**https://www.binarytides.com/socket-programming-c-linux-tutorial/**

**Socket programming in C on Linux – The Ultimate Guide for Beginners**

By [Silver Moon](https://www.binarytides.com/author/admin/) | May 17, 2020

[154 Comments](https://www.binarytides.com/socket-programming-c-linux-tutorial/#comments)

* [**Facebook**](https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.binarytides.com%2Fsocket-programming-c-linux-tutorial%2F&t=Socket%20programming%20in%20C%20on%20Linux%20-%20The%20Ultimate%20Guide%20for%20Beginners)
* [**Twitter**](https://x.com/intent/tweet?text=Socket%20programming%20in%20C%20on%20Linux%20-%20The%20Ultimate%20Guide%20for%20Beginners&url=https%3A%2F%2Fwww.binarytides.com%2Fsocket-programming-c-linux-tutorial%2F)
* Pinterest
* [**LinkedIn**](https://www.linkedin.com/shareArticle?url=https%3A%2F%2Fwww.binarytides.com%2Fsocket-programming-c-linux-tutorial%2F&title=Socket%20programming%20in%20C%20on%20Linux%20-%20The%20Ultimate%20Guide%20for%20Beginners&summary=Learn%20socket%20programming%20in%20C%20on%20the%20linux%20platform.%20Write%20socket%20servers%20and%20client%20programs%20in%20C.&mini=true)

**TCP/IP socket programming in C**

This is a quick tutorial on **socket programming in c** language on a Linux system. "Linux" because the code snippets shown over here will work only on a Linux system and not on Windows. The [windows api to socket programming](https://www.binarytides.com/winsock-socket-programming-tutorial/) is called winsock and we shall go through it in another tutorial.

Sockets are the "virtual" endpoints of any kind of network communications done between 2 hosts over in a network. For example when you type www.google.com in your web browser, it opens a socket and connects to google.com to fetch the page and show it to you. Same with any chat client like gtalk or skype. Any network communication goes through a socket.

The socket api on linux is similar to bsd/unix sockets from which it has evolved. Although over time the api has become slightly different at few places. And now the newer official standard is [posix sockets api](https://en.wikipedia.org/wiki/POSIX) which is same as bsd sockets.

This tutorial assumes that you have basic knowledge of C and pointers. You will need to have **gcc compiler** installed on your **Linux system**. An IDE along with gcc would be great. I would recommend geany as you can quickly edit and run single file programs in it without much configurations. On ubuntu you can do a sudo apt-get install geany on the terminal.

All along the tutorial there are code snippets to demonstrate some concepts. You can run those code snippets in geany rightaway and test the results to better understand the concepts.

**1. Create a socket**

This first thing to do is create a socket. The **socket** function does this.
Here is a code sample :

**#include<stdio.h>**

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

**int main(int argc , char \*argv[])d**

**{**

 **int socket\_desc;**

 **socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);**

 **if (socket\_desc == -1)**

 **{**

 **printf("Could not create socket");**

 **}**

 **return 0;**

**}**

Function socket() creates a socket and returns a descriptor which can be used in other functions. The above code will create a socket with following properties

**Address Family - AF\_INET (this is IP version 4) Type - SOCK\_STREAM (this means connection oriented TCP protocol) Protocol - 0 [ or IPPROTO\_IP This is IP protocol]**

Next we shall try to connect to some server using this socket.
We can connect to www.google.com

**Note**

Apart from SOCK\_STREAM type of sockets there is another type called SOCK\_DGRAM which indicates the **UDP protocol**. This type of socket is non-connection socket. In this tutorial we shall stick to SOCK\_STREAM or TCP sockets.

**2. Connect socket to a server**

We connect to a remote server on a certain port number. So we need 2 things, **ip address** and **port number** to connect to.

To connect to a remote server we need to do a couple of things. First is to create a sockaddr\_in structure with proper values.

struct sockaddr\_in server;

Have a look at the structure

**// IPv4 AF\_INET sockets:**

**struct sockaddr\_in {**

 **short sin\_family; // e.g. AF\_INET, AF\_INET6**

 **unsigned short sin\_port; // e.g. htons(3490)**

 **struct in\_addr sin\_addr; // see struct in\_addr, below**

 **char sin\_zero[8]; // zero this if you want to**

**};**

**struct in\_addr {**

 **unsigned long s\_addr; // load with inet\_pton()**

**};**

**struct sockaddr {**

 **unsigned short sa\_family; // address family, AF\_xxx**

 **char sa\_data[14]; // 14 bytes of protocol address**

**};**

The sockaddr\_in has a member called sin\_addr of type in\_addr which has a s\_addr which is nothing but a long. It contains the IP address in long format.

Function **inet\_addr** is a very handy function to convert an IP address to a long format. This is how you do it :

**server.sin\_addr.s\_addr = inet\_addr("74.125.235.20");**

So you need to know the IP address of the remote server you are connecting to. Here we used the ip address of google.com as a sample. A little later on we shall see how to find out the ip address of a given domain name.

The last thing needed is the **connect** function. It needs a socket and a sockaddr structure to connect to. Here is a code sample.

**#include<stdio.h>**

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

**#include<arpa/inet.h> //inet\_addr**

**int main(int argc , char \*argv[])**

**{**

 **int socket\_desc;**

 **struct sockaddr\_in server;**

 **//Create socket**

 **socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);**

 **if (socket\_desc == -1)**

 **{**

 **printf("Could not create socket");**

 **}**

 **server.sin\_addr.s\_addr = inet\_addr("74.125.235.20");**

 **server.sin\_family = AF\_INET;**

 **server.sin\_port = htons( 80 );**

 **//Connect to remote server**

 **if (connect(socket\_desc , (struct sockaddr \*)&server , sizeof(server)) < 0)**

 **{**

 **puts("connect error");**

 **return 1;**

 **}**

 **puts("Connected");**

 **return 0;**

**}**

It cannot be any simpler. It creates a socket and then connects. If you run the program it should show Connected.
Try connecting to a port different from port 80 and you should not be able to connect which indicates that the port is not open for connection.

OK , so we are now connected. Lets do the next thing , sending some data to the remote server.

**Connections are present only in tcp sockets**

The concept of "connections" apply to SOCK\_STREAM/TCP type of sockets. Connection means a reliable "stream" of data such that there can be multiple such streams each having communication of its own. Think of this as a pipe which is not interfered by other data.

Other sockets like UDP , ICMP , ARP dont have a concept of "connection". These are non-connection based communication. Which means you keep sending or receiving packets from anybody and everybody.

