Lecture 3: Socket Programming

Announcements:

1. Project 1 is being handed out today in class. Due Sept 23.
2. Please read lecture notes as well as your text book. Class will often go too fast to fully digest everything. When you read the notes, you should cover up the answers.

This entire lecture will be on Socket Programming.

1 Motivation

The intent of Project 1 is to give you practice with using network (the socket interface), whereas later projects will give you practice with building a network.

Project 1 is intended to be easy. Justin Weisz (jweisz@andrew) is the lead TA on this project. The project requires you to build a Web server that handles static requests under persistent connections. You will also be asked to implement secure connections using SSL. The project can be divided into 3 parts:

1. Getting the socket programming right.
2. Learning and using the HTTP protocol which is used by web servers.
3. Implementing secure connections.

We highly recommend that you first do the first component and get it totally right before you move to the remaining components. In this lecture I will cover everything you need for socket programming (1st component).

The goal: Process on one machine wants to send/receive messages from process on a different machine.

Process-to-process communication is the job of the transport layer protocol. The application doesn't need to know how this is done. The application simply uses the *interface* that the transport layer provides to it to send messages. This interface is referred to the **Socket API** (API stands for Application Programmer's Interface).

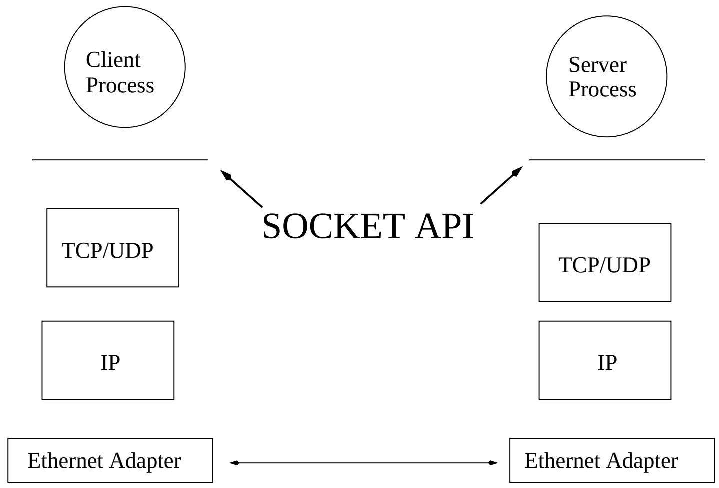

Figure 1: *Client and Server processes exchange messages over the network through a common Socket API.*

Different transport protocols provide somewhat different Socket APIs.

We will first describe the TCP Socket API and then explain the UDP Socket API.

2 Necessary background about Transport Protocols

I will proceed in a non-conventional order. Rather than just throwing the interface calls at you, I'll start by explaining the transport layer briefly. This will make very clear exactly what kind of information the transport layer needs from the application, which helps motivate why the interface looks the way it does.

The primary function of transport layer is the **multiplexing/demultiplexing of processes**. How is this done? We need a way of **identifying** the source process on the source machine and the destination process on the destination machine. This is done via **ports**. An IP address like 128.2.11.206 uniquely identifies a host (in this case unix6.andrew.cmu.edu). The IP address however doesn't specify a *process* running on the host. Each host has many ports associated with it. Note these ports are not physical entities, but rather abstractions. Each port may be thought of as a **logical doorway** to the host. When a message arrives at a host, it enters at one of several logical doorways. Behind each doorway is a process ready to read the message and understand the message.

For example, at every host at port 80 there sits a Web server process, which understands the HTTP language. Other examples: at port 79 sits a finger process; at port 23 sits a telnet process which understands the telnet protocol; at port 21 sits an ftp process which understands the ftp protocol, etc.. Check out

/etc/services

for a long list. Port numbers 0 – 1023 are used for **well-known ports**, whereas port number 1024 – 65535 are either private, or may be dynamically assigned as needed.

When a browser sends off an HTTP request to a server, it must know to direct the request to port 80 on the server machine – so that the request can be appropriately interpreted.

Given the above description, what exactly does the transport layer do wrt multiplexing/demultiplexing of processes?

