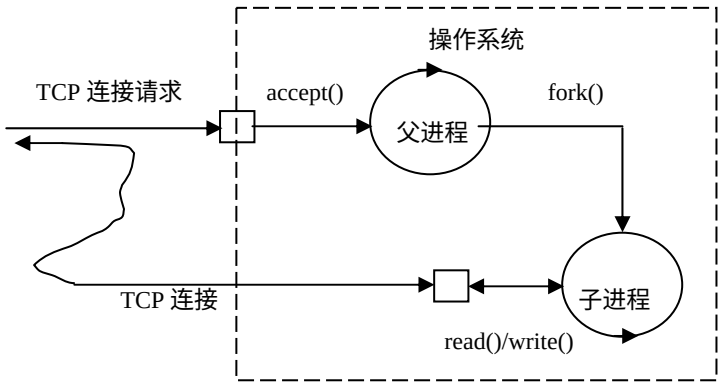


图 9-5 多进程服务器

[7] 下面的阐述都是采用Unix/Linux的编程技术，如fork函数、父/子进程、pthred_xxx 线程API等，但所有方案都不难针对Windows建立起对应的方案。

另一方面，以上方案也是有代价的：当被同时处理的服务数量很多时，会同时存在同样数量的子进程(且进程数量多于 CPU 数量)，这些进程之间频繁的上下文切换将随着进程数量的增长而成为系统负载的主要因素，因此这一方案提升吞吐量的能力会有一个上限。很明显，操作系统实现上下文切换的代价越小，这一吞吐量的上限也越高。

利用现代操作系统都支持的多线程结构，可以按照这一方案继续提高服务器吞吐量，具体做法是以线程代替子进程来实现并发服务，为每个建立的连接创建一个线程，处理该连接上到来的所有服务请求。

```
/*主线程*/
int   main() {
      int   fd = socket(AF_INET, SOCK_STREAM, 0);
      bind(fd, …);
      listen(fd,…)
      while(1) {
            conn = accept(fd,…);
            /*创建新线程*/
            pthread_create( thread_func, &conn );
      }
      ……
}
/*线程函数*/
int   thread_func( int* conn_fd ) {
      ….. …..
      read(conn_fd, …);
      解析服务请求并做相应处理;
      write(conn_fd, …); /*回送处理结果*/
      close(conn_fd);
}
```

