Pipes and Sockets

1. Introduction

A 'pipe' is a conduit that carries data from one process to another. A character written into the input end of the pipe is available to be read from the output end of the pipe. Pipes are essential to Unix programming. Programs are designed and written to perform specific tasks on a stream of data. Data flows from one special-purpose program to the next along these pipes just as automobiles flow from one special-purpose worker to the next along an assembly line. Although this strict separation of tasks is not always ideal for human workers, it has proven to be an effective and reliable system for processing data.

A logical extension of the idea of separating tasks into special programs operating concurrently connected by pipes is to run tasks on separate machines and connect them by pipes that flow from one machine to another. Such a design allows for true parallel processing, distributed data, and inter-computer communication. The original Unix pipe cannot connect processes on separate machines. A new inter-process data conduit, called a **socket**, performs this function.

2. An Example: Your Watch Stops

A simple example will introduce the major concepts involved in using sockets. You wake up in the morning and find that your watch has stopped. You need to set your watch, so you dial NER-VOUS, and a computerized voice at the other end of the connection tells you the time. You set your watch and hang up.

In network jargon, the computer that answered the call is a *time server*. It provides a service to anyone who calls it. The telephone number you dialed (NER-VOUS) is the *network address* of the service. This address has three parts: the NER part identifies what telephone exchange it belongs to, the VOUS part identifies a *port* in that exchange, and the fact that you are using the telephone rather than tuning in a time station on the radio means you are working in the telephone *address family*. The time server was passively awaiting requests for service. It *accepted* your call. You, on the other hand, were actively seeking the service. You are a *client*. You *connected* to the server and read data that the server *wrote* over the wire.

Your telephone has a network address, but neither you nor the time server needed to know it in order to connect and satisfy your request for service. If the time server billed for calls, though, it would need to know your network address. It could get this information if it wanted to.

3. A Parallel Example: Your Computer's Watch Stops

You turn off your computer to clean it, and when you turn it back on, its internal clock is wrong. In the old days, it would ask you, the human, for the time. Now, in the network era, it asks another computer.

Somewhere on the network is a computer running a program that knows what time it is. That program is a *time server*. It sleeps in the background waiting for calls from other programs. This server process has a *network address* which consists of three parts. The first part is the *address family* (more later about that), the second part is the machine identification number (a *host number*), and the third part is the service number for the time server (a *port number*.) This process passively awaits connections.

When your computer needs the time, it picks up a communication line (like your picking up the phone), it then connects that line to the service (like your dialing the time number). The server accepts the call and a data channel is established. The server writes some data into this channel and your computer reads that

data at its end. Your computer uses that information to set its own clock. Then it hangs up. The server, though, continues to run, waiting to accept another call.

4. Other Examples: Directory Assistance

The time server is simple, but it includes almost every aspect of network communication. The only aspect that is missing is that of an interesting *protocol*. The procedure for getting the time is to connect to the server, receive a message, then hang up.

Consider directory assistance. You can call a number to ask for Jane Doe's phone number. In this case, you use a more complicated protocol. The phone company has a phone number server. It waits passively to accept calls. You get a socket (pick up a phone), dial the server's network address (phone number) to connect to the server. When the server accepts the connection (picks up the phone), you begin a dialog. You tell it the name, it responds with a number. You give another name, it gives another number. You can do this a few times until you are done or the protocol limit is reached. You close the dialog with something like 'thanks, bye now.' and the server acknowledges the termination request with 'you are welcome, call again' and then the connection is closed. The server goes back to waiting, and you go on with your life.

5. Ok, Ok, I Get the Idea. What about these Sockets?

Now that you have the idea, let's write a time server that uses sockets. This program will run in the background, waiting to accept calls at a specified network address. When it gets a call, it will send the time and date, as a simple ascii string (from ctime()) down the wire to the calling party. It will then hang up. The code looks a little messy, but the ideas have all been covered in the above examples:

In this first part, we create our network address. This address goes into a struct of type 'sockaddr_in' which is an address for sockets in the internet address family. The three parts of the address are the family type, the machine the server is running on, and the specific port number (which we select arbitrarily to be 2000). Directory assistance is 411 everywhere, which is an arbitrary, but commonly known number. All we need to do is tell everyone that our number is 2000.

