Assignment 1 - Non-blocking sockets

Network Programming, ID1212

Bernardo Gonzalez Riede, begr@kth.se

November 17, 2017

1 Introduction

- Server and client have to communicate via blocking TCP connections. TCP is a transport protocol for a network, guaranteeing no lost packets iff a connection is established. Moreover, it uses a sequence number to ensure delivery in correct order.
- The client must not store any data. A stateless implementation of the client is preferred.
- The client must have a responsive user interface, i.e. being able to type commands while previous ones are still processed. This is at least best practice, if not a requirement in all programs.
- The server should be able to handle several clients simulteanously. Every client has to have its own game which doesn't get affected by other players.

2 Literature Study

The main literature for this assignment was a combination of all the videos related to streams and sockets, i.e. streams-, file-handler-prog-example-, sockets-, chat-program-part*.webm, and the related source code files. By studying the example and refering to javadocs the flow and layering coalesced to a fitting picture. Most importatn concepts obtained:

- The layering doesn't imply that in mvc some kind of view is always at the top. A more fitting description is *event initiator*, in case of the server the network layer which receives request from the client.
- Using an *enum* provides access to a finite set of constants which hold their own name as value. This is even faster than an integer for switching. Perfect for entering commands

- Observer pattern can be used to prevent upcalls from layers to layers above.
- (G)UI can be made responsive using an additional thread for commands with latency, i.e. network communication.
- Stream reading can be layered, e.g. a *DataOutputStream* over a *BufferedOutputStream* over a *FileOutputStream*.
- The user interface must be informative.

3 Method

The MVC approach was used to solve this problem. Although a bit overkill, practicing it will payoff over time. This meant to use various packages and to have high cohesion with low coupling. Additionally upcalls should be avoided.

The project is in pure Java, using the included libraries. Developing was done in Netbeans.

4 Result

Link to public Github repository with codehttps://github.com/MemBernd/ID1212-blockingSockets https://github.com/MemBernd/ID1212-blockingSockets

4.1 Blocking TCP sockets

The socket on the client side is created in client.net.ServerConnection on lines 29-31. For a socket to be able to connect it needs an adress, composed of the host and a port, and a timeout. The *setSoTimeout()* sets the time the socket is blocked.
The server has its implementation of sockets in server.net.Server on lines 31. Aftewards it stays in a loop accepting connection. When it receives a request on the specified port (configured default of 54321), it accept it, receiving an available socket form the OS for further communication. This socket is passed to *handleClient()* where timeouts are set.

4.2 Stateless client

The information of the game which have to tracked are:

- current word trying to guess
- progress in guessing the word
- attempts left
- score

The variables are stored in server.model.GameState. They are defined on lines 18-21. While the score is initialized at the creation of a client handler, the remaining variables are initialized on lines 15-78 inside *initializeGame()* which gets called when the user starts a game.

4.3 Multithreaded client

Client.view.Interpreter, the class which the user interacts with, extends the *Thread* class and isstarted by client.startup.Hangman. It creates a client.controller.Controller which itself creates a client.net.ServerConnection. This class holds an inner class, at line 56, *Listener*, extending *Runnable*. On line 35 of client.net.ServerConnection the second thread of every client is created, an instance of *Listener*.

4.4 Multithreaded server

The amount of threads of the server is 1 + amount of connected clients. The startup starts server.net.Server which listens permanently on line 33. When receiving a request it starts server.net.Handler, an extension to *Thread*. Since the users don't share any data, everyone having their own thread, a complete mvc, was selected. Because accepting a request on line 33 creates a new socket for each client, no need for an id or similar was needed to communicate a client with the correct thread.

4.5 Communication

Both, client & server have access to protocol.Constants where some constans, such as the delimiter for the communication are defined. Since the client doesn't have to process the received messages in any way, it just prints them out.

4.6 Layering

The client is divided as follows:

- The model is equivalent to the client.net package holding *ServerConnection*.
- The controller resides in client.controller.
- The view has classes in client.view, the most interesting one being *Interpreter*, the UI.
- Client.startup.Hangman is the startup class.

On the server side, the layering is quite similar:

- Server.model.GameState holds the game's state. Every server side client thread has its own.
- Server.controller.Controller is the controller.

- Equivalent to a view, in a sense of event driver, is the server.net package. It contains *Server* and (client)*Handler*.

To avoid a upcalls from client.net.ServerConnection to the view, client.net.OutputHandler has been created. The previously mentioned *Listener* thread expects such a class as argument when instancing. The client.view.Interpreter has an extending class to *OutputHandler* as a inner class which gets passed through the controller to ServerConnection. Since its a specialization, it can used wherever its superclass can be used. This enables it to print output without making an call to the view.

4.7 Informative UI

In Figure ?? several of the information provided to the user and his input can be seen. The instructions at the start of the client, establishing a connection, starting a game, doing a correct (lucky) input and an incorrect one.

