Tutor in Socket Programming under Linux

by Dr. Yingwu Zhu, January 2006

Table of Contents

1. Internet ………………. 2
2. Client and Server Model ……………….. 7
3. Preparing an Internet Address ……………….. 11
4. The Socket Interface ……………….. 14
5. Writing Clients ……………….. 19
6. Single-Client Servers ……………….. 22
7. Multi-Client Servers ……………….. 25
8. Conclusions ……………….. 30

Note: The purpose of writing this tutor is that I feel there is a need to have easy-to-read yet helpful documents in network programming during teaching at SeattleU. In spite of so many books in network programming, there are few able to allow students, especially undergraduate students, to quickly understand the basic idea in network programming. Therefore, I compiled this tutor. I hope it can serve this purpose. This tutor will be used in upper-level courses such as Concurrent & Distributed Systems, Computer Networks, and Peer-to-Peer Networks. In another tutor, I will also design several interesting projects along the thread of network programming. However, this is the first version, implying many potential typos or errors. I hope my readers --- the students, will give me the feedback to improve the tutor.

1. Internet

(Skip this if you know what Internet is, what protocols it uses, what kind of addresses are used over Internet, etc).

Preview

The Internet is a computer communication network. Every computer connected to the Internet is also known as a "host", so we could say that Internet's role is to allow hosts to talk amongst themselves. I assume you are already familiar with Internet, as a user of programs such as 'Telnet', 'Ftp', 'Irc' and others. Let's first discuss Internet addresses a little, before we talk about the Internet protocols, and various programming aspects regarding network uniformity.

Internet Addresses

(Skip this if you know what IP addresses are and what ports in Internet are).

An Internet address (or an IP address, or an IP number) is a number made of 4 bytes (numbers between 0 and 255), written in what is called 'a dot notation'. For example, "128.0.46.1" is a valid Internet address. Such an address identifies a computer which is connected to the Internet. Note that a computer might have more than one such address, if it has more than one physical connection to the Internet (such as having two Ethernet cards connected).

However, when a normal human being uses Internet, they usually use human-readable (oh, really?) addresses of the form "www.seattleu.edu". A system in the Internet called "The Domain Name System" or DNS for short, is responsible to translate between human-readable addresses and IP addresses. You will not have to know anything about DNS in order to use it in your programs, so do not worry about it now.

OK. We said that IP numbers define a computer. Well, usually there is more than one program that wants to use the network that runs on a given computer. For this purpose, the Internet people made up some extension for an IP address, called a port number. Each communications address is made up of an IP number AND a port number. Port numbers could be any number between 1 and 65535, although for certain reasons, you will use port numbers above 1024, unless you have a superuser privileges on your machine (also stated sometimes as "having root password").

For our purposes, we will use addresses of the form: 114.58.1.6:6072 where the "114.58.1.6" part is the IP number, and the "6072" part is the port number. Remember this for later usage.

Internet protocols

(Skip this if you know what IP, TCP and UDP are).

You probably heard the term "TCP/IP" and wondered, or had some vague idea about what it means. Let's make things clearer:

The Internet is a network. In order to talk on a network, you need some kind of a "language". That language is also called "a protocol". The Internet has many kinds of protocols used to talk on it, in a manner called "layering".

Layering means that instead of defining one protocol that will do everything, which will be very hard to design and implement, the tasks are divided between several protocols, sitting on top of each other.

What does this mean? Think about it as sending letters: you write your letter on a paper, and then put it in an envelope and write the address on it. The postman doesn't care WHAT you wrote in your letter, as long as you wrote the correct address on the envelope.

The same thing is done when layering protocols. If one protocol contains the data sent and knows how to make sure the data is correct, then a lower protocol will contain the address to be used, and will know how to transfer the data to the correct target. The lower protocol does not understand the format of the data in the upper protocol, and the upper protocol doesn't have to know how to actually transfer the data. This way, each protocol becomes much simpler to design and test. Furthermore, If we will want to use the same protocol for writing the data, but send the data on a different network, we will only need to replace the protocol that knows how to transfer the data over the network, not the whole set of protocols.

(By the way, this sort of packing up several protocols on top of each other is called Encapsulation.)

One other important notion about the Internet is that it forms something known as a "packet switching network". This means that every message sent out is divided into small amounts of information, called packets. The different protocols send the data in packets, which might get divided into smaller packets while it travels to the target host, due to "Electrical" limitations of physical networks. The target machine will eventually combine the small packets (also known as fragments) and build the original message again.

The packets are what allows several connections to use the same physical network simultaneously, in a manner transparent to the network users. No machine will take over a line completely, even if it needs to send a large message. Instead, it will send the message in small fragments, allowing other machines to send their packets too.

Let us now name out some of the Internet protocols, and explain briefly each one of them:

IP

IP is the basic protocol used on the Internet. It is responsible to make it possible to transfer packets from one computer to another, given their IP addresses. Most other protocols are placed on top of IP. As a programmer you will usually not use IP directly.

TCP

TCP is the most useful Internet protocol for a programmer. Most networking programs you use (such as 'Telnet', and 'Ftp') are placed on top of TCP. TCP is placed on top of IP, and adds 3 functionalities:

1. The notion of ports (IP supports only Internet addresses, as mentioned above).
2. The notion of a 'safe' protocol, i.e. no errors will be encountered in packets sent using TCP, so no error correcting mechanisms are required in programs using TCP (well, this is almost true, but lets assume so for now).
3. Connection-mode link. After you make a connection between two programs using TCP, you have a steadily open connection on which you can send data without having to specify who you want the data to be sent to. This works very similar to reading or writing on a pipeline, or on a regular file.

UDP

UDP is another protocol that is placed on top of IP. It is used for services that send small amounts of data between programs that do not require a long-time connection, or that send only little amount of data at a time. The 'talk' program uses the UDP protocol.

UDP adds only port numbers to the functionality IP gives, so the programmer needs to worry about checking for errors in messages (that come due to line noises and such), making sure the data sent using UDP arrives in the right order (which is not automatically achieved, due to IP's nature of not having a constantly open connection), and such.

There are various other protocols used in conjunction with IP, such as ARP, RARP, ICMP, SNMP and others, but those won't be dealt with in this document.

Network uniformity

(Skip this if you already know what byte ordering means, and what are 'well known ports' across Internet)

It is important to understand that protocols need to define some low-level details, in order to be able to talk to each other. We will discuss two such aspects here, in order to understand the example programs given later on.

Byte Order

It is an old argument amongst different computer manufacturers how numbers should be kept in a computer.

