Name 1: Tristan Overney
Name 2: Yoann Ponti

TCP/IP NETWORKING

LAB EXERCISES (TP) 3
SOCKET PROGRAMMING

Abstract

In this TP you will familiarize yourself with socket programming. You will have a chance to do experiments with different protocols (TCP, UDP) and different ways of communication (unicast, multicast). Furthermore, you will also learn about WebSockets and develop simple WebSockets application to see how WebSockets differ from low level TCP sockets.

ORGANIZATION OF THE TP

WHAT TO DO

If you have no prior experience with socket 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).

Please also note, that in order to complete parts 3 and 4 successfully, you need to be connected to the access point provided in INF019 (`lca2-tcpip-labs` wifi). For that reason, we will provide more access points in INF1 and INF2 for you during lab sessions on **Fridays** (that you can work in INF1 and INF2 freely). However if you would like to work on these parts in another day you have to be directly in INF019 room.

ENVIRONMENT

This is a programming lab. You have free choice of using your preferred operating system and programming language. Naturally, we are not able to support all the choices you might make. We will be supporting Python 3. 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, some parts of the lab are graded automatically and our test bench is also developed under Python. Hence, if you are not sure what to choose, we strongly suggest you use the same environment as we do. This will minimize the chances of problems due to incompatibilities between different environments. Please also note, that you **should not** set up a virtual machine for parts 1,2,3,4,5 of this lab. The VM is needed **only** for bonus part 6. **If you do not want to use Python, you are required to send an e-mail with your choice to `roman.rudnik@epfl.ch` before the end of the first week of the lab.**

TP REPORT

This document will be your TP report (one per group). Type the answers directly in the PDF document. Use Adobe Reader XI, as it supports saving forms. Do not forget to write your names on the first page of the report. You will also have to submit some source codes. Naming convention and other details will be described in the respective sections. Put all source codes and the TP report in one folder and upload .zip file of this folder on Moodle.

The deadline is Wednesday, November 9 at 23:59.

PART 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 socket 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.

1.1 –SOCKET PROGRAMMING IN PYTHON PRIMER

(Not Graded)

The following examples of code are inspired from the Python documentation (Section 18.1 for Python 3.5, available at `https://docs.python.org/3.5/library/socket.html`). Implement and run the two following examples 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 = 5001               #port number of communicating partner

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 = 5001       # 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('Received data: ', data.decode())
    print('From Address: ', addr)
```

Q1/ Examine the two codes

[A1.a] Is code A the code of a client or of a server? Client, indeed it is sending information to localhost.

[A1.b] Does it use a UDP or a TCP socket? UDP, by default a socket is a UDP socket.

Q2/ Launch the server first and then the client.

[A2.a] What does the client print? Message sent

[A2.b] What does the server print? Received data: Hello, RomeoFrom Address: ('127.0.0.1', 64027)

Remark: If you closely examine Code B, you will notice that the `while` loop is an infinite 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.**

1.2 – YET ANOTHER SOCKET EXAMPLE IN PYTHON

(Graded)

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

received = 0
expected = len(message)

while received < expected:
    data = sock.recv(16).decode()
    received += len(data)
    print ('received:', data)
while True:
    pass
```

Q3/ Examine the two codes. Do the codes use UDP or TCP sockets? IPv4 or IPv6 sockets?

[A3] It is a TCP IPv4 socket. Because it uses AF_INET SOCK_STREAM parameters.

Launch Code C and run the command `netstat -an | grep "5002"` (For windows users: netstat -an | findstr "5002") on a separate terminal.

Q4/ Explain the output of the command.

[A4] Tere is one line that is outputed. It indicates that there is one tcp ipv4 socket listening on 127.0.0.1:5002.

tcp4 0 0 127.0.0.1.5002 *.* LISTEN

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

Q5/ Is the output difference from the previous output. Explain what each line in the output represents.

[A5] Yes, it is. Now three lines are outputed. There is sitll the same one as before but two more are present. They represent the server and client socket which were open after the client was launched and the client established a connection to the server to send its message.

Q6/ Once you launch Code D, in how many lines does the server print the received message? How can you manage the number of lines in output?

Server, in 3 lines, The number of line can be "managed" using the buffer size (16 bytes for the server in the scripts).

[A6]

Q7/ Does the line `print("No more data from", addr)` at Code C execute when you launch Code D? If not, why not?

