Systems programming

Week 3 – Lab 5

Client-Server with Internet datagram sockets

The Linux Programming Interface Chapter 58

Internet domain sockets allow the communication of programs running in multiple computers using the Internet.

The way to establish communication using Internet domain sockets is similar with the use of datagram sockets. The main different is the way the socket addresses (identification of the recipient socket) are defined.

For two processes to communicate with Internet domain sockets it is necessary to create one socket on each program: each communication participant should create a socket.

To receive data from another remote socket it is necessary to assign/bind an address. This can be done explicitly with the bind function or implicitly by the **sendto**. In internet domain sockets, after sending data with the **sendto** function, such socket is ready to receive messages. In this case the recipient of the message will get an address that was implicitly assigned/bind during the **sendto**.

1 Internet domain sockets addresses

The Linux Programming Interface - section 58.5 59.4

The Linux Programming Interface - sections 59.2 59.6

In Unix domain socket, addresses are composed of a string (file path). In Internet domain sockets an address is composed of two distinct parts:

- the IP address (for instance 146.193.41.15)
- the port (for instance 80)

This information is stored in a structure of type (**struct sockaddr_in server_addr**) that contains in binary format this information. The relevant fields of this structure are:

- server_addr.sin_family
- server_addr.sin_port

- server_addr.sin_addr

1.1 Internet domain address initialization

The generic code to set the **struct sockaddr_in** structure is presented in the following example:

```
1. struct sockaddr_in server_addr;
2. server_addr.sin_family = AF_INET;
3. server_addr.sin_port = htons(SOCK_PORT);
4. inet_pton(AF_INET, linha, &server_addr.sin_addr);
```

Line 1 defines the structure that will contain the server address, in this case the type is sockaddr_**in** , as opposed to sockaddr_**un** for UNIX domain sockets.

The type of the address (defined in line 2) is also different:

- the filed is **sin_family**
- the value is **AF_INET**

In the Internet domain each socket is identified by a unique pair of address and port.

The port should be unique on the computer and for regular applications usually ranges between 4000 and 40000. In our example this value is defined in a constant as 5000.

The address of the remote computer (to be put in **server_addr.sin_addr**) must be discovered or communicated to the client. The function **inet_pton** converts from a string in the **n.n.n.n** format into binary (4 bytes). If the format of the supplied string correct and the address is correctly converted this function returns 1.

On the server, it is not necessary to assign the specific computer address in the **sin_addr** field. If the computer has multiple addresses, the programmer can use the INADDR_ANY to allow reception of messagens on all network cards:

```
local_addr.sin_addr.s_addr = INADDR_ANY;
```

In the Internet, values with 16 bits or larger (2 or more bytes) may be represented in different ways in different computers. The two ways are big-endian and little-endian. To

solve this possible differences it is necessary to call the functions **htons** and **ntohs** when assigning or reading a port number.

1.2 Available addresses in computers

Computer connected to the Internet can have multiple addresses.

One of such addresses is the **127.0.0.1**. All computers have this address and, if used in a client, the program will try to communicate with a server running on the same computer.

To accept remote connections it is necssary to tell the clients what are the available addresses on the computer that runs the server. To know such addresses there are a few commands to be executed on the server computer:

- in Linux – **ip** a or **ifconfig**
- MAC OS X – **ifconfig**
- Windows – **ipconfig**

These command should be executed in a terminal or command prompt window and will produce the following outputs.

ip a

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 . . .  qlen 1000
. . . .
    inet 127.0.0.1/8 scope host lo
. . . .
2: enp4s0f0: <BROADCAST,MULTICAST,UP,LOWER_UP> . . . qlen 1000
. . . .
    inet 146.193.41.15/24 brd 146.193.41.255 . . .  enp4s0f0
. . . .
```

ip a (Linux / WSL)

```
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
. . . .
   inet 127.0.0.1 netmask 0xff000000
```

```
. . . .
en0: flags=8863<UP,. . . . > mtu 1500
. . . .
   inet 146.193.41.40 netmask 0xffffff00 broadcast 146.193.41.255
. . . .
   status: active
```

ipconfig

```
Windows IP Configuration
Ethernet adapter vEthernet (WSL):
IPv4 Address. . . . . . . . . . . : 172.26.0.1
Ethernet adapter Ethernet 4:
IPv4 Address. . . . . . . . . . . : 192.168.56.1
Wireless LAN adapter Wi-Fi:
IPv4 Address. . . . . . . . . . . : 194.210.228.171
```

For different computers, the number of available addresses may change but the relevant ones are identified as **inet** xxx.xxx.xxx.xxx (in linux/MAC os X /WSL) or **Ipv4 Adress** xxx.xxx.xxx.xxx in windows.

Not all addresses are available remotely because of network or firewall configurations.

2 Datagram Internet sockets (duplex-communication)

The **example** directory contain a three programs (one server and two clients). The server receives a message from clients and replies to it. One of the clients does not wait for a reply (**client-simplex.c**) and the other waits for a reply (**client-duplex.c**).

The exchanged messages are similar to the examples of Lab 4. The main different is the type of socket (AF_INET), the addresses and the scope of communication is different.

To use Internet domain sockets, it is necessary to include the following header files:

```
#include<netinet/in.h>
#include<arpa/inet.h>
#include<sys/socket.h>
```

The next table shows the main parts of the code with a simple explanation of the code:

Socket creation

Each program create his own socket. The domain and type should be corrected taking into consideration the scope and type of communication. For internet communication the socket type should be **AF_INET**

Server

```
int sock_fd;
sock_fd = socket(
     AF_INET, SOCK_DGRAM, 0);
if (sock_fd == -1){
     perror("socket: ");
     exit(-1);
}
```

Client

```
int sock_fd;
sock_fd = socket(
       AF_INET, SOCK_DGRAM, 0);
if (sock_fd == -1){
     perror("socket: ");
     exit(-1);
}
```

Socket identification

In Internet domain sockets only the server need to assign (bind) one address, the client after calling the **sendto** gets an fully working implicit address.

The server address will be composed of the port and an address. Since each computer can have multiple addresses, the programmer can define what address will be activated for this program. If this program can receive messages sent to any address he can use the **INADDR_ANY** constant. The port number should be between 4000 and 40000, and needs to be converted with the **htons** function.

Server

```
struct sockaddr_in local_addr;
local_addr.sin_family = AF_INET;
local_addr.sin_port = htons(SOCK_PORT);
local_addr.sin_addr.s_addr = INADDR_ANY;
err = bind(sock_fd,
              &local_addr, sizeof(local_addr));
```

Sending of messages

In datagram sockets, every time a program sends a message it is necessary to specify what is the address of the recipient. The **sendto** function receives the data (similarly to the write function), but has two extra arguments that refer to the address of the recipient. In Internet domain sockets the address is composed of the port number and the network address. The port must be converted with the **htons** function, and the network address that was read from the keyboard should also be converted to binary using the **inet_pton** function.

Client

```
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SOCK_PORT);
inet_pton(AF_INET, linha,
          &server_addr.sin_addr);
nbytes = sendto(sock_fd,
            message, strlen(message)+1, 0,
            &server_addr, sizeof(server_addr));
```

Reception of the message

Since the server wants to send a reply to the client, it needs to retrieve the client address This is accomplished with the **recvfrom** function. This function receives the message, but also the client address.

Server

```
nbytes = recvfrom(sock_fd, buffer, 100, 0,
            ( struct sockaddr *)&client_addr,
            &client_addr_size);
printf("received %d bytes:\n",nbytes);
```

Extraction of address

Although the address structure can be used in following **sendto**, it may be necessary to print the address and port. It is necessary to convert the port using the **ntoh** function. The address should also be converted into a string using the **inet_ntop** function.

Server

```
char remote_addr_str[100];
int remote_port;
remote_port = ntohs(client_addr.sin_port);
inet_ntop(AF_INET, &client_addr.sin_addr,
    remote_addr_str, 100);
printf("received message from %s %d:\n",
                remote_addr_str, remote_port);
```

Sending of reply

With the reception of a message from the client, the server also get the client address. The server can use such address to send a reply.

Server

```
nbytes = sendto(sock_fd,
```

```
    reply_message, strlen(reply_message)+1, 0,
      &client_addr, client_addr_size);
```

Reception of reply

The client can now receive the reply using the **recv** function. The client knows that the message comes from the server, because only the server received the implicitly assigned/bind address.

