Chapter 9 Interprocess Communication, Part II

Concepts Covered

Sockets
API: dup, dup2, fpathconf, gethostbyname, mkfifo, mknod, pipe, pclose, popen, select, setsid, shutdown, syslog, tee.

9.1 Sockets

Pipes are a good segue into sockets. *Sockets* are used like pipes in many ways but, unlike pipes, they can be used across networks. Sockets allow unrelated processes on different computers on a network to exchange data through a channel, using ordinary `read()` and `write()` system calls. We call this *remote interprocess communication.* Formally, a socket is an endpoint of communication between processes. Although sockets are used primarily across networks, they can also be used on a single host in place of pipes.

9.1.1 Background

The development of sockets derives from the Berkeley distributions of UNIX in the early 1980's, having first appeared in 4.2BSD. In 1987, AT&T developed a different API for remote interprocess communication in their System V Release 3 (SVR3), called the *Transport Level Interface* (*TLI*). The Transport Layer Interface was the System V answer to the BSD sockets programming interface. TLI was later standardized as XTI, the X/Open Transport Interface.

In many ways, TLI has advantages over sockets. TLI and XTI were widely used and preferred over the POSIX Sockets API. They are still supported in SVR4-derived operating systems and systems such as Solaris and Mac OS. Today, the UNIX 03 Single UNIX Specification declares POSIX Sockets as the preferred API for new transport protocols.

Because sockets can be accessed like files and work the same whether on a local machine or across a network, programming them is easier than TLI programming, which requires many more structures. For this reason we focus on sockets here. In order to understand how to use sockets, you need to know the basics of networks.

9.1.2 Connections

There are two ways in which sockets can be used, corresponding roughly to the difference between making a telephone call and having an email conversation with someone. When you make a telephone call to someone, you have a conversation over a dedicated communication channel, the telephone line, and you stay connected with the person on the other end for the duration of the call. In socket parlance this is called a *connection oriented model.* When you have an email conversation with someone, the messages are sent to the other person across different paths, and there is no dedicated connection. In fact there is no guarantee that the messages that you send will arrive in the order you send them, and the only way for the person who receives them to know who sent them is for them to have a return address in their message header. In socket parlance this is the *connection-less model*.

The connection oriented model uses the *Transmission Control* Protocol, known as *TCP*. The connection-less model uses the *User Datagram Protocol*, or *UDP*. There are many important differences between TCP and UDP, or equivalently, between connection oriented and connection-less models, but we will not go into them at length here. The most important differences are that TCP provides a reliable, full-duplex, sequenced channel with *flow control*. (Flow control is the process of managing the rate of data transmission so that senders and receivers can operate at different speeds without loss of data or retransmissions.) UDP can be full-duplex but it is not reliable (no guarantee of packet delivery), not sequenced (packets can arrive in different order than they were sent), and has no flow control (a sender can send faster than the receiver can receive).

9.1.3 Communication Basics

In order to understand how to program with sockets, you need to have a basic understanding of the important concepts that underlie their use. This includes network addresses, communication domains (not internet domains), protocol families, and socket types.

Network Addresses, Ports, and Socket Addresses

For two processes to communicate, they need to know each other's network addresses. At the level of socket programming, a network address consists of two parts: an *internet (IP) address* and a *port number*. The IP address, if 32 bits, consists of four 8-bit *octets*, and is expressed in the standard dot-notation as in `"146.95.2.131"`. These 32-bit address are known as *IPv4* addresses. In 1995, a 128-bit address was developed, known as *IPv6*. Some computers have multiple network interface cards and therefore may have multiple internet addresses. It used to be the case that internet addresses had a specific structure and were divided into address classes. That is no longer the case. They are now just flat addresses.

The kernel does not represent IP addresses as strings of octets – that would be inefficient. It uses the `in_addr_t` data type, defined in `<arpa/inet.h>`, to represent an IP address. However, we will see that there are functions to convert from one format to the other.

Each server on a machine has to have a specific *port* that to use. There are many analogies that we could use, but if you think of an IP address as specifying a specific company's main telephone line, then the port is like a telephone extension within the company. The server uses a specific port for its services and the clients have to know the port number in order to contact the server. A port is a 16-bit integer.