As all computers divide memory into bytes (or octets) of information, each 8 bit long, there is no problem with dealing with byte-sized numbers. The problem arises as we use larger numbers: short integers (2 bytes long) and long integers (4 bytes long). Suppose we have a short integer number, FE4Ch (that is, FE4C in hexadecimal notation). Suppose also that we say this number is kept in memory address 100h. This could mean one of two things, let's draw them out:

1. Big Endian:

```
                ---------------
    Address:    | 100h | 101h |
                ---------------
    Contents:   |  FEh |  4Ch |
                ---------------
```

2. Little Endian:

```
                ---------------
    Address:    | 100h | 101h |
                ---------------
    Contents:   |  4Ch |  FEh |
                ---------------
```

In the first form, also called 'Big Endian', The Most Significant Byte (MSB) is kept in the lower address, while the Least significant Byte (LSB) is kept in the higher address.

In the second form, also called 'Little Endian', the MSB is kept in the higher address, while the LSB is kept in the lower address.

Different computers used different byte ordering (or different endianess), usually depending on the type of CPU they have. The same problem arises when using a long integer: which word (2 bytes) should be kept first in memory? the least significant word, or the most significant word?

In a network protocol, however, there must be a predetermined byte and word ordering. The IP protocol defines what is called 'the network byte order', which must be kept on all packets sent across the Internet. The programmer on a Unix machine is not saved from having to deal with this kind of information, and we'll see how the translation of byte orders is solved when we get down to programming.

Well Known Ports

When we want two programs to talk to each other across Internet, we have to find a way to initiate the connection. So at least one of the 'partners' in the conversation has to know where to find the other one. This is done by letting one partner know the address (IP number + port number) of the other side.

However, a problem could arise if one side's address is randomly taken over by a third program. Then we'll be in real trouble. In order to avoid that, there are some port numbers which are reserved for specific purposes on any computer connected to the Internet. Such ports are reserved for programs such as 'Telnet', 'Ftp' and others.

These port numbers are specified in the file /etc/services on any decent Unix machine. Following is an excerpt from that file:

```
daytime        13/tcp
daytime        13/udp
netstat        15/tcp
qotd           17/tcp     quote
chargen        19/tcp     ttytst source
chargen        19/udp     ttytst source
ftp-data       20/tcp
ftp            21/tcp
telnet         23/tcp
smtp           25/tcp     mail
```

Read that file to find that Telnet, for example, uses port 23, and Ftp uses port 21. Note that for each kind of service, not only a port number is given, but also a protocol name (usually TCP or UDP). Note also that two services may use the same port number, provided that they use different protocols. This is possible due to the fact that different protocols have what is called different address spaces: port 23 of a one machine in the TCP protocol address space, is not equivalent to port 23 on the same machine, in the UDP protocol address space.

So how does this relate to you? When you will write your very own programs that need to 'chat' over Internet, you will need to pick an unused port number that will be used to initiate the connection.

2. Client and Server model

(Skip this if you already know what clients and servers are, and what are the relations between them).

This section will discuss the most common type of interaction across the Internet - the Client and Server model. Note that this discussion is relevant to other types of networks too, and a few examples will be mentioned along the text.

We will first explain what the client-server model is, then detail the roles of the client, the roles of the server, and give examples of some famous servers and clients used.

The Client-Server model

(Skip this if you know what the client-server model basically means)

The client-server model is used to divide the work of Internet programs into two parts. One part knows how to do a certain task, or to give a certain service. This part is called the Server. The other part knows how to talk to a user, and connect that user to the server. This part is called the Client. One server may give service to many different clients, either simultaneously, or one after the other (the server designer decides upon that). On the other hand, a Client talks to a single user at a time, although it might talk to several servers, if it's nature requires that. There are other such complex possibilities, but we will discuss only clients that talk to a single server.

Roles of Clients

(Skip this if you already know what clients are supposed to do)

A client's main feature is giving a convenient User interface, hiding the details of how the server 'talks' from the user. Today, people are trying to write mostly graphical clients, using windows, pop-up-menus and other such fancy stuff. We will leave this to someone else to explain, and concentrate on the networking part. The client needs to first establish a connection with the server, given its address. After the connection is established, The Client needs to be able to do two things:

1. Receive commands from the user, translate them to the server's language (protocol) and send them to the server.
2. Receive messages from the server, translate them into human-readable form, and show them to the user. Some of the messages will be dealt with by the client automatically, and hidden from the user. This time, the Client designer's choice.

This forms the basic loop a client performs:

```
get the server's address
form a working address that can be used to talk over Internet.
connect to the server
while (not finished) do:
  wait until there's either information from the server, or from the
      user.
  If (information from server) do
    parse information
    show to user, update local state information, etc.
  else {we've got a user command}
    parse command
    send to server, or deal with locally.
done
```

In the end of this tutorial you will be able to write such clients.

Roles of Servers

(Skip this if you already know what servers are supposed to do)

A server main feature is to accept requests from clients, handle them, and send the results back to the clients. We will discuss two kinds of servers: a **single-client server**, and a **multi-client server**.

Single Client Servers

These are servers that talk to a single client at a time. They need to be able to:

1. Accept connection requests from a Client.
2. Receive requests from the Client and return results.
3. Close the connection when done, or clear it if it's broken from some reason.

this forms the main loop a Single-Client Server performs:

```
bind a port on the computer, so Clients will be able to connect
forever do:
  listen on the port for connection requests.
  accept an incoming connection request
  if (this is an authorized Client)
    while (connection still alive) do:
      receive request from client
      handle request
      send results of request, or error messages
    done
  else
    abort the connection
done
```

Multi Client Servers

These are servers that talk to a several Clients at the same time. They need to be able to:

1. Accept new connection requests from Clients.
2. Receive requests from any Client and return results.
3. Close any connection that the client wants to end.

this forms the main loop a Multi-Client Server performs:

```
bind a port on the computer, so Clients will be able to connect
listen on the port for connection requests.
forever do:
  wait for either new connection requests, or requests from existing
          Clients.
  if (this is a new connection request)
    accept connection
    if (this is an un-authorized Client)
      close the connection
  else if (this is a connection close request)
    close the connection
  else { this is a request from an existing Client connection}
    receive request from client
    handle request
    send results of request, or error messages
done
```

"Famous" Servers and Clients

(Skip this if you already know about too many server types, or if you're not interested in knowing about them)

In this section we will give short descriptions of some "famous" servers and clients that are used daily over Internet, and over some other famous kinds of networks. This is simply an illustrative section that can be safely skipped by a rushing reader.

