# TCP/IP NETWORKING

# LAB EXERCISES (TP) 3
# SOCKET PROGRAMMING
# With Solutions

November 3rd, 2022
**Deadline:** November 16th, 2022 at 23.55 PM

**Abstract**

Lab 3 is both a programming lab and an analysis lab.

On one hand, you will program client/server applications, you will apprehend TCP/IP principles from the programmer's point of view with the notion of *sockets*. You will experiment different protocols and ways of communication (TCP, UDP, unicast, multicast). Furthermore, you will see how Transport Layer Security (TLS) is used to secure TCP connections.

On the other hand, you will analyze traffic traces with *Wireshark* to understand the mechanisms hidden from the programmer's perspective. It will highlight some fundamental mechanisms of UDP, TCP (packetisation, acknowledgment,...) and TLS (secure handshake, certificates, ...).

The grade for this lab depends both on your score in the Moodle quizzes and on your score in the script scoring system.

## ORGANIZATION OF THE LAB

### WHAT PARTS TO DO

If you have no prior experience with programming, you should consider doing

- part 1 and 2 the first week,
- parts 3, 4 and 5 the second week.
- You may safely skip part 6 (which is optional and for bonus).

If you have some experience with programming, you may

- skip part 1.1 (which is not graded),
- do parts 1.2, 2, 3, 4 and 5
- and do part 6 if time permits (it is optional and for bonus).

### WHAT TO DO IN EACH PART

In each part, you will be required:

- to **DESIGN** one (or several) scripts according to the requirements and to **TEST** it against servers that we provide,
- to **ANALYZE** the behavior of the application by means of prints, packets captures, bash commands, and to **ANSWER** to quizzes within Moodle,
- to **SUBMIT** your script(s) for scoring.

### ENVIRONMENT - DESIGN AND TEST

For this lab, you **must** create your scripts using **Python3**. This lab has been designed to be performed on the virtual machine.

**Prerequisites:** As opposed to the previous lab, please make sure that the network adapter of the virtual machine (in VirtualBox) is set to **NAT Network** (and **not** NAT).

For this lab, we need to install a few additional packages to the virtual machine (this is to be performed only once). Open a terminal in the VM and type:

```
sudo apt install manpages-dev python3-pip idle openssl
```

```
pip3 install websocket-client
```

**Useful links for socket programming with Python:**

- Python documentation on socket: http://docs.python.org/3.7/library/socket.html
- The Python socket documentation is not complete. It is often useful to look at the Unix documentation of the corresponding functions. In particular, the socket options are described by typing `man 7 socket`, `man 7 ip` or `man 7 ipv6` in a terminal. These pages are also available directly on the internet, for example http://linux.die.net/man/7/ipv6.

### MOODLE QUIZZES - ANALYZE AND ANSWER

The quizzes on Moodle open on **Thursday, 3 November 2022, 15:00** and close on **Wednesday, 16th November 2022, 23:55** Lausanne local time.

### PREPARING YOUR SCRIPTS FOR SCORING - SUBMIT

The script scoring system is running on https://icsil1-ds-105.epfl.ch. It is accessible **from within EPFL (EPFL wifi) or via the EPFL VPN only**.

**For MiniX users :** You can install the VPN by following the **command line** explained here : https://network.epfl.ch/xtrn/download/vpn/vpn_linux_en.pdf

To succeed the tests of the scoring system, it is essential that you respect the **prototype** of each of your script, *i.e.* the specifications of the command line used to launch your script. You have unlimited attempts[1] and

[1]There is a (high) attempt limit but more for security purposes. If you do happen to reach that limit, send an email to the TA team.

only the best score for each part will be kept. However, as the scoring may take some time, we recommend that you thoroughly test your script before submitting it to the system.

You can of course discuss with other students but plagiarism will not be tolerated. Your scripts will be processed by JPlag https://github.com/jplag/JPlag to detect similarities with other students' scripts. The TA team will always make the final decision after a manual inspection of the scripts with high similarity. Do not write any personal data in your Python scripts: Neither your name, nor your email address, nor your SCIPER number.

The system opens on **Thursday, 3 November 2022, 15:00** and closes on **Wednesday, 16th November 2022, 23:55** Lausanne local time. All attempts submitted before the deadline will be processed.

# 1 Socket Programming Basics

This part of the lab aims at introducing you to socket programming. In particular, you will be first asked to understand and execute specific examples of code. Part 1.1 is not graded, however it covers different topics that should give you the necessary background for the rest of this lab. If you believe that you are already familiar with the topics covered here feel free to skip it and proceed to part 1.2. If you have no background on Python programming and you feel that you need more assistance with the following examples please do not hesitate to ask for help from any of the TAs.

Scripts presented in this part are available for download on Moodle.

## 1.1 [Not graded] Socket Programming in Python Primer

In this part, we will work with CodeA.py and CodeB.py. This part can be performed either on the EPFL Wifi or on the EPFL VPN.

