GOI ESKOLA POLITEKNIKOA
ESCUELA POLITÉCNICA SUPERIOR
MONDRAGON UNIBERTSITATEA

MILESTONE 1 – OPERATING SYSTEMS

LOREDI ALTZIBAR

NAHIA LI GÓMARA

AITOR LANDA

XABIER LANDA

ANDER OLASO

ANE SAJERAS

POPBL5 – TEAM 4

DEGREE OF COMPUTER

ENGINEERING Tutor: GOIURIA

SAGARDUI

Arrasate, January, 22nd 2020

CONTENT

1. INTRODUCTION

The goal of this document is to expound the concurrency problems that we have found during the POPBL. Firstly, it is compulsory to explain that the POPBL is about developing a multi-games web platform, where the user will have the opportunity to choose among different game-rooms and different game types. On that scenario, it is supposed that there is going to be more than one player connected at the same time. For that reason, it is a need to protect some variables from been accessed at the current time by many users in order to preserve the proper or expected functioning.

A concurrency issue is given when two or more application threads are sharing a common resource and they want to access it at the same time. A concurrent application must manage this issue in order to handle the common resource access.

Regarding the concurrency issues which have been detected in the application, there are three different main issues that have been solved. Although, there are more issues that can be solved with the same solution, for that issues there haven't been developed any solution because the solution proposed for the main issues can be adapted to the other issues.

2. CONCURRENCY PROBLEMS

2.1 FIRST PROBLEM

The first concurrency problem that we have found is the changes of the score table in the game. This table is changed by the players at the end of each question, the players will add their score to the table. Despite of the changes made by the players, there are also viewers that can view the score table. So, taking into account both type of users, we can see that the concurrency problem will be at the time when too many players want to change their score, and when a viewer is wanting to see the score table while a player is editing it.

If this problem is not solved, it might happen that the score can be changed aleatory or many times by different players, and in that way the viewer could not see the correct and updated scores.

2.2 SECOND PROBLEM

In the second concurrency problem we have found that when a player enters or exits in the lobby it is important to know maximum how many persons can be inside the lobby, and how much players there are. This last variable change when a player enters or exits a lobby. So, taking into account both type of users, we can see that the concurrency problem will be at the time when too many players want to enter and change the total and maybe in the same time viewer wants to enter into the same lobby.

If this problem is not solved, it might can happen that the total users can be wrong because if lot of people modify the same variable without exclusion we don't know exactly what is the value of the variable and we could not see the correct number of people in the room.

2.3 THIRD PROBLEM

The players usually do requests to the web sockets to communicate between them or with other objects. For that reason, it is important to protect the WebSocket, for example, if there are two players chatting in the lobby, and both users send a message at the same time, it can produce an error o an exception because two users are trying to use the same WebSocket. In order to manage this concurrency issue, it has been decided to control the user flow that tries to access the WebSocket communication.

3. SOLUTIONS

We know that the server uses a *ThreadPool* to manage all the threads which are running on the server. However, we do not know exactly how this *ThreadPool* performs. To develop the solutions explained below, we have imagined that there is a thread per user, it means, a thread is created when a user gets into the web page.

3.1 FIRST PROBLEM

To solve the first concurrency problem described before, we have though in implementing the readers and writer's problem with it. In this case, we have agreed to implement this solution in two ways, synchronization by monitors, and synchronization by semaphores. The main variable we want to protect is a HashMap of Players and their score. There are also counters that will be protected in order to avoid unnecessary changes or changes by errors.

On the one hand, at the synchronization by monitors, we have used the external monitors, it means, the monitor which are implemented by the Lock Java class. Furthermore, the implementation of the monitor is similar at the one we have done in class and we have Condition objects. In this case, we have given priority thinking in the players, because they are the ones who will change the table to be updated, in order to have the most updated table for the viewer, we have given to them the priority.

On the other hand, at the synchronization by semaphores, we have used the Semaphore Java Class. This class has a similar notation to the one we have seen in class. This time, we have given priority to the reader because we thought that the viewers must have available to the score table the most time as possible, and we have prioritized it over the fact of having the table updated.