Internet Clients

FTP
Ftp is a Client program used to transfer files across the Internet. The Client connects to an FTPD Server (see below) and allows a user to scan a tree-structure of files and directories on the remote machine, retrieve and transmit files, etc. It uses two connections - one to exchange commands, and another to exchange data (the files themselves).

TELNET
Telnet is a Client program that enables working on a remote machine, using the keyboard and screen of the local one. It allows connecting from one type of machine to a completely

different type. The Telnet Client could be used to connect to any ASCII services, although it might be more convenient to use specialized Clients for that.

TALK

talk is a Client that allows two users across the Internet to carry out a real-time chat. The screen is split into two halves, and the user sees what he types in the upper half, and what the other user types in the upper half.

IRC

Another chat Client, which allows connecting to Internet-wide network of users and carry out both private conversations and conference talks.

LYNX

An information-retrieval Client. Lynx knows a wide set of information retrieval protocols, including some hyper-text protocols (which allow a plain text file to contain pointers to relevant information that lynx can follow), ftp, and so on.

Internet Servers

FTPD

The server that normally accepts connections from Ftp Clients. It works off the well-known Ftp port number 21. This server is a Single-Client server, i.e. it handles one connection only, and then terminates. The operating system has to make sure one running copy of the server will be created for each interested client.

TELNETD

The Server that talks to the Telnet client. Uses the well-known Telnet port number 23. TELNETD is a Single-Client Server, i.e. one TELNETD server is used for each Telnet connection request that should be handled.

TALKD

Used to receive talk requests from a remote machine, and notify the user that such a request has arrived. It has a role only in the initiation of the connection, not during the conversation itself, which is carried directly between the two Clients.

HTTPD

A server that serves information for hyper-text Clients such as Lynx, or graphical-based ones (Netscape, Internet Explorer.... or have you heard of Mosaic?).

IRCD

A server that cooperates with other servers of its kind to form the IRC network. Allows exchanging of information, along with state information about channels used for conferencing, and such.

3. Preparing an Internet address.

(Skip this if you know how to obtain the address of a given machine, prepare the structures involved, and deal with byte order translations)

Preview

The first thing you need to do when writing network applications, is obtain the addresses of the two involved machines: your address, and the remote host's address. This process is made up of several stages:

1. Address Resolution - Translating the host name into an IP address.
2. Byte Ordering - Translating host byte order into network byte order.
3. Address Formation - Forming up the remote address in the right structure.

Address Resolution

Given a host name, we want to find it's IP address. We do that using the function gethostbyname(). This function is defined in the file /usr/include/netdb.h (or the equivalent for your system) as follows:

```
struct hostent *gethostbyname(char *hostname);
```

The input to the function will be the name of the host whose address we want to resolve. The function returns a pointer to a structure hostent, whose definition is as follows:

```
  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 compatibility */
  };
Lets see what each field in the hostent structure means:
```

- `h_name`: This is the official name of the host, i.e. the full address.
- `h_aliases`: a pointer to the list of aliases (other names) the host might have.
- `h_addrtype`: The type of address this host uses.
- `h_length`: The length of the address. Different address types might have different lengths.
- `h_addr_list`: A pointer to the list of addresses of the host. Note that a host might have more than one address, as explained earlier.
- `h_addr`: In older systems, there was only the h_addr field, so it is defined here so old programs could compile without change on newer systems.

Byte Ordering

As explained earlier, we need to form addresses using the network byte order. Luckily, most networking functions accept addresses in host byte order, and return their results in network byte order. This means only a few fields will need conversions. These will include the port numbers only, as the addresses are already supplied by the system.

We normally have several functions (or macros) to form 4 types of translations:

- `htons()` - short integer from host byte order to network byte order.
- `ntohs()` - short integer from network byte order to host byte order.
- `htonl()` - long integer from host byte order to network byte order.
- `ntohl()` - long integer from network byte order to host byte order.

For example, since a port number is represented by a short integer, we could convert it from host byte order to network byte order by doing:

```
short host_port = 1234;
net_port = htons(host_port);
```

The other functions are used in a similar way.

Address Formation

Forming addresses for Internet protocols is done using a structure named sockaddr_in, whose definition, as given in the file /usr/include/netinet/in.h, is as follows:

```
struct sockaddr_in {
     short int        sin_family; /* Address family    */
     unsigned short   sin_port;    /* Port number      */
     struct in_addr   sin_addr;    /* Internet address */

     /* Pad to size of `struct sockaddr'. */
     /* Pad definition deleted */
};
```

The fields have the following meanings:

- `sin_family`: Family of protocols for this address. We will want the Internet family.
- `sin_port`: The port part of the address.
- `sin_addr`: The IP number part of the address.
- Pad: This will be explained in the next section.

After seeing the structure used for address formation, and having resolved the host name into an IP number, forming the address is done as follows:

```
char*                  hostname;  /* name part of the address */
short                  host_port; /* port part of the address */
struct hostent*        hen;       /* server's DNS entry */
struct sockaddr_in     sa;        /* address formation structure */

/* get information about the given host, using some the system's */
/* default name resolution mechanism (DNS, NIS, /etc/hosts...).  */
hen = gethostbyname(hostname_ser);
if (!hen) {
    perror("couldn't locate host entry");
}

/* create machine's Internet address structure  */
/* first clear out the struct, to avoid garbage */
memset(&sa, 0, sizeof(sa));

/* Using Internet address family */
sa.sin_family = AF_INET;
/* copy port number in network byte order */
sa.sin_port = htons(host_port);
/* copy IP address into address struct */
memcpy(&sa.sin_addr.s_addr, hen->h_addr_list[0], hen->h_length);
```

Notes:

- `perror()` is a function that prints an error message based on the global errno variable, and exits the process.
- `memset()` is a function used to set all bytes in a memory area of a specified size, to a specified value. On some (older) systems there is a different function that should be used instead named `bzero()`. Read your local manual page for additional information on it's usage. Note that `memcpy` is a part of the standard C library, so it should be used whenever possible in place of `bzero`.
- `memcpy()` is a function used to copy the contents of one memory area into another area. On some systems there is a different function with the same effect, named `bcopy()`. Like the `bzero` function, it should be avoided whenever `memcpy` is available.

4. The socket interface

We will now describe the interface used to write network applications in Unix systems (Especially Unix flavors derived from the BSD4.3 system). This interface is used throughout all kinds of networking software, but we will concentrate on the Internet protocol family.

We will first describe what a socket is, and how it relates to normal files, then explain what kinds of sockets exist on most Unix systems, how they are created using the socket() system call, and finally, how they are associated with a specific network connection, and how data is passed through them.

What is a socket?

