SERVICES – SOCKETS AND PROJECT SPECIFICATION

1. INTRODUCTION

1.1 OBJECTIVES

The purpose of this class is to learn how a client-server network application works and how to make a protocol specification.

1.2 LEARNING OUTCOMES

At the end of this class, you should know:

- How sockets can be used to send and receive messages over the network.
- How to capture network traffic between applications using Wireshark.
- How to make a protocol specification, using time diagrams to show the different modes of operation and how to specify the message format.

2. GENERAL GUIDELINES

2.1 WORK PLAN

i) Preparation

You should have a Linux machine for the Lab. You may use a Linux VM. For building a VM, you can follow the following guide from the course web page: VM-installation.pdf

It is recommended to install a shared folder between your PC and the VM as described in the VM-installation.pdf guide.

Make sure you have Wireshark installed in the Linux machine.

Download the ZIP files with the code from the course webpage to the Linux machine. Unzip them.

You are recommended to have a look into the code. This code can be used as a skeleton for you to build your client-server project application.

2.2 VIDEO-CONFERENCE CLASS

This class will be through video-conference, using the Zoom system. Make sure you follow the guidelines for setting up the video-conference.

Before the class starts, you should start the Linux VM, so that you can try by yourself during the class.

3. ACTIVITIES

3.1 CLIENT SERVER TEST

In the Linux machine, open 3 terminal windows in the folder where you have the code extracted from the Projecto-UDP.zip file.

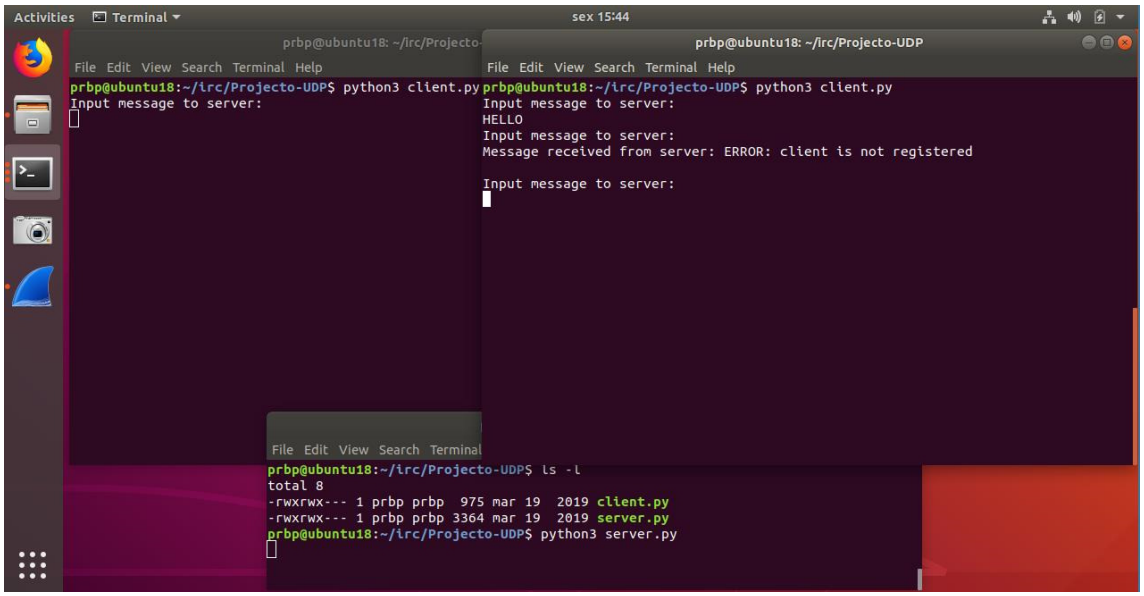
- Run the server code in one window using the command:

```
$ python3 server.py
```

- Run the client code in the two other windows using the command:

```
$ python3 client.py
```

- Open Wireshark and start capturing traffic in the "Loopback" interface.
- In one client type: **HELLO**
- Check if you get a new message with an error from the server. You should get something like this:



- Check the Wireshark window. You should have 2 UDP messages. If you have more messages, insert this filter "udp.port==12100" and press enter to apply the filter.
- Select the first packet and look into the UDP Data field. You should have 6 bytes: HELLO, plus a newline character.
- Select the second packet and look into the UDP Data field. You should have 32 bytes with the ERROR string shown in the terminal.

