

# Tutorial 3 Programming IP Sockets using Linux

## This tutorial will:

- Force you to read and trust someone else's code...very typical/important for OS programming and this is the primary purpose of this exercise believe it or not!
- Force you to play with C and UNIX, well Linux, (in case you don't get enough).
- Give you an overview of particular type of sockets but in reading the man pages you'll get a feel for what you can do
- Let you get a feel for socket programming which is the foundation of today's distributed systems.

## Introduction

This tutorial focuses on sockets and in particular IP sockets. Interestingly HTTP, SSL, etc all use this mechanism. The socket API used here is the standard Berkeley Sockets Interface - almost all network tools in Linux and other Unix-based operating systems rely on this interface. For this tutorial we only look at IPv4. At the transport layer, sockets support two specific protocols: TCP (transmission control protocol) and UDP (user datagram protocol). Sockets are a nice abstraction for communication as they cannot be used to access lower network layers (i.e. you don't need to know the network topology e.g. Ethernet/dialup etc). Nor do you need to worry about higher level protocols like HTTP; FTP etc (unless that's what you are using sockets to build).

When you program a sockets application, you have a choice to make between using TCP and using UDP. Each has its own benefits and disadvantages (I'll leave that to you as an exercise as this is not a networking module). All you have to bear in mind for this tutorial is that TCP establishes a continuous open connection between a client and a server, over which bytes may be written and where correct order is guaranteed, for the life of the connection. However, bytes written over TCP have no built-in structure, so higher-level protocols are required to delimit any data records and fields within the transmitted bytestream.

You will also have to understand what an IP address and a port is. An IP address is a 32-bit data value, usually e.g., 146.169.14.3. A port is a 16-bit data value, usually simply represented as a number less than 65536--most often one in the tens or hundreds range. Moreover, IP address gets a packet *to* a machine, a port lets the machine decide which process/service (if any) to direct it to. You will have to find an IP address for this tutorial and can use a number of tools to do this.

This tutorial will simply use sockets to build an echo server. This is a server that receives a string from a client and sends the exact same string back to that client. We are specifically using TCP and stream processing. Please look up the full description of sockets to see what other options we have (e.g. UDP, non Internet etc). Here you create a socket, establish a connection to the server, send some data to the server then receive data back, then at the end close the connection.

## Before you start:

1. *Pull down the **client.c** and **server.c** code into your specially created **osprograms** directory from my usual OSII webpage*
2. *Start at least two shell windows on two (or more) different machines*
3. *Using your favourite tool, find out the IP addresses of these machines*
4. *Fill in the missing code to build the client code; use the server code to help you. Also look at Linux' man/sockets pages. The missing code is indicated by **££££££££££££**'s*
5. *Compile and Run the server and a number of clients on different machines; play with the ports, the payload (string lengths) and numbers of clients. Watch the code on **top** or **somesuch** if you like.(Remember this code was designed for a Linux machine..if you accidentally run it on a sunOS machine, for example, you will have to make some changes beyond the scope of this tutorial - windows, don't even go there!). Kill your server when finished*
6. *If you are feeling particularly energetic change both client and server to talk UDP instead of TCP.*

## Setting up the client <sup>1</sup>

Look at the code for `client.c` that you have been given. Here a particular buffer size is defined, which limits the amount of data echoed and a simple error function is also defined to catch all the error messages in the programs.

### Creating the client socket

We initially check that the user has typed in the correct data that is required by the system to set up the socket. The arguments to the `socket()` call decide the type of socket: `PF_INET` just means it uses IP; `SOCK_STREAM` and `IPPROTO_TCP` go together for our TCP socket. The value returned is a socket handle; if the socket creation fails, it will return -1 rather than a positive numbered handle. *You will need to complete the **sock = code***

### Establishing the connection

Using the socket handle, we need to establish a connection with the server. A connection requires a `sockaddr` structure that describes the server. Specifically, we need to specify the server and port to connect to using `echoserver.sin_addr.s_addr` and `echoserver.sin_port`. The fact we are using an IP address is specified with `echoserver.sin_family`, but this will always be set to `AF_INET`. As with creating the socket, the attempt to establish a connection will return -1 if the attempt fails. Otherwise, the socket is now ready to accept sending and receiving data. *You will need to complete the **connect code***

