

## Exercise 4

The goal of this exercise is to create a TCP server for serving requests to clients. In the first part, we are going to create a server and test it with telnet, a program for connecting to a TCP server and send virtual text terminal. Don't forget to read also the general instructions for the assignment

(<https://moodle.njit.edu.cn/moodle/mod/resource/view.php?id=2158>).

*Remember that while it is acceptable to discuss with the other students, the solutions submitted must be individual and **plagiarism is not acceptable**. If you reuse small snippets of code that you didn't write yourself, remember to mention **in English** where they come from, using code comments inside your code. However, remember that you should still be the main author of the solutions you provide.*

### Telnet

First make sure that you can use telnet on your computer. If you are using Windows 10, telnet is disabled by default but can be enabled like this:

<https://social.technet.microsoft.com/wiki/contents/articles/38433.windows-10-enabling-telnet-client.aspx>

Another (and possibly better) alternative for Windows is to use PuTTY which an open source SSH and Telnet client for Windows. If you don't want to install PuTTY, you can download a single executable that contains only the Telnet client:

- 32-bit version: <https://the.earth.li/~sgtatham/putty/latest/w32/puttytel.exe>
- 64-bit version: <https://the.earth.li/~sgtatham/putty/latest/w64/puttytel.exe>

(These files are also available on Moodle)

If you are using MacOS, recent versions do not include telnet by default anymore and will need to install it (see for example: <https://osxdaily.com/2018/07/18/get-telnet-macos/>)

If you are using Linux, it is very like you can install telnet with your package manager (e.g. "sudo apt install telnet" on Ubuntu or Debian)

(On Linux and MacOS, an alternative is also to use nc (netcat), see for example:

<https://www.igorkromin.net/index.php/2018/07/12/mac-os-has-a-much-better-tool-than-telnet-for-testing-remote-server-connectivity/>)

The instructions in this exercise will assume that you use **Windows** with the **PuTTY Telnet client**.

## TCP Server

Create a new package `programming3.chatsys.tcp` in which you will put all your classes relating to the TCP server and client.

First, we will create a basic HTTP server. In Java, we can create a TCP server with a `java.net.ServerSocket`. Create a class `TCPChatServer` that has the following attributes:

- An integer with the port on which the server will listen
- An integer with the timeout for the TCP session
- A Database object
- A Boolean that is true if the server is running
- A `ServerSocket`

Create a constructor to initialize the attributes and add two public method `start` and `stop`.

The method `start` is used to start the server (i.e. initialize the `ServerSocket`) and should, in a loop, accept connections by calling the method `accept` on the `ServerSocket`. The method “`accept`” returns a `Socket` object. Set the timeout of the `Socket` using the method `setSoTimeout`, so that the socket is automatically closed if it doesn't receive anything from the client after a certain amount of time. Then we will handle the connection with the client with a new `Thread` and a `TCPChatServerSession` (see below).

The `stop` method simply sets the boolean of the server to false to close the loop and also call the method “`close`” on the `ServerSocket`.

## TCPChatServerSession

A `TCPChatServerSession` is a class that implements `Runnable`. It has two attributes that are initialized in the constructor:

- A Database object (the same as the one used by the `TCPChatServer`)
- A `Socket` object (the one obtained when the server accepts a connection with a client)

As the class is a `Runnable`, it needs to implement the method “`run()`”. The method does the following:

- Create a `BufferedWriter` from the `OutputStream` of the `Socket` (obtained with the method `getOutputStream`)
- Create a `BufferedReader` from the `InputStream` of the `Socket` (obtained with the method `getInputStream`)
- Read a line from the `BufferedReader`
- Parse the line as defined by the protocol of ChatSys 4.0
- Take some action based on the protocol messaged received from the client.

## Protocol

The server can receive the following messages from the client.

- “`OK\r\n`”: OK messages can be sent by the client to keep the server alive (i.e. prevent the connection from being closed because of timeout). These messages are simply ignored by the server.