The following examples of code are inspired from the Python documentation (for Python 3.7, available at https://docs.python.org/3.7/library/socket.html). Implement and run the two following examples (source codes available on Moodle) of a client and a server to test if your python implementation is working. For those who are not sure which editor to use for Python programming we suggest IDLE that comes by default with the version of Python distributed by www.python.org.

Code A:

```
import socket

HOST = '' # Symbolic name meaning all available interfaces
PORT = 50007 # Arbitrary non-privileged port

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind((HOST, PORT))
while True:
    data, addr = s.recvfrom(1024)
    print('From: ', addr)
    print('Received: ', data.decode('utf-8'))
```

Code B:

```
import socket

HOST = 'localhost' # The remote host
PORT = 50007 # The same port as used by the server

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(b'Hello, Romeo! ', (HOST,PORT) )
s.close()
print('Message sent')
```

**Q1/** Examine the two codes...

...and answer **Question 1 in Lab3 Part 1.1** on Moodle.

***Solution.*** *Code A is the code for a Server*

**Q2/** Examine the two codes...

...and answer **Question 2 in Lab3 Part 1.1** on Moodle.

***Solution.*** *Code A uses a UDP socket.*

**Remark:** If you closely examine Code B, you will notice that the `while` loop is a loop that has two `print` lines that print the received data and the source address. However, these lines are executed only a finite number of times (only once). The reason for this behaviour is because the code uses a blocking socket, i.e., if no incoming data is available at the socket, the `recv` call blocks and waits for data to arrive. In blocking sockets, the `recv`, `send`, `connect` (TCP only) and `accept` (TCP only) socket API calls will block indefinitely until the requested action has been performed. This behaviour can be modified either by setting a certain flags to make the socket non-blocking or by setting a timeout value on blocking socket operations. **Note that, in this lab we expect you to use only blocking sockets.**

**Q3/** Answer **Question 3 in Lab3 Part 1.1** on Moodle.

***Solution.*** *The client prints 'Message sent'*

**Q4/** Answer **Question 4 in Lab3 Part 1.1** on Moodle.

***Solution.*** *The server prints*

*`From:('127.0.0.1',port) Received:  Hello, Romeo!`,*

*where `port` is a port number.*

Don't forget to submit and review your attempt on Moodle by using the *Submit all and finish* button.

## 1.2 [GRADED] YET ANOTHER SOCKET EXAMPLE IN PYTHON

In this part, we will work with CodeC.py and CodeD.py. This part can be performed either on the EPFL Wifi or on the EPFL VPN.

There might be situations when you force a TCP program to terminate and you may want to restart it immediately after that. When you close the program (i.e., close the socket), the TCP socket may not close immediately. Instead it may go to a state called `TIME_WAIT`. The kernel closes the socket only after the

socket stays in this state for a certain time called the `Linger Time`. If we restart the program, before the `Linger Time` of the previous session expires, you may get `Address already in use` error message because the address and port numbers are still in use by the socket that is in `TIME_WAIT` state. This mechanism ensures that two different sessions are not mixed up in the case that there are delayed packets belonging to the first session.

Usually, this protection mechanism is not necessary because severe packet delays are not very likely in common networks. If you want to avoid seeing the above mentioned error you can do it by setting the reuse address socket option:`s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1).`

Below we give you another example such that one of the applications echoes back what it has received from the other. The example (Code C) also shows how the `SO_REUSEADDR` socket option is used.

Code C:

```
import socket

HOST = 'localhost'
PORT = 5002

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind((HOST, PORT))
sock.listen(1)

while True:
    connection, addr = sock.accept()
    while True:
        data = connection.recv(16).decode()
        print("received:", data)
        if data:
            connection.sendall(data.encode())
        else:
            print("No more data from", addr)
            break
    connection.close()
```

Code D:

```
import socket

HOST = "localhost"
PORT = 5002

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((HOST,PORT))
```

```
message = "Oh Romeo, Romeo! wherefore art thou Romeo?"
print("sending:", message)
sock.sendall(message.encode())

received = 0
expected = len(message)

while received < expected:
    data = sock.recv(8).decode()
    received += len(data)
    print ('received:', data)

while True:
    pass
```

**Q5/** Examine the two codes...

...and answer **Question 1 in Lab3 Part 1.2** on Moodle.

***Solution.*** *Codes C and D use TCP sockets*

**Q6/** Examine the two codes...

...and answer **Question 2 in Lab3 Part 1.2** on Moodle.

***Solution.*** *To use an IPv6 UDP socket, the following line:*

`sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`

*must be replaced by:*

`sock = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)`

Launch Code C and run the following command on a separate terminal:

```
ss -4an | head -n 1 && ss -4an | grep ":5002"
```

**Q7/** Examine the output of the command...

...and answer **Question 3 in Lab3 Part 1.2** on Moodle.