A socket is formally defined as an endpoint for communication between an application program, and the underlying network protocols. This odd collection of words simply means that the program reads information from a socket in order to read from the network, writes information to it in order to write to the network, and sets sockets options in order to control protocol options. From the programmer's point of view, the socket is identical to the network. Just like a file descriptor is the endpoint of disk operations.

Types of sockets

In general, 3 types of sockets exist on most Unix systems: **Stream** sockets, **Datagram** sockets and **Raw** sockets.

Stream sockets are used for stream connections, i.e. connections that exist for a long duration. TCP connections use stream sockets.

Datagram sockets are used for short-term connections that transfer a single packet across the network before terminating. The UDP protocol uses such sockets, due to its connection-less nature.

Raw sockets are used to access low-level protocols directly, bypassing the higher protocols. They are the means for a programmer to use the IP protocol, or the physical layer of the network, directly. Raw sockets can therefore be used to implement new protocols on top of the low-level protocols. Naturally, they are out of our scope.

Creating sockets

Creation of sockets is done using the `socket()` system call. This system call is defined as follows:

```
int socket(int address_family, int socket_type, int proto_family);
```

address_family defines the type of addresses we want this socket to use, and therefore defines what kind of network protocol the socket will use. We will concentrate on the Internet address family, because we want to write Internet applications.

socket_type could be one of the socket types we mentioned earlier, or any other socket type that exists on your system. We choose the socket type according to the kind of interaction (and type or protocol) we want to use.

proto_family selects which protocol we want to socket to use. We will usually leave this value as 0 (or the constant `PF_UNSPEC` on some systems), and let the system choose the most suitable protocol for us. As for the protocol itself, In the Internet address family, a socket type of `SOCK_STREAM` will cause the protocol type to be set to TCP. A socket type of `SOCK_DGRAM` (Datagram socket) will cause the protocol type to be set to UDP.

The socket system call returns a file descriptor which will be used to reference the socket in later requests by the application program. If the call fails, however (due to lack of resources) the value returned will be negative (note that file descriptors have to be non-negative integers).

As an example, suppose that we want to write a TCP application. This application needs at least one socket in order to communicate across the Internet, so it will contain a call such as this:

```
int s;    /* descriptor of socket */

/* Internet address family, Stream socket */
s = socket(AF_INET, SOCK_STREAM, 0);
if (s < 0) {
    perror("socket: allocation failed");
}
```

Associating a socket with a connection

After a socket is created, it still needs to be told between which two end points it will communicate. It needs to be bound to a connection. There are two steps to this binding. The first is binding the socket to a local address. The second is binding it to a remote (foreign) address.

Binding to a local address could be done either explicitly, using the `bind()` system call, or implicitly, when a connecting is established. Binding to the remote address is done only when a connection is established. To bind a socket to a local address, we use the `bind()` system call, which is defined as follows:

```
int bind(int socket, struct sockaddr *address, int addrlen);
```

Note the usage of a different type of structure, namely `struct sockaddr`, than the one we used earlier (`struct sockaddr_in`). Why is the sudden change? This is due to the generality of the socket interface: ***sockets could be used as endpoints for connections using different types of address families***. Each address family needs different information, so they use different structures to form their addresses. Therefore, a generic socket address type, struct sockaddr, is defined in the system, and for each address family, a different variation of this structure is used. For those who know, this means that `struct sockaddr_in`, for example, is an overlay of struct sockaddr (i.e. it uses the same memory space, just divides it differently into fields).

There are 4 possible variations of address binding that might be used when binding a socket in the Internet address family.

The first is binding the socket to a specific address, i.e. a specific IP number and a specific port. This is done when we know exactly where we want to receive messages. Actually this form is not used in simple servers, since usually these servers wish to accept connections to the machine, no matter which IP interface it came from.

The second form is binding the socket to a specific IP number, but letting the system choose an unused port number. This could be done when we don't need to use a well-known port.

The third form is binding the socket to a wild-card address called INADDR_ANY (by assigning it to the `sockaddr_in` variable), and to a specific port number. This is used in servers that are supposed to accept packets sent to this port on the local host, regardless of through which physical network interface the packet has arrived (remember that a host might have more than one IP address).

The last form is letting the system bind the socket to any local IP address and to pick a port number by itself. This is done by not using the `bind()` system call on the socket. The system will make the local bind when a connection through the socket is established, i.e. along with the remote address binding. This form of binding is usually used by clients, which care only about the remote address (where they connect to) and don't need any specific local port or local IP address. However, there are exceptions here too.

Sending and receiving data over a socket

After a connection is established (We will explain that when talking about Client and Server writing), There are several ways to send information over the socket. We will only describe one method for reading and one for writing.

The `read()` system call

The most common way of reading data from a socket is using the `read()` system call, which is defined like this:

```
int read(int socket, char *buffer, int buflen);
```

- socket - The socket from which we want to read.
- buffer - The buffer into which the system will write the data bytes.
- buflen - Size of the buffer, in bytes (actually, how much data we want to read).

The read system call returns one of the following values:

- 0 - The connection was closed by the remote host.
- -1 - The read system call was interrupted, or failed for some reason.
- n - The read system call put 'n' bytes into the buffer we supplied it with.

Note that `read()` might read less than the number of bytes we requested, due to unavailability of buffer space in the system.

The `write()` system call

The most common way of writing data to a socket is using the `write()` system call, which is defined like this:

```
int write(int socket, char *buffer, int buflen);
```

- socket - The socket into which we want to write.
- buffer - The buffer from which the system will read the data bytes.
- buflen - Size of the buffer, in bytes (actually, how much data we want to write).

The write system call returns one of the following values:

- 0 - The connection was closed by the remote host.
- -1 - The write system call was interrupted, or failed for some reason.
- n - The write system call wrote 'n' bytes into the socket.

Note that the system keeps internal buffers, and the write system call write data to those buffers, not necessarily directly to the network. thus, a successful `write()` doesn't mean the data arrived at the other end, or was even sent onto the network. Also, it could be that only some of the bytes were written, and not the actual number we requested. It is up to us to try to send the data again later on, when it's possible, and we'll show several methods for doing just that.

Closing a socket.

When we want to abort a connection, or to close a socket that is no longer needed, we can use the `close()` system call. it is defined simply as:

```
int close(int socket);
```

- socket - The socket that we wish to close. If it is associated with an open connection, the connection will be closed.

---

5. Writing Clients

This section describes how to write simple client applications, using the socket interface described earlier. As you remember (hmm, do you?) from the second section, a classic Client makes a connection to the server, and goes into a loop of reading commands from the user, parsing them, sending requests to the server, receiving responses from the server, parsing them and echoing them back at the user.