```
nbytes = recv(sock_fd, recv_message, 100, 0);
```

3 Exercise 1 (client and server on the same computer)

Observe the files in the **example** directory, compile the three program and execute various combinations of them in different windows, on the same computer:

- just client-simplex (without running server)
- multiple servers
- one serve and one client-simplex
- one serve and one client-duplex
- one server-simplex and multiple clients (simplex and duplex)

The clients read from the keyboard the address of the server. This is required even if the client and server are running in the same computer.

Using one of the command from Section 1.2, discover the various addresses of your computer and use them when running the clients.

After experimenting with the addresses provided by the ip/ifconfig/ipconfig commands,

experiment with other invented addresses.

After running multiple times the various options answer to the following questions

- what happens if the client-simplex sends a message without a receiving server?
- What happens if the client is killed (with Ctr-C) and restarted afterwards?
- What happens if multiple clients send data at the same time?
- What happens if tow servers try to run at the same time?
- What happens if a server runs after the other?

Compare Internet domain datagram sockets with UNIX datagram sockets.

4 Exercise 2 (client and server on different computers)

To experiment the provided code with multiple computers, student should run the server on the **sigma** computer and run the clients on their local computers. By using the Internet, the messages will go from the students computers (at home) to the server located at Técnico.

To transfer the files to **sigma** and compile and run the server, students should follow the instructions from Section 7 (Sigma computer).

The **sigma** computer has multiple addresses and not all of them are available from the internet, so try the client with all of them. Run the **ip a** command to know what are the configured addresses.

5 Exercise 3 - Remote control on the Internet

Modify the **exercise 4** from Lab 4 so that the server and clients run on different computers. The solution for this exercise is provided in directory **lab-4-exercise-4**.

To use the Internet for the clients and server to communicate, the only necessary

modifications are:

- create sockets with **AF_INET** domain;
- read the server address from the keyboard;
- use a suitable addresses structure (**struct sockaddr_in**) for the bind and **sendto**.

The client does not need to do the bind :), but needs to read the address of the server

from the keyboard.

Don’t forget to adapt the **machine-client.c** and the **human-client.c** to use Internet domain sockets.

6 Exercise 4 - Remote display client

Extend the solution of **exercise 3** with a new type of client that shows on the screen exactly the same information as the server. Multiple clients of this new type can run at the same time and all will replicate the screen of the server.

This client will contact the server and notify its existence, the server will store the address of this new client on a array, and every time the screen is updated, the server notifies this new client.

Whenever the server updates its screen (because a character moved) this updates should also be sent to all **remote-display-clients** stored in the array. These new clients should receive messages with the screen position to be updated and the corresponding character.

Follow these steps to implement this new functionality. The skeleton of this new client is provided in the **lab-4-exercise-4** directory as **remote-display-client.c**.

6.1 Step 1 (remote-char.h)

Modify the message structure that corresponds to include:

- two new message types (**remote-display-connect** and **draw**)
- the positions (x, y)

6.2 Step 2 (server.c)

Declare an array of addresses that will contain the address of each **remote-display-client** . Every time a new **remote-display-client** contacts the server, its address will be stored in this array.

6.3 Step 3 (remote-display-client.c)

Modify the supplied **remote-display-client.c** skeleton so that it send to the server a connect message on startup.

6.4 Step 4 (server.c)

After the **recvfrom**, implement the code that verifies if the message is of type **remote-display-connect** and stores the **remote-display-client** address in the array of addresses.

6.5 Step 5 (server.c)

In the server, implement a function that receives (among other things) a character and the windows coordinates (x, y), and sends this information to all the connected **remote-display-clients**. This function should send messages of type **draw**.

6.6 Step 6 (server.c)

Whenever there is a character movement, the server should send to all the remote-display-clients a message that deletes the old position and another message that places the character on a new position. The server should call the function developed in Step5 twice (one to delete the character and another to write it on the new position).

6.7 Step 7 (remote-display-client.c)

On the remote-display-client implement the code that receives a message from the server, verifies if it is of type draw and updates the screen.

6.8 Step 8

After implementing the previous steps and verify that it is working correctly with all the clients on the same computer as the server (using the 127.0.0.1 address), copy the server to sigma and experiment connect clients on different computers(yours and from a colleague of yours).

7 Sigma computer

Sigma is a server hosted at Técnico that can be remotely accessed by student. It runs Linux and is a shared memory multiprocessor computer, with most of the necessary tools (compilers, libraries, …)

Student can access this computer in two different ways to:

- remote terminal – to execute commands
- file transfer – to download and upload files

7.1 sigma remote terminal

To access a command line in sigma it is necessary to use the ssh protocol.

In linux or MAC os X the students only need to issue the follwoig command in a terminal:

```
ssh istXXXXX@sigma.ist.utl.pt
```

and replace **XXXXX** by the correct istID.

After connecting student should type the password. If correct, students can now start to use all the installed programs and compilers

In windows it is necessary to install a ssh client such as:

- putty - https://www.putty.org/

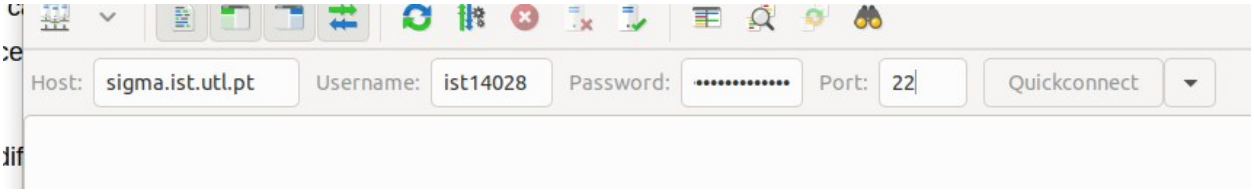
7.2 file transfer

To transfer files to/from the sigma it is necessary to use a **sftp** client.

The best **sftp** client is **filezilla** that is available for Windows, Linux and MAC OS X:

- https://filezilla-project.org/

to connect students can use the **quickconnect** as illustrated:



After connection students can transfer files by dragging files in the presented windows.