Introduction to Network Programming

Jie Wang
Department of Computer Science
UMass Lowell

Preface

The sole purpose of this note is to provide a quick introduction to students on writing network applications using BSD socket API (Application Program Interface). But you may also find it handy as a reference on writing client-server programs. BSD socket API resides on top of the TCP/IP protocol suite. This note was used as a supplement for "91.457 Computer Security" course I taught at UMass Lowell in Spring 2002. You are encouraged to consult other books (e.g., [Com97, CS96, KP90]) for more details.

All the examples given in this note have been tested under SunOS 5.6 using the GNU C compiler by me, and under Compaq Tru64 UNIX (alpha UNIX) by me and by 91.457 students in the semester of Spring 2002. Compiling command lines are different between SunOS and Compaq Tru64 UNIX. The difference is highlighted (on page 26), and please pay close attention when you compile these programs.

This note is updated from time to time. Additional exercise problems may also be added. I welcome comments and suggestions, and I can be reached by e-mail at wang@cs.uml.edu.

Contents

1 Client-Server Programming and BSD Socket Interface

The socket API (Application Program Interface) provides a system environment for writing network application programs. It is to be used on top of the TCP/IP protocol stack. Socket API is often referred to as socket interface.

The *client-server* programming is the most common type of network applications. A *server* is a program that provides service (serves up resources) to the client program. For instance, a file server makes files available, and a display server makes a bitmapped display (and keyboard and mouse) available to client programs. A *client* is a program that accesses these resources by connecting to a server program. The server program is always in running in the background on the server site (i.e., a computer that the server program is executed).

1.1 BSD sockets

It is natural to view a socket interface as an I/O environment for network application programs, very much in the same way as we view a file interface as an I/O environment for stand-alone computer applications. While an I/O environment provides a set of system calls for creating and modifying files in a stand-alone computer, an socket interface provides a set of system calls for creating a communication channel between the client and the server in a computer network for input and output. Hence, we can think of a socket as a file in network communication.

Sockets are created using a system call (i.e., by the operating system), and each socket created by the operating system is identified by a small integer called a *socket descriptor*, very much in the same way of creating files where the operating system assigns a *file descriptor* to each file it opens. Just as we use system calls *open/read/write/close* to operate on files, we use similar system calls to operate on sockets. When an application calls `socket`, the operating system allocates a new data structure to hold the information needed for client-server communication, and fills in a new descriptor table entry to contain a pointer to the data structure of the socket.

Since we are using TCP/IP protocols, a socket data structure typically consists of the following fields: family, service, Local IP, Remote IP, Local port, and Remote port, where family specifies the protocol type, and service specifies the type of service. The operating system leaves most of the fields in the internal data structure of a socket unfilled when it created the socket. The application that creates the socket must make additional system calls to fill in information in the socket data structure before the socket can be used.

Once a socket has been created, it either waits for an incoming connection, called a *passive socket*, or initiates a connection, called an *active socket*. A server creates a passive socket, and a client creates an active socket. The active socket seeks out a server and makes an active connection to the server through its passive socket. The passive socket sits and waits for a client, waiting to be connected to an active socket.

1.2 Client-server program paradigm

A server usually consists of a sequence of system calls to perform the following tasks:

1. Allocate a communication channel (passive socket).
2. Bind the socket to a port number (for clients to make connection to).
3. Inform the operating system of any relevant parameters needed.
4. Loop, waiting for incoming connections.
5. Process each connection.

A client usually consists of a sequence of system calls to perform the following tasks:

1. Allocate a communication channel (active socket).
2. Connect the socket to a port number and host where the desired service resides.
3. Wait for the connection to complete and check for failure.
4. Perform the task.
5. Disconnect and clean up if necessary.

It is always easier and quicker to learn how to write programs in a new framework by studying a few working examples in that framework. We will dissect and analyze, as the first example, a very simple client-server application using TCP/IP (written in C on UNIX), or on any system that supports the Berkeley socket API. The program is borrowed from [Com97] and has a simple goal: The server returns the number of times it has been visited by clients. We provide a number of modifications to make testing easier. We will then provide a network echo program as a second example. The programming style of the second example, with a more general Makefile, is somewhat different from the first, and is closer to what one would expect to see in practice.

2 A Simple Counting Server

2.1 Set up

Our first problem is to specify the correct include (header) files to use the socket interface. The following include statements are needed for creating sockets.

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

In addition, we may also find other header files useful. Listed below are a set of minimal header files for writing network application programs.

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
```

Just as in writing ordinary C programs, we may also want to include the following header files to handle standard I/O (input from keyboard and output to the monitor), and to handle strings.

```
#include <stdio.h>
#include <string.h>
```

The main() routine begins with some common and key declarations:

```
#define PROTOPORT 5193        /* default protocol port number */
#define BUFSIZE   1024
#define QLEN      128         /* size of request queue        */

int visits = 0;               /* counts client connections    */

int main(int argc, char *argv[])
{
   struct protoent *ptrp;  /* pointer to a protocol table entry  */
   struct sockaddr_in sad; /* structure to hold server's address */
   struct sockaddr_in cad; /* structure to hold client's address */
   int sd, sd2;            /* socket descriptors                 */
   int port;               /* protocol port number               */
   int alen;               /* length of address                  */
   char buf[BUFSIZE];      /* buffer for data from the server    */
```

2.2 Test port

A port number is a positive integer to identify a client-server pair in running. Hence, The same port must be used in both the server and client. Low-numbered ports (i.e., with port number less than 1024), such as ftp, are only available to super-users to execute standard network applications. When writing a client-server application we often select a port number that is greater than 1024 and is unique to the programmer, so multiple users can develop network applications at the same time on the same machines. For convenience, we call such a port a *test port*. For our program, we define a test port number named PROTOPORT, which could be any number greater than 1024. But it should be a number that has not been used by other programs.

2.3 Protocol entries

The protoent structure is filled in by a library call named getprotobyname(), and related calls. Listed below are the fields of protoent.

```
struct protoent {
   char *p_name;         /* official protocol name */
   char *p_aliases[];    /* any aliases            */
   int p_proto;          /* protocol number        */
};
```

If we are to use TCP protocol (not UDP) to establish a client-server connection, then the official protocol name will be the string tcp. If there were aliases and we gave one to getprotobyname(), this call could be used to translate back to the official name. For example, the upper-case version TCP is an alias of tcp, which is listed in p_aliases. Each standard transport layer protocol is also identified with a number, called the protocol number. The protocol number expected by other system and library calls is in p_proto (in our case, the protocol number is 6 for tcp).

The protocol entry is retrieved from a system file, in this case /etc/protocols. The entry in that file for tcp looks like the following.

```
tcp          6          TCP
```

Given below is a typical protocol file, which is found on the CS UNIX server machine at UMass Lowell. (The file is /etc/protocols.)

```
# @(#)$RCSfile: protocols,v $ $Revision: 4.1.6.2 $ (DEC) $Date: 1993/07/26 20:
49:04 $
#
# Internet (IP) protocols
#
# Description: The protocols file lists the known protocols in the Internet.
#
# Syntax:  Name  Number [ alias_1,...,alias_n ] [#comments]
#
# Internet (IP) protocols
#
# Description:  The protocols file lists the known protocols in the Internet.
#
# Syntax:  Name  Number [ alias_1,...,alias_n ] [#comments]
#
# Name      official protocol name
# Number    official protocol number
# alias_n       unofficial names or abbreviations for this protocol
# #comments     text following the '#' character is a comment (ignored)
#
#

ip     0   IP      # internet protocol, pseudo protocol number
```

```
icmp    1   ICMP    # internet control message protocol
igmp    2   IGMP    # internet group management protocol
ggp     3   GGP     # gateway-gateway protocol
tcp     6   TCP     # transmission control protocol
pup     12  PUP     # PARC universal packet protocol
udp     17  UDP     # user datagram protocol
```

2.4 Service ports

In some application programs, we may need to call certain standard applications such as ftp (although it is not the case in the program we are to write here, for our program is not a standard program). Typically, but not necessarily, there is a file on UNIX systems (`/etc/services`) with entries describing well-known service ports and services. The entry for ftp looks like the following.

```
ftp         21/tcp
```

The entry says that ftp is a TCP service assigned to port 21 (decimal). Many services have been pre-assigned to particular port numbers, either by being listed in the appropriate standard documents (RFC documents), or by prevailing convention.

We can obtain the service number of a standard application by using the getservbyname() library routine. This routine returns a servent structure with the following fields filled in (or a NULL pointer if the requested information cannot be found):

```
struct servent {
   char *s_name;       /* official service name           */
   char *s_aliases[];  /* other names for service, if any */
   int s_port;         /* well-known port for service     */
   char *s_proto;      /* protocol to use                 */
};
```

For example, let us look at the following statement.

```
struct servent *svp;
svp = getservbyname("tcp", "ftp");
```

In this case, `svp->s_name` is ftp. There are no aliases `s_aliases` for ftp, so this pointer will be empty. A port number `s_port` is called *well-known* if it is listed in an RFC document as being assigned to a particular service. The ftp program is assigned to port 21. We could skip this set up and just use port number 21, but it is generally good practice to get this information from the system, just in case it changes (not all services we may use will be as immutable as ftp probably is by now). Finally, the protocol `s_proto` will be tcp.

2.5 Socket addressing

A socket, identified by an integer, is used within the UNIX kernel for I/O operations. When a socket is created, it does not contain detailed information about how it will be used. In particular, the socket does not contain information about the protocol port numbers or IP addresses of either the local machine or the remote machine. Before an application uses a socket, it must specify one or both of these addresses. TCP/IP protocols define a *communication endpoint* to consist of an IP address and a protocol port number. (Other protocol families define their endpoint addresses in other ways.)

The address structure of a socket, analogous to a file name, is used to associate a socket with a particular host and service. We will fill it in with information obtained from the service entry call and the protocol entry call, or from other information coded into the program. Although several versions have been released, the latest Berkeley code defines a sockaddr structure to have three fields:

```
struct sockaddr {
   u_char sa_len;     /* total length of the address */
   u_char sa_family;  /* family of the address       */
   char sa_data[14];  /* the address itself          */
};
```

Field `sa_len` consists of a single byte that specifies the length of the address. Field `sa_family` specifies the family to which an address belongs. The socket API provides a symbolic constant `AF_INET` for TCP/IP addresses, and we will use it later. Finally, field `sa_data` contains bytes of the address.

For each address family the address structures are generally redefined in a format convenient for that protocol. For the internet protocols this is defined in the file `<netinet/in.h>` as a `sockaddr_in`, containing the following fields.