Certain port numbers are *well-known* and reserved by particular applications and services. For example, port 7 is for echo servers, 13 for daytime servers, 22 for SSH, 25 for SMTP, and 80 for HTTP. Port numbers from 1 to 1023 are the *well-known ports*. To see a list of the port numbers in use, take a look at the file `/etc/services`.

Ports 1024 through 49151 are *registered ports*. These numbers are not controlled and a service can use one if it is not already in use.

Ports 49152 through 65536 cannot be used. They are called *ephemeral ports*, which are assigned automatically by TCP or UDP for client use.

The `lsof` command can be used to view the ports that are currently open. The command is actually more general than this – it can be used to view all open files. To see a list of open ports, use either

```
lsof -Pnl +M -i4
```

(the i4 restricts to IPv4), or

```
netstat -lptu
```

to see listening sockets for both TCP and UDP. Read the man pages for `lsof` and `netstat` to learn more about these commands.

A *socket address* is a combination of a network address and a port.

Domains and Protocol Families

In order for two processes to communicate, they must use the same *protocol.* Part of the procedure for establishing communication involves specifying the communication domain and the protocol family. For example, the domain `AF_INET` specifies that the protocol family is the IPV4 set of protocols. Within that family there may be a choice of a specific protocol, such as TCP or UDP. The domain might instead be `AF_UNIX`, which specifies that the protocol family is restricted to the local machine. In this case there will not be a choice of protocol. When a socket is created, the domain and protocol are specified as two of the arguments to the function. The `socket` (2) man page lists various domains together with the man pages that contain possible protocols that can be used with them.

Socket Types

When a socket is created, its type must be specified. The type corresponds to the type of connection. A connection oriented communication uses *stream sockets*, of type `SOCK_STREAM`, whereas a connection-less communication uses *datagram sockets*, of type `SOCK_DGRAM`. There are also *raw sockets*, with type `SOCK_RAW`. Linux provides several other socket types, and POSIX requires support for the type, `SOCK_SEQPACKET`.

9.1.4 The Socket Interface[1]

A socket is identified within the operating system by an identifier. The socket() system call creates a socket and returns a file descriptor that represents it. We do not have to know how a socket is implemented to use it, however, you should think of a socket as something like the file structure that represents a file in UNIX. It is an internal structure in the kernel, accessed through a file descriptor, representing one end of a communication channel that has a specific network address, family (also called domain), port number, and socket type.

The `socket()` function creates what is called an unnamed socket:

[1]This section must be updated. Some of the methods described are now obsolete.

```
#include <sys/socket.h>
int socket( int domain, int type, int protocol);
```

The domain is an integer specifying the address family and protocol. These families are defined in <sys/socket.h>. Some of the common domain values are

Name	Purpose
AF_UNIX	Local communication
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols

The type can be one of `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW` or several others. The `SOCK_STREAM` type is the connection model, with full-duplex, reliable, sequenced transmissions.

The protocol can be used to specify a particular protocol in the case that there is more than one choice for the particular type of socket and address family. Setting it to 0 ensures that the kernel will pick the appropriate protocol.

The return value of `socket()` is a file descriptor that can be used to read or write the socket. As an example,

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

creates a connection-oriented socket that can be used for communication over the internet.

9.1.5 Setting Up a Connection Oriented Service

We will go through the steps that a server must take in a connection-oriented model. The basic steps that a server must take are below. Details on how to use the specific functions will follow.

1. Create a socket using `socket()`. This creates an endpoint of communication, but does not associate any particular internet address or port number to it.
2. Bind the socket to a local protocol address using `bind()`. This gives a "name" to the socket.
3. The socket created so far is an active socket, one that can connect to other sockets actively, like dialing another telephone number. Since this is the server, the purpose of this socket is not to "dial-out" but to listen for incoming calls. Therefore, the server must now call `listen()` to tell the kernel that all it really wants to do is listen for incoming messages and set a limit on its queue size. This call will basically put the socket into the `LISTEN` state in the TCP protocol.