Guess: The transport layer takes a segment from the application process at the source and encapsulates the segment with the port number of the destination host. It then hands the segment to IP which adds on the IP address of the destination host. The packet is then sent to the appropriate destination host, and finally to the transport layer at that destination host, which sends the segment to the appropriate port at that destination, identifying the correct target process.

Question: Are there any problems with the above description?

Hint: Suppose that the destination host and port refer to a Web server process, which handles many clients ...

Answer: Yes, There's a problem. The server might receive requests from several different client processes. How does the server know which source sent which request, so it can reply back to it?

Full Answer: The server needs to also be told client's IP address and client's port number, so the server process can communicate back to the right client process. As we'll see later in the course, the client process' port number is included in the transport header. The client's IP address is included in the IP (network layer) header.

A particular **connection between two processes** is specified by the following 4-tuple:

(source port, source address, destination port, destination address)

Some people prefer to use a 5-tuple, where the 5th entry is the name of the particular transport protocol maintaining the connection.

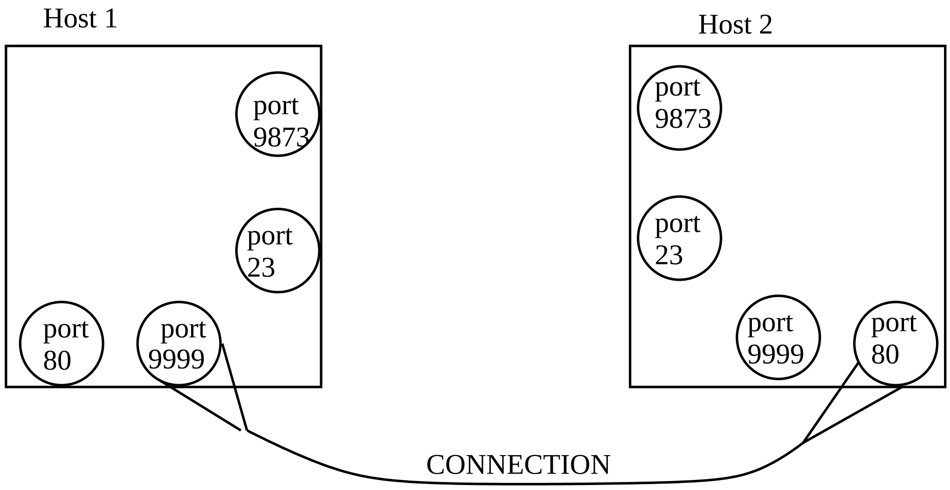

Figure 2: *Depicting a connection between two processes.*

The word **connection** however means *a lot more* than just the 4-tuple or 5-tuple above. Recall that *all* transport protocols are required to do multiplexing/demultiplexing between processes. Some transport layer protocols do little more than this: e.g., **UDP** (User Datagram Protocol). Others have additional functions like ensuring complete reliability: **TCP** (Transmission Control Protocol). In order to establish reliability, TCP must **establish a connection**. UDP on the other hand does *not* establish a connection, although it does do multiplexing/demultiplexing. To understand what it means to establish a connection, consider the following question:

Question: Recall that TCP relies completely on IP, which is an unreliable network protocol. How then does TCP establish reliability?

Answer: TCP must maintain state at the two endhosts. Every time host 1 sends a packet to host 2, it can't be sure that host 2 got it until it receives an **acknowledgement (ACK)** from host 2. The first data packet can't be sent until a **3-way handshake** has taken place between the 2 hosts. This 3-way handshake will be described in detail when we cover TCP. The purpose of the 3-way handshake is to give both hosts a chance to prepare the additional state/buffers that they need to implement reliable transport. The completion of the 3-way handshake is often referred to as **connection establishment**. Once again, this includes both the exchange of info about source ports and destination ports and also the establishment of state info, such as what the starting packet sequence number will be. Note that there is *no physical connection*; this is all just a logical connection, consisting of state information at the endhosts only. We say TCP is a **connection-oriented** protocol, whereas UDP is a **connection-less** protocol.

3 What is a Socket?

Now that we understand the role of the transport layer, we're ready to look at the interface that the transport layer provides to the application layer. Part of this interface is the notion of a **socket**.