***Solution.*** *The prompted table contains 1 row. The first row indicates that a TCP socket is waiting for connection request(s) on port 5002.*

Now launch Code D on a separate terminal and re-run the above command.

**Q8/** Examine the output of the command...

```
Line A :      tcp  <status>  0  0  127.0.0.1:<port> 127.0.0.1:5002
Line B :      tcp  <status>  0  0  127.0.0.1:5002   127.0.0.1:<port>
Line C :      tcp  <status>  0  1  127.0.0.1:5002   0.0.0.0:*
```

***Solution.*** *The prompted table contains 3 rows.*

*Line A gives the status of the client socket on port xxxx (any value except 5002 was accepted.). It indicates that the socket has an established connection with a remote end-point.*

*The remote end-point is running on the same computer, its socket's status is given by Line B.*

*The remaining line indicates that the server is still listening for other incoming connection requests.*

**Q9/** Answer **Question 5 in Lab3 Part 1.2** on Moodle.

***Solution.*** *Once you launch CodeD, in how many lines does the SERVER print the received message ? 3*

*What line of the SERVER code can you change to manage the number of lines on the server's output ? COPY and PASTE the unmodified corresponding line in the following field:*

`data=connection.recv(16).decode()`

**Q10/** Answer **Question 6 in Lab3 Part 1.2** on Moodle.

***Solution.*** *When you launch CodeD, the following line of CodeC:*

`print("No more data from", addr)`

*is not executed. Indeed, to access that part of the IF THEN ELSE structure, function* `recv()` *must return 0. All the following answers were also accepted on Moodle: return false, give false, give 0, return zero, give zero, give null, return null, return none, give none, return empty string, give empty string. (all case insensitive).*

*This happens only when the connection is orderly terminated OR the requested number of bytes to receive from the stream socket was 0.*

*In our situation,* `recv()` *blocks.*

**Q11/** Now press `Ctrl + c` to terminate Code D.

Answer **Question 7 in Lab3 Part .2** on Moodle.

***Solution.*** *When you terminate CodeD (`CTRL+C`), the line*

`print("No more data from", addr)`

*is executed. Indeed, CTRL+C raises a* `KeyboardInterrupt` *interruption in CodeD, making it terminates. As a consequence, the socket is closed and the connection is orderly closed.*

**Q12/** Answer **Question 8 in Lab3 Part 1.2** on Moodle.

***Solution.*** *The server sends and receives data to/from the client on the same socket it is listening on (true/false) ? False*

**N.B:** In this lab, messages exchanged using sockets are encoded as unicode objects. Therefore, when you want to send data, you must encode it first using `encode('utf-8')` or simply `encode()` (as can be seen in the example codes given above). Similarly, when you receive the data at the other end, you'll need to convert it back (decode it) using `decode('utf-8')` or simply `decode()`.

## 2 [GRADED] PACKETIZATION

This part of the lab can be performed either on the EPFL Wifi or on the EPFL VPN

In this part of the lab, you will write a TCP client. The aim of this part is to demonstrate the fact that TCP is a stream-oriented protocol. In other words, messages are written to a socket as a stream but packetization of this streamed data is an independent process. We will have a look at the difference between packets and messages.

**N.B:** In this part of the lab you are required to develop a solution that uses **IPv4 sockets**.

Let's assume we have a Phasor Measurement Unit (PMU) device as part of a Smart Grid infrastructure. The PMU runs a server application that waits for a command from a Phasor Data Concentrator (PDC). On receiving a command from the PDC, the PMU sends $n$ short messages with the text `This is PMU data` $i$, where $i$ is the message number such that $i = 0, 1, ..., n-1$. The server closes the connection after sending the $n$ messages. The PDC does not know the value of $n$ a priori. The command message from the PDC has the format `CMD_short:d`, where `d` is the time interval in seconds between two consecutive `send()` calls at the PMU. The commands should be given as a command line argument to the client.

For your convenience, we set up a TCP server at `tcpip.epfl.ch` that emulates the PMU application. The server port number is `5003`. Your job is to write the TCP client application of the PDC. The client application should display the received message on screen such that each of the $n$ messages is displayed on a separate line, i.e., line 0 should display `This is PMU data 0`, line 1 `This is PMU data 1`, ... and so on.

The prototype of your application `PDC.py` must be:

```
python3 PDC.py <server> <port> <command>
```

where:

- `<server>` is either the IPv4 address of the server or its domain name.
- `<port>` is the port number of the server
- `<command>` is the command to send to the server, *i.e.* either `CMD_short:0` or `CMD_short:1` or `CMD_floodme` (see later).

**Q13/** Use your application to send `CMD_short:d` with `d=0` and then `d=1` to the `PMU` server described above.

Answer **Question 1 in Lab3 Part 2** on Moodle.

