Inter-Process Communication Using TCP/IP

Mark Sitkowski C.Eng, M.I.E.E
http://www.designsim.com.au

Consider the situation, where we have forked twenty, or so child processes, each of which runs a cursor in the database. The child processes save the data in local data structures then, when the cursors have run to completion, they need to send their data back to the parent.
We have a number of inter-process communication techniques, capable of achieving this, which we will examine in detail.

TCP/IP Client-Server Systems

When we reach the point of requiring the data from our child processes, we call a function within the parent, which makes it act as a TCP/IP server. We can then transfer all of the data from our child processes, in parallel, back to the parent.
The transfer time, compared to sequential database access, is the run-time of the slowest child, as opposed to the run-time of twenty children.
The client-server configuration is probably the most efficient technique, for transferring data from a series of child processes, back to the parent. The data flows through the machine's local loopback connection, at the maximum speed of which Ethernet is capable.
Clearly, since the connection from client to server uses the TCP/IP protocol, the client processes can be running on any machine, anywhere in the world. However, we are only concerned with performance-critical applications, so we will confine this discussion to both client and server being resident on the same machine.
The code required to create either a client or server, is boilerplate code, which was laid down by Berkeley, and leaves no room for imaginative variations. If you do anything, apart from renaming the variables, it probably won't work.

The sequence of events for making a TCP/IP server process goes as follows:

1. Determine the socket type
2. Determine which port we're going to use
3. Create a socket
4. Bind the socket to the port
5. Listen to the socket
6. Accept a connection
7. Read and write to the socket

For obvious reasons, the server has to be created, and listening before the client can make a connection, so we have described it first.

Making the client is equally straightforward:

1. Determine the socket type
2. Determine which port we're going to use
3. Create a socket
4. Connect the socket to the port
5. Read and write to the socket

Server

If we examine the sequence of events in the server, we can see an obvious problem. As soon as a connection is made, the server is tied up, handling the client's request, so we need another step.
The accept() system call (server, step 6) returns a socket descriptor. This is a duplicate of the socket, on which the server is listening, and is bound to the lowest available port number. This means that, although all clients can continue to connect to the same port number, the actual conversation with the server occurs on a completely different port.
If we were only interested in receiving data from a multitude of remote clients, we would adopt the following technique:

1. Accept a connection from a client
2. Fork the server, passing the duplicate socket descriptor to the child
3. In the server's child , call a function to service the request arriving on the socket.

Unfortunately, this doesn't help us at all. The data from our child process, instead of arriving at the server, would now be sent to another child, and be lost when that child terminated. Whatever means we choose, all incoming data must remain within the parent process. Here's how we do it:

1. Accept a connection from the client
2. Start a thread, which calls a function to handle the client's request.

Using this approach, all incoming data remains within the parent and, since we are multi-threading the client handling, we can receive the data from our twenty child processes in parallel.

This leaves only one question unanswered. How do we know, when all of our child processes have completed their task?
Presumably, we counted them, in the parent, as we forked them, which means that we have a variable, sitting somewhere, with the number '20' in it. As each connection is accepted, we can decrement this variable and, when it reaches zero, we can use the wait() system call, to tell us when the last child finished its work. Then, we can stop being a server, and go and process the data we just acquired.
For the example which follows, we will assume that this approach is adequate but, what happens if one (or more) of our children exit unexpectedly, before they have had a chance to connect back to us?
The server will hang, waiting forever for a connection, which never comes.
One possible solution, is to actively monitor our children, by handling SIGCHLD, in our interrupt handler. The handler can then decrement our child count variable, and return to the server loop. This is a far more reliable method, but it still has one flaw. If a second interrupt occurs, while we are in the interrupt handler, we will miss it, and hang, waiting forever, for a child which has already terminated.
Perhaps the real question to be answered is, what are the implications of a child which exits prematurely?
Presumably, it met with a problem and, as a result, it would not be sending us any data. In that case, we need the child to inform us of this fact, so we can abort the current operation and quit. Accordingly, we would probably arrange for each child to

send us SIGUSR1 (or any other convenient signal) for all fatal errors. Our own signal handler can then allow us to quit gracefully.

Now, for the boilerplate code. There are three data structures, which form the basis of all TCP/IP transactions:

```
1. struct  hostent {
        char    *h_name;          /* official name of host */
        char    **h_aliases;      /* alias list */
        int       h_addrtype;     /* host address type */
        int       h_length;       /* length of address */
        char    **h_addr_list;    /* list of addresses from name server */
#define h_addr  h_addr_list[0]  /* address, for backward compatiblity */
};
```

The only contribution made by this structure is the member h_addrtype, which is used to load the sin_family member of the sockaddr_in structure. Since the value of this is always the symbolic value AF_INET, it is possible to load sin_family directly with this value.
However, a much more portable approach, is to call gethostbyname(), and pass it the machine name. The IP address, and other useful information is then extracted, either from the DNS server or, if that is unavailable, from /etc/hosts.

```
2. struct  servent {
        char     *s_name;       /* official service name */
        char    **s_aliases;    /* alias list */
        int       s_port;       /* port # */
        char     *s_proto;      /* protocol to use */
};
```