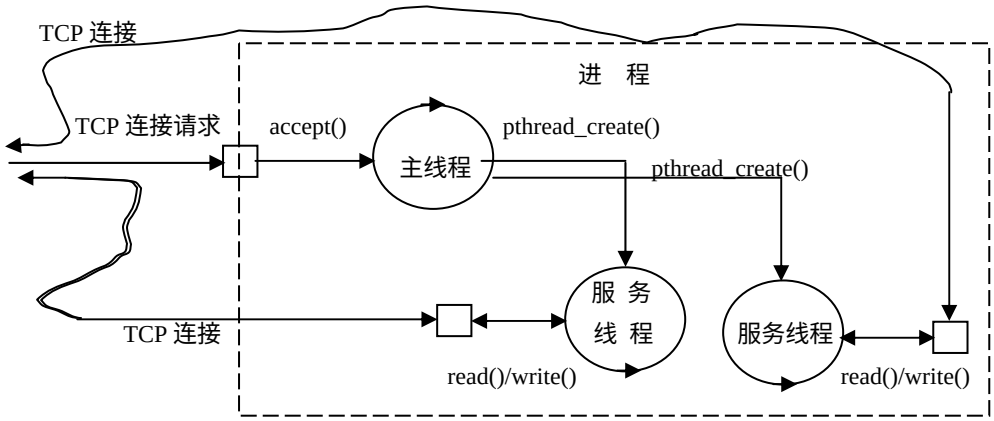


图 9-5 多进程服务器

当到达的服务请求密度很高时，以上服务器需要在短时间内创建大量线程，尽管创建线程的系统开销比进程要小得多，但短时间内创建很大数量的线程的总开销也会成为限制服务器响应速度的瓶颈，要想提高高密度服务请求下的响应速度，还可以进一步优化以上服务器的多线程调度结构，例如可以将完成服务处理的线程保留一段时间，这样在短期内再次有服务请求到达时，可以直接将该请求分配到空闲的线程，避免了创建新线程的开销，提高响应速度。这就是所谓线程池的思想，其程序结构如下：

```
/*主线程*/
int    main() {
       int    fd = socket(…SOCK_STREAM );
       bind(fd, …);
       listen(fd,…);
       ……
       创建一组线程;
       while(1) {
              conn = accept(fd,…);

              /*分配连接到一个线程.*/
              if(  如果存在一个空闲线程#thrd ) {
                    唤醒线程#thrd;    /*应用题 1*/
              } else {
                   /*没有空闲线程*/
                   pthread_create( thread_func, &conn );
                   唤醒该线程;
              }
        }
 }

 /*线程函数*/
 int    thread_func( int* conn_fd ) {
       ……..
       while( 1 ) {
           睡眠在预定义的事件上; /*应用题 1*/
           /*被唤醒*/
           read( conn_fd, …);
           request_proc(…);
           write(conn_fd,…);
           …..
       }
 }
```

上面非常概要地阐述了几种提高服务器吞吐量的方法。在实践中，不仅性能，而且服务器的可靠性也非常重要，例如要特别避免使服务器可能长期阻塞的任何条件，并且要仔细设计一旦这类阻塞发生如何使之有效解除。诸如此类的问题，都是网络服务器软件设计中极具挑战性的问题。

9.6 多协议服务器程序设计

如果要设计一个网络服务器使之能同时提供多个服务，例如同时提供 Web 与 FTP 服务，应该如何做？请读者看下面这一设计有什么问题(暂不考虑性能问题)：

```
/*创建两个 socket 对象*/
int fd_web=socket(AF_INET, SOCK_STREAM, 0);
int fd_ftp=socket(AF_INET, SOCK_STREAM, 0);
int fd1, fd2;

bind(fd_web, …); /*为一个 socket 对象指派 IP 地址和 Web 服务端口号*/
listen(fd_ftp,…); /*为另一个 socket 对象指派 IP 地址和 FTP 服务端口号*/
while(1){
      fd1=accept(fd_web, ….);
      read(fd1, buff, size);
        /*解析客户 HTTP 请求并响应*/
      request_proc_http(buff,size,…);
      write(fd1,…);
      close(fd1);

      fd2=accept(fd_http, ….);
      read(fd2, buff, size);
        /*解析客户 FTP 请求并响应*/
      request_proc_ftp(buff,size,…);
      write(fd2,…);
      close(fd2);
 }
```

这一服务器以此在两个 socket 对象上等待 TCP 连接，然后顺序服务。这里的问题是，如果某段时间内有 FTP 请求到达，但没有 HTTP 服务请求，由于该服务器总是先等待在 Web-socket 对象上，以至于永远无法响应 FTP 请求，直到有某个 HTTP 请求到达并服务完成后，服务器才会执行到 FTP-socket 对象上的 accept 函数。这样一种设计显然是不可行的。

多协议服务器意味着要同时检测多个socket对象上的可用状态，包括一组socket对象中哪些已经建立TCP连接、哪些当前可读、哪些当前可写等。通过上一节的学习，读者会想到用多线程方案实现，每个线程监视并处理一个socket对象上的服务。这固然是一种方法，但对性能要求一般的服务器，如何用简单的单线程结构实现呢？这里向读者介绍一个很有用的函数：select[8]。

该函数的原型是 int select(int ignore, fd_set* refds, fd_set* wrfds, fd_set* exfds, const struct timeval* timeout)，参数 refds、wrfds 和 exfds 是三个文件描述符数组，分别监视可读、可写和发生异常的文件描述符(实际上是这些文件描述符所标识的对象)。如果 refds 中有任何一个描述符可读—对一个准备接受 TCP 连接的 socket 对象，其涵义是其上已建立一个 TCP 连接，对已经建立 TCP 连接的 socket 对象，其涵义是有数据到达；或 wrfds 中任何一个描述符可写；或 exfds 中任何一个描述符发生异常，select 返回这些描述符的数量并设置相应数组项的状态，以便程序检查究竟是哪些描述符的状态发生了变化。参数 timeout 用来指定超时，如果从调用 select 开始经过时间 timeout 后没有任何描述符的状态发生变化，则 select 返回 0。timeout 为 0 表示 select 一直阻塞直到有某个任何描述符的状态发生变化。