***Solution.*** *How many messages do you receive when you send the command `CMD_short:0` ? 18*

*How many messages do you receive when you send the command `CMD_short:1` ? 18*

**Q14/** Start Wireshark and then run your client program with the two `d` values. Each time observe the captured traffic.

Answer **Question 2 in Lab3 Part 2** on Moodle.

***Solution.*** *In Wireshark, how many packets FROM the server with a PAYLOAD length STRICTLY positive do you observe:*

*With the command `CMD_short:1` ? 18*

*With the command `CMD_short:0` ? A smaller number, depending on the network's congestion, MTU, RTT, etc. On Moodle, any value between 1 (included) and 14 (included) was accepted.*

**Q15/** Answer **Question 3 in Lab3 Part 2** on Moodle.

***Solution.*** *The goal of this question is to explain the observations made in the two previous answers.*

*With `CMD_short:1`, you receive exactly 1 message per packet.*

*With `CMD_short:0`, you receive sometimes more than 1 message per packet.*

*If we take a look at the server code, we find the following line which is executed whatever the `CMD_short` command it receives:*

```
c.send(message.encode())
```

*where:*

*-`message` is one of the PMU message you identified in Question 1*

*-`c` is a socket provided by the `accept()` call when your PDC client connects to the PMU. See part 1.2 as well as the lecture.*

*The `send()` call made by the server's application puts message in a queue and asks the server OS to send the message, without specifying how. The TCP segment is created by the server OS when the server OS decides it, according to some internal algorithm.*

*Usually, the chosen algorithm is the Nagle's algorithm.*

**Q16/** Answer **Question 4 in Lab3 Part 2** on Moodle.

***Solution.*** *From a programmer's perspective, how can your application be sure it received all the messages from the PMU server ?*

*Your application should test the value returned by `recv()`. When this value has a `False` boolean representation, it means that the connection has been orderly closed by the remote end and there is no more data to be read.*

*From a network engineer's perspective, which of the following flags is set on the TCP header when the sender informs that it closes the connection and has nothing more to send ?*

*Answer: The flag `Fin`*

Now, modify your client application at the PDC such that it sends a different type of command with the format `CMD_floodme` to the TCP server (PMU). Again the command should be given as a command line argument to the client. On receiving this command, the server sends a large message to the client by invoking `send()` only once as opposed to the above scenario where `send()` was invoked $n$ times. The server closes the connection after sending the message to the client. Note that the exact size of the data the server sends is not known to the client a priori. Your client program should receive the whole message sent from the server and display it on screen.

**Q17/** Start Wireshark and then run your client program such that it sends the `CMD_floodme` to the server.

Answer **Question 5 in Lab3 Part 2** on Moodle.

***Solution.*** *In Wireshark, how may packets with a NON EMPTY payload have you seen coming from the PMU TCP server ?*

*This value depends on the network's congestion, MTU, etc. If you tested from the EPFL network, you should have around 5 to 10 packets. This can be higher on more congested networks (if you tested from your home router using the VPN).*

*On Moodle, any non-zero positive value was accepted.*

**Q18/** Answer **Question 6 in Lab3 Part 2** on Moodle.

***Solution.*** *How many times was the `recv()` call invoked at the client side ? It depends on the value you set for the buffer. On Moodle, any value was accepted.*

*Is the number of packets you see in Wireshark the same as the number of `recv()` invocations at your client*

*? No*

*What can you change IN YOUR CLIENT CODE to change the number of* `recv()` *invocations ? Answer with a short sentence:*

*You can change the buffer size you set on the* `recv()` *function. On Moodle, was accepted any answer containing the words (case-insensitive): 'buffer'+'size' or 'size'+'buffer' or 'in'+'recv' or 'recv'+'argument' or 'argument'+'recv'.*

**Q19/** Answer **Question 7 in Lab3 Part 2** on Moodle.

***Solution.*** *This question will summarize your findings of part 2. Complete the following text*

*TCP is a stream-oriented protocol.*

*From the programmer's perspective, TCP provides a service which is analogous to:*

*A garden hose without any hole: anything that enters the hose will exit it on the remote end in the same order, irrespective of whether the hose is transporting water, fertilizer, pesticides or soda.*

*Who decides how the data transported by TCP must be formatted and interpreted ? The application layer.*

*In which of the following applications is a stream-oriented transport protocol a good solution ?*

- *Downloading a large file using FTP (File Transfert Protocol)*
- *Loading a web page containing one unique large image using HTTP (HyperText Transfert Protocol)*

*On the contrary, using TCP for small homemade messages, spaced by 1 second, is not a good idea as it requires lot of decoding and doesn't use the mechanisms of TCP (packetization, congestion control, etc..)*

## 3 [GRADED] UDP PACKET TRANSMISSION

This part must be performed on the EPFL Wifi (not on the EPFL VPN). In Part 2, we have seen how a PDC can send specific commands to a PMU and receive replies from the PMU on a reliable (TCP) connection. In this section, we will see how a PDC informs a PMU to reset its clock using an unreliable (UDP) protocol.

