CS 570 Class Notes Fall 1989
Handout 9
InterProcess Communication Using Sockets
by Raj Yavatkar

1 Introduction

Pipes are a simple solution for communicating between a parent and child or between child processes. What if we wanted to have processes that have no common ancestor with whom to set up communication? Unix provides an IPC mechnism called *sockets* for this purpose. Two processes can separately create sockets and then have messages sent between them. These two processes may reside on the same machine or on separate machines interconnected using a computer network. In Berkeley UNIX 4.3BSD, one can create individual sockets, give them names, and send messages between them.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is drawn is referred to as a *domain*. There are several domains for sockets. We will use a domain called the Internet domain (AF_INET) which is the UNIX implementation of the DARPA Internet standard protocols IP/TCP/UDP. DARPA Internet is a worldwide collection of several networks.

Communication follows some particular *style*. Currently, communication is either through a *stream* or by *datagram*. Stream communication implies several things. Communication takes place across a *connection* between two sockets. Before beginning the communication, two processes execute a series of steps to ensure that both are ready to exchange messages. The result of this negotiation is a *connection*. The communication over a stream is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to *write()* on the sender's side or only part of the data from a single call, if there is not enough room for the entire message, or if not all the data from a large message has been transferred.

Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

Actual transfer of data takes place using a *protocol*. A protocol is a set of rules, data formats and conventions that regulate the transfer of data between participants in the communication. In general, there is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names of the sockets, sets up connections and transfers data between sockets, perhaps sending the data across a network.

One specifies the domain, style, and protocol of a socket when it is created. For example, the following Unix system call to *sockect* causes the creation of a stream socket with the protocol in the Internet domain.

socket(AF_INET, SOCK_STREAM, 0);

The constants AF_INET and and SOCK_STREAM are defined in $\langle sys/socket.h \rangle$. After a socket is created, it must be given an appropriate name in its domain. In the Internet domain, sockets are identified using a 8-byte transport address consisting of two parts: address of the host or machine on which the socket exists and a port number, or a delivery slot, on that machine. The following structure defined in file "netinet.h" describes the socket address.

```
struct sockaddr_in { /* Internet family socket address */
        short    sin_family;     /* adresss family */
        short    sin_port;       /* 2 byte port number */
        long     sin_addr;       /* 4 byte machine address */
        char     sin_data[8];    /* unused */
};
```

When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered.

For a given protocol, a socket is thus identified using the tuple

$$\langle protocol, localmachineaddress, localport \rangle.$$

An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the tuple

$$\langle protocol, localmachineaddress, localport, remotemachineaddress, remoteport \rangle.$$

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value INADDR_ANY.

Figure 1 shows an example program for sending datagram to a *well-known* server called the UDP *echo server* residing on machine s.

To determine a network address to which it can send the message, it looks up the host address by the call to *gethostbyname()*. The returned structure includes the host's network address, which is copied into the structure specifying the destination of the message.

The port number can be thought of as the number of a mailbox, into which the protocol places one's messages. Certain daemons, offering certain advertised services, have reserved or "well-known" port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number.

Port number for a well-known service may be obtained by using the system call *getservbyname(family, service).*

1.1 Binding local names to sockets

A soocket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, therefore, no messages may be received on it. the *bind* system call allows a process to specify local or its own half of the *association*, $\langle localaddress, localport \rangle$ and the *connect* and *accept* calls are used to complete a socket's association.

The *bind* system call is used as follows:

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

struct sockaddr_in sname;

        sname.sin_family = AF_INET;
        sname.sin_port = PORTNUM;
        sname.sin_addr.s_addr = INADDR_ANY;

        bind(fdTo,(char *)&sname, sizeof(sname));
```

1.2 Connection Establishment

A connection is typically used for client-server interaction. A server advertizes a particular server at a well-known address and clients establish connections to that socket to avail of the offered service. Thus the connection estblishment procedure is asymmetric.

A server creates a socket, binds it to a "well-known" port number associated with the service, and then passively "listens" on the socket for requests to be served. It is possible for any unrelated process to rendezvous with the server. A client requests services from a server by initiating a "connection" to the server's socket. The client uses the *connect* system call to initiate a connection. For example, the following call establishes a connection to a socket whose address is specified using the variable "sname".

```
struct sockaddr_in sname;
int s; /* socket descriptor returned by system call socket */

       sname.sin_family = AF_INET;
       sname.sin_port = PORTNUM; /* well-known port no */
       sname.sin_addr.s_addr = /* host address of the server */;

       connect(s, &sname, sizeof(sname));
```

For the server to receive a conection, it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
    listen(fdTo,5);
```

The second parameter specifies the maximum number of outstanding connections that may be queued by the system when server is busy.

