

Lab 7

7 May 2009

In this lab, you'll write a rudimentary chat program. There are actually two main parts: the part that sits and waits for connections (which corresponds to when you log in to iChat or AIM or whatever); and the part that actually initiates a specific connection (which corresponds to when you double-click on someone in your buddy list). We'll write them in that order, and call them `chatlogin` and `chatconnect` respectively.

1 Setting up `chatlogin`

First, you'll write a receiver, a very short program that accepts an incoming connection and then relays to the terminal everything coming in over that connection. To test it, you'll use my version of `chatconnect`, in the course directory.

For the general setup of this program, you need a `main` function that takes one argument, a number between 1000 and 32767. This will be the “port number”, described below. Then, you'll follow a few steps to create a connection; and once you've done so, you'll have two file descriptors (which act just like the file descriptors we've seen in plain old OS code) that let you read and write from the network connection—anything you read from the inbound file descriptor will be printed to `stdout`, and anything you read from `stdin` will just be written to the outgoing file descriptor.

There are four steps to establishing a connection from this end.

Making a socket

The first thing you need to do is establish a socket. A socket is where the metaphorical jumper cable is plugged in to make a connection. It is created with the `socket` system call. The *domain* of a socket is (roughly) the Network protocol that will operate the socket—in this case, IP, the Internet protocol, corresponding to the constant `AF_INET`. The *type* and *protocol* of the socket specify the sort of connection that will be made at the Transport level—in this case, TCP, a duplex (meaning two-way) connection-based byte stream, corresponding to the constant `SOCK_STREAM` (with protocol 0).

The return value, if it's not `-1`, is a file descriptor. Hang on to it.

Opening a port

Having created a socket, if you want to be able to receive any connections, you need to provide a label by which the outside world can find that socket. This is done by binding the socket to a “port number” of your own devising, using the `bind` system call to link the file descriptor for the socket to the desired port info.

Specifically, this port information is passed in the second argument to `bind`. How this works is poorly explained on the `bind` manpage, and slightly better on the `ip` manpage and the two `inet` manpages (the one in section 4 is better), but there's still a bit of a C hack going on. The prototype says that that argument is a “`struct sockaddr *`”. There are a number of different types of socket addresses, and in the case of TCP/IP the info is encoded in a `sockaddr_in` structure. In an OO language we'd just subclass the one from the other and be done with it, but that didn't, and doesn't, work in C. They've pulled a few tricks to make all the various socket address structure types be the same size, but you still have to manually cast it yourself. That is, you'll have a value of type `struct sockaddr_in` which you will manipulate, and when you pass it in to `bind`, you'll take its address (using the `&` operator) and cast that to a `(struct sockaddr *)`.

So here's what you need to do. Declare a variable of type `struct sockaddr_in`. Set its `sin_family` field to `AF_INET` (even though this seems redundant). Set the `sin_port` to the port number you read from the command line. Set its `sin_addr.s_addr` to the special value `INADDR_ANY` to indicate that you don't know or don't care what this computer's IP address is. Then, when you pass the address of this variable to `bind`, cast it to make the compiler happy. (The third argument to `bind` is just the `sizeof` one of these structs.)

That still won't quite work, but it'll be closer. Get the above to compile before you go any further in the lab.

The other monkey wrench in the situation is that for numbers larger than one byte, there is an endian problem. All numbers transmitted as part of the TCP/IP protocol need to be in network byte order (which is the reverse of x86 byte order). To convert between them, you can use `htons`, `htonl`, `ntohs`, and `ntohl` as appropriate, which convert back and forth between network and host byte order, for both short (16-bit) and long (32-bit) numbers. The

main ones to watch out for are the address, which is a long, and the port, which is a short. Here, you'll need to wrap the port number with a call to `htons`, and the pseudo-address `INADDR_ANY` with a call to `htonl`.

Remember to check the return value in case something goes wrong!

Open for business

So you've got a socket and the outside world can see it. Now you need to tell the TCP protocol that the port is open and awaiting connections; and you need to specify how many incoming connections are allowed to queue up. This is done with the `listen` call.

Specifically, you have to use `listen` to tell the system how many potential `chatconnect` clients will be able to queue up and wait to talk to you on a particular socket. (Before you call it, even though the socket and port might exist, zero clients can connect to it, so you're not really accepting connections yet!) For this program, you only need to queue up one at a time.

Get a connection

Finally, at long last, you get to settle in and actually get incoming connections. This is done with the `accept` call. In most server programs this call is the one that would be inside a loop, but for now that's not necessary.

Two of the arguments demand explanation. The second is another of those `struct sockaddr_in` structures, but in this case rather than passing the info into the function the argument is used as a way to pass info back to you, specifically, the address of whoever is connecting. You'll ignore this for now but you still need to give it an allocated `struct sockaddr_in` to put the info in. The third argument is much like the third argument to `bind`, except that here you're passing the address because the call to `accept` could modify it if the address it returns is shorter. (It won't, here, but again, you have to pass the function what it's expecting.)

What it returns is a file descriptor for the accepted connection. This is the fd that you will actually read from and write to while connected.

Put it together

All of that should give you a little setup and four function calls. At this time, you can just use `read` and `write` to communicate with your client; try it by running your own program with some port number, then run

```
/home/courses/cs226/chatconnect thishost yourport
```

except replacing `thishost` and `yourport` appropriately. My client assumes that the two sides of the conversation will take turns typing single lines.

2 The other direction

Having written the end that receives connections, now you get to turn around and write the side that makes the connections, which should operate more or less identically to the one I put in the course directory. It will accept a hostname and a port on the command line, connect to that port of that machine, and start sending bytes. At end of file (i.e. the user hits Ctrl-D), the connection is terminated. You should finish the receiver before starting this part, not just because you will use it to test, but also because the sender will use similar system calls.

Making a socket

Once again, the first thing you need to do is establish a socket. A socket for connecting is made in the exact same way as a socket for receiving connections.

Finding the host

Since you're given a hostname, the next thing you need to do is look up its address. This is done with the `gethostbyname` system call. Unlike a lot of similar calls, this one actually allocates its own data structure and returns a pointer to it (so you just need to make a pointer variable, and you don't need to pass in an address to already-allocated memory). Assuming it doesn't return an error and the hostname has at least one associated address, you can get this address out of the `h_addr_list` in the `struct hostent` that it

returns. Each address in this list is stored as a four-byte string, with each byte representing one octet of the IP address.

Connecting

Once you have an address (which you will need to copy into a `sockaddr` data structure), you will need to use the `connect` call to actually make the connection. (This is what strings the metaphorical jumper cable between the two sockets.) The call is similar to that made in the other half to `accept`.

Remember to check the return value for errors!

Communicating

The circuit is made, and now you can send things along it. Again, use the `read` and `write` system calls, this time with the socket returned by the `socket` system call, which is a readable/writable file descriptor.

When you're done, `close` the socket and exit the program.