As with the previous structure, the only contribution from servent, is one piece of information – the port number.
Before the client and server can connect, they must agree on a port, at which the server will listen, and to which the client will connect. Rather than pass a port number from server to client, it is best to use a 'well-known port'. This is a port which is defined in the file /etc/services, by a symbolic name, and which can be queried through a call to getservbyname().. If the root password is unavailable, thus precluding editing /etc/services, it is still possible to use a well-known port, by picking one which is either experimental, obsolete, or unused. Examples of these are: Ingreslock, listen, new-rwho, monitor, rmonitor, pcserver etc.

```
3. struct sockaddr_in {
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct  in_addr sin_addr;
    unsigned char  sin_zero[8];
};
```

Now that we have a port number, and the value AF_INET, we can fill in the two appropriate members of the sockaddr_in structure, and pass it to the bind() system call, to bind a socket to that port number.
First, we need to make a socket;
The socket() system call returns a socket descriptor, in much the same way that open() returns a file descriptor. The arguments passed to socket(), are

- Domain – which is AF_INET, for our purposes. The other kind of socket is AF_UNIX, which creates a disk-based named socket, quite unsuitable for our purpose.
- Type: - which is SOCK_STREAM in our case, since we are interested in reliable, two-way, sequential data transfers.
- Protocol – which we set to the symbolic value PF_UNSPEC, i.e zero, and let the system figure it out. Strictly speaking, we should use PF_INET, for IPV4, and PF_INET6 for IPV6.

Since we need to make our network variables accessible to the threads we intend to launch, the above structures, which are referenced by both client and server, are declared globally:

```
struct sockaddr_in sa, sb;              /* one for the server, one for the client */
struct hostent *ha, *hb;                /* one for the server, one for the client */

struct servent *svs, *svc;              /* one for the server, one for the client */

int c, s, ds;                           /*
                                         * socket descriptors for
                                         * client, server, and duplicate for accept()
                                         */
char hname[2048];                       /* where we put our machine name */
unsigned short sport, cport;            /* server port, client port */
```

We code our server function as follows:

```
server()                /* server */

{

int status;
int th = 0;

   memset(&sa, 0, sizeof(struct sockaddr_in));          /* initialise the structure */
   gethostname(hname, sizeof(hname));                   /* find out who we are */

   if((ha = gethostbyname(hname)) == NULL){             /* fill in part of this structure */
      exit(-1);
   }
   if((svs = getservbyname("ingreslock", "tcp")) == NULL){  /* get the port number */
      printf("%s doesn't exist\n");
      exit(-1);
   } else {
      sport = svs->s_port;
   }
   printf("Server listening to port %d on host %s\n", sport, hname);
   sa.sin_family = ha->h_addrtype;                      /* fill in this structure */
   sa.sin_port = htons(sport);

   if((s = socket(AF_INET, SOCK_STREAM, 0)) < 0){    /* make a socket */
      exit(-1);
   }
```

```
                                                    /* bind the socket to the port */
    if(bind(s, (struct sockaddr *)&sa, sizeof(struct sockaddr_in)) < 0){
        printf("Can't bind to port %d\n", sport);
        close(s);
        exit(-1);
    }

    th = 0;

    listen(s, 3);                                   /* listen to the port */

    while(1){                                       /* service connections forever */

        if((ds = accept(s, NULL, NULL)) < 0){       /* get the dup'd socket descriptor */
            exit(-1);
        } else {
            children--;                             /* one more has connected */
            if((pthread_create(&thr[th], NULL, xserve, (void *)&ds)) != 0){
                printf("Failed to create thr[%d]\n",th);
            }
            if(pthread_detach(thr[th]) != 0){
                printf("Failed to start thr[%d]\n",th);
            }
            th++;
        }
        printf("Threads running: %d Children: %d\n", th, children);
        if(children <= 0){
            waitpid((pid_t)-1, &status, 0);
            printf("Last thread terminated. Operation complete\n");
            break;
        }
    }
    close(s);

}                       /* server */
```

Most of the above code is fairly straightforward, but the following points may be worth noting.

The listen() system call takes, as arguments, a socket descriptor and a number, which represents the maximum number of pending connections which it is willing to queue. This number varies from one version of Unix to another, and is, variously, between 3 and infinity. Although we expect to handle all incoming connections as fast as they arrive, there is always the possibility that several connection requests will arrive, virtually simultaneously. This might create a temporary backlog, for a few milliseconds, which needs to be queued, or we might lose a client connection.

As mentioned in the preamble to this section, the accept() system call returns a duplicate socket descriptor, before resuming its interrogation of the original socket. The two arguments which we have set to NULL, are the returned values of the IP address, and its length, of the connecting client. These are of no interest to us, in this application.

As each connection is accepted, we create a thread, and call the connection handler, xserve(), passing to it the dup'd socket descriptor, which it will have to read, in order

to communicate with the connecting client. Since we do not want to wait for the thread to terminate, we detach it, and go back to wait for the next connection.