Once a socket is marked as listening, a server may *accept* a connection:

```
struct sockaddr_in from;
int fromlen;

       fromlen  = sizeof(from);
       newsock = accept(fdTo, &from,  &fromlen);
```

A new socket descriptor is returned on receipt of a connection. The identity of the client requesting connection is returned in the structure *from* and the length of that information is returned in *fromlen*. Figure 2 shows program for a remote login server that accepts requests for remote login from remote clients. Note that "login" is a well-known servicve and therefore has a well-known port number.

Once a connection is established, both client and server may exachnge data using several system calls.

1.3 Data Transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are usable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

The *send* and *recv* are virtually identical to *read* and *write*, and the extra *flags* argument is important only in special circumstances.

1.4 Connectionless Sockets

A datagram socket provides a *connectionless* communication interface. Under this model, communicationg processes need not set up a connection before they exchange messages. Instead, sender specifies a destination address in each message. There is no gurantee that the recipient will be ready to receive the message and there is no error returned if the message cannot be delivered. Messages are sent and received using the system calls *sendto* and *recvfrom*.

Datagram sockets are created as before. If a particular local address is needed, the *bind* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return 1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

Figure 3 shows an example of a user-defined datagram-based server that echoes back messages sent by a client.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time; a second connect will change the destination address, and a connect to a null address (family AF_UNSPEC) will disconnect. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). *Accept* and *listen* are not used with datagram sockets.

```
                          /* Figure 1 */
/* a client program that communicates with the UDP echo server
using Internet datagrams */

/* include files omitted */
main()
{
        struct sockaddr_in  saTo;
        int  fdTo, fromlen;
        struct servent        *pServent;
        struct hostent *pHostent;
        char        *snd_buf;  /* must allocate memory */
        int        cch;

        /* create a socket */
        fdTo = socket(AF_INET, SOCK_DGRAM, 0);

        /* first find the address of the server */
        pServent = getservbyname("echo", "udp");

        /* find the machine address given a host name */
        pHostent = gethostbyname("s.ms.uky.edu");

        /* fill in info inthe socket address sturcture */
        bzero((char *)&saTo, sizeof(saTo));
        bcopy(pHostent->h_addr, (char *)&saTo.sin_addr,pHostent->h_length);
        saTo.sin_family = AF_INET;
        saTo.sin_port = pServent->s_port;

        /* send a message */
        cch = sendto(fdTo, tmp_buf, strlen(tmp_buf), 0, &saTo, sizeof(saTo));

        ........
}
```

```
                          /* Figure 2 */
/* Remote login server program */

main()
{
        struct sockaddr_in  from;
        int    fd, rstat;
        struct servent        *pServent;

        fdTo = socket(AF_INET, SOCK_STREAM, 0);

        pServent = getservbyname("rlogin", "tcp");

        bzero((char *)&from, sizeof(from));

        from.sin_family = AF_INET;
        from.sin_port = pServent->s_port;
        from.sin_addr.s_addr = INADDR_ANY;

        /* bind bind */
        fd(fdTo,(char *)&from, sizeof(from));

        listen(fd, 5);                /* listen to requests */
        while(1) {
            fromlen = sizeof(from);
            rstat = accept(fd, &from, &fromlen);

            /* error checking */
            /* fork off a chilld to handle this connection */
        }
}
```

```
                        /* Figure 3 */
/* An example of a simple, datagram-based echo server that echoes
   back client messages.*/
#include <stdio.h>
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORTNUM 2005    /* user defined port, not a well-known one */

main()
{
        struct sockaddr_in  saTo;
        int  fdTo, fromlen;
        char        *req_buf;  /* must allocate memory */
        int         cch;

        fdTo = socket(AF_INET, SOCK_DGRAM, 0);

        bzero((char *)&saTo, sizeof(saTo));
        saTo.sin_family = AF_INET;
        saTo.sin_port = PORTNUM;
        saTo.sin_addr.s_addr = INADDR_ANY;

        /* bind fd */
        bind(fdTo,(char *)&saTo, sizeof(saTo));

        while(1) {        /* server loop */
            fromlen = sizeof(saTo);
            cch = recvfrom(fdTo, req_buf, 80, 0, &saTo, &fromlen);

            /* reply */
            sendto(fdTo, req_buf, cch, 0, &saTo, sizeof(saTo));
        }
}
```