   After `listen()` has returned, two queues have been created for the server. One queue stores incoming connection requests that have not yet completed the TCP handshake protocol. The other queue stores incoming requests that have completed the handshake. These requests are ready to be serviced.

4. Enter a loop in which it repeatedly accepts new connections and processes them. It can accept a new connection by calling `accept()`. The `accept()` function removes the request at the front of the completed connection queue and creates a second socket that the server can use for talking with this client. The return value of `accept()` is a file descriptor that represents this socket. The original socket continues to exist. The idea is that the original socket is just for listening, not talking to clients. In fact it is called the *listening socket*, and the new socket is called the *connected socket.* When the connection is closed, this connected socket is removed.

That is the essence of the server's tasks. Now what remains is to see how to program this.

9.1.6 Programming a Connection Oriented Server

We have already seen how the `socket()` call works. The step of binding a local protocol address to the socket is carried out with `bind()`, but before we look at `bind()` we need to see how these addresses are represented. A generic socket address is defined by the `sockaddr` structure defined in `<sys/socket.h>`:

```
struct sockaddr {
    sa_family_t sa_family; /* address family */
    char sa_data[];        /* socket address */
};
```

This is a generic socket address structure because it is not specific to any one address family. When you call `bind()`, you will be specifying a particular family, such as `PF_INET` or `PF_UNIX`. For each of these there is a different form of socket address structure. The address structure for `PF_INET`, defined in `<netinet/in.h>`, would be

```
struct sockaddr_in {
    sa_family_t    sin_family;  /* internet address family */
    in_port_t      sin_port;    /* port number */
    struct in_addr sin_addr;    /* IP address */
    unsigned char  sin_zero[8]; /* padding */
};
```

The `bind()` system call takes the socket file descriptor returned by `socket()` and an address structure like the one above, and "binds" them together to form the end of a socket that can now be used by processes living somewhere in the internet to find this server:

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

int bind(int sockfd, const struct sockaddr *address,
         socklen_t addrlen);
```

Putting these few steps together, we might start out with the following code:

```
int listenfd;
int size  = sizeof(struct sockaddr_in);
struct sockaddr_in server = {PF_INET, 25555, INADDR_ANY};

if ( (listenfd = socket(PF_INET, SOCK_STREAM, 0)) == -1 ) {
    perror("socket call failed");
    exit(1);
}

if ( bind( listenfd, ( struct sockaddr *) &server, size ) == -1 )  {
    perror(" bind call failed");
    exit(1);
}
```

The `sockaddr_in` struct is initialized to use the IPv4 protocol family with a port of 25555, large enough to be safe for our purposes, and `INADDR_ANY` as the local IP address. Specifying this constant means that if there is more than one IP address for this host, any will do. The bind call is given the listenfd descriptor, the address of this struct, and its size.

The next step is to call listen(), which is defined by

```
#include <sys/socket.h>
int listen(int sockfd, int queue_size);
```

The first argument is the descriptor for the already bound socket, and the second is the maximum size of the queue of pending (incomplete) connections.

The `accept()` call is defined as follows:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr,
socklen_t *addrlen);
```

The `accept()` call expects the socket descriptor of a socket that has been created with `socket()`, bound to a local address with `bind()`, and set to listen with `listen()`. The second argument, if not `NULL`, is a pointer to a generic socket address structure, and the third is the address of a variable that stores its length in bytes. After the call, the address will be filled with the client's socket address, and the size will reflect the true size of the client's specific socket address struct. The return value will be the descriptor of a connected socket. The `accept()` will block waiting for a connection.

We can put all of this together in a simple concurrent server that, yes, once again, does lower to upper case conversion. This time it will handle just one character at a time. We will move on to a more interesting task afterwards. This code is based on an example from *[Haviland et al]*. It forks a child process to handle each incoming connection. The client and the server will share a common header file, `sockdemo1.h`, which is displayed first, followed by the server code.

```
Listing sockdemo1.h

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <ctype.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>

#define SOCKADDR          (struct sockaddr*)
#define SIZE              sizeof(struct sockaddr_in)
#define DEFAULT_HOST      "localhost"
#define PORT              25555
#define  ERROR_EXIT( _mssg, _num)   perror(_mssg);exit(_num);
#define  MAXLINE          4096
```

Comments.