We will begin by showing the C code of a simple Client without user-interaction. This Client connects to the standard time server of a given host, reads the time, and prints it on the screen. Most (Unix) Internet hosts have a standard server called daytime, that awaits connections on the well-known port number 13, and when it receives a connection request, accepts it, writes the time to the Client, and closes the connection.

Let's see how the Client looks. Note the usage of a new system call, `connect()`, which is used to establish a connection to a remote machine, and will be further explained immediately following the program text.

```
#include <stdio.h>          /* Basic I/O routines          */
#include <sys/types.h>      /* standard system types       */
#include <netinet/in.h>     /* Internet address structures */
#include <sys/socket.h>     /* socket interface functions  */
#include <netdb.h>          /* host to IP resolution       */

#define  HOSTNAMELEN   40   /* maximal host name length */
#define  BUFLEN        1024  /* maximum response size  */
#define  PORT          13   /* port of daytime server   */

int main(int argc, char *argv[])
{
    int                 rc;             /* system calls return value storage */
    int                 s;              /* socket descriptor                 */
    char                buf[BUFLEN+1]; /* buffer server answer              */
    char*               pc;             /* pointer into the buffer           */
    struct sockaddr_in  sa;             /* Internet address struct           */
    struct hostent*     hen;            /* host-to-IP translation            */

    /* check there are enough parameters */
    if (argc < 2) {
        fprintf(stderr, "Missing host name\n");
        exit (1);
    }

    /* Address resolution stage */
    hen = gethostbyname(argv[1]);
    if (!hen) {
        perror("couldn't resolve host name");
    }

    /* initiate machine's Internet address structure */
    /* first clear out the struct, to avoid garbage  */
    memset(&sa, 0, sizeof(sa));

    /* Using Internet address family */
    sa.sin_family = AF_INET;
    /* copy port number in network byte order */
    sa.sin_port = htons(PORT);
    /* copy IP address into address struct */
    memcpy(&sa.sin_addr.s_addr, hen->h_addr_list[0], hen->h_length);

    /* allocate a free socket                 */
    /* Internet address family, Stream socket */
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("socket: allocation failed");
    }

    /* now connect to the remote server. the system will */
    /* use the 4th binding method (see section 3)        */
    /* note the cast to a struct sockaddr pointer of the */
    /* address of variable sa.                           */
    rc = connect(s, (struct sockaddr *)&sa, sizeof(sa));

    /* check there was no error */
    if (rc) {
        perror("connect");
    }

    /* now that we are connected, start reading the socket */
    /* till read() returns 0, meaning the server closed    */
    /* the connection.                                     */
    pc = buf;

    while (rc = read(s, pc, BUFLEN - (pc-buf))) {
        pc += rc;
    }

    /* close the socket */
    close(s);

    /* pad a null character to the end of the result */
    *pc = '\0';

    /* print the result */
    printf("Time: %s\n", buf);

     /* and terminate */
    return 0;
}
```

The complete source code for this client may be found in the daytime-client.c file.

The Client's code should be pretty easy to understand now. All we did was combine the features we have seen so far into one program. The only new feature introduced here is the `connect()` system call.

This system call is responsible to making the connection to the specified address of the remote machine, using the specified socket. Note that the address is being type-cast into the general address type, `struct sockaddr`, because this same system call is used to establish connections in various address families, not just the Internet address family. How will the system then know we want an Internet connection? The answer is given in the socket's information. If you remember, we specified this socket will be used in the Internet address family (`AF_INET`) when we created it.

Note also how the reading loop is performed. We are asking the system to read as much data as possible in the `read()` system call. However, the system might need several reads before it has consumed all the bytes sent by the server, that's why we used the while loop. Remember, never assume a `read()` system call will return the exact number of bytes you specified in the call. If less is available, the call will return quickly, and will not wait for the rest of the data. On the other hand, if no data is available, the call will block (not return) until data is available. Thus, when writing "Real" Clients and Servers, some measures have to be taken in order to avoid that blocking.

We will not discuss right now Clients that read user input. This subject will be differed until we learn how to read information efficiently from several input devices.

---

6. Single-Clients Servers

Now that we have seen how a Client is written, let's give it a different server to talk to. We will write the "hello world" (didn't you wait for this?) server.

The "hello world" Server listens to a predefined port of our choice, and accepts incoming connections. It then writes the message "hello world" to the remote Client, and closes the connection. This will be done in an infinite loop, so we can serve a new Client after finishing with the current.

Note the introduction of two new system calls, `listen()` and `accept()`. The `listen()` system call asks the system to listen for new connections coming to our port. The `accept()` system call is used to accept (how obvious) such incoming connections. Both system calls will be explained further following the "hello world" Server's code.

```
#include <stdio.h>       /* Basic I/O routines          */
#include <sys/types.h>   /* standard system types       */
#include <netinet/in.h>  /* Internet address structures */
#include <sys/socket.h>  /* socket interface functions  */
#include <netdb.h>       /* host to IP resolution       */


#define  PORT   5050           /* port of "hello world" server */
#define  LINE   "hello world"  /* what to say to our clients   */

void main()
{
    int                rc;       /* system calls return value storage  */
    int                s;        /* socket descriptor                  */
    int                cs;       /* new connection's socket descriptor */
    struct sockaddr_in sa;       /* Internet address struct            */
    struct sockaddr_in csa;      /* client's address struct            */
    int                size_csa; /* size of client's address struct    */

    /* initiate machine's Internet address structure */
    /* first clear out the struct, to avoid garbage  */
    memset(&sa, 0, sizeof(sa));
    /* Using Internet address family */
    sa.sin_family = AF_INET;
    /* copy port number in network byte order */
    sa.sin_port = htons(PORT);
    /* we will accept connections coming through any IP */
    /* address that belongs to our host, using the      */
    /* INADDR_ANY wild-card.                            */
    sa.sin_addr.s_addr = INADDR_ANY;

    /* allocate a free socket */
    /* Internet address family, Stream socket */
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("socket: allocation failed");
    }
    /* bind the socket to the newly formed address */
    rc = bind(s, (struct sockaddr *)&sa, sizeof(sa));

    /* check there was no error */
    if (rc) {
        perror("bind");
    }

    /* ask the system to listen for incoming connections   */
    /* to the address we just bound. specify that up to    */
    /* 5 pending connection requests will be queued by the */
    /* system, if we are not directly awaiting them using  */
    /* the accept() system call, when they arrive.         */
    rc = listen(s, 5);

    /* check there was no error */
    if (rc) {
        perror("listen");
    }

    /* remember size for later usage */
    size_csa = sizeof(csa);

    /* enter an accept-write-close infinite loop */
    while (1) {
        /* the accept() system call will wait for a     */
        /* connection, and when one is established, a   */
        /* new socket will be created to handle it, and */
        /* the csa variable will hold the address       */
        /* of the Client that just connected to us.     */
        /* the old socket, s, will still be available   */
        /* for future accept() statements.              */
        cs = accept(s, (struct sockaddr *)&csa, &size_csa);

        /* check for errors. if any, enter accept mode again */
        if (cs < 0)
            continue;

        /* oak, we got a new connection. do the job... */
        write(cs, LINE, sizeof(LINE));

        /* now close the connection */
        close(cs);
    }
}
```

The complete source code for this server may be found in the hello-world-server.c file.

