Simple TCP Clients and Servers on *nix with C.

Aims.

This worksheet introduces a simple client and a simple server to experiment with a daytime service. It shows how **telnet** can be used to test the server. The programs are further adapted to access the standard file */etc/hosts* by using the functions **gethostbyaddr()** and **gethostbyname()**.

Socket Internet Data Structures.

To enable us to use sockets we must be able to manipulate various structures that hold information for the socket functions. These structures are fairly complex and therefore have many functions available to manipulate them. A complication comes from the fact that to generalise things a certain amount of casting of data types is required by the functions! Here is a list of those structures used in this worksheet.

sockaddr.

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

struct sockaddr {
  uint8_t     sa_len;
  sa_family_t sa_family;  /* Address family e.g. AF_xxxx value */
  char        sa_data[ 14]; /* Protocol specific data*/
};
```

This allows a standard data structure to be passed to functions, which then use the sa_family to determine how to cope with the data.

sockadd_in and in_addr.

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

struct sockaddr_in {
  uint8_t         sin_len;     /* length of struct (16 bytes) */
  sa_family_t     sin_family;  /* AF_INET */
  in_port_t       sin_port;    /* 16-bit TCP or UDP port- */
                                /* number network-byte ordered */
  struct in_addr  sin_addr;    /* 32-bit IPv4 addr network- */
                               /* byte ordered */
  char            sin_zero[ 8]; /* unused always zero!*/
};

struct in_addr {
    in_addr_t     s_addr;   /* 32_bit Ipv4 address network- */
                             /* byte ordered */
};
```

sockaddr_in is the 'internet socket address' structure. This is the actual structure that contains the port number and the host IP address, both in network byte order rather than the byte order used by the host machine. The actual IP address is held in a separate structure (for historical reasons!). This complicates things a bit because we can refer to the host address in two ways.

If we declare a variable called servaddr to hold our socket internet data we need

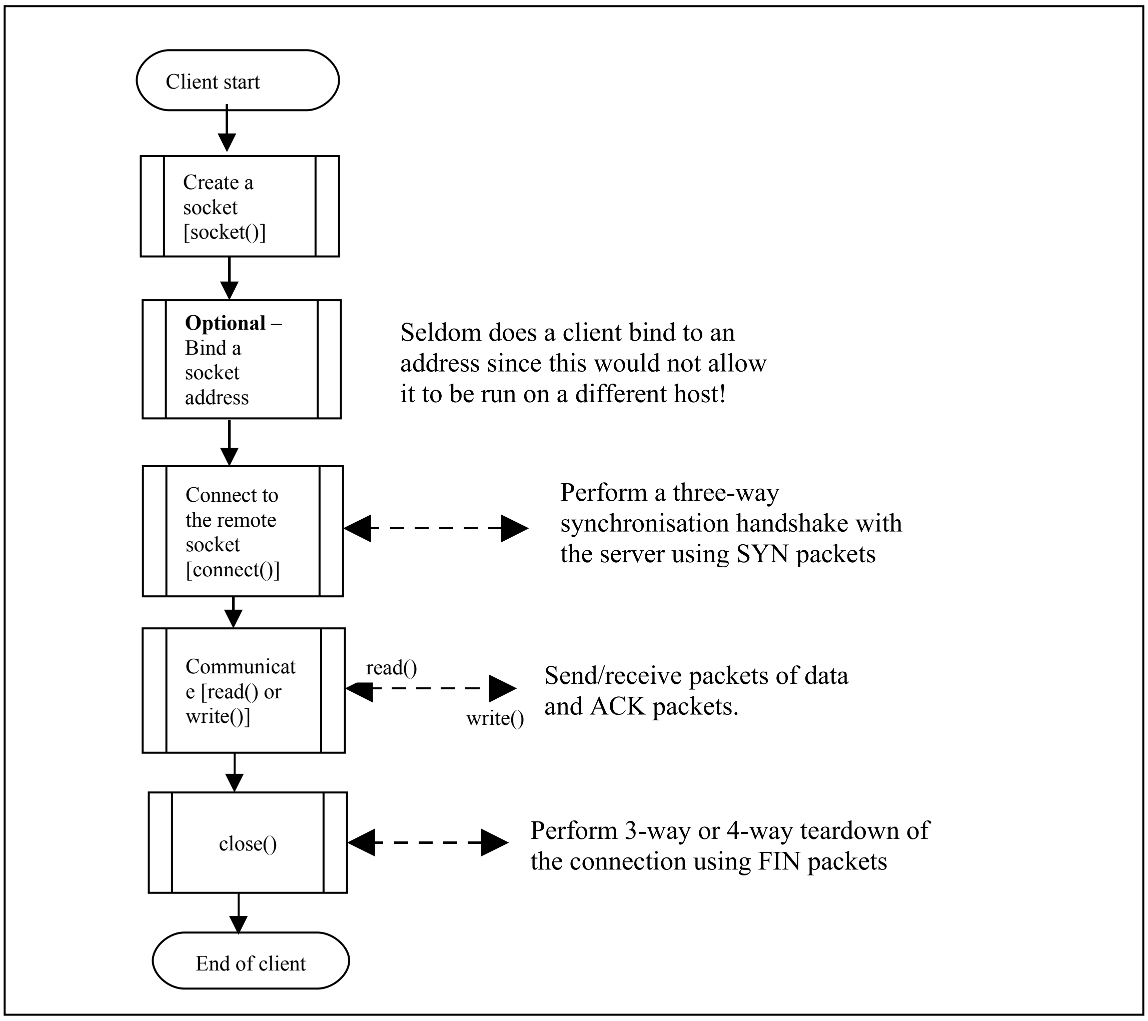
```
    struct sockaddr_in servaddr;
```

