

Homework 1, Sockets

Network Programming, ID1212

Vasileios Charalampidis, vascha@kth.se

November 17, 2017

1 Introduction

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

To meet these goals, a Java application was developed to represent the classic Hangman game. 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 the client and the server must communicate by sending messages over a TCP connection, using blocking TCP sockets. This implies that a specific host and port are defined to establish communication over listening TCP sockets. The second requirement is that the client must not store any data. This means that all data entered by the user are sent to the server for processing, and all data displayed to the user must be received from the server. The third requirement is that the client must have a responsive UI, which means it must be multithreaded. The user must be able to give commands, for example to quit the program, even if the client is waiting for a message from the server. The last requirement is that the server must be able to handle multiple clients playing concurrently, which means it must be multithreaded.

2 Literature Study

In order to accomplish this assignment, it was very important at the beginning to understand the given task and all its requirements by carefully reading the instructions. After realizing the requested concepts, the given material i.e. code and videos, at canvas for this course, was more than helpful. The videos were very informative, as the instructor explained in good detail all the basic necessary concepts for the assignment. The code of the chat-program was close to Hangman game, with the big difference that different clients expect different results i.e. every client plays a different game. Except from the provided course material, information about Java implementation of specific concepts found on the web was also part of dealing with the application.

What became clear from the materials used, was that multithreading is very important in terms of achieving a responsive user interface, as well as improving the response time in a distributed system where communication latency is a significant factor. About TCP sockets (and multithreading), they have a specific way of implementation in Java, which was well illustrated in the course material, so as to be able to also introduce it to the requested Java application.

3 Method

To write the Java game, 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 hangman game, some tests were performed to smaller units e.g. a simple TCP socket for sending and reading requests, etc. so that to be sure that specific concepts will work in the final implementation.

The implementation of the program follows the layered architecture as presented by the instructor in the course material. It is an MVC architecture (top-down program flow i.e. from View to Model via Controller), which aims to make clearer the structure of the program and divide the duties of each part (client and server) properly. More specifically, three packages were used to represent the client, the server and the common methods (Figure 1). The first two were furthermore divided in an MVC style.

Closely following the reference '[here](#)', the model is the central component of the MVC pattern. It expresses the application's behavior in terms of the problem domain, independent of the user interface. It directly manages the data, logic and rules of the application. The view can be any output representation of information. The third part or section of MVC design, the controller, accepts input and converts it to commands for the model or view.

For the client package, four different packages were used in the given order of flow: a)

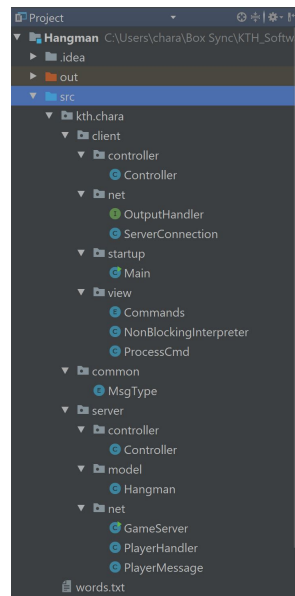


Figure 1: The architecture of the Hangman game

the "startup", with a Main class, just to run several clients for the game. It is the first step for client program flow, b) the "view", with NonBlockingInterpreter and ProcessCmd classes, and Commands enum. The NonBlockingInterpreter class corresponds to a multithread process which aims to give a responsive UI to the client. The ProcessCmd class is used to define client's message type and message body. The Commands enum is just used to define a collection of constant commands of the user, c) the "controller", with just the Controller class. It is used as an intermediate step that connects the view package with the net package. It handles client's connection with the server, starting a new game, making a new guess and disconnecting. The first three duties are handled asynchronously, d) the "net", with the ServerConnection class and the OutputHandler interface. The first one is used to actually connect with a server's socket at a specific host and port. This class also sends to the server the messages of starting a new game, making a guess and disconnecting from it. It also introduces a multithread method for listening of new responses from the server, which is handled by the OutputHandler interface. With this interface, we skip the need of going from net package to the view package through the controller package.

For the server package, three different packages were used in the given order of flow: a) the "net", with a GameServer, PlayerHandler, and PlayerMessage classes. The first one is used to open a listening socket for new clients. It is a blocking socket as it waits for client's connection. Each client is handled in a separate thread to reduce response time and improve performance. The PlayerHandler class implements the multithread process for each client. It also sends requests to the controller package for handling

client's requests. The `PlayerMessage` class is used just to extract the message type and message body of the client's commands, b) the "controller", with just the `Controller` class, which as happened with the corresponding client's class, it works as an intermediate step for sending the requests from the `GameServer` class to the `Hangman` class in the final package, which is c) the "model". In this package, the actual game algorithm is introduced.

