TCP/IP NETWORKING

LAB EXERCISES (TP) 3
SOCKET PROGRAMMING

November 4th, 2021
Deadline: November 17th, 2021 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, ...).

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

You have free choice of using your preferred operating system and programming language. However, we highly recommend you to use **Python3** as it will be the only language supported by the TA team during the labs. Python3 is available for most operating systems (see `https://www.python.org/downloads/`). If your choice is different, we will be there to answer all your theoretical questions. However, depending on the specific questions you might have about other environments, the support you might get might be limited.

Furthermore, the scoring system only supports Python3 scripts as well. If your choice is different, the scoring of your scripts will be performed manually by the TA team and you won't get the chance for several attempts.

If you do not want to use **Python3**, you are **required to send an e-mail with your choice to `marc.egli@epfl.ch` before the end of the first week of the lab.**

Except specifically asked, the use of a virtual machine is not required.

MOODLE QUIZZES - ANALYZE AND ANSWER

Your analysis skills will be tested on **quizzes available on Moodle**. 7 quizzes are available on Moodle. The first one is relative to section 1.1 and is not graded. The last one is relative to section 6 and is graded only for bonus. Thus, only 5 quizzes are mandatory and graded.

You have unlimited attempts and only the best score among all your attempts will be kept for each quiz. More information will follow for each part.

The quizzes open on **Thursday, 4th November 2021, 23:55** and close on **Wednesday, 17th November 2021, 23:55** Lausanne local time. Attempts need to be **submitted** before the deadline to be graded. Don't forget to use the *Submit all and finish* button on Moodle.

Useful links

- 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`.

PREPARING YOUR SCRIPTS FOR SCORING - SUBMIT

The 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 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.

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, 4th November 2021, 23:55** and closes on **Wednesday, 17th November 2021, 23:55** Lausanne local time. All attempts submitted before the deadline will be processed.

[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.

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

DESIGN and TEST: For this part, you don't even need to be connected to the Internet. You will need it only at the end to answer Moodle questions.

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 = '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')
```

Code B:

```
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'))
```

ANALYZE and ANSWER: For this part, answer the quiz **[Not graded] Lab3 - Part 1.1** on Moodle.

Q1/ Examine the two codes...

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

Q2/ Examine the two codes...

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

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.

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

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

DESIGN and TEST: For this part, you don't even need to be connected to the Internet. You will need it only to answer Moodle questions.

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
```

ANALYZE and ANSWER: For this part, answer the quiz **[Graded] Lab - Part 1.2** on Moodle.

Q5/ Examine the two codes...
...and answer **Question 1 in Lab3 Part 1.2** on Moodle.

Q6/ Examine the two codes...
...and answer **Question 2 in Lab3 Part 1.2** on Moodle.

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

- on Linux:

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

- on Windows:
  - If you are using `Powershell` (preferred):

```
netstat -an | Select -First 4 ; netstat -an | findstr ":5002"
```

  - If you are using `cmd` :

```
netstat -an | findstr ":5002"
```

- on MacOS:

```
netstat -an | head -n 2 && netstat -an | grep "5002"
```

Q7/ Examine the output of the command...
...and answer **Question 3 in Lab3 Part 1.2** on Moodle.

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

Q8/ Examine the output of the command...
...and answer **Question 4 in Lab3 Part 1.2** on Moodle.

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

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

Q11/ Answer **Question 7 in Lab3 Part 1.2** on Moodle.

Q12/ Now press `Ctrl + c` to terminate Code D.
Answer **Question 8 in Lab3 Part .2** on Moodle.

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

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

DESIGN and TEST: For this part, you need to be connected to the Internet by any mean of your choice. For example, you can be connected to the `epfl` wifi or to the `epfl VPN`. A server is running for you at `tcpip.epfl.ch`, port `5003`.

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).

ANALYZE and ANSWER: For this part, answer the quiz **[Graded] Lab3 - Part 2** on Moodle.

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.

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.

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

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

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.

