Practical Guide - Introduction to Sockets

Objectives:

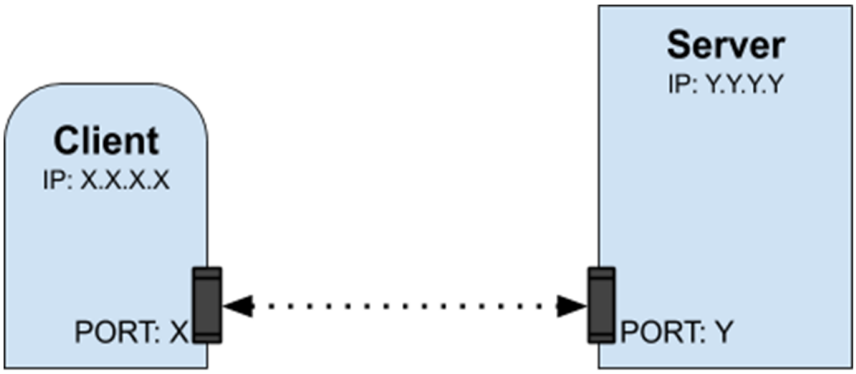
- understand the need for sockets in communications;
- learn the basic course of actions to establish communication between two machines;
- use the knowledge gathered in sockets to create a basic chat room application in c.

Requirements:

- This guide is defined with instructions for Linux. You can do it on windows, but it does not have instructions for compilation and execution.
- You need a c compiler, this guide uses gcc (to install it, run sudo apt install gcc in the terminal).
- You Also need a user account with administrator permissions. If you can execute a terminal command starting with sudo (example sudo ls), that user has the necessary permissions.
- If possible, use two computers to run the code to see the communication between two different machines; otherwise, one computer is enough, but you will only see the communication between two programs.

Introduction:

A socket is the interface that allows communications between machines connected to a network.
Without sockets, you would not be able to surf on the internet or link your computer with any network. And, even if you could, imagining the whole internet had been built without a concept like sockets, we would need to constantly and exhaustively monitor all the information of each host with which we intend to communicate.
Thus, sockets provide us an easy way to store all the information needed to reach the intended host in one place.
The two most important pieces of information stored in the socket of an ongoing communication with a given host is the **IP address** and the **port** on which it is listening.



The figure above shows the common use of sockets. On the server, there is a socket, linked **to a port number and the machine's IP**, which is actively listening for new connection requests from clients. When a client wants to communicate with a server, he **must know the server's IP and port in advance** to communicate. Then, it creates a socket with the information to access the server and uses it whenever it wants to send/receive data. On the server side, when it receives a new connection from a client, it stores a socket that represents the client and uses that socket to communicate with him.
To summarize, in a basic communication link, **the client only has one socket**, with the server information. But **the server has two sockets**, one linked to its IP and a port, from which it will receive new connection requests, and the other represents the client, through which the server-client communication can take place.
Before describing the types of sockets available, you should be aware that **a socket is nothing more than a file descriptor**, that is, an integer associated with a file or any input/output resource.

Types of sockets:
There are two mainly used sockets: **SOCK_STREAM** , used for the **TCP** communications and **SOCK_DGRAM**, for the **UDP** communications. These two types will be worked on **Group I** of this guide. However, whenever you want, for instance, to have complete control of the packets exchanged and do not want to be stuck in one of the two previous sockets, you can also use SOCK_RAW, raw sockets, they operate in the **OSI network layer** and, therefore, has **no association with a port number**, this makes these sockets extremely flexible and interesting. In **Group II**, you will have the opportunity to see these types of being used. In addition to these three, there are several other types of sockets, but they are not commonly used, so they will not be covered in this guide.

Work to do:

TCP/UDP Sockets

UDP Sockets:

From the available files, download **test_client_UDP.c** and **test_server_UDP.c**.

If possible, run the client and the server in two different computers, to see the communication between two different machines; otherwise, use the same computer, but you will only see communication between two different programs.

