

Final Project

Network Programming, ID1212

Vasileios Charalampidis, vascha@kth.se

January 03, 2018

1 Introduction

The purpose of this final project is to apply the knowledge of previous homeworks, in a context that was not previously covered in any of them.

For that purpose, a Java application was deployed that implements the case of the classic 'Rock-Paper-Scissors' game (rock wins scissors, paper wins rock, scissors wins paper; more info [here](#)), with at least 3 players playing at the same time, and a server dealing with the communication and the results of the game. More specifically, each player connects to the server in a way that will be explained later on. When 3 players connect to the server, the last sends a notification to the connected players that the game is ready to start. Then, each player sends a play move (rock or paper or scissors), and the server notifies back each player that a move was played by the specific player that made the move. When all players send their moves, the server defines the points for each player depending on their move (if n the total number of players, award m players ($n - m$) points each if they choose the same gesture and beat the other ($n - m$) players). The players will then be asked to continue or not the game. If at least 1 player quits, the game is finished, and the other players that are still connected wait for other players to connect and achieve a total number of at least 3 players in a game.

The deployed program meets some specific requirements, so as to present an acceptable solution for this project. The first requirement is that there should be a well defined communication paradigm between clients and server. The paradigm must be used throughout the project. The second requirement is that each client must run exactly the same program and there may not be any "master" clients. The server will handle all the calculations and the communication between the clients. The third requirement is that the program must allow at least 3 clients to participate in the game. The fourth requirement is that the clients 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 another client. The last requirement is that the user interface must be informative. This means that the current state of the program must be clear to the user, and the user must understand what to do next.

2 Literature Study

In order to accomplish this project, the materials used from previous homeworks (especially the ones from first assignment) were taken into consideration. The context of the program was taken from task 2, illustrated in the first homework of the course. It was very important at the beginning to realize the requirements and decide about the communication paradigm that needed to be implemented to achieve a higher grade for the project.

The communication protocol used for the project was the RabbitMQ. It is an open source message broker software (sometimes called message-oriented middleware) that originally implemented the Advanced Message Queuing Protocol (AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), MQTT, and other protocols. The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover. Client libraries to interface with the broker are available for all major programming languages ([ref. here](#)).

What became clear from the materials used, was that RabbitMQ can be easily implemented for the purposes of 'Rock-Paper-Scissors' game and it can give very reliable results for a distributed 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 'RPS' game, some tests were performed to smaller units e.g. send a simple message from client to server and vice versa with messaging queues, distribute the same message from a single server to many clients' queues etc. so that to be sure that specific concepts will work in the final implementation.

4 Result

4.1 The MVC architecture

The architecture used for this assignment can be seen in Figure 1. It consists of 3 main packages: the client, the server and the common. The client package contains the following packages:

- The 'startup' package, which includes the 'StartPlayer' class. It aims to begin adding new players at the game. The 'start' method from 'PlayerUI' class in 'view' package is called ([line 11](#)).

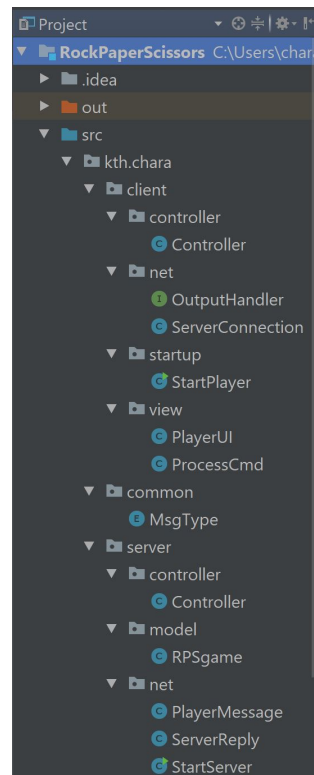


Figure 1: The MVC architecture of the RPS game

- The 'view' package, which includes: a) the 'PlayerUI' class, which corresponds to the user interface that the player runs. It implements a 'Runnable' method for responsive UI (line 15), b) the 'ProcessCmd' class, which is used to analyze the message of the player and export the type (line 58) and body of it (line 62).
- The 'controller' package, which includes the 'Controller' class. It aims to connect the 'view' package with the 'net' package, by introducing asynchronously tasks with the 'CompletableFuture' for starting a game (line 19) and playing a new move (line 34). The class also disconnects a player from the game (line 47) and returns if a player is actually connected to the server or not (line 55).
- The 'net' package, which includes: a) the 'ServerConnection' class, which is used to receive messages from the server in a specific queue (line 55) and send this message to client's console (line 56); and also to send messages to the server's messaging queue (line 31), by declaring a specific queue name (line 30), b) the 'OutputHandler' interface, which is used to connect the net package directly with the 'PlayerUI' class in 'view' package for displaying messages from server.