```
struct sockaddr_in {
   u_char sin_len;           /* total length of the address    */
   u_char sin_family;        /* family of the address (type)   */
   u_short sin_port;         /* port to connect to             */
   struct in_addr sin_addr;  /* host's IP address              */
   char sin_zero[8];         /* padding (set to zero), ignored */
}
```

The first two fields of structure `sockaddr_in` corresponds exactly to the first two fields of the generic `sockaddr` structure. The last three fields define the exact form of address that TCP/IP protocols expect. There are two points to notice. First, each address identifies both a host and a particular application on that host. Field `sin_addr` contains the IP address of the host, and field `sin_port` contains the protocol port number of an application. Second, although TCP/IP needs only six bytes to store a complete address, the generic `sockaddr` structure reserves fourteen bytes. Thus, the final field in structure

`sockaddr_in` defines an 8-byte field of 0's, which pad structure to the same size as `socketaddr`. In other words, each TCP/IP endpoint address consists of a 2-byte field that identifies the address type (it must contain `AF_INET`), a 2-byte port number field, a 4-byte IP address field, and an 8-byte field that remains unused.

2.6 Numbers and networking—a slight digression

The internet protocols specify a format in which binary numbers are passed around between hosts. This format is called *network byte order*. It represents integers with the most significant byte first. It does not apply to the application data—we can pass that around any way we like—but information that is used to establish the connection must be in the correct format. Different processors store these bytes within a word differently (either in little-endian format or in big-endian format). To find out how a system stores bytes, we may run the following simple program.

```
#include <stdio.h>
/*
 * byteorder.c
 */

int main()
{
    char *p;
    long i;
    int j;

    p = (char *) & i;

    for (j = 0; j < sizeof(i); j++)
         *p++ = j+1;
    printf(" 0x0%lx\n", i);
    return 0;
}
```

On a Sun/Sparc (or RS/6000) system, the output is 0x01020304. This is network byte order. Most systems provide routines to swap these bytes to fit in different systems. Listed below are the relevant routines. Similarly, can you find out the byte order of Compaq Tru64 UNIX system?

```
htons()     Host to network order, short word
htonl()     Host to network order, long word
ntohs()     Network to host order, short word
ntohl()     Network to host order, long word
```

The numbers returned by getservbyname() and getprotobyname() are in network byte order. This means that if we want to print them out on a system with a different byte order we would need to convert them. We may use the following statement to get the network byte order of ftp.

```
printf("ftp service port is %d/n", ntohs(svp->s_port));
```

Similarly, if we supply our own port number, which is common when developing new services, we may use the following example, or something similar, to set a port number.

```
#define MYPORT       8888
     port = htons(MYPORT);
```

Because the routines are guaranteed to do the right thing on any host, even if no reordering is necessary, it is a good idea to use them in all of our program code.

2.7 Network database library calls

If we use standard applications, we need to look up the numeric port information corresponding to that program. For ftp, for example, we can use the following statement.

```
svp = getservbyname("ftp", "tcp");
```

To look up the information corresponding to the TCP protocol for use later in the program, we may use the following statement.

```
ptrp = getprotobyname("tcp");
```

2.8 Setting up the socket structure

We will fill in the socket address structure as below.

```
memset((char *)&sad, 0, sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET;             /* set family to Internet   */
sad.sin_addr.s_addr = INADDR_ANY;     /* set the local IP address */

if (argc > 1) {
   port = atoi(argv[1]);
} else {
   port = PROTOPORT;
}

if (port > 0)
   sad.sin_port = htons((u_short)port);
else {
   fprintf(stderr, "bad port number %s\n", argv[1]);
exit(1);
}
```

1. Zero out the structure using the memset library call. The purpose of this call is to make sure that any areas not used (such as the padding area at the end) are filled with 0's.

2. Set the address family, which is `AF_INET`. (The `sin_family` only communicates the address family information to the kernel, and so does not need to be corrected for byte order.)

3. Set the `s_addr` element to the system constant `INADDR_ANY`, which tells the system that we will take connections from anywhere.

If a port number is provided by the user, then use that port number; otherwise, use the default port number. In either case, we convert the port number to the correct network byte order and copy it into the socket address structure.

The socket procedure creates a socket and returns an integer descriptor:

```
descriptor = socket(protofamily, type, protocol)
```

Argument `protofamily` specifies the protocol family to be used with the socket. For example, the value `PF_INET` is used to specify the TCP/IP protocol suite, and `PF_APPLETALK` is used to specify AppleTalk protocols.

Argument `type` specifies the type of communication the socket will use. The two most common types are a connection-oriented stream transfer (specified with the value `SOCK_STREAM`), and a connectionless message-oriented transfer (specified with the value `SOCK_DGRAM`).

Argument `protocol` specifies a particular transport protocol used with the socket. Having a protocol argument in addition to a type argument allows a single protocol suite to include two or more protocols that provide the same service.

Now we use the protocol number we just acquired to get a socket and bind it to the service we plan to provide. We can do so as follows.

```
sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
```

The system constants `AF_INET` and `SOCK_STREAM` are defined in the system header file `<sys/socket.h>`, where `PF_INET` is an alias of `AF_INET`. The `ptrp->proto` element in this case will be the value indicating a TCP socket is desired. It was filled in by our earlier call to getprotobyname().

2.9 Binding a server's socket to a service

When created, a socket has neither a local address nor a remote address. A server uses the bind() procedure to supply a protocol port number at which the server will wait for contact. The bind() procedure follows the following format.

```
bind(socket, localaddr, addrlen)
```

Argument `socket` is the descriptor of a socket that has been created but not previously bound; the call is a request that the socket be assigned a particular protocol port number. Argument `localaddr` is a structure that specifies the local address to be assigned to the socket, and argument `addrlen` is an integer that specifies the length of the address.

Hence, in our case, binding a server's socket to a service can be done as below.

```
bind(sd, (struct sockaddr *)&sad, sizeof(sad));
```

The combination of these two calls, `socket()` and `bind()`, is analogous to the UNIX `open()` call. The `socket()` call allocates an object similar to a file descriptor, and `bind()` connects it to a particular object with which the operating system knows how to communicate.

2.10 Setting the backlog

After specifying a protocol port, a server must instruct the operating system to place a socket in passive mode so it can be used to wait for contact from clients. To do so, a server calls the listen() procedure, which takes two arguments:

```
listen(socket, queuesize)
```

Argument `socket` is the descriptor of a socket been created and bound to a local address, and argument `queuesize` specifies a length for the socket's request queue.

The operating system builds a separate request queue for each socket. Initially, the queue is empty. As requests arrive from clients, each is placed in the queue; when the server asks to retrieve an incoming request from the socket, the system returns the next request from the queue. If the queue is full when a request arrives, the system rejects the request.

Although calling `listen()` is not strictly necessary, it is common and expected in network server applications to set the maximum backlog permitted. Our server program might be bogged down servicing a request when, suddenly, another client tries to connect to the server. The operating system is willing to make the client wait. But there are limits to how many clients it is reasonable to keep waiting.

In our case, we can use the following `listen()` system call to set the *backlog*:

```
listen(sd, QLEN);
```

The operating system will impose some limit on how many clients may be queued up. We may also replace QLEN by a system constant SOMAXCONN. Using `listen()` we set a limit for our server application that is equal to or less than the system-imposed limit (such as the use QLEN here).

2.11 The main loop

We now enter an infinite loop awaiting new connections:

```
while (1) {
  alen = sizeof(cad);
  if ((sd2 = accept(sd, (struct sockaddr *)&cad, &alen)) < 0) {
    fprintf(stderr, "accept failed\n");
    exit(1);
  }
  visits++;
  sprintf(buf, "This server has been contacted %d time%s\n",
      visits,visits==1?".":"s.");
  send(sd2,buf,strlen(buf),0);
  close(sd2);
 }
} /* end of main */
```

The `accept()` call blocks execution until a new client comes along, much like a scanf call that causes the system to wait for the user to enter new input.

The return value of accept is a new socket for communicating with the newly arrived client. The socket address structure passed to `accept()` is filled with host address information for the remote end of the connection.

Note: In other application programs, we may replace the counting and the printing statements by our own `doit()` routine.

2.12 The send, sendto, and sendmsg procedure

Both clients and servers may need to send information. Usually, a client sends a request, and server sends a response. If the socket is connected, procedure send can be used to transfer data, which has the following format.

```
send(socket, data, length, flags)
```

Arguments `socket` is the descriptor of a socket to use, `data` is the address in memory of the data to send, `length` is an integer that specifies the number of octets of data, and `flags` contains bits that request special options. (Many options are intended for system debugging, and are not available to conventional client and server programs.) In our case, we have used send() in the following way.

```
send(sd2,buf,strlen(buf),0)
```

Procedures sendto and sendmsg allow a client or server to send a message using an unconnected socket. (For example, when using the UDP protocol suite, these two system calls would become handy.) The sendto procedure has the following form.

```
sendto(socket, data, length, flags, destaddress, addresslen)
```

The first four arguments correspond to the four argument of the send subroutine. The final two arguments specify the address of a destination and the length of that address. The form of the address in argument `destaddress` is the `sockaddr` structure (or the `sockaddr_in` structure when used with TCP/IP).

The sendmsg subroutine performs the same operation as sendto, but abbreviates the arguments by defining a structure. This may a statement easier to read (and write, too). It has the following form.

```
sendmsg(socket, msgstruct, flags)
```

Argument `msgstruct` is a structure that contains information about the destination address, the length of the address, the message to be sent, and the length of the message.

2.13 The closing procedure

The close procedure tells the system to terminate an individual socket. It prevents the application from sending more data, and the transport protocol stops accepting incoming messages directed to the socket, preventing the application from receiving more data.

2.14 Exercise problems

1. Explore the protocol entries on the host you logon. What is the protocol name and number of the IPv6 internet control message protocol?

2. Find out the standard network application names and their port numbers on the host you logon. What is the port number of telnet?

3. Find out the byte order used on Alpha machines and Pentium machines.

3 A Simple Counting Client

3.1 Set up

The set up of a client is similar to the set up of a server with a few exception. In particular, we need to get host information of the computer that runs the server, and establish a new connection for each new request from the client (and close the connection when service is provided and finished). We may still use the same set of header files as for the server, but we need to include a few extra declarations, including hostent and the default host location.

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

```
#include <arpa/inet.h>
#include <netdb.h>

#include <stdio.h>
#include <string.h>

#define PROTOPORT 5193        /* default protocol port number */
#define BUFSIZE 1024

char localhost[] = "localhost"; /* default host name */

int main(int argc, char *argv[])
{
  struct hostent *ptrh;   /* pointer to a host table entry      */
  struct protoent *ptrp;  /* pointer to a protocol table entry  */
  struct sockaddr_in sad; /* structure to hold server's address */
  int sd;                 /* socket descriptor                  */
  int port;               /* protocol port number               */
  char *host;             /* pointer to host name               */
  int n;                  /* number of characters read          */
  char buf[BUFSIZE];      /* buffer for data from the server    */

  memset ((char *)&sad, 0, sizeof(sad)); /* clear sockaddr structure */
  sad.sin_family = AF_INET;              /* set family to Internet   */

  /* Check command-line argument for protocol port and extract
   * port number if one is specified. Otherwise, use the default
   * port value given by constant PROTOTYPE
   */

  switch (argc) {
  case 1:
    break;
  case 2:
    port = atoi(argv[1]); /* get port number */
    if (port <= 0) {
       fprintf(stderr, "bad port number %s\n", argv[1]);
       exit(1);
    }
    break;
  case 3:
    port = atoi(argv[2]); /* get port number */
    if (port <= 0) {
```

```
        fprintf(stderr, "bad port number %s\n", argv[2]);
        exit(1);
    }
    host = argv[1];       /* get host name */
    break;
  default:
    fprintf(stderr, "usage: visit [host [port]]\n");
    fprintf(stderr, "usage: visitd [port]\n");
    break;
  }

  sad.sin_port = htons((u_short)port);
```

The client expects one or two command line arguments, a host's domain name (or a host's numeric name), and a user a port number of the server on the server computer. We make these two arguments as optional inputs by using a default host name and a default port number.

3.2 Looking up a domain name

A client must specify the address of a server using structure `sockaddr_in`. By doing so it means converting an address in dotted decimal notation (or a domain name in text form) into a 32-bit IP address represented in binary. Converting from dotted decimal notation to binary is trivial. Converting from a domain name, however, requires considerably more effort. The socket interface in BSD UNIX includes library routines, `inet_addr()` and `gethostbyname()`, that perform the conversions. Function `inet_addr()` takes an ASCII string that contains a dotted decimal address and returns the equivalent IP address in binary. Function `gethostbyname()` takes and ASCII string that contains the domain name for a machine, and returns the address of a `hostent` structure that contains, among other things, the host's IP address in binary. The `hostent` structure is declared in `netdb.h`, which has the following form.