利用 select，我们可以实现一个单线程的多协议服务器如下：

```
 /*创建多个 socket 对象*/
int fd_web=socket(AF_INET, SOCK_STREAM, 0);
int fd_ftp=socket(AF_INET, SOCK_STREAM, 0);
…….
int fd1, fd2,…., fdn;

bind(fd_web, …); /*为一个 socket 对象指派 IP 地址和 Web 服务端口号*/
listen(fd_ftp,…); /*为一个 socket 对象指派 IP 地址和 FTP 服务端口号*/
……
while(1){
    FD_SET(fd_web, &refds); /*FD_SET 和下面的 FD_ISSET 是宏*/
    FD_SET(fd_ftp, &refds);
    ……
    select(FD_SETSIZE, &refds, (fd_set *)0, (fd_set *)0, (struct timeval *)0);
    if(FD_ISSET(fd_web, &refds)){
        fd1=accept(fd_web, ….);
        read(fd1, buff, size);
        /*解析客户 HTTP 请求并响应*/
```

[8] select 源自 Unix System V，但在 Linux 和 Windows 上也可用。Win32 中的对应 API 是 WaitForMultipleObjectsEx，另外还有一个源自BSD Unix的功能相似但用法不同的API poll。

```
            request_proc_http(buff,size,…);
            write(fd1,…);
            close(fd1);
        }
        if(FD_ISSET(fd_ftp, &refds)) {
            fd2=accept(fd_http, ….);
            read(fd2, buff, size);
            /*解析客户 FTP 请求并响应*/
            request_proc_ftp(buff,size,…);
            write(fd2,…);
            close(fd2);
        }
        ……..
    }
```

事实上，select 并非 socket API 特有的函数，它可以应用与一切文件对象，如管道、共享内存等。

9.7 远程过程调用

socket-模型并非实现网络程序的唯一方式，与此相对还有另一种很不相同的网络编程方式：远程过程调用(RPC: Remote Procedure Call)。基于 RPC 实现的程序有这样的特点，一个客户程序调用一个远程服务和调用本程序内的一个普通函数方式完全相同。事实上，客户程序根本不必区分本地服务和远程服务的概念，对它而言只有一种服务，这就是已知的某个函数。这也正是 RPC 所追求的目标。

RPC广泛应用于现代分布式计算，实际上RPC是当代任何一种分布式计算标准的最基础的机制[9]，无论是OMG CORBA、JAVA/J2EE[10]，还是MS COM/DCOM，组件之间相互作用的基本过程都是建立在RPC之上的。

这里并不打算讲解基于 RPC 的编程—这是分布式计算教程的内容，而只是概要阐述 RPC 是如何实现的，以此使读者看到一种实现网络程序的颇为有益的做法。我们以图 9-6 为例来说明 RPC 的工作机理。

客户程序创建一个 TClassA 类对象，这是一个远程对象—例如一个 CORBA 对象或一个 DCOM 组件—但究竟是本地对象还是远程对象，客户进程既不知道也不关心，它所知道的是

[9] 每种分布式计算标准所依赖的具体的RPC不尽相同, 但思想和主要方面是完全一致的,事实上它们都源于 Sun RPC.

[10] J2EE的RPC称为RMI: Remote Method Invokation。

该对象提供哪些服务，即 TClassA 类对象上有哪些方法，并且如调用一个本地对象那样调用一个方法。

然而，RPC 运行环境(RPC Runtime)—一组实现 RPC 运行的共享库—明确知道 TClassA 类对象是一个远程对象，更精确地说，RPC 运行环境明确知道实现 TClassA 类对象的服务器进程在哪个主机上运行、TClassA 类对象上有哪些方法、每个方法带哪些类型的参数、以及每个方法返回值的类型。特别是，对象指针 a 所指的并非客户进程空间内的一个 TClassA 类对象，而是由 RPC 运行环境所实现的一个 TClassA 类对象代理，这是一段可与客户进程共享的程序代码，它的功能是将客户进程用来调用 TClassA 类对象上方法的每个参数按统一的格式编码，以使其能够通过网络传输到服务器进程。