- (Perhaps out of laziness, I finally wrote a little macro (`ERROR_EXIT`) so that I do not have to keep typing the `perror(); exit()` combination on failures of system calls. It is included in this header file. Rather than making it a function, I made it a macro so that the code is faster.)
- The `SOCKADDR` macro reduces typing.
- The server and client are compiled with the same header so that the port number is hard-coded into each. The number 25555 appears to be unused on all of the machines I have run this example on.

The server code follows.

```
Listing sockdemo1_server.c

#include "sockdemo1.h"
#include <sys/wait.h>

#define LISTEN_QUEUE_SIZE   5

/* The following typedef simplifies the function definition after it */
typedef void    Sigfunc(int);   /* for signal handlers */

/* override existing signal function to handle non-BSD systems */
Sigfunc*  Signal(int signo, Sigfunc *func);
```

```
/* Signal handlers */
void on_sigpipe( int signo );

void on_sigchld( int signo );

/*************************************************************************/

/* This needs to be global because the signal handler has to access it */
int connectionfd;

/*************************************************************************/
int main(int argc, char* argv[])
{
    int listenfd;     /* holds the file descriptor for the socket */
    char c;           /* this example is a ToUpcase server */

    /* A sockaddr_in struct is a struct that stores address and */
    /* port information for a socket for network communications. */
    /* The sockaddr_in IS the socket. In this case it is an Internet */
    /* socket (AF_INET) using port 7000, and accepting connections */
    /* on any network interface (INADDR_ANY) just in case the host */
    /* has multiple interfaces. */

    struct sockaddr_in server = { AF_INET, PORT, { INADDR_ANY } };

    /* The following 2 lines are here to deal with signals
         that the server can receive. the first is SIGPIPE.  If
         the server tries to send data "down the socket" but the
         process on the other end has died, or the connection was
         broken for some other reason, the server will receive a SIGPIPE
       signal. To keep it alive, it handles the signal.

         The server will also receive SIGCHLD signals when its children
         terminate.
    */

    Signal(SIGCHLD, on_sigchld);
    Signal(SIGPIPE, on_sigpipe);

    /* The socket() call creates an endpoint of communication.
       In the call below, the endpoint is for an Internet socket
       (PF_INET) of the connection-oriented type (i.e., TCP rather
       than UDP). The third parameter is the protocol. A 0 tells
       the compiler to use the default protocol for the SOCK_STREAM,
       which is TCP/IP. The socket() call returns a file descriptor
       that the process can use for listening to the socket.
    */

    if ( (listenfd = socket(PF_INET, SOCK_STREAM, 0)) == -1 ) {
        ERROR_EXIT("socket call failed",1)
    }

    /* The server now has to bind the socket file descriptor, listenfd,
       to the socket data structure. This, in effect, connects the file
       descriptor to the actual network/port address.
    */
    if ( bind( listenfd, ( struct sockaddr *) &server, SIZE ) == -1 ) {
        ERROR_EXIT(" bind call failed",1)
    }

    /* The next step for the server is to listen for incoming connection
       requests. The listen() call establishes the numberof simultaneous
       connections that the server will handle. I.e., it is the size of
       the queue of received, but not accepted requests. Here we accept
       LISTEN_QUEUE_SIZE requests.
    */
    if (listen(listenfd, LISTEN_QUEUE_SIZE ) == -1) {
        ERROR_EXIT(" listen call failed",1)
    }

    /* start the infinite loop to listen for and accept incoming */
    /* connection requests */
    for ( ; ; )
    {
        /* The accept call returns the next completed connection from the
           front of the completed connection queue. If there are no
           completed connections in the queue, the process blocks
           The accept call returns a file descriptor that can be used
           to read (recv) and/or write (send) data in the socket.
           The returned descriptor is the connected socket descriptor;
           the listening descriptor remains available to listen to the
           socket.
        */
        if ( (connectionfd = accept(listenfd, NULL, NULL ) ) == -1 ){
            if ( EINTR == errno )
                continue;
            else
                perror(" accept call failed");
        }

        switch ( fork() ) {
        case -1:
            ERROR_EXIT("fork call failed",1)
        case  0:
            /*
               The child executes this code.

               You can use the ordinary read and write calls, but
               recv and send are more flexible. recv allows peeking
               without reading, waiting for full buffers, and discarding
               all but out-of-band data.
            */
            while ( recv(connectionfd, &c, 1, 0) > 0 )
            {
                c = toupper(c);                /* convert c to upeprcase */
                send(connectionfd, &c, 1, 0 );/* send it back */
            }
```

```
                close(connectionfd);
                exit(0);
            default:
                /* server code */
                close(connectionfd);
                /* note that the server cannot wait for the child processes, */
                /* otherwise it will not return to the top of the loop to */
                /* accept new connections. Instead it has a SIGCHLD handler. */
            }
        }
}

/* if a SIGPIPE is received: */
void on_sigpipe( int sig)
{
    close(connectionfd);
    exit(0);
}

void on_sigchld( int signo )
{
    pid_t pid;
    int   status;

    while ( (pid = waitpid(-1, &status, WNOHANG) ) > 0 )
        ;
    return;
}


Sigfunc*  Signal(int signo, Sigfunc *func)
{
    struct  sigaction act, oact;

    act.sa_handler    = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags  = 0;
    if (SIGALRM != signo ) {
        act.sa_flags |= SA_RESTART;
    }
    if ( sigaction(signo, &act, &oact) < 0 )
        return ( SIG_ERR );
    return ( oact.sa_handler);
}
```

