We have chosen to put our server on port 2000. Ports 0-1023 are reserved for the so-called **well-known ports** corresponding to standard TCP/IP services. You can see a list of well-known ports in the UNIX file **/etc/services**.

Line 24: We will use arrays for our **read()** and **write()** system calls. They need to be large enough for the data we send. In this case, 1000 characters will be sufficient. Note carefully, though, that this limit is NOT there for the purpose of keeping our packets small; we can send as long a message as we like, with a single call to **write()**. Remember, TCP will break up long messages into shorter ones for us, without us even seeing it, so we do not have to worry about it.

Lines 36-39:

We open a socket, using the **socket()** system call. The first parameter indicates whether this is an Internet socket, versus a socket purely local to this machine (a "UNIX socket"), not going out onto the Internet; the Internet case here is designated by AF_INET. The second parameter indicates the service, i.e. TCP, UDP or others; it is TCP in this case, designated by SOCK_STREAM.[21] We have defaulted the third parameter here (and will not worry about what other possibilities there are for it).

The function's return value, assigned to **SD** here, is a **socket descriptor**, quite analogous to a file descriptor.

Lines 41-54:

We are building up a data structure named Addr ("address") which we will use in line 51. Its type, **sockaddr_in** (line 31) comes from the #include file and is a standard socket type. Clearly, it is a complex type, with many fields, and we will not go into the details here. See the man pages if you are interested for more information.

One thing to beware of is that for the system call connect() and many other socket functions, you will see that the man pages say you should use a struct of type **sockaddr**, as opposed to the **sockaddr_in** type we have used here. Yet the **sockaddr** type is just meant as a dummy, to be replaced by another struct type which is specific to the network protocol being used. In our case, we are using TCP/IP, so we choose the **sockaddr_in** type, where "in" stands for "Internet." There are also types such as **sockaddr_ns** for the Xerox Networks Systems protocol, though of course TCP/IP has become virtually ubiquitous.

Having already opened a socket, we have to connect it to a specific port at a specific machine. Recall that the user specifies an Internet host name, such as **garnacha.engr.edu**. But we need the machine's numerical Internet address. This is obtained on line 46. The return value is a pointer to another standard data structure.

In line 47, we call **memcopy()**, a system call which copies strings from one part of memory to another, in this case from various fields of the struct pointed to by **HostPtr** to **Addr**. A given host may have several addresses; here we are not bothering to check for that, but simply using the first address, contained in **h_addr_list[0]**. (Here 'h' stands for "host.") The **h_length** field gives the length of the address.

In line 51, we now connect the socket to the destination host. (It is here that the negotiation between source and destination hosts will occur, as to packet sizes, and so on.)

Note that the port number we have specified is for the port at the server, not the client. There is a hidden port number for the client here, which we will discuss later.

Line 57:

[21] There are also various others, such as SOCK_RAW for raw sockets.

Here we write either "w" or "ps" to the destination host, depending on what the user requested. Note that the function **write()** is identical to the one used for low-level file access, except that we have as the first parameter a socket descriptor instead of a file descriptor.

Lines 60-61:

On line 60 we read the message sent by the destination host, which will be the output of that host's running either the **w** or **ps** command. On line 61 we then write that message to the user's screen. (For convenience, we do so again using **write()**, making use of the fact that the standard output has file descriptor 1, though of course could have used **printf()**.)

Note carefully that if we had been expecting more voluminous data from the server than is the case here, we may have had to do repeated calls to **read()**.

In addition to **read()** and **write()**, we may also access sockets via other very similar system calls **recv()**, **send()**, **recvfrom()** and sendto(). (And in non-UNIX environments, we *must* use these, since those OSs do not treat socket and file I/O the same.)

Now let's take a look at the server code.

Line 68:

Here the server creates a socket.

Lines 77-85:

The system function **bind()** is called in line 80. This associates the socket with a particular port and with a particular IP address on the local machine, i.e. the machine on which the program which calls **bind()** is running, in this case the server machine. Recall that a machine may have multiple IP addresses, either because it has several NICs or because it may have multiple IP addresses assigned to the same NIC.

For example, suppose one of the NICs at the server machine corresponds to a local private intranet. Then we could specify that particular address in our call to **bind()**, which would enable our making the program available only on this intranet.

Or, suppose we are running an Internet service provider (ISP). We may be hosting many different customer Web sites, all with different names (**www.acmegroceries.com**, **www.flatearthsociety.org**, etc.), each with a different IP address. We want a given server, say the one for Acme Groceries, to respond only to clients accessing the **www.acmegroceries.com**, so our call to **bind()** would specify that address.

In our case here, we wish this server to be accessible from via of its IP addresses, which we specify in line 79 by using the constant INADDR_ANY.

Note that we did not have a call to **bind()** in our client code. We could have had one if we wanted the client to access this port only through a particular one of the IP addresses of the client machine. In the intranet example above, for instance, if the information to be exchanged by the client and server really needs full security, it might be safer to make sure the client does not accidentally send its information outside the intranet (say because a routing table becomes corrupted).[22]

To summarize, in a server program, calling **bind()** associates with the given socket the port number and IP address that "phone calls" to this socket will be allowed on.

[22] If we do call **bind()** in a client, this must be done before calling **connect()**.

