December 2020

Computer Communication Networks 2

Final Report - Lab 5

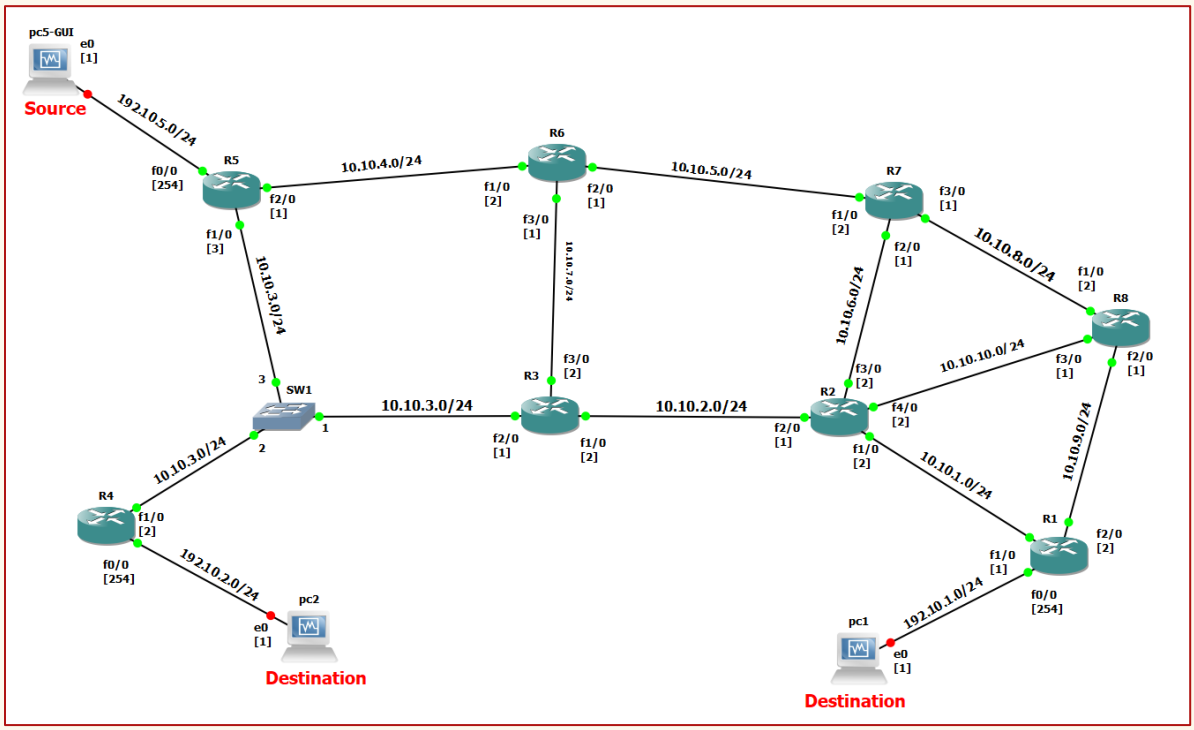
Socket programming

Ilan Klein, ID 317635258

Shir Granit, ID 205531445

Pair number: 10

1. TCP socket programming



1.13. "Open a socket and set it's properties":

```
/*SOCKET PROCESS*/

// open a TCP socket and set it's properties
myTCPsocket = socket(AF_INET , SOCK_STREAM , 0);
// sockaddr struct - set vars:
sock_struct.sin_family = AF_INET ;
sock_struct.sin_port = htons(port_num) ;
sock_struct.sin_addr.s_addr = inet_addr(str_ip_addr_sender);
```

- "socket()" – open a new socket.
- "sock_struct.sin_family" – Use IPv4 protocol.
- "sock_struct.sin_port" – Assign the port number to the socket.
- "sock_struct.sin_addr.s_addr" – Assign the server IP address.