Processes need a way of accessing connections, so they can write/read data to/from that connection. They do this via a "connection-descriptor", commonly called a **socket-descriptor**. A socket-descriptor is an integer, and is analogous to a file-descriptor, which processes use for accessing files. A socket can be thought of as one handle to the connection. The socket-descriptor is the name of this handle.

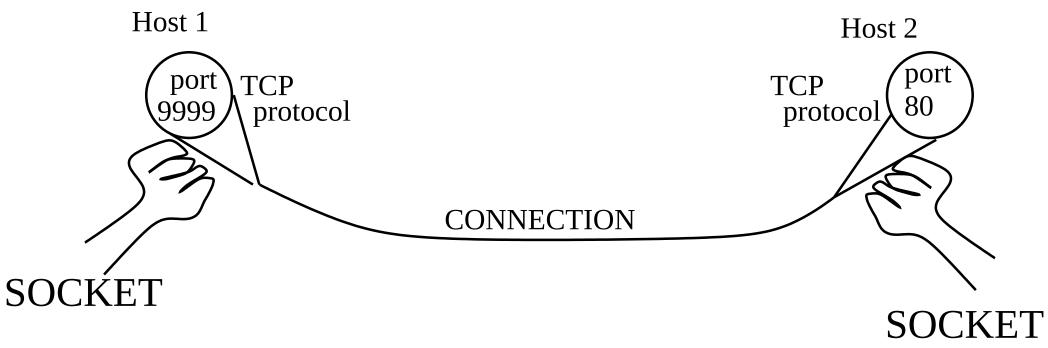

Figure 3: *Each hand represents a socket.*

A socket can be thought of as made up of several components:

1. The name of the protocol family stack (e.g., AF_INET refers to the Internet Protocol Stack which we've been studying).
2. The name of the transport protocol (e.g., TCP).
3. The port number of the process at the host.
4. The Internet Address (IP address) of the host.

```
struct in_addr{
  u_long s_addr;
};

struct sockaddr_in {
  u_char sin_family; /* Address Family */
  u_short sin_port; /* Port number */
  struct in_addr sin_addr; /* IP address */
  char sin_zero[8]; /* unused */
};
```

The structure `in_addr` represents a 32-bit IP address. This address is network byte ordered.

The structure `sockaddr_in` represents a socket address.

[Some details: The `u` in `u_long` stands for unsigned. The address family is an 8-bit integer.]

4 Sockets over TCP

Important: For the purpose of clarity, I have removed all error-checking from the code I'm showing you. Your code should include FULL checking for errors if you want full credit. To do this properly, you will need to look up each command in the man pages.

TCP server program

Below we describe a simple TCP server program, which establishes a connection with a client, reads the client's request (written in English, not HTTP) and replies.

```
int fd, newfd, nbytes, nbytes2;
char buf[512], response[512];
struct sockaddr_in srv;

fd = socket(AF_INET, SOCK_STREAM, 0);
```

This first line creates a socket. The application fills in the information about the socket, to be used/interpreted by the transport layer. First it tells us that the protocol family stack for this socket is AF_INET, the Internet Protocol Family. The application also tells the transport layer that it wants to use the SOCK_STREAM transport protocol, which denotes TCP. The socket call returns fd, a socket descriptor for this socket.

Question: What are the other things that must be specified about the socket?

Answer: The Server Host's Internet address and the Port number of the server process.

```
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr(``128.2.15.9'');
bind(fd, (struct sockaddr*) &srv, sizeof(srv));
```

Above the application specifies the port number of the server process to be port 80. The command htons converts between host byte ordering and network byte ordering where the argument is of type short. (Use your web resources to learn about variants of this command, like htonl.) The point is that different hosts order bytes in different directions. The way that the host orders bytes may be reversed from how the network does it. The application specifies the IP address of the server machine to be 128.2.15.9, and the command inet_addr() converts this string into a network-byte-ordered integer. Note, it is much more typical to replace this line with the command:
srv.sin_addr.s_addr = htonl(INADDR_ANY); which is much more flexible and says that the host address of the server machine is any of the server's many possible IP address (server may have several ethernet cards, each with its own IP address). The bind command binds the socket descriptor fd with this new additional information. This ensures that every time the server process does a read(fd), the data that it reads will be only data that has been directed to the server's port number (the operating system takes care of this). At this point the socket is fully described.

