



---

# Lab 8

## Computer Communication Networks

---

Socket programming



### The lab's schedule

Published date:	27/5/21
Quiz date:	30/05/21-03/06/21
Deadline for the final report:	15/06/21



# General Instructions

---

- ✚ The final report will be based on the answers of this document.
- ✚ Remember to pay attention to the "Final report submission" in the Introduction Lab.
- ✚ Most of the laboratory experiments based on the book: "*Mastering Networks: An Internet Lab Manual*"
- ✚ For each exercise, it is recommended to **read it all to understand the main idea before you do it.**
- ✚ IP address is composed of four octets ( "octet1.octet2.octet3.octet4" ).  
In our Labs all IP addresses will be according to the Network Figure, except *octet2*.  
***Octet2* will be according to the pair's number ( "10.X.0.1", *X = pair's number* ).**
- ✚ Write the number *X* clearly on the title page of your final report.

## Reading Material

---

- ✚ Read about Berkeley sockets:  
  
[https://en.wikipedia.org/wiki/Berkeley\\_sockets](https://en.wikipedia.org/wiki/Berkeley_sockets)
- ✚ Read the class presentation "Socket Programing.pdf" and go over the basic functions in the presentation (including their links) such as Bind, Setsockopt etc.
- ✚ More on the basics of sockets-
  - <http://www.dummies.com/programming/networking/cisco/network-basics-tcpudp-socket-and-port-overview/>

# Preliminary Questions

---

- ✚ What is a socket?
- ✚ Learn about and understand all the socket programming related functions found in the class presentation “Socket Programing.pdf”. (There is no need to remember specifics, just the basic idea) .
  - In particular:
    - socket()
    - bind()
    - accept()
    - listen()
    - close()
    - htonl() & ntohl()
- ✚ What is the right order of operations when opening a socket? Is there a difference when opening a socket for a TCP connection and a UDP connection?
- ✚ What are the **main** differences between the UDP and the TCP protocols? Which one is more reliable?
- ✚ What is a “welcome socket” (in relation to TCP sockets)?
- ✚ Can a UDP socket and a TCP socket exist on the same port?

# The Practical Section

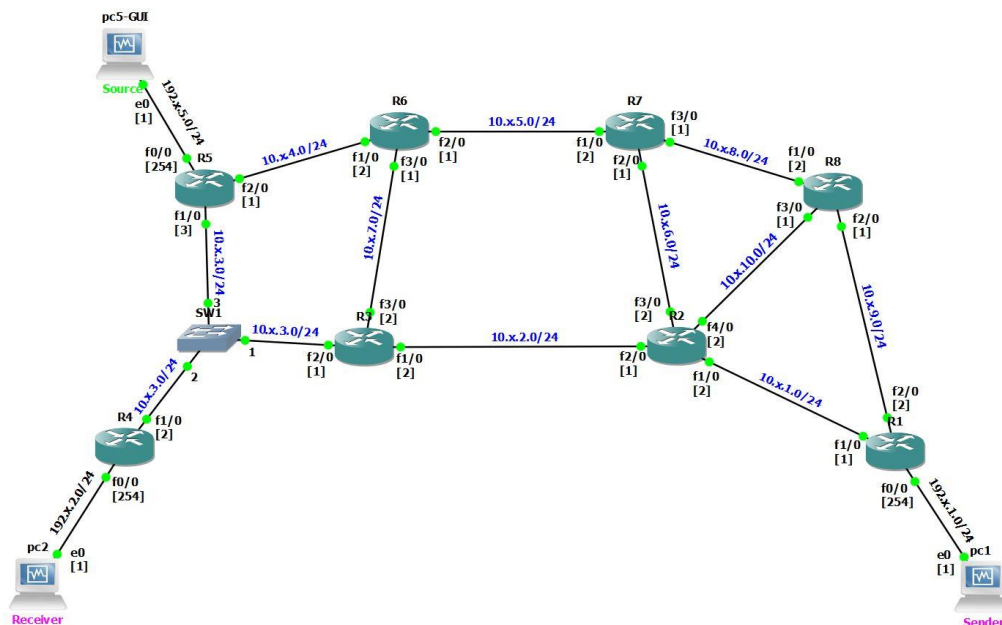
## Preliminaries Socket Programming :