1.14. 'listen()' function:

This function is in use when a server wishes to connect to clients. In order to establish this connection the server needs to commit to a port and listen to its traffic. In our case, we are listening to the welcome socket we created in the sender, and define the default value of multiple sockets- 128 (through "SOMAXCONN").

```
//wait for a connection - listen
l = listen(welcomeTCPsocket, SOMAXCONN);
//listening...
if(!l){ printf("Listening...\n"); }
else{ printf("ERR\n"); return -1; }
```

1.15. The sender IP address:

First notice we only used the IP address of the sender in the sender code only on the configuration of the 'sock_struct.sin_addr.s_addr'. We can use the address 0.0.0.0, instead of the input on the command-line, and thus address the socket to all outside potential clients.

1.16. The client side:

The client can accept more bytes then agreed on, it only closes the connection when "read_from_socket" reaches 0 – meaning the server has finished sending process (but haven't closed yet!):

```
//while: not all data received OR connection closed by sender
do
{
        //read from socket
        read_from_socket = recv ( myTCPsocket , buf , buffer_size , 0 );
        //read socket data into buffer & write to file / print to screen
        printf("%s", buf);
        //update actual_byte_num
        actual_byte_num = actual_byte_num + read_from_socket ;
}
while( read_from_socket > 0 );
```

1.17. TCP sender – “send():

Sending 2 num_parts:

71	107.058500	192.10.1.1	192.10.5.1	TCP	66 49208 → 5555 [FIN, ACK] Seq=1 Ack=10242 Win=52384 Len=0 TSval=60…
72	107.111204	192.10.5.1	192.10.1.1	TCP	66 5555 → 49208 [ACK] Seq=10242 Ack=2 Win=29056 Len=0 TSval=6001799…
123	321.815586	192.10.1.1	192.10.5.1	TCP	74 49209 → 5555 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TS…
124	321.879967	192.10.5.1	192.10.1.1	TCP	74 5555 → 49209 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SAC…
125	321.901251	192.10.1.1	192.10.5.1	TCP	66 49209 → 5555 [ACK] Seq=1 Ack=1 Win=29216 Len=0 TSval=6054077 TSe…
126	321.964794	192.10.5.1	192.10.1.1	TCP	1090 5555 → 49209 [PSH, ACK] Seq=1 Ack=1 Win=29056 Len=1024 TSval=605…
127	321.964794	192.10.5.1	192.10.1.1	TCP	1090 5555 → 49209 [FIN, PSH, ACK] Seq=1025 Ack=1 Win=29056 Len=1024 T…
128	321.986266	192.10.1.1	192.10.5.1	TCP	66 49209 → 5555 [ACK] Seq=1 Ack=1025 Win=32096 Len=0 TSval=6054098 …
129	321.997003	192.10.1.1	192.10.5.1	TCP	66 49209 → 5555 [ACK] Seq=1 Ack=2050 Win=35008 Len=0 TSval=6054101 …
130	322.018504	192.10.1.1	192.10.5.1	TCP	66 49209 → 5555 [FIN, ACK] Seq=1 Ack=2050 Win=35008 Len=0 TSval=605…
131	322.071894	192.10.5.1	192.10.1.1	TCP	66 5555 → 49209 [ACK] Seq=2050 Ack=2 Win=29056 Len=0 TSval=6055539 …