The actual IP part of this structure may be referenced as yet another structure (an *in_addr structure*) by *servaddr.sin_addr*, or as a binary representation by the more complicated looking *servaddr.sin_addr.s_addr*! Which method we use depends on the functions used for manipulation of the data.

Note that the *struct sockaddr_in* has a member *sin_zero[]* which is unused and always has its elements cleared to zero. In fact, by convention, we always zero all the members of a variable of this type before setting the values we actually want.

A Simple TCP Daytime Client.

The client code takes a standard form using the functions *socket(), connect(), read()/write() and close()*. The order in which these functions are used is shown by a flow diagram:


Client start
Create a socket [socket()]
Optional – Bind a socket address
Seldom does a client bind to an address since this would not allow it to be run on a different host!
Connect to the remote socket [connect()]
Perform a three-way synchronisation handshake with the server using SYN packets
Communicate [read() or write()]
read()
write()
Send/receive packets of data and ACK packets.
close()
Perform 3-way or 4-way teardown of the connection using FIN packets
End of client

Notice that in the following client code there is no use of the bind function. This means that the socket address for output will be 'wild'. The kernel can choose. Typically the client host has only one Network Interface Card and hence only one IP address, so this will be the one used. The port number allocated by the operating system will be the next free one!

Create an appropriate directory to hold your test programs, e.g. `mkdir Sockets`, and open that directory, `cd Sockets`, for use. Use **emacs** (or your favourite text editor) to enter the following client program, e.g. type:

```
emacs dayclient.c &
```

Once the source code has been entered compile it with:

```
gcc –Wall dayclient.c –o dayclient
```

to produce all **W**arnings and an executable file called *dayclient*.