The server now has its end of the connection ready to go. It now must sit at this end and listen waiting for a connection request to come along from a client:

```
listen(fd, 5);
newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
```

The listen command is non-blocking. It sets up a listen queue and when connection requests come in from the clients, these requests are put on this listen queue. The 5 indicates that up to 5 connections may be pending accept, before connection requests are simply lost. We will come back to this in more detail soon.

The accept call is blocking. It blocks until there is a connection on the listen queue. As soon as there is a connection on the listen queue, the connection's source IP address and source port get written into the structure `cli` for use by the server process. Furthermore a new socket-descriptor, `newfd` is assigned to the connection.

Question: Why is `newfd` needed?

Answer: `fd` is used for listening. Each new connection needs its own socket on the server end. Server process uses `newfd` for reading/writing to client process.

The server process reads the client's request, and sends it a response.

```
nbytes = read(newfd, buf, sizeof(buf));
sprintf(response, ``Here's your response'');
nbytes2 = write(newfd, response, strlen(response));
```

Question: What's wrong with the above read and write, aside from having left out error-checking?

Answer: Can't assume that read has read whole request, or that write has written whole request. `read()` needs to be in a loop where keep reading until see special end-delimiter. Every application protocol has such an end-delimiter, specifically for this reason. For `HTTP` the end-delimiter is:

```
\r\n\r\n
```

Likewise, `write()` needs to be in a loop where keep writing until you've written all bytes of the response.

The most common mistake made by new socket programmers is forgetting to put reads and writes into a while loop.

Finally, when the server process is done, it must close its socket descriptors.

```
close(fd);
close(newfd);
```

We will soon make this code more complicated and discuss it in more depth. Before we do this however, let's take a look at the client code.

TCP client

Below we describe a simple TCP client program, which establishes a connection with a server and writes a request to that server (written in English, not HTTP).

```
int fd, nbytes, nbytes2;
char buf[512], request[512];
struct sockaddr_in srv;

fd = socket(AF_INET, SOCK_STREAM, 0);
```

This first line creates a socket. The application fills in information about the socket, to be used/interpreted by the transport layer. First it tells us that the protocol family stack for this socket is `AF_INET`, the Internet Protocol Family. The application also tells the transport layer that it wants to use the `SOCK_STREAM` transport protocol, which denotes `TCP`. The socket call returns `fd`, a socket descriptor for this socket.

The client now needs to ask the server to connect it's socket to the server's socket. To do this, the client has to know the address of the server's socket, meaning the server's IP address and port.

```
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr(``128.2.15.9'');
connect(fd, (struct sockaddr*) &srv, sizeof(srv));
```

In the above code the client asks to "make a connection with the server." The `connect` call generates something called a `SYN` packet in TCP. The `connect` will block until the server's TCP protocol sends back a `SYNACK` saying indicating that it is willing to establish this connection. At this point the client's TCP will send back a final `ACK` and the connection will be formed. This is called the **three-way handshake**. Thus, when the `connect` call returns, the connection between the client and server is established. The client can now write to the server and read from the server using the socket descriptor `fd`:

```
sprintf(request, ``Here's my request'');
nbytes2 = write(fd, request, strlen(response));
close(fd);
```

At this point, you should have a lot of questions, because there is a lot that is unclear here:

Question: Recall that the server does an explicit `bind()` to bind its socket to its address. Howcome the client never does a bind? Where does the client's IP address and port get specified? How does the server learn this information, if the client never specifies it?

Answer: The client process never picks a port. A port is chosen for it by TCP. TCP picks a port that is not currently in use on that machine. This is convenient since there's no good way for the client process to know which ports are in use. Since TCP picks the client port, there is no reason for this to be part of the socket API (the client application process doesn't have to get involved). The client's port and its IP address are sent to the server at the time of the `connect()` call, but the client-side application doesn't see this.

Question: Who has to start running first, the client process or the server process, and why?

Answer: The server must run first, because if `listen()` hasn't been called at the time the client is run, the client will SEGFAULT when it gets to the `connect()` call.