```
struct hostent {
   char  *h_name;       /* official host name */
   char **h_aliases;    /* other aliases      */
   int    h_addrtype;   /* address type       */
   int    h_length;     /* address length     */
   char **h_addr_list;  /* list of addresses  */
};
```

Hosts often have more than one name. The first element, `h_name`, is the official name of the host, and the `h_aliases` array will contain all its other names. We may pass `gethostbyname()` any name for the host that appears in either the `h_name` or `h_aliases` fields.

The address type is intended to distinguish between address families: Internet, ISO, DECnet, etc. The `h_length` filed gives the length in bytes of the binary address. Finally, the `h_addr_list` is an array of pointers to binary addresses for the host. Many hosts have more than one address, as they act as gateways between two or more networks. What is nearly impossible to know, from the application, is which of these addresses is the best to use. In some cases, only one address may be reachable from the client's host. One approach to this dilemma will be to simply try each address until one works. This method does not guarantee choosing an optimal address. (The address we chose would imply a route by which our packets travel.)

Consider a simple example of name conversion. Suppose a client has been passed the domain name host1.cs.uml.edu in string form and needs to obtain the IP address. The client can call `gethostbyname()` as follows.

```
struct hostent *hptr;
char   *examplenam = ‘‘host1.cs.uml.edu’’;

if (hptr = gethostbyname(examplenam)) {
      /* IP address is now in hptr->h_addr */
} else {
      /* error in name - handle it */
}
```

Using the `gethostbyname()` routine, we have the following program fragment.

```
/* Convert host name to equivalent IP address and copy to sad. */
ptrh = gethostbyname(host);
if (((char *)ptrh) == NULL) {
   fprintf(stderr, "invalid host: %s\n", host);
   exit(1);
}
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

/* Map TCP transport protocol name to protocol number. */

if (((int)(ptrp = getprotobyname("tcp"))) == 0) {
   fprintf(stderr, "cannot map \"tcp\" to protocol number");
   exit(1);
}

/* Create a socket */
   sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
   if (sd < 0) {
   fprintf(stderr, "socket creation failed\n");
   exit(1);
```

```
}

/* Connect the socket to the specified server */
   if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
   fprintf(stderr, "connect failed\n");
   exit(1);
}
```

The setup using getprotobyname() and getservbyname() is identical to that of the server, and we will not repeat the details. Both of these procedures are trying to access the same service using the same protocol and the same service. The initialization of the socket address structure is also similar, except for the initialization of the socket address element. We will not use a wildcard (`INADDR_ANY`), but instead need to give a specific host address to query.

3.3 Numeric addresses

People may specify host addresses in a numeric, dotted-digit format rather than by the host's domain name. It is generally expected that an application that takes domain names will also take numeric, dotted-digit format (such as 129.63.8.2, the IP address of saturn.cs.uml.edu). A good way to check for a numeric host number specified as a string versus a (possibly) host name is to call `inet_addr(string)`, where `string` is the string you want checked. The routine will return the value -1 if the string is not a proper address, and we should then proceed to assume it is a host name, and let **gethostbyname()** decide if it really is or not. One advantage to using `inet_addr()` rather than checking the string ourselves is that the function understands some variations, which will treat 18.1 as 18.0.0.1.

3.4 Connect

There is no bind() or listen() call in the client; these are normally only used by servers. Instead, the client uses procedure connect() to establish connection with a specific server, which is analogous to bind() used in the server. The form of connect() is as follows.

```
connect(socket, saddress, saddresslen)
```

Arguments `socket` is the descriptor of a socket on the client's computer to use for the connection, `saddress` is a `sockaddr` structure that specifies the server's address and protocol port number, and `saddresslen` specifies the length of the server's address measured in octets. In our case, we have used the following statement for connection.

```
connect(sd, (struct sockaddr *)&sad, sizeof(sad))
```

When connect() succeeds in returning a nonnegative value, the socket is connected through the network to a server. All that is left is to use ordinary UNIX I/O calls

to read data from the socket and echo it to the user's standard output (probably, but not necessarily, the terminal). As with file I/O, returning from a read with the value 0 indicates that there is no more data to be read. We then close the socket and exit. The remaining code of the client is the main loop for getting data.

3.5 The main loop

```
    /* Repeatedly read data from socket and write to user's screen */
    n = recv(sd, buf, sizeof(buf), 0);
    while (n > 0) {
       write(1, buf, n);
       n = recv(sd, buf, sizeof(buf),0);
    }

    /* Close the socket */
    close(sd);

    /* Terminate the client program gracefully */

    return 0;
}
```

3.6 The recv, recvfrom, and recvmsg procedures

A client and a server each need to receive data sent by the other. The socket API provides several procedures that can be used. For example, an application can call recv to receive data from a connected socket. The procedure has the following form, which is the counter part of the send procedure.

```
recv(socket, buffer, length, flags)
```

Argument `socket` is the descriptor of a socket from which data is to be received. Argument `buffer` specifies the address in memory in which the incoming messages should be placed, and argument `length` specifies the size of the buffer. Finally, argument `flags` allows the caller to control details (e.g., to allow an application to extract a copy of an incoming message without removing the message from the socket).

If a socket is not connected (as in the case of using the UDP protocol), it can be used to receive messages from an arbitrary set of clients. In such cases, the system returns the address of the sender along with each incoming message. Procedures recvfrom and recvmsg are counter parts of sendto and sendmsg, respectively. The recvfrom procedure has the following form.

```
recvfrom(socket, buffer, length, flags, sndraddr, saddrlen)
```

And the recvmsg procedure has the following form.

```
recfmsg(socket, msgstruct, flags)
```

More information about these system calls can be found in Section 6.

4 The Complete Counting Server-Client Code

4.1 The server

```
/*------------------------------------------------------------------
 * Filename: visitd.c --- server program
 * Purpose:  allocate a socket and then repeatedly execute the following:
 *           (1) wait for the next connection from a client
 *           (2) send a short message to the client
 *           (3) close the connection
 *           (4) go back to step 1
 * Syntax:   visitd [port]
 *           port --- protocol port number to use
 * Note:     The port argument is optional. If no port is specified,
 *           the server uses the default given by PROTOPORT
 *------------------------------------------------------------------
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <stdio.h>
#include <string.h>

#define PROTOPORT 5193        /* default protocol port number */
#define BUFSIZE 1024
#define QLEN 128              /* size of request queue        */

int visits = 0;               /* counts client connections    */

int main(int argc, char *argv[])
{
  struct protoent *ptrp;  /* pointer to a protocol table entry  */
  struct sockaddr_in sad; /* structure to hold server's address */
  struct sockaddr_in cad; /* structure to hold client's address */
  int sd, sd2;            /* socket descriptors                 */
```

```
int port;               /* protocol port number               */
int alen;               /* length of address                  */
char buf[BUFSIZE];      /* buffer for data from the server    */

memset((char *)&sad, 0, sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET;             /* set family to Internet   */
sad.sin_addr.s_addr = INADDR_ANY;     /* set the local IP address */


/* Check command-line argument for protocol port and extract number
 * if one is specified. Otherwise, use the default port value
 */
if (argc > 1) {
  port = atoi(argv[1]);
} else {
  port = PROTOPORT;
}

if (port > 0)
  sad.sin_port = htons((u_short)port);
else {
  fprintf(stderr, "bad port number %s\n", argv[1]);
  exit(1);
}

/* Map TCP transport protocol name to protocol number. */
if (((int)(ptrp = getprotobyname("tcp"))) == 0) {
  fprintf(stderr, "cannot map \"tcp\" to protocol number");
  exit(1);
}

/* Create a socket */
sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0) {
  fprintf(stderr, "socket creation failed\n");
  exit(1);
}

/* Bind a local address to the socket */
if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
  fprintf(stderr, "bind failed\n");
  exit(1);
}
```

```
  /* Specify size of request queue */
  if (listen(sd, QLEN) < 0) {
    fprintf(stderr, "listen failed\n");
    exit(1);
  }

  /* Main server loop --- accept and handle requests */
  while (1) {
    alen = sizeof(cad);
    if ((sd2 = accept(sd, (struct sockaddr *)&cad, &alen)) < 0) {
      fprintf(stderr, "accept failed\n");
      exit(1);
    }
    visits++;
    sprintf(buf, "This server has been contacted %d time%s\n",
        visits,visits==1?".":"s.");
    send(sd2,buf,strlen(buf),0);
    close(sd2);
  }
}
```

4.2 The client

```
/*-------------------------------------------------------------------
 * Filename: visit.c --- client software
 * Purpose:  allocate a socket, connect to a server, and print the number
 *           of visits
 * Syntax:   visit [host] [port], or visit [port]
 *           host --- name of a computer on which server is executing
 *           port --- service port number server is using
 * Note:     Both arguments are optional. If no host name is specified,
 *           the client uses "localhost"; if no protocol port is
 *           specified, the client uses the default given by PROTOPORT.
 *-------------------------------------------------------------------
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
```

```
#include <stdio.h>
#include <string.h>

#define PROTOPORT 5193        /* default protocol port number */
#define BUFSIZE 1024

char localhost[] = "localhost"; /* default host name */

int main(int argc, char *argv[])
{
  struct hostent *ptrh;   /* pointer to a host table entry      */
  struct protoent *ptrp;  /* pointer to a protocol table entry  */
  struct sockaddr_in sad; /* structure to hold server's address */
  int sd;                 /* socket descriptor                  */
  int port;               /* protocol port number               */
  char *host;             /* pointer to host name               */
  int n;                  /* number of characters read          */
  char buf[BUFSIZE];      /* buffer for data from the server    */

  memset ((char *)&sad, 0, sizeof(sad)); /* clear sockaddr structure */
  sad.sin_family = AF_INET;  /* set family to Internet */