1. Start by **analysing the code**, its order and structure of commands.
2. Now that you have read and analysed the code, let's run it. For that, you need to **compile** it.
   Open terminal and go to the folder where you have **test_client_UDP.c** and **test_server_UDP.c**.
   To compile **test_client_UDP.c** use the following command in the terminal:
   ```
   $ gcc -o client test_client_UDP.c
   ```
   For the **test_server_UDP.c** use:
   ```
   $ gcc -o server test_server_UDP.c
   ```

Two computers:

3. In the case where you are using **two computers,** run the following line in the terminal to find the IP address of the server computer:
   ```
   $ hostname -I
   ```
   The previous command will provide all IPv4 and IPv6 addresses of the server, this information will be necessary for the client.
   Execute the following command on terminal to run server code.
   ```
   $ ./server
   ```
4. Now that you have the server running, run the client.
   Select one of ipv4 given by the hostname command and run the following code in the client computer (replace IPv4address with the server's IP):
   ```
   $ ./client IPv4address 56789
   ```

One computer:

3. If using only **one computer** execute the following command on the terminal to run server code:
   ```
   $ ./server
   ```
4. And to run the client open a second terminal or use **ctrl+shift+T** to open a new tab in terminal. Where you can run the following code:
   ```
   $ ./client 127.0.0.1 56789
   ```

Note: the two inputs provided to the client are the server's IP address and PORT where he is listening.

5. Relate the sequence of commands with the type of socket.
6. Now download **test_client_TCP.c** and **test_server_TCP.c** and perform steps 1 through 5 again with this code.
7. **Relate the differences** in commands with what you know from these two data transfer protocols.
8. Compare the required input arguments in the **send()/recv()** and **sendto()/recvfrom()** functions and then relate it with **connection-oriented** and **connectionless** packet switching.

Part II:

For this part you'll need to learn a new helpful function available in c:

- `int fork();`

When called, this function divides the current process in two, creating a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.
The code, after the function call, is executed twice, in parallel.
The int value returned by the fork function is a positive number in the parent process, which corresponds to the child process ID and '0' on the child process.

Basing on the **examples presented in Part I**, you will develop a client application capable of sending and receiving messages simultaneously, using TCP sockets. To do that, copy the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>
#define BUFFERSIZE 1024

int main(int argc, char *argv[])
{
  int pid;
  // add here the declarations of variables needed