**3. Send data over socket**

Function **send** will simply send data. It needs the socket descriptor , the data to send and its size.
Here is a very simple example of sending some data to google.com ip :

**#include<stdio.h>**

**#include<string.h> //strlen**

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

**#include<arpa/inet.h> //inet\_addr**

**int main(int argc , char \*argv[])**

**{**

 **int socket\_desc;**

 **struct sockaddr\_in server;**

 **char \*message;**

 **//Create socket**

 **socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);**

 **if (socket\_desc == -1)**

 **{**

 **printf("Could not create socket");**

 **}**

 **server.sin\_addr.s\_addr = inet\_addr("74.125.235.20");**

 **server.sin\_family = AF\_INET;**

 **server.sin\_port = htons( 80 );**

 **//Connect to remote server**

 **if (connect(socket\_desc , (struct sockaddr \*)&server , sizeof(server)) < 0)**

 **{**

 **puts("connect error");**

 **return 1;**

 **}**

 **puts("Connected\n");**

 **//Send some data**

 **message = "GET / HTTP/1.1\r\n\r\n";**

 **if( send(socket\_desc , message , strlen(message) , 0) < 0)**

 **{**

 **puts("Send failed");**

 **return 1;**

 **}**

 **puts("Data Send\n");**

 **return 0;**

**}**

In the above example , we first connect to an ip address and then send the string message "GET / HTTP/1.1\r\n\r\n" to it.
The message is actually a http command to fetch the mainpage of a website.

Now that we have send some data , its time to receive a reply from the server. So lets do it.

**Note**

When sending data to a socket you are basically writing data to that socket. This is similar to writing data to a file. Hence you can also use the **write** function to send data to a socket. Later in this tutorial we shall use write function to send data.

**4. Receive data on socket**

Function **recv** is used to receive data on a socket. In the following example we shall send the same message as the last example and receive a reply from the server.

**#include<stdio.h>**

**#include<string.h> //strlen**

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

**#include<arpa/inet.h> //inet\_addr**

**int main(int argc , char \*argv[])**

**{**

 **int socket\_desc;**

 **struct sockaddr\_in server;**

 **char \*message , server\_reply[2000];**

 **//Create socket**

 **socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);**

 **if (socket\_desc == -1)**

 **{**

 **printf("Could not create socket");**

 **}**

 **server.sin\_addr.s\_addr = inet\_addr("74.125.235.20");**

 **server.sin\_family = AF\_INET;**

 **server.sin\_port = htons( 80 );**

 **//Connect to remote server**

 **if (connect(socket\_desc , (struct sockaddr \*)&server , sizeof(server)) < 0)**

 **{**

 **puts("connect error");**

 **return 1;**

 **}**

 **puts("Connected\n");**

 **//Send some data**

 **message = "GET / HTTP/1.1\r\n\r\n";**

 **if( send(socket\_desc , message , strlen(message) , 0) < 0)**

 **{**

 **puts("Send failed");**

 **return 1;**

 **}**

 **puts("Data Send\n");**

 **//Receive a reply from the server**

 **if( recv(socket\_desc, server\_reply , 2000 , 0) < 0)**

 **{**

 **puts("recv failed");**

 **}**

 **puts("Reply received\n");**

 **puts(server\_reply);**

 **return 0;**

**}**

Here is the output of the above code :

**Connected**

**Data Send**

**Reply received**

**HTTP/1.1 302 Found**

**Location: http://www.google.co.in/**

**Cache-Control: private**

**Content-Type: text/html; charset=UTF-8**

**Set-Cookie: PREF=ID=0edd21a16f0db219:FF=0:TM=1324644706:LM=1324644706:S=z6hDC9cZfGEowv\_o; expires=Sun, 22-Dec-2013 12:51:46 GMT; path=/; domain=.google.com**

**Date: Fri, 23 Dec 2011 12:51:46 GMT**

**Server: gws**

**Content-Length: 221**

**X-XSS-Protection: 1; mode=block**

**X-Frame-Options: SAMEORIGIN**

**<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">**

**<TITLE>302 Moved</TITLE></HEAD><BODY>**

**<H1>302 Moved</H1>**

**The document has moved**

**<A HREF="http://www.google.co.in/">here</A>.**

**</BODY></HTML>**

We can see what reply was send by the server. It looks something like Html, well IT IS html. Google.com replied with the content of the page we requested. Quite simple!

**Note**

When receiving data on a socket , we are basically reading the data on the socket. This is similar to reading data from a file. So we can also use the **read** function to read data on a socket. For example :

**read(socket\_desc, server\_reply , 2000);**

Now that we have received our reply, its time to close the socket.

**5. Close socket**

Function **close** is used to close the socket. Need to include the unistd.h header file for this.

**close(socket\_desc);**

Thats it.

**6. Summary**

So in the above example we learned how to

**1. Create a socket 2. Connect to remote server 3. Send some data 4. Receive a reply**

Your web browser also does the same thing when you open www.google.com
This kind of socket activity represents a **socket client**. A client is an application that connects to a remote system to fetch or retrieve data.

The other kind of socket application is called a **socket server**. A server is a system that uses sockets to receive incoming connections and provide them with data. It is just the opposite of Client. So www.google.com is a server and your web browser is a client. Or more technically www.google.com is a HTTP Server and your web browser is an HTTP client.

Now its time to do some server tasks using sockets. But before we move ahead there are a few side topics that should be covered just incase you need them.

**Get ip address of hostname**

When connecting to a remote host , it is necessary to have its IP address. Function **gethostbyname** is used for this purpose. It takes the domain name as the parameter and returns a structure of type hostent. This structure has the ip information. It is present in **netdb.h**. Lets have a look at this structure

**/\* Description of data base entry for a single host. \*/**

**struct hostent**

**{**

 **char \*h\_name; /\* Official name of host. \*/**

 **char \*\*h\_aliases; /\* Alias list. \*/**

 **int h\_addrtype; /\* Host address type. \*/**

 **int h\_length; /\* Length of address. \*/**

 **char \*\*h\_addr\_list; /\* List of addresses from name server. \*/**

**};**

The **h\_addr\_list** has the IP addresses. So now lets have some code to use them.

**#include<stdio.h> //printf**

**#include<string.h> //strcpy**

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

**#include<netdb.h> //hostent**

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

**int main(int argc , char \*argv[])**

**{**

 **char \*hostname = "www.google.com";**

 **char ip[100];**

 **struct hostent \*he;**

 **struct in\_addr \*\*addr\_list;**

 **int i;**

 **if ( (he = gethostbyname( hostname ) ) == NULL)**

 **{**

 **//gethostbyname failed**

 **herror("gethostbyname");**

 **return 1;**

 **}**

 **//Cast the h\_addr\_list to in\_addr , since h\_addr\_list also has the ip address in long format only**

 **addr\_list = (struct in\_addr \*\*) he->h\_addr\_list;**

 **for(i = 0; addr\_list[i] != NULL; i++)**

 **{**

 **//Return the first one;**

 **strcpy(ip , inet\_ntoa(\*addr\_list[i]) );**

 **}**

 **printf("%s resolved to : %s" , hostname , ip);**

 **return 0;**

**}**

Output of the code would look like :

**www.google.com resolved to : 74.125.235.20**

So the above code can be used to find the ip address of any domain name. Then the ip address can be used to make a connection using a socket.

Function **inet\_ntoa** will convert an IP address in long format to dotted format. This is just the opposite of **inet\_addr**.