Look how little we had to add to the basic stuff in order to form our first server. The only two additions were the `listen()` and the `accept()` system calls. Lets examine them a little more.

If we want to serve incoming connections, we need to ask the system to listen on the specified port. If we don't do that, the remote Client will get a "connection refused" error. Once the system listens on the port, It could happen that more than one Client will ask for service simultaneously. We can tell the system how many Clients may "wait in line". This will be the second parameter to the `listen()` system call.

After issuing the `listen()` system call, we still need to actively accept incoming connections. This is done using the `accept()` system call. We tell it which socket is bound to the port we want to accept connection from, and give it the address of a variable in which the call will give us the address of the remote Client, once a connection is established. It will also update the size of the address, based on the address family used, in the variable whose address we pass as the third argument. We are not using the Client's address in our simple server, but other servers that might want to authenticate their Clients (or just to know where they are coming from), will use it.

Finally, the `accept()` system call returns a number of a new socket, which is allocated for the new established connection. This gives us a socket bound to the correct local and remote addresses, while not destroying the binding of the original socket that we can later use to accept new connections.

---

7. Multi-Clients Servers

If single-Client Servers were a rather simple case, the multi-Client ones are a tougher nut. There are two main approaches to designing such servers.

Approach 1.

The first approach is using one process that awaits new connections, and one more process (or thread) for each Client already connected. This approach makes design quite easy, cause then the main process does not need to differ between servers, and the sub-processes are each a single-Client server process, hence, easier to implement.

However, this approach wastes too many system resources (if child processes are used), and complicates inter-Client communication: If one Client wants to send a message to another through the server, this will require communication between two processes on the server, or locking mechanisms, if using multiple threads.

Approach 2.

The second approach is using a single process for all tasks: waiting for new connections and accepting them, while handling open connections and messages that arrive through them. This approach uses less system resources, and simplifies inter-Client communication, although making the server process more complex.

Luckily, the Unix system provides a system call that makes these tasks much easier to handle: the `select()` system call.

The `select()` system call puts the process to sleep until any of a given list of file descriptors (including sockets) is ready for reading, writing or is in an exceptional condition. When one of these things happen, the call returns, and notifies the process which file descriptors are waiting for service.

The select system call is defined as follows:

```
int select(int numfds,
           fd_set *rfd,
           fd_set *wfd,
           fd_set *efd,
           struct timeval *timeout);
```

- numfds - highest number of file descriptor to check.
- rfd - set of file descriptors to check for reading availability.
- wfd - set of file descriptors to check for writing availability.
- efd - set of file descriptors to check for exceptional condition.
- timeout - how long to wait before terminating the call in case no file descriptor is ready.

`select()` returns the number of file descriptors that are ready, or -1 if some error occurred.

We give `select()` 3 sets of file descriptors to check upon. The sockets in the rfd set will be checked whether they sent data that can be read. The file descriptors in the wfd set will be checked to see whether we can write into any of them. The file descriptors in the efd set will be checked for exceptional conditions (you may safely ignore this set for now, since it requires a better understanding of the Internet protocols in order to be useful). Note that if we don't want to check one of the sets, we send a `NULL` pointer instead.

We also give `select()` a timeout value - if this amount of time passes before any of the file descriptors is ready, the call will terminate, returning 0 (no file descriptors are ready).

NOTE - We could use the `select()` system call to modify the Client so it could also accept user input, Simply by telling it to `select()` on a set comprised of two descriptors: the standard input descriptor (descriptor number 0) and the communication socket (the one we allocated using the `socket()` system call). When the `select()` call returns, we will check which descriptor is ready: standard input, or our socket, and this way will know which of them needs service.

There are three more things we need to know in order to be able to use select. One - how do we know the highest number of a file descriptor a process may use on our system? Two - how do we prepare those sets? Three - when select returns, how do we know which descriptors are ready - and what they are ready for?

As for the first issue, we could use the `getdtablesize()` system call. It is defined as follows:

```
int getdtablesize();
```

This system call takes no arguments, and returns the number of the largest file descriptor a process may have. On modern systems, we could instead use the `getrlimit()` system call, using the `RLIMIT_NOFILE` parameter. Refer to the relevant manual page for more information.

As for the second issue, the system provides us with several macros to manipulate fd_set type variables.

`FD_ZERO(fd_set *xfd)`
    Clear out the set pointed to by 'xfd'.
`FD_SET(fd, fd_set *xfd)`
    Add file descriptor 'fd' to the set pointed to by 'xfd'.
`FD_CLR(fd, fd_set *xfd)`
    Remove file descriptor 'fd' from the set pointed to by 'xfd'.
`FD_ISSET(fd, fd_set *xfd)`
    check whether file descriptor 'fd' is part of the set pointed to by 'xfd'.

An important thing to note is that `select()` actually modifies the sets passed to it as parameters, to reflect the state of the file descriptors. This means we need to pass a copy of the original sets to `select()`, and manipulate the original sets according to the results of `select()`. In our example program, variable 'rfd' will contain the original set of sockets, and 'c_rfd' will contain the copy passed to `select()`.

