Department of Computer Science

Notes on Interprocess Communication in Unix

Jean Dollimore ,Oct.1990, last revised Feb. 1996

UNIVERSITY OF LONDON

§§These notes explain how you can write "distributed programs" in C or C++ running over Unix. In particular, we tell you how to arrange that a process in one computer can send information that is received by a process in another computer.

The facilities for interprocess communication (IPC) originate from 4.2 BSD Unix. They are implemented as a collection of system functions in the Unix kernel and are accessed through a system call interface.

There are two approaches to the use of IPC. In the first approach communication consists of simple one-off messages known as datagrams. In the second approach, the communicating processes establish a prior connection and then communicate by transmitting information via the connection.

The first section of these notes introduces the objects used for datagram communication in Unix IPC, some of which are also used in stream communication. **Sockets** are the end points of communication in processes, computers have **internet addresses** and **port numbers**. **Messages** are sent to **socket addresss**es. Section 2 introduces the system calls for using these objects to send datagrams. Section 3 explains how to make processes communicate via connections.

1 Objects in Unix IPC

1.1 Sockets

Interprocess communication consists of an exchange of some information by transmitting it in a message between a socket in one process and a socket in another process. A socket is created by a process that wants to communicate by using the *socket* system call. The socket call returns a descriptor by which the socket may be referenced in subsequent system calls. The socket lasts until it is *closed* or until every process with the descriptor exits. A pair of sockets may be used for communication in both or either direction between processes in the same or different computers.

The term **communication domain** is used to refer to a particular family of protocols with a particular address family. The Unix IPC facilities were designed to work with several communication domains. Therefore, when a socket is created, a communication domain must be specified. These notes refer to the **Internet domain.** An internetwork is an environment in which sets of computers and gateways are linked together by networks. The Internet protocols are designed for such an environment and are intended to cope with networks such as the Ethernet that do not report on the success of message delivery.

Another property that must be specified for new sockets is the **type of communication.** i.e. the desired properties of communication. One type is *stream* - which will require the prior establishment of a connection and then delivers data reliably (in order, without duplicates and without loss). Another type is *datagram* - which allows single messages to be sent without establishing a connection, but the delivery is potentially unreliable.

Each socket has a *protocol* associated with it. The protocol is selected (usually by the system) to suit the type. Examples of protocols are TCP (for streams) and UDP (for datagrams).

1.2 Internet addresses

Each computer in an internet is given an ***Internet address.*** This consists of 4 bytes containing the number of the network and the number of the computer within the network (see CDK3 Section 3.4.1 for more details). Any computer that is attached to more than one network has an address on each. An Internet address can be used to address a message to any computer in an Internet. If you want to see the Internet addresses of local computers, try “ypcat hosts", or use *nslookup*. (By the way, don't "hard-wire" these addresses into your programs).

1.3 Port Numbers

Within a computer ***each protocol*** has a large number of **port numbers** that may be used for sending and receiving messages. A port number is a 16 bit integer. Although a computer may have only a single address on the Internet, it will have many port numbers at that address. Port numbers are implemented entirely by (kernel) software. Each message is “addressed to a port number”. This is

equivalent to the sender specifying a destination port number which is transmitted in the message. On arrival, the port number in the message determines the port number to which the message is delivered. Ports are protocol specific - e.g. port number 345 in the UDP protocol is completely unrelated to port number 345 in the TCP protocol. Some protocols (particularly host to host management protocols) don't have port numbers at all. Ports numbered 0 to IPPORT_RESERVED-1 are reserved for use by services such as *login*, *who*, *uucp*, *ftp*, *telnet* and so forth. You can see the information about these services and their port numbers in the file "/etc/services".

1.4 Socket addresses

Each a socket has a descriptor which is private to the process that created the socket. Therefore it cannot be used directly by another process as a source from which to receive a message or a destination to which a message may be sent. Before a pair of processes can communicate, each one must bind its socket descriptor to a public **socket address**. This is generally described as "binding a socket to a socket address".

In the Internet domain, a socket address consists of a port number and Internet address pair. Socket addresses are public in the sense that they can be used by any process.

