Computer networking

R. ZITOUNI, T. MAGADIS
rafik.zitouni@ece.fr
tmagadis@inseec-edu.com

Chapters

01 **Application & Transport Layers, Addressing**

02 **Routing and MAC layers of OSI model**

03 **Network analysis and Programming**

04 **Wireless and Multimedia Networks**

Network analysis and Programming

Technical requirements

The example of **c programs** can be compiled with **GCC on Linux.**

- Use the **Kali VM or Ubuntu VM** operating system running on VirtualBox
- Native if you have already installed Linux distribution on your computers
- Use a preferred **text editor** or a light IDE like **Atom**
  - You can integrate **platformio-ide-terminal** plugin to make easy the terminal interactions

Sockets

Sockets

A socket is one endpoint of a communication link between systems. Your application sends and receives all of its network data through a socket.

Portable Operating System Interface (**POSIX**) sockets or Berkeley socket (BSD) → Linux and macOs

Winsock sockets → Windows' socket API compatible with BSD

Historically, sockets were used for **inter-process communication** (**IPC**)

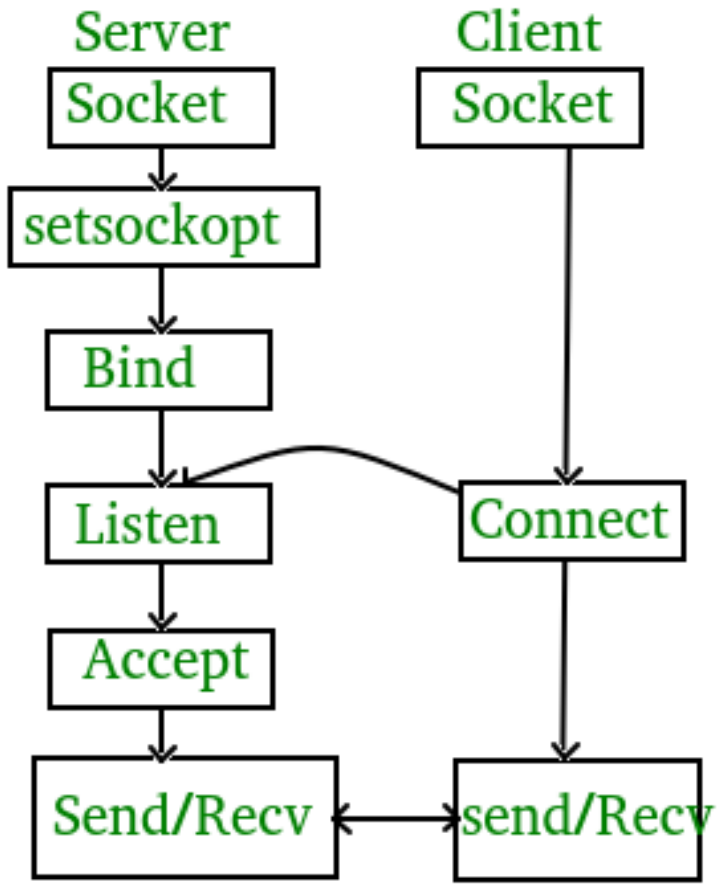
In this chapter, we use sockets only for communication with TCP and UDP.

Sockets

Sockets come in two basic types:

1) Connection-oriented → TCP

2) Connectionless → UDP



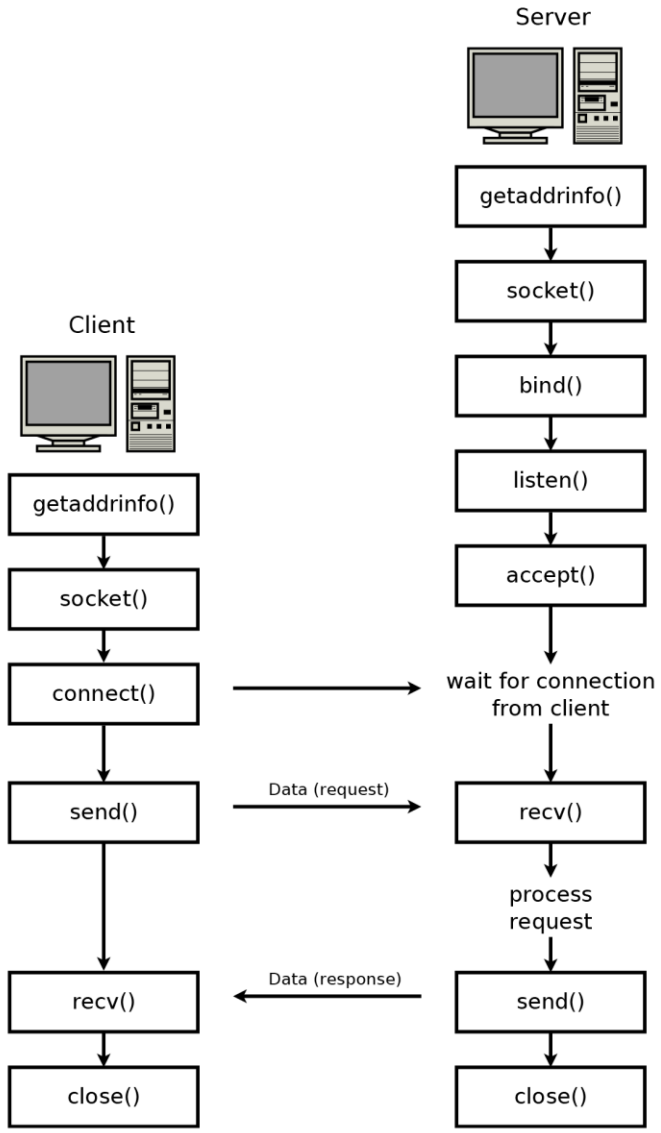
Sockets functions

1) **socket()** creates and initializes a new socket.
2) **bind()** associates a socket with a particular local IP address and port number.
3) **listen()** is used on the server to cause a TCP socket to listen for new connections.
4) **connect()** is used on the client to set the remote address and port. In the case of TCP, it also establishes a connection.
5) **accept()** is used on the server to create a new socket for an incoming TCP connection.
6) **send()** and **recv()** are used to send and receive data with a socket. You may see some networking programs using **read()** and **write()**.
7) **sendto() and recvfrom()** are used to send and receive data from sockets without a bound remote address.
8) **close()** (Berkeley sockets) are used to close a socket. In the case of TCP, this also terminates the connection.
9) **shutdown()** is used to close one side of a TCP connection. It is useful to ensure an orderly connection teardown.
10) **select()** is used to wait for an event on one or more sockets.
11) **getnameinfo() and getaddrinfo()** provide a protocol-independent manner of working with hostnames and addresses.
12) **setsockopt()** is used to change some socket options.
13) **fcntl()** (Berkeley sockets) are also used to get and set some socket options.

TCP Sockets

Connection-oriented → TCP

- The socket APIs are blocking by default
  - When we use **accept()** to wait for an incoming connection, program's execution is blocked until a new incoming connection is established.
  - When you use **recv()** to read incoming data, your program's execution blocks until new data is actually available.

