

Homework 2, Non-Blocking Sockets

Network Programming, ID1212

Vasileios Charalampidis, vascha@kth.se

November 24, 2017

1 Introduction

The purpose of this second assignment is to develop a distributed application, while using a specific communication protocol for process interaction using non-blocking TCP sockets. Also, concurrent threads in nodes are introduced in the homework, as an efficient way of improving scalability and performance, so as to hide communication latency.

To meet these goals, the Java application that was deployed in the first assignment (Hangman game) has been updated here. The rules are exactly the same. The game is played with a client (or many clients at the same time) and a server. More specifically, the client connects to the specific server, and after the establish of the connection he is able to start a new game. The server responds with a new hidden word that the client needs to find out. The client has as many attempts as the letters in the hidden word. He is able to guess with one letter or give the whole word. If the client gives a wrong guess then his attempts decrease by one. Otherwise, the attempts stay stable. If the client manages to find the word then his score increases by one. Otherwise, if he loses, the score decreases by one (possible negative score is acceptable). The client can start or quit the game at any time.

The deployed program meets some specific requirements, so as to present an acceptable solution for this assignment. The first requirement is that only non-blocking sockets are used, in comparison with the first assignment that we had blocking TCP sockets. The second requirement is that both client and server are multithreaded. This means the same as in homework one, that the server can handle multiple simultaneous clients and that the client has a responsive user interface. The third requirement is that the program still works as expected. This will be shown by some screen shots from the execution of the program.

2 Literature Study

In order to accomplish this assignment, it was very important at the beginning to understand the differences between non blocking and blocking TCP sockets. After the

realization of the advantages that come with non blocking sockets, the given material i.e. code and videos, at canvas for this course, was more than helpful for the java implementation. The videos were very informative, as the instructor explained in good detail all the basic changes that needed to be made to have a non blocking socket representation with the appropriate thread handling. The code of the chat-program was again close to Hangman game, with the difference that different clients expect different results i.e. every client plays a different game. Except from the provided course material, information about Java nio package found on the web was also part of dealing with the application.

What became clear from the materials used, was that non blocking sockets have several advantages over the blocking ones. The non blocking sockets are capable of accepting new connections immediately, whereas the blocking sockets if they perform a specific task may block new incoming connections. Also, in non blocking sockets, threads will not wait for read/write methods. One thread can be used for all operations (accept, read, write), while in blocking sockets we had one thread for each connected client. This means that large number of connections can be handled better in non blocking than in blocking sockets. In the case of non blocking sockets, a selector is used with acknowledgment of when to accept/read/write. As for the thread that is bound with the specific selector, the handling of a specific message is passed to a different thread (use a threadpool for example), in order to save time. The drawback of using non blocking sockets over blocking ones is the fact that the programming of the first can become a little bit more complex. If this complexity can be appropriately managed, then the performance will be certainly higher compared with blocking sockets.

3 Method

To make the appropriate changes to the first Java implementation, again the "IntelliJ IDEA" by JetBrains was used. It is a Java integrated development environment (IDE) for developing computer software, with focus on Java applications. It is a convenient tool for the user as it returns code predictions, snippets, error highlighting, and many more.

Before proceeding with the actual update of the hangman game, some tests were performed to smaller units e.g. a simple TCP non blocking socket for sending and reading requests, etc. so that to be sure that specific concepts will work in the final implementation.

To evaluate that the final solution met the requested requirements, several manual acceptance tests were performed to the code e.g. client to quit at any time of the game, several clients playing at the same time, etc. Also, appropriate system messages were introduced as a way for following up the program flow.

4 Result

A layered architecture was again used for this updated assignment (Figure 1). The only difference with the structure here has to do with the dismissal of the client's controller package. The only reason to use a controller in homework 1 was to handle threading by using a threadpool, but here there is not such thing. There is only a dedicated thread handling of all network communications introduced in client's net package in `ServerConnection` class (line 25).