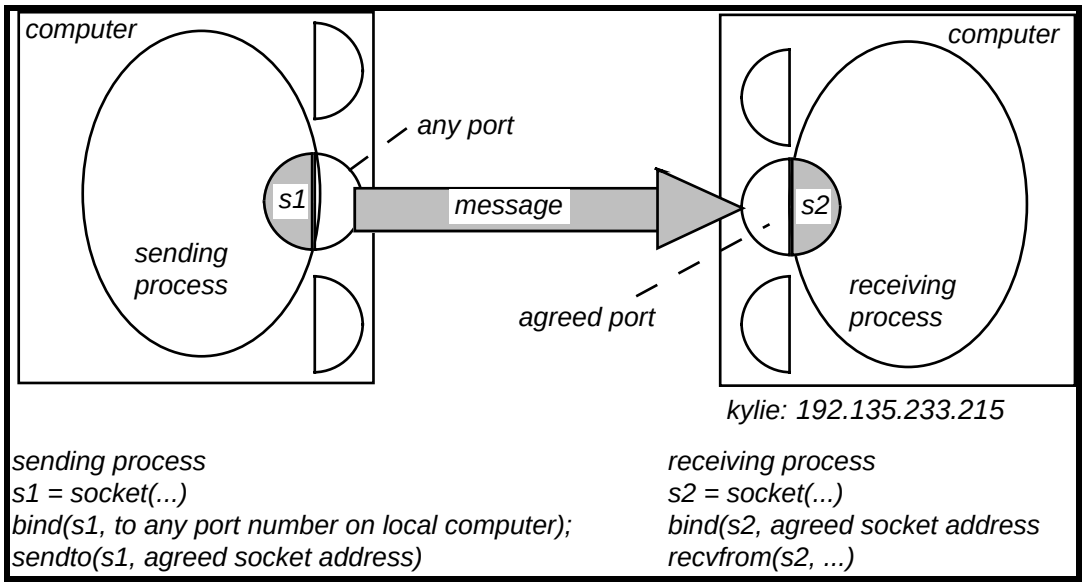

Figure 1. Sending a message from one process to another

See Figure 1 in which a process is oval and a computer is a rectangle. Sockets and ports are shown as matching semicircles - sockets are shaded and shown within a process. The *binding* is indicated by putting the socket and port together. After a remote process has bound its socket to a socket address, the socket may be addressed indirectly by another process referring to the appropriate socket address.

1.5 Messages

Every message is sent by a process on one computer through a socket with a particular protocol via a port number on that computer. It is received by a process on another computer through a socket using the same protocol via a port number on the latter computer. To summarise the steps:

a process on the destination computer opens a socket and binds it

a process on the source computer sends a message through a **socket** via a **port number** on its own computer

the message arrives at the destination **port number** on destination computer

waiting process on destination computer receives message through a **socket**

(both sockets must use the same protocol)

Sockets are private to processes and port numbers belong to computers. Processes can use the socket addresses of local or remote port numbers. Therefore the path of communication is:

socket (in sender) → local port number → destination socket address → socket (in receiver)

where sender and receiver are processes.

Messages are uninterpreted sequences of 8-bit bytes (in C terminology, unsigned char). It is the programmer's responsibility to make sure that the message is intelligible to the receiver, which may not be on the same type of computer as the sender.

Eight bit bytes may be transmitted in messages as they are. However, anything larger than a single byte (e.g. an integer) must be sent in a standard network ordering. The library functions *htons* and *htonl* (*h*ost *to n*etwork *s*hort or *l*ong) may be used to convert a short or long integer from host ordering to network ordering. The functions *ntohs* and *ntohl*(*n*etwork *to h*ost *s*hort *or l*ong) may be used to convert them back

You should also note that C structures may have slightly different sizes on different machines, with invisible padding added between elements of different types so that integers are aligned conveniently.