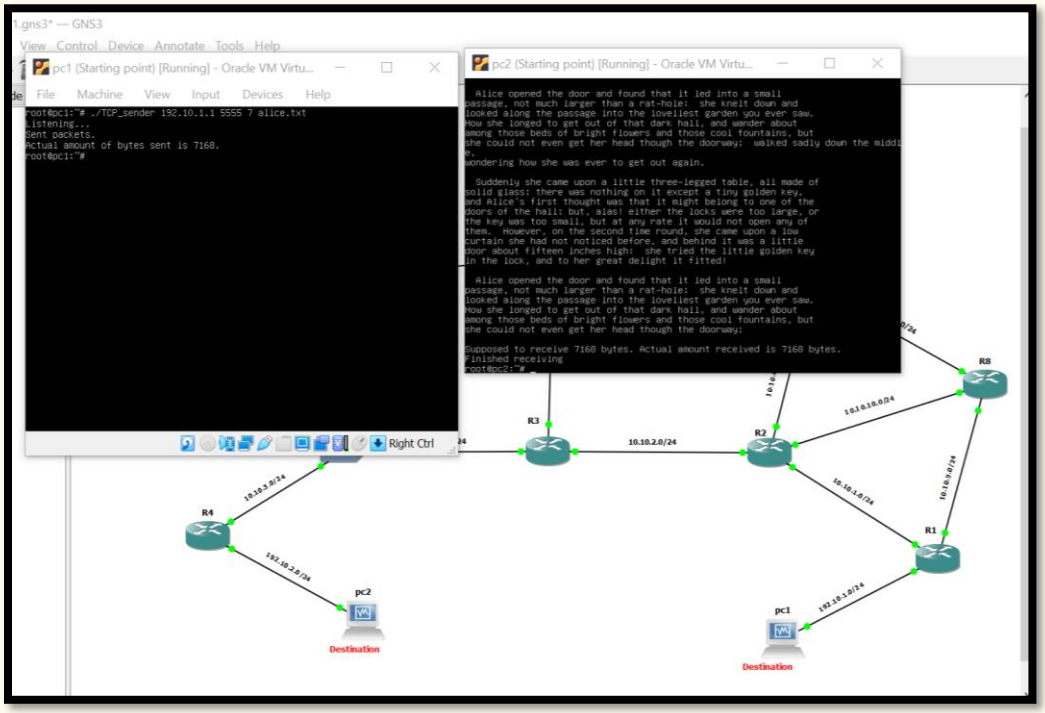
1.17.1. No.

1.17.2. Yes.

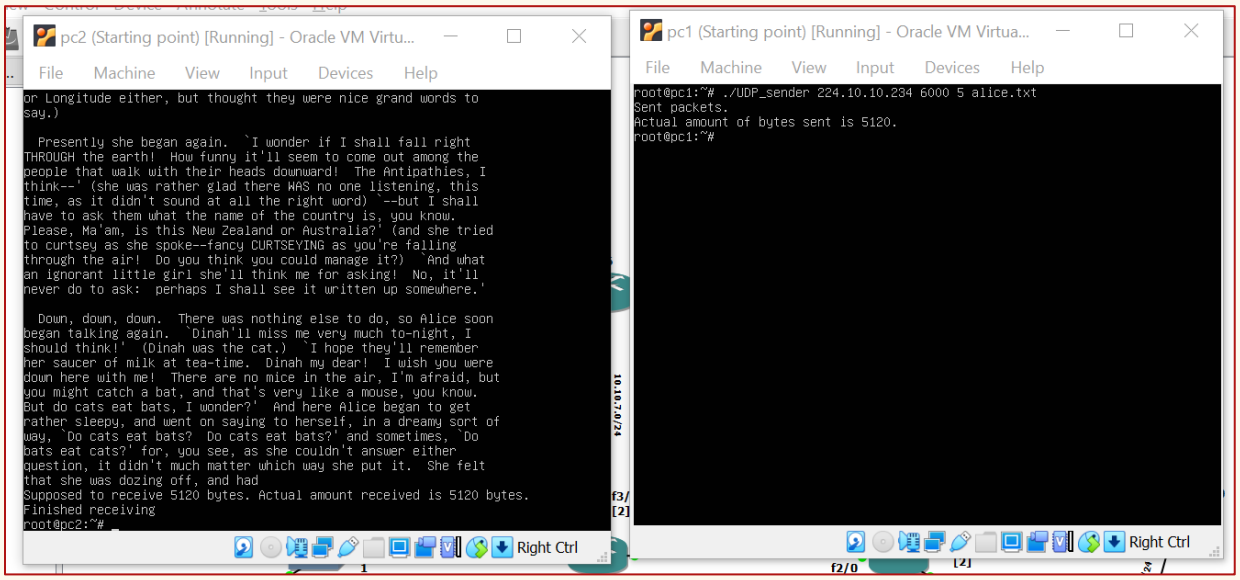
1.17.3. The receiver does not to send a message for each message he wants to receive – it only needs to establish his connection with the sender.