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

#define   PORTNUM  2000       /* our time service phone number */
#define   oops(msg)      { perror(msg) ; exit(1) ; }

void main(ac, av)
char **av;
{
        struct  sockaddr_in   saddr;   /* build our address here */
     struct    hostent         *hp;   /* this is part of our    */
     char hostname[256];         /* address             */
     int  slen,sock_id,sock_fd;  /* line id, file desc     */
     FILE *sock_fp;              /* use socket as stream   */
        char    *ctime();             /* convert secs to string */
        long    time(), thetime;      /* time and the val       */

       /*
        *      step 1: build our network address
        *               domain is internet, hostname is local host,
        *               port is some arbitrary number
        */

       gethostname( hostname );               /* where am I ?         */
       hp = gethostbyname( hostname );        /* get info about host  */

       bzero( &saddr, sizeof(saddr) );        /* zero struct          */
                                              /* fill in hostaddr     */
       bcopy( hp->h_addr, &saddr.sin_addr, hp->h_length);
       saddr.sin_family = AF_INET ;           /* fill in socket type  */
       saddr.sin_port = htons(PORTNUM);       /* fill in socket port  */
```

Now we have an address. The next steps are to get a line put in, attach our address to that line, and then start listening for calls on that line. The system call to get a line is 'socket()' ; the call to attach our selected phone number to that line is 'bind()'; the call to announce we want to receive calls on that line at that number is 'listen()'. It is no more complicated than getting a phone set up for incoming business calls.

```
       /*
        *      step 2: ask kernel for a socket, then bind address
        */

       sock_id = socket( AF_INET, SOCK_STREAM, 0 );    /* get a socket */
       if ( sock_id == -1 ) oops( "socket" );
       if ( bind(sock_id, &saddr, sizeof(saddr)) != 0 )/* bind it to   */
               oops( "bind" );                         /* an address   */

       /*
        *      step 3: tell kernel we want to listen for calls
        */

       if ( listen(sock_id, 1) != 0 ) oops( "listen" );
```

The call to socket() takes three arguments. The first argument is the address family. We are going to take calls over the network, and use the 'internet' addressing scheme. The second argument is the type of line we want. The SOCK_STREAM type is like a pipe or file descriptor; it is sequential, reliable delivery of data. Accept nothing less. The third argument is the protocol type, and that does not apply to streams in the internet family, so we leave it zero. We get in return a socket_id number.

The call to bind() takes three arguments. The first argument is the socket_id. We need to attach our phone number to the socket, so we have to tell the kernel which socket to identify. The second argument is the socket address we want to attach to that socket. We have selected and created our address in the first part of

the program. The last argument is the size of the address. Different sorts of addresses have different lengths so we need to be specific.

If bind() succeeds, we have a working line with our choice of address, and we just tell the kernel to allow incoming calls on that socket.

Now we enter a simple loop to take calls, get the time and write the time over the line.

```
        while ( 1 ){
                sock_fd = accept(sock_id, NULL, NULL); /* wait for call */
                if ( sock_fd == -1 )
                        oops( "accept" );       /* error getting calls  */

                sock_fp = fdopen(sock_fd,"w");  /* we'll write to the   */
                if ( sock_fp == NULL )          /* socket as a stream   */
                        oops( "fdopen" );       /* unless we can't      */