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

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

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

SUBMIT: To submit your script, be sure you are connected to the EPFL network (either on campus or remotely through VPN). Submit your PDC script for scoring: go to https://icsil1-ds-105.epfl.ch and in "New submission", select "Part2 - Packetization".
Please note that the grading might take up to 5 min.
For all your code log, use this command

```
print(message, flush=True)
```

3 [GRADED] UDP PACKET TRANSMISSION

DESIGN and TEST: For this part, you don't even need to be connected to the Internet. You will need it only at the end to answer Moodle questions and for script submission. You will use the provided virtual machine on Moodle.

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 UDP packets containing the reset command stated above. The client must keep retransmitting the message until it receives an acknowledgement from the PMU. The acknowledgement from the PMU has the format `OFFSET=`X where X is the offset it chose to reset its clock. Use a timeout value of 1 sec to wait for an acknowledgement from the PMU before retransmitting again.

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

Using your client, send the reset commands to a UDP server (PMU) running at `localhost` on port `5004` (as explained below). This machine is dual-stack and, depending on the time of day, the server runs either in IPv4 or IPv6. Your client must be able to send the command to the server on **both IPv4 and IPv6 sockets** and should detect automatically if the server runs on IPv4 or on IPv6 (the scoring system we will test both with a server on IPv4 and a server on IPv6).

The code of the PMU contains the following lines:

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

As you can observe, the PMU drops some packets on purpose with a certain probability.

Thus, you should send out packets both in IPv4 and IPv6, since you do not know whether the packets are lost due to the server dropping them, or due to the fact that you used the wrong protocol.

Test your script several times.

In order to avoid IPv6 tunnel interoperability issues, you will launch the server on `localhost` (the virtual machine). It is listening on port `5004`. To this end, we provide you with the compiled python files:

- `run_dual_server.pyc`
- `p3server_v4.pyc`
- `p3server_v6.pyc`

Be sure to put all these files within the same directory. The server can be launched using:

```
python3 run_dual_server.pyc
```

In order to resolve the `localhost` in both IPv4 and IPv6 addresses, check the file `/etc/hosts` and verify that the following lines are added:

```
127.0.0.1 localhost
::1 localhost loopback
```

If not, you need to add the missing lines. As you can notice, `/etc/hosts` is the local DNS.

SUBMIT: Submit your script for scoring: go to https://icsil1-ds-105.epfl.ch and in "New submission", select "Part3 - Dual UDP ipv6/ipv4".
Please note that the grading might take up to 5 min.

ANALYZE and ANSWER: For this part, answer the quiz **[Graded] Lab3 - Part 3** on Moodle.

Run the client 60 times and count how many packets you need to send before receiving an acknowledgement.

You may modify your above script in order to add a `for` loop taking care of those 60 tests. However, **do not send** that modified version to the scoring system.

This experiment can take more than 5 minutes. So you can leave the client running and continue with the next part.

Q20/ Answer **Question 1 in Lab3 Part 3** on Moodle.

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

4 [GRADED] UDP MULTICAST

DESIGN and TEST: For this part, you don't even need to be connected to the Internet. You will need it only at the end to answer Moodle questions and for script submission. You will use the provided virtual machine on Moodle.

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`

ANALYZE and ANSWER: For this part, answer the quiz **[Graded] Lab3 - Part 4** on Moodle.

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

Start Wireshark and launch your receiver.

SUBMIT: To submit your script, be sure you are connected to the EPFL network (either on campus or remotely through VPN). Submit your receiver script for scoring: go to `https://icsil1-ds-105.epfl.ch` and in "New submission", select "Part4 - Multicast: Receiving multicast messages".
Please note that the grading might take up to 5 min.
For all your code log, use this command

```
print(message, flush=True)
```

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

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

In order to test you multicast sender, launch the multicast receiver you wrote in (4.1) and then launch your multicast sender with your sciper number and send a text of your choice. Your multicast receiver should receive the text, in addition to the text it receives from the Swisscom Internet TV.