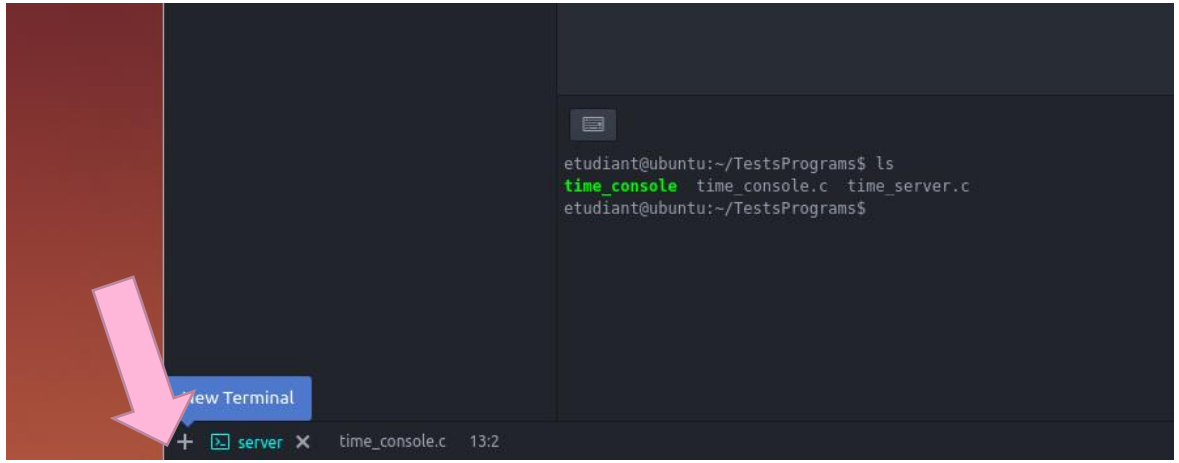


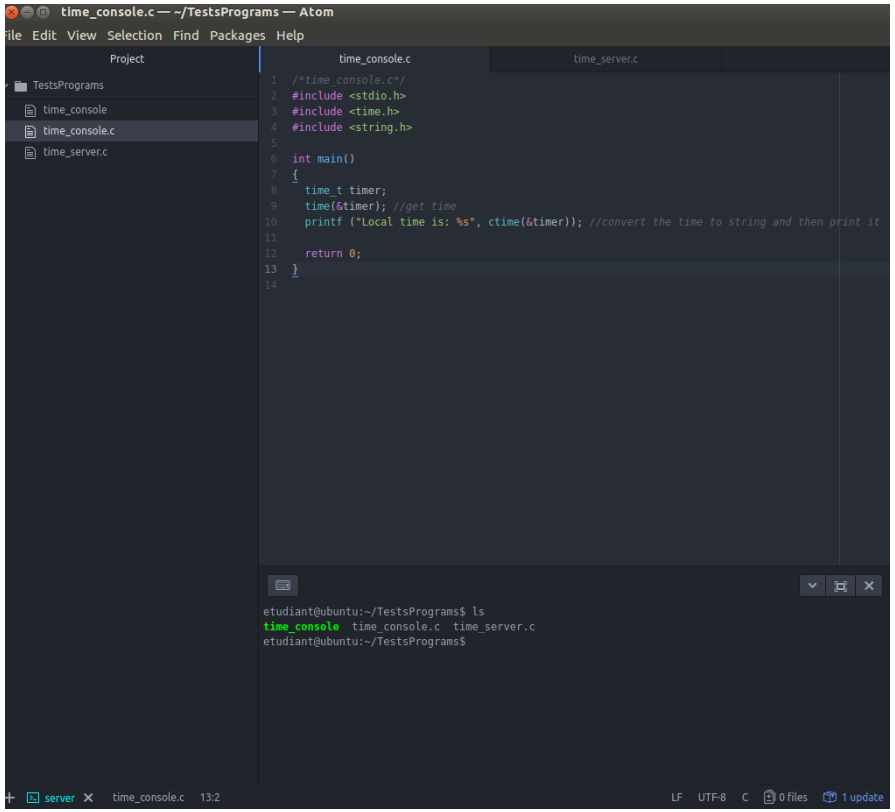
Example: Create a web server

In this example you will implement **step by step a web server**. Client web page **send a request** (or connect) and the server answer giving the current date and hour.

Ensure that you can open Atom IDE and create a new work folder and a c file of your program.

If **platformio-ide-terminal** plugin is correctly installed, you can add new terminals.





Example: Create a web server

1) Implement the simple time server :

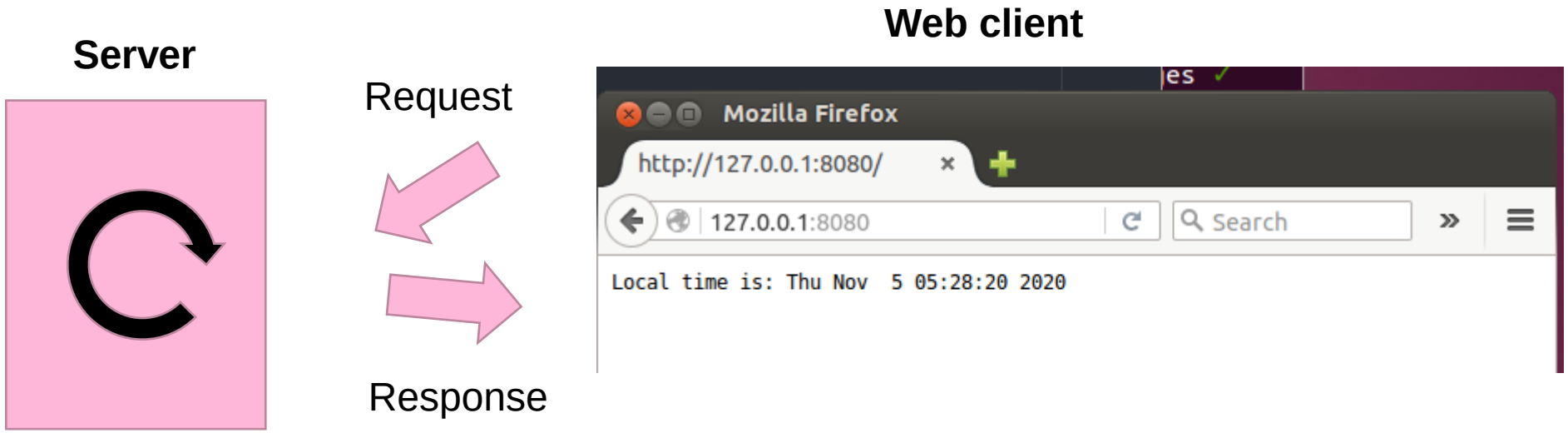
```c
/*time_console.c*/
#include <stdio.h>
#include <time.h>
#include <string.h>

int main()
{
  time_t timer;
  time(&timer); //get time
  printf ("Local time is: %s", ctime(&timer)); //convert the time to string and then print it
  return 0;
}
```