The client needs to connect to a remote host. First try the loopback system, type:

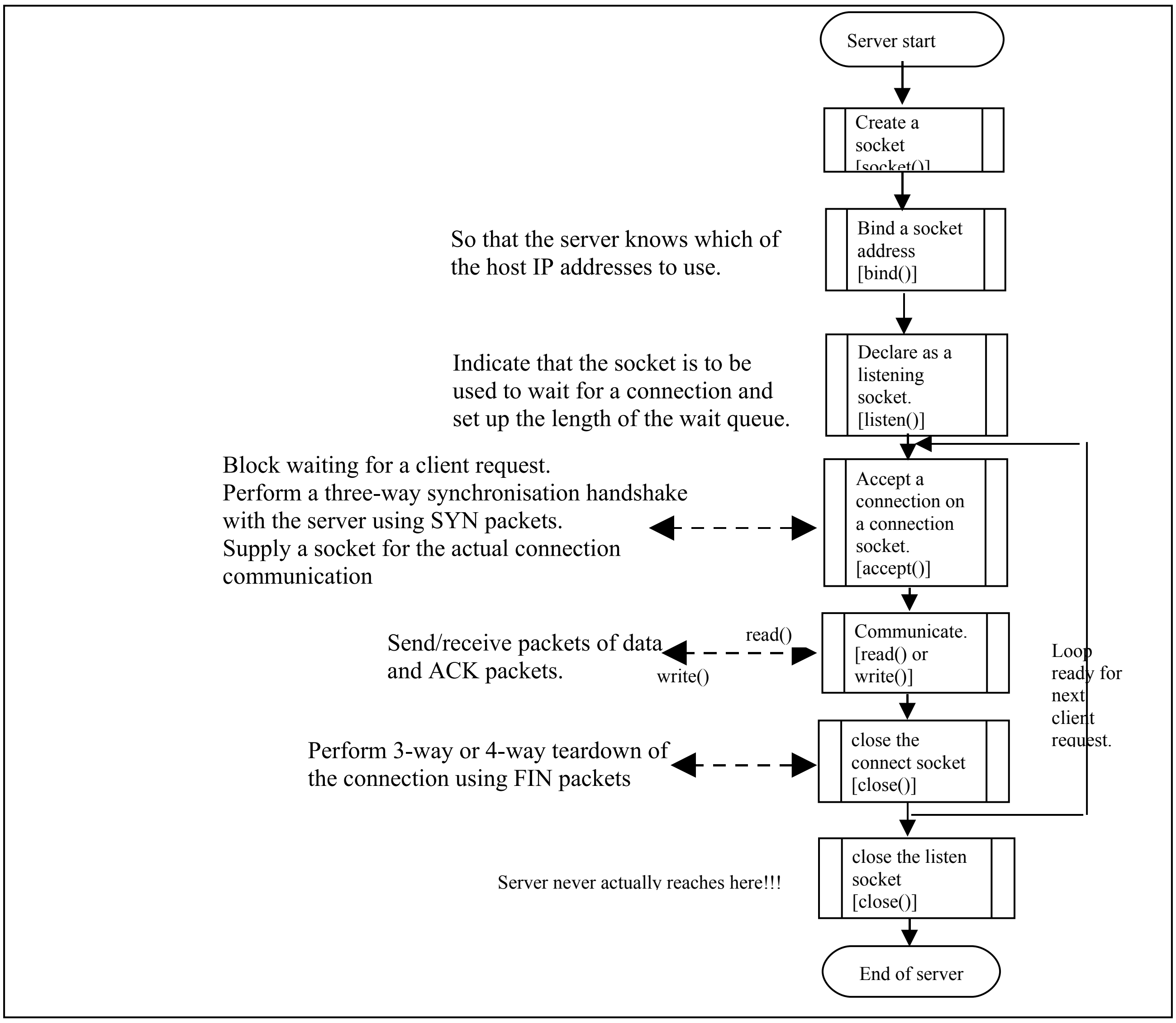
```
./dayclient 127.0.0.1
```

You should get the date and time. Try using *localhost* or the host name instead of the *loopback* IP address. Try using the IP address or the name of a true remote host. Do all these attempts work? If they do not work try to explain why not (you might need to refer to the **man** pages for the functions!). In other words experiment to see what might happen in as many situations as possible!

```
/*-------1---------2---------3---------4---------5---------6---------7--------*/
/* Page 6 of Stevens, dayclient.c
 * A simple daytime client, i.e. connects to port 13.
 * Requires IP address to be given on command line,
 * e.g. dayclient 127.0.0.1
 */
#include <stdio.h>
#include <sys/socket.h> /* socket(), connect() */
#include <string.h>     /* bzero() - should now use memset() */
#include <netinet/in.h> /* struct sockaddr_in, htons() */
#include <arpa/inet.h>  /* inet_aton() - should now use inet_pton()! */
#include <unistd.h>     /* read() */

#define MAXLINE 4096

int main( int argc, char * argv[]) {
   int sockfd;        /* the socket file descriptor */
   int nbytes;        /* number of bytes actually read from the socket */
   char recvline[ MAXLINE + 1];  /* to receive the day and time string */
   struct sockaddr_in servaddr;  /* socket internet address structure */

   if( argc != 2) {
      fprintf( stderr, "usage: %s <IPaddress>\n", argv[ 0]);
      return 1;
   }
   /* Create an Internet (AF_INET) stream (SOCK_STREAM) socket
    * with default protocol, i.e. a TCP socket.
    * Try changing the default protocol from 0 to 1 to force error! */
   if( (sockfd = socket( AF_INET, SOCK_STREAM, 0)) < 0) {
      perror( "socket error");
      return 1;
   }
   /* zero the entire socket address structure to ensure we set up
    * only what we want!  Then specify internet address family to AF_INET.
    * Set the port to that of the daytime server (i.e. 13) converting it from
    * host integer format to network format.
    * Finally convert the ascii quad format of the IP addr to network format.
    */
   bzero( &servaddr, sizeof( servaddr));
   servaddr.sin_family = AF_INET;   /* should now use PF_INET! */
   servaddr.sin_port = htons( 13);  /* daytime server 'well known' port */