To summarise: from the point of view of a process, the end points of communication are a private socket descriptor and a public socket address. The socket descriptor must refer to a socket created by the process concerned. The socket address must have been bound to a socket descriptor at the destination process. In other words, sender and receiver respectively:

send a message through one of my *sockets* to *socket address*

receive a message through one of my *sockets*

2 Using Unix system calls to send datagrams

This section describes how to use the Unix system calls to send and receive datagrams. The following system calls are discussed:

socket: to create a socket and get a file descriptor for it

bind: to bind a socket address to the file descriptor of a socket

sendto: send a message through a bound socket to a socket address

recvfrom: to receive a message through a socket

select: to detect whether a message has yet arrived (really for I/O in general)

close: to destroy the socket when it is no longer needed (c.f. files)

In addition to these notes, you should read the on-line manual information provided for each of the above system calls. When you invoke these system calls in your programs, you should check if the result returned is -1 (indicating an error). The following *includes* are required in any C program using these calls :

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
```

For C++ programs, the prototypes of the system calls are required in an extern "C" definition. The most convenient way of providing these is to include the file (as addressed at QMW):

```
/import/GCC/lib/g++-include/sys/socket.h
```

If you need to use the library function inet_ntoa (as illustrated in Figure 3 of these notes) you will have to include its prototype.

```
extern "C" {
        char * inet_ntoa(struct in_addr);
}
```

To send a message from one process to another process, each process must first create its own socket. In Figure 1, the sender's socket descriptor is *s1* and the receiver's *s2* - these descriptors are private to the processes that created them.

Secondly, the two processes must agree on the address of the socket to be used. That amounts to agreeing to the internet address and port number of the recipient. Once this has been decided, the receiving process is started on the computer with the agreed internet address and binds its socket to the **agreed socket address**.

The sender binds its socket to a socket address referring to any available local port number. The recipient calls *recvfrom* in order to receive an incoming message through its socket. The sender calls

sendto and names *s1* and the agreed socket address. Figure 2 is a sample procedure that sends two messages to a specified port number on a given machine. For example, the procedure might be called:

sender("hello", "how are you", "kylie", 4567);

to send the two messages "hello" and "how are you" to port number 4567 on machine named "kylie".

2.1 Creating a socket

First a socket must be created:

int socket (int domain, int type, int protocol);

For example, in Figure 2, we create a new socket for datagram communication in the Internet domain. A socket is an object that contains information as to the type of communication required, e.g datagram or stream, the protocol in use (e.g. UDP or TCP), options (e.g. broadcast) and references to buffers for the incoming and outgoing messages. It has operations for creating it, binding it to a socket address, sending and receiving messages through it (or others associated with connections).

```
/*      send 2 messages to machine at port number */
  void sender(char *message1, char *message2, char *machine, int port)
  {       int s, n;
          struct sockaddr_in mySocketAddress, yourSocketAddress;
          if(( s = socket(AF_INET, SOCK_DGRAM, 0))<0) {
                perror("socket  failed");
                return;
          }
          setBroadcast(s);        /*see Section 2.7 */
          makeLocalSA(&mySocketAddress);
          if( bind(s, &mySocketAddress, sizeof(struct sockaddr_in))!= 0){
                perror("Bind  failed\n");
                close (s);
                return;
          }
          makeDestSA(&yourSocketAddress,machine, port);
          if( (n = sendto(s, message1, strlen(message1), 0, &yourSocketAddress,
                sizeof(struct sockaddr_in))) < 0) perror("Send 2 failed\n");
          if( (n = sendto(s, message2, strlen(message2), 0, &yourSocketAddress,
                sizeof(struct sockaddr_in))) < 0) perror("Send 2 failed\n");
          close(s);
  }
  void makeLocalSA(struct sockaddr_in *sa)
  {
          sa->sin_family = AF_INET;
          sa->sin_port = htons(0);
          sa-> sin_addr.s_addr = htonl(INADDR_ANY);
  }
  void makeDestSA(struct sockaddr_in * sa, char *hostname, int port)
  {       struct hostent *host;
          sa->sin_family = AF_INET;
          if((host = gethostbyname(hostname))== NULL){
                printf("Unknown host name\n");
                exit(-1);
          }
          sa-> sin_addr = *(struct in_addr *) (host->h_addr);
          sa->sin_port = htons(port);
}
```

Figure 2 - sample sending procedure

The first argument of the ***socket*** system call gives the communication domain in which the socket should be created. The second argument specifies whether we want datagram or stream. The last argument may be used to specify a particular protocol, but setting it to zero causes the system to select a suitable protocol.

2.2 Binding a socket to a socket address

After a socket has been created it must be bound to a socket address:

int bind (int s, struct sock_addr * socketName, int addrlength);

The first argument is a socket descriptor returned by a socket system call. The second argument is a *sock_addr* structure specifying the name of the socket. The third argument is the size of the structure in the second argument.

A socket address is represented by a structure whose first field specifies the communication domain. The following fields give the information expected for that domain. In Figure 2, the communication domain is the Internet domain and the other fields require a port number and an internet address. The constant AF_INET given in the first argument is actually redundant (because it was supplied in the socket call), but you have to fill it in all the same.

The ***sockaddr_in*** structure is defined in the file */usr/include/netinet/in.h*:

```
/* Socket name, internet style */          /*  internet address */
struct sockaddr_in {                       struct in_addr {
    short  sin_family;                         union {
    u_short sin_port;                      -------- /* we don't need to know about this*/
    struct in_addr sin_addr;               --------
    char  sin_zero[8];                         u_long S_addr;
};                                         } S_un;
```

The fields of a *sockaddr_in* structure are used to fill in the protocol family, the port number and the internet address. The *struct in_addr sin_addr* field is another structure in which the field *s_addr* should be filled in network order. The port number must also be in network order. A *(struct sockaddr_in *)* structure which is the Internet form of a socket address may be used as an argument where the corresponding parameter requires a *(struct sockaddr *)*. This applies to the system calls *bind*, *sendto* and *recvfrom*.

When an application plans to use a socket to receive a message, it must decide on a port number which will also be used in the *sendto* call in the process that transmits the message. The destination socket address in the sender procedure in Figure 2 uses the agreed port, given as argument. The procedure *makeDestSocketAddress* is used to fill in a destination internet domain socket address.

The library function *gethostbyname* takes the name of a computer as argument and returns a pointer to a structure (*struct hostent*) whose fields containing its Internet address. The information it supplies is already in network order. You should try to design programs to call this function once only for each computer involved in the communication.

Another library function *inet_ntoa* may be used to make an ascii string from an internet addresses. Its argument is a *struct in_addr*. See Figure 3 for a procedure for printing socket addresses.

```
void printSA(struct sockaddr_in sa)
{
        printf("sa = %d, %s, %d\n", sa.sin_family,
              inet_ntoa(sa.sin_addr), ntohs(sa.sin_port));
}
```

Figure 3 - printing a socket address

When an application plans to send a message it gives zero as the port number and the system will select an unused port number. The first binding in Figure 2 binds the socket with descriptor *s* to the socket address *mySocketAddress*.

The local internet address is specified as a pattern rather than a fixed address - the value INADDR_ANY in the procedure *makeLocalSocketAddress* in Figure 2 means: “use any of my IP addresses to accept messages”.

2.3 To Send a message

Sendto is used to transmit a message to another socket.

int sendto(int s, char * msg, int len, int flags, struct sockaddr *to, int tolen)

The first argument is a socket descriptor returned by a *socket* system call. The next two arguments supply the message and the number of bytes in the message. The *flags* argument is normally zero. The socket address of the receiver is given in *to* with *tolen* specifying its size.

The *sendto* call specifies a message to be sent to a socket address (see Figure 2). It hands the message to the underlying UDP and IP protocols and returns the actual number of characters sent. As we have requested datagram service, the message is transmitted to its destination without further ado or acknowledgement. The message will in fact be lost unless a process has already opened a socket and bound it to the destination socket address. Messages will if necessary be queued until *recvfrom* is called on that socket. If the message is too long to be sent, there is an error return (and the message is not transmitted). Most environments restrict the length of datagrams to 8 kilobytes. It is a good idea to limit the size of the messages sent by your programs to something acceptable by all of the computers in the network. I suggest you use 1K for most purposes, but if you need a larger message, then you could use up to 8K.

The socket address of the recipient must include the internet address of the computer and the agreed port number. In Figure 2, the *sendto* call uses the address filled in before the binding.

2.4 Receiving a message

The *recvfrom* from system call receives a single message via a socket into its buffer and returns the number of characters received.

int recvfrom(int s, char *buf, int len, int flags, struct sockaddr * from, int *fromlen)

Its use is illustrated in the *receiver* procedure of Figure 3.