For the common package, there is only the `MsgType` enum, which corresponds to all the possible message types for the client and server.

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. Also, appropriate system messages were introduced as a way for following up the program flow e.g. server's console to print when new player connects, starts a new game, makes a new guess, disconnects.

The final code of the Java application can be found [here](#).

4 Result

Let us consider each requirement for this assignment and how this was met. Firstly, we have that the client and server must communicate by sending messages over a TCP connection using blocking TCP sockets. The `java.net.Socket` class was used to represent a socket (line 7 in `GameServer` class in server's net package, and line 10 in `ServerConnection` class in client's net package), and the `java.net.ServerSocket` class was used at the server package (line 6 in `GameServer` class) to provide a mechanism for the server program to listen for clients and establish connections with them.

To establish a TCP connection using sockets the following steps occurred: a) The server instantiates a `ServerSocket` object, denoting which port number communication is to occur on (line 37 in `GameServer` class), b) The server invokes the `accept()` method of the `ServerSocket` class (line 40 in `GameServer` class). This method waits until a client connects to the server on the given port (Figure 2). The appropriate linger time and socket timeout were set for handling the clients (line 56, 57 in `GameServer` class), c) After the server is waiting, a client instantiates a `Socket` object (Figure 3), specifying the server name and the port number to connect to (line 27, 28 in `ServerConnection` class). The appropriate connection and socket timeout were set (line 28, 29 in `ServerConnection` class), d) The constructor of the `Socket` class attempts to connect the client to the specified server and the port number. If communication is established (Figure 4), the client now has a `Socket` object capable of communicating with the server, e) On the server side, the `accept()` method returns a reference to a new socket on the server that is connected to the client's socket (line 40, 41 in `GameServer` class).

After the connections are established, communication can occur using I/O streams. Each socket has both an `OutputStream` and an `InputStream`. The client's `OutputStream`

(line 31 in ServerConnection class) is connected to the server's InputStream (line 32 in PlayerHandler class in server's net package), and the client's InputStream (line 32 in ServerConnection class) is connected to the server's OutputStream (line 33 in PlayerHandler class).

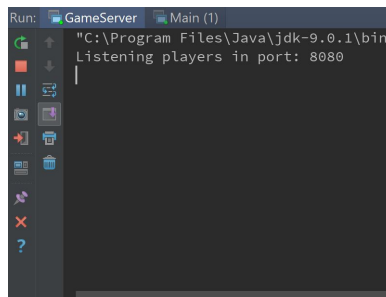


Figure 2: The server has opened a TCP socket and waits for new clients

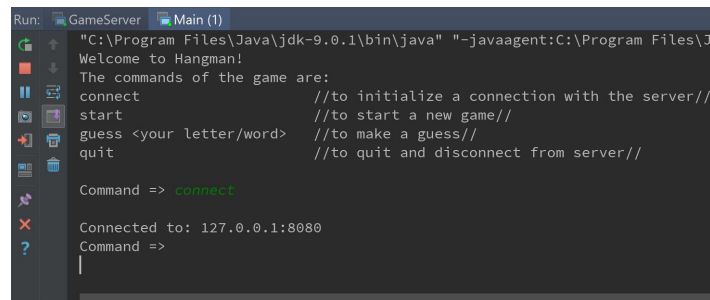


Figure 3: A new client has connected to a specified host and port

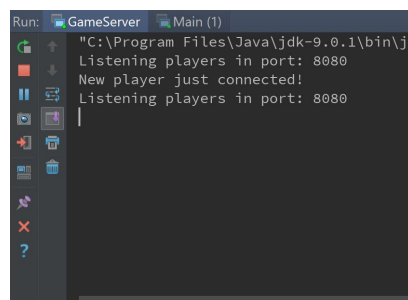


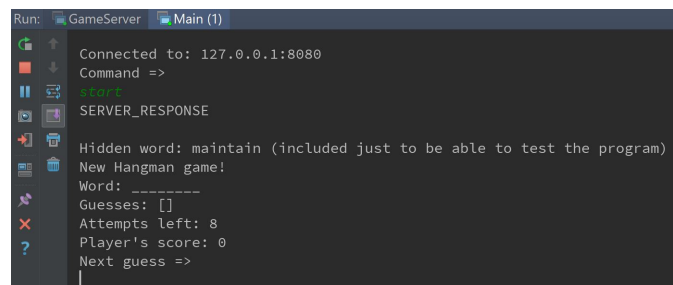
Figure 4: A new client has established connection with the server