2) Compile and execute ***time_console.c***

```
TestsPrograms$ gcc time_console.c -o time_console
[redacted]:~/TestsPrograms$ ls
time_console  time_console.c  time_server.c
[redacted]:~/TestsPrograms$ ./time_console
Local time is: Thu Nov  5 05:19:26 2020
[redacted]:~/TestsPrograms$
```

Example: Create a web server

Client/server interaction



Example: Create a web server

Steps to implement a server:

1) **Import libraries and create macros**
2) **Socket settings: configurate the local address on which the server listen to client requests**
3) **Create a socket**
4) **Socket binding**
5) **Listening for client's connection**
6) **Socket acceptance: waiting for client's connection**
    1) Send a server response back.
    2) Access to the client request (print data)
7) **Close socket connection**

Example: Create a web server

1) Import libraries and create macros

```
#include <stdio.h>
#include <time.h>
#include <string.h>
```

Include the librairies of simple c program

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>
```

Include the needed libraries of a network program with sockets

Example: Create a web server

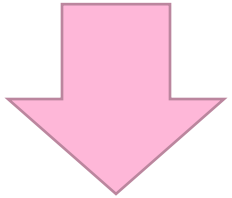
1) Import libraries and create macros

```
//Define Macros to maninpulate sockets

#define ISVALIDSOCKET(s) ((s) >= 0) // Check and return 1 if the socket is valid

#define CLOSESOCKET(s) close(s) // Close a socket

#define SOCKET int // initialize the socket

#define GETSOCKETERRNO() (errno) //Manage errors try the command "$ man errno" to learn more
```

Example: Create a web server

2) Configuration of the local address on which the server listen to the client requests (1st part)

```
int main() {

    printf("Configuring local address...\n");

    struct addrinfo hints; // addrinfo structure with hints or indicator information

    memset(&hints, 0, sizeof(hints)); //We zeroed out hints using memset() first.

.
.
.
```



Example: Create a web server

2) Configuration of the local address on which the server listen to the client requests (2nd part)

```
hints.ai_family = AF_INET; //We are looking for an IPv4 address. AF_INET6 to make our web server listen on an IPv6 address instead

hints.ai_socktype = SOCK_STREAM; //We're going to be using TCP. SOCK_DGRAM would be used if we were doing a UDP server instead

hints.ai_flags = AI_PASSIVE; //We want getaddrinfo() to bind to the wildcard address. We listen on any available network interface.

struct addrinfo *bind_address; // A pointer to a struct addrinfo structure, which holds the return information from getaddrinfo().

getaddrinfo(0, "8080", &hints, &bind_address); //getaddrinfo() to fill in a structure addrinfo with the needed information. 0 is a node value and 8080 is the port number of a service
```

Example: Create a web server

3) Create a socket

```
printf("Creating socket...\n");

SOCKET socket_listen; //we define socket_listen as a SOCKET type. Macro defining it as int

socket_listen = socket(bind_address->ai_family,
                       bind_address->ai_socktype,
                       bind_address->ai_protocol);

//check that socket_listen is valid using the ISVALIDSOCKET() macro we defined earlier.
if (!ISVALIDSOCKET(socket_listen)) {

    // print an error message, GETSOCKETERRNO() macro retrieves the error number
    fprintf(stderr, "socket() failed. (%d)\n", GETSOCKETERRNO());

    return 1; //Exit the program with error message
}
```

Example: Create a web server

4) Socket binding

```
printf("Binding socket to local address...\n");

//we call bind() to associate the socket with our address from getaddrinfo()

if (bind(socket_listen, bind_address->ai_addr, bind_address->ai_addrlen)) {
        fprintf(stderr, "bind() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
}

freeaddrinfo(bind_address); //we call the freeaddrinfo() function to release the address memory.
```

Example: Create a web server

5) Start listening for client's connection

```
//Once the socket has been created and bound to a local address, we can cause it to start listening for connections with listen()

printf("Listening...\n");

if (listen(socket_listen, 10) < 0) { //10 tells listen() how many connections is allowed to queue up.

    fprintf(stderr, "listen() failed. (%d)\n", GETSOCKETERRNO()); //error when listen() return a value

return 1;

}
```

Example: Create a web server

6) Socket acceptance: start waiting for client's connection

```
printf("Waiting for connection...\n");
struct sockaddr_storage client_address;
socklen_t client_len = sizeof(client_address);

//We store the return value of accept() in socket_client.
//We declare a new struct sockaddr_storage to store the address info for the connecting client.
//client_len with the length of that address.