So far we have see some important structures that are used. Lets revise them :

1. **sockaddr\_in** - Connection information. Used by connect , send , recv etc.
2. **in\_addr** - Ip address in long format
3. **sockaddr**
4. **hostent** - The ip addresses of a hostname. Used by gethostbyname

In the next part we shall look into creating servers using socket. Servers are the opposite of clients, that instead of connecting out to others, they wait for incoming connections.

**Socket server**

OK now onto server things. Socket servers operate in the following manner

**1. Open a socket 2. Bind to a address(and port). 3. Listen for incoming connections. 4. Accept connections 5. Read/Send**

We have already learnt how to open a socket. So the next thing would be to bind it.

**1. Bind socket to a port**

The bind function can be used to bind a socket to a particular "address and port" combination. It needs a sockaddr\_in structure similar to connect function.

**int socket\_desc;**

**struct sockaddr\_in server;**

**//Create socket**

**socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);**

**if (socket\_desc == -1)**

**{**

 **printf("Could not create socket");**

**}**

**//Prepare the sockaddr\_in structure**

**server.sin\_family = AF\_INET;**

**server.sin\_addr.s\_addr = INADDR\_ANY;**

**server.sin\_port = htons( 8888 );**

**//Bind**

**if( bind(socket\_desc,(struct sockaddr \*)&server , sizeof(server)) < 0)**

**{**

 **puts("bind failed");**

**}**

**puts("bind done");**

Now that bind is done, its time to make the socket listen to connections. We bind a socket to a particular IP address and a certain port number. By doing this we ensure that all incoming data which is directed towards this port number is received by this application.

This makes it obvious that you cannot have 2 sockets bound to the same port.

**2. Listen for incoming connections on the socket**

After binding a socket to a port the next thing we need to do is listen for connections. For this we need to put the socket in listening mode. Function **listen** is used to put the socket in listening mode. Just add the following line after bind.

**//Listen**

**listen(socket\_desc , 3);**

Thats all. Now comes the main part of accepting new connections.

**3. Accept connection**

Function **accept** is used for this. Here is the code

**#include<stdio.h>**

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

**#include<arpa/inet.h> //inet\_addr**

**int main(int argc , char \*argv[])**

**{**

 **int socket\_desc , new\_socket , c;**

 **struct sockaddr\_in server , client;**

 **//Create socket**

 **socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);**

 **if (socket\_desc == -1)**

 **{**

 **printf("Could not create socket");**

 **}**

 **//Prepare the sockaddr\_in structure**

 **server.sin\_family = AF\_INET;**

 **server.sin\_addr.s\_addr = INADDR\_ANY;**

 **server.sin\_port = htons( 8888 );**

 **//Bind**

 **if( bind(socket\_desc,(struct sockaddr \*)&server , sizeof(server)) < 0)**

 **{**

 **puts("bind failed");**

 **}**

 **puts("bind done");**

 **//Listen**

 **listen(socket\_desc , 3);**

 **//Accept and incoming connection**

 **puts("Waiting for incoming connections...");**

 **c = sizeof(struct sockaddr\_in);**

 **new\_socket = accept(socket\_desc, (struct sockaddr \*)&client, (socklen\_t\*)&c);**

 **if (new\_socket<0)**

 **{**

 **perror("accept failed");**

 **}**

 **puts("Connection accepted");**

 **return 0;**

**}**

**Program output**

Run the program. It should show

**bind done**

**Waiting for incoming connections...**

So now this program is waiting for incoming connections on port 8888. Dont close this program , keep it running.
Now a client can connect to it on this port. We shall use the telnet client for testing this. Open a terminal and type

**$ telnet localhost 8888**

On the terminal you shall get

**Trying 127.0.0.1...**

**Connected to localhost.**

**Escape character is '^]'.**

**Connection closed by foreign host.**

And the server output will show

**bind done**

**Waiting for incoming connections...**

**Connection accepted**

So we can see that the client connected to the server. Try the above process till you get it perfect.

**4. Get the ip address of the connected client**

You can get the ip address of client and the port of connection by using the sockaddr\_in structure passed to accept function. It is very simple :

**char \*client\_ip = inet\_ntoa(client.sin\_addr);**

**int client\_port = ntohs(client.sin\_port);**

We accepted an incoming connection but closed it immediately. This was not very productive. There are lots of things that can be done after an incoming connection is established. Afterall the connection was established for the purpose of communication. So lets reply to the client.

We can simply use the **write** function to write something to the socket of the incoming connection and the client should see it. Here is an example :

**#include<stdio.h>**

**#include<string.h> //strlen**

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

**#include<arpa/inet.h> //inet\_addr**

**#include<unistd.h> //write**

**int main(int argc , char \*argv[])**

**{**

 **int socket\_desc , new\_socket , c;**

 **struct sockaddr\_in server , client;**

 **char \*message;**

 **//Create socket**

 **socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);**

 **if (socket\_desc == -1)**

 **{**

 **printf("Could not create socket");**

 **}**

 **//Prepare the sockaddr\_in structure**

 **server.sin\_family = AF\_INET;**

 **server.sin\_addr.s\_addr = INADDR\_ANY;**

 **server.sin\_port = htons( 8888 );**

 **//Bind**

 **if( bind(socket\_desc,(struct sockaddr \*)&server , sizeof(server)) < 0)**

 **{**

 **puts("bind failed");**

 **return 1;**

 **}**

 **puts("bind done");**

 **//Listen**

 **listen(socket\_desc , 3);**

 **//Accept and incoming connection**

 **puts("Waiting for incoming connections...");**

 **c = sizeof(struct sockaddr\_in);**

 **new\_socket = accept(socket\_desc, (struct sockaddr \*)&client, (socklen\_t\*)&c);**

 **if (new\_socket<0)**

 **{**

 **perror("accept failed");**

 **return 1;**

 **}**

 **puts("Connection accepted");**

 **//Reply to the client**

 **message = "Hello Client , I have received your connection. But I have to go now, bye\n";**

 **write(new\_socket , message , strlen(message));**

 **return 0;**

**}**

Run the above code in 1 terminal. And connect to this server using telnet from another terminal and you should see this :

**$ telnet localhost 8888**

**Trying 127.0.0.1...**

**Connected to localhost.**

**Escape character is '^]'.**

**Hello Client , I have received your connection. But I have to go now, bye**

**Connection closed by foreign host.**

So the client(telnet) received a reply from server.

We can see that the connection is closed immediately after that simply because the server program ends after accepting and sending reply. A server like www.google.com is always up to accept incoming connections.

It means that a server is supposed to be running all the time. Afterall its a server meant to serve. So we need to keep our server RUNNING non-stop. The simplest way to do this is to put the **accept** in a loop so that it can receive incoming connections all the time.