SUBMIT: To submit your script, be sure you are connected to the EPFL network (either on campus or remotely through VPN). Submit your sender script for scoring: go to `https://icsil1-ds-105.epfl.ch` and in "New submission", select "Part4 - Multicast: Sending multicast messages".
Please note that the grading might take up to 5 min.
For all your code log, use this command

```
print(message, flush=True)
```

5 [GRADED] EXCHANGE OF INFORMATION VIA A TLS SERVER

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.

5.1 TLS LAB EXERCISE

DESIGN and TEST: For this part, you don't need to be connected to the Internet. You will need it only at the end to answer Moodle questions and for script submission. You need the `openssl` software. We will provide command-line instructions for Ubuntu operating systems. We will provide as well a compiled version of the PDC to use in this part. In order to avoid compatibility issues, you are highly recommended to use the virtual machine.

As mentioned above, 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 $d = 0$. In Section 2, we are grading you on your client (PDC) program but in this section, 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, we first need to generate the PMU's secret/public key pair. This can be done using an openssl command under Linux, if you have another OS you can simply use the same VM as in Lab 1. Note that openssl is not installed by default. In order to install it, make sure the Network access mode of your network settings is set to NAT and type the following command in a terminal:

```
# sudo apt-get install openssl
```

Once you have an openssl-enabled environment you can generate an RSA secret/public key pair of 2048 bits with the following command:

```
# openssl genrsa -out CAMIPRO1_key.pem 2048
```

The key pair is stored in the CAMIPRO1_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 CAMIPRO1_key.pem -out CAMIPRO1.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.

SUBMIT: We have created a Certificate Authority for this lab that will sign your 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. . To access https://icsil1-ds-105.epfl.ch, be sure you are connected to the EPFL network (either on campus or remotely through VPN).
Please note that the grading might take up to 5 min.

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).

ANALYZE and ANSWER: For this part, answer the quiz **[Graded] Lab3 - Part 5** on Moodle.

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

Q24/ Answer **Question 2 in Lab3 Part 5** on Moodle.

Q25/ Answer **Question 3 in Lab3 Part 5** on Moodle.

Q26/ Answer **Question 4 in Lab3 Part 5** on Moodle.

SUBMIT: To submit your script, be sure you are connected to the EPFL network (either on campus or remotely through VPN). Submit your PMU script for scoring, together with your certificate and your private key: go to `https://icsil1-ds-105.epfl.ch` and in "New submission", select "Part5 - TLS : Secure PMU".
Please note that the grading might take up to 5 min.
For all your code log, use this command

```
print(message, flush=True)
```

6 [BONUS] WEBSOCKETS

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 to decode the data and understand how they 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 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 (RFCs 1945, 2068, 2616 and 7230-7237), FTP (RFC 3659) and SMTP (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

DESIGN and TEST: For this part, you need to be connected to the Internet by any mean of your choice. For example, you can be connected to the `epfl` wifi or via the `epfl VPN`. A server is running for you at `tcpip.epfl.ch`, port `5006`.

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. A WebSocket client can be implemented using different

languages. However, to be consistent with the rest of the lab, we encourage you to use python3 to implement your client. For this, you will have to install the `websocket-client`[2] module for python3. 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 `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 web-socket 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

ANALYZE and ANSWER: For this part, answer the quiz **[Bonus] Lab3 - Part 6** on Moodle.

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

Q27/ Answer **Question 1 in Lab3 Bonus** on Moodle.

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 Bonus** on Moodle.

Q29/ Answer **Question 3 in Lab3 Bonus** on Moodle.

Q30/ Answer **Question 4 in Lab3 Bonus** on Moodle.

[2] using `sudo pip3 install websocket-client` for a linux-based OS with pip installed

SUBMIT: To submit your script, be sure you are connected to the EPFL network (either on campus or remotely through VPN). Submit your websocket-based PDC script for scoring: go to https://icsil1-ds-105.epfl.ch and in "New submission", select "Part6(Bonus) - Websocket".
Please note that the grading might take up to 5 min.
For all your code log, use this command

```
print(message, flush=True)
```