No, it does not. The connection is not close by the client thus the socket blocks and wait for new data. Altought the line is printed when we stop (kill) the code D. Indeed the connection is closed by D and thus the line 'print("No more data from", addr)' is extecuted.

[A7]

Q8/ Does the server perform receive and send on the socket it is listening on? Explain.

Yes, it can be seen in the server code. Indeed when the server receive data (sock.recv(16)) it print it and then send it back to the client where it gets printed.

[A8]

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()`.

Useful links

- Python documentation on socket: `http://docs.python.org/3.5/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`.

PART 2 – PACKETIZATION

(Graded)

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.

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 $i = 1, 2, ..., n$. The server closes the connection after sending the n messages. The PDC does not know the value of n a priori. The command signal 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 sends two commands with $d = 0$ and with $d = 1$. For both cases, 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 1 should display `This is PMU data 1`, line 2 `This is PMU data 2`, ... and so on. The client should display all the messages sent by the server for both cases.

Q9/ How many messages have you received coming from PMU?

20 messages numbered 0 to 19 per command.

[A9]

Q10/ Start Wireshark and then run your client program with the two d values. Each time observe the captured traffic. How many packets with payload length more than 0 have you seen coming from the TCP server (PMU):

1. when d=0?
2. when d=1?

1. 2 packets
2. 20 packets

[A10]

Q11/ If your answers for the above question are different for the two d values, explain what caused the difference?

when d=1, we receive each PMU data message in it's own separate packet
when d=0, we only receive PMU data message 0 separately then all other PMU data message are recieved together in a single packet. By searching on the web, this might come from the Nagle algorithm which is a means of improving the efficiency of TCP/IP networks by reducing the number of packets that need to be sent over the network

[A11]

Q12/ How can your client be sure that it has received all the messages from the server?

In the TCP header of the packet, the last packet of the serie sent by the server has the flags bit for "FIN" set to 1 which indicates it's the last one for the sequence. Also if calling recv() on the socket return 0 bytes. It means that the connection has been terminated following an error or that the server has finished sending all its data.

[A12]

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.

Q13/ Start Wireshark and then run your client program such that it sends the `CMD_floodme` to the server. How many packets with payload length more than 0 have you seen coming from the TCP server (PMU)?

Only one!

[A13]

Q14/ Copy several first lines of your output and paste them here:

ACT I

SCENE I. Elsinore. A platform before the castle.

[A14] FRANCISCO at his post. Enter to him BERNARDO

Q15/

1. How many times was the `recv()` invoked at the client?
2. Is the number of packets you see in Wireshark the same as the number of `recv()` invocations at your client?
3. What are the factors that affect the number of `recv()` invocations in your client?

1. 71 times. (when using recv(16)) => it depends on the buffer size.
2. No, it is not. Once again it depends on the buffersiz
3. The size of the full message that we need to receive (tot), the amount of bytes that we read for one "recv" (read_size) and if the packet contains the "FIN" flag or if it's in a separate packet (fin: 1 if false else 0). => ceil(tot / read_size) + fin

[A15]

What source code to submit? A single client (PDC) program that correctly implements all the different scenarios with the different commands. Name the file as `Part2_CAMIPRO1_CAMIPRO2` and put it in the same submission folder `Lab3_CAMIPRO1_CAMIPRO2` with all the other files you will submit. `CAMIPRO1` and `CAMIPRO2` are the Camipro numbers of the group members.

PART 3 – UDP PACKET TRANSMISSION

(Graded)

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 $0 \leq X \leq n$ as 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.

Your client should send the reset commands to a UDP server (PMU) running at `lab3.iew.epfl.ch` on port `5004`. However, 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 connect to the server on **both IPv4 and IPv6 sockets** and should detect automatically if the server runs on IPv4 or on IPv6 (for grading, 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.

Q16/ Write down the time at which you received an acknowledgement and if this acknowledgement was sent over IPv4 or IPv6.

[A16.a] Date and time: Fri Nov 4 12:05:05 2016

[A16.b] IPv4 or IPv6? IPv4

Run the client 60 times and count how many packets do you need to send before receiving an acknowledgement. This experiment can take more than 5 minutes. So you can leave the client running and continue with the next part.

de

Q17/ On average, how many packets do you need to send before receiving an acknowledgement? What is (approximately) the loss probability that you observe?