Comments.

- This program uses a user-defined `Signal()` function to encapsulate the logic of registering the signal handlers. Since we have been registering multiple handlers in most of our programs, it would have been a good idea to create this function earlier and put it in a library to reuse. The idea for this is from *[Stevens]*.
- The program could have used ordinary `read()` and `write()` system calls. Instead, as a way to

introduce two socket-specific communications primitives, it uses `recv()` and `send()`. `recv()` is one of a set of three socket-reading functions:

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

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
struct sockaddr *from, socklen_t *fromlen);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

- The `recvfrom()` and `recvmsg()` functions are the most general. The prototype for `recv()` is the same as that of `read()` except that it has a fourth argument that can be used to set various flags to control how the `recv()` behaves. The flags can be used to turn on non-blocking operation (`MSG_DONTWAIT`), to notify the kernel that the process wants to receive out-of-band data (`MSG_OOB`), or to peek at the data without reading it (`MSG_PEEK`), to name a few. In our program, no flags are used, so the fourth argument is zero, and `recv(s, buf, n, 0)` is identical to `read(s, buf, n)`.

- The `send()` function is also one of a set of three:

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

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```

- The `sendto()` and `sendmsg()` functions are the most general. The prototype of `send()` is identical to that of `write()` except for the additional argument. Like `recv()`, the fourth argument of `send()` is a set of flags that can be or-ed together. Some of the flags are the same, such as `MSG_OOB`, which allows the process to send out-of-band data. See the man page for more details. We will return to the use of `sendto()` and `recvfrom()` when we look at a connection-less server.

The code for the client is next.

```
Listing sockdemo1_client.c

#include "sockdemo1.h"

int main(int argc, char** argv)
{
    int                 sockfd;
    char                c, rc;
    char                ip_name[256] = "";
    struct sockaddr_in  server;
    struct hostent      *host;

    if ( argc < 2 )
```

```
        strcpy(ip_name, DEFAULT_HOST);
    else
        strcpy(ip_name, argv[1]);

    if ( (host = gethostbyname(ip_name)) == NULL) {
        ERROR_EXIT("gethostbyname", 1);
    }

    memset(&server, 0, sizeof(server));
    memcpy(&server.sin_addr, SOCKADDR *host->h_addr_list, SIZE);
    server.sin_family = AF_INET;
    server.sin_port   = PORT;

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) == -1 ) {
        ERROR_EXIT("socketcall failed",1)
    }

    if ( connect (sockfd, SOCKADDR &server, sizeof(server)) == -1) {
        ERROR_EXIT("connect call failed",1);
    }

    for ( rc = '\n';;) {
        if ('\n' == rc )
            printf("Input a lowercase character\n");
        c = getchar();
        write(sockfd, &c, 1);
        if ( read(sockfd, &rc, 1) > 0 )
            printf("%c", rc);
        else {
            printf("server has died\n");
            close(sockfd);
            exit(1);
        }
    }
}
```