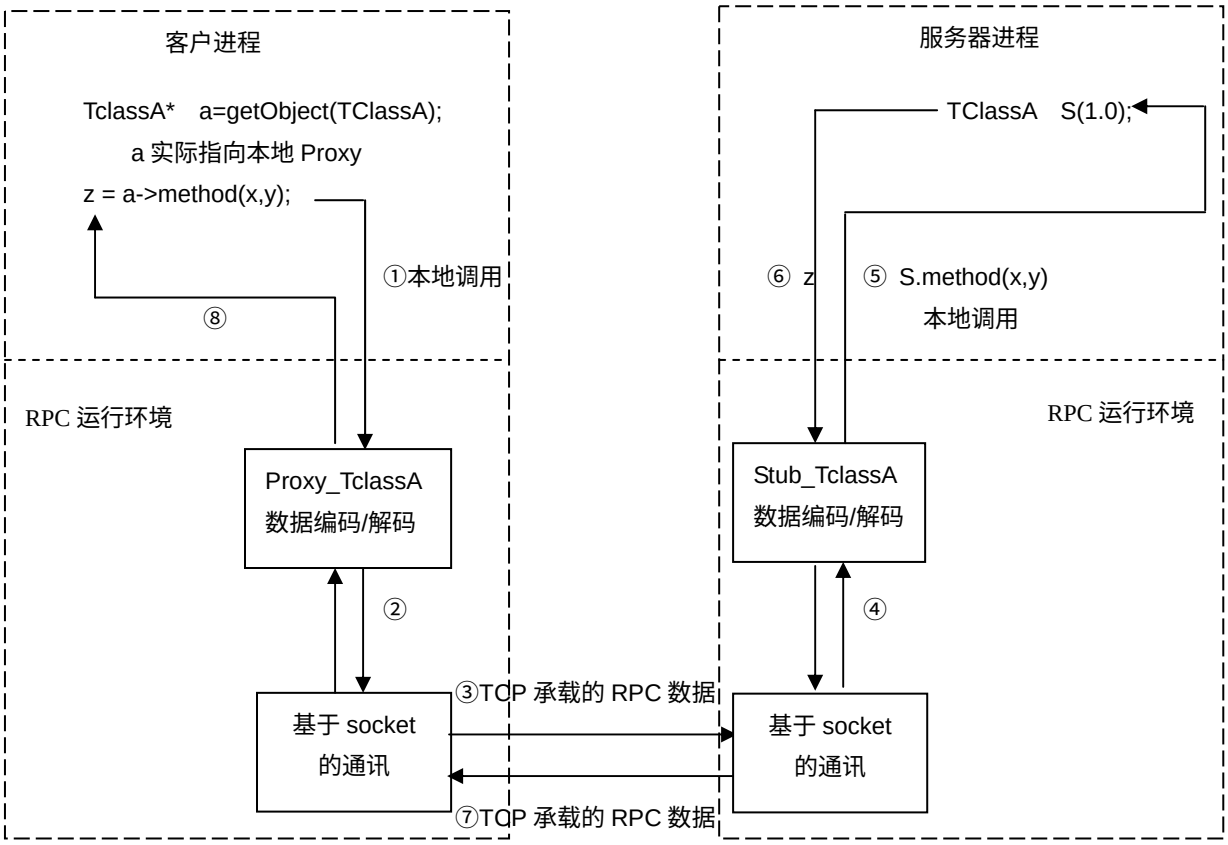


图 9-6 RPC 的一般性过程

由于客户和服务器可以运行与完全不同的机器和操作系统，并且可以用完全不同的语言实现，因此客户与服务器之间的数据交换需要一种统一的中间格式以正确表达彼此的数据类型，这一中间格式就是 XDR(External Data Representation)，这是一个典型的表示层协议，是 RPC 不可分割的组成部分。

图 9-6 中实现参数编码功能的 TClassA 类对象代理就是 Proxy_TClassA，实际上客户进程中的语句 a->method(x,y)所直接引发的不过是一个本地调用，它调用 Proxy_TClassA 完成对参数 x 和 y 的正确编码。然后，Proxy_TClassA 通过 socket 对象实现与远程服务器的通讯，传递参数，接收服务器返回的值并正确解码，最后将这些值返回客户进程。直到这时，a->method(x,y)才返回。