### Send then receive the data

Now that the connection is established, a call to `send()` takes as arguments the socket handle itself, the string or payload to send, the length of the sent string, and a flag argument. Normally the flag is the default value 0. The return value of the `send()` call is the number of bytes successfully sent. *You will need to complete the **send code**.*

The `recv()` call is not guaranteed to get everything in-transit on a particular call--it simply blocks until it gets *something*. Therefore, we loop until we have gotten back as many bytes as were sent, writing each partial string as we get it. Obviously, a different protocol might decide when to terminate receiving bytes in a different manner (perhaps a delimiter within the bytestream). *You will need to complete the **receive code**.*

### Client Cleanup

At the end of the process, we want to call `close()` on the socket.

## Socket server Walk Through

Here we talk you through the Socket Server. The socket server we present here is able to handle multiple client requests. Basically, there are two aspects to a server: handling each established connection, and listening for connections to establish. In our example we split the handling of a particular connection into support function--which looks quite a bit like a TCP client application does. We name that function `HandleClient()`.

Listening for new connections is a bit different from client code. The trick is that the socket you initially create and bind to an address and port is not the actually connected socket. This initial socket acts more like a socket factory, producing new connected sockets as needed. Here we will handle pending connected sockets in synchronous order.

### TCP echo server setup

Our echo server starts out with pretty much the same few `#include s` as the client did, and defines some constants and an error handling function. The `BUFSIZE` constant limits the data sent per loop. The `MAXPENDING` constant limits the number of connections that will be queued at a time (only one will be *serviced* at a time in our simple server). The `Die()` function is the same as in the client.

---

<sup>1</sup> Acknowledgements to David Mertz

## The connection handler

The connection handler receives any initial bytes available, then cycle through sending back data and receiving more data. For short echo strings (particularly if less than `BUFSIZE` ) and typical connections, only one pass through the while loop will occur. But remember the underlying sockets interface (and TCP/IP) does not make any guarantees about how the bytestream will be split between calls to `recv()` .The socket that is passed in to the handler function is one that already connected to the requesting client. Once we are done with echoing all the data, we should close this socket; the parent server socket stays around to spawn new children, like the one just closed.

## A TCP echo server (configuring the server socket)

Creating a socket has a slightly different purpose for a server than for a client. Though it has the same syntax as the client; the echoserver structure is setup with information about the server itself (i.e. not the peer it wants to connect to). We use the special constant `INADDR_ANY` to enable client request to be received on any IP address the server supplies; in principle, such as in a multi-hosting server, you could specify a particular IP address instead.

*Incidentally, both IP and port addresses are converted to network byte order for the `sockaddr_in` structure. (If you ever need to reverse this to return to native byte order, use `ntohs()` and `ntohl()` all to do with with endian we are talking about see [http://linux.about.com/library/cmd/blcmdl3\\_ntohs.htm](http://linux.about.com/library/cmd/blcmdl3_ntohs.htm) ). These functions are no-ops (ie blank or nothing to be done) on some platforms, but it is still wise to use them for cross-platform compatibility.*

## Binding and listening

Whereas the client application `connect()` 'd to a server's IP address and port, the server `bind()` 's to its own address and port. Once the server socket is bound, it is ready to `listen()` . As with most socket functions, both `bind()` and `listen()` return -1 if they have a problem. Once a server socket is listening, it is ready to `accept()` client connections, acting as a factory for sockets on each connection.

## Socket factory

Creating new sockets for client connections is the crux of a server. The function `accept()` does two important things: it returns a socket pointer for the new socket; and it populates the `sockaddr_in` structure pointed to, by `echoclient` . We can see the populated structure in `echoclient` with the `fprintf()` command that accesses the client IP address. The client socket pointer is passed to our `HandleClient()` routing which is at the top of the server code.

Fin