Comments.

- The client does hostname-to-address translation to make it more generic. The user can supply the name of the server on the command line rather than having to remember the IP address. If the IP address changes, the program still works. The `gethostbyname()` call returns a pointer to a `hostent` structure, given a string that contains a valid hostname.

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);
```

- The `hostent` struct is defined in the `<netdb.h>` header file:

```
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 */
    }

    #define h_addr h_addr_list[0] /* for backward compatibility */
```

- The `h_name` field is the official name of the host. For example, if it is given the name `"eniac"` when running on a host on our network, it is able to resolve the name, and the `h_name` field will be filled in with `"eniac.geo.hunter.cuny.edu"`. The aliases member is a pointer to a list of strings, each of which is an alias, i.e., another name listed in the hosts database for the same machine. The `h_addr_list` is a pointer to a list of internet addresses for this host. If the host has just one network interface card, then only `h_addr_list[0]` is defined. Each entry is of type `in_addr`, which is why, in the client code, if can be assigned directly (with a cast) to the `sin_addr` field of the `sock_addr` structure. We do not use the `h_addrtype` or `h_length` fields here.

The client uses ordinary `read()` and `write()` calls for its I/O operations.

9.1.7 A Connection-Oriented Client Using Multiplexed I/O

Consider the client from the `upcase` example in Chapter 8. It reads a line from standard input, writes it into a pipe, and then reads the converted text from a second pipe. In that example, two pipes were needed because a pipe cannot be used as a bi-directional channel. However, we can replace the pair of pipes by a single socket, which can then be used for both sending the raw text to the server and receiving the converted text from it. In addition, by making the socket an Internet-domain socket, the client and server can be on different machines.

If we keep the original design for the client but just replace the pipes by a socket, the client would read the raw text from standard input, write it to the socket, and then read the converted text from the same socket, in a loop of the form

```
while ( true ) {
    get text from standard input;
    write text to socket;
    read converted text from socket;
    write response on standard output;
}
```

Since input can be redirected, it can arrive much faster than the responses that it receives from the server, because the server might be a long distance away. The client would spend most of its time blocked on the call to read the socket, even though both the server and the socket itself could handle much larger throughput. The same thing could happen in the interactive case as well if the user enters text very quickly but the round-trip time for the socket is large. In this case the client would be delayed in displaying a prompt to the user on the terminal. Therefore, it makes sense in this client to multiplex the standard input and the socket input using the `select()` call. By using the `select()` call, the client will only block if neither the user nor the server has data to read. As long as text arrives on the standard input stream, it will be forwarded to the server. If text arrives on standard input much faster than the round-trip time, the text will keep being sent to the server, which will process the lines one after the other and send them back in a steady stream.

An analogy will help. Imagine a thirty-person fire brigade trying to put out a fire with a single bucket. The bucket is filled with water and passed from one person to the next to the fire, poured on the fire, and then passed back to the water supply, where this is repeated. Suppose it takes one minute for the round trip and the bucket holds 5 gallons of water. This supplies 5 gallons per minute to the fire. Now suppose there are 60 buckets available. The first bucket is filled and handed to the next person, and the second bucket is filled, and so on, until all 60 buckets are filled. Assuming the people know how to pass the full buckets past the empty buckets and the exchange rate is uniform, although the round-trip time has not changed, there will be 60 buckets in the brigade at each instant, and each second, a full bucket will arrive at the fire. The fire will be supplied 5 gallons per second, or 300 gallons per minute.

This is how using `select()` can increase the throughput in the case that the bottleneck is the length of time it takes for the data to make a round trip from client to server and back. The code follows. It uses the same header file as was used in `sockdemo1`. This client will not accept a file name on the command line; it uses a single command line argument, which is the name of the host on which the server is running.