SOCKET socket_client = accept(socket_listen,  //it will block your program until a new connection is made.
                              (struct sockaddr*) &client_address,
                               &client_len);

//Just check if everything is ok with accept()
if (!ISVALIDSOCKET(socket_client)) {
  fprintf(stderr, "accept() failed. (%d)\n", GETSOCKETERRNO());
  return 1;
}
```

Example: Create a web server

6-1) After the TCP connection is established, we can print the client IP address

```
//TCP connection has been established to a remote client.
printf("Client is connected... ");
char address_buffer[100];

//takes the client's address and address length. The address length is needed because
//getnameinfo() can work with both IPv4 and IPv6 addresses
getnameinfo((struct sockaddr*)&client_address, client_len,
                                address_buffer, sizeof(address_buffer),
                                0, 0,  NI_NUMERICHOST); //This flag specifies that we want to see the hostname as
                                                        //an IP address.

printf("%s\n", address_buffer); // print the content of address_buffer
```

Example: Create a web server

6-2) Send a server response back (1st part)

```
printf("Sending response...\n");
const char *response = "HTTP/1.1 200 OK\r\n" //Tells the client that the request is OK, with (empty) blank line
                       "Connection: close\r\n" //the server will close the connection
                       "Content-Type: text/plain\r\n\r\n" //data sent plain text
                       "Local time is: ";

// send the data and return thenumber of bytes sent
int bytes_sent = send(socket_client, response, strlen(response), 0);

printf("Sent %d of %d bytes.\n", bytes_sent, (int)strlen(response));
```

Example: Create a web server

6-2) Send a server response back. The HTTP header and the beginning of our message are sent, (2nd part)

```
time_t timer;

time(&timer);

char *time_msg = ctime(&timer);

bytes_sent = send(socket_client, time_msg, strlen(time_msg), 0); //use of send() function

printf("Sent %d of %d bytes.\n", bytes_sent, (int)strlen(time_msg));
```

Example: Create a web server

7) Close a connection

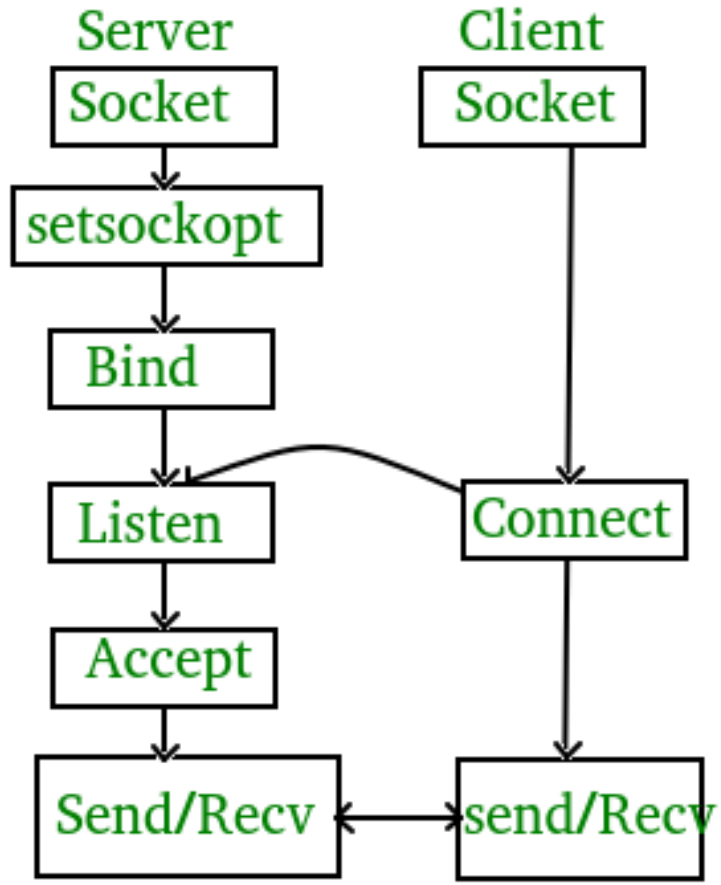
```
//Close the client connection to indicate to the browser that we've sent all of our data:

printf("Closing connection...\n");
CLOSESOCKET(socket_client);

