Example Application

The textbook has a nice example of a web server.

I will show a simple chat server.

- Any number of clients can communicate in a group chat.
- The first line sent by a client is their name.
- Each time a client completes a line it is sent to all clients.
- Clients can join or leave the conversation at any time.

Note on my code vs the text's code

- I've done a couple of things differently from the book.
- I used getaddrinfo for DNS lookup. The book uses gethostbyname.
- I made mine work for both IPv4 and IPv6

The code can be found at https://github.com/theodore-norvell/TCP-CHAT.

Names that are underlined are part of the UNIX interface.

The Client Code

The overall algorithm is

look up the server's IP address
create a socket
connect to the server
send the person's name to the server
loop
wait until either the user or server has some input
if *the user has input*
try to read one character of user input
if *end of input, exit the loop*
if *that character is a newline, send a line to the server*
if *the server has sent input*
try to read one character of server input
if *end of input, exit the loop*
if *that character is a newline, print one line to the user*
close the socket

We will look at how each part of this is done using C and the UNIX sockets interface.

Looking up the servers address

This is probably the trickiest part

We start with the server's name and port both as C strings and pass them into getaddrinfo along with a *hint*. The hint says that we are only interested streaming interfaces, such as TCP.

```
struct addrinfo *listP ;
struct addrinfo hint ;
```

```
memset( &hint, 0, sizeof(hint) ) ;
hint.ai_socktype = SOCK_STREAM ;
hint.ai_canonname = NULL, hint.ai_addr = NULL,
hint.ai_next = NULL ;
int error = getaddrinfo( hostName, portName, &hint, &listP
) ;
```

The result is a list of *address information structures*, the pointer listP is set to point to that list.

We search the list for an address that uses TCP and either IPv4 or IPv6

```
for( ; p != NULL ; p = p->ai_next ) {
    if( (p->ai_family == AF_INET || p->ai_family ==
    AF_INET6)
    && p->ai_socktype == SOCK_STREAM
    && (p->ai_protocol == 0 || p->ai_protocol ==
    IPPROTO_TCP ) ) {
        break ;
    }
}
```

If the search failed, we quit

```
if( p == NULL ) { printf( "No TCP service found for %s\n",
hostName ) ; exit(1) ; }
```

Otherwise, we copy the address to a storage area and also save its length

```
memcpy(serverAddressP, p->ai_addr, p->ai_addrlen ) ; //
Copy the record.
*lengthP = p->ai_addrlen ; // Return the length of the
address record.
```

Create a socket

This is easy. We use the family from the address we found in the previous step. The family indicates IPv4 or IPv5. The 0 indicates TCP, which is the default protocol for streams.

```
int sockFD = socket( family, SOCK_STREAM, 0 ) ;
check( sockFD >= 0, "socket failed" ) ;
```

The sockFD is a file descriptor.

Connect to the server

Here we use the file descriptor found in the previous step and the socket address found earlier

```
int err = connect( sockFD, sockaddrP, length ) ;
check( err==0, "connect failed." ) ;
```

If the connection fails, we could try more times in a loop. I simply quit.

Sends the person's name to the server

The sprintf function 'prints' to a character array

```
sprintf( fromUser, "%s\n", name ) ;
send1Line( sockFD, fromUser ) ;
```

The send1Line function looks as follows

```
void send1Line( int fileDesciptor, char *data ) {
    // Let len be the number of characters up to and including
    the first newline.
    int len = strstr( data, "\n") - data + 1 ;
    int sent = 0 ;
    // Send the first len characters from data
```

```
    while( sent < len ) {
        int result = write( fileDesciptor, data + sent, len - sent )
        ;
        if( result < 0 && (errno == EINTR || errno ==
        EAGAIN) ) {
            result = 0 ; }
        check( result >= 0, "Write failed\n" ) ;
        sent += result ; }
}
```

It sends everything up to and including a newline character.

As we've seen before, we may need to call write more than once.

wait until either the user or server has some input

Here we use the select function. The idea is to construct a set of file descriptors and wait until at least one of them is "ready".

"ready" means that a call to read would not block. I.e. it would return without waiting.

Sets of file descriptors have their own data type in unix, called fd_set.

We start by construcing a set of 2 file descriptors.

- sockFD representing input from the server and
- 0 representing standard input

```
fd_set setOfFileDescriptors ;
FD_ZERO( &setOfFileDescriptors ) ;
```

```
FD_SET( 0, &setOfFileDescriptors ) ;
FD_SET( sockFD, &setOfFileDescriptors ) ;
```

Next we call select, which waits until one or both of the descriptors is ready.

```
int result = select(sockFD+1, &setOfFileDescriptors,
NULL, NULL, NULL) ;
check( result >= 0, "select failed" ) ;
```

Select can also be used to wait for other things, which is why there are a bunch of NULL arguments. These represent unused capabilities of select.

The way select indicates which file descriptors are ready is via the set.

So we check to see if there is input from the user we can do

```
if( FD_ISSET( 0, &setOfFileDescriptors ) )
```

Try to read one byte from the user

Here we use the read system call

```
result = read( 0, fromUser+fromUserSize, 1 ) ;
check( result >= 0, "read failed" ) ;
```

If there is no error the result will be 0 or 1. If it is zero, there is no more user input, so we leave the loop and close the socket.

If it is one, we look for a newline and if there is one, send the line to the server.

Try to read one byte from the server

This is similar to reading from the user.

This time if there is newline, we print a line to standard out.

That completes the client.

The server code

The pseudo-code looks like this

```
look up the name of this host
look up the server's IP address
make a socket
bind the socket to the address
listen on the socket for new connections
while true
    wait for input from any client or the listening socket
    if there is input on the listening socket
        accept a new client
        add its file descriptor to the set of client
        announce to all clients that there is a new client
    for each client
        if there is input from the client
            try to read one byte
            if socket was closed by the client
                    announce to all clients that the client has left
            else
                    if the byte is a newline
                            send the line to all clients along with the
            clients name
```

look up the name of this host

I started by asking the OS for the name of the current host.

```
char hostName[256] ;
gethostname( hostName, 256) ;
```

look up the server's IP address

Same as for the client

create a listening socket

Same as for the client

bind the socket to the address

```
int err = bind( sockFD, socketAddressP, length ) ;
check( err == 0, "bind failed" ) ;
```

listen on the socket for new connections

```
err = listen( sockFD, 10 ) ;
check( err >= 0, "listen failed" ) ;
```

The 10 indicates the number of clients that can be queued waiting to connect.

The rest

The rest of the server isn't very different from the client.

It uses select, read, and write in a similar way.

The only thing that is different is when there is input on the listening socket.

Instead of reading it, we do an accept call to obtain a file descriptor for a new client.

```
int newClientFD = accept(listeningFD, &socketAddress,
&length ) ;
check( newClientFD >= 0, "accept failed" ) ;
```

My advice

It is good to understand what is happening at the application/OS interface, but it is tricky and not portable to other operating systems.

Leave low-level programming like this to the experts.

Use a cross-platform library such as

- boost.asio for C++
- Qt for C++
- The Java standard library for Java

End of networks-03.