# Computer networking :

# Report - $2^{nd}$ project

---

Julien Bolland - Thomas MAZUR
s161622 - s162939

Prof G. Leduc

Academic Year 2018-2019

## 1 Software Architecture

In order to find a solution, we have implemented the following classes.

### 1.a WebServer.java

This is the main class of the project. It opens the connection on port 8018 and waits for clients to connect. We used pools so that the server can't be overwhelmed by to much clients. When calling the class, we choose the number of clients that can connect to the server at the same time.

The webserver class take as an argument the maximum simultaneous requests it accepts. If the user doesn't specify this number it is set by default as 10.

### 1.b Worker.java

This class inherits from the class Thread, which means that we can use multi-threading on it. It is basically the door between the server and the client, because it handles the reponses of the server according to the requests of the client.

First,

— Prior to anything, the worker checks if any cookie has expired and handles it accordingly

— The worker sets a timeout on the socket which means the connection will be dumped if the client takes too much time to sent his request.

— The request is received and returned as a string thanks to BattleshipReceiver class

— The string is parsed by HttpHandler class

— A second HttpHandler is instantiated in order to initialize a response

After that, the worker is split into two part, using the getMethod() method of HttpHandler, we can determine the method of the request (either "GET" or "POST")

#### GET method

When we know the method used is "GET", the pseudo-code is as followed :

```

        if Path is empty then
            redirectTo(/play.html)

        else if Path is /play.html then
            if Game exists and query is valid then
                checkAttempt(query)
                respond(jsonfile)
            else if Game exists and !query then
                pageActualisation()
            else if Game !exists and !query then
                createNewGame()
            else
                respond(error 400) /*Bad request*/

        else if Path is /halloffame.html then
            respond(HallOfFame page)

        else
            respond(error 404) /*Not found*/
```

explanation : First, we look at the path we want to reach. If the path is "/", we redirect the client by setting the response to the error 303 and specifying "/play.html". Then, if the path is "play.html", we can either :

1. Check the attempt requested by the client and respond with a json file. Normally, the client will send request using the script sent by the server. This is handled by the action.js file.
2. Send a response containing all the html code that has to be computed by the client (*i.e.* the browser), so that our main page is displayed. The entire page is generated using the BattleshipHTML class and sent with BattleshipEmitter.

If the request page is halloffame.html, then we only send a response containing the HTML code of the page. It is generated thanks to BattleshipHTML and the Hall of Fame is stored in a Ranking object (see later for further explanations).

#### POST method

In that case, it means that JavaScript is disabled. Therefore, we can't click on a tile to hit it. We handle this by creating a form, where the user can choose the line and the column to hit. Then he or she clicks on submit and the entire page is reloaded with the corresponding effect on the selected tile.

The pseudo-code of this part is similar to the "GET" method but it shouldn't be possible to create a new game, actualize one or access /halloffame.html. Thus these are managed as an error 400.

### 1.c BattleshipReceiver.java

This class, given a socket, reads a http request on the input stream of the socket and returns it as a String. It can only read not chunked nor compressed data.

### 1.d BattleshipEmitter.java

This class, given a socket and an HTTPHandler object, will send the HTTP message on the output stream of the socket and, if specified into the "Transfer-Encoding" header, can send the body as chunks of data according to the HTTP/1.1 standards.

### 1.e HttpHandler.java

This class uses constructor polymorphism. If an object is instantiated without argument, a dummy **response** is created set to match an "ordinary" response for this project ( 2OO OK, Content-Type : text/html ; utf-8, ...). If a String is given, it will parse it like an http message (request or response) breaking it into wanted pieces, *i.e.* the headers fields and the body. Then it instantiates the URL that will be used to access our pages.

The class also contains methods to modify the message and ultimately a method to get the message as a byte array, with the first element being a byte array containing the headers and state line and a second element being another byte array containing the body, according to the HTTP standard.

### 1.f BattleshipHTML.java

This class handles the generation of the HTML code. With a certain game going on, we can retrieve all the states of the tiles and then display the corresponding image (hit, splash or not hit yet). Images are handled with the class BattleshipImages, which encodes them with Base64.

### 1.g Game.java

This class implements the behaviour of a game corresponding to a certain cookie. It keeps track of the battlefield and the state of the tiles of this game. It is useful to check an attempt made by the player, without the worker knowing the game states (and in particular the tiles that have not yet been touched). A worker can know if the current game is lost or not only by asking this class.

### 1.h Ranking.java

This class is responsible for the memorization of the Hall of Fame. We chose to use an ArrayList containing HallOfFame objects, defined by a cookie and a score. When we instantiate a Ranking object, it automatically updates the ranking if it has to. The usage of this object is shared among the workers, so that we had to ensure the correctness of data thanks to synchronization.

### 1.i BattleshipException.java

This class allow us to display the messages of the corresponding HTTP error code on the browser. When this exception is thrown, we use a method that writes the HTML code to be displayed and we send this page to the browser on the input stream of the server. This class covers the codes 400, 404, 411, 501 and 505. Note that codes 200 and 303 are not handled by this class, as it only handles errors.

## 2 Multi-thread coordination

As already said, we used pools in WebServer in order to limit the number of connection on the server.

In addition to that, we have two variables that can be shared among the workers : the list of games and the ranking necessary for the Hall of Fame. To avoid several workers to access them and read or write on them, we enclosed the sensitive part in a `synchronized` close. Like that, each worker must quire the lock on these variable before being allowed to access them.

## 3 Limits

It is sure that if we allow to much clients to connect to the server, it can only crash, even with the pool mechanism. For example, it couldn't handle efficiently DOS attacks. If a user instantiates 1000000 games to a pool that allows 100 connections, then with our

100 milliseconds of socketTimeOut, the server would take :

$$\frac{1000000}{100} \times 0.1 = 1000 \text{ seconds} = 16,6 \text{ minutes}$$

to close all the connections. A new user will then have to wait this amount of time to play. Therefore, our web application is not well suited against big attacks.

## 4 Possible Improvements

Instead of simply displaying a new image when selecting a tile, we could also display which type of ship we hit. Moreover, adding a game status and the current score could be great, because it could help the user to know what's left to hit and how far it is to the end of the game.

In order to be more user-friendly, despite the fact that the style of the page could be remade, the images could be interactive, *i.e.* when we hit them, an animation is displayed.

As for technical improvements, the weight of the sent page could be lessen. When the page is generated, each tile is assigned the entire data of the picture it contains. For example, when a new game is created, we send a hundred times the Base64 representation of the water picture. The problem is that this image is really heavy compared to the rest of the page.

A possible solution would be to send just one copy of each picture. Then, we would use a script to assign each tile its correct picture (but it wouldn't work for a POST method). Another solution would be to implement a get instruction to provide an image to the client but this one is outside the scope of this assignment.