The PDC sends a reset command `RESET:n`, where `n = 20` is the number of seconds by which the PMU has to advance its clock. On receiving this command, the PMU chooses a random value $X$ where $0 \leq X \leq n$ is the actual offset it uses to reset its clock. The PMU also informs the PDC about the exact offset value it uses.

Your task is to write a UDP client (PDC) that sends the above reset command to the PMU using UDP. The PMU runs at `tcpip.epfl.ch`, port `5004` but depending on the time of the day, it runs either using IPv4 or using IPv6. Worse, the PMU is not reliable: with some loss probability, it might ignore the reset requests. Indeed, its code contains the following lines:

```
while True:
  data,addr = recvfrom();
  if random.random() >= some_probability:
  s.sendto(b'OFFSET=X',addr);
```

Therefore, your PDC application must keep retransmitting the reset command until it receives an acknowledgement from the PMU. To do so, use a timeout value of 1 sec to wait for an acknowledgement from the PMU before retransmitting again.

Note: if you don't get a reply for a packet with a given IP version, you cannot know whether this is because the PMU ignored the packet or because the PMU is running on the other IP version. Therefore, you should use both IPv4 and IPv6 when sending the reset command.

The prototype of your application `PDC.py` must be:

```
python3 PDC.py <server> <port>
```

where:

- `<server>` is the domain name of the server (from which you have to retrieve both its IPv4 and Ipv6 addresses)
- `<port>` is the port number of the server

Create the PDC script as described above then test it a few times with the PMU running at `tcpip.epfl.ch`, port `5004`. Then submit your PDC script for scoring.

Now, run your client 60 times, count each time how many packets you need to send before receiving an acknowledgment and compute the average.

You may modify your script to add a for loop that takes care of these 60 tests. However, do not send this modified version to the scoring system. This experiment can take 15 minutes, so you can leave the client running and continue with the next part (doing more than 60 tests is fine).

**Q20/ Question 1 in Lab3 Part 3**

On average, how many packets do you need to send before receiving an acknowledgement ?

What is (approximately) the loss probability that you observe ?

Hint: for $0 \leq x < 1$, we have $\sum_{i=0}^{+\infty} ix^i = \frac{x}{(1-x)^2}$.

***Solution.*** *The value set on the server for the average number of requests before receiving an acknowlegment is 7.14. Any value between 4.64 and 24.5 was accepted on Moodle.*

*If $p$ is the loss probability, then the probability $P(\#(\texttt{req}) = i)$ of receiving the acknowlegment after $i$ requests sent is*

$$P(\#(\texttt{req}) = i) = p^{(i-1)}(1-p)$$

*($i-1$ requests losts and one request not lost).*

*Then the expected value of $\#(\texttt{req})$ is*

$$\begin{aligned} E(\#(\texttt{req})) &= \sum_{i=0}^{+\infty} i \cdot P(\#(\texttt{req}) = i) \\ &= \sum_{i=0}^{+\infty} i p^{(i-1)}(1-p) \\ &= \frac{1-p}{p} \sum_{i=0}^{+\infty} i p^i \\ &= \frac{1}{1-p} \end{aligned}$$

*We obtain $p = 1 - \frac{1}{\#(\texttt{req})}$. The value of the loss probability set on the server is 0.86. Any value between 0.76 and 0.96 was accepted.*

# 4 [GRADED] UDP MULTICAST

This part can be performed on the EPFL Wifi or the EPFL VPN.

The Swisscom Internet TV has a weekly cultural TV program on which they multicast their production to their subscribers on the internet on a given multicast address. This week, they are multicasting the play `Romeo and Juliet` to their subscribers on an IPv4 multicast group address `224.1.1.1` on port `10505`, using any source multicast, in the following format:

- The first 6 bytes represent an ID coded as a Python bytes object (e.g., `b'swcmTV'`).
- The next bytes form the message from the play.

The Swisscom multicast server is emulated by a server running on the virtual machine. Hence, you do not need to be connected to the internet. All what you need is to create/launch the provided virtual machine.

## 4.1 LISTENING TO SWISSCOM INTERNET TV PROGRAM

Your first task is to write a multicast receiver that joins the above multicast group, listens on port `10505`, and displays the exchanged multicast messages. Your receiver will run on the virtual machine.

The prototype of your application `receiver.py` must be:

```
python3 receiver.py <group> <port>
```

where:

- `<group>` is the multicast group address on which to listen
- `<port>` is the port number on which to listen

Your receiver application must print the received messages (one message per line).

You can check that the multicast server is running with the following command `ps aux | grep python`

Create the multicast receiver, test it with the group `224.1.1.1`, port `10505` and then answer the following question.