This shows how Wireshark can capture traffic between 2 applications running in the same machine. If they were in different machines, it would also be possible to capture the traffic by selecting the appropriate interface for capturing the traffic in Wireshark.

Wireshark is a great debugging tool for network applications!

The client application reads text from the keyboard and sends the text inside a message to the server. The server decodes the message, interprets its data and replies with another message to the client. This is a network protocol! The client receives the message and output the result to the user terminal.

3.2 CLIENT-SERVER PROTOCOL

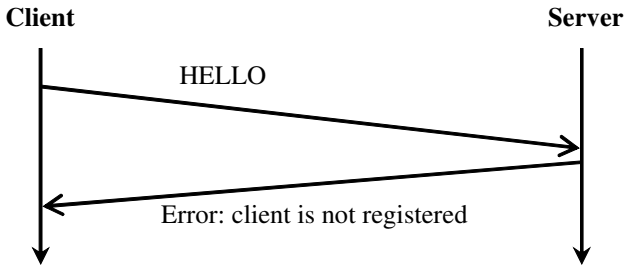
A network protocol can be specified with both:

- Time diagrams showing the valid or invalid sequences of messages between the network entities;
- Specification of the message formats.

i) Time diagrams

The time diagram for the previous test is something like:

A client sends an HELLO message without registering first with the server



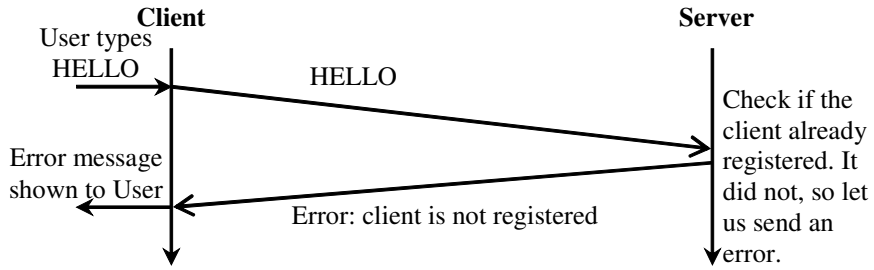
The vertical lines correspond to the time that flows from top to bottom. Each vertical line is for a specific entity communicating through the network. A label for each entity is placed on the top of the time vertical line.

Messages are represented by arrows flowing from one entity to the next. For each message, the message type and the most important message parameters should be shown.

Finally, the time diagram should have a brief explanation for the situation that is pictured in the diagram above it, like a title.

It is possible to include, in the time diagram, also the interactions with the users and what the server does:

A client sends an HELLO message without registering first with the server



In the first time diagram, they were omitted to simplify the time diagram. They will be omitted in the following diagrams, since the explanation above the diagram should make clear what happened.

The application allows for chatting between different users. Let us try those functions now. You can also check the traffic captured in Wireshark.

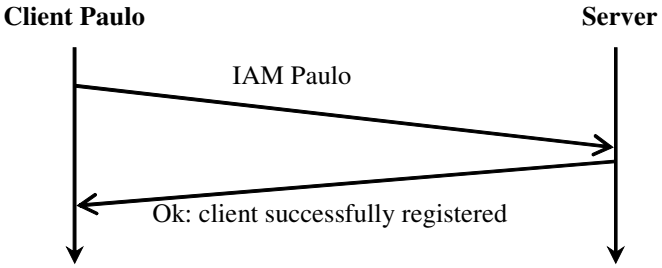
First, a user should register itself with the server.

- In one client, type: **IAM Paulo**

- You should get a success message.

The corresponding time diagram is:

Client Paulo registers with the server, by sending his own name. The registration is successful.

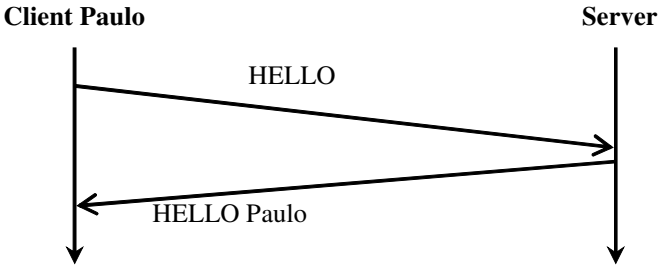


Once the client is registered, he can test the server, to check if the server is working correctly.