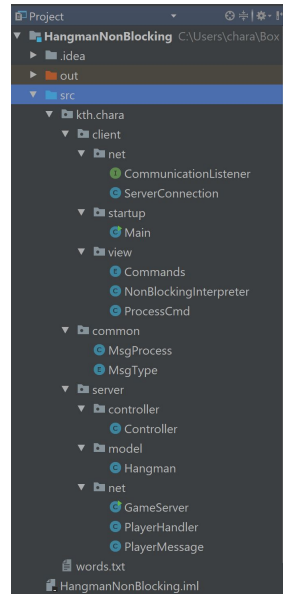


Figure 1: The architecture of the updated Hangman game

Going through the changes introduced in this updated version, in the common package a new class "MsgProcess" was added. This class is used from both client and server. The main idea behind the class was the fact that now the messages include a length header, as usually happens in most real case applications.

There are several methods inside "MsgProcess" class:

- a "messageJoiner" method (line 23), which wraps a string message to a ByteBuffer (without length header).
- a "withLengthHeader" method (line 31), which returns a string message including the length header of it.
- an "appendMsg" method (line 43), which adds a new received message to previously received ones.

- a "nextMsg" method ([line 52](#)), which returns a String[] with the message type and body, while removing it from the message queue ([line 53](#)) that was previously created ([line 15](#))
- a "hasNextMsg" method ([line 60](#)), which returns true or false depending if the message queue is empty
- an "extractMsg" method ([line 64](#)), which returns true or false if the complete message was received. At the same time it adds the message to the message queue
- a "completeMsg" method ([line 85](#)), which is used by the previous method to define the correctness of the received header length of the message

Continuing with the client's view package, changes were made to the "NonBlockingInterpreter" class. The class continues to implement a Runnable method ([line 14](#)), which aims to give a responsive UI to the client (one of our requirements for the assignment). A new thread is started for each client ([line 23](#)), that handles the UI and the calls to the "ServerConnection" class. A change was made in [line 52](#), where each client registers as a listener to receive responses from the server ([line 81](#)), by implementing the "CommunicationListener" interface, found in client's net package.

In client's net package changes were made at the "ServerConnection" class, so that to manage all communications by implementing a Runnable method ([line 25](#)). This class now introduces non blocking sockets ([line 28](#)). A new socket address is created for the client to connect with the specific server by giving the host and port ([line 52](#)). After that a new thread starts ([line 53](#)). In the run() method, a socket channel is opened ([line 59](#)), which is configured to be non blocking ([line 60](#)). After that the specific socket channel establishes a connection with the server ([line 61](#)). A selector is opened ([line 64](#)), that the socket channel bounds to ([line 65](#)), by setting an interest in connecting operations. A loop is initiated while being connected with the server. At the beginning, the selection key was created for connection operations. As a result, the "finishConnection" method is called ([line 82](#)), to actually finish the socket channel connection with the server ([line 96](#)), by setting the key interest to write operations ([line 97](#)) and notifying the listener for the connection with the server ([line 98](#)). Now, the selector blocks the flow of the "ServerConnection" class ([line 73](#)) until a selection is made.

When the client writes a new command to send to the server, this message is firstly wrapped as a ByteBuffer ([line 154](#)) and added to the message queue of the client ([line 155](#)). At the same time, an order to the selector is send to wake up ([line 157](#)). As a result, the selector unblocks and the "sendToServer" method is called ([line 86](#)) to write the message to the buffer and remove it from the message queue. After that, the interest is set to reading operations ([line 135](#)). If a new response comes from the server, then the "responseFromServer" method is called ([line 84](#)). In this method the ByteBuffer is properly changed to readable string ([line 109](#)), which is then send to the "notifyListeners" method to notify the corresponding client. The last method ([line 168](#)) executes a threadpool for multithreading.

Furthermore, about the server package, changes were made to "GameServer" and "PlayerHandler" classes in the net package. For the first one, a selector is opened ([line 34](#)), in which the server socket channel is registered ([line 39](#)), while setting the interest to accept operations. The "connectPlayer" method ([line 54](#)) is initially called. This method will create a non blocking socket channel for each client that connects for the first time ([line 70](#)). Each socket channel will be registered to the selector ([line 75](#)), by setting at the same time the interest to read operations. In [line 72](#) a new player handler is created by passing the unique socket channel of the corresponding client. The selector blocks the flow until a new message come from the client ([line 47](#)).