```
/*receive two messages through port given as argument*/
void receiver(int port)
{
        char message1[SIZE], message2[SIZE];
        struct sockaddr_in mySocketAddress, aSocketAddress;
        int s,aLength, n;

        if((s = socket(AF_INET, SOCK_DGRAM, 0))<0) {
                perror("socket  failed");
                return;
        }
        makeReceiverSA(&mySocketAddress, port);
        if( bind(s, &mySocketAddress, sizeof(struct sockaddr_in))!= 0){
                perror("Bind  failed\n");
                close(s);
                return;
        }
        aLength = sizeof(aSocketAddress);
        aSocketAddress.sin_family = AF_INET; /* note that this is needed */
        if((n = recvfrom(s, message1,  SIZE, 0, &aSocketAddress, &aLength))<0)
                perror("Receive 1") ;
        }
        if((n = recvfrom(s, message2,  SIZE, 0, &aSocketAddress, &aLength))<0)
                perror("Receive  2");
        }
        close(s);
}
void makeReceiverSA(struct sockaddr_in *sa, int port)
{
        sa->sin_family  =  AF_INET;
        sa->sin_port = htons(port);
        sa-> sin_addr.s_addr  =  htonl(INADDR_ANY);
}
```

Figure 3 - sample receiving procedure

Once the *recvfrom* call has been invoked it will wait until a message arrives at the port number associated with its socket. As soon as a message arrives, it will collect it and return. The socket address of the sender is also supplied (via an argument) .

The first argument of *recvfrom* is a socket descriptor returned by *socket* system call. The next two arguments supply a buffer for receiving the message and specify the length of the buffer which should accommodate the maximum message size expected. The flags argument is normally 0. The 5th argument receives the socket address of the sender. The final argument tells the size of the buffer provided for the 5th argument and on return, supplies the actual size of the address stored.

2.5 Select

The *select* system call can be used to find out which of the file descriptors in your program currently has something ready to receive (or generally input or output). You can use it to avoid calling a blocking receive on a particular socket. It has a timeout parameter that may be set to any convenient value. This allows you to decide how long the select call should wait if no input arrives. It will return immediately input becomes available on any descriptor. The accompanying sample program shows an example of how to use *select*.

2.6 Close socket

When you have finished with a socket, the *close* call should be used to release the descriptor and the memory buffers.

2.7 Broadcast messages

Processes can send broadcast messages via the Ethernet. A broadcast message will be received by any process that has executed a *recvfrom* on the appropriate port number - a particular port number or any local port number. To address a broadcast message, set the Internet address of the destination to "192.135.233.255" for the network "197.135.233" shown in Figure 1.

```
void setBroadcast(int s)
{
        int arg;
        #ifdef SO_BROADCAST
        arg =1;
        if(setsockopt(s, SOL_SOCKET, SO_BROADCAST, &arg, sizeof(arg)) <0){
              perror("setsockopt SO_BROADCAST---");
              exit(-1);
        }
        #endif
}
```

Figute 4 - Broadcasting

Sockets contain information as to their functionality - for example, whether they can be used to transmit broadcasts. The usual default is not to be able to transmit broadcasts. The library function *setsockopt* - set socket option - can be used to set the parameter of the socket to allow it to transmit broadcasts. The procedure in Figure 4 will turn on broadcasting if necessary. See Figure 2 for an example of where to insert a call to this procedure.

3 Communication via streams

A pair of sockets may be ***connected*** in advance and then used for transmitting data using streams and TCP. The arrangement is asymmetric because one of the sockets will be listening for a request for a connection and the other will be asking for a connection. Once the pair of sockets has been connected, they may be used to send and receive information in both or either direction. That is - they behave like streams in that any available data is read immediately in the same order as it was written and there is no indication of boundaries of messages.

3.1 System calls for stream communication

socket: to create a socket and get a file descriptor for it

bind: to bind a socket address to the file descriptor of a socket

connect: to make a connect request

listen: to say how many requests should be queued

accept: to accept a connect request

write: send information via connected sockets

read: receive information via connected sockets

close: to destroy the socket when it is no longer needed (c.f. files)

Creation of a connection between two sockets usually requires that each socket has an address in a *sockaddr* structure bound to it. The process that *listens* for requests and responds to a *connect* request is sometimes called a server process - this is shown in Figure 5. The process that requests the connection is called the client process - it uses the *connect* system call.

3.2 Server or listening process

We consider now the server process with the socket that listens for a request for a connection. The procedures for the server are shown in Figure 5. The *startUp* procedure is intended to open a socket and listen on it for client requests. The *acceptConnection* procedure accepts a connection from a client and returns a new socket for the connection with the client. The *readAndWrite* procedure uses the new socket to receive the number of bytes given in the second argument and then send them all back to the client via the same connection.