Secondly, we have that the client must not store any data. This requirement was

fulfilled by just providing a specific message type (connect, start, guess, quit) and body (only for the guess command) from the client to the server (Figure 5), and wait for the corresponding response from the server (Figure 6). As mentioned before, the message of the client corresponds to an output stream that is send to the server, and the server responds with an input stream after doing all the processing of data. All the messages of the server are printed to client's console using the `OutputHandler` interface (in client's net package). This interface is used in a multithread way in `ServerConnection` class (line 59 in client's net package). The interface is implemented in view package (line 79) so that to print all the messages in client's console. Using this kind of interface we avoid the need of moving from the net package to the view package of the client through the controller.

```
switch (processCmd.getCommand()){
    case CONNECT:
        //Connect to localhost at port 8080
        controller.connect(host:"127.0.0.1", port:8080, new ServerResponse());
        break;
    case START:
        controller.startHangman();
        break;
    case GUESS:
        controller.makeGuess(processCmd.getMsgBody());
        break;
    case QUIT:
        newInput = false;
        controller.disconnect();
        break;
}
```

Figure 5: All possible commands that are send to the server with an output stream

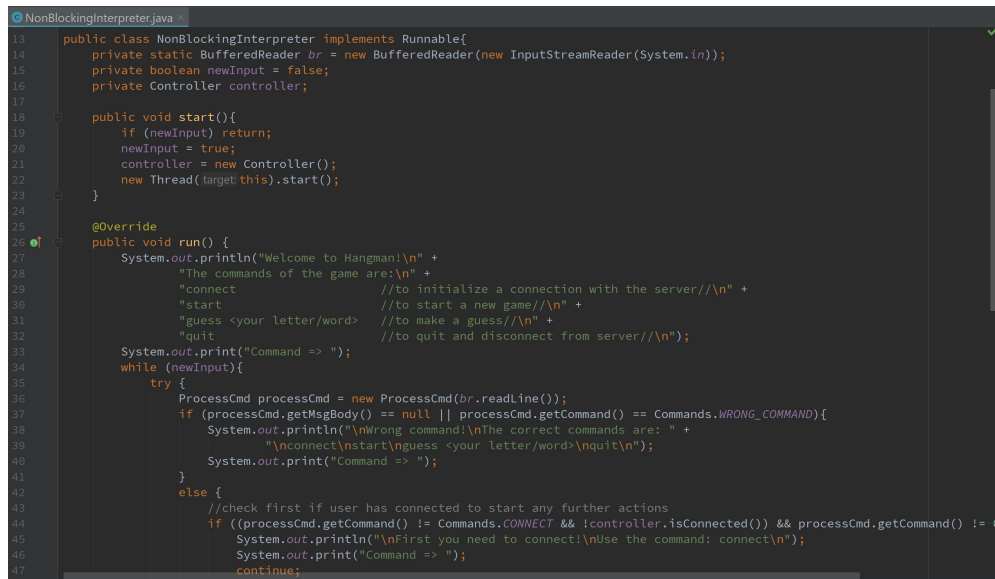


```
Run: GameServer Main (1)
Connected to: 127.0.0.1:8080
Command =>
start
SERVER_RESPONSE
Hidden word: maintain (included just to be able to test the program)
New Hangman game!
Word: -----
Guesses: []
Attempts left: 8
Player's score: 0
Next guess =>
```

Figure 6: The client doesn't store any data. Just send a command and wait for a server response