- “GET recent messages <N>\r\n”: a client sends this message to obtain the N most recent messages. The server responds with a list of N messages which includes the username of the author, the timestamp and the message. It sends the following messages back to the client
  - “MESSAGES <N2>\r\n” to tell the clients how many messages to read. N is usually the same as N2, but if the number of messages in the database is smaller than N, than N2 is the number of messages in the database.
  - “MESSAGE <username> <time> <message>\r\n” for each message to send.
- “REGISTER <username> <fullname> <password>\r\n”: a client sends this message to register as a user. The server replies with:
  - “ERROR username already taken\r\n” if the username already exists.
  - “OK\r\n” if the registration was successful.
- “LOGIN <username> <password>\r\n”: a client sends this message to authenticate itself, the server responds with:
  - “OK\r\n” if the username exists and the password match the one in the Database.
  - “ERROR <error message>\r\n” if the username or passwords are invalid.
- “POST <message>\r\n”: a client sends this message and the server replies with
  - “OK\r\n” if the user is authenticated.
  - “ERROR user unauthenticated\r\n” otherwise.
- “GET unread messages\r\n”: a client sends this message and the server sends back the list of messages since the last time this message was sent by the authenticated user. The server responds with:
  - If the user is authenticated, a similar answer as for “GET recent messages <N>” but with N2 the number of unread messages. (Don’t forget that the server also needs to update the user’s lastReadId in the Database!)
    - “MESSAGES <N3>\r\n” to tell the clients how many messages to read. N3 is the number of unread messages.
    - “MESSAGE <username> <time> <message>\r\n” for each message to send.
  - If the user is not authenticated, “ERROR unauthenticated user\r\n”.
- When receiving an invalid message that cannot be parsed, the server will reply with “ERROR <error message>\r\n”. Every time the server sends an ERROR message, it closes the connection with the client.

Start by implementing the server to handle the “GET recent messages <N>” messages.

## Testing the server

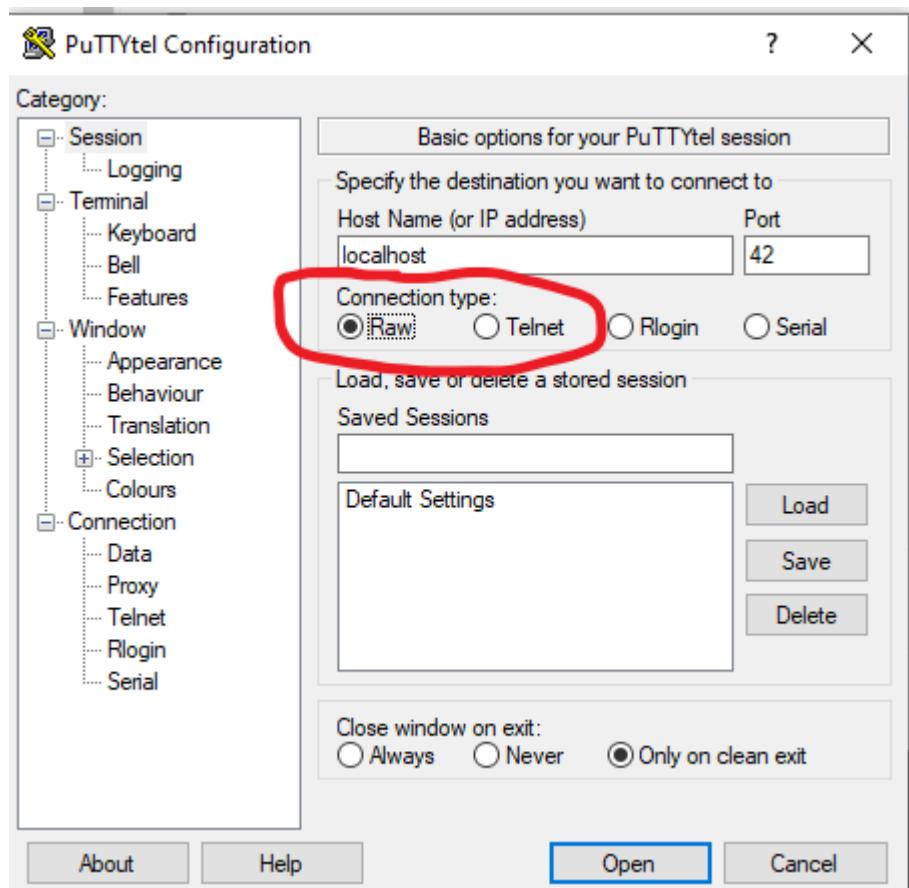
Before implementing the other messages of the protocol, you should test that it works properly. There are different ways of testing your server:

- Using unit tests. Ideally, you should split the logic of the server in different methods. Then you can create individual unit tests with Junit to make sure that small pieces (i.e. units) of your program work properly. For example, it is easier to write unit tests if you have a method for:
  - Determining the type of message of a given line
  - Parsing each type of protocol message
  - Generating the response message(s) for each request
- Testing manually with Telnet
- Testing by implementing a TCP Client

Ideally, you should use these three techniques to make sure that your server behave as expected when it receives a correct message, but can also handle incorrect messages. Remember that in the end, **your server must be able to keep serving clients if it receives an incorrect message!**

## Testing with Telnet

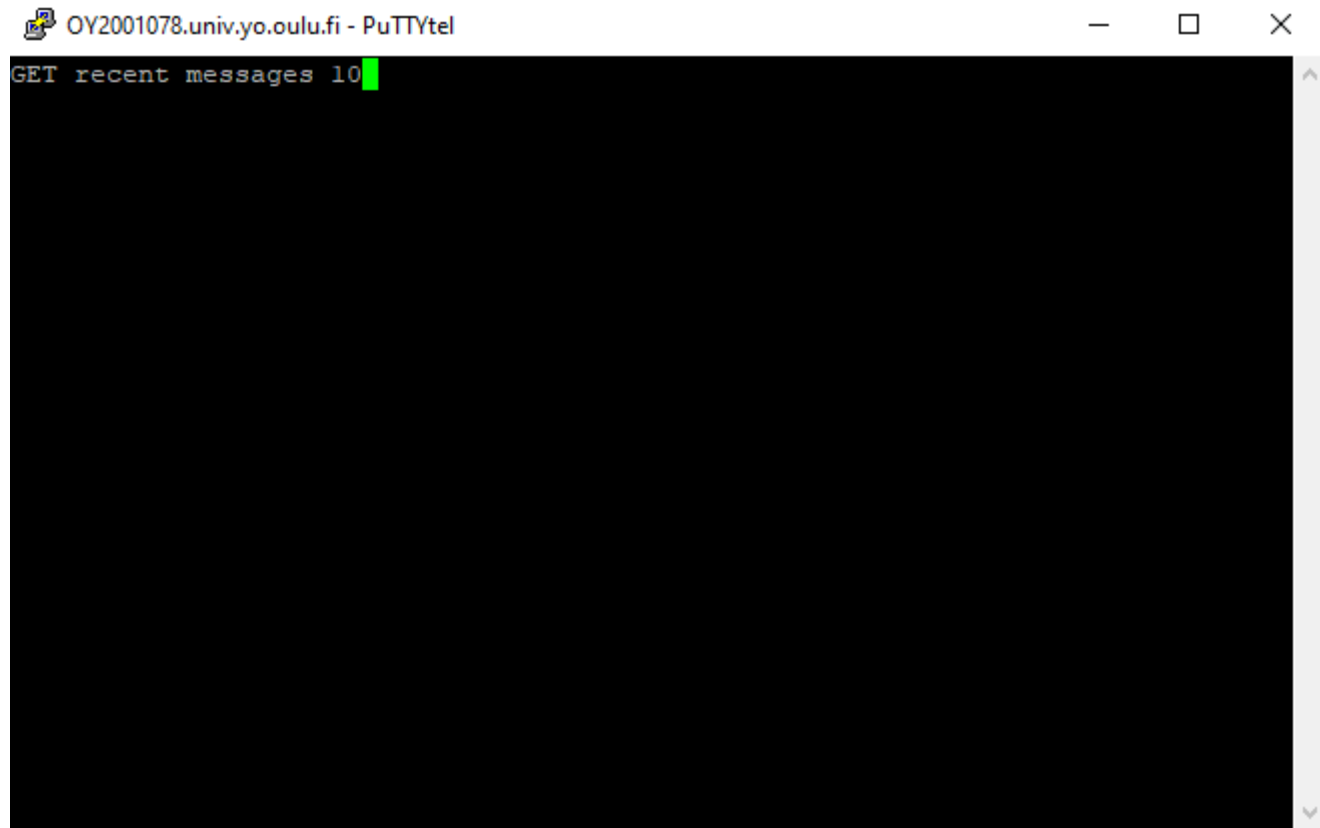
Telnet can be used to connect to your server. Assuming your server is running on port 42, you can connect to it like this:



Notice how the **connection type** is set to “**Raw**” and not to “**Telnet**” by default. This is important as our server is not a real Telnet server.