  /* Check command-line argument for protocol port and extract
   * port number if one is specified. Otherwise, use the default
   * port value given by constant PROTOTYPE
   */

  port = PROTOPORT;
  host = localhost;

  switch (argc) {
  case 1:
    break;
  case 2:
    port = atoi(argv[1]); /* get port number */
    if (port <= 0) {
      fprintf(stderr, "bad port number %s\n", argv[1]);
      exit(1);
    }
    break;
  case 3:
    port = atoi(argv[2]); /* get port number */
    if (port <= 0) {
```

```
    fprintf(stderr, "bad port number %s\n", argv[2]);
    exit(1);
  }
  host = argv[1];       /* get host name */
  break;
default:
  fprintf(stderr, "usage: visit [host [port]]\n");
  break;
}

sad.sin_port = htons((u_short)port);

/* Convert host name to equivalent IP address and copy to sad. */
ptrh = gethostbyname(host);
if (((char *)ptrh) == NULL) {
  fprintf(stderr, "invalid host: %s\n", host);
  exit(1);
}
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

/* Map TCP transport protocol name to protocol number. */

if (((int)(ptrp = getprotobyname("tcp"))) == 0) {
  fprintf(stderr, "cannot map \"tcp\" to protocol number");
  exit(1);
}

/* Create a socket */
sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0) {
  fprintf(stderr, "socket creation failed\n");
  exit(1);
}

/* Connect the socket to the specified server */
if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
  fprintf(stderr, "connect failed\n");
  exit(1);
}

/* Repeatedly read data from socket and write to user's screen */
n = recv(sd, buf, sizeof(buf), 0);
while (n > 0) {
```

```
    write(1, buf, n);
    n = recv(sd, buf, sizeof(buf),0);
  }

  /* Close the socket */
  close(sd);

  /* Terminate the client program gracefully */

  return 0;
}
```

4.3 Stream service and multiple recv calls

Although the server makes only one call to the send procedure to transmit data, the client code iterates to receive data. During each iteration, the client calls the recv procedure to obtain data; the iteration stops when the client obtains an end-of-file condition (i.e. a count of zero).

4.4 Compiling and running process

We may use the following command line to compile `visit.c` on SunOS. **On Compaq Tru64 UNIX, the two links (i.e., `-lnsl -lsocket`) are not needed**. The compiling of `visitd.c` is similar.

```
gcc visit.c -lnsl -lsocket
```

In general, we can create a Makefile file to make the compiling process easier, particularly when several files are involved. Note: Use the tab key for indention.

```
# filename: Makefile

CC = gcc
LIBS = -lnsl -lsocket

all:

visit: visit.o
        gcc  $(CFLAGS) -o $@ visit.o $(LIBS)
visitd: visitd.o
        gcc  $(CFLAGS) -o $@ visitd.o $(LIBS)

visit.o: visit.c
visitd.o: visitd.c
```

```
# DO NOT DELETE THIS LINE -- make depends on it.
```

To compile, type the following lines.

```
make visit
make visitd
```

To run the application, on the server's computer (assume that the host name is host1.cs.uml.edu), type

```
visitd 4321 &
```

Here 4321 is a port number. On the client's computer, type

```
visit host1.cs.uml.edu 4321
```

If the server and the client are running on the same machine, type

```
visitd 9876 &
visit 9876
```

We may also simply use the default port number by typing

```
visitd &
visit
```

5 Framework of TCP Client and Server Programs

5.1 Using socket calls in a program

Figure 1 illustrates a sequence of system calls made by a client and a server using TCP.

5.2 The TCP client framework

Building client software is usually easier than building server software. Because TCP handles all reliability and flow control problems, building a client that uses TCP is the most straightforward of all network programming tasks. A TCP client follows the following framework to form a connection to a server and communicate with it.

1. Find the IP address and protocol port number of the server with witch communication is desired.
2. Allocate a socket.
3. Specify that the connection needs an arbitrary, unused protocol port on the local machine, and allow TCP to choose one.

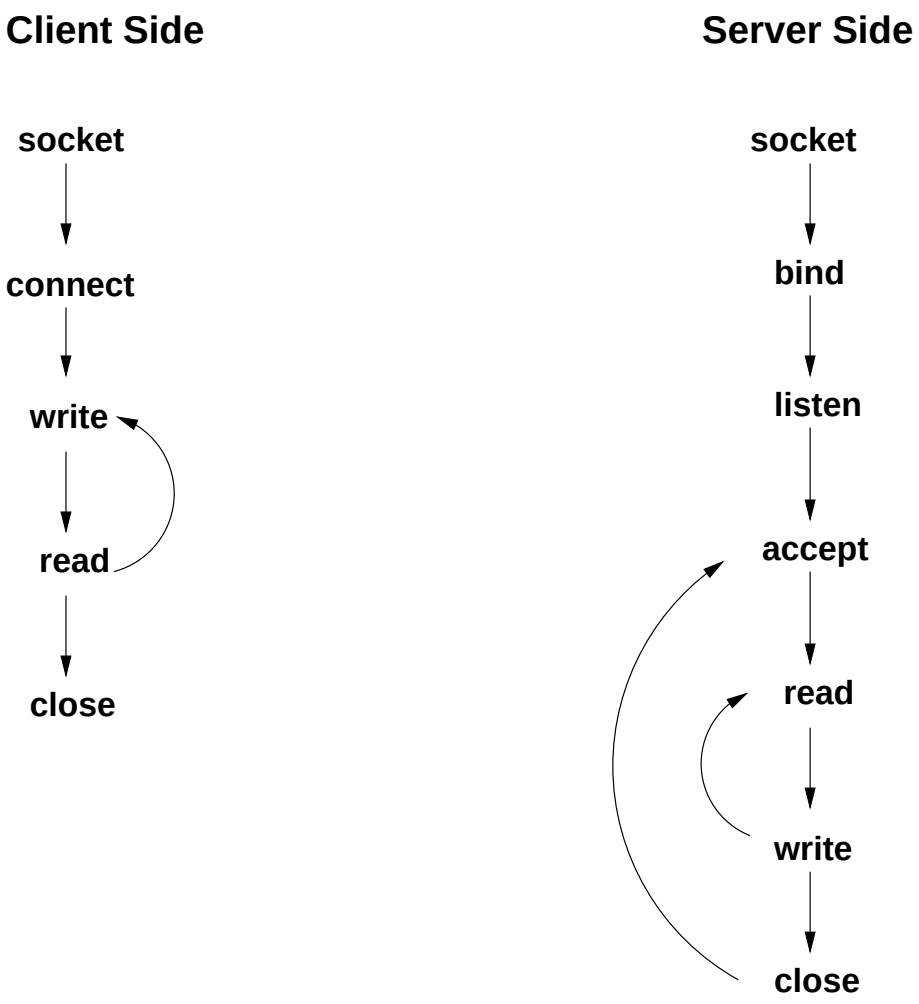

Figure 1: An example sequence of socket API system calls made by a client and server using TCP. The server runs forever. It waits for a new connection on the well-known port, accepts the connection, interacts with the client, and then closes the connection one by one.

4. Connect the socket to the server.

5. Communicate with the server using the application-level protocol (this usually involves sending requests and awaiting replies).

6. Close the connection.

5.3 The TCP server framework

Presented below is the framework for an iterative server accessed via the TCP connection-oriented transport.

1. Create a socket and bind to the well-known address for the service being offered.

2. Place the socket in passive mode, making it ready for use by a server.

3. Accept the next connection request from the socket, and obtain a new socket for the connection.

4. Repeatedly read a request from the client, formulate a response, and send a reply back to the client according to the application protocol.

5. When finished with a particular client, close the connection and return to step 3 to accept a new connection.

6 Socket API System Calls

We provide in this section a brief summary of the system functions related to BSD sockets. Unless otherwise stated, all functions can be used in both client and server programs. Note that when an error occurs in each of the following system calls, the global variable `errno` will contain information indicating what type of error has occurred. For more information on system calls provided by the BSD Socket API, the reader is referred to Appendix 1 in [CS96].

6.1 socket

The socket system call creates a descriptor for use in network communication.

- **Type**: `int socket(int family, int type, int protocol)`
- **Use**: `retcode = socket(family, type, protocol);`
- **Description**: The socket system call creates a socket used for network communication, and returns an integer descriptor for that socket.

- **Arguments**:

  `family`—Protocol or address family (`PF_INET` for TCP/IP, `AF_INET` can also be used).

  `type`—Type of service (`SOCK_STREAM` for TCP or `SOCK_DGRAM` for UDP).

  `protocol`—Protocol number to use or 0 to request the default for a given family and type.

- **Return code**: a socket descriptor or -1 if an error has occurred.

6.2 connect

The connect system call connects (a client program) to a remote peer (a server program).

- **Type**: `int connect(int socket, sockaddr_in *addr, int addrlen)`
- **Use**: `retcode = connect(socket, addr, addrlen);`
- **Description**: The connect function allows the caller to specify the remote endpoint address for a previously created socket. If the socket uses TCP, connect uses the 3-way handshake to establish a connection; if the socket uses UDP, connect specifies the remote endpoint but does not transfer any datagrams to it.
- **Arguments**:

  `socket`—The descriptor of a socket.

  `addr`—The remote machine endpoint.

  `addrlen`—The length of the second argument.

- **Return code**: 0 if successful and -1 if an error has occurred.

6.3 write

The write system call sends outgoing data across a connection.

- **Type**: `int write(int socket, char *buf, int buflen)`
- **Use**: `retcode = write(socket, buf, buflen);`
- **Description**: The write system call permits an application to transfer data to a remote machine.
- **Arguments**:

  `socket`—A socket descriptor created by the socket call.

  If you want to directly display the data on the client's screen, use the value 1 for this argument. namely, use `write(1, buf, buflen)`. Another altenative is to use the following sequence:

```
write(socket, buf, buflen);
fputs(buf, stdout);
```

`buf`—The address of a buffer containing data.

`buflen`—The number of bytes in `buf`.

- **Return code**: the number of bytes transferred if successful and -1 if an error has occurred.

6.4 read

The read system call acquires incoming data from a connection.

- **Type**: `int read(int socket, char *buff, int buflen)`
- **Use**: `retcode = read(socket, buff, buflen);`
- **Description**: Clients or servers use read to obtain input from a socket.
- **Arguments**:

  `socket`—A socket descriptor created by the socket function.

  `buff`—A pointer to an array of characters (i.e., a string) to hold the input.

  `buflen`—An integer that specifies the number of bytes in the `buff` array.

- **Return code**: The read function returns 0 if it detects an end-of-file condition on the socket, the number of bytes if it obtains input, and -1 if an error has occurred.

6.5 close

The close system call terminates communication and de-allocates a descriptor.

- **Type**: `int close(int socket)`
- **Use**: `retcode = close (socket);`
- **Description**: An application calls close after it finishes using a socket. The close system call terminates communication gracefully and removes the socket. Any unread data waiting at the socket will be discarded.

  In practice, UNIX implements a reference count mechanism to allow multiple processes to share a socket. If n processes share a socket, the reference count will be n. The close function decrements the reference count each time a process calls it. Once the reference count reaches zero (i.e., all processes have called close), the socket will be deallocated.

- **Arguments**:

  `socket`—The descriptor of a socket to be closed.

- **Return code**: 0 if successful and -1 if an error occurred.

6.6 bind

The bind system call binds a local IP address and protocol port (an endpoint address) to a socket.

- **Type**: `int bind(int socket, sockaddr *localaddr, int addrlen)`
- **Use**: `retcode = bind(socket, localaddr, addrlen);`
- **Description**: The bind system call specifies a local IP address and protocol port number for a socket. It is primarily used by servers, which need to specify a well-known protocol port.
- **Arguments**:

  `socket`—A socket descriptor created by the socket call.

  `localaddr`—The address of a structure that specifies an IP address and protocol port number.

  `addrlen`—The size of the address structure in bytes.

- **Return code**: 0 if successful and -1 if an error has occurred.

6.7 listen

The listen system call places the socket in passive mode and set the number of incoming TCP connections the system will enqueue (for a server).

- **Type**: `int listen(int socket, int queuelen)`
- **Use**: `retcode = listen(socket, queuelen);`
- **Description**: Servers use listen to make a socket passive (i.e., ready to accept incoming requests). The listen function also sets the number of incoming connection requests that the protocol software should enqueue for a given socket while the server handles another request. (Note: listen only applies to sockets used with TCP.)
- **Arguments**:

  `socket`—A socket descriptor created by the socket call.

  `queuelen`—The size of the incoming connection request queue (usually up to a maximum of 5).

- **Return code**: 0 if successful and -1 if an error has occurred.

6.8 accept

The accept system call accepts the next incoming connection (for a server).

- **Type**: `int accept(int socket, sockaddr *addr, int *addrlen)`
- **Use**: `retcode = accept(socket, addr, addrlen);`
- **Description**: Servers use the accept function to accept the next incoming connection on a passive socket after they have called socket to create a socket, bind to specify a local IP address and protocol port number, and listen to make the socket passive and to set the length of the connection request queue. The accept system call removes the next connection request from the queue (or waits until a connection request arrives), creates a new socket for the request, and returns the descriptor for the new socket. Note that accept only applies to stream socket (e.g., those used with TCP).
- **Arguments**:

  `socket`—A socket descriptor created by the socket function.

  `addr`—A pointer to an address structure. The accept function fills in the structure with the IP address and protocol port number of the remote machine.

  `addrlen`—A pointer to an integer that initially specifies the size of the `sockaddr` argument and, when the call returns, specifies the number of bytes stored in argument `addr`.
- **Return code**: a nonnegative socket descriptor if successful and -1 if an error has occurred.

6.9 recv

The recv system call receives the next incoming datagram.

- **Type**: `int recv(int socket, char *buffer, int length, int flags)`
- **Use**: `retcode = recv(socket, buffer, length, flags);`
- **Description**: The recv system call obtains the next message from a socket.
- **Arguments**:

  `socket`—A socket descriptor created by the socket system call.

  `buffer`—The address of a buffer to hold the message.

  `length`—The length of the buffer.

  `flags`—Control bits that specify whether to receive out-of-band data and whether to look ahead for messages.

- **Return code**: the number of bytes in the message if successful and −1 if an error has occurred.

6.10 recvmsg

The recvmsg system call receives the next incoming datagram (this is a variation of recv).

- **Type**: `int recvmsg(int socket, msghdr *msg, int flags)`, where `msghdr` is a structure with the following format:

```
struct msghdr {
   caddr_t        msg_name;          /* optional address        */
   int            msg_namelen;       /* size of address         */
   struct iovec  *msg_iov;           /* scatter/gather array    */
   int            mst_iovlen;        /* # elements in msg_iov   */
   caddr_t        msg_accrights;     /* rights sent/received    */
   int            msg_accrghtslen;   /* length of prev. field   */
};
```

- **Use**: `retcode = recvmsg(socket, msg, flags);`

- **Description**: The recvmsg system call returns the next message that arrives on a socket. It places the message in a structure that includes a header along with the data.

- **Arguments**:

  `socket`—Socket descriptor created by the socket system call.

  `msg`—Address of a message structure.

  `flags`—Control bits that specify out-of-band data or message look-ahead.

- **Return code**: the number of bytes in the message if successful and −1 if an error has occurred.

6.11 recvfrom

The recvfrom system call receives the next incoming datagram and record its source endpoint address.

- **Type**: `int recvfrom(int socket, char *buffer, int buflen, int flags, sockaddr *from, int *fromlen)`

- **Use**: `retcode = recvfrom(socket, buffer, buflen, flags, from, fromlen);`

- **Description**: The recvfrom system call extracts the next message that arrives at a socket and records the sender's address (enabling the caller to send a reply).

- **Arguments**:

  `socket`—A socket descriptor created by the socket system call.

  `buffer`—The address of a buffer to hold the message.

  `buflen`—The length of the buffer.

  `flags`—Control bits that specify out-of-band data or message look-ahead.

  `from`—The address of a structure to hold the sender's address.

  `fromlen`—The length of the `from` buffer, returned as the size of the sender's address.

  Section 2.5 describes the `sockaddr` structure.

- **Return code**: the number of bytes in the message if successful and -1 if an error has occurred.

6.12 send

The send system call sends an outgoing datagram.

- **Type**: `int send(int socket, char *msg, int msglen, int flags)`
- **Use**: `retcode = send(socket, msg, msglen, flags);`
- **Description**: Applications call send to transfer a message to another machine.
- **Arguments**:

  `socket`—Socket descriptor created by the socket system call.

  `msg`—A pointer to the message.

  `msglen`—The length of the message in bytes.

  `flags`—Control bits that specify out-of-band data or message look-ahead.

- **Return code**: the number of bytes sent if successful and -1 if an error has occurred.

6.13 sendmsg

The sendmsg system call sends an outgoing datagram (this is a variation of send).

- **Type**: `int sendmsg(int socket, msghdr *msg, int flags)`
- **Use**: `retcode = sendmsg(socket, msg, flags);`
- **Description**: The sendmsg sends a message, extracting it from a `msghdr` structure.

- **Arguments**:

  `socket`—Socket descriptor created by the socket system call.

  `msg`—A pointer to the message structure.

  `flags`—Control bits that specify out-of-band data or message look-ahead.

- **Return code**: the number of bytes sent if successful and -1 if there is an error.

6.14 sendto

The sendto system call sends an outgoing datagram, usually to a pre-recorded endpoint address.

- **Type**: `int sendto(int socket, char *msg, int msglen, int flags, sockaddr *to, int tolen)`

- **Use**: `retcode = sendto(socket, msg, msglen, flags, to, tolen):`

- **Description**: The sendto system call sends a message by taking the destination address from a structure.

- **Arguments**:

  `socket`—Socket descriptor created by the socket system call.

  `msg`—A pointer to the message.

  `msglen`—The length of the message in bytes.

  `flags`—Control bits that specify out-of-band data or message look-ahead.

  `to`—A pointer to the address structure.

  `tolen`—The length of the address in bytes.

- **Return code**: the number of bytes sent in a message if successful and -1 if an error has occurred.

6.15 shutdown

The shutdown system call terminates a TCP connection in one or both directions.

- **Type**: `int shutdown(int socket, int direction)`

- **Use**: `retcode = shutdown(socket, direction);`

- **Description**: The shutdown system call applies to connected TCP sockets, and is used to partially close the connection.

- **Arguments**:

  `socket`—A socket descriptor created by the socket system call.

  `direction`—The direction in which shutdown is desired: 0 means terminate further input, 1 means terminate further output, and 2 means terminate both input and output.

- **Return code**: 0 if successful and -1 if an error has occurred.

6.16 getpeername

The getpeername system call obtains the remote machine's endpoint address from a socket, after a connection arrives.

(Note that functions `gethostbyname` and `getservbyname` are library calls)

- **Type**: `int getpeername(int socket, sockaddr *remaddr, int *addrlen)`
- **Use**: `retcode = getpeername(socket, remaddr, addrlen);`
- **Description**: An application uses getpeername to obtain the remote endpoint address for a connected socket. Usually, a client knows the remote endpoint address because it calls connect to set it. However, a server that uses accept to obtain a connection may need to interrogate the socket to find out the remote address.
- **Arguments**:

  `socket`—A socket descriptor created by the socket system call.

  `remaddr`—A pointer to a `sockaddr` structure that will contain the endpoint address.

  `addrlen`—A pointer to an integer that contains the length of the second argument initially, and the actual length of the endpoint address upon return.

  Section 2.5 describes the `sockaddr` structure.

- **Return code**: 0 if successful and -1 if an error has occurred.

7 Concurrent Server Programs

In order to provide faster response time to each individual client in the case when multiple clients are requesting the same service, we are going to introduce the concept and mechanism of master-slave programming, which is a common form of concurrent programming in the C environment. In particular, if forming a response requires significant I/O, or the processing time required varies dramatically among requests, or the server executes on a computer with multiple processors, then concurrency could greatly improve response time.

The BSD socket API is designed to work with concurrent programs. The details depend on the underlying operating system, but implementations of the socket API adhere to the following principle.

> Each new thread that is created inherits a copy of all open sockets from the thread that created it.

7.1 fork—a concurrency mechanism

The system call fork is used in the C language to provide concurrency. In essence, fork divides the running program into two (almost) identical processes, both executing at the same place in the same code. The two processes continue just as if two users had simultaneously started two copies of the application. To illustrate the usage of fork, let us look at a simple example.

The following code is a conventional C program that prints the integers from 1 to 5 along with their sum.

```
/* sum.c - A conventional C program that sums integers from 1 to 5 */
#include <stdlib.h>
#include <stdio.h>
int sum;

main()
{
    int i;
    sum = 0;
    for (i = 1; i <= 5; i++) {
       printf("The value of i is %d\n", i);
       fflush(stdout);
       sum += i;
    }
    printf("The sum is %d\n", sum);
    exit(0);
}
```

When executed, the program emits six lines of output:

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
```

The following modified version of the above example calls fork to create a new process.

```
#include <stdlib.h>
#include <stdio.h>
int sum;

main()
{
    int i;
    sum = 0;
    fork();             /* create a new process */
    for (i = 1; i <= 5; i++) {
       printf("The value of i is %d\n", i);
       fflush(stdout);
       sum += i;
    }
    printf("The sum is %d\n", sum);
    exit(0);
}
```

When a user executes the concurrent version of the program, the system begins with a single process executing the code. However, when the process reaches the call to fork, the system duplicates the process and allows both the original process and the newly created process to execute. Each process has its own copy of the variables that the program uses. The easiest way to envision what happens is to imagine that the system makes a second copy of the entire running program. Then imagine that both copies run (just as if two users had both simultaneously executed the program).

One one particular uniprocessor system, the execution of the example produces twelve lines of output.