To sum up, we have solve the same problem in two different ways, prioritizing writers and prioritizing readers, but if we must to choose which to implement in our web pages, we think that the best one would be prioritizing the players (writers). As we have explained before, prioritizing players gives to the players a faster way to interact with the web, and to improve their experience, and it gives to the viewers a more updated information on the table.

Even though we only have identified a problem, we have planned to identify more issues during the project. Nowadays, we are implementing new functionalities and implementing WebSocket's, so, while we are progressing in the project, we will add more concurrency issues, which will be implemented by implicit monitor, and Runnable interfaces.

3.2 SECOND SOLUTION

As it is said on the previous section about second problem, it might happens that two or more players want to get into a lobby at the same time, so to avoid the unneeded changes into this variable, it will be protected with a monitor.

Focusing on the monitors, a *int* has been protected, it will be the shared resource, to protect that variable, it is use *synchronized blocks,* despite of the previous solution it will

not be the entire method. Only, the necessary code will be protected. Protecting only a piece of code allows to lock only the variable that is wanted to use.

3.3 THIRD SOLUTION

To provide a solution to this third problem, the solution has been developed considering the players that will try to access into the *WebSocket* at the same time. The solution will be implemented by means of monitors.

Regarding the monitor that has been used to develop this solution, it is an implicit monitor, it means that it is used *synchronized* method. This method type creates like a mutex at the beginning and at the end of the method. The method only will be used by a thread, while a thread is using this method, other threads can not access to it.

4. DEMOSTRATIONS

4.1 DEMO 1 – SCORE TABLE (MONITORS)

Implementing the demo, we have divided it into 5 classes: *User, Player, Viewer, Game, GameController. User* class is an object that defines the attributes that every user who login in the website will have. *Player* and *Viewer* are two objects that extends of *User, Player* has a variable called *score* this will be the variable that is going to be copied into the *scoreTable*.

ScoreTable is a *HashMap* which is part of *GameController* class, this class will manage the monitor methods, it has the external monitor and the conditions. *Game* is the main class, it contains *main* method, here is initiated the players and viewers lists and the threads.

4.2 DEMO 2 – SCORE TABLE (SEMAPHORES)

Implementing the demo, we have divided it into 4 classes: Player, Viewer, Lobby, GameController. Player and Viewer are two objects, Player has a variable called score this will be the variable which is going to be copied in the “tablaPuntuaciones”.

“tablaPuntuaciones” is a is a HashMap, it means that is a structure where the key is the player and the other variable is an Integer with the total score of that Player. Lobby class is the main class where we have the creation of the threads, players and viewers. GameController is a class that has the control of the game and controls the common resources using semaphores, because we have the HashMap, where players need to write and viewers to see.

Then we have a variable to count the number of viewers that are using this structure and reading, because we prioritize the viewers in this case.

4.3 DEMO 3 – LOBBY COUNTER

Implementing the demo, we have divided it into 4 classes: User, Player, Game and GameController. User class is an object which define the attributes that every user who login in the website will have. Player is an object which extends of User.

GameController will manage the monitor methods, it has the implicit monitor, specifically synchronized block, in this class we have managed the 3 variables to control the number of players, numbers of viewers and the number of total users. Game is the main class, it contains main method, here is initiated the players and viewers lists and the threads.

4.4 DEMO 4 – WEBSOCKET USE

Implementing the demo, there are 3 classes: *Player, WebSocket, WebSocketMain*. As it is said before, this demo includes a *synchronized* method. This method will protect the call to the websocket, so, it only will be possible to call to the websocket one by one.

Player class will define the object with the players attributes, in addition, it will have the synchronized method which simulates the *websockets*. *WebSocket* class extends of *Runnable,* so, it will be part of all the threads as *run()* method. There is going to call to the *WebSocket.* Finally, the *WebSocketMain* class contains the main, and players initialization.