With each accepted connection, we count down the number of outstanding child processes. When the last one makes a connection, we call waitpid(), with the argument –1, after the thread has detached. Here we wait for the child to terminate, then break out of the accept() loop.

Client

The steps involved in making a client are simpler than those for making a server. First, we gather the information needed to create a socket, namely the domain, which is AF_INET, the socket type, which is SOCK_STREAM, and the protocol type, which we set to zero.
Then we create the socket, in exactly the same way:

```
if((c = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        exit(-1);
 }
```

When we have determined the port number, with a call to getservbyname(), we can execute the connect() system call, and connect to the server.

Connect() takes three arguments. The first is the socket descriptor, the second is a structure called sockaddr (not sockaddr_in) and the last is the structure's size.
The sockaddr structure is defined :

```
struct sockaddr {
        sa_family_t     sa_family;      /* address family */
        char            sa_data[14];    /* up to 14 bytes of direct address */
};
```

This means that a cast is possible from sockaddr_in, since the 14 element array of char can overlay the port number and address information in sockaddr_in.

Unlike the server, where we make virtually no use of the information in sockaddr_in, the client needs to know this address, as the server may not be running on the same machine as the client. Accordingly, we have the additional operation:

```
memcpy((char *)&sb.sin_addr, hb->h_addr, hb->h_length);
```

which tells the connect() system call the server's IP address.

```
if(connect(c, (struct sockaddr *)&sb, sizeof(sb)) < 0){
        printf("Client %d:Unable to connect to %d\n", getpid(), cport);
        close(c);
        exit(-1);
   }
```

The complete listing of the client, is as follows:

```
client(char *cmesg)             /* client */
```

```
{

int nread;              /* how much we read */
int nwrite;             /* how much we wrote */
char cresp[2048];  /* buffer for server's reply */
   gethostname(hname, sizeof(hname));

   if((hb = gethostbyname(hname)) == NULL){    /* get server's IP address */
      perror("gethostbyname");
      return(-1);
   }
    /* get the port number */
   if((svc = getservbyname("ingreslock", "tcp")) == NULL){
      printf("Client %d: service 'ingreslock' doesn't exist\n", getpid());
      return(-1);
   } else {
      cport = svc->s_port;
   }
   memset(&sb, '\0', sizeof(sb));
   memcpy((char *)&sb.sin_addr, hb->h_addr, hb->h_length); /* set address */
   sb.sin_family = hb->h_addrtype;                         /* family is AF_INET */
   sb.sin_port = htons((u_short)cport);                    /* Port number */

   /* Now create the socket */
   if((c = socket(hb->h_addrtype, SOCK_STREAM, 0)) < 0){
      perror("socket");
      return(-1);
   }
    /* connect to the server */
   if(connect(c, (struct sockaddr *)&sb, sizeof(sb)) < 0){
      printf("Client %d:Unable to connect to %d\n", getpid(), cport);
      perror("Connect");
      close(c);
      return(-1);
   }
   printf("Client %d (%d): Connected to server\n", getpid(), which);

   /*
    * Now we can talk to the server
    */
   xclient()
   return(0);


}                  /* client */
```

Data Types

The client processes were forked to obtain data, and send it back to the server. In all probability, each client will have an array of data structures, which differ from those used by any other client. Rather than impose on the server the need to maintain temporary arrays of dozens of different structures, we will consider all incoming data to be a binary byte stream, and capture it in an array of unsigned char.

Protocols

The above example of client and server stops at the point where the client successfully connects to the server. At that point, the server sends a thread to execute the handler function xserve(), and the client executes the xclient() function. What happens next, is an exchange of information, between the client and server, according to some rules, which we need to establish, called the protocol.
This should not be misconstrued, as being a re-invention of X.25, or HDLC. It is just a set of rules, governing who says what, and when.
In any exchange of information, someone has to begin the conversation, and it is purely arbitrary, whether it is the client, or the server.
For our purposes, we will decide that it is convenient for the conversation to begin, with the client identifying itself to the server and, since we are going to transfer data, we will also need to pass back information on how much of it the server should expect.
This means that the first action in the xserve() function should be

```
if(read(sock, sresp, sizeof(sresp)) == -1){
        perror("Read type and size");
        close(sock);
        th--;
        pthread_exit(NULL);
    } else {
        printf("Server read:>%s<\n", sresp);
    }
```

where sock is a socket descriptor, and sresp is an array of unsigned char.

The server starts a read() call, which blocks until a response arrives, therefore, the first action in the xclient() function must be

```
sprintf(cmesg, "%d %d", which, size);            /* data type and length */

    if(write(c, cmesg, sizeof(cmesg)) == -1){
        printf("Client %d: Fatal: Unable to write to server\n", getpid());
        perror("Write");
        return(-1);
    }
```

Note, that we send two integers from the client to the server. When the server reads the response, it will have to interpret it, and it is considerably easier to make sense of an array of two integers, than to parse ASCII strings.