```
Listing sockdemo2_client.c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#include "sockdemo1.h"



#define  MAXFD( _x, _y)  ((_x)>(_y)?(_x):(_y))

int main(int argc, char* argv[])
{
    int                  sockfd;
    char                 ip_name[256] = "";
    fd_set               readset;
    int                  maxfd, n;
    char                 recvline[MAXLINE];
    char                 sendline[MAXLINE];
    int                  s, stdin_eof = 0;

    struct addrinfo      hints;
    struct addrinfo     *result;
    struct addrinfo     *resulting_address;
    char                 portstr[20];

    /* Check if there is a host name on command line;
       if not use default */
    if ( argc < 2 )
       strcpy(ip_name, DEFAULT_HOST);
```

```
else
   strcpy(ip_name, argv[1]);

printf("Searching for server %s\n", ip_name );

/* Initialize the hints addrinfo structure before calling
   getaddrinfo(). This is used to define criteria to use
   when it searches for a suitable host/port/service for
   the client.
*/
memset(&hints, 0, sizeof(struct addrinfo)); /* zero it out */
hints.ai_family = AF_UNSPEC;           /* allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_STREAM;       /* stream connection  */
hints.ai_flags = 0;                    /* no flags           */
hints.ai_protocol = 0;                 /* any protocol       */

/* convert numeric port to a string */
sprintf(portstr, "%d", PORT);

/* Get the network info; if non-zero return, there was an error */
if ( 0 != (s =  getaddrinfo(ip_name, portstr, &hints, &result) )  ) {
    /* call gai_strerror() to get string for error number */
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(EXIT_FAILURE);
}

/* Search through every addrinfo structure in the list pointed to by the
   result pointer in the call to getaddrinfo(). These are the structures
   containing possible family/socket-type/protocol combinations. The
   structures are ordered within the list by the rules specified in the
   RFC 3484 standard, so the first one for which a socket can be created
   is the one to use.
*/
resulting_address = result;
while ( NULL != resulting_address )  {
    sockfd = socket(resulting_address->ai_family,
                    resulting_address->ai_socktype,
                    resulting_address->ai_protocol);
    if ( sockfd == -1) {
        resulting_address = resulting_address->ai_next;
        continue;
    }
    if ( connect(sockfd, resulting_address->ai_addr,
         resulting_address->ai_addrlen) != -1)
        break;                  /* Success */

    close(sockfd);
    resulting_address = resulting_address->ai_next;
}

if ( resulting_address == NULL) {   /* No address succeeded */
    fprintf(stderr, "Could not connect\n");
    exit(EXIT_FAILURE);
}
```

```
    freeaddrinfo(result);           /* No longer needed */

    printf("Connection made to server\n");

    maxfd = MAXFD(fileno(stdin), sockfd) +1;

    for ( ;;) {
        FD_ZERO(&readset);
        if ( stdin_eof == 0 )
            FD_SET(fileno(stdin), &readset);
        FD_SET(sockfd, &readset);
        if ( select( maxfd, &readset, NULL, NULL, NULL ) > 0 ) {
            if ( FD_ISSET(sockfd, &readset)) {
                if ( ( n = read(sockfd, recvline, MAXLINE-1)) == 0 ) {
                    if (stdin_eof == 1)
                        return 0;
                    else
                        ERROR_EXIT("Server terminated prematurely.", 1);
                }
                recvline[n] = '\0';
                fputs(recvline, stdout);
            }

