**实例 1：Socket 进程通信**

Socket 应用程序接口即是基于该范型的。Socket 是一种逻辑结构，利用这一构造，两个进程可以按如下范型来交换数据：发送者将消息写入或插入到 Socket 中；在另一端，接收者从 Socket 中读取或提取消息。

Socket 进程通信是由传输层提供的一种使用最广泛的 IPC 机制,它属于消息传递通信方式，所有的 Socket 都由传输层服务来管理。Socket 有一个局部地址和一个全局地址。局部地址是端口号，由传输层服务提供和管理的，而全局地址指的是网络主机地址，在使用时这两个地址必须结合在一起。Socket 既支持可靠的面向连接的通信，又支持不可靠的面向非连接的通信。每个 Socket 只属于一个进程，两个 Socket 之间的通信其实就是两个进程之间的通信。本书只介绍UNIX环境下的Socket通信机制。Windows环境下的socket称为Winsock。

UNIX socket 原语可通过 C 程序库使用。UNIX 支持两种类型的 socket，一种是 UNIX socket，另一种是 Internet socket。在 Internet socket 中使用 socket 原语时需要一个变量，它是指向 socket 地址的结构指针，这个结构为 struct sockaddr，在文件<sys/socket.h>中定义：

```
struct sockaddr{
        u_short sa_family;    /* AF_UNIX 说明是 UNIX socket */
                              /* AF_INEI 说明是 Internet socket */
        char sa_data[14];     /* 随协议不同应作不同解释的 14 字节 */
};
```

其中，char sa_data[14]随协议不同应作不同解释。对于 Internet socket，它是 struct sockaddr_in 结构，该结构在文件<netinet/in.h>中进行了定义：

```
struct in_addr{
        u_long s_addr;  /* 32_bit netid/hosted */
                        /* network byte ordered */
};
struct sockaddr_in{
        short sin_family;       /*AF_INET*/
        u_short sin_port;       /*16_bit port number*/
                                /*network byte ordered*/
        struct in_addr sin_addr;    /*32_bit netid/hostid*/
                                    /*network byte ordered*/
        char sin_zero[8];           /*unused*/
};
```

在 struct sockaddr_in 中，sin_port 是局部地址，而 sin_addr 是全局地址。

Berkeley 4.2 UNIX 支持 TCP/IP 协议，它们通过一组 socket 原语来访问网络，获得极为广泛的应用,成为事实上的标准。Sun 工作站的操作系统 SunOS 也使用这种方法访问网络。Berkeley 的原语由一组允许用户访问传输层服务的系统功能调用来实现。

主要的系统调用列举下：

（1）s=int socket(domain，type，protocol)。建立一个 socket s。当 domain 的值是 AF_UNIX 时，s 是 UNIX socket；当 domain 的值是 AF_INET 时，s 是 Internet socket。当 type 的值是 SOCK_STREAM 时，s 是面向连接的 socket；当 type 的值是 SOCK_DGRAM 时，s 是面向非连接的 socket。在 Internet 中，由于使用面向连接的 socket，可选择的协议只有 TCP，而当使用面向非连接的 socket，可选择的协议只有 UDP，由于无可选择，所以可用 0 作为 protocol 的缺省值。

（2）int bind(s，localaddr，addrlen)。使前面创建的 socket s 和一个地址结合，结合成功后,该地址就是 socket s 的地址。localaddr 是指向这个地址的指针,类型是 struct sockaddr_in *。addrlen 是这个地址结构的字节数。

（3）int connect(s，server，serverlen)。在面向连接的通信中，发起与远端 socket 的一个连接，远端的 socket 由其地址指针 server 指出，serverlen 是这个地址结构的字节数。

（4）listen(s，n)。在面向连接的通信中，建立一个队列以存储想连接到 socket s 的外来的连接请求。n 是该队列的长度，它允许一个服务能处理多个通信请求。

（5）new_sock=int accept(s，from，fromlen)。在面向连接的通信中，将一个连向 socket s 的连接请求从队列中移出或等待一个连向 socket s 的连接请求。如果成功，产生一个新的 socket new_sock,以后用这个新的 socket new_sock 同远程发出连接请求的 socket 通信。远程 socket 的地址由 from 指出，from 是指向这个地址的指针，类型是 struct sockaddr_in *。fromlen 是这个地址结构的字节数。

（6）send(s，buf，buflen，flags)。在给定的 socket s 上发送一个报文（有连接），报文由 buf 指定，报文长度由 buflen 指定。

（7）recv(s，buf，buflen，flags)。在给定的 socket s 上接收一个报文(有连接)。所收到的报文放在缓冲区 buf 里，缓冲区 buf 的大小为 buflen。