The server package contains the following packages:

- The 'net' package, which includes: a) the 'StartServer' class, which is used to start the receiving messaging queue of the server ([line 25](#)). When a new message arrives, it is sent to 'ServerReply' class ([line 29](#)), b) the 'ServerReply' class, which is used to process the players message depending on the message type ([line 36](#)), and send a reply to all the registered with the server clients ([line 29](#)), c) the 'PlayerMessage' class, which is used to define and return the id ([line 50](#)), type ([line 42](#)) and body of the client's message ([line 46](#)).
- The 'controller' package, which includes the 'Controller' class. It aims at connecting the 'net' package with the 'model' one. It includes methods for starting a new game ([line 14](#)), playing a new move ([line 23](#)), exiting a game ([line 31](#)) and getting the results of the game ([line 39](#)).
- The 'model' package, which includes the 'RPSgame' class with all the game handling. When a new player registers to the server, its id is saved at a list with 'Player' type ([line 25](#)). When all the players respond with a new move ([line 50](#)), the server compares the move with the one of the other players and assigns accordingly the points of the current round and increases the total points of the corresponding player ([line 98](#)). If a player exits the game, then the list with all the players is cleared to be reassigned later in case of a new game ([line 70](#)).

As for the common package, it contains just the 'MsgType' class, which is used both from 'client' and 'server' package, as enumerator for possible client commands i.e. START ([line 9](#)), for starting a new game, PLAY ([line 11](#)), for playing a new move, QUIT ([line 13](#)), for quitting the game and the server queue, and the WRONG_COMMAND ([line 15](#)), for anything else that was mistyped.

4.2 The communication paradigm

As it was mentioned earlier, the RabbitMQ was used to send messages between the clients and the server, and vice versa. More specifically, what was tried to achieve was that many different clients can send messages to the same server messaging queue ([Figure 2](#)), while the server can send the same message to all connected clients in their unique messaging queue ([Figure 3](#)). For the last case, the server first declares an 'EXCHANGE_NAME' that the clients need to know in order to bind their queues with the server.



Figure 2: A client (P) sends a message to the server's (C) queue, [reference](#)

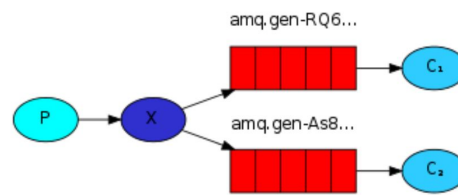


Figure 3: The server (P) sends the same message to all bind client (C1, C2...) queues, [reference](#)

In the game implementation, the RabbitMQ was used: a) in the client package, in the 'ServerConnection' class for sending messages through a specific channel ([line 28](#)), while establishing a new connection ([line 27](#)); and for receiving new messages from the server by binding to the specific 'EXCHANGE_NAME' mentioned earlier ([line 49](#)), b) in the server package, in the 'StartServer' class for receiving new messages from the clients by declaring a channel with a specific queue name ([line 20](#)); in the 'ServerReply' class for sending replies to all clients by publishing to a specific 'EXCHANGE_NAME' ([line 29](#)). The results of the RabbitMQ paradigm in the case of messages exchanged between server and clients can be seen in Figure 4.

```

StartServer StartPlayer StartPlayer StartPlayer
"C:\Program Files\Java\jdk-9.0.1\bin\java" "-javaagent:C:\Program File
Server is up and running!
Waiting for messages...
New game!
[x] Sent 'New player connected!'
[x] Received '58 START'
New game!
[x] Sent 'New player connected!'
[x] Received '90 START'
New game!
[x] Sent 'Game started! Play a move!'
[x] Received '277 START'
New move from a player!
[x] Sent 'Player #58 has made a move!'
[x] Received '58 PLAY r'
New move from a player!
[x] Sent 'Player #90 has made a move!'
[x] Received '90 PLAY s'
New move from a player!
[x] Sent '
Player #58-> Played: r, RoundScore: 2, TotalScore: 2
Player #90-> Played: s, RoundScore: 0, TotalScore: 0
Player #277-> Played: s, RoundScore: 0, TotalScore: 0
For a new round just play a new move. Otherwise type quit to exit.
[x] Received '277 PLAY s'

```

Figure 4: The messages between server and clients as obtained from the server package for a single round

4.3 Game requirements

Considering now each requirement separately:

- Each client must run exactly the same program, there may not be any “master” clients. In the Java program, all the clients run the same code without any exceptions. The only difference between different clients is that each client has each own messaging queue (line 48) for receiving messages from the server. In Figure 5 it can be easily observed that all clients receive the same messages depending their registration order with the server.

```

StartServer  StartPlayer  StartPlayer  StartPlayer
"C:\Program Files\Java\jdk-9.0.1\bin\java" "-javaagent:C:\Program
Welcome to Rock-Paper-Scissors game - Player# 90 !
The commands of the game are:
start //to start a new game//
play r/p/s //to play a move, where 'r' for rock, 'p' for paper, 's' for scissors
quit //to quit the game//
id#90>
Connected to server! The game will begin soon!
id#90> New player connected!
id#90> New player connected!
id#90> Game started! Play a move!
id#90>
Player #90 has made a move!
id#90> Player #39 has made a move!
id#90>
Player #90-> Played: r, RoundScore: 2, TotalScore: 2
Player #39-> Played: s, RoundScore: 0, TotalScore: 0
Player #151-> Played: s, RoundScore: 0, TotalScore: 0
For a new round just play a new move. Otherwise type quit to exit.
id#90>

StartServer  StartPlayer  StartPlayer  StartPlayer
"C:\Program Files\Java\jdk-9.0.1\bin\java" "-javaagent:C:\Program
Welcome to Rock-Paper-Scissors game - Player# 39 !
The commands of the game are:
start //to start a new game//
play r/p/s //to play a move, where 'r' for rock, 'p' for paper, 's' for scissors
quit //to quit the game//
id#39>
Connected to server! The game will begin soon!
id#39> New player connected!
id#39> Game started! Play a move!
id#39> Player #90 has made a move!
id#39>
Player #39 has made a move!
id#39>
Player #90-> Played: r, RoundScore: 2, TotalScore: 2
Player #39-> Played: s, RoundScore: 0, TotalScore: 0
Player #151-> Played: s, RoundScore: 0, TotalScore: 0
For a new round just play a new move. Otherwise type quit to exit.
id#39>

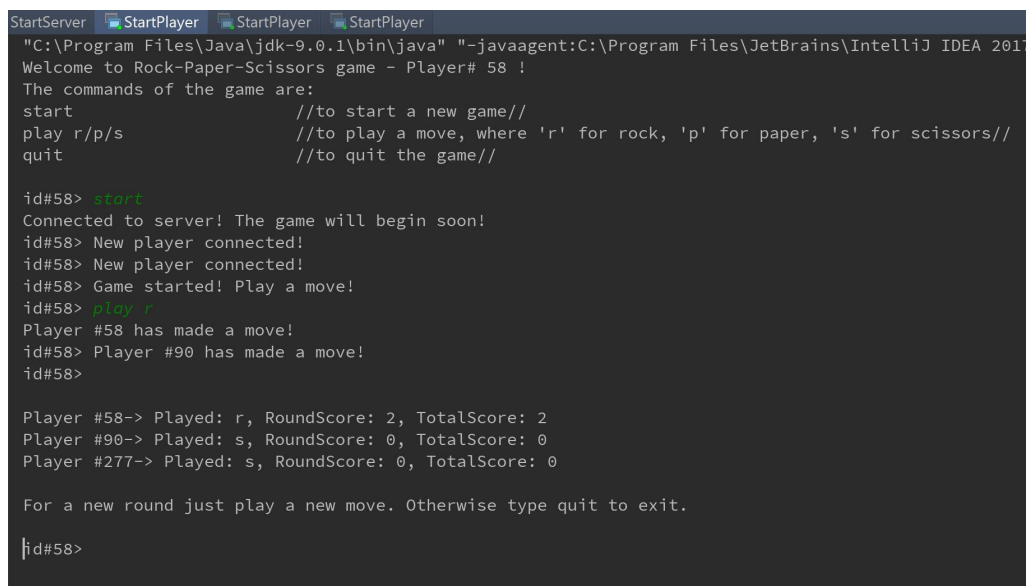
StartServer  StartPlayer  StartPlayer  StartPlayer
"C:\Program Files\Java\jdk-9.0.1\bin\java" "-javaagent:C:\Program
Welcome to Rock-Paper-Scissors game - Player# 151 !
The commands of the game are:
start //to start a new game//
play r/p/s //to play a move, where 'r' for rock, 'p' for paper, 's' for scissors
quit //to quit the game//
id#151>
Connected to server! The game will begin soon!
id#151> Game started! Play a move!
id#151> Player #90 has made a move!
id#151> Player #39 has made a move!
id#151>
Player #90-> Played: r, RoundScore: 2, TotalScore: 2
Player #39-> Played: s, RoundScore: 0, TotalScore: 0
Player #151-> Played: s, RoundScore: 0, TotalScore: 0
For a new round just play a new move. Otherwise type quit to exit.
id#151>

```

Figure 5: All clients receive the same messages

- The program must allow at least three clients to participate in the game. The minimum number of clients that can play the game is defined in the server’s model package, in ‘RPSgame’ class (line 11). This number can be easily changed according to the needs.
- The client must have a responsive user interface, which means it must be multi-threaded. The threading was handled by client’s ‘PlayerUI’ class (line 15) 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 27). 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.
- The user interface must be informative. This is clear by considering Figure 6. The player starts a new game by typing ‘start’. If he types something not expected, then a warning message will be displayed (Figure 7).