**5. Live Server**

So a live server will be alive for all time. Lets code this up :

**#include<stdio.h>**

**#include<string.h> //strlen**

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

**#include<arpa/inet.h> //inet\_addr**

**#include<unistd.h> //write**

**int main(int argc , char \*argv[])**

**{**

 **int socket\_desc , new\_socket , c;**

 **struct sockaddr\_in server , client;**

 **char \*message;**

 **//Create socket**

 **socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);**

 **if (socket\_desc == -1)**

 **{**

 **printf("Could not create socket");**

 **}**

 **//Prepare the sockaddr\_in structure**

 **server.sin\_family = AF\_INET;**

 **server.sin\_addr.s\_addr = INADDR\_ANY;**

 **server.sin\_port = htons( 8888 );**

 **//Bind**

 **if( bind(socket\_desc,(struct sockaddr \*)&server , sizeof(server)) < 0)**

 **{**

 **puts("bind failed");**

 **return 1;**

 **}**

 **puts("bind done");**

 **//Listen**

 **listen(socket\_desc , 3);**

 **//Accept and incoming connection**

 **puts("Waiting for incoming connections...");**

 **c = sizeof(struct sockaddr\_in);**

 **while( (new\_socket = accept(socket\_desc, (struct sockaddr \*)&client, (socklen\_t\*)&c)) )**

 **{**

 **puts("Connection accepted");**

 **//Reply to the client**

 **message = "Hello Client , I have received your connection. But I have to go now, bye\n";**

 **write(new\_socket , message , strlen(message));**

 **}**

 **if (new\_socket<0)**

 **{**

 **perror("accept failed");**

 **return 1;**

 **}**

 **return 0;**

**}**

We havent done a lot there. Just the accept was put in a loop.

Now run the program in 1 terminal , and open 3 other terminals. From each of the 3 terminal do a telnet to the server port.

Each of the telnet terminal would show :

**$ telnet localhost 8888**

**Trying 127.0.0.1...**

**Connected to localhost.**

**Escape character is '^]'.**

**Hello Client , I have received your connection. But I have to go now, bye**

And the server terminal would show

**bind done**

**Waiting for incoming connections...**

**Connection accepted**

**Connection accepted**

**Connection accepted**

So now the server is running nonstop and the telnet terminals are also connected nonstop. Now close the server program.
All telnet terminals would show "Connection closed by foreign host."
Good so far. But still there is not effective communication between the server and the client.

The server program accepts connections in a loop and just send them a reply, after that it does nothing with them. Also it is not able to handle more than 1 connection at a time. So now its time to handle the connections , and handle multiple connections together.

**6. Handle multiple socket connections with threads**

To handle every connection we need a separate handling code to run along with the main server accepting connections.
One way to achieve this is using threads. The main server program accepts a connection and creates a new thread to handle communication for the connection, and then the server goes back to accept more connections.

On Linux threading can be done with the pthread (posix threads) library. It would be good to read some small tutorial about it if you dont know anything about it. However the usage is not very complicated.

We shall now use threads to create handlers for each connection the server accepts. Lets do it pal.

**#include<stdio.h>**

**#include<string.h> //strlen**

**#include<stdlib.h> //strlen**

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

**#include<arpa/inet.h> //inet\_addr**

**#include<unistd.h> //write**

**#include<pthread.h> //for threading , link with lpthread**

**void \*connection\_handler(void \*);**

**int main(int argc , char \*argv[])**

**{**

 **int socket\_desc , new\_socket , c , \*new\_sock;**

 **struct sockaddr\_in server , client;**

 **char \*message;**

 **//Create socket**

 **socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);**

 **if (socket\_desc == -1)**

 **{**

 **printf("Could not create socket");**

 **}**

 **//Prepare the sockaddr\_in structure**

 **server.sin\_family = AF\_INET;**

 **server.sin\_addr.s\_addr = INADDR\_ANY;**

 **server.sin\_port = htons( 8888 );**

 **//Bind**

 **if( bind(socket\_desc,(struct sockaddr \*)&server , sizeof(server)) < 0)**

 **{**

 **puts("bind failed");**

 **return 1;**

 **}**

 **puts("bind done");**

 **//Listen**

 **listen(socket\_desc , 3);**

 **//Accept and incoming connection**

 **puts("Waiting for incoming connections...");**

 **c = sizeof(struct sockaddr\_in);**

 **while( (new\_socket = accept(socket\_desc, (struct sockaddr \*)&client, (socklen\_t\*)&c)) )**

 **{**

 **puts("Connection accepted");**

 **//Reply to the client**

 **message = "Hello Client , I have received your connection. And now I will assign a handler for you\n";**

 **write(new\_socket , message , strlen(message));**

 **pthread\_t sniffer\_thread;**

 **new\_sock = malloc(1);**

 **\*new\_sock = new\_socket;**

 **if( pthread\_create( &sniffer\_thread , NULL , connection\_handler , (void\*) new\_sock) < 0)**

 **{**

 **perror("could not create thread");**

 **return 1;**

 **}**

 **//Now join the thread , so that we dont terminate before the thread**

 **//pthread\_join( sniffer\_thread , NULL);**

 **puts("Handler assigned");**

 **}**

 **if (new\_socket<0)**

 **{**

 **perror("accept failed");**

 **return 1;**

 **}**

 **return 0;**

**}**

**/\***

 **\* This will handle connection for each client**

 **\* \*/**

**void \*connection\_handler(void \*socket\_desc)**

**{**

 **//Get the socket descriptor**

 **int sock = \*(int\*)socket\_desc;**

 **char \*message;**

 **//Send some messages to the client**

 **message = "Greetings! I am your connection handler\n";**

 **write(sock , message , strlen(message));**

 **message = "Its my duty to communicate with you";**

 **write(sock , message , strlen(message));**

 **//Free the socket pointer**

 **free(socket\_desc);**

 **return 0;**

**}**

Run the above server and open 3 terminals like before. Now the server will create a thread for each client connecting to it.

The telnet terminals would show :

**$ telnet localhost 8888**

**Trying 127.0.0.1...**

**Connected to localhost.**

**Escape character is '^]'.**

**Hello Client , I have received your connection. And now I will assign a handler for you**

**Hello I am your connection handler**

**Its my duty to communicate with you**

This one looks good , but the communication handler is also quite dumb. After the greeting it terminates. It should stay alive and keep communicating with the client.

One way to do this is by making the connection handler wait for some message from a client as long as the client is connected. If the client disconnects , the connection handler ends.

So the connection handler can be rewritten like this :

**/\***

 **\* This will handle connection for each client**

 **\* \*/**

**void \*connection\_handler(void \*socket\_desc)**

**{**

 **//Get the socket descriptor**

 **int sock = \*(int\*)socket\_desc;**

 **int read\_size;**

 **char \*message , client\_message[2000];**

 **//Send some messages to the client**

 **message = "Greetings! I am your connection handler\n";**

 **write(sock , message , strlen(message));**

 **message = "Now type something and i shall repeat what you type \n";**

 **write(sock , message , strlen(message));**

 **//Receive a message from client**

 **while( (read\_size = recv(sock , client\_message , 2000 , 0)) > 0 )**

 **{**

 **//Send the message back to client**

 **write(sock , client\_message , strlen(client\_message));**

 **}**

 **if(read\_size == 0)**

 **{**

 **puts("Client disconnected");**

 **fflush(stdout);**

 **}**

 **else if(read\_size == -1)**

 **{**

 **perror("recv failed");**

 **}**

 **//Free the socket pointer**

 **free(socket\_desc);**

 **return 0;**

**}**

The above connection handler takes some input from the client and replies back with the same. Simple! Here is how the telnet output might look

**$ telnet localhost 8888**

**Trying 127.0.0.1...**

**Connected to localhost.**

**Escape character is '^]'.**

**Hello Client , I have received your connection. And now I will assign a handler for you**

**Greetings! I am your connection handler**

**Now type something and i shall repeat what you type**

**Hello**

**Hello**

**How are you**

**How are you**

**I am fine**

**I am fine**

So now we have a server thats communicative. Thats useful now.