}
```

TCP Client

TCP Client

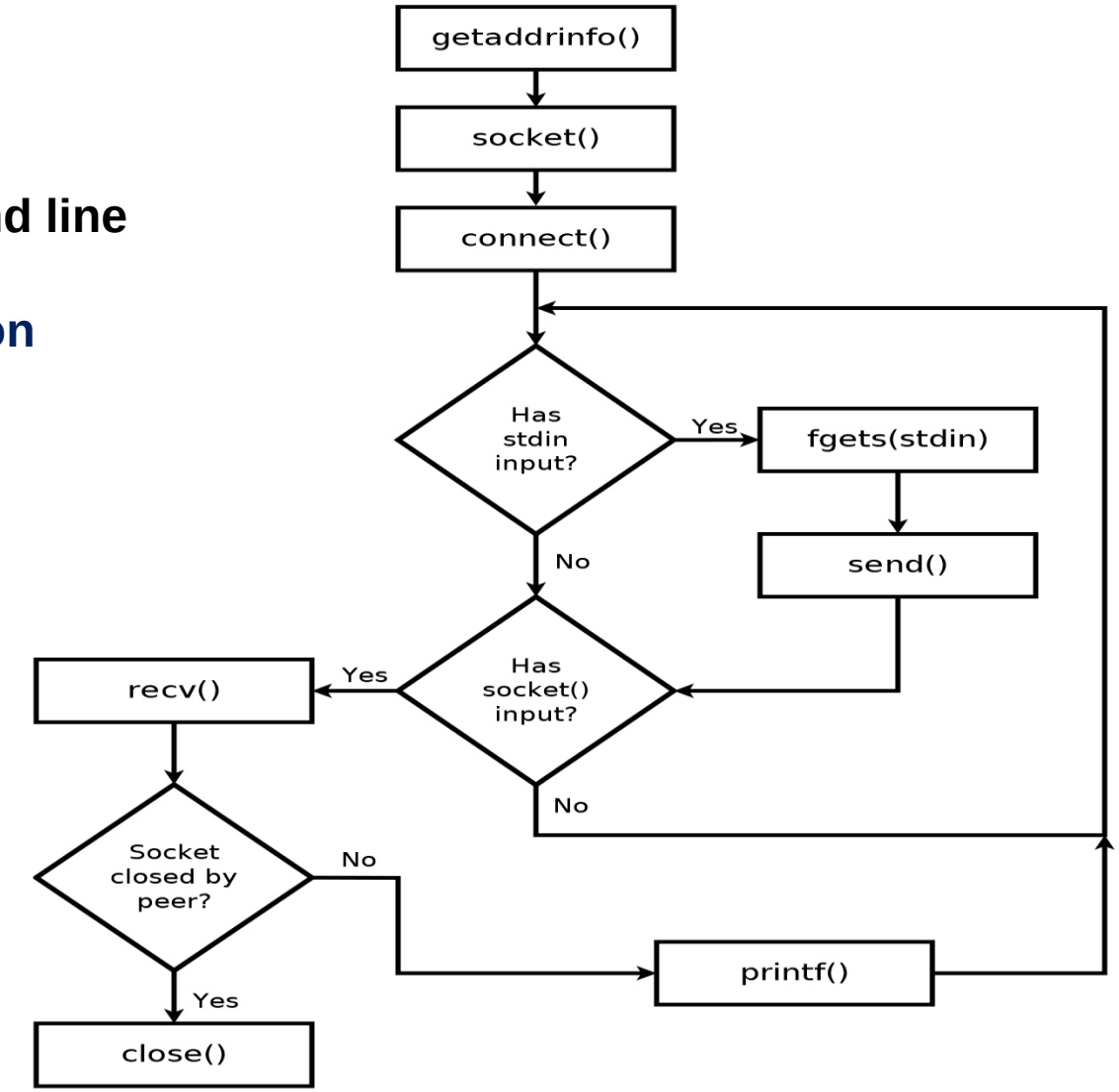
It will be useful for us to have a TCP client that can connect to any TCP server. This TCP client will take in a hostname (or IP address) and port number from the command line. It will attempt a connection to the TCP server at that address.

The program first uses **getaddrinfo()** to resolve the server address from the command line arguments. Then, the socket is created with a call to **socket().**
The socket has **connect()** called on it to connect to the server. We use **select()** to monitor for socket input. **select()** also monitors for terminal/keyboard input.
If terminal input is available, we send it over the socket using **send().** If **select()** indicated that socket data
is available, we read it with **recv()** and display it to the terminal. This **select()** loop is repeated until the socket is closed.



TCP Client

1) **Import libraries and create macros**
2) **Check the arguments of execution command line**
3) **Configurate a remote address for connection**
4) **Create socket**
5) **Establish the connection**
6) **Waiting for an event on socket : select()**
7) **Send request and receive data**
8) **Close socket**



TCP Client

1) Import libraries and create macros