- In one client, type: **HELLO**

You should get something like what is pictured in the following time diagram:

A client, which is successfully registered with the server, sends an HELLO message to test the server and gets a HELLO back.



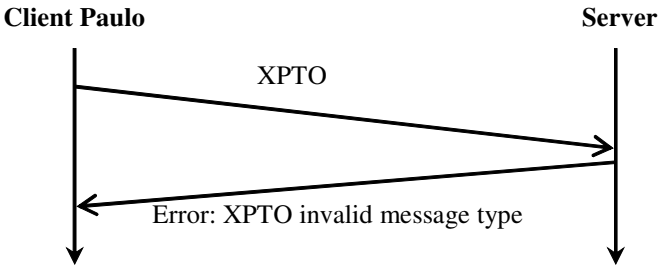
Note that now the server answer is different from the first case, because the client is correctly registered.

Now let us try some invalid message:

- In one client, type: **XPTO**

You should get something like what is pictured in the following time diagram:

A client sends an invalid message and gets an error back.

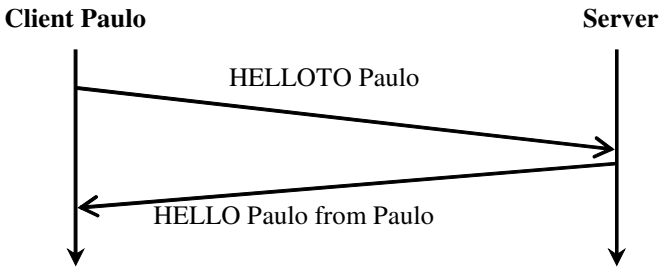


In this chat application, a registered client can send messages to other users. With a single user registered, he can send an HELLOTO himself.

- In one client, type: **HELLOTO Paulo**

You should get something like what is pictured in the following time diagram:

A client sends an HELLOTO to himself.

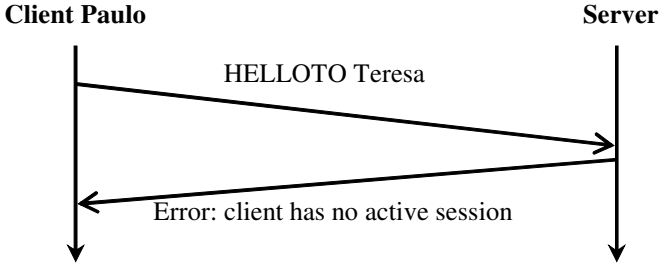


If the user sends an HELLOTO to an user that is not registered, he should get an error.

- In one client, type: **HELLOTO Teresa**

You should get something like what is pictured in the following time diagram:

A client sends an HELLOTO to user Teresa that is not registered with the server.

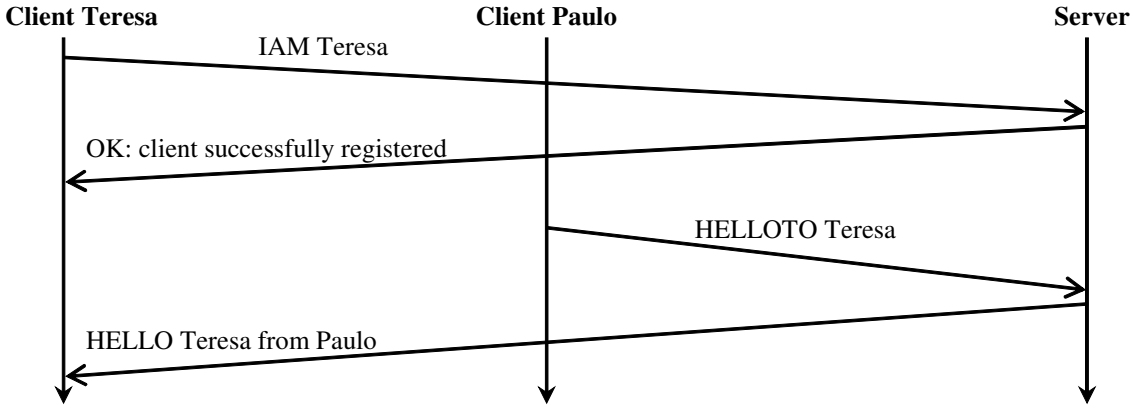


Now let us try with two users. First, register a second user. Then send an HELLOTO from one user to the other.