Once we click on “Open”, we get a terminal that allows us to send text to the server, and get an answer.

For example, we can type “GET recent messages 10”:

A screenshot of a PuTTYtel terminal window. The title bar at the top reads "OY2001078.univ.yo.oulu.fi - PuTTYtel" and includes standard window controls (minimize, maximize, close). The terminal area has a black background with text in a monospaced font. The text "GET recent messages 10" is entered in green, followed by a green cursor block. The terminal has a vertical scrollbar on the right side.

And once we press enter, we get as reply:

```
GET recent messages 10
MESSAGES 10
MESSAGE 634      client7 1603370046296   Hello client5!
MESSAGE 635      client1 1603370046308   Hello client8!
MESSAGE 636      client3 1603370046321   Hello client8!
MESSAGE 637      client9 1603370046334   Hello client8!
MESSAGE 638      client5 1603370046345   Hello client8!
MESSAGE 639      client7 1603370046360   Hello client8!
MESSAGE 640      client1 1603370046392   Hello client9!
MESSAGE 641      client3 1603370046426   Hello client9!
MESSAGE 642      client5 1603370046436   Hello client9!
MESSAGE 643      client7 1603370046454   Hello client9!
```

To end the connection with the server, we can press Ctrl+D. If we don't do anything for a certain amount of time, the connection will be closed by the server based on the timeout set to the Socket when the server accepted the connection (by calling `setSoTimeout`).

### Testing by implementing a TCP Client

While testing with telnet is relatively simple, it has several disadvantages. First, we need to type the protocol messages manually. If we make typos, we might make mistakes. Also, we must verify manually that the output is correct and cannot rerun the same test easily. In addition, the timestamp is shown as a number rather than a formatted date, which makes it not very user friendly.

To solve those problems, we can implement a client in Java instead. Moreover, if we wanted to provide a graphical user interface, we would also need for the GUI to act as a client with our server.

Once you are done testing your client with telnet, you should start implementing a client in Java. For that, create a new class (e.g. you can call it `TCPChatClient`). Creating a TCP client in Java is easier than a server: all we need to do is create a new Socket. For example, in the example above with Telnet, the server is running locally (i.e. localhost is the address of the server which points to our computer) and on the port 42. We can create a TCP (client) socket in Java to the server like this: `Socket socket = new Socket("localhost", 42);`

In the same way as for the server, we can obtain from this socket an `InputStream` and an `OutputStream` for creating a `BufferedReader` and a `BufferedWriter` so we can exchange messages with the server. Implement a client that ask the server for the 10 most recent messages and print them. Your client should provide an output that is more user friendly. For example, the timestamp of the message should be displayed as a formatted date and time rather than as a number.

### Mid-term assignment - Implement the full protocol

Now you should have a working client and server for receiving the most recent messages in the database. You should now implement the missing protocol messages defined above in order to get a fully functional chat server based on TCP. Make sure to test your server, with Telnet, unit tests and a client written in Java that your program behave correctly and can handle incorrect messages.

Once you are done, **make sure your code fulfills the requirements** for the mid-term assignment (<https://moodle.njit.edu.cn/moodle/mod/resource/view.php?id=2158>).

### Bonus: SSL connection

There is one issue with our TCP client: when a user register or authenticate, it sends its password over the network. Thus, a hacker could intercept the messages sent over the network and get access to the user's password!

In a real-world scenario, to prevent this from happening we would use secure connections that are encrypted. It is very common to do this using SSL/TLS which provides encryption on top of a TCP session.

As a bonus, implement SSL as part of your server (and client) to secure the connection between them.