Normally we do not need to call **bind()** in a client program. Yet the client still needs a port number and IP address to use in accepting messages which come from the server. If we do not call **bind()** in the client, then calling **connect()** in the client will cause the OS to assign to the client a port, called an **ephemeral port**, as well as an IP address (in the case that the client has more than one IP address).[23]

Note that nowhere in the client or server code above do we see any mention of the client's ephemeral port number. (WPSPORT is the number of a port at the server, not at the client.) But the OS at the client will indeed notify the server regarding the identity of the ephemeral port (when the client calls **connect()**),

Normally we do not need to know which ephemeral port has been assigned at a client, but if we need it then we can get it by calling **getsockname()** in the client after calling **connect()**.

Line 88:

Be calling the **listen()** function, we are notifying the OS that this program will be a server, not a client. We also notify the OS as to how many incoming calls (in the form of clients invoking the **connect()** function) will be allowed to be pending at one time, in this case five. If a call arrives when the queue is full, the call will be discarded (so it is best when writing the client to put the call to **connect()** in a loop, looping until **connect()** succeeds).

Lines 93-115:

Here we loop indefinitely, continuing to process calls one at a time. The **accept()** function (line 95) accepts a pending call, returning a socket which we will use to exchange messages to the client. By making it a separate socket, we can have several client sessions active simultaneously, though we are not doing so here. The original socket is then called a **listening** socket, whose job is only to listen for connection requests from clients, rather than for actual information exchange with clients. The sockets created by **accept()**, which do the actual message exchange with the clients, are called **connected sockets**.

We read the client's command, "s" or "q", on line 108, and then respond to the client on line 114.

Lines 35-56:

There are two systems calls you might not be familiar with here. The function **system()** (lines 50, 52 and 54) actually submits a shell-level command. Note that we are saving the response in a file, **tmp.client**. The file is later removed, using the **unlink()** function (line 42), which is the system-call level analog of the shell-level **rm** command.

5.1.2 Who Shall I Say Is Calling?

There are many other TCP/IP functions available. For example, after line 95 in the server code, we could call **getpeername()** if we needed to know the Internet address of the client. To use this function, declare a variable, say **s**, of type **sockaddr_in** and initialize its **sin_family** field to AF_INET. Also, declare a variable, say **i**, of type **int** and initialize it to **sizeof(sockaddr_in)**. Then the call to **getpeername()** will have as its arguments to the socket descriptor (**ClntDescriptor** in our example here), **&s** and **&i**. To then get the address as a character string in "Internet dot" form, call the function **inet_ntoa()** on **s.sin_addr**.

That would give us the numerical Internet address, and if we needed the alphabetic name, we could get this

[23] This is for TCP. In the case of UDP, where one normally does not call **connect()**, this function is performed by the call to **sendto()**.

5.3 Nonblocking I/O

In many applications a server has sockets open to several clients at once. In this case, the server needs a mechanism for determining which sockets have data waiting and which do not. One way to handle this is to make the sockets **nonblocking**, which means that a call to **read()** will not wait until data is ready. If data is ready at that socket, the call to **read()** will read that data, but if not, the call immediately returns, with a value of -1. You code can then repeatedly poll all sockets, testing for input data at each one, and reading that data if it is there. Here is an example of code to make a socket nonblocking:

```
Flag = 1;
ioctl(S,FIONBIO,&Flag);
```

Here **S** was the return value from a call to **socket()**. You will need the proper include-files; check the **man** page for **ioctl()**.

A much more flexible and sophisticated tool for dealing with multiple sockets is the **select()** function. A newer such tool is **poll()**.

5.4 Debugging Client/Server Programs

As a quick check, you can first try to use **telnet** to check whether the server has called **bind()**, **listen()** and **accept()** properly. Though you are probably accustomed to using **telnet** simply as a means of remote login, it also can be used to communicate with servers at specified ports. What **telnet** does is open a connection to a given host at a given port; whatever bytes the user types will be sent to that port, and whatever bytes the port sends will appear on the user's screen.

This could be used to help our debugging process. If for example we have a server on **pc8.cs.ucdavis.edu**, running on port 1088, we could type

```
telnet pc8.cs.ucdavis.edu 1088
```

If we get a response here but not from our own client program, the latter may have an error in **connect()** or whatever, such as misspecifying the server's IP address or port. (On the other hand, if we get no response, it may also be due to the system configuration not allowing **telnet** access to that port.)

In general, debugging a server/client pair, using a debugging tool **(which you should do when debugging any program)** will be a bit more difficult, because you will need to invoke the tool once for the server and once for the client. So, even though I typically use a GUI to **gdb**, such as **ddd**, for debugging a server/client pair I sometimes use just the plain-text **gdb**, since my screen would not conveniently fit two GUIs at the same time. Or, I might just use the debugging tool on the client while running the server without a debugging tool, or vice versa.

You may find a tool such as **strace**, available on many UNIX systems (and also similar programs such as **ktrace**, **truss**, etc.) to be useful. It will print out each system call made by a program, and the result of each call. In our case here, that means calls to **accept()**, **connect()**, etc. Since the ouptut of **strace** may be voluminous, you may wish to pipe its **stderr** output through **more**, say as

```
strace application_program_name application_arguments |& more
```