```
int startUp(int serverPort)
{
        struct sockaddr_in serverSocketAddress;
        int s;
        s = socket(AF_INET, SOCK_STREAM, 0);
        makeReceiverSA(&serverSocketAddress, serverPort);
        if( bind(s, &serverSocketAddress, sizeof(struct sockaddr_in))!= 0)
              perror("Bind failed\n");
        listen(s,5);
        return(s);
}

/*server accepts connection through new socket that it returns */
int acceptConnection(int s)
{
        struct sockaddr_in clientSocketAddress;
        int sNew, clientLen;
        clientLen = sizeof(clientSocketAddress);
        if((sNew = accept(s, &clientSocketAddress, &clientLen)) <0) {
              perror("Accept fails:\n");
              close(s);
              exit();
        }
        return sNew;
}
```

Figure 5 - server listens and accepts connection, and returns new socket

The server process binds its socket to a socket address as in datagram communication and then gets ready to accept requests for connection. Note that the second argument to the *socket* system call is given as SOCK_STREAM, to indicate that stream communication is required. If the third argument is left as zero, the TCP protocol will be selected automatically.

The first stage is to use the *listen* system call to specify the maximum number of requests for connections that can be queued at this socket. This number is usually set to five and means that if the number of outstanding requests exceeds 5 they will be ignored. The second stage is to use the *accept* system call to *accept* any connection that is requested. These two stages are shown in Figure 5.

Figure 5 does not show the server closing the socket on which it listens. Normally a server would first listen and then fork a new process to accept the connection and communicate with the client. Meanwhile it will continue to listen in the original process.

```
void readAndWrite(int s, int amount) /*server receives amount bytes & sends it back*/
{
        int  n, nRead;
        char buf[SIZE], *p = buf;

        nRead = 0;
        if(amount>SIZE)perror("Amount too much");
        while (nRead < amount) {
              if((n = read(s, p, amount-nRead))<0) perror("Receive");
              else if(n==0) break;
              else{
                p += n;
                nRead += n;
              }
        }
        if((n =  write(s, buf, nRead) )< 0) perror("Send2  failed\n");
              else printf("wrote %d\n",n);
        close(s);
}
```

Figure 6- server receives data and sends it back

3.3 listen and accept

Once the server has opened its socket and bound it to the server port number, it *listens* on its socket for requests from clients for connections.

int listen(int s, int backlog);

The first arguments of the **listen** system call is a socket descriptor and the second is the number of requests for connection that can be queued at that socket. Once *listen* has been used on a socket, that socket will never be used directly for communication; its only purpose is to hold the queue of communication requests.

The *accept* system call acts rather like a combination of *socket* and *bind* - it returns a socket descriptor for a new socket bound with both local and remote addresses.

int accept (int s, struct sockaddr * clientAddress, int * clientLength);

The *accept* system call accepts the first connection in the queue at socket s. The second argument is for the socket address of the remote socket. The result is a descriptor for a new socket that has been created for use as one end of the stream.

3.4 Client process - requesting a connection

The client process makes a request for a connection. It uses the *connect* system call to request a connection via the socket address of the listening process. As the *connect* call automatically binds a socket address to the caller's socket, prior binding is unnecessary. Figure 7 shows a procedure that requests a connection and then uses it to send two messages and read any replies.

The *connect* call is used to request a connection from another process

int connect(int s, struct sockaddr *server, int addrLen)<0)

The first argument of the **connect** system call is a socket descriptor returned by the socket system call. The second argument is the socket address of a remote (server) socket. The 3rd argument gives the size of the of socket address structure in the 2nd argument.