**Linking the pthread library**

When compiling programs that use the pthread library you need to link the library. This is done like this :

**$ gcc program.c -lpthread**

**Conclusion**

By now you must have learned the basics of **socket programming in C**. You can try out some experiments like writing a chat client or something similar.

If you think that the tutorial needs some addons or improvements or any of the code snippets above dont work then feel free to make a comment below so that it gets fixed.

# <https://www.binarytides.com/raw-sockets-c-code-linux/>

# How to Code Raw Sockets in C on Linux

By [Silver Moon](https://www.binarytides.com/author/admin/) | July 26, 2020

[39 Comments](https://www.binarytides.com/raw-sockets-c-code-linux/#comments)

* [**Facebook**](https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.binarytides.com%2Fraw-sockets-c-code-linux%2F&t=How%20to%20Code%20Raw%20Sockets%20in%20C%20on%20Linux)
* [**Twitter**](https://x.com/intent/tweet?text=How%20to%20Code%20Raw%20Sockets%20in%20C%20on%20Linux&url=https%3A%2F%2Fwww.binarytides.com%2Fraw-sockets-c-code-linux%2F)
* Pinterest
* [**LinkedIn**](https://www.linkedin.com/shareArticle?url=https%3A%2F%2Fwww.binarytides.com%2Fraw-sockets-c-code-linux%2F&title=How%20to%20Code%20Raw%20Sockets%20in%20C%20on%20Linux&summary=Raw%20tcp%20sockets%20in%20C%20Raw%20sockets%20can%20be%20used%20to%20construct%20a%20packet%20manually%20inside%20an%20application.%20In%20normal%20sockets%20when%20any%20data%20is%20send%20over%20the%20network%2C%20the%20kernel%20of%20the%20operating%20system&mini=true)

### Raw tcp sockets in C

Raw sockets can be used to construct a packet manually inside an application. In normal sockets when any data is send over the network, the kernel of the operating system adds some headers to it like IP header and TCP header. So an application only needs to take care of what data it is sending and what reply it is expecting.

But there are other cases when an application needs to set its own headers. Raw sockets are used in security related applications like nmap , packets sniffer etc.

In this article we are going to program raw sockets on linux using native sockets.

Windows for example does not support raw socket programming directly. To [program raw sockets on windows](https://www.binarytides.com/raw-sockets-packets-with-winpcap/) a packet crafting library like winpcap has to be used.

In this article we are going to do some raw socket programming by constructing a raw TCP packet and sending it over the network. Before programming raw sockets, it is recommended that you learn about the basics of [socket programming in c](https://www.binarytides.com/socket-programming-c-linux-tutorial/).

### Raw TCP packets

A TCP packet is constructed like this

Packet = IP Header + TCP Header + Data

The plus means to attach the binary data side by side. So when making a raw tcp packet we need to know how to construct the headers properly. The structures of all headers are established standards which are described in RFCs.

#### IP Header Structure

The structure of IP Header as given by [RFC 791](https://tools.ietf.org/html/rfc791) :

**0 1 2 3**

**0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**|Version| IHL |Type of Service| Total Length |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Identification |Flags| Fragment Offset |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Time to Live | Protocol | Header Checksum |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Source Address |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Destination Address |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Options | Padding |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

The "Source Address" field stores the ip address of the system sending the packet and the "Destination Address" contains the ip address of the destination system. Ip addresses are stored in long number format. The "Protocol" field stores a number that indicates the protocol, which is TCP in this case.

#### Structure of TCP header

The structure of a TCP header as given by RFC 793

**0 1 2 3**

**0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Source Port | Destination Port |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Sequence Number |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Acknowledgment Number |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Data | |U|A|P|R|S|F| |**

**| Offset| Reserved |R|C|S|S|Y|I| Window |**

**| | |G|K|H|T|N|N| |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Checksum | Urgent Pointer |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| Options | Padding |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

**| data |**

**+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+**

So we need to construct the headers according to the formats specified above.

#### Raw tcp sockets

Create a raw socket like this

**int s = socket (AF\_INET, SOCK\_RAW, IPPROTO\_TCP);**

The above function call creates a raw socket of protocol TCP. This means that we have to provide the TCP header along with the data. The kernel or the network stack of Linux shall provide the IP header.

If we want to provide the IP header as well then there are 2 ways of doing this

1. Use protocol IPPROTO\_RAW - This will allow to specify the IP header and everything that is contained in the packet.

**int s = socket (AF\_INET, SOCK\_RAW, IPPROTO\_RAW);**

2. Set the IP\_HDRINCL socket option to 1 - This is same as the above. Just another way of doing.

**int s = socket (AF\_INET, SOCK\_RAW, IPPROTO\_TCP);**

**int one = 1;**

**const int \*val = &one;**

**if (setsockopt (s, IPPROTO\_IP, IP\_HDRINCL, val, sizeof (one)) < 0)**

**{**

 **printf ("Error setting IP\_HDRINCL. Error number : %d . Error message : %s \n" , errno , strerror(errno));**

 **exit(0);**

**}**

When using the IP\_HDRINCL the protocol used in the socket function is effectively of no use.

**In this example we are creating raw sockets where we specify the Ip header and TCP header. The packet that moves out of the machine actually has 1 more header attached to it called the Ethernet header. So the actual packet structure is somewhat like this. Packet = Ethernet header + Ip header + TCP header + Data Take a look at the packets sniffed by wireshark to understand this better. It is important to note here that the Ethernet header is provided by the OS kernel and we do not have to construct it. However it is possible to make such raw packets where we can even specify the ethernet header, but we shall look into those in a separate article.**