  // create a TCP socket and connect to the server

  pid=fork();
  while(1==1){
    if (pid==0){

      // Ask user which message he want to send here

    }else{

      // Receive any message from the server and print it here

    }
  }
  close(sockfd);
  return 0;
}
```

To test your client application, download and run the executable file "**server_TestClient** ". This program creates a server on port **56789** that periodically sends a "poke" message and sends the last message received from the client. To complete the code provided, it must have the following specifications:

1. The main must receive **only** one input argument, the server IP address;
2. Whenever a string is written on the console, it must be sent as a message to the server;
3. All messages received from the server must be displayed on the console.

Part III:

After creating a client that can send and receive messages in parallel, you will create a server that can handle multiple client connections. For that, you will need to be introduced to a new concept:

- **Master Socket**

When a new client wants to make first contact with a server, **the server itself** must have a default socket to connect with that client. This socket is commonly called **Master Socket .**
The sole purpose of the **Master Socket** is to serve as a reference that the new incoming client uses to connect to the server, which is listening, without this fixed socket, a new client would not know how to access the server.
After the **Master Socket** is created and linked to the server's **IP** and **PORT number**, clients can connect to it, the server will store that socket next to all the other stored sockets from all the other clients (in a socket array, for example).

Based on the **examples presented in Part I** and using the concepts introduced in this part, create a **TCP server**, running on port **56789** that can accept connections from clients through a **Master socket**.
The server must have the following features:

1. Always be ready to accept a new connection from a new incoming client.
2. Save the sockets created for each client in a socket array. Each client must have an unique **Host ID**.
3. When a socket is created for a new client, the following actions must be performed:
    a. Send a welcome message to the new client and inform him of his **Host ID**: **"Welcome to the server, your Host ID is #"**.
    b. Inform the new client about the total number of clients connected to the server, including himself.
    c. Announce to all the clients that a new client has just connected to the server by sending the message: **"Host # has just connected to the server".**
    d. Print new connections to the console in the format: **"Host # just connected."**.

Test your server with several clients using the program created in **Part II** (using only its receiving feature).

Part IV:

After completing the server and testing the server in **Part III**, make the necessary changes to create a server that can receive messages from any connected client and redirect the messages to all the other clients. Essentially, you will be creating a chat room!
As you know, there are several approaches that can be implemented to redirect the messages to multiple machines (unicast, multicast, broadcast). In this practical guide, you will use unicast (the same approach used so far in this guide).
To do this, you will need to be introduced to the **select()** function and the **file descriptor set** from the socket API:

- **int select(int *nfds*, fd_set **readfds*, fd_set **writefds*, fd_set **exceptfds*, struct timeval **timeout*);**

The **select()** function allows a program to monitor multiple file descriptors previously added into a **file descriptor set** (and **a socket in C is a file descriptor**), waiting until one or more of the file descriptors are "ready" for some class of I/O operation.
In the context of socket programming and this practical guide, the **select()** function is used to monitor the activity on the **Master Socket** (thus monitoring the incoming new client connections) or in any other socket from a **connected client** (being able to detect message requests).

Function inputs:

- ***nfds* -** the file descriptor with the highest number in any of the three sets, plus 1.
- ***readfds* -** set of file descriptors that will be observed to see if any of them have characters available for reading.
- ***writefds* -** set of file descriptors that will be observed to check if space is available for writing. It must be set to ***NULL*** in the context of this practical guide.
- ***exceptfds* -** set of file descriptors that will be observed for exceptional condition. It should be set to ***NULL*** in the context of this practical guide.
- ***timeout* -** determines the maximum time that the **select** function is blocking.

Function output: The **select()** function returns the total number of bits that are set in ***readfds***, ***writefds***, ***exceptfds***. Therefore, a **positive number** is returned if there is any activity on any of the descriptor sets. **Zero** is returned if the defined timeout is finite and there was no activity in any of the descriptor sets, and in case of error, **-1** is returned, and ***error*** must be set appropriately.
To initialize a descriptor set:

- **fd_set** setName;

The macros you'll need to manipulate the descriptor sets:

- **FD_ZERO(&set) :** clears the set (needs to be done after a new select() call);
- **FD_SET(socket, &set) :** adds **socket** to the **set**;

- **FD_CLR(socket, &set)** : removes **socket** from the **set**;
- **FD_ISSET(socket, &set)** : after a select() call on **set**, returns **1** if the **socket** was the file descriptor from the **set** who received a new message, or **0** if it received nothing in the **socket** (**socket** is an element in the **set**).

After being introduced to **select()** and to **file descriptor sets**, you are now ready for the next task, to do that, copy the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT        56789
#define BUFFERSIZE  1024
#define SOCKSIZE    100

int main( int argc, char *argv[] ) {
  static fd_set rfds;
  static int retval;
  static int max;
  struct timeval timeout;
  int poke = 10;
  static int sockfd;
  // add here the declarations of variables needed

  // create a TCP socket and connect to the server

  listen(sockfd,5);
  max=sockfd;
  timeout.tv_sec = poke;
  timeout.tv_usec = 0;
  while(t==0){
    FD_ZERO(&rfds);
    FD_SET(sockfd, &rfds);

    // add other sockets to descriptor set

    retval=select(max+1,&rfds,NULL,NULL,&timeout);
    if (retval < 0)
    {
      perror("ERROR on select");
      exit(1);
    }
    if(retval==0)
```

```
    {
      timeout.tv_sec = poke;
      timeout.tv_usec = 0;

      // send periodic message

    }
    else
    {
      if(FD_ISSET(sockfd, &rfds)){

        // accept a new connection

      }

      // verify if any other socket received any message

    }
  }
  close(sockfd);
  return 0;
}
```

Complete some of the missing parts of the previous code with the code that you made for Part III, and add the following functionality:

1. When the server receives a message from a connected client, the server sends **individual** messages to all clients, **except for the sending client**, by sending the following string: **"Host #: (message)"**, where **#** is the unique **Host ID** of the client that sent the message.
2. Print to the console all messages exchanged, also in the format **"Host #: (message)"**.

