Lab 7

Multiple clients

[Compulsory]

Authors: Ragnar Nohre 2017, modified and moved to English Johannes Schmidt 2018-2022

This lab's goal is to deepen your knowledge in socket programming. We will implement a chat-server that can handle multiple clients.

7.1 The Problem

In this lab we will implement a *chat-server* that can handle multiple clients simultaneously. When one writes such a server program one has to solve the following problem:

> **Problem:** When a server program reads data with `sock.recv()` the program is *blocked* until the client sends data, and while the server is blocked it can not serve other clients!

This problem can be solved in at least four different ways:

1. We can let the server create a new *thread* every time a new client connects. That is, every client is handled by a dedicated thread. This solution has some disadvantages, however:
   (a) When the threads need to access common data, there is the risk of one thread modifying the data while another is reading it. The result can be completely unpredictable. There are solutions to this, but the program becomes quickly very complicated and an unfortunate implementation can lead to a *deadlock*.

   (b) Using a dedicated thread for each client does actually not completely solve our problem: while a thread is waiting for incoming data (with a blocking call to `sock.recv()`), it can happen that we need to send data to just *that* client.

2. One can use an asynchronous variant of the socket-API. The asynchronous variant of `recv` does not block and wait for the client's data. Instead it will call a so-called *call-back* function when the client has sent data. Since a call-back function is usually handled by another thread, one still has to handle common data appropriately.

3. One can use the `recv`-function in *non-blocking* mode, in order to implement the *polling* strategy. While this method still has disadvantages, we will explore it in the next lab.

4. In this lab we apply a fourth method: we write a single-threaded server that is sleeping (while listening) and waiting for all clients simultaneously. As soon as a client has data sent or a new client wants to connect, the server wakes up, handles the request and returns immediately to sleeping/listening.

7.2 Requirements on the chat-server

The server to implement shall have the following properties:

1. The server is single-treaded an uses *select* in order to handle multiple clients.

2. Several (unlimited in number) clients shall be able to connect to the server.

3. When a new client connects, the server shall send the message

   ```
   [IP-addr:portno] (connected)
   ```

   to *all* connected clients (excluding the client that just connected). `IP-addr` stands for the new client's IP-address and `portno` for its port number.

4. When a client sends a message (data) to the server, the server shall send that message to *all* connected clients (including the client who sent the message). Like this:

   ```
   [IP-addr:portno] M E S S A G E
   ```

5. When a client disconnects, the server shall send the message

   `[IP-addr:portno] (disconnected)`

   to *all* other clients (excluding the one that just disconnected).

Figure 7.2 on page 5 shows three different clients connected to the server, each has sent a message to the server.

7.3 Task and code skeleton

The code skeleton in figure 7.1 is half finished. Do not use copy-and-paste, instead type it yourself, understand it and replace then the `pass` and `# TODO` with appropriate code.

The program uses one socket for each client, and additionally one socket (`sockL`) in order to listen for new clients. The variable `listOfSockets` is a list that contains all those socket objects.

The call

```
tup = select.select(listOfSockets, [], [])
```

blocks the program until any of the sockets in the list has incoming data or a new clients wants to connect. Thus it is this line of code that *listens to all clients simultaneously* and additionally listens on the server socket (`sockL`) for new incoming connections. Read more on *select* in the documentation[1].

Test!

Test the server with an appropriate client. For instance *telnet*, *netcat*, or the program *client.exe* (Windows only, found on Canvas). You can also do the next lab where you will implement your own GUI-client for this lab's chat-server. Connect at least three clients to your server. Test your server locally (client and server running on the same computer) and over the network. Send messages and verify the server behaves as intended.

[1] https://docs.python.org/3/library/select.html

```
import socket
import select

port = 60003
sockL = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sockL.bind(("", port))
sockL.listen(1)

listOfSockets = [sockL]

print("Listening on port {}".format(port))

while True:
    tup = select.select(listOfSockets, [], [])
    sock = tup[0][0]

    if sock==sockL:
        pass
        # TODO: A new clients connects.
        # call (sockClient, addr) = sockL.accept() and take care of the new client
        # add the new socket to listOfSockets
        # notify all other clients about the new client
    else:
        # Connected clients send data or are disconnecting...
        data = sock.recv(2048)
        if not data:
            pass
            # TODO: A client disconnects
            # close the socket object and remove from listOfSockets
            # notify all other clients about the disconnected client
        else:
            pass
            # TODO: A client sends a message
            # data is a message from a client
            # send the data to all clients
```

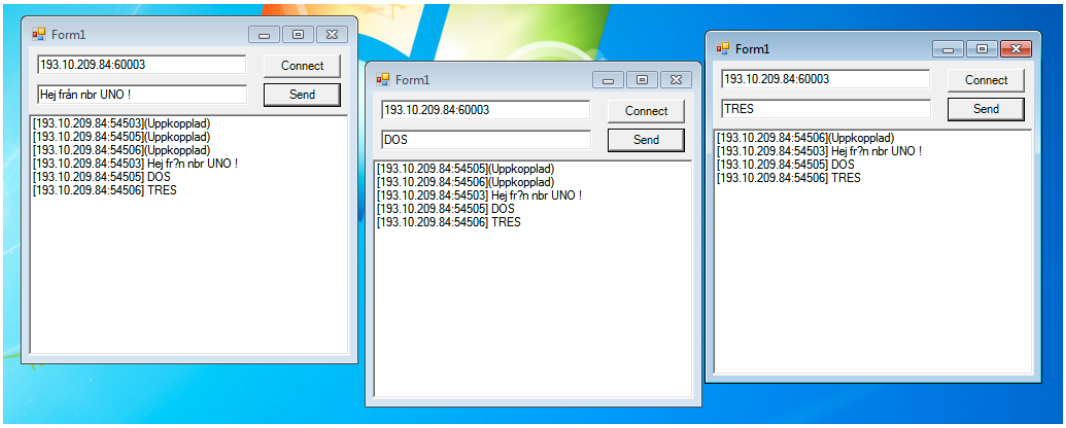
Figur 7.1: Code skeleton.

7.4 Hints

It can be useful to use the method `sock.getpeername()` in order to retrieve a client socket object's IP-address and port number.

7.5 Examination

Submit your file on Canvas and present your chat-server to your lab assistant.



Figur 7.2: Three clients are connected and have sent a message each. The client program from John McTainsh is downloaded from *http://www.codeproject.com/Articles/1608/Asynchronous-socket-communication*