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
```

In practice, the process created by fork is not intended to be absolutely identical to the original process: It differs in one small detail. In other words, we would like the two processes to perform different tasks based on a given condition.

The system function fork returns a value to its caller. When the function call returns, the value returned to the original process differs from the value returned to the newly created process. In the newly created process, the fork returns zero; in the original process, fork returns a small positive integer that identifies the newly created process. Technically, the value returned is called a *process identifier* or *process id* (*pid*, in short).

Concurrent programs use the value returned by fork to decide how to proceed. In the most common case, the code contains a conditional statement that tests to see if the value returned is nonzero as shown below.

```
#include <stdlib.h>
int sum;

main()
{
   int pid;
   sum = 0;
   pid = fork();
   if (pid !=0 ) {    /* original process */
      printf("The original process prints this.\n");
      } else {
         printf("The new process prints this.\n");
   }
   exit(0);
}
```

7.2 Master and slave processes of server programs

It is possible for a server to achieve some concurrency using a single process, but most concurrent servers use multiple processes in the master-slave fashion. A single *master server process* begins execution initially: It opens a socket at the well-known port, waits for the next request, and creates a *slave server process* to handle each request. Thus, the master server never communicates directly with a client; instead, it passes that responsibility to a slave process. Each slave process handles communication with one client. After the slave forms a response and sends it to the client, it exits.

In UNIX, the master server creates a new slave server process for each connection using the fork system call. Listed below are the steps that a concurrent server uses for a connection-oriented protocol.

For master:

1. Create a socket and bind to the well-known address for the service being offered. Leave the socket unconnected.

2. Place the socket in passive mode, making it ready for use by a server.

3. Repeatedly call accept to receive the next request from a client, and create a new slave process to handle the response.

For slave:

1. Receive a connection request (i.e., socket for the connection) upon creation.

2. Interact with the client using the connection: Read request(s) and send back response(s).

3. Close the connection and exit. The slave process exits after handling all requests from one client.

For simple applications, a single server program can contain all the code needed for both the master and slave processes. After the call to fork, the original process loops back to accept the next incoming connection, while the new process becomes the slave and handles the connection. (In some cases, however, it may be more convenient to have the slave process execute code from a program that has been written and compiled independently. UNIX can handle such cases easily: It allows the slave process to call execve after the call to fork.)

8 The Second Example: Echoing Network Communications

We present another example in this section, which allows the client program to send a message to the server program, and the server will echo the message to the client. We use a somewhat different programming style, in which we create separate programs for passive socket and active socket, and include a more general error handling procedure. In addition, we provide a more general Makefile for compiling.

8.1 The errexit() procedure

The function `errexit()` is used to handle errors. It is defined in the file `errexit.c` as follows.

```
/* errexit.c - errexit */

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

/*------------------------------------------------------------------------
```

```
 * errexit - print an error message and exit
 *------------------------------------------------------------------------
 */

int errexit(const char *format, ... )
{
   va_list args;

   va_start(args, format);
   vfprintf(stderr, format, args);
   va_end(args);
   exit(1);
}
```

Dissection of code

- The ellipses ... indicate to the compiler that the number and type of the remaining arguments may vary.
- Variable arguments: <stdarg.h>. This header file provides the programmer with a portable means of writing functions such as printf() that have a variable number of arguments. The header file contains one typedef and three macros. How these are implemented is system dependent; the following is one way it can be done.

  ```
  typedef char * va_list;

  #define va_start(ap,v)   ((void (ap = (va_list) &v + sizeof(v)))
  #define va_arg(ap, type) (*((type *)(ap))++)
  #define va_end(ap)       ((void) (ap = 0))
  ```

  In the macro va_start(), the variable v is the last argument that gets declared in the header to your variable argument function definition. This variable cannot be of storage class register, and it cannot be an array type, or a type such as char that gets widened by automatic conversions. The macro va_start() initializes the argument pointer ap. The macro va_arg() accesses the next argument in the list. The macro va_end() performs any cleanup that may be required before function exit. The following is an example illustrating the use of these constructs.

  ```
  #include <stdio.h>
  #include <stdarg.h>

  int va_sum(int cnt, ...);
  ```

```
main ()
{
  int a = 1, b = 2, c = 3;

  printf("First call:  sum = %d\n", va_sum(2,a,b));
  printf("Second call: sum = %d\n", va_sum(3,a,b,c));
}

int va_sum(int cnt, ...)      /* sum the arguments */
{
  int      i, sum = 0;
  va_list  ap;

  va_start(ap,cnt);           /* startup */
  for (i = 0; i < cnt; ++i)
     sum += va_arg(ap,int);   /* get the next argument */
  va_end(ap);                 /* cleanup */
  return sum;
}
```

We will see the following output when we run this program.

```
First call:  sum = 3
Second call: sum = 6
```

- Function vfprintf correponds to function fprintf. They have the following prototypes:

```
int  fprintf(FILE *fp, const char *cntro_string, ...);
int vfprintf(FILE *fp, const char *cntrl_string, va_list ap);
```

Function fprintf writes formatted text into the file associated with fp and returns the number of characters written. If an error occurs, the error indicator is set and a negative value is returned. For function vfprintf, instead of a variable length argument list, it has a pointer to an array of arguments as defined in stdarg.h.

8.2 Passive socket

```
/* passivesock.c - passivesock */

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

#include <netinet/in.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <netdb.h>

extern int errno;

int errexit(const char *format, ... );

u_short portbase = 0; /* port base, for non-root servers */

/*------------------------------------------------------------------------
 * passivesock - allocate & bind a server socket using TCP or UDP
 *------------------------------------------------------------------------
 */
int passivesock(const char *service, const char *transport, int qlen)
/*
 * Arguments:
 *      service   - service associated with the desired port
 *      transport - transport protocol to use ("tcp" or "udp")
 *      qlen      - maximum server request queue length
 */
{
   struct servent  *pse;     /* pointer to service information entry  */
   struct protoent *ppe;     /* pointer to protocol information entry */
   struct sockaddr_in sin;   /* an Internet endpoint address          */
   int s, type;              /* socket descriptor and socket type     */

   memset(&sin, 0, sizeof(sin));
   sin.sin_family = AF_INET;
   sin.sin_addr.s_addr = INADDR_ANY;

   /* Map service name to port number */
   if (pse = getservbyname(service, transport))
      sin.sin_port = htons(ntohs((u_short)pse->s_port) + portbase);
   else if ((sin.sin_port = htons((u_short)atoi(service))) == 0)
        errexit("can't get \"%s\" service entry\n", service);

   /* Map protocol name to protocol number */
   if ((ppe = getprotobyname(transport)) == 0)
      errexit("can't get \"%s\" protocol entry\n", transport);

   /* Use protocol to choose a socket type */
```

```
   if (strcmp(transport, "udp") == 0)
      type = SOCK_DGRAM;
   else
      type = SOCK_STREAM;

   /* Allocate a socket */
   s = socket(PF_INET, type, ppe->p_proto);
   if (s < 0)
      errexit("can't create socket: %s\n", strerror(errno));

   /* Bind the socket */
   if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
      errexit("can't bind to %s port: %s\n", service, strerror(errno));
   if (type == SOCK_STREAM && listen(s, qlen) < 0)
      errexit("can't listen on %s port: %s\n", service, strerror(errno));
   return s;
}
```

8.3 The passiveTCP() procedure

```
/* passiveTCP.c - passiveTCP */

int passivesock(const char *service, const char *transport, int qlen);

/*------------------------------------------------------------------------
 * passiveTCP - create a passive socket for use in a TCP server
 *------------------------------------------------------------------------
 */
int passiveTCP(const char *service, int qlen)
/*
 * Arguments:
 *      service - service associated with the desired port
 *      qlen    - maximum server request queue length
 */
{
   return passivesock(service, "tcp", qlen);
}
```

8.4 The server program

```
/* TCPechod.c - main, TCPechod */
```

```
#include <sys/types.h>
#include <sys/signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/errno.h>
#include <netinet/in.h>

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define QLEN         5    /* maximum connection queue length */
#define BUFSIZE   4096

extern int errno;

void reaper(int);
int  TCPechod(int fd);
int  errexit(const char *format, ...);
int  passiveTCP(const char *service, int qlen);

/*------------------------------------------------------------------------
 * main - Concurrent TCP server for ECHO service
 *------------------------------------------------------------------------
 */
int main(int argc, char *argv[])
{
   char   *service = "echo";     /* service name or port number  */
   struct  sockaddr_in fsin;     /* the address of a client      */
   int     alen;                 /* length of client's address   */
   int     msock;                /* master server socket         */
   int     ssock;                /* slave server socket          */

   switch (argc) {
   case 1:
       break;
   case 2:
       service = argv[1];
       break;
   default:
```

```
        errexit("usage: TCPechod [port]\n");
    }

    msock = passiveTCP(service, QLEN);

    (void) signal(SIGCHLD, reaper);

    while (1) {
        alen = sizeof(fsin);
        ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
        if (ssock < 0) {
            if (errno == EINTR) /* EINTR means that a signal arrived before */
                                /* the read operation could deliver data.   */
                continue;
            errexit("accept: %s\n", strerror(errno));
        }
        switch (fork()) {
        case 0:  /* child */
            (void) close(msock);
            exit(TCPechod(ssock));
        default: /* parent */
            (void) close(ssock);
            break;
        case -1:
            errexit("fork: %s\n", strerror(errno));
        }
    }
}

/*------------------------------------------------------------------------
 * TCPechod - echo data until end of file
 *------------------------------------------------------------------------
 */
int TCPechod(int fd)
{
    char buf[BUFSIZ];
    int  cc;

    while (cc = read(fd, buf, sizeof buf)) {
        if (cc < 0)
            errexit("echo read: %s\n", strerror(errno));
        if (write(fd, buf, cc) < 0)
            errexit("echo write: %s\n", strerror(errno));
```