            if ( FD_ISSET(fileno(stdin), &readset)) {
                if ( fgets(sendline, MAXLINE-1, stdin) == NULL ) {
                    stdin_eof = 1;
                    shutdown(sockfd, SHUT_WR);
                    FD_CLR(fileno(stdin), &readset);
                    continue;
                }
                write(sockfd, sendline, strlen(sendline));
            }
        }
    }
    return 0;
}
```

Comments.

- This client sends entire lines, one after the other, to the server. It appends a null character to each line it receives before printing it to standard output, even though in principle all lines received should be null-terminated, since they are identical to the null-terminated lines that it sent to the server, except for conversion of lowercase to uppercase letters in the line.
- The `shutdown(sockfd, SHUT_WR)` system call turns off writing to the socket. When the client detects the end-of-file on the standard input stream, it cannot close the socket completely, because if it did, it would not receive any lines sent to the server but not yet converted to uppercase. On the other hand, it has to send a notification to the server that there is no more input on the socket, so that the server's `read()` on the socket can return. The `shutdown()` accomplishes this; the server's `read()` returns and the socket stays open until the server closes

its end of the socket. Without `shutdown()` there would be no way to achieve this. Its synopsis is:

```
#include <sys/socket.h>
int shutdown(int s, int how);
```

Here, the integer how can be replaced by one of `SHUT_WR`, `SHUT_RD`, or `SHUT_RDWR`.

- Once the client detects the end-of-file, it also sets a flag for itself, `stdin_eof`, which it uses to decide whether to set a bit in the descriptor mask for the standard input. If end-of-file has been detected, it stops setting that bit; otherwise it sets it. In addition, when the `read()` on the socket returns 0 bytes, it uses this flag to distinguish between two cases: whether the server has stopped sending text because there is none left to send, or there was an error on the socket before the end-of-file condition occurred.

The server's main function is displayed below. Because the signal handling code is no different in this example than in `sockdemo1_server`.c, it is not included.

```
Listing sockdemo2_server.c

#include "sockdemo1.h"
#include "sys/wait.h"

#define LISTEN_QUEUE_SIZE   5

/* The following typedef simplifies the function definition after it */
typedef void    Sigfunc(int);   /* for signal handlers */

/* override existing signal function to handle non-BSD systems */
Sigfunc*  Signal(int signo, Sigfunc *func);

/* Signal handlers */
void on_sigchld( int signo );
void str_echo(int sockfd);


int main(int argc, char **argv)
{
      int                    listenfd, connfd;
      pid_t                 childpid;
      socklen_t             clilen;
      struct sockaddr_in     client_addr, server_addr;
      void                 sig_chld(int);

    if ( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ) {
        ERROR_EXIT("socket call failed",1)
    }

      bzero(&server_addr, sizeof(server_addr));
      server_addr.sin_family      = AF_INET;
      server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
      server_addr.sin_port        = PORT;
    if ( bind( listenfd, SOCKADDR &server_addr, sizeof(server_addr))
        == -1 ) {
        ERROR_EXIT(" bind call failed",1)
    }

    if ( -1 == listen(listenfd, LISTEN_QUEUE_SIZE ) ) {
        ERROR_EXIT(" listen call failed",1)
    }

    Signal(SIGCHLD, on_sigchld);

    for ( ; ; ) {
        clilen = sizeof(client_addr);
        if ((connfd = accept(listenfd, SOCKADDR &client_addr, &clilen))
            < 0) {
            if (  EINTR == errno )
                continue;          /* back to for() */
            else
                ERROR_EXIT("accept error",1);
        }

        if ( (childpid = fork()) == 0) {     /* child process */
            close(listenfd);     /* close listening socket */
            str_echo(connfd);     /* process the request */
            exit(0);
        }
        close(connfd);              /* parent closes connected socket */
    }
}


void str_echo(int sockfd)
{
    ssize_t n;
    int     i;
    char    line[MAXLINE];

    for ( ; ; ) {
            if ( (n = read(sockfd, line, MAXLINE-1)) == 0)
            return;          /* connection closed by other end */

        for ( i = 0; i < n; i++ )
            if ( islower(line[i]))
                line[i] = toupper(line[i]);
        write(sockfd, line, n);
      }
}
```

Comments.

All of the logic is encapsulated in the `convert()` function, which the child executes. `convert()` reads the connected socket until it receives the end-of-file and then it terminates, which causes the child to exit in main.