   if( inet_aton( argv[ 1], &servaddr.sin_addr) <= 0) {
      /* should now use inet_pton( AF_INET, argv[ 1], &servaddr.sin_addr);
      * even older versions use servaddr.sin_addr.s_addr =inet_addr( arrgv[ 1]);!
      */
      fprintf( stderr, "inet_aton error for %s\n", argv[ 1]);
      return 1;
   }

   if( connect( sockfd, (struct sockaddr *) &servaddr, sizeof( servaddr)) < 0) {
      perror( "connect error");
      return 1;
   }
   /* Read data from the network. No bytes read implies end or error! */
   while( (nbytes = read( sockfd, recvline, MAXLINE)) > 0) {
      recvline[ nbytes] = '\0';  /* end-of-string char */
      if( fputs( recvline, stdout) == EOF) {
         perror( "fputs error");
         return 1;
      }
   } /* while */

   if( nbytes < 0) {
      perror( "read error");
      return 1;
   }
   exit( 0);
}  /* main */
```

A Simple TCP Daytime Server.

The server code takes a standard form using the functions *socket(,) bind(), listen(), accept(), read()/write() and close()* (possibly two closes!). The order in which these functions are used is shown by a flow diagram:


Server start
Create a socket [socket()]
Bind a socket address [bind()]
So that the server knows which of the host IP addresses to use.
Declare as a listening socket. [listen()]
Indicate that the socket is to be used to wait for a connection and set up the length of the wait queue.
Accept a connection on a connection socket. [accept()]
Block waiting for a client request. Perform a three-way synchronisation handshake with the server using SYN packets. Supply a socket for the actual connection communication
Communicate. [read() or write()]
read()
write()
Send/receive packets of data and ACK packets.
close the connect socket [close()]
Perform 3-way or 4-way teardown of the connection using FIN packets
Loop ready for next client request.
close the listen socket [close()]
Server never actually reaches here!!!
End of server

Note that:
A server must bind a server IP address to the socket. If the server has more than one network interface card it can elect to bind to any of its available addresses.
A server listens for a connection request and queues requests that it cannot deal with at the moment.
A server must accept a request from those queued by the *listen*() function. It associates the accepted connection with a new socket descriptor.

