Assignment 1 - Non-blocking sockets

Network Programming, ID1212

Bernardo Gonzalez Riede, begr@kth.se

November 24, 2017

1 Introduction

Goals needed to accomplish Task 1, Hangman with non-blocking sockets:

- Usage of only non-blocking sockets.
- Server being able to handle concurrent clients, i.e. multithreading.
- Responsive user interface, i.e. multithreaded.
- Hangman game functioning as expected.

2 Literature Study

Although not literature in a classic sense, the provided example of the chat server and client shed much light over the usage of selectors. Instead of having a thread block on a method, e.g. readLine(), an abstraction of a stream, called channel in the nio package, can be created and registered under one common selector. The connection of a channel with a selector is called a key. Said key can hold an attachment and has an operation, which its waiting for, registered. The result is only one blocking method, Selector.select(), which returns when a change in a channel happens or when Selector.wakeup() is manually called.
A Selector can have multiple channels attached.

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 code: https://github.com/MemBernd/ID1212-Non-Blocking-Sockets

4.1 Non Blocking TCP sockets

4.1.1 Client

The usage of a Selector to provide non-blocking sockets has been implemented in client.net.ServerConnection The very first method call on it is connect() (l.93) where a new InetSocketAddress is created with the hostname and port provided. Moreover this class, being an implementation of Runnable, is submitted to its own thread. This starts its run() (l.47) which initializes the channel by:

- Opening Selector and channel
- Configure channel as non-blocking
- connect to the InetSocketAddress
- register the channel in the Selector with interest in the *connect* operation

From there on, the thread lives in a loop, reacting to changes and wakeups in the selector. This loop has different actions associated with different operations (l. 61 - 68) In the first iteration, having the operation set to *connect*, it will finish establishing the connection. Being the initiator in all communication, the client has to write a command which passes through sendMessage(String) (l.115). Using a lock, this method appends the message to a queue (having added a length header), sets a boolean flag to true and wakeups the Selector, aka the loop of ServerConnection.

The boolean flag makes sure the operation is set to *write* and empties the queue by calling sendToServer(). This function loops through the queue and sends every message on line 144, channel.write(ByteBuffer). Afterwards the operation of the key is set to *read*, expecting an answer from the server.

When the channel has information to be read, the selector calls receiveMessage() (l.123). An if conditions makes sure that the connection isn't lost, which would return -1. Afterwards verifyMessage(String) (l.176) is used to match the length header to the actual body size of the message. If everything is right, the body of the message is returned and printed, otherwise an exception is raised.

4.1.2 Server

The Selector implementation can be found in server.net.Server, while the functions which change the operation of the channel are called from the corresponding server.net.Handler. The server lives in serve() in a loop including the selector.select() and the actions to handle the different operations of the channels. Again, at the first run, following actions are performed:

- Opening Selector and channel
- Configure channel as non-blocking
- connect to the InetSocketAddress
- register its own channel in the Selector with interest in the *accept* operation

The operation of its own channel won't change during execution, instead a new channel is created every time a client connects. It gets registered with the same Selector, with operation *read*, and a server.net.Handler is created and attached (l. 93 - 98). Every change in a channel in its state of operation returns selector.select() and calls the appropriate action. The logic itself is similar to the one from the client, except the sequence of methods jumps between handler and server, and that the server stays in *read* only changing briefly to *write* when sending the reply. Sequence of jumps:

1. Key has been selected because there was a change (write) and is readable (Server l. 56).
2. The attached Handler is parsed from the attachement of the key and its sendMessage() is called (Server l. 67 - 70).
3. The channel is read and and stored in a global (in Handler) ByteBuffer (Handler l. 91 - 93).
4. The Handler submits itself to a ForkJoinPool, therefore run() (Handler l. 41) starts.
5. Depending on the content of the message, the correct actions is exectued.
6. Assuming the action involves sending a reply, Server.reply(socketChannel) is called.
7. Server.reply(SocketChannel) sets the operation of the key to *write* and wakes up the selector. (Server l. 78 + 79)
8. Because of the operation *write* being set, Server.send(SelectionKey) is used, which, again, parses the Handler to execute sendMessage() in it.
9. Handler.sendMessage() writes the previously (during run()) created reply to the channel and returns. (Handler l. 101)
10. Still in Server.send(SelectionKey), the operation is set to *read* again.

4.2 Multithreaded client

Same as in previous assignment, 1 thread for input, 1 for output.

4.3 Multithreaded server

The server has one thread for listening for incoming new connections and when processing incoming messages, the Handler submits itself to a ForkJoinPool (l. 97). Therefore the amount of concurrent threads is limited and the threads die after processing the message and having created the reply.

4.4 Communication & handling of incomplete messages

Both, client & server have access to protocol.Constants where some constants, such as the delimiter for the communication are defined. The same methods for prepending the length header and verifying the length header exists in server.net.Handler (l. 111 - 126) and in client.net.ServerConnection (l. 169 - 184).

4.5 Layering

The Layering hasn't changed from the previous assignment, but has been included anyway.

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.

4.6 Hangman game

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 and doing some input.

Figure 1 shows 2 clients playing at once.

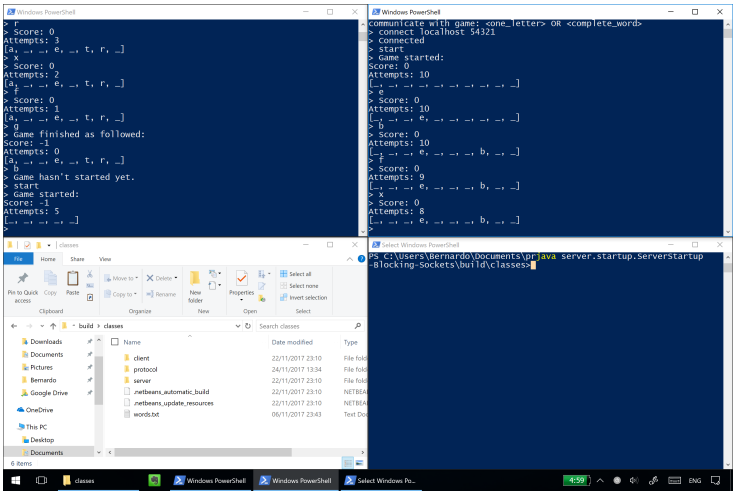

Figure 1: Concurrent players.

5 Discussion

A slight downside to the approach of using locks, i.e. the synchronized keyword in Java, for accessing the message queue which should be sent to the server, is the following observed behaviour. If the sending takes long, e.g. inserting Thread.sleep(10000) on line 143 (ServerConnection.sendToServer()), and the user types more commands which have to be sent then no *prompts* are printed instantly. This is the result of the sendMessage() waiting for the lock to release to be able to queue the command/message. This doesn't affect the responsiveness per se, since the user is still able to input commands which get executed.

This behaviour can be observed on the provided chat example, too, if Thread.sleep(X) is inserted in ServerConnection.sendToServer() on line 201.

6 Comments About the Course

The time invested in this project were 13 hours.

- 6h 46min for coding.

- 2h 25min reading the code.
- 1h 57min for the report.
- 2h watching videos, taking notes and thinking about how to tackle the problem.