（8）close(s)。拆除一个 socket 上的连接。

（9）shutdown(s，how)。终止一个 socket 上的连接。

（10）sendto(s，buf，buflen，flags，to，tolen)。在无连接的 socket 上发送一个报文，该报文发送给远程 socket，远程 socket 的地址由 to 指出，to 是指向这个地址的指针，类型是 struct sockaddr_in *。tolen 是这个地址结构的字节数。

（11）recvfrom(s，buf，buflen，flags，from，fromlen)。在无连接的 socket 上接收一个报文，该报文是从远程 socket 发送来的，远程 socket 的地址由 from 指出，from 是

指向这个地址的指针，类型是 struct sockaddr_in *。fromlen 是这个地址结构的字节数。

（12）select(nfds，readfds，writefds，exceptfds，timeout)。检查一组 socket，看它们是否可读或可写。select 调用对建立了几条连接的进程很有用处。在很多情况下，这样的进程可以在有报文到达时，在它处理的任何 socket 上执行 recv 调用，但它不知道哪个 socket 上已有报文，哪个上没有。如果它选取一个 socket，而报文在其他一些 socket 上时，它可能因等待报文而封锁很长时间。此时它可以用 select 调用，此调用终止时，调用者被告知哪些 socket 上有报文，哪些没有。nfds 指定文件说明符范围，timeout 指定 select 等待的时间，其余 3 个参数作为位掩码，分别使用读、写、异常处理文件设置。

send()和 recv()调用本质上和 read()及 write()一样。flags 是一个很重要的参数，可以使用以下的值：

MSG－OODB　发送/接收紧急数据；

MSG－PEEK　　检查数据但不读；

MSG－DONTROUTE　发送不带路由包的数据。

一旦 socket 不再使用，可以使用 close()调用释放。此时，如果有连接的 socket 还有数据，则系统将力争继续把该数据传送出去。但是如果相当长时间(如 4 分钟)后该数据仍未送出去，则将作废。如果不用此 socket 收发任何数据，可在用 close()调用前使用 shutdown()调用，其中的参数 how 可取 0，1，2。当用户不再想读数据时为 0，不再发送数据时为 1，不再发送或接收数据时为 2。对 socket 使用 shutdown()操作可使在排队中的数据立即作废。

socket 用于数据报服务，发送与接收数据时使用 sendto()和 reevfrom()原语，to 和 tolen 指报文的接收者和长度，而 from 和 fromlen 指从何处接收及长度。

图 1 给出了在典型的面向连接的传输方案里的时间关系，即服务首先启动，过一些时候客户启动，并连接到服务。

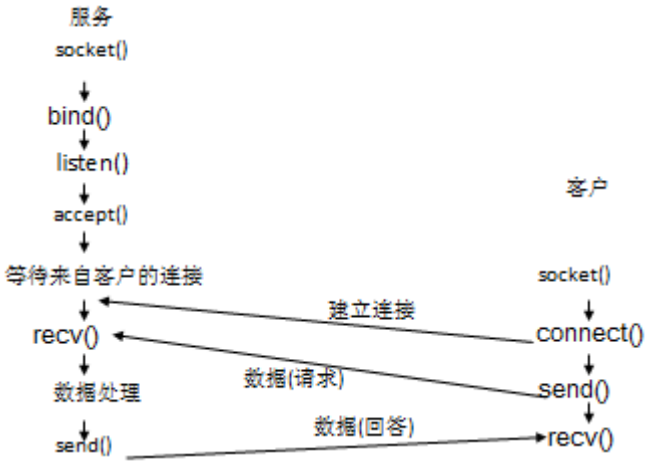


**图 1　面向连接协议的 socket 通信模型**

对于采用无连接协议的客户－服务系统，所用的系统调用是不同的。图 2 给出了这些系统调用。客户不是和服务建立连接，而是使用 sendto 系统调用向服务发送一个数据报，作为参数请求目的地址(服务)。同样，服务也不必接受来自客户的连接，而是发出系统调用