                thetime = time(NULL);           /* get time             */
                                                /* and convert to strng */
                fprintf( sock_fp, "%s", ctime(&thetime) );
                fclose( sock_fp );              /* release connection   */
        }
}
```

And that concludes the construction of our time server. This program should be run in the background. It will set up the socket, bind its address to the socket, take calls. Each call will be accepted. The accept() system call takes three arguments, the first is the socket on which to take calls. The other args would point to a buffer for the address of the caller and an int to hold the length of the caller's address. Since we don't care who calls, we put NULLs there. If we wanted to keep a log of who asked for the time, we could use the other two arguments.

The accept() call returns a regular file descriptor, suitable for read()ing and write()ing just like any old file or pipe.

6. The Client End

The code shown above implements the server. This server runs on some machine ready to take calls and tell the clients the time. The client program has to get a socket, connect to the server, and read the time from the socket. The code is:

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

#define        PORTNUM         2000
#define        MAXHOSTLEN      255
#define        oops(msg)       { perror(msg); exit(1); }


main()
{
     struct sockaddr_in  servadd;        /* the number to call */
     struct hostent      *hp;            /* used to get number */
     int    sock_id, sock_fd;            /* the socket and fd  */
        char   message[BUFSIZ];             /* to receive message */
        int    messlen;                     /* for message length */
     char   host[MAXHOSTLEN];

       /*
        *      build the network address of where we want to call
        */

       if ( gethostname(host, MAXHOSTLEN) == -1 ) oops("gethostname");
       hp = gethostbyname( host );
       if ( hp == NULL ) oops("no such computer");

       bzero( &servadd, sizeof( servadd ) );   /* zero the address     */
       servadd.sin_family = AF_INET ;          /* fill in socket type  */
                                               /* and machine address  */
       bcopy( hp->h_addr, &servadd.sin_addr, hp->h_length);
       servadd.sin_port = htons(PORTNUM);      /* host to num short    */
```

This code builds the 'phone number' of the time service. The three parts are included: the address family (AF_INET), the machine the server runs on (the host), and the port on that machine (2000). All we need to do is get a socket and use that socket to connect to that address. Here goes..

```
       /*
        *      make the connection
        */

       sock_id = socket( AF_INET, SOCK_STREAM, 0 );    /* get a line   */
       if ( sock_id == -1 ) oops( "socket" );          /* or fail      */
                                                       /* now dial     */
       if ( connect( sock_id, &servadd, sizeof(servadd)) !=0  )
               oops( "connect" );

       /*
        *      we're connected to that number, read from socket
        */

       messlen = read( sock_id, message, BUFSIZ );     /* read stuff   */
       if ( messlen == - 1 )
               oops( "read" ) ;
       if ( write( 1, message, messlen ) != messlen )  /* and write to */
               oops( "write" );                        /* stdout       */
       close( sock_id );
}
```

The client end is different from the server end in the following crucial way. For the server, the accept() call returns a file descriptor. The socket is left untouched and is available for later accept() calls. In the client, though, the connect() call connects on the socket_id; it does not generate some other number for the connection.

7. Phones are Bidirectional, What About Sockets?

Sockets are different from pipes. A socket is a bidirectional channel. You do not need two sockets to have a dialog. When one process writes into the socket, the other one reads the data from the other end. When that process then writes its response into the socket, the first process reads the response.

The coordination of question and response must be established by a protocol set up by the programs. Just as you and the directory assistance person can confuse each other by talking at the same time, a network server and client can confuse each other by violating their protocol.

8. Can You Give Me An Example?

Here is a server/client pair for remote execution of commands. The server accepts one shell command from the client and sends the output of the command back to the client. When the server receives the 'quit' command, it shuts down operation.

This program seems limited. It only accepts one command. Why not let it take a command from the client, send the output back through the socket, then accept another command, run it, then another command, just like a regular shell? Think this through carefully; how can the client/server pair interact the same way a human and a shell interact?