**Q21/** Answer **Question 1 in Lab3 Part 4** on Moodle.

***Solution.*** *Do you need to set any special socket option in your RECEIVER to be able to receive multicast messages from the group ? Yes*

*IP_ADD_MEMBERSHIP*

## 4.2 ACTIVE PARTICIPATION FROM SUBSCRIBERS

In addition to listening to the cultural program, Swisscom encourages its subscribers to actively participate in the program by sending their opinions about the program to the multicast group.

Your second task is to write a program that reads text from the keyboard and sends the text to the multicast group in the same format as the messages from Swisscom, i.e., the UDP packets should contain in the first 6 bytes a CAMIPRO number identifying the source (e.g., `b'123456'`), followed by the text you write in the standard input.

The prototype of your application `sender.py` must be:

```
python3 sender.py <group> <port> <sciper>
```

where:

- `<group>` is the multicast group address on which to send the message
- `<port>` is the port number on which to send
- `<sciper>` is your sciper number

Create the sender script. Test it by launching your multicast receiver in a terminal, and your multicast sender with your Camipro in another terminal. Use the group 224.1.1.1 port 10505. Use your sender to send some text to the multicast group. You should see your messages on your multicast receiver.

Then answer the following question.

**Q22/** Answer **Question 2 in Lab3 Part 4** on Moodle.

***Solution.*** *Do you need to set any special socket option in your SENDER to be able to send multicast messages to the group ? No, because the server is running on the same LAN (actually on the same machine). Indeed, if the server were on a different LAN, we need to set the option IP_MULTICAST_TTL in order to allow the packet to go through the router, because the TTL of multicast packets is set by default to 1.*

# 5 [GRADED] EXCHANGE OF INFORMATION VIA A TLS SERVER

This part can be performed with the EPFL Wifi or the EPFL VPN. For this part, we will need the compiled script `Part5_PDC.py` and the certificate `Part5_ca.crt`. Both files are on Moodle.

This part of the lab is an extension of Section 2. In 2, we saw how a PDC sends a set of commands to a PMU and how the PMU replies to such commands. The communication between the PDC and the PMU was over an insecure channel.

In this part of the lab, we ask you to secure the communication between the PDC (client) and the server (PMU) when the PDC sends the command `CMD_short:d` with $d = 0$. As seen in class, the TLS protocol adds security to TCP by enforcing confidentiality, entity authentication and message authentication.

In Part 2, we are grading you on your client (PDC) program. But in this Part 5, we provide an executable of the client program (on Moodle) and ask you to create the server (PMU) program. Hence, we will grade only the PMU program. The message exchange protocol itself should stay the same as in Section 2.

In order to secure the communication between your PMU and the PDC, you first need to generate a pair of public/private key for your PMU.

To generate a RSA public/private key pair of 2048 bits, use the following command:

```
# openssl genrsa -out <CAMIPRO>_key.pem 2048
```

Replace ¡CAMIPRO¿ by your sciper. The key pair is stored in the ¡CAMIPRO¿_key.pem file.

Now that you have generated a key pair for the PMU, you need to ask a trusted Certificate Authority to sign a certificate binding the ID of the PMU, with its public key. For this, you must generate a certificate request with the following command:

```
# openssl req -new -key <CAMIPRO>_key.pem -out <CAMIPRO>.csr
```

You should be asked to enter some contact information (you can be creative if you want). At the point where you are asked to "enter the following 'extra' attributes", please leave empty answers. This will create a `.csr` certificate request file.

Now that you have created the certificate request file, you need to ask a trusted Certificate Authority to sign your certificate.

We have created for you a Certificate Authority that is trusted by the PDC.

Upon submitting your certificate, the Certificate Authority will use the information that you provided during the creation of the certificate request to thoroughly check if you can be trusted. If so, it will deliver your signed certificate.

To obtain your signed certificate, go to https://icsil1-ds-105.epfl.ch and in "New submission", select "Part5 - TLS: Sign your certificate".

After sucessuful completion, use the link to download the full content of your submission. The signed certificate is in the "output" folder.

You now have three important files that will be required by the TLS/SSL protocol that you will use:

- your signed certificate `.crt`,
- your key pair file `CAMIPRO1_key.pem`,
- the CA certificate `Part5_ca.crt` available on Moodle.

Using the above files, you need to create a secure PMU application. It needs to run on localhost and use an IPv4 socket. When launched, the PDC will try to open a secure connection with your server on port 5003, requiring its authentication, and then it will send you the command `CMD_short:0`. On reception, your PMU server must answer with AT LEAST `This is PMU data 0`, followed by several other messages `This is PMU data i`, for a total number of messages of your choice.

The prototype of your application `secure_pmu.py` must be:

```
python3 secure_pmu.py <certificate> <key>
```

where:

- `<certificate>` is the path to your signed certificate
- `<key>` is the path to your secret key

This time, as opposed to previously, you need to hardcode the address "localhost" as well as the port number 5003.

