University of Liège

Project (part 2) : Battleship

Introduction to computer networking

Maxime Meurisse (20161278)
Valentin Vermeylen (20162864)

3e year of Bachelor Civil Engineer

Academic year 2018-2019

1 Software architecture

This project consists of several files :

- `battleship.css` : file containing the style of the different web pages;
- `battleship.js` : file containing the javascript of the project. He takes care of sending the `GET` in `AJAX`;
- `Entity.java` : an entity is an element of the game grid (water or ship);
- `GameConstants.java` : contains all the game constants. The game can be completely modified with this file, without causing any error;
- `GameManager.java` : this class allows you to create a new game and play it (attack a position, see the status of the game, ...);
- `Grid.java` : this class implements the game grid and the operations that can be done;
- `HTMLHandler.java` : this class is used to generate and compress the `HTML` code of each web page;
- `HTTPException.java` : this class is used to raise custom exceptions, especially `HTTP` exceptions;
- `HTTPHandler.java` : this class is used to read `HTTP` requests;
- `ImageHandler.java` : this class is used to compress the images of the game in base 64 and retrieve them on the web pages;
- `ServerWorker.java` : a worker can handle user requests on a game;
- `WebServer` : main class of the project. This is the server that creates the workers;
- some `png` files.

1.1 `battleship.js`

This file contains the javascript code of the project. This one consists mainly of a `hitPos` function. When the user clicks on a position (and javascript is enabled), this function is called with the position as argument. An `AJAX` request containing this position is then sent to the server. The server response is read and the web page is updated.

1.2 `Entity.java`, `Grid.java` and `GameManager.java`

These classes, already used in project 1, are the implementation of the game itself. They could work independently of all classes concerning the server.

When a player wishes to play a game, he must instantiate a new `GameManager`. A game is therefore represented by a `GameManager`.

1.3 `HTTPHandler.java`

This class is used to read an `HTTP` request (`parse` method) and save the content (request type, header information, content). Methods are used to retrieve the saved content.

If the request is incorrect, an `HTTPException` is thrown.

1.4 `ServerWorker.java`

A `ServerWorker` is instantiated for each request. When a worker is instantiated, he is responsible for a game.

A `ServerWorker`, thanks to all other classes, will display the web pages, create / delete a new game (if the game is over), interpret the user's requests and update the game status and ranking.

A `ServerWorker` is a thread, so all of its actions are in the `run` method. In order to facilitate its readability, some repetitive action has been created in the form of methods.

1.5 `WebServer.java`

This is the main class of the project. She is responsible for creating a new `ServerWorker` for each request. She also takes care, thanks to the methods `addFame` and `getFame`, to maintain a ranking of the best scores and to allow the display.

2 Multi-thread coordination

2.1 The workers

In this project, a new thread (`ServerWorker`) is created for each new `HTTP` request (`GET` or `POST`). The request is executed and the thread dies.

A thread takes care of a single request concerning a single game. It is therefore not possible to create several threads acting on the same game at the same time, since the `HTTP` requests are executed sequentially. Even if the game is open in several tabs, the requests will be executed one after the other, the state of the game will therefore remain consistent at all times.

2.2 The game list

The list of games running on the server are saved in an array. Each created thread can access this array to add or remove a game. As this array is shared for all threads, it is essential that the piece of code handling it is `synchronized` to ensure consistency.

2.3 Hall of fame

In addition, the list of results of each winning party (the *hall of fame*) is also saved in an array on the server. The method to add a score (`addFame`) is also `synchronized` to prevent multiple threads at the same time adds scores (may lead to inconsistency or loss of information).

Since games are independent and do not parish any data, no method, variable or piece of code requires synchronization mechanisms.

3 Limits

3.1 Robustness

Methods for updating the game state based on user input (the function to attack a position for example) are robust. Indeed, we can not assume that the user will systematically enter a valid position. We must anticipate a potential bad entry so as not to crash the server.

In general, the other methods are also robust, that is, they prevent bad input from crashing the server.

3.2 Data backup

Regarding data backup, the program does not have the right to manipulate files, it is therefore impossible for him to permanently save the game data (if the server turns off, all scores are lost).

3.3 Loading of web page

Regarding the loading of the web page, we noticed that when we used images a bit too large (squares of 250 * 250 for the grid of game for example), the page took time to be displayed (the grid s displayed square by square). With this in mind, it seems unthinkable, with the techniques used, to display on the web pages very high quality images (4K for example).

4 Possible improvements

In general, the game is quite user-friendly : it is easy to play (just click on a position, or choose from a drop-down list), the game information are displayed (the grid in real time , the number of trials and ships remaining) and the interface is rather pleasant (however not yet worthy of the major American studios).

However, the user's name is generated randomly. This is not very convenient for the user : he must remember his username if he wants to see his achievements in the hall of fame, and he has no idea who are the other players.

To improve this, we could ask the user to choose a custom pseudonym at the beginning of each game.

Still concerning the users, these are identified by a cookie which has a lifespan of 10 minutes. When this cookie dies, it is impossible for the user to recover his game session, and therefore to improve his score. In addition, the scores themselves are saved in a variable on the server. This means that if the server shuts down, all scores are lost.

To overcome this, we could imagine working with a database (or at least files on the server) : all users and scores would be recorded. Thus, a user would only have to log in with his credentials in order to retrieve all his game data, and the scores would never be lost.

In this perspective of a database, we could display more advanced statistics for each player: its number of games played, wins, losses, hours of play, ...