Below is an example code which constructs a raw TCP packet with some data

### Final Code

**/\***

 **Raw TCP packets**

**\*/**

**#include <stdio.h> //for printf**

**#include <string.h> //memset**

**#include <sys/socket.h> //for socket ofcourse**

**#include <stdlib.h> //for exit(0);**

**#include <errno.h> //For errno - the error number**

**#include <netinet/tcp.h> //Provides declarations for tcp header**

**#include <netinet/ip.h> //Provides declarations for ip header**

**#include <arpa/inet.h> // inet\_addr**

**#include <unistd.h> // sleep()**

**/\***

 **96 bit (12 bytes) pseudo header needed for tcp header checksum calculation**

**\*/**

**struct pseudo\_header**

**{**

 **u\_int32\_t source\_address;**

 **u\_int32\_t dest\_address;**

 **u\_int8\_t placeholder;**

 **u\_int8\_t protocol;**

 **u\_int16\_t tcp\_length;**

**};**

**/\***

 **Generic checksum calculation function**

**\*/**

**unsigned short csum(unsigned short \*ptr,int nbytes)**

**{**

 **register long sum;**

 **unsigned short oddbyte;**

 **register short answer;**

 **sum=0;**

 **while(nbytes>1) {**

 **sum+=\*ptr++;**

 **nbytes-=2;**

 **}**

 **if(nbytes==1) {**

 **oddbyte=0;**

 **\*((u\_char\*)&oddbyte)=\*(u\_char\*)ptr;**

 **sum+=oddbyte;**

 **}**

 **sum = (sum>>16)+(sum & 0xffff);**

 **sum = sum + (sum>>16);**

 **answer=(short)~sum;**

 **return(answer);**

**}**

**int main (void)**

**{**

 **//Create a raw socket**

 **int s = socket (PF\_INET, SOCK\_RAW, IPPROTO\_TCP);**

 **if(s == -1)**

 **{**

 **//socket creation failed, may be because of non-root privileges**

 **perror("Failed to create socket");**

 **exit(1);**

 **}**

 **//Datagram to represent the packet**

 **char datagram[4096] , source\_ip[32] , \*data , \*pseudogram;**

 **//zero out the packet buffer**

 **memset (datagram, 0, 4096);**

 **//IP header**

 **struct iphdr \*iph = (struct iphdr \*) datagram;**

 **//TCP header**

 **struct tcphdr \*tcph = (struct tcphdr \*) (datagram + sizeof (struct ip));**

 **struct sockaddr\_in sin;**

 **struct pseudo\_header psh;**

 **//Data part**

 **data = datagram + sizeof(struct iphdr) + sizeof(struct tcphdr);**

 **strcpy(data , "ABCDEFGHIJKLMNOPQRSTUVWXYZ");**

 **//some address resolution**

 **strcpy(source\_ip , "192.168.1.2");**

 **sin.sin\_family = AF\_INET;**

 **sin.sin\_port = htons(80);**

 **sin.sin\_addr.s\_addr = inet\_addr ("1.2.3.4");**

 **//Fill in the IP Header**

 **iph->ihl = 5;**

 **iph->version = 4;**

 **iph->tos = 0;**

 **iph->tot\_len = sizeof (struct iphdr) + sizeof (struct tcphdr) + strlen(data);**

 **iph->id = htonl (54321); //Id of this packet**

 **iph->frag\_off = 0;**

 **iph->ttl = 255;**

 **iph->protocol = IPPROTO\_TCP;**

 **iph->check = 0; //Set to 0 before calculating checksum**

 **iph->saddr = inet\_addr ( source\_ip ); //Spoof the source ip address**

 **iph->daddr = sin.sin\_addr.s\_addr;**

 **//Ip checksum**

 **iph->check = csum ((unsigned short \*) datagram, iph->tot\_len);**

 **//TCP Header**

 **tcph->source = htons (1234);**

 **tcph->dest = htons (80);**

 **tcph->seq = 0;**

 **tcph->ack\_seq = 0;**

 **tcph->doff = 5; //tcp header size**

 **tcph->fin=0;**

 **tcph->syn=1;**

 **tcph->rst=0;**

 **tcph->psh=0;**

 **tcph->ack=0;**

 **tcph->urg=0;**

 **tcph->window = htons (5840); /\* maximum allowed window size \*/**

 **tcph->check = 0; //leave checksum 0 now, filled later by pseudo header**

 **tcph->urg\_ptr = 0;**

 **//Now the TCP checksum**

 **psh.source\_address = inet\_addr( source\_ip );**

 **psh.dest\_address = sin.sin\_addr.s\_addr;**

 **psh.placeholder = 0;**

 **psh.protocol = IPPROTO\_TCP;**

 **psh.tcp\_length = htons(sizeof(struct tcphdr) + strlen(data) );**

 **int psize = sizeof(struct pseudo\_header) + sizeof(struct tcphdr) + strlen(data);**

 **pseudogram = malloc(psize);**

 **memcpy(pseudogram , (char\*) &psh , sizeof (struct pseudo\_header));**

 **memcpy(pseudogram + sizeof(struct pseudo\_header) , tcph , sizeof(struct tcphdr) + strlen(data));**

 **tcph->check = csum( (unsigned short\*) pseudogram , psize);**

 **//IP\_HDRINCL to tell the kernel that headers are included in the packet**

 **int one = 1;**

 **const int \*val = &one;**

 **if (setsockopt (s, IPPROTO\_IP, IP\_HDRINCL, val, sizeof (one)) < 0)**

 **{**

 **perror("Error setting IP\_HDRINCL");**

 **exit(0);**

 **}**

 **//loop if you want to flood :)**

 **while (1)**

 **{**

 **//Send the packet**

 **if (sendto (s, datagram, iph->tot\_len , 0, (struct sockaddr \*) &sin, sizeof (sin)) < 0)**

 **{**

 **perror("sendto failed");**

 **}**

 **//Data send successfully**

 **else**

 **{**

 **printf ("Packet Send. Length : %d \n" , iph->tot\_len);**

 **}**

 **// sleep for 1 seconds**

 **sleep(1);**

 **}**

 **return 0;**

**}**

**//Complete**

### Compile and Run

Compile by program by doing a gcc raw\_socket.c at the terminal. Remember to run the program with root privileges. Raw sockets require root privileges.

**$ gcc raw\_socket.c -o raw\_socket**

**$ sudo ./raw\_socket**

Note the while loop in the above program. It has been put for testing purpose and should be removed if you dont intend to flood the target.

Use a packet sniffer like wireshark to check the output and verify that the packets have actually been generated and send over the network. Also note that if some kind of firewall like firestarter is running then it might block raw packets.

### Resources

<http://linux.die.net/man/7/raw>

<https://www.binarytides.com/raw-sockets-packets-with-winpcap/>

**Raw socket programming on windows with Winpcap**

By [Silver Moon](https://www.binarytides.com/author/admin/) | August 1, 2020

[22 Comments](https://www.binarytides.com/raw-sockets-packets-with-winpcap/#comments)

* [**Facebook**](https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.binarytides.com%2Fraw-sockets-packets-with-winpcap%2F&t=Raw%20socket%20programming%20on%20windows%20with%20Winpcap)
* [**Twitter**](https://x.com/intent/tweet?text=Raw%20socket%20programming%20on%20windows%20with%20Winpcap&url=https%3A%2F%2Fwww.binarytides.com%2Fraw-sockets-packets-with-winpcap%2F)
* Pinterest
* [**LinkedIn**](https://www.linkedin.com/shareArticle?url=https%3A%2F%2Fwww.binarytides.com%2Fraw-sockets-packets-with-winpcap%2F&title=Raw%20socket%20programming%20on%20windows%20with%20Winpcap&summary=Raw%20sockets%20with%20winpcap%20A%20previous%20post%20explains%20how%20to%20send%20on%20windows%20xp.%20However%20the%20winsock%20api%20has%20limited%20raw%20socket%20support%20in%20windows%20versions%20greater%20than%20windows%20xp%2Bsp1.%20Therefore%20winpcap%20has%20to%20be&mini=true)

**Raw sockets with winpcap**

A previous post explains how to send [raw packets using winsock api](https://www.binarytides.com/raw-sockets-using-winsock/) on windows xp.