```
    }
    return 0;
}

/*------------------------------------------------------------------------
 * reaper - clean up zombie children
 *------------------------------------------------------------------------
 */
/*ARGSUSED*/
void reaper(int sig)
{
    int status;

    while (wait3(&status, WNOHANG, (struct rusage *)0) >= 0)
    /* empty */;
}
```

Dissection of code

As the example shows, the calls that control concurrency occupy only a small portion of the code. A master server process begins executing at main. After it checks its arguments, the master server calls passiveTCP to create a passive socket for the well-known protocol port. It then enters an infinite loop.

During each iteration of the loop, the master server calls accept to wait for a connection request from a client, which blocks until a request arrives. After the underlying TCP protocol software receives a connection request, the system creates a socket for the new connection, and the call to accept returns the socket descriptor.

After accept returns, the master server creates a slave process to handle the connection. To do so, the master process calls fork to divide itself into two processes. The newly created child process first closes the master socket, and then calls procedure TCPechod to handle the connection. The parent process closes the socket that was created to handle the new connection, and continues executing the infinite loop. The next iteration of the loop will wait at the accept call for another new connection to arrive. Note that both the original and new processes have access to open sockets after the call to fork, and that they both must close a socket before the system de-allocates it. Thus, when the master process calls close for the next connection, the socket for that connection only disappears from the master process. Similarly, when the slave process calls close for the master socket, the socket only disappears from the slave process. The slave process continues to retain access to the socket for the new connection until the slave exits; the master server continues to retain access to the socket that corresponds to the well-known port.

After the slave closes the master socket, it calls procedure TCPechod, which provides the ECHO service for one connection. Procedure TCPechod consists of a loop that re-

peatedly calls read to obtain data from the connection and then calls write to send the same data back over the connection.

Cleaning up errant processes

Concurrent servers generate processes dynamically, which introduce a potential problem of incompletely terminated processes. UNIX solves the problem by sending a signal to the parent whenever a child process exits. The exiting process remains in a zombie state until the parent executes a wait3 system call. wait3 delays its caller until a signal is received or one of its child processes terminates or stops due to tracing. If any child process has died or stopped due to tracing and this has not already been reported, return is immediate, returning the process ID and status of one of those children. If that child process has died, it is discarded. If there are no children, -1 is returned immediately. If there are only running or stopped but reported children, the cal- ling process is blocked.

To terminate a child completely (i.e., to eliminate a zombie process), our example ECHO server catches the child termination signal and executes a signal handling function. The call `signal(SIGCHLD, reaper)` informs the operating system that the master server process should execute function reaper whenever it receives a signal that a child process has exited (signal SIGCHLD). After the call to signal, the system automatically invokes reaper each time the server process receives a SIGCHLD signal.

Function reaper calls system function wait3 to complete termination for a child that exits. The function wait3 blocks until one or more children exit (for any reason). It returns a value in the status structure that can be examined to find out about the process that exited. To ensure that an erroneous call does not deadlock the server, the program uses argument WNOHANG, which means specifies that execution of the calling process is not suspended if status is not immediately available for any child process. Hence, it specifies that wait3 should not block waiting for a process to exit, but should return immediately, even if no process has exited.

8.5 Active socket

```
/* connectsock.c - connectsock */

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

#include <netinet/in.h>
#include <arpa/inet.h>

#include <netdb.h>
#include <string.h>
#include <stdlib.h>
```

```
#ifndef INADDR_NONE
#define INADDR_NONE     0xffffffff
#endif  /* INADDR_NONE */

extern int errno;

int errexit(const char *format, ... );

/*------------------------------------------------------------------------
 * connectsock - allocate & connect a socket using TCP or UDP
 *------------------------------------------------------------------------
 */
int connectsock(const char *host, const char *service, const char *transport)
/*
 * Arguments:
 *      host      - name of host to which connection is desired
 *      service   - service associated with the desired port
 *      transport - name of transport protocol to use ("tcp" or "udp")
 */
{
   struct hostent    *phe;      /* pointer to host information entry    */
   struct servent    *pse;      /* pointer to service information entry */
   struct protoent   *ppe;      /* pointer to protocol information entry*/
   struct sockaddr_in sin;      /* an Internet endpoint address         */
   int s, type;                 /* socket descriptor and socket type    */


   memset(&sin, 0, sizeof(sin));
   sin.sin_family = AF_INET;

   /* Map service name to port number */
   if (pse = getservbyname(service, transport))
      sin.sin_port = pse->s_port;
   else if ((sin.sin_port = htons((u_short)atoi(service))) == 0)
        errexit("can't get \"%s\" service entry\n", service);

   /* Map host name to IP address, allowing for dotted decimal */
   if (phe = gethostbyname(host))
      memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
      else if ((sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE)
           errexit("can't get \"%s\" host entry\n", host);

   /* Map transport protocol name to protocol number */
```

```
   if ((ppe = getprotobyname(transport)) == 0)
      errexit("can't get \"%s\" protocol entry\n", transport);

   /* Use protocol to choose a socket type */
   if (strcmp(transport, "udp") == 0)
      type = SOCK_DGRAM;
   else
      type = SOCK_STREAM;

   /* Allocate a socket */
   s = socket(PF_INET, type, ppe->p_proto);
   if (s < 0)
      errexit("can't create socket: %s\n", strerror(errno));

   /* Connect the socket */
   if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
      errexit("can't connect to %s.%s: %s\n", host, service,
              strerror(errno));
   return s;
}
```

8.6 The connectTCP() procedure

```
/* connectTCP.c - connectTCP */

int connectsock(const char *host, const char *service,
                const char *transport);

/*------------------------------------------------------------------------
 * connectTCP - connect to a specified TCP service on a specified host
 *------------------------------------------------------------------------
 */
int connectTCP(const char *host, const char *service)
/*
 * Arguments:
 *      host    - name of host to which connection is desired
 *      service - service associated with the desired port
 */
{
   return connectsock( host, service, "tcp");
}
```

8.7 The client program

```
/* TCPecho.c - main, TCPecho */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int errno;

int  TCPecho(const char *host, const char *service);
int  errexit(const char *format, ... );
int  connectTCP(const char *host, const char *service);

#define LINELEN 128

/*------------------------------------------------------------------------
 * main - TCP client for ECHO service
 *------------------------------------------------------------------------
 */
int main(int argc, char *argv[])
{
    char  *host = "localhost";  /* host to use if none supplied */
    char  *service = "echo";    /* default service name         */

    switch (argc) {
    case 1:
      host = "localhost";
      break;
    case 3:
      service = argv[2];
      /* FALL THROUGH --- get the host name in case 2 */
    case 2:
      host = argv[1]; /* use the default service port */
      break;
    default:
      fprintf(stderr, "usage: TCPecho [host [port]]\n");
      exit(1);
    }
    TCPecho(host, service);
    exit(0);
}
```

```
/*------------------------------------------------------------------------
 * TCPecho - send input to ECHO service on specified host and print reply
 *------------------------------------------------------------------------
 */
int TCPecho(const char *host, const char *service)
{
    char buf[LINELEN+1];     /* buffer for one line of text   */
    int s, n;                /* socket descriptor, read count */
    int outchars, inchars;   /* characters sent and received  */

    s = connectTCP(host, service);

    while (fgets(buf, sizeof(buf), stdin)) {
        buf[LINELEN] = '\0';   /* insure line null-terminated */
        outchars = strlen(buf);
        (void) write(s, buf, outchars);

        /* read it back */
        for (inchars = 0; inchars < outchars; inchars+=n ) {
            n = read(s, &buf[inchars], outchars - inchars);
            if (n < 0)
               errexit("socket read failed: %s\n", strerror(errno));
        }
        fputs(buf, stdout);
    }
}
```

8.8 Makefile

To compile, create the following Makefile and type make.

```
# filename: Makefile

INCLUDE =

CLNTS = TCPecho
SERVS = TCPechod
OTHER =

DEFS =
CFLAGS =
```

```
HDR =
CSRC = TCPecho.c
CXSRC = connectTCP.c connectsock.c errexit.c

SSRC = TCPechod.c
SXSRC = passiveTCP.c passivesock.c

CXOBJ = connectTCP.o connectsock.o errexit.o
SXOBJ = passiveTCP.o passivesock.o errexit.o

PROGS = ${CLNTS} ${SERVS}

all: ${PROGS}

CC = gcc
LIBS = -lnsl -lsocket

${CLNTS}: ${CXOBJ}
   ${CC} -o $@ ${CFLAGS} $@.o ${CXOBJ} $(LIBS)

${SERVS}: ${SXOBJ}
   ${CC} -o $@ ${CFLAGS} $@.o ${SXOBJ} $(LIBS)

clients: ${CLNTS}
servers: ${SERVS}

TCPecho: TCPecho.o
TCPechod: TCPechod.o

# DO NOT DELETE THIS LINE - maketd DEPENDS ON IT (The end of the file)
```

The following is a running sample on the host named host1. Lines in bold face are what the user types; other lines are echoed.

make
TCPechod 4000 &
TCPecho host1 4000
This is a test
This is a test
This is a second test
This is a second test

9 The Third Example: Listing the Contents of a Directory on a Remote Host

In this section we will show how to do a UNIX command on a remote host. The command we will implement is ls, which lists the contents of the default directory on a remote host. The default directory is the directory where the server (daemon) program is in running.

As in the second example, we will use the following files provided in Section 8: errexit.c, passivesock.c, passiveTCP.c, connectsock.c, and connectTCP.c.

9.1 The server program

```
/* TCPlsd.c - main, TCPlsd */

#include <sys/types.h>
#include <sys/signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <dirent.h>

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define QLEN            5       /* maximum connection queue length  */
#define BUFSIZE      4096

extern int       errno;

void    reaper(int);
int     TCPlsd(int fd);
int     cmd_ls(int fd);
int     errexit(const char *format, ...);
int     passiveTCP(const char *service, int qlen);


/*------------------------------------------------------------------------
 * main - Concurrent TCP server for LS service
 *------------------------------------------------------------------------
```

```
 */
int main(int argc, char *argv[])
{
  char  *service = "echo";      /* service name or port number  */
  struct sockaddr_in fsin;      /* the address of a client      */
  int alen;                     /* length of client's address   */
  int msock;                    /* master server socket         */
  int ssock;                    /* slave server socket          */

  switch (argc) {
  case  1:
    break;
  case  2:
    service = argv[1];
    break;
  default:
    errexit("usage: TCPlsd [port]\n");
  }

  msock = passiveTCP(service, QLEN);

  (void) signal(SIGCHLD, reaper);

  while (1) {
    alen = sizeof(fsin);
    ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
    if (ssock < 0) {
      if (errno == EINTR)
    continue;
      errexit("accept: %s\n", strerror(errno));
    }
    switch (fork()) {
    case 0:     /* child */
      (void) close(msock);
      exit(TCPlsd(ssock));
    default:    /* parent */
      (void) close(ssock);
      break;
    case -1:
      errexit("fork: %s\n", strerror(errno));
    }
  }
}
```

```
/*---------------------------------------------------------------------
 * TCPlsd - list contents of a directory
 *---------------------------------------------------------------------
 */

int TCPlsd(int fd)
{
  char  cmd[BUFSIZE];

  while (1) {
    read(fd, cmd, sizeof(cmd));
    if (strcmp(cmd, "LS") == 0) {
      cmd_ls(fd);
    } else
      errexit("%s: unsuported operation.\n", cmd);
  }
}

/*---------------------------------------------------------------------
 * cmd_ls - run the "ls" command
 *---------------------------------------------------------------------
 */

int cmd_ls(int fd)
{
  DIR *dirp;
  struct dirent *direntp;
  char serverbuf[BUFSIZE];

  memset(serverbuf, 0, BUFSIZE);
  if ((dirp = opendir(".")) == NULL)
     sprintf(serverbuf, "Permission denied!\n");
  else {
    while ((direntp = readdir(dirp)) != NULL)
      sprintf(serverbuf, "%s\n%s", serverbuf, direntp->d_name );
    sprintf(serverbuf, "%s\n", serverbuf);
    closedir(dirp);
  }

  if (send(fd, serverbuf, sizeof(serverbuf), 0) < 0)
      errexit("send: %s\n", strerror(errno));
  return 0;
```

```
}


/*------------------------------------------------------------------------
 * reaper - clean up zombie children
 *------------------------------------------------------------------------
 */
/*ARGSUSED*/
void reaper(int sig)
{
  int   status;

  while (wait3(&status, WNOHANG, (struct rusage *)0) >= 0)
    /* empty */;
}
```

9.2 The client program