Here is the source code of a Multi-Client echo Server. This Server accepts connection from several Clients simultaneously, and echoes back at each Client any byte it will send to the Server. This is a service similar to the one give by the Internet Echo service, that accepts incoming connections on the well-known port 7. Compare the code given here to the algorithm of a Multi-Client Server presented in the Client-Server model section.

```
#include <stdio.h>          /* Basic I/O routines          */
#include <sys/types.h>      /* standard system types       */
#include <netinet/in.h>     /* Internet address structures */
#include <sys/socket.h>     /* socket interface functions  */
#include <netdb.h>          /* host to IP resolution       */
#include <sys/time.h>       /* for timeout values          */
#include <unistd.h>         /* for table size calculations */

#define  PORT    5060             /* port of our echo server */
#define  BUFLEN  1024             /* buffer length           */

void main()
{
    int                i;              /* index counter for loop operations  */
    int                rc;             /* system calls return value storage  */
    int                s;              /* socket descriptor                  */
    int                cs;             /* new connection's socket descriptor */
    char               buf[BUFLEN+1]; /* buffer for incoming data           */
    struct sockaddr_in sa;             /* Internet address struct            */
    struct sockaddr_in csa;            /* client's address struct            */
    int                size_csa;       /* size of client's address struct    */
    fd_set             rfd;            /* set of open sockets                */
    fd_set             c_rfd;          /* set of sockets waiting to be read  */
    int                dsize;          /* size of file descriptors table     */

    /* initiate machine's Internet address structure */
    /* first clear out the struct, to avoid garbage  */
    memset(&sa, 0, sizeof(sa));
    /* Using Internet address family */
    sa.sin_family = AF_INET;
    /* copy port number in network byte order */
    sa.sin_port = htons(PORT);
    /* we will accept connections coming through any IP */
    /* address that belongs to our host, using the      */
    /* INADDR_ANY wild-card.                            */
    sa.sin_addr.s_addr = INADDR_ANY;

    /* allocate a free socket */
    /* Internet address family, Stream socket */
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("socket: allocation failed");
    }
```

```
/* bind the socket to the newly formed address */
rc = bind(s, (struct sockaddr *)&sa, sizeof(sa));

/* check there was no error */
if (rc) {
    perror("bind");
}

/* ask the system to listen for incoming connections   */
/* to the address we just bound. specify that up to    */
/* 5 pending connection requests will be queued by the */
/* system, if we are not directly awaiting them using  */
/* the accept() system call, when they arrive.         */
rc = listen(s, 5);

/* check there was no error */
if (rc) {
    perror("listen");
}

/* remember size for later usage */
size_csa = sizeof(csa);

/* calculate size of file descriptors table */
dsize = getdtablesize();

/* close all file descriptors, except our communication socket */
/* this is done to avoid blocking on tty operations and such.  */
for (i=0; i<dsize; i++)
    if (i != s)
        close(i);

/* we initially have only one socket open,    */
/* to receive new incoming connections.       */
FD_ZERO(&rfd);
FD_SET(s, &rfd);
/* enter an accept-write-close infinite loop */
while (1) {
    /* the select() system call waits until any of  */
    /* the file descriptors specified in the read,  */
    /* write and exception sets given to it, is     */
    /* ready to give data, send data, or is in an   */
    /* exceptional state, in respect. the call will */
    /* wait for a given time before returning. in   */
    /* this case, the value is NULL, so it will     */
    /* not timeout. dsize specifies the size of the */
    /* file descriptor table.                       */
    c_rfd = rfd;
    rc = select(dsize, &c_rfd, NULL, NULL, NULL);

    /* if the 's' socket is ready for reading, it   */
    /* means that a new connection request arrived. */
    if (FD_ISSET(s, &c_rfd)) {
        /* accept the incoming connection */
        cs = accept(s, (struct sockaddr *)&csa, &size_csa);
```

```
            /* check for errors. if any, ignore new connection */
            if (cs < 0)
                continue;

            /* add the new socket to the set of open sockets */
            FD_SET(cs, &rfd);

            /* and loop again */
            continue;
        }

        /* check which sockets are ready for reading, */
        /* and handle them with care.                 */
        for (i=0; i<dsize; i++) {
            if (i != s && FD_ISSET(i, &c_rfd)) {
                /* read from the socket */
                rc = read(i, buf, BUFLEN);

                /* if client closed the connection... */
                if (rc == 0) {
                    /* close the socket */
                    close(i);
                    FD_CLR(i, &rfd);
                }
                /* if there was data to read */
                else {
                    /* echo it back to the client         */
                    /* NOTE: we SHOULD have checked that */
                    /* indeed all data was written...    */
                    write(i, buf, rc);
                }
            }
        }
    }
}
```

The complete source code for this server may be found in the multi-client-echo-server.c file.

8. Conclusions

That's it. If you got as far as here, and hopefully tried playing a little with the code examples, you probably got the basic notion of what it takes to write simple clients and servers that communicate using the TCP protocol. We didn't cover UDP-based clients and servers, but hopefully, you'll manage getting there by referring to one of the more advanced resources mentioned below.

And remember: clients, and especially servers, are expected to be robust creatures. Yet, the network is a too shaky ground to assume everything will work smoothly. *Expect the unexpected*. Check the return value of any system call you use, and act upon it. If a system call failed, try to figure out why it failed (using the returned error code and possibly the `errno` variable), and if you cannot write code to bypass that kind of failure - at least give your users an error message they can understand.

---

9. See Also - or, where to go from here

The following references might be good places to continue exploring socket programming:

- Internetworking with TCP/IP - Volume I, by Douglas Comer
  This fine book explains in great details the internals of the TCP/IP protocol suite: what protocols comprise the TCP/IP protocol family; how each protocol performs its chores and how its packets are ordered. It also explains about local area networks (LANs), such as Ethernet and Token-Ring, as well as the history of the Internet.
- Unix Network Programming - By W. Richard Stevens
  The "bible" for this subject. Teaches how to write programs that communicate, Not only via sockets and TCP/IP, but also using other types of IPC (Inter-Process Communications). A "Must Read" for every serious network programmer. If you're interested in simply writing network programs, without delving into too much theory, this should be your next stop.

Part II Non-blocking I/O

1.1 The O_NONBLOCK flag

The finger client in Project 1 is only as fast as the server to which it talks. When the program calls connect, read, and sometimes even write, it must wait for a response from the server before making any further progress. This doesn't ordinarily pose a problem; if finger blocks, the operating system will schedule another process so the CPU can still perform useful work.

On the other hand, suppose you want to finger some huge number of users. Some servers may take a long time to respond (for instance, connection attempts to unreachable servers will take over a minute to time out). Thus, your program itself may have plenty of useful work to do, and you may not want to schedule another process every time a server is slow to respond.

For this reason, Unix allows file descriptors to be placed in a non-blocking mode. A bit associated with each file descriptor O_NONBLOCK, determines whether it is in non-blocking mode or not. We can set the O NONBLOCK bit of a file descriptor non-blocking with the fcntl system call.

```
if ((n = fcntl (s, F_GETFL)) < 0 || fcntl (s, F_SETFL, n | O_NONBLOCK) < 0)....
```

There are a number of ways to achieve this goal other than placing sockets in non-blocking mode. For instance, a threads package could be used to spawn a process for each request; requests would then be made in concurrently in blocking mode. An alternative, signal-driven asynchronous socket interface also exists in a number of Unix operating systems. The method described here, however, avoids the common pitfalls of preemptive concurrency and is more portable than signal driven methods.