Question: At what point is the 3-way handshake complete (the connection established) between the client and the server?

Answer: Many books and teachers imply that this is done at the point where the server does the `accept()` call. That is, the `connect()` is blocked waiting on an `accept()` from the server. This is **FALSE**. Tonight, try out the following experiment: Write a client and a server, as above. Just before the `accept()` call in the server, insert a `sleep(30)` command. This puts the server to sleep for 30 seconds after the `listen()`. You will find that the client is able to send its request and `close()` long before the server ever wakes up. That is, the client does not block waiting for an `accept()`. In fact, when the server wakes up, it will do an `accept()` and a `read()`, and it will find that the client's entire request is right there, although the client is long gone.

Question: What does this experiment tell you? Where is the connection actually established?

Answer: The connection is actually established by the `listen()` call. The `listen()` call is all that's necessary to complete the 3-way handshake.

Question: So then what's the `accept()` call for?

Answer: The accept call takes these open connections off of the listen queue (the listen queue only has room for 5 such connections, in our example). From the perspective of the server process, the purpose of the accept call is to give the server process a handle (socket descriptor) to the new connection.

Question: Why is it a good thing that `listen()` establishes the connection rather than `accept()`?

Answer: `accept()` is blocking. `listen()` isn't.

5 Persistent Connections

In the case of **persistent connections**, the server leaves the TCP connection open after sending responses. Subsequent requests and responses between the same client and server can be sent over the same connection.

Observe that in implementing persistent connections, the server's reads get much more complicated, since the server might potentially end up reading 2 requests within a single `read()`. Warning: watch out for this in your project code!

Under persistent connections, the server doesn't close the connection until the client does a close(). (This generates a client packet with a "Connection: close" header, which indicates to the server that it can close the connection.)

6 Select

In the server code above, we only called `accept()` once. Suppose that the Web server wishes to handle several connections simultaneously (as all Web servers do).

Idea 1: We might try putting a loop around the `accept()` call so that `accept()` is called over and over again.

Question: What's the problem with this?

Answer: The `accept()` call is blocking. While an `accept()` is blocked waiting for a connection request, the server might be missing a chance to `read()`. Likewise the `read()`

call is blocking. If the server calls `read()` and there's nothing there to read, it will just block waiting and nothing else will get done.

Idea 2: Use non-blocking `read()`. Attempt to read from one socket descriptor and then the other, back and forth, until get something.

Question: What's the problem with this?

Answer: Waste a lot of CPU.

Idea 3: Use asynchronous I/O, in which you ask Unix to send you a SIGIO signal whenever some file descriptor changes.

Question: What's the problem with this?

Answer: Difficult to code.

Idea 4: We can spawn a separate thread per connection. When a thread blocks on reading, it goes to sleep, but another thread gets the CPU, so cycles aren't wasted.

This is a good idea, and many Web servers are written this way. Multi-threaded programming has a lot of implementation overhead too, though. Web servers aiming for speed (like *Flash*) often try a single-threaded implementation.

Question: Is there a way to write a server with a single thread that can handle multiple connections (as required by Project 1)?

Answer: Yes! The `select()` call.

The `select()` call allows us to ask whether any of a set of socket descriptors have something to read. This allows us to then call a `read()` on the appropriate socket descriptor without fear of blocking.

Here are the definitions you'll need for `select()`: *Please also check the man pages! They're really good on this.*

```
int select(int maxfds, fd_set *readfds, *writefds, *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set * fds); /* This clears the bit for fd in fds */
FD_ISSET(int fd, fd_set *fds); /* Is the bit for fd turned on in fds?  */
FD_SET(int fd, fd_set *fds); /* turn on the bit for fd in fds */
FD_ZERO(fd_set *fds); /* clear all bits in fds */
```

`maxfds` represents the highest-numbered descriptor in any of the three sets, plus 1. The descriptors which will be tested are $0, 1, 2, \ldots, maxfds - 1$. Initially, `readfds` is a set of socket descriptors which we want to check to see if they have anything available to read. After the `select()` call, `readfds` has set to 1 only those socket descriptors which have something to read. We won't be needing the writefds or exceptfds fields.

The `select()` call blocks until one of the descriptors is ready for I/O, or until the timeout.