- In this lab, we will learn the fundamentals of socket programming, to achieve this, we will prepare 5 different short programs, a TCP receiver, a TCP sender, a UDP sender, a UDP receiver capable of listening to a UDP stream as well as a UDP receiver which listens to several UDP sockets simultaneously use the `select()` function.
- For this lab, you may use 5 example programs, as well as the 2 chat client-server programs, these are meant only as a reference, and for you to test your code after each phase, they might not be flawless. You will find them on the course website. Instructions for their use are found at the *end of this document*. Your programs **must** operate using the same commands; take notice of the input parameter order and the program's name.
- **Test each of the socket-related function calls** (`socket`, `setsockopt`, `connect`, etc... ) for errors, in case of an error, you need to exit the program and output the failed function and reason for the error (you must use `perror()` and `errno`), remember to free any resources you used during the program before exiting (memory, **sockets**, etc).

The exercise is based on [Topology 8](#), [Topology 9](#), or [Topology 10](#).

**But first**, save it as a new project, before making any changes.

- Add and configure the PC according to the figure. Connect the GUI-PC5 to R5 (you will need to use it for the lab).




# 1. TCP socket programming

TCP sender and receiver.

In this part you will write two very basic apps that send a number of messages over a TCP connection.

## Pre

- 1.1. Your programs **must** run on PC1 to PC4. We encourage you to develop the software on PC5-GUI and test it against the executable files we provided, PC5-GUI has the same kernel as PCs 1-4.
- 1.2. You should use  to debug your programs. You may want to set the filter to "tcp".

## Do

*First part: TCP Receiver:*

- 1.3. Build a *TCP client* that connects to a server and receives several blocks of data (i.e.  $1024 * num\_parts$  bytes).
- 1.4. The receiver's input is:  $\{IP\_address, port, num\_parts\}$   
*IP\_address* : The server's IP.  
*port* : The TCP port.  
*num\_parts* : The number of parts of data that need to be received.  
Note that each part is 1024 bytes in size.  
\*you may assume that user input is correct.
- 1.5. **This program's executable must be called** "*TCP\_receiver*".

- 1.6. You must follow the following pseudo code.

```
TCP_receiver
Define: Buffer_size=1024
Input: IP_address, port, num_parts;
Open a socket and set it's properties;
Connect to server;
While (not all data received || connection closed by server)
    Wait for socket to have data
    Read socket data into buffer
    Write buffer to file/print buffer to screen.

Print number of bytes we needed to receive and actual amount received;
Free all resources;
```

*Second part: TCP Sender:*

- 1.7. Build a TCP sender that waits for a single receiver to connect, and then sends several parts of a file. Each part is 1024 bytes in size.
- 1.8. The sender's input is:  $\{IP\_address, port, num\_parts, file\_name\}$   
*IP\_address* : The server's IP (i.e. the sender's local IP address).  
*port* : The TCP port.  
*num\_parts* : The number of parts to send from the file.  
*file\_name* : The file to send.  
Note that each part is 1024 bytes in size.  
\*you may assume that user input is correct.
- 1.9. **This program's executable must be called** "*TCP\_sender*".

1.10. You must follow the following pseudo code.

```
TCP_sender
Define: Buffer_size=1024
Open a socket and set it's properties;
Wait for a connection:
While (not all data sent || any connection error)
    Read file into buffer
    Send buffer to client
Print number of bytes sent;
free all resources;
```

### *Third part: Test your programs*

- 1.11. Test your programs (Is the correct data being sent? Are the correct number of messages being sent? ETC), use Wireshark to see messages sent in the network.
- 1.12. After you are confident your programs are working properly, answer the following questions.

### Attach to your report

- 1.13. For the line of the pseudo code: “Open a socket and set it's properties;”; What are the function you used for this part of the program (related to socket programming)? For each of them give a general overview of their propose in your code. Answer for the receiver and sender separately.
- 1.14. In the *TCP sender* you probably used the “*listen()*” function, what is this function used for? What is it's input in your program? Would the input change for a more general program?
- 1.15. It is very likely that **no** packets sent by the *TCP sender* have been lost in your experiment but let us assume that a single data packet was actually lost in transmission. Are there any steps that your program needs to make to insure that **all** of the data is received eventually? (i.e. you need to retransmit the failed buffer etc.)
- 1.16. The *TCP sender* has it's own IP as input, however, this is not necessary. How would you change your program so that the *Sender* no longer requires it's own IP as part of it's input? Refer to the changed code and explain the purpose of each change.