recvfrom，等待来自客户的数据到达。这一系统调用返回客户进程的网络地址和数据报，于是服务可以向对方进程发出响应。

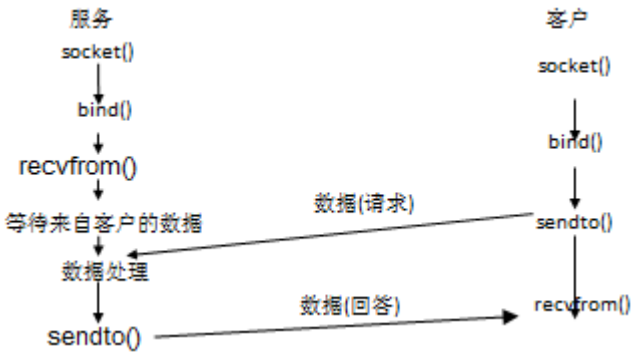


**图 2　面向非连接协议的 socket 通信模型**

**实例 2：MPI 进程通信**

随着高性能多计算机系统的出现，开发者寻求着能够让他们很容易地开发高效应用程序的报文传递机制。这意味着报文传递原语应该处于一个方便的抽象层上以利于应用程序的开发，并且开发代价较低。Sockets 在这两个方面都显得不足。首先，对用户来说，它们处于一个不适当的抽象层上，并且只有 send 和 recv 两个原语。第二，Sockets 是使用通用的协议栈来设计和实现网络通信的，例如 TCP/IP，人们认为这些协议栈不适合作为在高速互连网络上进行开发的专用协议。在像 COWs 和 MPPs 这样的高速互连网络上，协议的用户接口应该能处理更多的高级特性，例如不同类型的缓冲和同步机制。

由于以上原因，在大多数的高速互连网络上和高性能多计算机系统上安装了专用的通信函数库，这些函数库提供了大量高层的通常也是高效的通信原语。但是这些函数库，彼此并不兼容，所以程序设计者面临着程序移植的问题。

程序设计者对函数库的硬件独立性的需求最终导致了报文传递标准的定义，这个标准被称为报文传递接口（Message-Passing Interface），即 MPI。

MPI 中通信总是发生在一个已知的进程组中，每个进程组都被赋予了一个组标识符（groupID），进程还有一个进程标识符（processID）。（groupID，processID）唯一地指出了报文的源地址或目的地址，不使用传输层地址。在一个计算中可能发生多个进程组重叠的现象，并且这多个进程组可以同时执行。

进程通信原语是 MPI 的核心，能够支持很多通信类型。MPI 中最主要的通信原语有：

（1）MPI_bsend 原语。该原语支持异步通信，发送者提交要发送的报文，一般来说这个报文首先被拷贝到 MPI 运行时系统（MPI runtime system）的本地缓冲区中。当报文已经拷贝到 MPI 运行时系统的本地缓冲区中后，发送者就可以继续向下执行。一旦接收者调用一个接收原语，本地的 MPI 运行时系统就将报文从本地缓冲区中取走并负责进行传送。

（2）MPI_send 原语。该原语有一个阻塞操作，其语义依赖于实现。MPI_send 原语可以使调用者阻塞直到特定的报文被拷贝到发送方的 MPI 运行时系统中，或者使调用者阻塞直到

接收方启动了一个接收操作。

（3）MPI_ssend 原语。该原语用于同步通信，该原语使得调用者被阻塞，直到报文已经被接受者接收。

（4）MPI_sendrecv 原语。该原语用于实现最强的异步通信。当发送者调用 MPI_sendrecv 原语，向接收者发送一个报文之后，发送者一直阻塞直到接收者返回一个应答。该原语和通常的 RPC 类似。

（5）MPI_isend 原语。使用该原语，发送者将要发送的报文的指针传递给 MPI 运行时系统之后，发送者马上就可以继续往下执行，由 MPI 运行时系统负责通信。由于这时 MPI 运行时系统不保存要发送的报文，只知道该报文在什么地方，所以要防止一个报文在通信完成之前被覆盖，为此 MPI 要提供检查通信是否完成和对用户报文缓冲区进行阻塞的语义。MPI_isend 原语实际上是 MPI_send 原语的一种变体，它不需要将报文传递给 MPI 运行时系统的内部缓冲区。对于 MPI_send 原语而言，报文是已经被接收者实际接收还是仅仅被拷贝到本地 MPI 运行时系统的内部缓冲区，这个问题不需要说明。

（6）MPI_issend 原语。同 MPI_isend 原语一样，发送者仅仅将要发送的报文的指针传递给 MPI 运行时系统。当 MPI 运行时系统指出已经处理完这个报文时，发送者就可以解除阻塞，继续向下执行，并可以确认接收者已经收到报文。MPI_issend 原语实际上是 MPI_ssend 原语的一种变体，它不需要将报文传递给 MPI 运行时系统的内部缓冲区。

（7）MPI_recv 原语。该原语用于接收一个报文，调用这个原语的进程被阻塞，直到一个报文到达为止。

（8）MPI_irecv 原语。通过这个原语接收者向 MPI 运行时系统指示它准备好了接收一个报文，它是一个异步通信原语，接收者稍后可以检测是否确实有一个报文已经到达，接收者甚至可以阻塞直到有一个报文到达。

MPI 通信原语的语义并不总是直接的，不同的原语有时可以互换而不影响程序的正确性。为什么它提供了那么多的通信方式，是因为它为了给 MPI 系统的实现者提供优化系统性能的最大可能性。MPI 是为高性能并行应用程序设计的，所以通信原语很多。