To the client from **Part II**, add the following functionality:

1. When you print something on the console, keep a string saying **"You: "** on the bottom line, so that clients can know where to type new messages and identify which ones are theirs.

Test your server with multiple clients and see if everything is working as expected.

Part V:

Your chatroom is almost ready! Only one detail missing. Users must be able to disconnect from chat and the other users must be notified.
Let's learn a bit more:

Identifying a socket closure/disconnection:
When a client socket is **close()** (for example, when a client uses the keyboard interrupt on a console, **Ctrl+C**), on the server side, a message of size **0** is received, so the server detects that the user disconnected with a condition like **recv(...)==0.**

Use this information to add to your server from **Part IV** the following functionality:

1. When a host disconnects, delete its entry in the socket array and send the string **"Host # just disconnected."** to all connected clients and print it on the console.

Test your server and client and see if everything is acting as expected:

1. Test with several clients.
2. Test with two computers if possible: One runs both the server and the client program, the other runs another client (remember that you will need to provide the IP address of the server for each client).

GROUP II - RAW Sockets

As suggested in the intro, a RAW Socket is a type of socket that allows access to the underlying transport provider. Unlike programming using TCP or UDP where only the application protocol header and data provided by the application were built by the application, RAW socket programming allows headers of lower level protocols to be built and provided by the application.
Most protocols do not support this type of sockets because, to use RAW sockets, the application must have all the detailed information of the fundamental protocol being used. Applications capable of opening RAW sockets must filter all information and packets to separate useful and relevant information from all available information.
In the next two parts, you will see two brief examples of programs using raw sockets, which serve as a demonstration of what you can do with them.

Part I:

The following code is necessary in order to read a RAW Socket.
Analyse the code and **identify** the main differences from what you know about sockets presented in **Group I** of this guide.

```
// raw_sock.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<netinet/ip.h>
#include<sys/socket.h>
#include<arpa/inet.h>

int main(){
      // Structs that contain source IP addresses
      struct sockaddr_in source_socket_address, dest_socket_address;

      int packet_size;

      // Allocate string buffer to hold incoming packet data
      unsigned char *buffer = (unsigned char *)malloc(65536);
      // Open the raw socket
      int sock = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);
      if(sock == -1)
```

```
      {
        //socket creation failed, may be because of non-root privileges
        perror("Failed to create socket");
        exit(1);
      }
      while(1) {
        // recvfrom is used to read data from a socket
         packet_size = recvfrom(sock , buffer , 65536 , 0 , NULL, NULL);
         if (packet_size == -1) {
           printf("Failed to get packets\n");
           return 1;
         }
         struct iphdr *ip_packet = (struct iphdr *)buffer;
         memset(&source_socket_address, 0, sizeof(source_socket_address));
         source_socket_address.sin_addr.s_addr = ip_packet->saddr;
         memset(&dest_socket_address, 0, sizeof(dest_socket_address));
         dest_socket_address.sin_addr.s_addr = ip_packet->daddr;

         printf("Incoming Packet: \n");
         printf("Packet Size (bytes): %d\n",ntohs(ip_packet->tot_len));
         printf("Source
Address:%s\n",(char*)inet_ntoa(source_socket_address.sin_addr));
         printf("Destination
Address:%s\n",(char*)inet_ntoa(dest_socket_address.sin_addr));
         printf("Identification: %d\n\n", ntohs(ip_packet->id));
      }

      return 0;
}
```

1. Copy the code to a file called **raw_sock.c**. Compile it with the following command line:
   $ gcc -o raw_socket raw_sock.c
2. Try to **run** the code with: **./raw_socket**
3. The code does not run. It generates the following error: *`Failed to create socket: Operation not permitted`*. Can you give any reason for this to happen.
4. Now run the code with administrator permissions, to do that run the following command: **sudo ./raw_socket**
5. Explain why you need administrator permission to execute the code.