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

/*
 *    rem_exec daemon - accepts a shell commmand, sends its output
 */
#define    PORTNUM    2001
#define    oops(msg)  { perror(msg) ; exit(1) ; }

void main(ac, av)
char **av;
{
     struct     sockaddr_in     saddr;     /* a struct to hold a socket address */
     struct     hostent         *hp;
     char hostname[256];        /* get hostname here */
     int  slen, sock_id, sock_fd, ch;
     FILE *sock_fp, *cmdfp;
     char cmd[BUFSIZ];          /* command to execute */
     int  cmdlen;

     /*
      *    step 1: build our network address
      *          domain is internet, hostname is local host,
      *          port is some arbitrary number
      */
     gethostname( hostname );        /* where am I ?      */
     hp = gethostbyname( hostname );      /* get info about host    */
     bzero( &saddr, sizeof(saddr) );      /* zero struct       */
                              /* fill in hostaddr  */
     bcopy( hp->h_addr, &saddr.sin_addr, hp->h_length);
     saddr.sin_family = AF_INET ;         /* fill in socket type    */
     saddr.sin_port = htons(PORTNUM);     /* fill in socket port    */
```

```
        /*
         *   step 2: ask kernel for a socket, then bind address
         */
        sock_id = socket( AF_INET, SOCK_STREAM, 0 );   /* get a socket */
        if ( sock_id == -1 ) oops( "socket" );
        if ( bind(sock_id, &saddr, sizeof(saddr)) != 0 )/* bind it to  */
             oops( "bind" );                /* an address   */

        /*
         *   step 3: tell kernel we want to listen for calls
         */
        if ( listen(sock_id, 1) != 0 ) oops( "listen" );
        while ( 1 ){
             sock_fd = accept(sock_id, NULL, NULL); /* wait for call */
             if ( sock_fd == -1 ) oops( "accept" ); /* error taking call */

             cmdlen = read( sock_fd, cmd, BUFSIZ ); /* get command   */
             if ( strncmp( cmd, "quit", 4 ) == 0 )  /* exit request? */
                  exit(0);
             if ( cmdlen == -1 ) oops( "read" );
             cmd[cmdlen] = '\0';

             sock_fp = fdopen(sock_fd,"w"); /* let's buffer output  */
             if ( sock_fp == NULL )         /* make it a stream     */
                  oops( "fdopen" );    /* unless we can't   */

             cmdfp = popen( cmd, "r" );/* and read from the */
             if ( cmdfp == NULL )      /* command      */
                  fputs( "didnt work\n", sock_fp);
             else {
                  while ( (ch = getc( cmdfp )) != EOF )
                       putc(ch, sock_fp );
                  pclose( cmdfp );
             }
             fclose( sock_fp );        /* release connection     */
        }
   }
```

client program on next page

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

/*
 *   rem_exec client program, read a command and return its output
 */
#define         HOSTNAME   "host2"
#define         PORTNUM         2001
#define         oops(msg)  { perror(msg); exit(1); }

main()
{
     struct sockaddr_in rxa;
     struct hostent    *hp;
     FILE *sock_fp;
     int  sock_id, sock_fd, ch;
     char cmd[BUFSIZ];

     /*
      *   build the network address of where we want to call
      */
     hp = gethostbyname( HOSTNAME );
     if ( hp == NULL ) oops("no such computer");

     bzero( &rxa, sizeof( rxa ) );        /* zero the address  */
     rxa.sin_family = AF_INET ;      /* fill in socket type    */
                              /* and machine address    */
     bcopy( hp->h_addr, &rxa.sin_addr, hp->h_length);
     rxa.sin_port = htons(PORTNUM);       /* format number     */

     /*
      *   make the connection
      */
     sock_id = socket( AF_INET, SOCK_STREAM, 0 );   /* get a line   */
     if ( sock_id == -1 ) oops( "socket" );         /* or fail */
     if ( connect( sock_id, &rxa, sizeof(rxa))!=0 ) /* dial number  */
          oops( "connect" );

     /*
      *   we're connected to that number,
      *   open socket as a readable stream and copy to stdout
      */
     gets( cmd );
     if ( write( sock_id, cmd, strlen(cmd) ) == -1 )
          oops( "write" );
     sock_fp = fdopen( sock_id , "r" );        /* open to read */
     if ( sock_fp == NULL )
          oops( "fdopen" ) ;
     while ( (ch = getc(sock_fp)) != EOF )          /* copy from    */
          putchar( ch );                 /* stream  */
     fclose( sock_fp );
}
```