基于 socket 对象的通讯机制与前面几节描述的完全相同，这里要明确的是，基于 socket 对象的通讯实际上发生在 RPC 运行环境之间，而非直接发生于客户/服务器进程之间。RPC 作为一个应用层协议，既可以由 UDP 也可以由 TCP 承载。

当 RPC 调用请求到达服务器主机时，所发生的过程与客户机上的过程基本相反：所有数据由服务器端的 RPC 运行环境解码，然后唤醒实现 TClassA 类对象的服务器进程、调用该对象上的方法 method(x,y)，并将其返回的数据重新以 XDR 格式编码后传输回客户端的 RPC 运行环境。

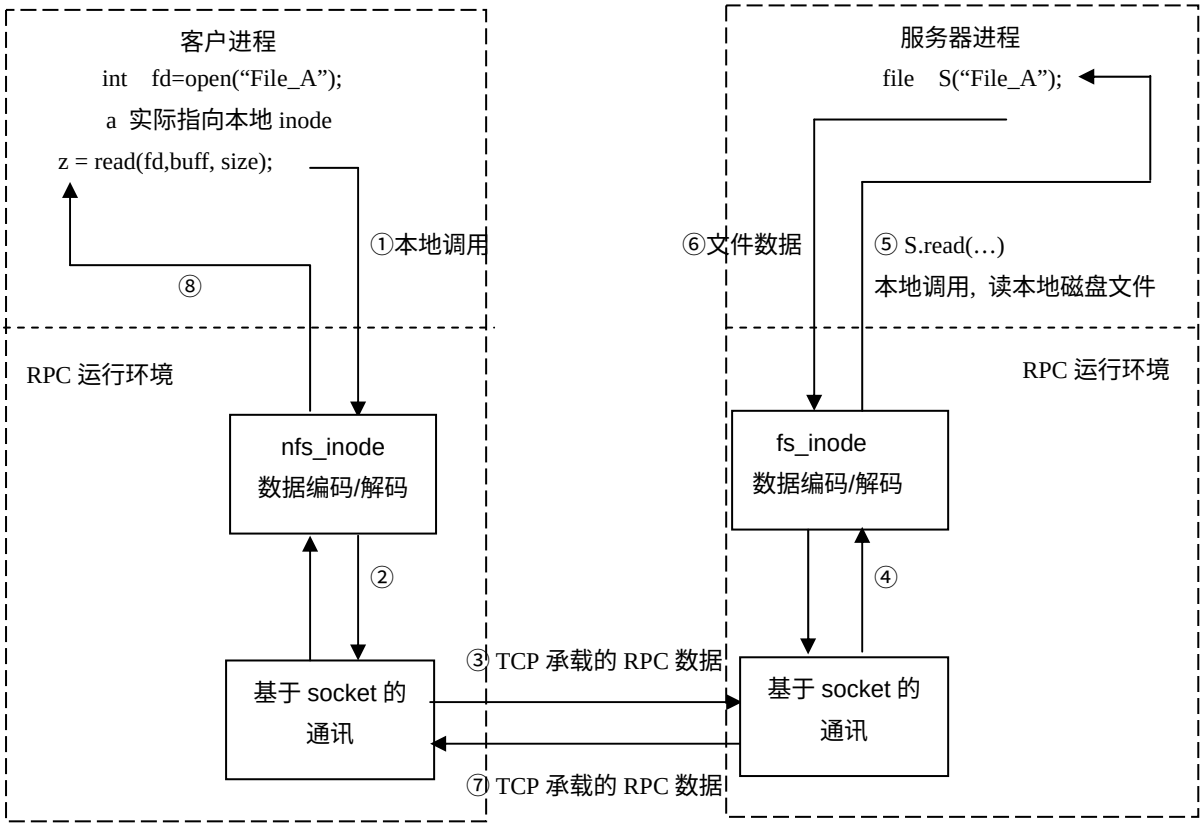


图 9-7 RPC 的例子: NFS

从 RPC 的实现可以看到，这里的关键是巧妙地利用了对象代理，从而将看似神秘的远程过程调用归结为应用程序/RPC 运行环境之间的本地调用和 RPC 运行环境之间的网络通讯。