```
#include <stdio.h>
#include <time.h>
#include <string.h>
```

Include the librairies of simple c program

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>
```

Include the needed libraries of a network program with sockets

TCP Client

1) Import libraries and create macros

```
//Define Macros to maninpulate sockets

#define ISVALIDSOCKET(s) ((s) >= 0) // Check and return 1 if the socket is valid

#define CLOSESOCKET(s) close(s) // Close a socket

#define SOCKET int // initialize the socket

#define GETSOCKETERRNO() (errno) //Manage errors try the command "$ man errno" to learn more
```

TCP Client

2) Check the arguments of execution command line

Client program takes the **hostname** and **port number** of the server it should connect to as command-line arguments. We have our program check that these command-line arguments are given. If they aren't, it displays guidance information:

```
int main(int argc, char *argv[]) {
// argc contains the number of argument values available in the execution command
// The actual values themselves are stored in argv[]

    if (argc < 3) {
        fprintf(stderr, "usage: tcp_client hostname port\n");
        return 1;
    }
```

TCP Client

3) Configurate a remote address for connection

```
printf("Configuring remote address...\n");

struct addrinfo hints;  //similar to how we called getaddrinfo() in the server part
memset(&hints, 0, sizeof(hints));  // whereas this time, we want it to configure a remote address
hints.ai_socktype = SOCK_STREAM; //we want a TCP connection
struct addrinfo *peer_address;

//the hostname and port are two arguments passed directly in from the command line
// If everything goes well, then our remote address is in the peer_address variable.
if (getaddrinfo(argv[1], argv[2], &hints, &peer_address)) {
        fprintf(stderr, "getaddrinfo() failed. (%d)\n", GETSOCKETERRNO()); //else manage error
        return 1;
}
```

TCP Client

4) Create socket

```
printf("Creating socket...\n");
SOCKET socket_peer; //This call to socket() is done in exactly the same way as it was in the server
socket_peer = socket(peer_address->ai_family, peer_address->ai_socktype, peer_address->ai_protocol);
if (!ISVALIDSOCKET(socket_peer)) {
        fprintf(stderr, "socket() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
}
```

TCP Client

5) Establish the connection

```
printf("Connecting...\n");
//connect() takes three arguments—the socket, the remote address, and the remote address length.
if (connect(socket_peer, peer_address->ai_addr, peer_address->ai_addrlen)) {
        fprintf(stderr, "connect() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
}
freeaddrinfo(peer_address); //free the memory for peer_address.
```

TCP Client

6) Waiting for an event on socket : select()

```
printf("To send data, enter text followed by enter.\n");
while(1) { //We begin our loop and set up the call to select()

        fd_set reads; //to store our socket set.
        FD_SET(socket_peer, &reads);
        FD_SET(0, &reads);//We then zero the "reads". Use "man 2 FD_ZERO to learn more"

        //we use select() to monitor for terminal input.
        if (select(socket_peer+1, &reads, 0, 0, NULL) < 0) {
                fprintf(stderr, "select() failed. (%d)\n", GETSOCKETERRNO());
                return 1;
        }
.
.
```

TCP Client

7) Send request and receive data

```
.
.

if (FD_ISSET(socket_peer, &reads)) { //check to see whether our socket is set in reads.
        char read[4096];
        int bytes_received = recv(socket_peer, read, 4096, 0); //call recv() to read the new data.
        if (bytes_received < 1) {
                printf("Connection closed by peer.\n");
                break;
        }
        printf("Received (%d bytes): %.*s", bytes_received, bytes_received, read);
}
.
.
```

TCP Client

7) Send request and receive data

```
        .
        .
        if(FD_ISSET(0, &reads)) { //send request
                char read[4096];
                if (!fgets(read, 4096, stdin)) break; //includes the newline character from the input.
                printf("Sending: %s", read);
                int bytes_sent = send(socket_peer, read, strlen(read), 0);
                printf("Sent %d bytes.\n", bytes_sent);
        }
} //end while(1)
```

