University of Liège

Project (part 1) : Battleship

Introduction to computer networking

Maxime Meurisse (20161278)

3e year of Bachelor Civil Engineer

Academic year 2018-2019

1 Software architecture

The problem was to make a simplified version of the *Battleship* game between a client and a server using Java sockets.

In order to solve this problem, I took advantage of the Java object-oriented paradigm and created several classes:

- `BattleshipClient` : this class implemented the client side of the game;
- `BattleshipServer` : this class implemented the server side of the game;
- `Entity` : this class represents an element of the game grid (a ship or an empty slot);
- `GameManager` : this class takes care of starting a game and managing it (shooting a position and displaying the status of the game);
- `Grid` : this class implements the game grid;
- `ServerWorker` : this class deals with managing the game of a player on the server side.

The details of each class and their methods are clearly explained in the code. In the remainder of this report, only the major points that helped manage the requests will be discussed.

1.1 `BattleshipClient`

This class interacts directly with the client via the terminal.

First, it sends a request for a new game on port `2278`. When it receives the response from the server, if it is positive (*i.e.* if the server did not reject the connection), a menu with several choices is presented to the user:

1. **Try a tile** : the client chooses a position in the grid to attack. This one, if it is valid, is sent to the server via a request (`1 1 position`). When the server response has been received, a message indicating which ship has been hit is displayed;
2. **See game status** : the client sends a request (`1 2`) to the server to obtain the list of the positions he has already attacked and the ship touched (or not) at each position. When the server response is received, each position-ship (or not) pair is displayed;
3. **Quit** : End the game (but does not stop the server thread).

1.2 `BattleshipServer`

The server listens on port `2278`. With each new connection received, it starts a new game on a new thread.

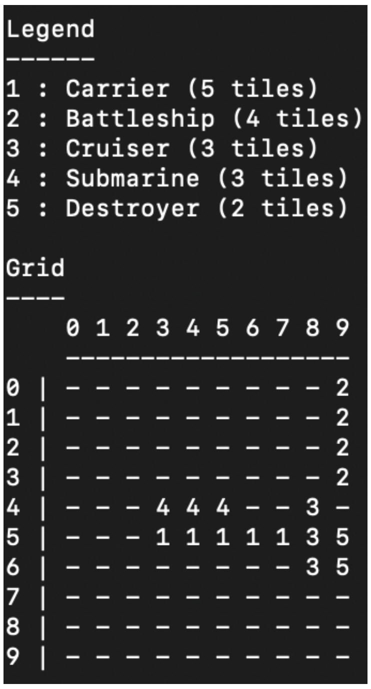
1.3 GameManager

This class implements the game itself. It creates a new grid and initializing it. It has methods to attack a position, display the status of the game and get the list of positions already attacked.

When the server starts a new game on a new thread, this class is instantiated.

1.4 Grid

This class represents a game grid. The grid is a one-dimensional array with 100 elements (because in this case, the size of the grid is 10x10). As stated in the client, the coordinates of the grid are represented as shown in figure 1.



```
Legend
------
1 : Carrier (5 tiles)
2 : Battleship (4 tiles)
3 : Cruiser (3 tiles)
4 : Submarine (3 tiles)
5 : Destroyer (2 tiles)

Grid
----
    0 1 2 3 4 5 6 7 8 9
    -------------------
0 | - - - - - - - - - 2
1 | - - - - - - - - - 2
2 | - - - - - - - - - 2
3 | - - - - - - - - - 2
4 | - - - 4 4 4 - - 3 -
5 | - - - 1 1 1 1 1 3 5
6 | - - - - - - - - 3 5
7 | - - - - - - - - - -
8 | - - - - - - - - - -
9 | - - - - - - - - - -
```

Figure 1 – Example of a game grid, with the conventions for coordinates.

The grid deals mainly with placing the different ships and keeping their states up to date.

1.5 ServerWorker

This class is responsible for managing the server-side game. It is an extension of the `Thread` class. It is waiting for a request from the client :

- **Request a new game** : when the client sends a request for a new game, the server instantiates the `GameManager` class and sends the client that the game has started;
- **Request a shoot at a given position** : when the client sends a request to attack a position, the server reads the position, attacks this position using a `GameManager`'s method and sends back to the client the identifier of the affected ship (0 for no ship, greater than 0 for a ship) returned by the `GameManager`;

- **Request the game status** : when the client sends a request to obtain the status of the game, the server retrieves the list of position-ship pairs using a `GameManager` method. This list is converted to byte and sent to the client.

In all other cases where the client's request does not match one of the previous ones, a request error (`1 4`) is sent.

2 Multi-thread coordination

With each new connection on the server, the `BattleshipServer` class creates a new `Thread` (a new game, *i.e.* a new `GameManager`). When the game is over, the `Thread` dies (during a timeout or an exception, because the player can not send a request to indicate that it ends the game).

In this case, no synchronization is required between threads. Indeed, each client plays its own part, so the threads do not share any object or segment of memory.

3 Limits

With the current protocol, it is not possible to know when the client has finished his game (if he wins, loses or prematurely leaves the game). So we have to catch an `IOException` to leave the current thread. It would be better to have a request dedicated to the one in the protocol so that the server can leave the game cleanly.

In terms of robustness, checks are made in the client and in the server to verify properly that the requests are valid (or not). There are also local checks to prevent the client from entering a position that is not in the grid. An invalid request from the client is processed by the server, and vice versa. The client-server communication is thus a priori robust.

However, the other classes of the game are not very robust : no verification of the validity of the parameters of the methods are performed. In theory, it is not possible that these parameters are false since all the client's inputs are checked and all the game data (size of the ships, grid size, ...) are implemented statically. It would be better, however, to implement custom exceptions in a more general context.

4 Possible improvements

Regarding the protocol, we could add other types of messages to have better communication between the client and the server. For example, a message that allows the client to indicate that they have finished a game so that the client stops the thread properly (or starts a new game).

One could also have several error messages (instead of just having 1 4) to specifically tell the client what kind of error occurred.

As part of this project, I made the choice of a very simple implementation so as not to clutter the code and keep only the essential.

However, improvements could be made to make the game more *user-friendly* : the game grid could be displayed to the client (only with the positions he attacked) instead of just the positions in text and colors could also be used to represent ships on the grid. It could also be suggested to the client to restart a game when it finishes one (instead of simply stopping the execution of the client).

In a more general context, a graphical interface could even be developed.