```
/* TCPls.c - main, TCPls */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

extern int  errno;

int     TCPls(const char *host, const char *service);
int     errexit(const char *format, ...);
int     connectTCP(const char *host, const char *service);


#define LINELEN        128
#define BUFSIZE       4096

/*------------------------------------------------------------------------
 * main - TCP client for LS service
 *------------------------------------------------------------------------
 */
int main(int argc, char *argv[])
{
  char  *host = "localhost";    /* host to use if none supplied */
```

```
  char  *service = "ls";        /* default service name         */

  switch (argc) {
  case 1:
    break;
  case 3:
    service = argv[2];
    host = argv[1];
    break;
  case 2:
    service = argv[1];
    break;
  default:
    fprintf(stderr, "usage: myftp [host [port]]\n");
    break;
  }
  TCPls(host, service);
  exit(0);
}

/*------------------------------------------------------------------------
 * TCPls - send input to LS service on specified host and print reply
 *------------------------------------------------------------------------
 */
int TCPls(const char *host, const char *service)
{
  char  cmd[LINELEN];
  char  CMD[LINELEN];
  char  buf[BUFSIZE];
  int   sd, n;          /* socket descriptor, byte count */

  printf("(rls) ");

  gets(cmd);
  strcpy(CMD, cmd);

  sd = connectTCP(host, service);
  if (sd < 0)
    errexit("connect: %s\n", strerror(errno));

  if (strcmp(cmd, "ls") == 0)
    strcpy(CMD, "LS");
```

```
  if (write(sd, CMD, sizeof(CMD)) < 0)
      errexit("write: %s\n", strerror(errno));

  if (recv(sd, buf, sizeof(buf),0) < 0)
    errexit("recv: %s\n", strerror(errno));

  puts(buf);
  close(sd);
}
```

9.3 Makefile

```
# Makefile

INCLUDE =

CLNTS = TCPls
SERVS = TCPlsd
OTHER =

DEFS =
CFLAGS =

HDR =
CSRC = TCPls.c
CXSRC = connectTCP.c connectsock.c errexit.c

SSRC = TCPlsd.c
SXSRC = passiveTCP.c passivesock.c

CXOBJ = connectTCP.o connectsock.o errexit.o
SXOBJ = passiveTCP.o passivesock.o errexit.o

PROGS = ${CLNTS} ${SERVS}

all: ${PROGS}

CC = gcc
LIBS = -lnsl -lsocket

${CLNTS}: ${CXOBJ}
        ${CC} -o $@ ${CFLAGS} $@.o ${CXOBJ} $(LIBS)
```

```
${SERVS}: ${SXOBJ}
        ${CC} -o $@ ${CFLAGS} $@.o ${SXOBJ} $(LIBS)

clients: ${CLNTS}
servers: ${SERVS}

TCPls: TCPls.o
TCPlsd: TCPlsd.o


# DO NOT DELETE THIS LINE - maketd DEPENDS ON IT
```

10 Appendix: the make Utility

C programs are usually written in multiple .c files. They are compiled separately as needed. The make utility is used to keep track of source files and to provide convenient access to libraries and associated header files. The make command reads a file whose default name is Makefile. This file contains the dependencies of the various modules, or files, making up the program, along with appropriate actions to be taken. In particular, it contains the instructions for compiling or recompiling.

10.1 A simple example

For simplicity, we imagine that we have a program contained in two files: main.c and sum.c, and that a header file named sum.h is included in each of the .c files. We want the excutable code for this program to be in the file sum. Here is a simple Makefile to be used for program development and maintenance.

```
# Makefile

sum: main.o sum.o
      gcc -o sum main.o sum.o

main.o: main.c sum.h
      gcc -c main.c

sum.o: sum.c sum.h
      gcc -c sum.c
```

The first line indicates that the file sum depends on the two object files main.o and sum.o. It is an example of a dependency line; it must start in column 1. The second line indicates how the program is to be compiled if one or more of the .o files have been changed. It is called an action line or a command. There can be more than one action

following a dependency line. A dependency line and the action lines that follow it make up what is called a rule. **Caution:** Each action line must begin with a tab key.

By default, the make command will make the first rule that it finds in the Makefile. But dependent files in that rule may themselves be dependent on other files as specified in other rules, causing the other rules to be made first. These files, in turn, may cause yet other rules to be made.

The second rule in our Makefile states that main.o depends on the two files main.c and sum.h. If either of these two files is changed, then the action line shows what must be done to update main.o. After this Makefile has been created, the programmer can compile or recompile the program sum by giving the command make.

Certain rules are built into make, including the rule that a .o file depends on the corresponding .c file. Because of this, the following is an equivalent Makefile.

```
sum: main.o sum.o
        gcc -o sum main.o sum.o

main.o: sum.h
        gcc -c main.c

sum.o: sum.h
        gcc -c sum.c
```

The make utility recognizes a number of built-in macors. Using one of them, we get yet another equivalent Makefile:

```
sum: main.o sum.o
        gcc -o sum manin.o sum.o

main.o sum.o: sum.h
        gcc -c $*.c
```

The second rule here states that the two .o files depend on sum.h. If we edit sum.h, then both main.o and sum.o must be remade. The macro $*.c expands to main.c when main.o is being made, and it expands to sum.c when sum.o is being made.

A Makefile consists of a series of entries called rules that specify dependencies and actions. A rule begins in column 1 with a series of blank-separated target files, followed by a colon; it is then followed by a blank-separated series of prerequisite files, also called source files.

10.2 A more advanced example

We know that qsort() is the system-supplied quicksort routine. Suppose we have written our own quicksort() routine and an another qsort() routine from a different system for which we were able to borrow the code. We would like to compare the running times for these three sorting routines. The following is a Makefile for this purpose.

```
# Makefile to compare sorting routines

BASE = /afs/uncg.edu/user/j/jonedoe/c
CC = gcc
CFLAGS = -O
EFILE = $(BASE)/bin/compare_sorts
INCLS = -I$(BASE)/include
LIBS = $(BASE)/lib/g_lib.a $(BASE)/lib/u_lib.a

OBJS = main.o  another_qsort.o chk_order.o compare.o quicksort.o

$(EFILE): $(OBJS)
         @echo "linking ..."
         @$(CC) $(CFLAGS) -o $(EFILE) $(OBJS) $(LIBS)

$(OBJS): compare_sorts.h
         $(CC) $(CFLAGS) $(INCLS) -c $*.c
```

Dissection of the Makefile for the `compare_sorts` program

- `# Makefile to compare sorting routines`

  Comments can be put in a Makefile, beginning with a `#` and extends to the end of the line.

- `BASE = /afs/uncg.edu/user/j/jonedoe/c`

  This is an example of a macro definition. The general form of a macro definition is

  ```
  macro_name = replacement_string
  ```

  By convention, macro names are usually capitalized, but they do not have to be. The replacement string can contain white space. If a backslash `\` occurs at the end of the line, then the replacement string continues to the next line. The macro `BASE` represents a home directory on UNCG machine.

- ```
  CC = gcc
  CFLAGS = -O
  ```

  The first macro specifies the C compiler that we are using, in this case the GNU C compiler. The second macro specifies the options that will be used with the gcc command.

- ```
  EFILE = $(BASE)/bin/compare_sorts
  INCLS = -I$(BASE)/include
  LIBS = $(BASE)/lib/g_lib.a $(BASE)/lib/u_lib.a
  ```

These macros specify the executable file, a directory for include files preceded by the `-I` option, and two libraries, respectively. The first library, `g_lib.a`, is the graceful library; thte second library, `u_lib.a`, is the utility library. Since our program invokes functions that are in these libraries, the libraries must be made available to the compiler. We keep the associated header files `g_lib.h` and `u_lib.h` in the directory `}$(BASE)/include`. The compiler will have to be told to look in this directory for header files.

- `OBJS = main.o  another_qsort.o chk_order.o compare.o quicksort.o`

  In this macro definition the replacement string is the list of object files that occurs on the right side of the equal sign. Although we put `main.o` first and then list the others alphabetically, order is not important.

- ```
  $(EFILE): $(OBJS)
            @echo "linking ..."
            @$(CC) $(CFLAGS) -o $(EFILE) $(OBJS) $(LIBS)
  ```

  The first line is a dependency line. The second and third lines specify the actions to be taken. Note carefully that they begin with a tab. The `@` symbol means that the action line itself is not to be echoed on the screen. Macro invocation has the form

  ```
  $(macro_name)
  ```

  Thus, the construct `$(EFILE)` is replaced by

  ```
  $(BASE)/bin/compare_sorts
  ```

  which in turn is replaced by

  ```
  /afs/uncg.edu/user/j/johndoe/c/bin/compare_sorts
  ```

  Similarly, `$(OBJS)` is replaced by the list of object files, and so on. Thus, the dependency line states that the executable file depends on the object files. If one or more of the object files has been updated, then the specified actions occur. The second action line gets expanded to

  ```
  @gcc -O -o /afs/uncg.edu/user/j/johndoe/c/bin/compare_sorts main.o \
     another_qsort.o chk_order.o compare.o quicksort.o \
     /afs/uncg.edu/user/j/johndoe/c/lib/g_lib.a \
     /afs/uncg.edu/user/j/johndoe/c/lib/u_lib.a
  ```

  Although we have written it on four lines due to space limitations, it is acturally generated as a single line.
  ```

- `$(OBJS): compare_sorts.h`
  `$(CC) $(CFLAGS) $(INCLS) -c $*.c`

  The dependency line says that all the object files depend on the header file `comapre_sorts.h`. The construct `$*` is a predefined macro called the *base file name macro*, which expands to the file name being built, excluding any extension. For example, if main.o is being built, then `$*.c` expands to main.c, and the action line becomes

  ```
  gcc -O -I/c/include -c main.c
  ```

  Certain dependencies are built into the make utility. For example, each .o file depends on the corresponding .c file. This means that if a .c file is changed, then it will be recompiled to produce a new .o file, and this in turn will cause all the object files to be relinked.

- `-I/c/include`

  An option of the form `-I`*dir* instructs the compiler to look in the directory *dir* for `#include` files. This option complements our use of libraries. At the top of the .c files making up this program, we have the line

  ```
  #include "compare_sorts.h"
  ```

  and at the top of `compare_sorts.h` we have the lines

  ```
  #include "g_lib.h"
  #include "u_lib.h"
  ```

  These header files contain the function prototypes for the functions in our libraries. The `-I` option tells the compiler where to find these header files.

Note that the make utility is not restricted to C. It can be used to maintain programs in any language. More generally, it can be used in any kind of project that consists of files with dependencies and associated actions.

References

[Com97] D. Comer. *Computer Networks and Internets*. Prentice Hall, 1997.

[CS96] D. Comer and D. Stevens. *Internetworking with TCP/IP, Volume III—Client-Server Programming and Applications*. Prentice Hall, 1996.

[KP90] A. Kelley and I. Pohl. *A Book on C: Programming in ANSI C*, 2nd edition, Benjamin/Cummings Publishing Company, 1990.