Writing a C-based Client/Server

Working the Socket

Consider for a moment having the massive power of different computers all simultaneously trying to compute a problem for you -- and still being legal! With the commonplace interconnectivity the network brings, you can do just that. All you need is a login and a compiler -- and a few system calls.

Network programming extends the ability to solve computational problems. Nearly all network programs use sockets to provide connection points between the source and destination. You can think of sockets as an in/out box for messages between tasks on different computers. All you need is an host address and port, and you can send and receive messages from anywhere on your network.

For example, SETI@Home uses others' computers to process the block of data with complex computations. This network of free processing time essentially creates a massively parallel multiprocessor. In just a one year (5/99 - 2/00), SETI had processed over 165,000 years of data. All this is possible because of sockets.

Connecting the Client

You can program your own multiprocessing application with the tools on your computer. Most UNIX operating systems include a C compiler, and you can try out the coding examples directly from this article or the Linux Socket website (www.linuxsocket.org). Programs that connect to the network are really straightforward, with the understanding that some of the system calls are a little hard to find. Use the "Linux Socket Programming" (SAMS Publishing) book as your guide to find all the needed calls, tips, and tricks.

Sockets always have two ends: the sender and the receiver. And, all messages eventually boil down to individual message blocks (or *frames*) on the physical network. Typically, the computer that initiates communication is client; and the server accepts the message. The client opens a connection to the server, following a syntax (e.g., MIME) and protocol (e.g., HTTP).

But before the client can connect to the server, the client has to know the address and port of the server. No computer on the network really uses the hostname. Instead, all connections use the host's IP address, and the client is responsible for converting the hostname to the IP address. The user that runs your program may supply a hostname; you can get the IP address from the operating system's Domain Name Services (DNS) with the `gethostbyname()` library call:

```
#include <netdb.h>
struct hostent* host;
host = gethostbyname("lwn.net");
printf("IP address = %s\n", inet_ntoa(host->h_addr_list[0]));
```

If successful, `host` points to a structure that contains the network byte-order address.

Writing a C-based Client/Server

The next piece of information you need is the port number. The port is an agreed upon connection point between the client and server. You can use any port you like -- except zero "#0", and you must have root privilege to use port #1 through #1023. All computers connected to a network use a published list of ports for standard services. You can find this list in /etc/services. For example, the standard port for HTTP is 80, and TELNET is 23.

You can use this file for getting the exact port, or you can select a port directly. Many times, if you want to use a standard service, remembering the number is a pain and cause compatibility problems. The `getprotobyname()` library call can convert the name into the port:

```
#include <netdb.h>
struct protoent *proto;
proto = getprotobyname("http");
printf("%s: port=%d\n", proto->p_name, ntohs(proto->p_proto));
```

Again, the call converts the port number (`p_proto`) into the network byte-order for you. Once you have the address and port number you can connect to the server.

To connect to the server, the program needs a socket. The socket may seem very primitive is its design. However, it hooks right into the file I/O subsystem, making it a very powerful primitive. After creating a socket and connecting to the server, you can treat it like a regular file.

The program creates a socket with the `socket()` system call:

```
#include <sys/socket.h>
#include <sys/types.h>
#include <resolv.h>

int sd;
sd = socket(PF_INET, SOCK_STREAM, 0);  /* create socket */
```

It's that simple. The constants `PF_INET` and `SOCK_STREAM` mean "TCP/IP" network and "TCP" protocol. The Linux operating system offers you many different types of networks and protocols. This is just one combination. The `socket()` system call only creates a connection point: To really start the flow of data, you need to connect to the server:

```
#include <sys/socket.h>
#include <sys/types.h>
#include <resolv.h>

struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));         /* create & zero struct */
addr.sin_family = AF_INET;        /* select internet protocol */
addr.sin_port = proto.p_proto;              /* set the port # */
addr.sin_addr.s_addr = host->h_addr_list[0];  /* set the addr */
connect(sd, &addr, sizeof(addr));                 /* connect! */
```

NOTE: If any of the calls yield an error you need to stop and check errno. Network programming is very prone to error and unreliability. Following good programming practices can save you a lot of long debugging time.

Writing a C-based Client/Server

After you have connected, you can begin the dialog. Since sockets hang off of the file I/O subsystem, you can use the standard `read()`/`write()` system calls. Or, you can use specialized `recv()`/`send()` system calls. The example, below, converts the socket connection into a `FILE` stream, and works the connection from there:

```
char s[200];
FILE *fp;
fp = fdopen(sd, "r+");                    /* convert into stream */
fprintf(fp, "GET / HTTP/1.0\n\n");            /* send request */
fflush(fp);                                /* ensure it got out */
while ( fgets(s, sizeof(s), fp) != 0 )    /* while not EOF ...*/
  fputs(s, stdout);                        /*... print the data */
fclose(fp);
```

HTTP is a very simplified protocol. More complicated ones may require logins, encryption, more interaction, etc. When defining your own SETI@home -style of network program, you have full flexibility to binary (sending straight data structures) or whatever you need.

NOTE: The only limitation is that you cannot send pointers. Once the pointer leaves the program that owns it, it is meaningless (even going to the same host in loopback).

Establishing the Server

The client uses the server's protocol to request information. When the client's request for connection arrives, the server is ready to accept the connection and begin processing.