However the winsock api has limited raw socket support in windows versions greater than windows xp+sp1.

Therefore winpcap has to be used to send raw packets on higher windows versions.

Winpcap is a packet driver useful for packet capturing and sending raw packets on the windows platform.

Raw means we have to cook the whole packet ourselves.

A TCP packet for example consists of:

**1. Ethernet header 2. IP header 3. TCP header 4. The data supposed to be send**

PACKET = ETHERNET\_HEADER + IP\_HEADER + TCP\_HEADER + DATA

Each header has its own job to do in the whole transmission process.

Code :

**u\_char packet[65536];**

Winpcap gives us one function called pcap\_sendpacket() to throw the packet on the network adapter which forwards it. We have to responsibly construct the ethernet , ip and tcp headers and attach the data.

**Ethernet Header**

**Structure**

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Ethernet destination address (first 32 bits) |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Ethernet dest (last 16 bits) |Ethernet source (first 16 bits)|

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Ethernet source address (last 32 bits) |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Type code | |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

C structure for ethernet header :

**//Ethernet Header**

**typedef struct ethernet\_header**

**{**

 **UCHAR dest[6]; //Total 48 bits**

 **UCHAR source[6]; //Total 48 bits**

 **USHORT type; //16 bits**

**} ETHER\_HDR , \*PETHER\_HDR , FAR \* LPETHER\_HDR , ETHERHeader;**

Ethernet destination address is the mac-address of the *primary gateway* of the network interface being used.
Ethernet source is the mac-address of the network interface itself.
Type field determines the type of the packet e.g. IP , ARP etc.

Now our first task is to get the source and destination mac address.
Winpcap gives the ip-addresses of all available network interfaces that can be used.

**Get Mac address**
If srcip has the source ip in in\_addr or long format then we can get the mac-address of this ip address using the function GetMacAddress. This function is codes in the source code.

**GetMacAddress(s\_mac , srcip);**

**printf("Selected device has mac address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X",s\_mac[0],s\_mac[1],s\_mac[2],s\_mac[3],s\_mac[4],s\_mac[5]);**

GetMacAddress is like :

**void GetMacAddress(unsigned char \*mac , in\_addr destip)**

**{**

 **DWORD ret;**

 **in\_addr srcip;**

 **ULONG MacAddr[2];**

 **ULONG PhyAddrLen = 6; /\* default to length of six bytes \*/**

 **srcip.s\_addr=0;**

 **//Now print the Mac address also**

 **ret = SendArp(destip , srcip , MacAddr , &PhyAddrLen);**

 **if(PhyAddrLen) {**

 **BYTE \*bMacAddr = (BYTE \*) & MacAddr;**

 **for (int i = 0; i < (int) PhyAddrLen; i++)**

 **mac[i] = (char)bMacAddr[i];**

 **}**

**}**

**SendArp** is the method that is used to retrieve the "mac-address of a IP". It is defined in iphlpapi.dll

The above demonstration is mostly self-explaining. We got the mac-address of the network interface or IP we want to use. This method shall be used to get the mac address of local computer and the gateway.

Next task is to get the ip address of the primary gateway of a certain interface.

**Get the Gateway IP**

Next we need the IP address of the primary gateway of this interface and then it mac-address.

GetGateway gets the gateway :

**void GetGateway(struct in\_addr ip , char \*sgatewayip , int \*gatewayip) {**

 **char pAdapterInfo[5000];**

 **PIP\_ADAPTER\_INFO AdapterInfo;**

 **ULONG OutBufLen = sizeof(pAdapterInfo) ;**

 **GetAdaptersInfo((PIP\_ADAPTER\_INFO) pAdapterInfo, &OutBufLen);**

 **for(AdapterInfo = (PIP\_ADAPTER\_INFO)pAdapterInfo; AdapterInfo ; AdapterInfo = AdapterInfo->Next) {**

 **if(ip.s\_addr == inet\_addr(AdapterInfo->IpAddressList.IpAddress.String))**

 **strcpy(sgatewayip , AdapterInfo->GatewayList.IpAddress.String);**

 **}**

 **\*gatewayip = inet\_addr(sgatewayip);**

**}**

GetAdaptersInfo is the function that retrieves a lot of information about a adapter.
This and SendArp are inside iphlpapi.dll ; IP helper api which we shall load and get the function pointers inside!

Buzz!

**void loadiphlpapi() {**

 **HINSTANCE hDll = LoadLibrary("iphlpapi.dll");**

 **GetAdaptersInfo = (pgetadaptersinfo)GetProcAddress(hDll,"GetAdaptersInfo");**

 **if(GetAdaptersInfo==NULL)**

 **printf("Error in iphlpapi.dll%d",GetLastError());**

 **SendArp = (psendarp)GetProcAddress(hDll,"SendARP");**

 **if(SendArp==NULL)**

 **printf("Error in iphlpapi.dll%d",GetLastError());**

**}**

By now we have the following information available :

1. Source IP address - IP of local computer.
2. Mac address of local computer.
3. Primary gateway of local computer.
4. Mac address of primary gateway.

The above 4 things are enough to build the ethernet header. Enjoy!

It is simple as :

**ETHER\_HDR \*ehdr;**

**memcpy(ehdr->source , s\_mac , 6); //Source Mac address**

**memcpy(ehdr->dest,d\_mac,6); //Destination MAC address**

**ehdr->type = htons(0x0800); //IP Frames**

**TCP Packet Structure**

The tcp packet structure has 2 parts. The IP header and the TCP header. First we shall take a look at the structures as defined in the RFC and then code them in C.

**IP Header**

RFC 791 gives the structure of an IP header as:

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

|Version| IHL |Type of Service| Total Length |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Identification |Flags| Fragment Offset |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Time to Live | Protocol | Header Checksum |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Source Address |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Destination Address |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Options | Padding |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

**C structure for IP header:**

**typedef struct ip\_hdr**

**{**

 **unsigned char ip\_header\_len:4; // 4-bit header length (in 32-bit words) normally=5 (Means 20 Bytes may be 24 also)**

 **unsigned char ip\_version :4; // 4-bit IPv4 version**

 **unsigned char ip\_tos; // IP type of service**

 **unsigned short ip\_total\_length; // Total length**

 **unsigned short ip\_id; // Unique identifier**

 **unsigned char ip\_frag\_offset :5; // Fragment offset field**

 **unsigned char ip\_more\_fragment :1;**

 **unsigned char ip\_dont\_fragment :1;**

 **unsigned char ip\_reserved\_zero :1;**

 **unsigned char ip\_frag\_offset1; //fragment offset**

 **unsigned char ip\_ttl; // Time to live**

 **unsigned char ip\_protocol; // Protocol(TCP,UDP etc)**

 **unsigned short ip\_checksum; // IP checksum**

 **unsigned int ip\_srcaddr; // Source address**

 **unsigned int ip\_destaddr; // Source address**

**} IPV4\_HDR, \*PIPV4\_HDR, FAR \* LPIPV4\_HDR , IPHeader;**

**TCP Header**

The following is the structure of a TCP header.