[A17.a] Number of packets before an acknowledgement: 4.2

[A17.b] Loss probability: ~80%

What source code to submit? A single client (PDC) program named as `Part3_CAMIPRO1_CAMIPRO2` and put it in the same submission folder `Lab3_CAMIPRO1_CAMIPRO2`.

PART 4 – UDP MULTICAST

(Graded) 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 `Hamlet` to their subscribers on an IPv4 multicast group address `224.1.1.1` on port `5005` 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.

Subscribers to the program need to be connected to the wireless access point provided in INF019 (`SSID: lca2-tcpip-labs`) to be able to successfully receive the multicast messages from the Internet TV program.

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 `5005`, and displays the exchanged multicast messages.

Q18/ Launch your multicast receiver and copy some of the exchanged messages in the multicast group and paste them here:

We only keep the real payload (drop the first 6 bytes).
There art thou happy: Tybalt would kill thee,

[A18] The law that threaten'd death becomes thy friend

Q19/ Are there any special socket options that need to be set in your receiver?

Yes, we have to join the multicast group by setting the IP_ADD_MEMBERSHIP option.

[A19]

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.

While using multicast, one of the important socket options is `IP_MULTICAST_TTL` for the multicast datagrams. In most operating systems this value is set to 1 by default.

Q20/ What is the affect of setting `IP_MULTICAST_TTL` to 1 in your sender? Do you need to change this value?

IP_MULTICAST_TTL = 1 is usefull if we don't want the multicast to go further than a single subnetwork. It controls the "scope" of the multicast. No, because the default is one.

[A20]

Q21/ Does your program (multicast sender) need to set any other special socket options to send the messages to the multicast group?

No, juste create the socket and then send a message

[A21]

Launch the multicast receiver you wrote in (4.1).–Also launch the two instances of your multicast sender (one for each group member, identified by your two CAMIPRO numbers) and have each send a text of at least 300 characters of your choice to the multicast group.

Q22/ Copy and paste the message that you get as a response in the multicast group once you send more than 300 characters to the group. (Copy the messages you get for both instances of your program.)

@224805: Your task is now completed. Congrats! Remember to upload your source code on Moodle! By the way, What kind of eggs does a wicked chicken lay? Deviled eggs.

[A22]

Note that, we have launched our program that joins the multicast group and records all the exchanged multicast messages. Therefore, we will know exactly what messages you will have received when we do postmortem analysis on the recorded data.

What source code to submit? Two files - one that implements the multicast receiver and named as `Part4_Receiver_CAMIPRO1_CAMIPRO2` and the other that implements the multicast sender and named as `Part4_Sender_CAMIPRO1_CAMIPRO2`. Put both files in the same submission folder `Lab3_CAMIPRO1_CAMIPRO2`.

PART 5 – WEBSOCKETS

(Graded)

5.1 – BACKGROUND

The WebSocket protocol (https://tools.ietf.org/html/rfc6455) is an 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 can allow for sending of messages where initial data is available but the complete length of the message is unknown.

The WebSocket resource URL uses its own custom schema: "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`

5.2 – 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. A WebSocket client can be implemented using different languages. However, to be consistent with the rest of the lab, we encourage you to use python to implement your client. For this, you will have to install the `websocket-client` module for python. 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 type and number 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. The client displays each such message from the server in a separate line.

Q23/ Start WireShark and run your WebSocket client such that it sends the `CMD_short:0` to the server. How many packets containing (part of) the message from the server have you seen coming from the TCP server (PMU)? Is this number different from the number of packets you observed in Part 2~~when~~ when $d = 0$?

[A23]

Q24/ How many times was the `on_message` event invoked at the client? Compared to Section Part 2~~when~~ when $d = 0$, which one needs more complex message handling operations to display the individual messages in a separate line?

[A24]

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.

Q25/ On observing on wireshark, how many packets do you see coming from the websocket server carrying part of the replied text? How many times is the `on_message` event handler triggered to receive the whole message? Comparing this to the number of `recv()` invocations in Part 2 ~~what~~ what does it say about the difference between WebSockets and simple TCP sockets?

[A25]

What source code to submit? One file that implements the WebSocket client and named as `Part5_CAMIPRO1_CAMIPRO2`. Put this file in the same submission folder `Lab3_CAMIPRO1_CAMIPRO2`.