When a new message arrives, the "readFromPlayer" method is called ([line 56](#)). In this method, firstly the attachment from the corresponding key is retrieved ([line 81](#)). Then the "receiveMessage" method in PlayerHandler class is called ([line 83](#)) to retrieve the ByteBuffer in a String format and perform further actions by the server depending on the message type.

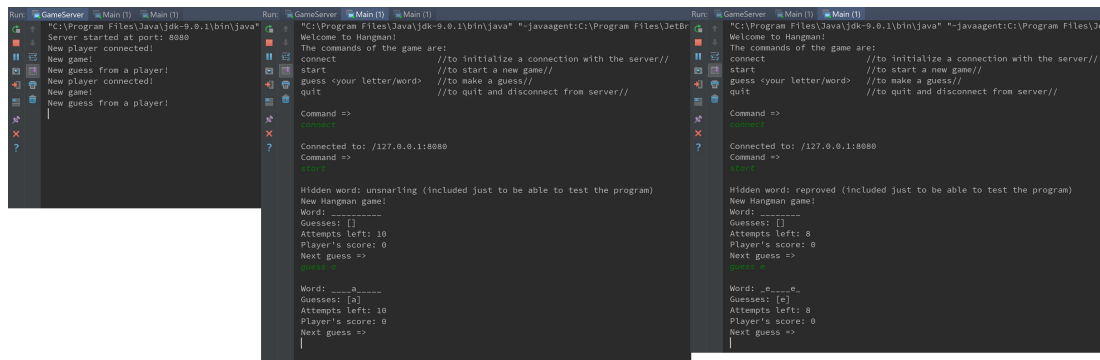
When the server is ready to response to the client, the "writeToPlayer" method is called ([line 58](#)). In this method, the "sendMsg" method from PlayerHandler class is called ([line 92](#)) to write a ByteBuffer at the socket channel. The interest is again set to read operations ([line 93](#)).

Other important methods used in the "GameServer" class are a) the "deleteClient" method, which disconnects the player and cancels his key ([line 99](#)), b) the "selectorOn" method for waking up the selector ([line 103](#)), and c) the "appendQueueKey" method for adding keys in a specific queue for writing operations ([line 108](#)).

The "PlayerHandler" class implements a Runnable method ([line 18](#)). As mentioned earlier, it corresponds to the handling of all communication with one particular client, e.g. receiving a message ([line 38](#)), extracting the message from a ByteBuffer ([line 49](#)), sending the server response to write in a buffer ([line 86](#)), and disconnecting the client by closing his socket channel ([line 106](#)). The run() method ([line 57](#)) sends orders to the controller package for handling the game purpose operations (which do not change).

The above paragraphs explained pretty much the first two requirements of this specific assignment i.e. use only non blocking sockets and that both client and server are multithreaded. The last requirement refers to a fully functional Java code, which follows the rules defined in the introduction part. This can be seen clearly in [Figure 2](#) where everything works properly without any issues to observe.

The final code of the Java application can be found [here](#). Several comments are included there discussing about essential parts of the program.



The image shows three terminal windows side-by-side, illustrating the execution of a Hangman game. The leftmost window shows the server's output, including messages like 'Server started at port: 8080', 'New player connected!', and 'New guess from a player!'. The middle window shows the client's command-line interface, displaying the game rules, a list of commands (connect, start, guess, quit), and the game state (hidden word, guesses, attempts left, player's score). The rightmost window shows the client's output, including the hidden word 'unsnarling', the guesses '[]', and the attempts left '10'.

Figure 2: The system works as intended without errors

5 Discussion

The main goals of the assignment were to learn about non blocking TCP sockets and the corresponding multithreading, while becoming familiar on how to implement those concepts in a real Java application that deals with the classic hangman game.

The requirements of the presented code can be summarized in applying only non blocking sockets, using the appropriate multithreading for both client and server, and having a fully functional system. They were all accomplished in an efficient and proper way, without having any errors or bugs to report. Any problems that occurred during the execution were immediately identified as many 'throw exception' rules were implemented in the Java code.

If something was to be done differently, that would be again as in the first assignment, the testing part. It would have been beneficial to perform unit tests (e.g. [JUnit](#)), so that to guaranty that everything performs fine without any issues.

6 Comments About the Assignment

The time spent on this second assignment was approximately two days (with several breaks in between).