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Source Port | Destination Port |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Sequence Number |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Acknowledgment Number |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Data | |U|A|P|R|S|F| |

| Offset| Reserved |R|C|S|S|Y|I| Window |

| | |G|K|H|T|N|N| |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Checksum | Urgent Pointer |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Options | Padding |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| data |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

**C structure for TCP header:**

**// TCP header**

**typedef struct tcp\_header**

**{**

 **unsigned short source\_port; // source port**

 **unsigned short dest\_port; // destination port**

 **unsigned int sequence; // sequence number - 32 bits**

 **unsigned int acknowledge; // acknowledgement number - 32 bits**

 **unsigned char ns :1; //Nonce Sum Flag Added in RFC 3540.**

 **unsigned char reserved\_part1:3; //according to rfc**

 **unsigned char data\_offset:4; /\*The number of 32-bit words**

 **in the TCP header.**

 **This indicates where the data begins.**

 **The length of the TCP header**

 **is always a multiple**

 **of 32 bits.\*/**

 **unsigned char fin :1; //Finish Flag**

 **unsigned char syn :1; //Synchronise Flag**

 **unsigned char rst :1; //Reset Flag**

 **unsigned char psh :1; //Push Flag**

 **unsigned char ack :1; //Acknowledgement Flag**

 **unsigned char urg :1; //Urgent Flag**

 **unsigned char ecn :1; //ECN-Echo Flag**

 **unsigned char cwr :1; //Congestion Window Reduced Flag**

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

 **unsigned short window; // window**

 **unsigned short checksum; // checksum**

 **unsigned short urgent\_pointer; // urgent pointer**

**} TCP\_HDR , \*PTCP\_HDR , FAR \* LPTCP\_HDR , TCPHeader , TCP\_HEADER;**

**Build the IP and TCP Headers**

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* IP Header \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

 **iphdr = (PIPV4\_HDR)(packet + sizeof(ETHER\_HDR));**

 **iphdr->ip\_version = 4;**

 **iphdr->ip\_header\_len = 5; //In double words thats 4 bytes**

 **iphdr->ip\_tos = 0;**

 **iphdr->ip\_total\_length = htons (sizeof(IPV4\_HDR) + sizeof(TCP\_HDR) + strlen(dump));**

 **iphdr->ip\_id = htons(2);**

 **iphdr->ip\_frag\_offset = 0;**

 **iphdr->ip\_reserved\_zero=0;**

 **iphdr->ip\_dont\_fragment=1;**

 **iphdr->ip\_more\_fragment=0;**

 **iphdr->ip\_frag\_offset1 = 0;**

 **iphdr->ip\_ttl = 3;**

 **iphdr->ip\_protocol = IPPROTO\_TCP;**

 **iphdr->ip\_srcaddr = inet\_addr("1.2.3.4"); //srcip.s\_addr;**

 **iphdr->ip\_destaddr = inet\_addr("1.2.3.5");**

 **iphdr->ip\_checksum =0;**

 **iphdr->ip\_checksum = in\_checksum((unsigned short\*)iphdr, sizeof(IPV4\_HDR));**

 **// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TCP Header \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

 **tcphdr = (PTCP\_HDR)(packet + sizeof(ETHER\_HDR) + sizeof(IPV4\_HDR));**

 **tcphdr->source\_port = htons(SOURCE\_PORT);**

 **tcphdr->dest\_port = htons(80);**

 **tcphdr->sequence=0;**

 **tcphdr->acknowledge=0;**

 **tcphdr->reserved\_part1=0;**

 **tcphdr->data\_offset=5;**

 **tcphdr->fin=0;**

 **tcphdr->syn=1;**

 **tcphdr->rst=0;**

 **tcphdr->psh=0;**

 **tcphdr->ack=0;**

 **tcphdr->urg=0;**

 **tcphdr->ecn=0;**

 **tcphdr->cwr=0;**

 **tcphdr->window = htons(64240);**

 **tcphdr->checksum=0;**

 **tcphdr->urgent\_pointer = 0;**

**Add some Data**

After preparing the IP and TCP headers we can add application data to the packet.

**char \*dump = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";**

**data = (char\*)(packet + sizeof(ETHER\_HDR) + sizeof(IPV4\_HDR) + sizeof(TCP\_HDR));**

**strcpy(data,dump);**

**Send Packet**

The pcap\_sendpacket function is used to send the raw packet on the network. The call to the function looks as follows:

**pcap\_sendpacket(fp , packet , sizeof(ETHER\_HDR) + sizeof(IPV4\_HDR) + sizeof(TCP\_HDR) + strlen(dump));**

Thats should send the packet. Use Wireshark to check whether the packet was successfully transmitted.

The above was an example of a TCP packet. Similarly UDP ICMP or any other packet can be build.

**Compile and Run**

The download link for the source code is given below. It is a VC++ 6.0 project. Open then and compile and run.
This program needs Winpcap.

Download and install winpcap. Also download the winpcap developer files which include the necessary headers (pcap.h and others) and the library files (wpcap.lib) to be linked.

Place the winpcap header and library files somewhere in your project workspace. Then in Vc++ go to *Tools > Options > Directories* and add the directory path of include and library files.

When compiling you might get a Winpcap error related to \_W64 macro. Follow this [article](https://www.binarytides.com/blog/winpcap-compile-error-in-pcap-stdinc-h-in-vc-6-0/) to solve it.

**Output**

**Username : abc**

**Retrieving the available devices...Retrieved.**

**The following devices found :**

**1)**

**rpcap://\Device\NPF\_{EA7C1F00-CD10-4288-8B0D-EBD63C22F468}**

**Description: Network adapter 'Intel(R) 82566DC Gigabit Network Connection (Micro**

**soft's Packet Scheduler) ' on local host**

**Loopback: No**

**Address Family: #2**

**Address Family Name: AF\_INET**

**Address: 192.168.0.101**

**Netmask: 255.255.255.0**

**Broadcast Address: 255.255.255.255**

**Enter the device number you want to use : 1**

**Selected device has mac address : 00-1C-C0-F8-79-EE**

**Selected device has gateway : 192.168.0.1 (Mac : 00-1E-58-B8-D4-69)**

**Opening the selected device...Opened**

**Sending Packet...Send**

**Press any key to continue**

**Download Full Source Code**

The full source code for the above program can be downloaded here:

[Download Source Code](https://www.binarytides.com/blog_files/winpcap_raw.zip)

**Conclusion**

The program by default sends 1 packet. If you want to flood the destination then uncomment the while loop.
If any firewall like Zonealarm is running then packets might not be send.

So switch them off when experimenting with this code.

If you have any questions or feedback, let us know in the comments below.