Question: So if I want the `select()` call to be non-blocking, I do what?

Answer: Set `timeout` to 0.

Question: How do you think you can use `select()` in a Web server, which has to both listen for new connection requests, and make sure that it's reading from all connections?

```
int fd, next = 0; /* fd is listening descriptor */
int newfd[10]; /* newfd is array of accepted descriptors */
int maxfd; /* This holds the maximum numbered descriptor */
while(1){
  fd_set readfds;
  FD_ZERO(&readfds);
  FD_SET(fd, &readfds);
  /* Fill in ...  use FD_SET to turn on bits for all newfds */
  select(maxfd + 1, &readfds, 0,0,0); /*does anyone have anything for me?*/
  if(FD_ISSET(fd, &readfds)){
    newfd[next++] = accept(fd, &cli, ...);
  }
  /* do the following for each descriptor newfd[i] */
  if(FD_ISSET(newfd[i], &readfds)){
    read(newfd[i], buf, sizeof(buf));
    /* process data */
  }
}
```

Observe that FD_SET is first used to turn on the bits for all the socket descriptors within readfds that we care about. When the `select` call is entered, `readfds` indicates what socket descriptors we want to check out. When the `select` call is exited, the original contents of `readfds` have been erased and `readfds` now has a 1 bit for each of the socket descriptors that have something to be read. Because `readfds` is modified each time that `select` is called, we need to reset the appropriate file descriptors (using FD_ZERO and FD_SET) each time before calling `select` again. Note that we have set a 0 timeout value for select.

7 Sockets over UDP

This section describes a simple UDP server and client program.

UDP server program

```
/* fill in declarations */
fd = socket(AF_INET, SOCK_DGRAM, 0);
```

This first line creates a socket. The only difference here from what we're used to is that the application uses the `SOCK_DGRAM` option, which specifies the UDP protocol.

Next, as usual we bind the socket to a port.

```
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = htonl(``128.2.15.9'');
bind(fd, (struct sockaddr*) & srv, sizeof(srv));
```

Question: Do we now need to `listen()`?

Answer: No, There's no need to maintain a queue of connection requests, because there aren't going to be any connection requests. The client will just send data and the server must read it. Everything comes in on this one socket descriptor `fd`.

```
nbytes = recvfrom(fd, buf, sizeof(buf), 0, (struct sockaddr*) &cli, &cli_len);
```

Question: Why don't we get a new socket descriptor?

Answer: We don't have connections.

Question: Why are we using `recvfrom()` rather than `read`?

Answer: `read()` doesn't give us the client's address and we will need that to send stuff to the client.

UDP client

The UDP client is equally simple.

```
        /* Fill in declarations */
        fd = socket(AF_INET, SOCK_STREAM, 0);



    srv.sin_family = AF_INET;
    srv.sin_port = htons(80);
    srv.sin_addr.s_addr = inet_addr(``128.2.15.9'');




sprintf(request, ``Here's my request'');
nbytes = sendto(fd, request, sizeof(request), 0, (struct sockaddr*) &srv, sizeof(srv));
```

Note, socket descriptor `fd` is not maintaining state here, in the way it was under TCP.

8 Some helpful hints

Here are some include files that I used. Pick whatever you need.

```
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <strings.h>
#include <sys/time.h>
#include <errno.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/uio.h>
```

When you compile your client and server, you'll find that you need to use `-lsocket` and `-lnsl`. Please look these up and understand them.

One last thing: IP addresses are commonly written as strings ("128.2.15.9"), but programs deal with IP addresses as integers.

To convert a string to a numerical IP address which is network-byte-ordered:

```
struct sockaddr_in srv;
srv.sin_addr.s_addr = inet_addr(``128.2.15.9'');
```

To convert an IP address structure to a string:

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
```

9 Examples of Application Programming

In Project 1 you will explore HTTP, the protocol used by web applications to communicate with each other. Some equally interesting protocols are FTP, for transferring files, and SMTP, for sending email. These are a lot of fun to explore. Your textbook, *Computer Networking*, Chpt 2, does a particularly good job on describing these protocols. Remember that you can always define your own secret application protocol and pick a port number over which to communicate with your friends via this protocol.