在基于 RPC 实现的各种软件中，网络文件系统 NFS 大概是应用最广泛的，图 9-7 是 NFS 文件访问过程，读者可以自己体会出 read 函数是如何被巧妙地赋予远程访问语义的。

9.8 小 结

本章非常概要地阐述网络编程技术，主要阐述 socket API 所依据的抽象模型和这些函数的典型用法，并且解释了 socket API 与内核 TCP 和 UDP 协议程序之间的关系。本章还概要阐述了网络服务器程序的设计。最后，讨论了另一种风格的网络编程技术 RPC 并概要解释了 RPC 的实现。

9.9 进一步学习的指南

写程序犹如写作文学作品，在写作方面中国文人的古训之一是“功在诗外”。同样，要写出好的程序，除了需要熟练、准确地掌握某种编程语言和 API 之外（毕竟一个文法和修辞水平上的平庸之辈不可能写出能打动读者的作品），真正的功夫在于对要所解决的问题有清晰的认识、对解决问题的方法有准确理解，以及对程序细节孜孜不倦、精益求精的洞察。总之，优良的程序是高度技巧和清晰思想的完美结合，正如同优秀的传世之作永远是那些修辞幽雅同时思想深刻的作品一样。

从以上观点看，本章是非常概要性的，目的仅仅在于给读者勾画一幅网络编程的概念性图象，包括网络编程需要考虑的最主要因素和技术。然而，网络编程是一个博大精深、应用甚广的软件技术，需要较长期的认真训练，充实大量细节性的理解。读者不仅需要熟练掌握 TCP/IP 协议和 socket API，事实上需要对网络程序运行于其上的整个操作系统—无论它是 Unix、Linux 还是 Windows—的内核机理有一个深入的认识。可以毫不夸张地说，网络编程实际上要用到一个操作系统的几乎所有 API，特别是那些与并发、同步和异步事件有关的复杂而微妙的 API，而决不仅仅只限于 socket API。最后，也是最重要的，你需要对你的问题中多个并发程序之间的相互作用关系有一个清晰的认识，而这只能得益于孜孜不倦的思想上的刻苦训练。这也就是“功在诗外”的真正涵义。

关于网络编程，以下几本著作值得认真钻研：

Stevens R. Unix Networking Programming, 2th ed, Vol. 1, socket-APIs and XTI, Prentice Hall, 1997

Stevens R. Unix Networking Programming, 2th ed, Vol. 2, IPC, Prentice Hall, 1999. 该卷在最后两章详细讨论了远程过程调用编程。

(美) Stevens R 著. UNIX 高级环境编程，尤晋元 译，北京：机械工业出版社，1999

Comer D.E., Internetworking with TCP/IP, Vol.3: Client/Server Programming and Applications, 3th ed, Prentice Hall, 1995. 该卷第 19-24 章详细讨论了XDR、RPC、NFS及实现，还有两章详细给出了Telnet客户程序的实现。

另外，Stevens 在其著作“ TCP/IP 详解 ”第二卷中有关于 socket 实现的详细描述。所有这些著作在以前的章节都已给出过，作者在这里再次列举出来，无非是强调它们的价值，期望读者也能象作者一样从中获得从技巧到思想的升华。

最后要指出一点，本章和以上著作都是关于网络应用程序编程的阐述。关于如何实现内核网络程序，需要另一类技术，这方面尚没有好的著作，开发者只能参照操作系统的设备驱动程序规范，如 Windows 的 NDSI 和 Unix STREAM 等。操作系统厂商都提供这些规范和支持工具 SDK。

习 题

一、概念题

1 实际应用中的大多数情形都是系统中的服务器数量少、客户程序数量多，但不要据此以相对数量的多寡来区分客户/服务器角色。划分客户/服务器的依据只能是行为方式。请举出一个服务器数量多、客户程序数量少的系统的例子，并解释为什么在这一系统中你关于客户和服务器的划分是正确的。

2 操作系统中的文件子系统仅仅将文件作为字节流而不做任何结构性假设的好处是什么？类似地，socket 仅仅将被传输的信息作为字节流而不做任何结构性假设的好处是什么？