In your Sockets directory use **emacs** to enter the following server code, type:

```
emacs dayserver.c &
```

```
/*-------1---------2---------3---------4---------5---------6---------7--------*/
/* Page 13 of Stevens, dayserver.c
 * A simple daytime server, replacing the normal one on port 13.
* Use telnet 127.0.0.1 portNo to connect.
 * Try telnet localhost or to a neighbour's host and port number.
 * On a slow machine try netstat again to see left over connection!
 * When experimentation complete kill -9 pid to remove the server.
 * The port may still be in use if restart too quickly - bind() error!

 */
#include <stdio.h>
#include <sys/socket.h> /* socket(), bind(), listen(), accept() */
#include <string.h>     /* bzero() - should now use memset() */
#include <netinet/in.h> /* struct sockaddr_in, htons(), htonl() */
#include <unistd.h>     /* write(), close() */

#include <time.h>

#define MAXLINE 4096
#define MAXQ 10

int main( int argc, char * argv[]) {
   int listenfd;
   int connfd;
   struct sockaddr_in servaddr;
   struct sockaddr_in cliaddr;
   int lenstruct;
   char buff[ MAXLINE];
   time_t ticks;

   /* Create socket for listening on and set up internet address strucure */
   listenfd = socket( AF_INET, SOCK_STREAM, 0);
   if( listenfd < 0) {
      perror( "socket error");
      return 1;
   }

   bzero( &servaddr, sizeof( servaddr));
   servaddr.sin_family = AF_INET;   /* should now use PF_INET! */
   servaddr.sin_port = htons( 60000);  /* daytime 'well known' port in use! */
   /* allow connection on any of my IP addresses - if I have more than one! */
   servaddr.sin_addr.s_addr = htonl( INADDR_ANY);

   lenstruct = sizeof( servaddr);
   /* bind the inet address structure to the listening socket ready for use */
   if( (bind( listenfd, (struct sockaddr *) &servaddr, lenstruct)) < 0) {
      perror( "bind error");
      return 1;
   }
   /* actually put the socket in listen mode and set the q length */
   if( listen( listenfd, MAXQ) < 0) {
      perror( "listen error");
      return 1;
   }

   for( ; ; ) { /* loop forever */
      /* wait (block) for a client request on the listen socket
       * when a request appears connect it to the new socket connfd
       * ready for the actual communication with the client.
       * A successful connection will also get client IP details.
       */
      connfd = accept( listenfd, (struct sockaddr *) &cliaddr, &lenstruct);
      if( connfd < 0) {
         perror( "accept error");
         return 1;
      }
      /* get the date and time, and write it to the connected socket */
      ticks = time( NULL);
      snprintf( buff, sizeof( buff), "%.24s\r\n", ctime( &ticks));
      if( write( connfd, buff, strlen( buff)) < 0) {
         fprintf( stderr, "write error\n");
         return 1;
      }
      /* close the connected socket */
      close( connfd);
   }  /* for ever */

   /* should never get here, will need to kill the process!
   * So the listen socket will remain open for a while -
   * until garbage collection clears up! */
   close( listenfd);  /* close the listening socket */
}  /* main */
```

Compile the code with:

```
gcc –Wall dayserver.c –o dayserver
```

Run the server as a background job by:

```
./dayserver &
```

Use **ps** to check the pid of the server (needed to kill the background process later!).
Use `netstat -ap | more` to check listening status.

The server is sitting waiting for a connection. To test it out we will use **telnet**. Since the normal daytime service is active we could not use its port number. In the above program we used 60000 as the port number. We must use this for **telnet** to connect, so type:

```
telnet 127.0.0.1 60000
```

Try `telnet localhost 60000`, or your hostname, or a neighbour's version!
If all has gone well **kill** the server using the *pid* noted from using **ps** by:

```
kill –9 pid
```

Re-run the server in the background. If you manage this too soon after the **kill** you may get a *bind error*. This is because the server did not actively close the listening socket so its details have not been fully removed from the system. Just be patient and the garbage collection will eventually deal with it. This is a potential problem. If the server accidentally dies we want to be able to start it again as soon as possible!