The server's algorithm looks a lot like the client's --*with a few modifications*. When you write a server program, you don't need some of the client's library and system calls (like `gethostbyname()` or `connect()`). The server adds a few new calls that turns the socket into a listening server socket.

After creating the socket (identical to the client's, above), you need to tell it what port to listen on. The program does this with `bind()`:

```
#include <sys/socket.h>
#include <sys/types.h>
#include <resolv.h>

struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = proto->p_proto;
addr.sin_addr.s_addr = INADDR_ANY;            /* any interface */
bind(server_sd, &addr,         /* bind port/address to socket */
    sizeof(addr));
```

Except for the `INADDR_ANY` and `bind()`, this code fragment looks a lot like the `connect()` code fragment, above. This call tells the operating system that you want a specific port number. Without this call, the OS assigns any available port number that's available from a pool. By assigning a specific number you are, in effect, publishing that port for use.

The `bind()` system call allows you to isolate the service to one or all network interfaces. Each active network interface (NIC) owns at least one IP address, and each host on the network has at

least one NIC. Placing `INADDR_ANY` in `addr.sin_addr`, tells the OS that you want to accept connections from all possible IP addresses on the host. Otherwise, you can select just one IP address to offer a service. This is very effective if your host is acting as a firewall, where one side of the firewall should not have access to the service.

After binding the address, the TCP connection expects some client to request a connection. In order to get the socket to accept that connection you must convert the socket into a listening socket:

```
listen(server_sd, 10);   /* make into listener with 10 slots */
```

The `listen()` system call changes attributes of the socket. First it makes it into a listening socket (you cannot use the socket for data transmission anymore), and second, it creates a waiting queue with the depth you specify. In the example, above, up to ten pending connections can wait while the server serves the current active connection.

Now that the server has a listening socket, it can wait for a connection. When the client makes the request to establish a dialog. The server catches that request in an accept call:

```
int client_sd;
FILE *fp;
char s[200];
while (1)                      /* process all incoming clients */
{
  client_sd = accept(server_sd, 0, 0);   /* accept connection */
  fp = fdopen(client_sd, "r+");          /* convert into FILE* */
     /*--- Process client's requests ---*/
  while (fgets(s, sizeof(s), fp) != 0  &&
      strcmp(s, "bye\n") != 0)      /* proc client's requests */
  {
    printf("msg: %s", s);                    /* display message */
    fputs(s, strlen(s), fp);                 /* echo it back */
  }
  fclose(fp);                      /* close the client's channel */
}
```

This code fragment reads each line from the client and displays in on the console and echoes it to the client. When the client connects, the listening socket creates a new data socket through the `accept()` system call. If you recall, the program cannot use the listening socket for data transfer. The `accept()` system call provides the needed I/O channel. Of course, the new socket needs to be closed when you are done with it.

This "echo server" is really the best way to start programming sockets. It demonstrates how to connect and perform I/O. From this point, you simply add your own data.

Threading Your Server

The server, above, can only handle one connection at a time, because the current task is always busy processing the longest waiting client. Processing only one client at a time is really an inefficient use of the server's timeslice, because the program is usually blocked on I/O. One way to recover the lost time is to split off processes or threads to handle each incoming connection.

Writing a C-based Client/Server

It's easier to fork off separate tasks, but that's a little heavy handed. Instead, you can use POSIX threads.

Threads are special processes that share as much data space with the parent as possible. This reduces the time it takes to flush and reload the caches and memory managers. Most Linux distributions include pThreads which is a POSIX 1c-compliant library.

You use two main calls to create and use simplified threads: `pthread_create()` and `pthread_detach()`. These two calls allow you to create and "let go" of a child thread. If you don't let the thread go, you have to track it like any other task, waiting for the child to terminate (see *Linux Socket Programming* for a complete discussion of multitasking). Consider an example of how to add threading to the earlier code snippet:

```
    /*--- Process client's requests ---*/
#include <pthread.h>
pthread_t child;
pthread_create(&child, 0, servlet, fp);        /* start thread */
pthread_detach(child);                      /* don't track it */
```

This code creates the child thread, telling the child to run `servlet()` with `fp` as its parameter. From the point that program calls `pthread_create()`, you must consider that the child is running. So, both the child and the parent can access any global or shared data. This is very powerful and very dangerous. If you are not careful, both tasks can corrupt each other's data regions.

The function `servlet()` is something you need to create, and all threads in the pThread library requires a particular function prototype:

```
void *servlet(void *arg)                          /* servlet thread */
{ FILE *fp = (FILE*)arg;                /* get & convert the data */

  while (fgets(s, sizeof(s), fp) != 0  &&
      strcmp(s, "bye\n") != 0)      /* proc client's requests */
  {
    printf("msg: %s", s);                     /* display message */
    fputs(s, strlen(s), fp);                  /* echo it back */
  }
  fclose(fp);                    /* close the client's channel */
  return 0;                          /* terminate the thread */
}
```

Because the parent detaches the thread, you only have to worry about firing off as many threads as the clients need (and the host can support). Likewise, you can multithread your clients, letting do other things while waiting for the network.

Summary

To compile the threaded example, you need to add `-lpthreads` as the **last** argument to the cc compile & link command.

Writing a C-based Client/Server

The programs in this article show you how to write a socket client and server. The client connects to the server, while the server waits for connection requests. You can add more power and flexibility to your programs by integrating multitasking, specifically threads.

You can get source code from the *Linux Socket Programming* website, http://www.cs.utah.edu/~swalton.