3 尽可能多地列举将 socket 对象实现为操作系统的文件对象的好处。

4 socket 对象被实现为操作系统的文件对象，从面向对象观点，socket 对象类是操作系统文件对象类的继承类，特别是 socket 对象上继承了普通文件对象的 read/write 方法。但从方法实现的内部来看，socket 对象上的 read/write 方法和普通文件对象上的 read/write 方法当然有本质不同，前者控制内核中的 TCP/IP 软件，后者控制的是本地磁盘设备。在面向对象方法中，这是一种什么现象？

5 就你所知，操作系统中还有那些实体被实现为文件对象？这样做的优点在哪里？

6 图 9-4(d)中的客户端(左面)socket 对象直到第二个 FIN/ACK 消息交换完成后才被释放，而不是在调用 close 时就被立刻释放，为什么？

二、应用题

1 对 9.5 节中的线程池方案，试利用你所熟悉的任何一种商用操作系统功能，设计一种线程的睡眠-唤醒机制。

参考答案

一、概念题

1 网络管理系统是一个典型的这类系统，其中管理员控制程序(即使很大的网络可能也只有一个该程序运行)是客户，而大量的所谓网管代理程序(NM-agent)都是服务器。实际上每个支持网管的设备和主机都必须运行网管代理程序，它们的功能包括收集本地设备的运行数据与状态、检测本地设备故障、在控制程序请求收集时向它发送这些信息，以及接受来自控制程序的指令进行本地设备控制(远程配置)。网管代理与控制程序之间的通讯协议有 SNMP、SNMPv2、RMON 和 RMON2。

4 多态性(polymorphism)，即将“做什么”和“怎么做”分离。

5 这样的例子非常多，这里给读者提供一个不很直观的例子：现代虚拟内存系统中的共享地址空间。实际上读者可以想一想 Windows 或 Unix/Linux 中有哪些对象是以文件句柄或文件描述符标识的？

二、应用题

1 对 Win32，可以利用全局事件对象及 WaitForSingleObject()与 SetEvent()这样的 API，对 Unix/Linux 可以利用 POSIX 互斥对象及 pthread_mutex()与 pthread_signal()这样的 API。读者还可以利用其它许多机制。

2 设计一个 FTP 客户程序及 FTP 服务器程序。注意 FTP 协议建立两个连接：一个由客户端发起建立的 FTP 控制连接和一个由服务器发起建立的数据连接。

3 (1)设计一个浏览器程序，要求支持 HTTP 1.1 的所有命令并能执行 Applet；

(2)一个单线程 Web 服务器的结构如下，试根据 HTTP 1.1 协议，详细实现函数 request_proc，并且将该服务器改造为多线程结构，使每个 TCP 连接对应有一个线程：

```
/*Web Server*/
main(){
  int    fd=socket(AF_INET, SOCK_STREAM, 0);
   int fd1;
   ……
   bind(fd, …);
   listen(fd,…);
   while(1){
        fd1=accept(fd, ….);
        read(fd1, buff, size);
         /*解析客户请求并响应*/
        ret = request_proc(buff,size,output);
        if(ret>=0){ /*ret<0 表示有错误，例如非法请求*/
            write(fd1,…);
        }
        close(fd1);
    }
}

int   request_proc(char* buff, int size, char* output)
{
    /*buff[]中是 HTTP 消息*/
    验证 buff[]中的 HTTP 首部信息;
    http_cmd=取 buff[]中的 HTTP 命令;
    switch(http_cmd){
      case   HTTP_CMD_GET:
          path=取网页的路径名;
          读网页文件，如果是 ASP 文件则解析之并动态生成 HTML 文档;
          break;
       ……
    }
}
```

4 Microsoft ASP 有所谓 Session-对象，可以支持跨越多的 TCP 连接的事务。你能设计一种方法使上题中的服务器支持 ASP Session-对象吗？

5 设计一个 Telnet 服务器程序。Telnet 客户程序的详细设计可以参考 Comer 的教科书第 3 卷。

6 Unix 中的 rshell(实际上 Windows 也有类似程序)是用来执行远程命令的客户/服务器程序，

使用户在客户机器上键入的命令可以在远程机器上执行并返回结果，就象在本地执行程序一样。rshell 基于 TCP 完成通讯，试设计一个 rshell 服务器程序 rshd。注意在你的方案中充分利用 shell 程序。