The provided client program `Part5_PDC.pyc` (on Moodle) can be launched using:

```
python3 Part5_PDC.pyc CMD_short:0 <ca-certificate>
```

where:

- `<ca-certificate>` is the path to the CA certificate

After making sure your code works, open `Wireshark`, run your code (run the server before the client).

**Q23/** Answer **Question 1 in Lab3 Part 5** on Moodle.

***Solution.*** *There are two packets exchanged during the handshake phase with TLSv1.3.*

**Q24/** Answer **Question 2 in Lab3 Part 5** on Moodle. ***Solution.*** *In the handshake protocol of TLSv1.3, the first sent packet is a [Client Hello], which consists of the [client] initiating the communication with the [server], this packet contains information on the [supported cipher suites]. Then, the second packet is a [Server Hello] in which the chosen cipher suite is specified. Finally a [Change Cipher Spec] packet is sent by the client to confirm the chosen cipher suite, this packet also contains some [encrypted application data] and is not part of the [handshake protocol].*

**Q25/** Answer **Question 3 in Lab3 Part 5** on Moodle. ***Solution.*** *It is possible that the PMU agrees to send data to a fake PDC because the PDC is not authenticated to the PMU. In contrast, it is not possible that the PDC accepts data from a fake PMU because the PMU is authenticated to the PDC. In this scenario, we have server-side encryption, it is often like this on the web, clients are sure of the server's identity but the server communicates with any client.*

**Q26/** Answer **Question 4 in Lab3 Part 5** on Moodle. ***Solution.*** *For a two-way authentication, it does not matter if the two certificates are signed by the same Certificate Authority.*

# 6 [BONUS] WEBSOCKETS

This part can be performed on the EPFL Wifi or on the EPFL VPN.

Remember that in Part 2, you have designed a PDC that receives data from a PMU using an unsecure TCP connection. We have seen that TCP is a stream-oriented protocol. Messages sent over TCP may be grouped together within a same packet, and they are delivered to the application without any separator between them. We have seen that in this case, it is your responsibility as an application programmer to decode the data and understand how the messages are separated from each other.

In Part 3, you have implemented a datagram-oriented protocol based on UDP. We have seen that each UDP packet contains only one message, which reduces the effort required to decode the data. However you have seen that this effort is transferred into implementing your own retransmission mechanism.

To reduce the effort of data decoding left to the programmer while keeping a reliable TCP-based transport, protocols have been designed on top of TCP. HTTP (Hyper Text Transfer Protocol, RFCs 1945, 2068, 2616 and 7230-7237), FTP (File Transfer Protocol, RFC 3659) and SMTP (Simple Message Transfer Protocol, RFCs 821 and 5321) are famous examples. In this part, you will study WebSocket, one of these protocols, designed to transport messages with very low decoding left to the programmer.

## 6.1 BACKGROUND

The WebSocket protocol (https://tools.ietf.org/html/rfc6455) is a TCP-based application layer protocol that provides a persistent full-duplex TCP connection between a client and a server. Although it was originally designed to be implemented in web browsers and web servers, it can be used between any server and any client. The WebSocket protocol consists of an initial handshake phase followed by basic message framing mechanism on top of TCP.

The WebSocket handshake phase is based on HTTP and utilizes the `HTTP GET` method with an "Upgrade" request. The `HTTP GET` "Upgrade" request is sent by the client and then answered by the server with an HTTP 101 status code. Once the handshake is completed, the connection upgrades from HTTP to the WebSocket protocol. After the upgrade to the WebSocket protocol, both the client and server reuse the underlying TCP connection for sending WebSocket messages and control frames to each other. This connection is persistent and can be used for multiple message exchanges. A WebSocket defines message units to be used by applications for the exchange of data. A single message can optionally be split across several data frames. This allows sending messages where initial data is available but the complete length of the message is unknown.

The WebSocket resource URL uses its own custom prefix: "ws" for plain-text and "wss" for secure WebSocket connections. In this lab we will focus only on the plain-text WebSocket connections. But suffice to say that the secure WebSocket connection is established using TLS with TCP transport.

**Resources:**

- https://tools.ietf.org/html/rfc6455
- http://chimera.labs.oreilly.com/books/1230000000545/ch17.html

## 6.2 WEBSOCKETS IN ACTION

In this part of the lab, we are going to repeat the same set of experiments we did in Part 2, but this time using WebSockets instead of simple TCP sockets. We have already implemented the WebSocket server for you and it is waiting for incoming connections at `ws://tcpip.epfl.ch:5006`. Your task is to implement a WebSocket client that connects to the server.

For this, you can use the `websocket-client` module for Python3, that you can import with

```
import websocket
```

This module provides the low level APIs to implement your client.

Following the same approach as in the previous experiment, your WebSocket client will send commands to the WebSocket server and the server will reply different types and numbers of messages depending on the type of command it receives.

