# Socket Programming

Compro Prasad

October 27, 2016

## Contents



## 1 Create a socket

```
int socket_desc = socket(AF_INET, SOCK_STREAM, 0);

if (socket_desc == -1) {
printf("Could not create socket");
}
```

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]
```

## 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**.

### 2.1 `sockaddr_in` structure

To connect to a remote server firstly we create `sockaddr_in` structure with proper values.

```
struct sockaddr_in server;
```

The structure definations are as follows:

```
// 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 needed
};

struct in_addr {
unsigned long s_addr;         // load with inet_pton()
};

struct sockaddr {
unsigned short sa_faminly;    // address family, AF_xxx
char           sa_data;       // 14 bytes of protocol address
}
```

`s_addr` of `in_addr` structure will contain the **IP address** in long format.

### 2.2 `inet_addr` function

To convert an **IP address** to a long format `inet_addr(const char *)` function is used.

For example,

```
server.sin_addr.s_addr = inet_addr("176.34.135.167");
```

can be used for connecting to DuckDuckGo search engine where `176.34.135.167` is the **IP address** passed as a `string` parameter.

### 2.3 `connect` function

`connect` is a function for connecting to a remote server. A sample code is given below:

```
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\n");
return 1;
}

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

So, we have **created** a socket and **connected** it to a server. Now we are going to **send** / **transmit** data to the remote server.

#### 2.3.1 Note: Connections are present only in TCP sockets

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 communication of its own. It can be considered a pipe **not interfered** by other data.

UDP(User Datagram Protocol), ICMP(Internet Control Message Protocol), ARP(Address Resolution Protocol) are **non-connection** based communication which means packets can be sent to anybody and everybody.

## 3 Sending data over socket

### 3.1 `send` function

It needs the **socket descriptor** returned after creating a socket, the **data to send** and **its size**. We have the following code which sends data to DuckDuckGo.

```
// Send some data
char *message = "GET /HTTP/1.1\n";
if (send(socket_desc, message, strlen(message), 0) < 0) {
puts("Send failed\n");
return 1;
}
puts("Data sent\n");
```

The `message` string is actually commanding the server to **get** the mainpage of a website.
In the next section we try to recieve a reply from the server.

# 4 Recieve data on socket

## 4.1 `recv` function

The `recv` function will try recieving data through socket from a web server.

```
// Receive a reply from the server
char server_reply[2000];
if (recv(socket_desc, server_reply, 2000, 0) < 0) {
puts("recv failed\n");
}
puts("Reply received\n");
puts(server_reply);
```

### 4.1.1 Note:

When receiving data on a socket, we are basically **reading** it. This is similar to reading data from a file(remember the Unix philosophy?). So we can use the `read` function to read data on a socket. For example:

```
read(socket_desc, server)
```

# 5 Closing a socket

Just like files, sockets also need to be closed. We can use the primary `close` function which accepts a **file descriptor** as an argument.

```
close(socket_desc);
```