Now that you can run it, what did you observe?
Note that you have a code **capable of receiving all packets destined to you**.
Additionally, with your network card in **promiscuous mode**, you can receive **any** packet on your network and thus, for instance, you can develop a basic Wireshark.

Part II:

A **ping** application does not use **TCP** nor **UDP**; therefore, if we want to implement it, we can do it using RAW sockets!
Analyse the **ping method** from the following example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <net/if.h>
#include <netinet/ip.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#include <errno.h>
 enum {
      DEFDATALEN = 56,
      MAXIPLEN = 60,
      MAXICMPLEN = 76,
      MAXPACKET = 65468,
      MAX_DUP_CHK = (8 * 128),
      MAXWAIT = 10,
      PINGINTERVAL = 1, /* 1 second */
};

static int in_cksum(unsigned short *buf, int sz);

static void ping(const char *host)
{
    struct hostent *h;
    struct sockaddr_in pingaddr;
    struct icmp *pkt;
    int pingsock, c;
    char packet[DEFDATALEN + MAXIPLEN + MAXICMPLEN];
    static char *hostname;
    if ((pingsock = socket (AF_INET, SOCK_RAW, 1)) < 0) {
      // Create socket, 1 == ICMP/
      perror("ping: creating a raw socket");
      exit(1);
    }
    pingaddr.sin_family = AF_INET;
    if (!(h = gethostbyname(host))) {
       fprintf(stderr, "ping: unknown host %s\n", host);
       exit(1);
    }
```

```
    memcpy(&pingaddr.sin_addr, h->h_addr,sizeof(pingaddr.sin_addr));
    hostname = h->h_name;

    //Build the icmp packet + checksum, send it
    pkt = (struct icmp *) packet;
    memset(pkt, 0, sizeof(packet));
    pkt->icmp_type = ICMP_ECHO; //Set ICMP type
    pkt->icmp_cksum = in_cksum((unsigned short *) pkt,sizeof(packet));
    c = sendto(pingsock, packet, sizeof(packet), 0, (struct sockaddr *)
&pingaddr, sizeof(struct sockaddr_in));

    /* listen for replies */
    while (1) {
      struct sockaddr_in from;
      int fromlen = sizeof(from);
      if ((c = recvfrom(pingsock, packet, sizeof(packet), 0, (struct
sockaddr *) &from, &fromlen)) < 0) {
           if (errno == EINTR)
                continue;
           perror("ping: recvfrom");
           continue;
      }
      if (c >= 76) {
           /* ip + icmp */
           struct iphdr *iphdr = (struct iphdr *) packet;
           pkt = (struct icmp *) (packet + (iphdr->ihl << 2));
           /* skip ip hdr */
           if (pkt->icmp_type == ICMP_ECHOREPLY)
                break;
      }
    }
    printf("%s is alive!\n", hostname);
    return;
}

int main (int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(stderr,"usage %s hostname/IP\n", argv[0]);
        exit(0);
    }
    ping(argv[1]);
```

```
}
static int in_cksum(unsigned short *buf, int sz)
{
    int nleft = sz;
    int sum = 0;
    unsigned short *w = buf;
    unsigned short ans = 0;

    while (nleft > 1) {
       sum += *w++;
       nleft -= 2;
    }

    if (nleft == 1) {
       *(unsigned char *) (&ans) = *(unsigned char *) w;
       sum += ans;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    ans = ~sum;
    return ans;
}
```

In the code presented, **identify** the sections of the **ping** method that translate to:

1. Find host name, address or any aliases for host;
2. Create the socket address for the ping target;
3. Builds the ICMP Header;
4. Pings and checks if there is connection or not.

Run the code and check its functionality.