PART 6 – EXCHANGE OF INFORMATION VIA A TLS SERVER

(Optional, for bonus)

This part of the lab is an extension of Part 2 ~~.~~ In Part 2, ~~w~~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 communication between the PDC (client) and the server (PMU) only for the part where the PDC sends the command `CMD_short:d` where $d = 0$. You are required to use TLS to secure the communication. Section 18.2 of Python 3.5 is a good starting point for you as it contains everything you need to do in this exercise including some examples (`https://docs.python.org/3/library/ssl.html`). We will grade only the server (PMU) part but for testing purposes you should also implement the client (PDC).

TLS stands for Transport Layer Security. It is a cryptographic protocol that adds encryption and authentication on top of TCP. Before the actual exchange of secured packets begins there is a TLS handshake phase during which the two communicating parties establish a TLS session. Messages exchanged during the handshake are secured with the asymmetric cryptography. Asymmetric cryptography relies on the certificate authorities and public/private key infrastructure. As the name says public key is available to everyone and it is distributed in the form of certificates that are signed by the certification authorities. Information encrypted with the public key can be decrypted only with the private key in the possession of the party to which a certificate with the corresponding public key is issued by the certification authority.

During the handshake phase, two communicating parties can authenticate to each other by exchanging the certificates. Furthermore, they also agree about a cipher and a symmetric secret key that will be used during the actual exchange of packets. Symmetric cryptography is used once the handshake is finished as it adds less computational overhead compared to the asymmetric cryptography. In the symmetric cryptography both parties use the same secret key.

We suggest you to read more about certificates and public-key/private-key security in general before proceeding. You can start by reading dedicated Section 18.2.4 in Python 3.5 documentation. After that you should be able to understand what follows. To simplify the exercise we ask you to implement only one-way certificate validation. Hence, only the PMU (server) will be required to provide its certificate. To this end, the TCP/IP TA crew has created a certification authority (CA) that relies on self-signed certificate `cacert.pem` (available on Moodle).

You are required to create two files: your private key (`CAMIPRO1_key.pem`) and certificate signing request (CSR) (`CAMIPRO1.csr`) which you will communicate to us (instructions below). We will act as a CA and provide you with your own certificate (`CAMIPRO1_cert.pem`). Make sure that, when testing your solution, you place CA certificate in the appropriate location on your respective OSs.

As noticed above, we use `CAMIPRO1` as part of the file names used in this lab. Always replace it with one of the team members' SCIPER numbers (e.g. 123456_key.pem and be consistent, of course).

The message exchange protocol itself should stay the same as in Section Part 2 –

Instructions for obtaining private key and certificate signed by our CA:

In order to generate a private key, execute the following command under Linux. If you have another OS you can use Ubuntu VM distribution you have in your virtual environment. Openssl is not installed by default. However, you can install it in two easy steps. First, you make sure you have Internet connectivity on any of the PCs (in the network settings choose NAT for Network access mode). Secondly, use the following command to install openssl:

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

Once you have openssl-enabled environment you can continue.

```
# openssl genrsa -out CAMIPRO1_key.pem 1024
```

This will generate a 1024 bit `.pem` RSA private key file. The run the following command to generate the certificate request. On executing this command, you will be prompted to enter some contact information (you can be creative if you want). Leave the challenge password empty and optional company name at the end. Once you finish you will have the `.csr` certificate request file.

```
# openssl req -new -key CAMIPRO1_key.pem -out CAMIPRO1.csr
```

Once you have your CAMIPRO1.csr file, send it to `roman.rudnik@epfl.ch`. Within 48h you will receive a reply with `CAMIPRO1_cert.pem file`. This is the certificate signed by our CA that you can use, in combination with the previously generated private key `CAMIPRO1_key.pem` and CA certificate `cacert.pem`.

Important instructions for automatic grading:

In order to automate grading we ask you to do minor changes in the files you upload. You should do these changes only when you finish testing in your own environment. In principle you should always use the latest version of a software. However, for maximum interoperability we will use SSLv2.3 as in the example in Python 3.5 documentation (Section 18.2.5).

```
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
```

What source code to submit? One file that implements the server (PMU) and named as `Part6_PMU_CAMIPRO1_CAMIPRO2`. You should also submit the private key `CAMIPRO1_key.pem` and the certificate `CAMIPRO1_cert.pem`. Put all these files in the same submission folder `Lab3_CAMIPRO1_CAMIPRO2`.