- 1.17. Does the *TCP receiver* needs to know in advance how much data will be sent by the *Sender* in order to reach the last line in the code (free resources)? explain how does the receiver knows the connection had been closed? (*note: if your receiver does not close the connection itself when the sender closes its connection, then you must change the client's code so that it will close itself once the connection had been closed by the server.*).
- 1.18. If the *Sender* runs the “*send()*” function “*num\_parts*” times, does this mean that :
  - 1.18.1. A Wireshark output will show “*num\_parts*” data messages (not including control messages) being sent from receiver to transmitter?
  - 1.18.2. The *Receiver* necessarily uses the “*receive()*” function “*num\_parts*” times?
  - 1.18.3. If the number in either case is different than “*num\_parts*” explain why does this happen and how does the receiver overcomes this problem?
- 1.19. Does the *Sender* (the app itself) know it has *successfully* sent a message? How can it tell? Alternatively, if not, who is aware that a message had *successfully* been sent?

## 2.UDP socket programming

UDP sender and receiver. For this part you will write a UDP sender and receiver, the sender will transmit a stream of UDP messages while the receiver receives them.

### Pre

- 2.1. **Your programs must run on PC1 and PC2.** We encourage you to develop the software on PC5-GUI and test it against the executable files we provided.
- 2.2. You should use *Wireshark* to debug your programs. You may want to set the filter to “udp”.

### Do

*First part: UDP receiver:*

- 2.3. Write a program that opens a UDP socket. It prints the received data to screen until it receives the number of messages specified in the “*num\_parts*” input variable. Then it closes the connection.
- 2.4. The receiver's input is: {*ip\_address* , *port* , *num\_parts*}  
*ip\_address* : The IP.  
*port* : The UDP port.  
*num\_parts* : The number of parts of data that need to be received.  
Note that each part is 1024 bytes in size.  
 \* You may assume that user input is correct.
- 2.5. **This program's executable must be called “*UDP\_receiver*”.**



2.6. You must follow the following pseudo code.

```
UDP_receiver
Define: Buffer_size=1024
Input: ip_address ,port, num_parts.
Open a UDP socket and set it's properties;
While i < num_parts
    Wait for socket to have data
    Read socket data into buffer
    print buffer to screen.
    i++;
Print number of bytes we needed to receive and actual amount received;
free all resources;
```

*Second part: UDP\_sender:*

2.7. Write a program that opens a UDP socket. The program opens a file and sends a number of messages specified in the “*num\_parts*” input variable, to an IP address.

2.8. The sender’s input is: {*ip\_address* , *port* , *num\_parts* , *file\_name*}

*ip\_address* : The IP you will send messages to.

*port* : The UDP port.

*num\_parts* : The number of parts to send from the file.

*file\_name* : The file to send.

Note that each part is 1024 bytes in size.

\*you may assume that user input is correct.

2.9. **This program’s executable must be called** “*UDP\_sender*”.

2.10. You must follow the following pseudo code.

```
UDP_sender
Define: Buffer_size=1024
Input: ip_address, port, num_parts, file_name.
Open a UDP socket and set it's properties;
Set appropriate properties for a udp socket;
While i< num_parts:
    Read file into buffer;
    Transmit buffer data;
    i++;
free all resources.
```

### *Third part: Test your programs*

- 2.11. Test your programs (Is the correct data being sent? Are the correct number of messages being sent? ETC), use Wireshark to see messages sent in the network.
- 2.12. After you are confident your programs are working properly, answer the following questions.

### Attach to your report

- 2.13. For the line of the pseudo code: “Open a UDP socket and set it’s properties;”; What are the functions you used for this part of the program (related to socket programming)? For each of them give a general overview of their propose in your code. Answer for the receiver and sender separately.
- 2.14. What are the main differences between the TCP and UDP receiver and sender you can see in the code? Explain how do those differences relate to the protocols we are using?
- 2.15. Alice, a CSE student, wrote a UDP app, much like the ones described in this lab. It always works fine the first time she runs it, however if she tries to run it a second time (with the same port and address) she gets an error for the function **bind**: “Address already in use”. What is (probably) happening and why? Describe a simple programmatical solution to this problem, does this solution have any drawbacks?