TCP Client

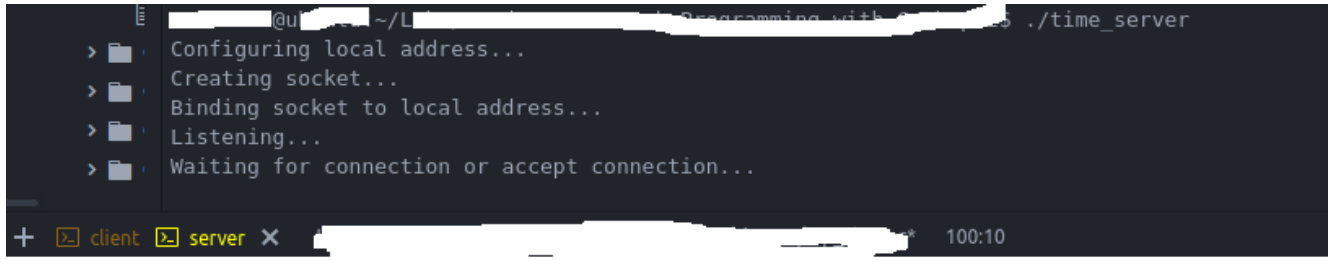
8) Close socket

```
    printf("Closing socket...\n");
    CLOSESOCKET(socket_peer);
    printf("Finished.\n");
    return 0;
} // end of main
```

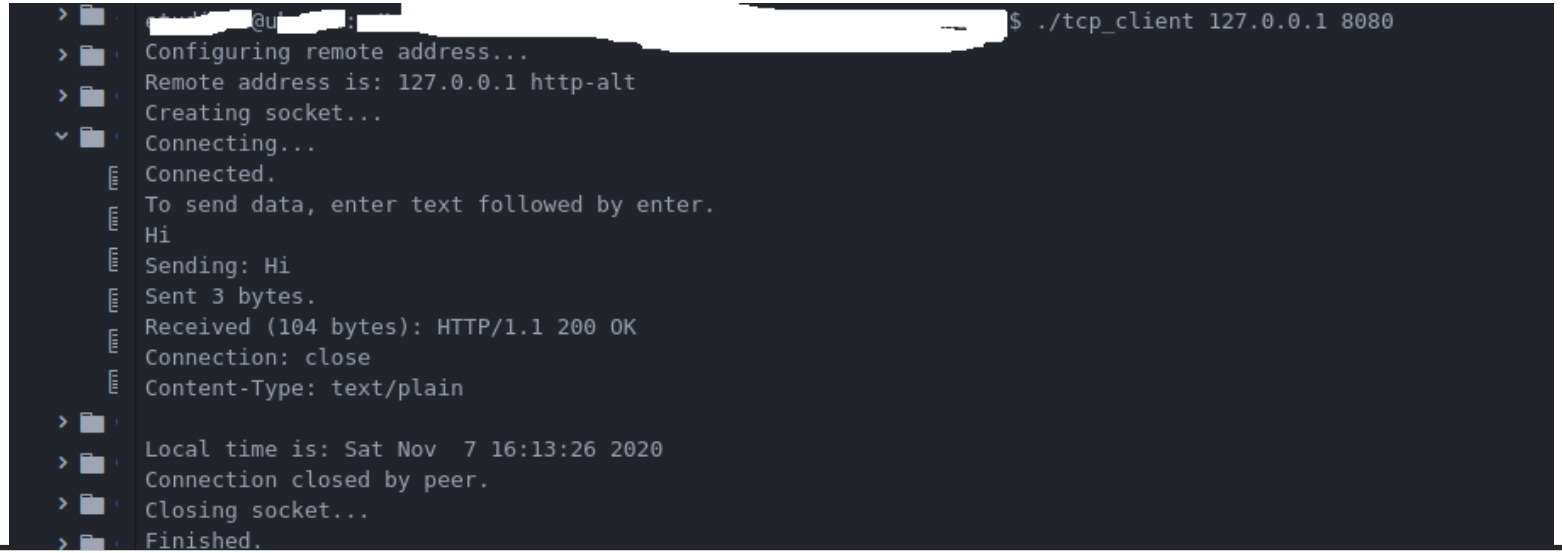
TCP Client/Server

Now, you can run the server and then the client in the separate terminals (shell).

Server



Client



References

Lewis Van Winkle, Hands-On Network Programming with C, Published by Packt Publishing Ltd, UK, 2019