1.18. The sender:

The sender can know whether a connection has been lost, yet it cannot know if the data itself was received correctly – for that we have to define a feedback of some kind that the receiver return to the sender.



2. UDP socket programming



10…	21:38:39.292743	192.10.1.1	224.10.10.234	UDP	1066 34391 → 6000 Len=1024
10…	21:38:39.293720	192.10.1.1	224.10.10.234	UDP	1066 34391 → 6000 Len=1024
10…	21:38:39.293720	192.10.1.1	224.10.10.234	UDP	1066 34391 → 6000 Len=1024
10…	21:38:39.293720	192.10.1.1	224.10.10.234	UDP	1066 34391 → 6000 Len=1024
10…	21:38:39.293720	192.10.1.1	224.10.10.234	UDP	1066 34391 → 6000 Len=1024

2.13. "Open a UDP socket and set it's properties;"

This code segment is identical in both the UDP receiver and UDP sender:

```
//open a UDP socket
myUDPsocket = socket(AF_INET , SOCK_DGRAM , 0);

//set socket properties
sock_struct.sin_family = AF_INET;
sock_struct.sin_port = htons(port_num);
sock_struct.sin_addr.s_addr = inet_addr(str_mc_addr);
```

- "socket()" – open a new socket.
- "sock_struct.sin_family" – Use IPv4 protocol.
- "sock_struct.sin_port" – Assign the port number to the socket.
- "sock_struct.sin_addr.s_addr" – Assign the multicast address group as the destination address.

2.14. Differences between the TCP and UDP receiver and sender:

In TCP we build segments in the code that depend previous ones- establishing a connection between the sender and the receiver. In UDP we only connect the receiver to listen to a specified multicast group, and the sender to send the messages to that multicast group – we don't get a verification in the sender that the receiver got the messages.

2.15. Non-multicast different than the multicast receiver/sender:

In unicast, we send to a specific user, which we need to have its IP address. However, in a multicast group we can choose a group and tell the receivers to listen to that multicast group – no specific IP addresses are needed.

2.16. UDP sender:

The UDP sender can serve several receivers simultaneously, it is exactly the benefit of using multicast group and no a unicast IP address. We can activate more then one receiver, the only condition is that each one need to listen to the multicast address agreed on.

2.17. TTL value:

The default TTL value is 1:

> To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255 as the destination address in a sendto(3SOCKET) call.
>
> By default, IP multicast datagrams are sent with a time-to-live **(TTL) of 1,** which prevents them from being forwarded beyond a single subnetwork. The socket option IP_MULTICAST_TTL allows the TTL for subsequent multicast datagrams to be set to any value from 0 to 255, to control the scope of the multicasts
>
> From docs.oracle.com

We needed to change the default in out code of UDP_sender:

```
//multicast definitions
u_char ttl;
ttl = 32;
setsockopt( myUDPsocket , IPPROTO_IP , IP_MULTICAST_TTL ,&ttl , sizeof(ttl));
```

We chose first the value 16, thinking the max hop count in RIP is infinity – 16. Yet that didn't reach all edges in the network. We chose again the next value – 32, which works. We assume that 16 may be not enough for a message to travel the network if we want to ensure that it reaches all edges.

2.18. Alice, a CSE student, wrote a UDP app:

Alice probably didn't reset the values in the struct "sockaddr_in", which do net release at the end of the previous run:

```
//set socket properties
sock_struct.sin_family = AF_INET;
sock_struct.sin_port = htons(port_num);
sock_struct.sin_addr.s_addr = inet_addr(str_mc_addr);
```