### 3. Selecting Server

In this part you will practice with using the `select()` c function. `select()` allows us to handle several sockets at the same time. And this server will showcase this ability in the simplest manner.

#### Pre

- 3.1. **Your programs must run on PC1 to PC4.** We encourage you to develop the software on PC5-GUI and test it against the executable files we provided.
- 3.2. You should use *Wireshark* to debug your programs. You may want to set the filter to “UDP”.

#### Do

*select server:*

Write a program that receives several port number and opens a UDP socket for each one. The program will print Client with port i has sent a message when it receives data on socket i, and continue listening.

The server's input is: { port\_1 , port\_2,..., port\_n }

port\_i : a udp port which the server listen to. There are up n ports, you may assume that

$$1 \leq n \leq 10$$

\* You may assume that user input is correct.

You may use the previous "UDP\_sender" to send data to the select\_server. Use it to send one packet of information.

This program's executable must be called “Select\_server”.

You must follow the following pseudo code.

```
Select_server
Input: port_1 , port_2,..., port_n;
Open n udp sockets and set their properties;
While (1)
    Wait for user sockets to have data //use select()
    if(socket with port i has data)
        print (Client with port i has sent a message)
        reset the select function and sockets.
Free all resources;
```

### *Last part: Test your programs*

- 3.3. Run the server on one PC, configure three other PCs on different locations in the network to test your server. Run the UDP\_sender several times to see that the server indeed works (more than once...).
- 3.4. Test your program, use Wireshark to see messages sent in the network if needed.
- 3.5. After you are confident that your programs are working properly, answer the following questions.

## Attach to your report

- 3.6. What are the different input parameters of the select function? Explain each separately. Which did you actually have to use in the code?
- 3.7. Let us denote Select's return value as "p". p can be either p=0, p=-1 or p>1. For each case, explain the meaning of the return value.
- 3.8. Could `select()` detect that more than one socket has received data with one function call (i.e. there is more than one socket which has data on it)?
- 3.9. Is `select()` only useful for sockets? Give examples for other uses for `select()`.

## 4. Submission instruction

This time, in addition to the *lab report*, you need to submit your source code.

**Submit a single zip** file named according to the same convention you used for the lab reports: "`Lab_8_ID1_ID2.zip`". Inside submit the report itself (as a .pdf file named "`Lab_8_ID1_ID2.pdf`") and a folder containing four source code files with the same names as the programs i.e. `UDP_sender.c`, `TCP_sender.c` etc.

Add a makefile to your zip that will compile all programs. There's no need to add a compiled version.

## Appendix:

### How to use the example programs:

Transfer the programs and a file (you can use any large file, or just use alice.txt) to each of the lab's PC's using SCP or FTP (see the useful commands file for further instructions).

To run TCP receiver /sender:

On one of the lab's PCs first run the **TCP sender:**

`./TCP_sender <sender_address(host PC address)> <port> <number_of_file_parts> <file name>`

For example:

`./TCP_sender 192.168.1.1 5555 10 alice.txt`

The server should now wait for a connection. Now you can run the client on another PC.

To run the TCP receiver:

`./TCP_receiver <sender_address> <port> <number_of_file_parts>.`

To run UDP receiver /sender:

On one of the lab's PCs first run the **UDP receiver:**

`./UDP_receiver <ip_address> <port> <number_of_file_parts>.`

The client should now wait for a UDP stream. Now you can run the sender on another PC.

To run the UDP sender:

`./UDP_sender < ip_address > <port> <number_of_file_parts> <file name>.`

To run select server:

On one of the lab's PCs first run the **UDP receiver:**

`./select_server <port_1> <port_2>....<port_n>`

The server should now wait for a UDP packet. Now you can run the sender on *another* PC. Send one packet to one of the ports "i".

To run the UDP sender:

`./UDP_sender < ip_address > <port_i> 1 <file name>.`

Try and do this for all ports!



Good Luck!