Using the Lookup Tables.

So far we have supplied the IP address of the server host, it would be useful if we could use a host name instead. This can be found by comparing the IP address to those in the /etc/hosts file to get the corresponding name. The function to do this is **gethostbyaddr()**. An alternative function gets the IP address given the host name, this function is **gethostbyname()**. If there is no entry in the /etc/hosts file then the operating system should automatically use the DNS service!

First let us modify the server so that it displays the IP address of the connected client. This will need the function **inet_ntoa()** which converts a network form of an IP address to ASCII dot quad form.

Make a copy of your dayserver.c code so that we can edit the copy, type:

```
    cp dayserver.c dserveNames.c
```

Since we have the client details in the struct *cliaddr* (as a result of the *accept()*), we can simply add a line before getting the date and time. Add the following line:

```
fprintf( stderr, "Addr of connected client: %s\n", inet_ntoa( cliaddr.sin_addr));
```

Note that you will also probably need to add an include to all the others:

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

Save and compile this server by:

```
    gcc –Wall dserveNames.c –o dserveNames
```

Run the server in one window, `dserverNames &`

Test it from another window by using **telnet** as before:

```
    telnet localhost 60000
```

Experiment using your host's IP address and name, try connecting to someone else's server!

Assuming all has gone well we shall now try to get the client machine's name.

Add yet a further include:
```
    #include <netdb.h> /* gethostbyaddr() */
```

Add a new variable ready to take host entry details from the /etc/hosts file:

```
  struct hostent * hostptr; /* ptr to /etc/hosts entry for the client */
```

After the recently added *fprintf()* statement add the following to get the host entry for the client IP address:

```
    hostptr = gethostbyaddr( (char *)&cliaddr.sin_addr, sizeof(
                                cliaddr.sin_addr), cliaddr.sin_family);
    if( !hostptr) { /* if no entry available display an error message */
         perror( "gethostbyaddr error");
         return 1;
    }
    fprintf( stderr, "The client name is %s\n", hostptr -> h_name);
```

Save and compile this new version and test it as before. This time the server should display the connecting client IP address and its name.

Note that before running this new server in the background you must kill the presently running one using its pid!

Now let us make the day client able to use a server host name not just a server host IP address. Copy the client to create an editable version:

```
cp dayclient.c dclientNames.c
```

Edit *dclientNames.c* to add the following lines:

```
#include <netdb.h>  /* gethostbyname() */

struct hostent * servinfo;  /* ptr to server /etc/hosts file entry */
```

Before the attempt to open a socket add the lines:

```
servinfo = gethostbyname( argv[ 1]);  /* get ptr to the
entry from /etc/hosts */
if( !servinfo) {   /* if no /etc/hosts entry */
    perror( "gethostbyname error");
    return 1;
}
```

Replace the 4 lines that check the use of *inet_aton()* to convert ASCII dot quad to network form with the following:

```
/* copy the host server address to the inet address structure */
memcpy( (char *)&servaddr.sin_addr, servinfo -> h_addr, servinfo -> h_length);
```

Save and compile this new client:

```
gcc –Wall dclientNames.c dclientNames
```

Test the client using the standard daytime server:

```
./dclientNames localhost
```

Try the client with both host names and IP addresses.

<u>Using the dayclient and dayserver together.</u>

Modify the *dayclient* so that it uses the port number that the *dayserver* binds to. Further modify it so that if no IP address is supplied on the command line then the default *localhost* is used.

Normally a server would not display an error message and end every time there is a problem. Decide which of the error conditions should result in such behaviour. The other situations would result in an error message being logged in a *logfile* and the server would remain active. Try to modify the *dayserver* to create and maintain such a *logfile* (**logf** say).