Thirdly, we have that the client must have a responsive user interface, which means it must be multithreaded. The threading was handled by client's `NonBlockingInterpreter` class (line 13) using the `Runnable` method. The class reads and interprets client's commands. The command interpreter will run in a separate thread, which is started by calling the 'start' method (line 22). Commands are executed in a thread pool, a new prompt will be displayed as soon as a command is submitted to the pool, without waiting

for command execution to complete (Figure 7).



```

13 public class NonBlockingInterpreter implements Runnable{
14     private static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
15     private boolean newInput = false;
16     private Controller controller;
17
18     public void start(){
19         if (newInput) return;
20         newInput = true;
21         controller = new Controller();
22         new Thread(target this).start();
23     }
24
25     @Override
26     public void run() {
27         System.out.println("Welcome to Hangman!\n" +
28             "The commands of the game are:\n" +
29             "connect //to initialize a connection with the server//\n" +
30             "start //to start a new game//\n" +
31             "guess <your letter/word> //to make a guess//\n" +
32             "quit //to quit and disconnect from server//\n");
33         System.out.print("Command => ");
34         while (newInput){
35             try {
36                 ProcessCmd processCmd = new ProcessCmd(br.readLine());
37                 if (processCmd.getMsgBody() == null || processCmd.getCommand() == Commands.WRONG_COMMAND){
38                     System.out.println("\nWrong command!\nThe correct commands are: " +
39                         "\nconnect\nstart\nguess <your letter/word>\nquit\n");
40                     System.out.print("Command => ");
41                 }
42                 else {
43                     //check first if user has connected to start any further actions
44                     if ((processCmd.getCommand() != Commands.CONNECT && !controller.isConnected()) && processCmd.getCommand() != Commands.QUIT){
45                         System.out.println("\nFirst you need to connect!\nUse the command: connect\n");
46                         System.out.print("Command => ");
47                     }
48                     continue;
49                 }
50             } catch (IOException e) {
51                 e.printStackTrace();
52             }
53         }
54     }
55 }

```

Figure 7: The user interface is multithreaded by implementing the Runnable method in NonBlockingInterpreter class

Lastly, the final requirement is that the server must be able to handle multiple clients playing concurrently, which means it must be multithreaded (Figure 8). In GameServer class in server's net package, a new thread is created each time a new client has connected (line 62). This thread starts the PlayerHandler class, again in server's net package, which implements a Runnable method (line 13). When a new client wants to play a game, a new instance of the Hangman class is created by giving the order to the server's controller through the PlayerHandler constructor (line 25). Then, the server's controller class calls a Hangman object for a new Hangman instance (line 10).

Several comments are included in the Java code found on [GitHub](#), discussing about essential parts of the program. As for the layered architecture used, information were provided at the 'Method' section 3.

5 Discussion

The main goals of the assignment were to learn about TCP sockets and 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 establishing client-server communication through blocking TCP sockets, avoiding client storing data, giving

```

Run: GameServer Main (1) Main (1)
"C:\Program Files\Java\jdk-9.0.1\bin\jav
Listening players in port: 8080
New player just connected!
Listening players in port: 8080
New game!
New guess from a player!
New guess from a player!
New guess from a player!

Run: GameServer Main (1) Main (1)
"C:\Program Files\Java\jdk-9.0.1\bin\java" "-jav
Welcome to Hangman!
The commands of the game are:
connect //to initialize a con
start //to start a new game//
guess <your letter/word> //to make a guess//
quit //to quit and disconn
Command => connect
Connected to: 127.0.0.1:8080
Command =>
guess p
SERVER_RESPONSE
Hidden word: predominate (included just to be ab
New Hangman game!
Word: p-----
Guesses: [p]
Attempts left: 11
Player's score: 0
Next guess =>
guess a
SERVER_RESPONSE
Word: p-----
Guesses: [p]
Attempts left: 11
Player's score: 0
Next guess =>

Run: GameServer Main (1) Main (1)
"C:\Program Files\Java\jdk-9.0.1\bin\java" "-javaagent:
Welcome to Hangman!
The commands of the game are:
connect //to initialize a connection
start //to start a new game//
guess <your letter/word> //to make a guess//
quit //to quit and disconnect from
Command => connect
Connected to: 127.0.0.1:8080
Command =>
guess a
SERVER_RESPONSE
Hidden word: postgraduate (included just to be able to
New Hangman game!
Word: a-----
Guesses: [a]
Attempts left: 12
Player's score: 0
Next guess =>
guess p
SERVER_RESPONSE
Word: a-----
Guesses: [a]
Attempts left: 12
Player's score: 0
Next guess =>

```

Figure 8: The server handles multiple clients playing concurrently at the same time

responsive UI to client and the server to handle multiple players at the same time. 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 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 Course

The course of Network Programming is very well organized, with all the necessary information provided in the course material. The instructor is very active at the canvas discussion section, giving sufficient feedback to students' questions. The time spent on this first assignment was approximately three days (with several breaks in between).