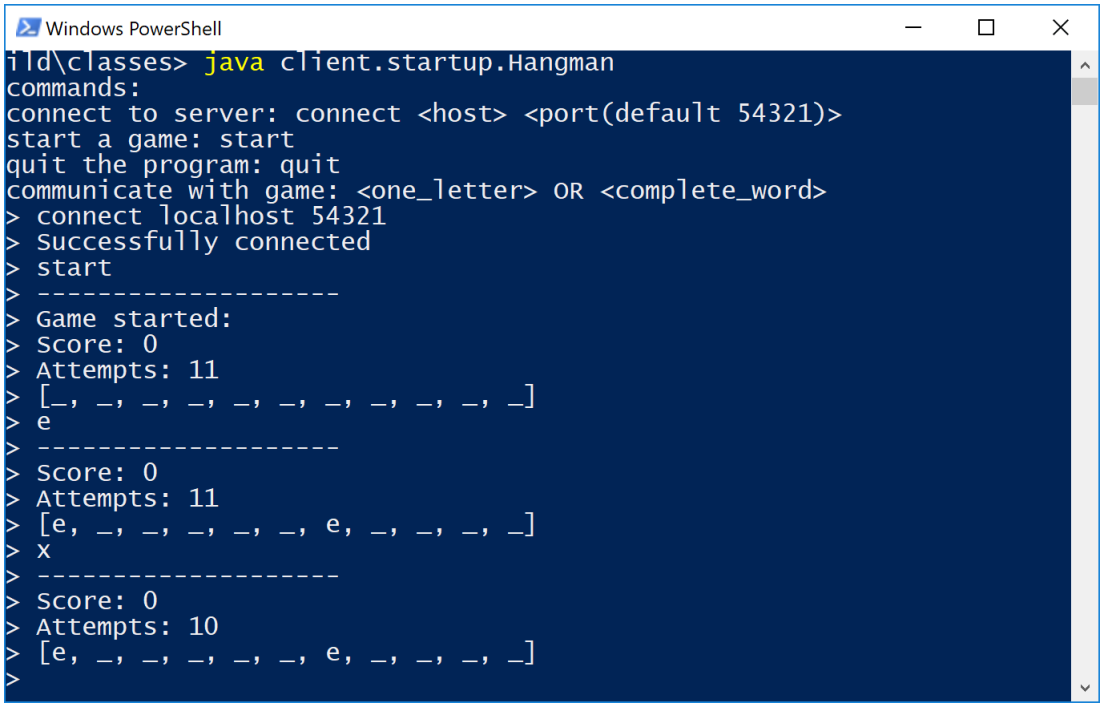

Figure 1: Instructions and game state on client side

Figure ?? shows 2 clients playing at once. Moreover, the timeout used can be seen on the server thread in the right bottom command line.

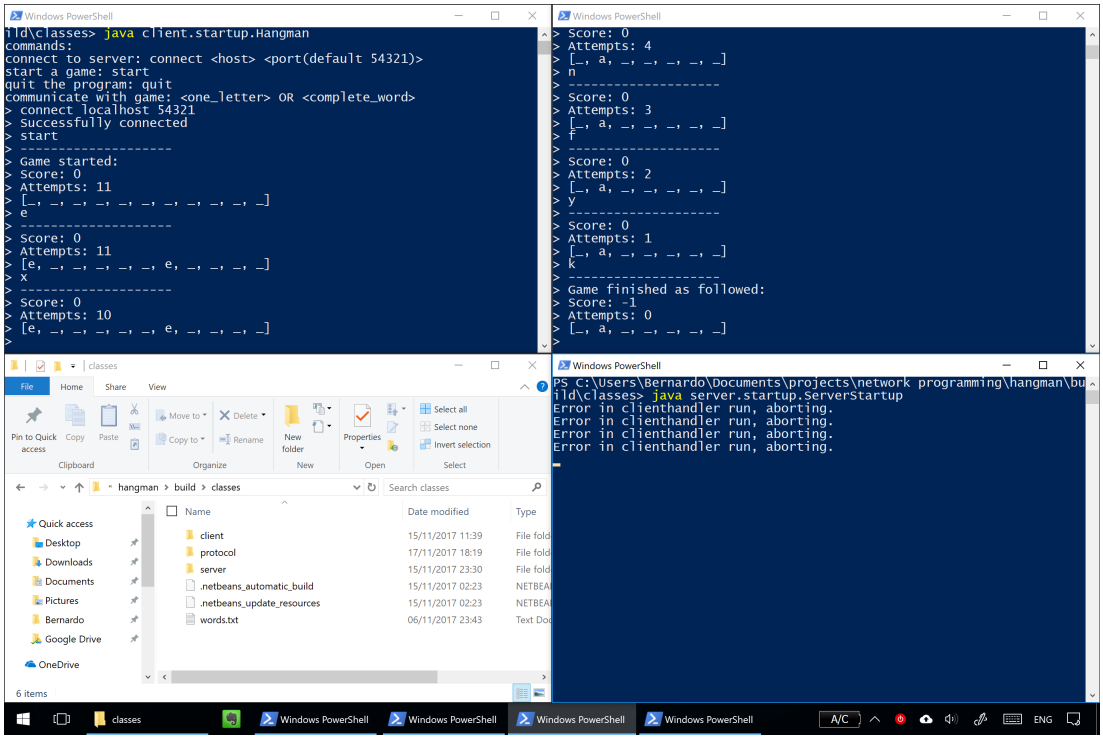

Figure 2: Concurrent players.

5 Discussion

All requirements have been met. Difficulties, once having an understanding, were only encountered regarding on how to use the functions. It was a lengthy process due to wanting to recreate an elegantly structured program. At some points the struture of the code itself suffered from it, e.g. server.model.GameState.finishes() needs cleaning.

6 Comments About the Course

The code for the sockets is elegant. Surely enough it creates a bias towards how the implementation can/should be done, resulting in a similar code. I guess this is why it's mentioned that the copy cannot be copied, though it may be similar.

It took me about 17+-1 hours to do everything, watching videos, planning, coding and writing the report. I tracked the time with toggl.