For the first part of the experiment, your client should send the `CMD_short:0` command to the server. On receiving this command, the WebSocket server replies with n consecutive short messages with the payload `This is PMU data i` to the server with 0 Seconds sleep time between each call to `send()` and then closes the connection. Your client must display each such message from the server in a separate line.

The prototype of your websocket-based `PDC.py` must be:

```
python3 PDC.py <server> <port> <command>
```

where:

- `<server>` is either the IP address of the server or its domain name
- `<port>` is the port to connect to
- `<command>` is the command to send

Use your application so that it sends the command `CMD_short:0` to the server.

**Q27/** Answer **Question 1 in Lab3 Part 6** on Moodle.

***Solution.*** *How many packets with a TCP payload STRICTLY positive do you receive from the server ?*

*Depends on the network congestion, etc. But at least 2 because the first answer from the server is the HTTP101 answer. On Moodle, any value greater or equal than 2 was accepted.*

*How many messages did you receive ? 10*

*How many time was the function `recv()` called ? (or the event `on_message` called) 10*

*Which of TCP or Websocket required for you the most complex handling of messages in order to print one message per line ? TCP*

Now, modify your client application such that it sends the `CMD_floodme` command to the server. On receiving the command, the server replies with a large text using a single `send()` invocation to your client and closes the connection. Your client does not know the exact size of the message from the server a priori. Your client should receive and display the whole message properly.

**Q28/** Answer **Question 2 in Lab3 Part 6** on Moodle.

***Solution.*** *How many packets with a TCP payload STRICTLY positive do you receive from the server ?*

*A given number, more than 2. Any value was accepted.*

*How many time was the function `recv()` called ? (or the event `on_message` called): Only once*

**Q29/** Answer **Question 3 in Lab3 Part 6** on Moodle.

***Solution.*** *Let's conclude !*

*TCP is a stream-oriented transport-layer protocol.*

*Websocket is a message-oriented application-layer protocol.*

*UDP is a datagram-oriented transport-layer protocol.*

**Q30/** Answer **Question 4 in Lab3 Part 6** on Moodle.

***Solution.*** *While the notions of datagram-oriented an message-oriented might be close, there are some subtle yet fundamental differences, let's review them:*

*After successful graduation with outstanding grades in the TCP/IP lecture, you quickly become chief system architect for a project within a big Internet operator.*

*The system you are in charge is based on the Internet protocol. For each of the following independent situations, we provide a requirement for that system. For each situation, choose the solution that best fits the requirement with the minimal effort.*

- *Situation A:*
  ***Requirement:*** *The system exchanges periodic messages of control data, of size 2500 bytes.*
  *Which solution do you choose ? A solution based on WebSockets.*
  *Indeed, Websocket allow the transport of messages of any size. Whereas with UDP we will have to cut the data into several datagrams and manage our-self the reordering of packets, the partial delivery, etc.*
- *Situation B:*
  ***Requirement****: In the system, power, energy and computational capabilities are extremely limited.*
  *Which solution do you choose ? A solution based on UDP.*
  *Indeed, UDP is lightweight, has a small header. With Websoket, you require all the TCP header + all the kernel algorithms to operate TCP (congestion control, packetization, etc.) + the Websocket protocol (with the HTTP "upgrade" handhsake, etc).*
- *Situation C:*
  *Sometimes, real-time systems are classified into two categories:* ***soft real-time*** *or* ***hard real-time****.*
  *In a* ***real-time system****, delayed information has still a positive added value, but this value is decreasing with time.*
  *Example: The news system. Imagine you learn that in 2018, a big company has bought the rights to create a TV series based on "The Foundation" books by I.Asimov. If your receive the information six months late, that piece of news has still a positive added value for you (you learned something). But if you receive the piece of news 10 years late (that is, after the TV series has been produced and after you watched it), then the information has lost most of its value, but in any case the value is never negative.*
  *In a* ***hard real-time system****, delayed information has null or negative added value.*
  *Example: A fire alarm. Imagine you hear a fire alarm delayed by one day. Then this information is of no use and can even have a negative effect, spreading panic among people.*
  ***Requirement****: The system must be* ***hard-real time****.*
  *Which solution do you choose ? A solution based on UDP.*
  *Indeed, if a delayed data has a negative value, then we don't want to retransmit lost data because the re-transmission incurs delays (not only for the lost data but also for the followings, see the lecture about the "Head-of-line Blocking" issue).*
- *Situation D:*
  ***Requirement****: The system is to be operated on the public Internet. Congestion and middle-boxes must be expected*
  *Which solution do you choose ? A solution based on WebSockets.*
  *Indeed, if congestion is to be expected, we may want to use a congestion-control mechanism. Besides, TCP is more likely to be accepted by middle-boxes (for security reasons), especially in a situation with congestion.*