- In the other terminal window where the client is running, register user Teresa by typing: **IAM Teresa**
- In the terminal window where client Paulo is registered, type: **HELLOTO Teresa**

You should get something like what is pictured in the following time diagram:

Client Teresa registers successfully with the server.
Then client Paulo, which is already registered, sends an HELLOTO to user Teresa. Since user Teresa is already registered with the server, the message is correctly delivered to Teresa's client.



Note that now there are three entities in communication, so there are three vertical lines in the time diagram.

Exercise 1: send an HELLOTO message from Teresa to Paulo.

The application is not fool proof. Let us try some invalid situation.

- In one client, type: **HELLOTO**

What happened?

Note that the message parameter is missing, so it is an invalid message. However, the server should cope with invalid messages and provide the corresponding error message. It is very annoying for clients if some error shuts-down the service. All other clients will get no service.

Have a look at the code.

The client code is very basic. It has a select system call that allows simultaneously listening to the keyboard to receive what the user typed in the keyboard and listening to a socket that receives messages from the server. When something is typed by the user in the keyboard, a message with the text read from the keyboard is sent to the server. The server address should be known. When some message is read from the socket, the corresponding text is written in the terminal.

The server code implements all the functionality. The server is listening to the socket for receiving client messages. When a message is received, the type of message and the parameters are decoded. According to the type of message, a different code is executed to process the message and parameters, store the data received if required, find the data for the reply and generate a reply message. Then, the reply message is sent through the socket to the client. Finally, the server goes again listening for the next message.

Exercise 2: look into the server code, explain why it failed and try to correct this error by modifying the server code to cope with this error.

Note that there other situations that are not tested in the server and cause it to malfunction.

Exercise 3: try to find some other situation that is not tested and causes the server to malfunction (e.g. to die with some error).

In your project, all the applications (clients and servers) should be robust to misbehaving users, to misbehaving clients and to misbehaving servers.

You may terminate the server and the clients by pressing CONTROL+C in each terminal window.

ii) Message Format

To complete the protocol specification, all messages should be listed, including all the details about their parameters. It is particularly important to specify the message parameter separator and how messages are terminated. In the examples above,

message parameters are separated by a space character and messages are terminated by a newline character.

If the application protocol operates over UDP, application messages are delivered as a whole, or they are not delivered at all. What this means is that when an application sends a message, it sends the entire message, and when an application reads a message, it reads the entire message. Naturally, there is a message size limit, which is about 64KB, in modern Linux operation systems. But the message size limit may depend on the operating system you are using.

If the application protocol operates over TCP, which works as a byte stream, messages may be delivered partially, particularly if they are very large. If you recall the HTTP protocol, a message can carry a picture, which can be very large. Depending on the receiver buffer size and the network speed, the receiver may not receive the entire message in a single read call.

The differences between TCP and UDP will be explained in detail in the theoretical lectures. So, for TCP, it is very important for the client to be able to determine when it has read the entire message. This can be made by including a length field or a message terminator.

For the client-server example provided, the messages are:

Message Format Specification. **All messages are terminated with a newline character: '\n'** **All parameters are separated with spaces** **Optional parameters are shown with brackets, e.g. [type_of_parameter]** **Mandatory parameters are shown between, e.g. <type_of_parameter>**	
Direction	**Message format** **Brief explanation**
Client→Server	IAM <name>\n Message for user with name <name> to register with the server
Server→Client	OK: <explanation text>\n Success message, showing what the server did
Server→Client	ERROR: <explanation text>\n Error message, showing the cause of the error in the <explanation text>
Client→Server	HELLO\n Message to check if the server is running and a user is registered
Server→Client	HELLO <name>\n Message showing that the server is running and the user is registered with name <name>
Client→Server	HELLOTO <name>\n

	Message to send an hello to user <name>
Server→Client	HELLO <name1> from <name2>\n Message notifying user <name1> that he received an hello from user <name2>
Client→Server	KILLSERVER Message to terminate the server

The possible error messages the server can send to the client are (they are self-explanatory, so no brief explanation is provided):

Error messages
client is not registered
client has no active session
invalid message type

The possible success messages the server can send to the client are (they are self-explanatory, so no brief explanation is provided):

Success messages
client successfully registered
client registration updated

3.2 Online Quiz

You should answer one online quiz at the Fenix system about this class.