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



```

StartServer StartPlayer StartPlayer StartPlayer
"C:\Program Files\Java\jdk-9.0.1\bin\java" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2017.2\lib\idea_rt.jar=5000:C:\Program Files\Java\jdk-9.0.1\bin" -Dfile.encoding=UTF-8
Welcome to Rock-Paper-Scissors game - Player# 58 !
The commands of the game are:
start //to start a new game//
play r/p/s //to play a move, where 'r' for rock, 'p' for paper, 's' for scissors//
quit //to quit the game//

id#58> start
Connected to server! The game will begin soon!
id#58> New player connected!
id#58> New player connected!
id#58> Game started! Play a move!
id#58> play r
Player #58 has made a move!
id#58> Player #90 has made a move!
id#58>

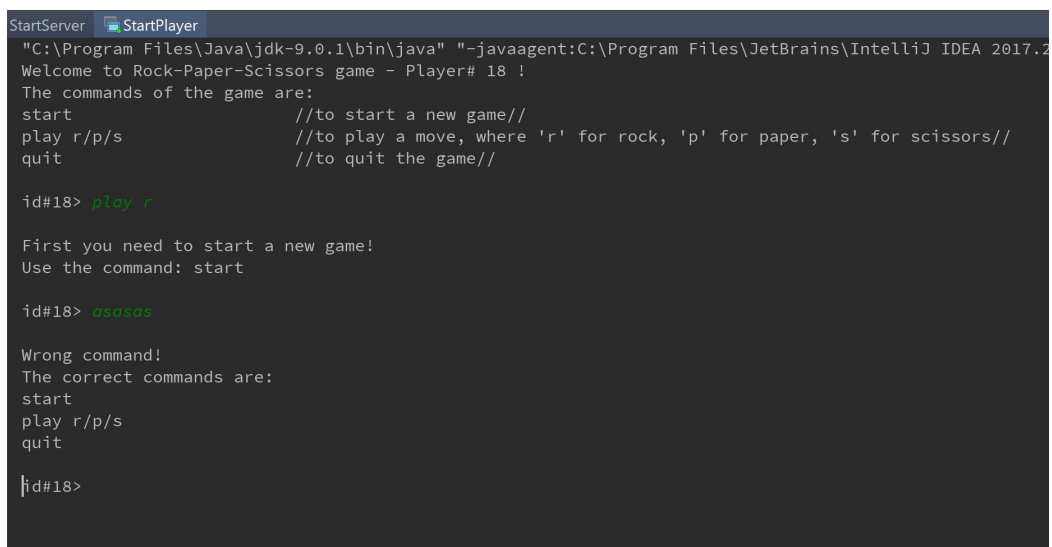
Player #58-> Played: r, RoundScore: 2, TotalScore: 2
Player #90-> Played: s, RoundScore: 0, TotalScore: 0
Player #277-> Played: s, RoundScore: 0, TotalScore: 0

For a new round just play a new move. Otherwise type quit to exit.

id#58>

```

Figure 6: The UI of a client



```

StartServer StartPlayer
"C:\Program Files\Java\jdk-9.0.1\bin\java" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2017.2\lib\idea_rt.jar=5000:C:\Program Files\Java\jdk-9.0.1\bin" -Dfile.encoding=UTF-8
Welcome to Rock-Paper-Scissors game - Player# 18 !
The commands of the game are:
start //to start a new game//
play r/p/s //to play a move, where 'r' for rock, 'p' for paper, 's' for scissors//
quit //to quit the game//

id#18> play r

First you need to start a new game!
Use the command: start

id#18> asasas

Wrong command!
The correct commands are:
start
play r/p/s
quit

id#18>

```

Figure 7: The warning message when the client types something not expected by the system

5 Discussion

The main goal of the project was to implement the knowledge acquired by previous assignments, while introducing a new communication paradigm.

The requirements of the presented code can be summarized in introducing a well defined communication protocol, clients running the same program, participating clients must be at least 3, having a responsive UI for the clients, and having an informative UI for the corresponding user of the program. 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 to introduce some unit tests (e.g. [JUnit](#)), so that to guaranty that everything performs fine without any issues.