Many system calls behave slightly differently on file descriptors which have O_NONBLOCK set:

read.
When there is data to read, read behaves as usual. When there is an end of file, read still returns 0. If, however, a process calls read on a non-blocking file descriptor when there is no data to be read yet, instead of waiting for data, read will return **-1** and set errno to **EAGAIN**. In general, a read call to a non-blocking socket will return return immediately. Under this definition, data is available to be read when it has been loaded into a kernel buffer by the OS and is ready to be copied into the user-specified buffer.

write.
Like read, write will return -1 with an errno of **EAGAIN** if there is no buffer space available for the operating system to copy data to. If, however, there is some buffer space but not enough to contain the entire write request, write will take as much data as it can and return a value smaller than the length specified as its third argument. Code must handle such “short writes” by calling write again later on the rest of the data.

connect.
A TCP connection request requires a response from the listening server. When called on a non-blocking socket, connect cannot wait for such a response before returning. For this reason, connect on a non-blocking socket usually returns -1 with errno set to EINPROGRESS. Occasionally, however, connect succeeds or fails immediately even on a non-blocking socket, so you must be prepared to handle this case.

accept.
When there are connections to accept, accept will behave as usual. If there are no pending connections, however, accept will return -1 and set errno to EWOULDBLOCK. It's worth noting that, on some operating systems, file descriptors returned by accept have O_NONBLOCK **clear**, whether or not the listening socket is non-blocking. In a asynchronous servers, one often sets O_NONBLOCK immediately on any file descriptors accept returns.

1.2 select: Finding out when sockets are ready

O_NONBLOCK allows an application to keep the CPU when an I/O system call would ordinarily block. However, programs can use several non-blocking file descriptors and still find none of them ready for I/O. Under such circumstances, programs need a way to avoid wasting CPU time by repeatedly polling individual file descriptors. The select system call solves this problem by letting applications sleep until one or more file descriptors in a set is ready for I/O.

select usage:

```
  int select (int nfds, fd_set *rfds, fd_set *wfds, fd_set *efds, struct timeval *timeout);
```

select takes pointers to sets of file descriptors and a timeout. It returns when one or more of the file descriptors are ready for I/O, or after the specified timeout. Before returning, select modifies the file descriptor sets so as to indicate which file descriptors actually are ready for I/O. select returns the number of ready file descriptors, or -1 on an error.

select represents sets of file descriptors as bit vectors --- one bit per descriptor. The first bit of a vector is 1 if that set contains file descriptor 0, the second bit is 1 if it contains descriptor 1, and so on. The argument nfds specifies the number of bits in each of the bit vectors being passed in. Equivalently, nfds is one more than highest file descriptor number select must check on.

These file descriptor sets are of type fd_set. Several macros in system header files allow easy manipulation of this type. If fd is an integer containing a file descriptor, and **fds** is a variable of type fd_set, the following macros can manipulate fds:

-- FD_ZERO (&fds);
Clears all bits in a fds

-- FD_SET (fd, &fds);
Sets the bit corresponding to file descriptor fd in fds.

-- FD_CLR (fd, &fds);
Clears the bit corresponding to file descriptor fd in fds.

-- FD_ISSET (fd, &fds);
Returns a true if and only if the bit for file descriptor fd is set in fds.

select takes three file descriptor sets as input. rfds specifies the set of file descriptors on which the process would like to perform a **read** or **accept**. wfds specifies the set of file descriptors on which the process would like to perform a **write**. efds is a set of file descriptors for which the process is interested in exceptional events such as the arrival of out of band data. In practice, people rarely use efds. Any of the fd_set * arguments to select can be NULL to indicate an empty set.

The argument timeout specifies the amount of time to wait for a file descriptor to become ready. It is a pointer to a structure of the following form:

```
struct timeval {
  long tv_sec; /* seconds */
  long tv_usec; /* and microseconds */
};
```

timeout can also be NULL, in which case select will wait indefinitely.

Tips and subtleties

File descriptor limits.

Programmers using select may be tempted to write code capable of using arbitrarily many file descriptors. Be aware that the operating system limits the number of file descriptors a process can have. If you don't bound the number of descriptors your program uses, you must be prepared for system calls like socket and accept to fail with errors like EMFILE. By default, a modern Unix system typically limits processes to 64 file descriptors (though the setrlimit system call can sometimes raise that limit substantially). Don't count on using all 64 file descriptors, either. All processes inherit at least three file descriptors (standard input, output, and error), and some C library functions need to use file descriptors, too. It should be safe to assume you can use 56 file descriptors, though.

If you do raise the maximum number of file descriptors allowed to your process, there is another problem to be aware of. The **fd_set** type defines a vector with FD_SETSIZE bits in it (typically 256). If your program uses more than FD_SETSIZE file descriptors, you must allocate more memory for each vector than an fd_set contains, and you can no longer use the FD_ZERO macro.

Using select with connect.

After connecting a non-blocking socket, you might like to know when the connect has completed and whether it succeeded or failed. TCP servers can accept connections without writing to them (for instance, our finger server waited to read a username before sending anything back over the socket). Thus, selecting for readability will not necessarily notify you of a connect's completion; you must check for writability.

When select does indicate the writability of a non-blocking socket with a pending connect, how can you tell if that connect succeeded? The simplest way is to try writing some data to the file descriptor to see if the write succeeds. This approach has two small complications. First, writing to an unconnected socket does more than simply return an error code; it kills the current process with a SIGPIPE signal. Thus, any program that risks writing to an unconnected socket should tell the operating system that it wants to ignore SIGPIPE. The signal system call accomplishes this:

```
signal (SIGPIPE, SIG_IGN);
```

The second complication is that you may not have any data to write to a socket, yet still wish to know if a non-blocking connect has succeeded. In that case, you can find out whether a socket is connected with the getpeername system call. getpeername takes the same argument types as accept, but expects a connected socket as its first argument. If getpeername returns 0 (meaning success), then you know the non-blocking connect has succeeded. If it returns -1, then the connect has failed.

Trick on timeout:

```
/* Enable keepalives to make sockets time out if servers go away. */
n = 1;
if (setsockopt (s, SOL_SOCKET, SO_KEEPALIVE, (void *) &n, sizeof (n)) < 0)....
```

1.3 Putting all together

Write an asynchronous fingerclient program that fingers multiple users from multiple servers.