```
int requestConnection(char *serverMachine, int serverPort)
{
        int s, n;
        struct sockaddr_in serverSocketAddress;
        s = socket(AF_INET, SOCK_STREAM, 0);
        makeDestSA(&serverSocketAddress, serverMachine, serverPort);
        if (connect(s, &serverSocketAddress, sizeof(struct sockaddr_in))<0) {
                perror("Connect fails:\n");
                close(s);
                exit();
        }
        return s;
}
void writeAndRead(int s, char *message1, char *message2, int len1, int len2)
{
        int n;
        char buf [SIZE];

        if((n = write(s, message1, len1))<0) perror("Send");
        if((n = write(s, message2, len2))<0) perror("Send");
        do{
                if((n = read(s, buf, SIZE))<0) perror("Receive");
        } while(n > 0);
        close(s);
}
```

Figure 7 - client requests connection and sends and receives messages via it

Figure 8 shows the sequence of events at client and server that lead up to the establishment of a connection. Note that both processes opens sockets and bind them. The server listens and offers to accept connections via its socket s2. The client requests a connection via its socket s1. The accept succeeds and a new socket s3 is opened for the connection to s1.

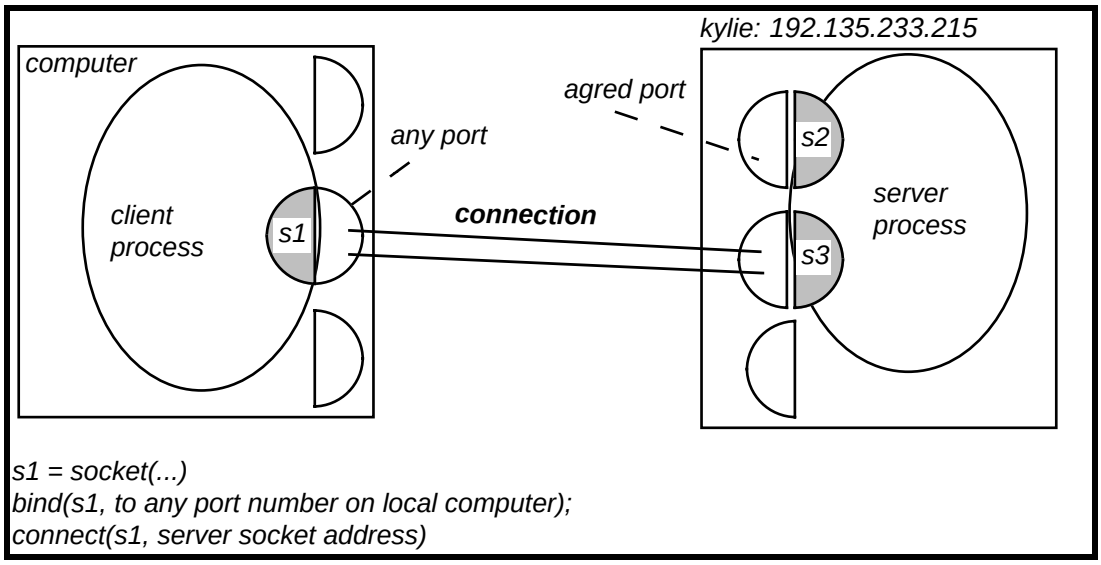

Figure 8 - client and server establish a connection

The ***write*** system call takes arguments specifying a socket descriptor, a message and its length. The *write* call is similar to the write call for files. It specifies a message to be sent to a socket address. It hands the message to the underlying TCP and IP protocols and returns the actual number of characters sent.

The *read* system call receives some characters in its buffer and returns the number of characters received. The connection behaves like a stream - any available data is read immediately, in the same sequence as it was written by the client write calls. There is no indication of message boundaries.

server	*client*
s2 = startUp(AGREED_PORT)	
s3=acceptConnection(s)	
	s1 = requestConnection("it063", AGREED_PORT)
readAndWrite(s3, 10)	**writeAndRead**(s1, "hello", "there", 5, 5)
close(s2)	

Figure 9- actions of server and client

Figure 9 illustrates how to start server and client processes that respectively call the procedures in Figure 6 and 7. In this example, the client process sends two messages; the server process can receive both the messages in the same read. On the other hand the read call may not get all the characters written at the first read. You have to call *read* repeatedly until you have read sufficient characters.

4 Conclusions

These notes have described the Unix IPC primitives.

If you need to implement clients and servers you should normally use remote procedure calling (RPC) or remote method invocation - it does all the work of unpacking the procedure name and arguments, selecting the procedure to be executed and returning the results. RPC facilities in Unix may be constructed in a layer above the Unix IPC system functions.

There are some situations in which you may need to use message passing, in which case IPC may be more appropriate.

The programs given in these notes are in an accompanying file.