Faculdade de Engenharia
da Universidade do Porto

Relatório

Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática e Computação

Batalha Naval

Afonso Jorge Moreira Maia Ramos up201506239@fe.up.pt
Edgar Filipe Amorim Gomes Carneiro up201503784@fe.up.pt
Rúben José Da Silva Torres up201405612@fe.up.pt
Vítor Emanuel Fernandes Magalhães up201503447@fe.up.pt

Abril 2018

Conteúdo

1 Introdução

No âmbito da unidade curricular de Sistemas Distribuídos, foi proposta, pelo grupo, a implementação de um jogo de Batalha Naval, utilizando recursos lecionados nas aulas teóricas. Esta versão do jogo contém algumas funcionalidades diferentes da Batalha Naval original, criando assim uma versão aplicada aos dias de hoje.

O projeto foi implementado como uma aplicação distribuída do tipo cliente-servidor com notificações, onde as comunicações são feitas com recurso ao serviço web REST.

Um jogador, ao conectar-se ao servidor do jogo, é adicionado ao mapa, juntando-se aos outros jogadores. O objetivo é afundar o barco dos outros jogadores, indicando as coordenadas que deseja atacar. Caso o jogador acerte no barco de um outro jogador, esse jogador irá perder o jogo visto ter o seu barco afundado. Ao contrário da versão original de Batalha Naval, esta não contém turnos entre jogadores, fazendo antes uso de intervalos de tempo no qual os jogadores poderão jogar, sendo este intervalo, neste caso, de quatro segundos entre cada disparo.

Tal como nos, recentemente, famosos jogos *.io*, como, por exemplo, o *agar.io*, não existe um vencedor. Porém, existe, simplesmente, um jogador que pode ser considerado o jogador mais forte, visto ter conseguido afundar uma maior quantidade de barcos.

Este relatório será estruturado da seguinte forma:

1. **Introdução** - Apresentação sucinta da aplicação;
2. **Arquitetura** - Descrição da Arquitetura e componentes principais;
3. **Implementação** - Detalhes da Implementação;
4. **Aspetos Relevantes** - Aspetos importantes dentro da aplicação;
5. **Conclusão** - Conclusões retiradas.

2 Arquitetura

A aplicação foi estruturada em camadas(packages de *java*), cada uma com as suas funcionalidades. Assim, é garantido um isolamento de cada módulo, mas sem comprometer a comunicação entre as camadas. Foram criadas camadas de Comunicação, Interface de Utilizador, Jogador, Lógica de Jogo, Mensagens, Servidor e Segurança, assim como Utilitários usados pelo programa.

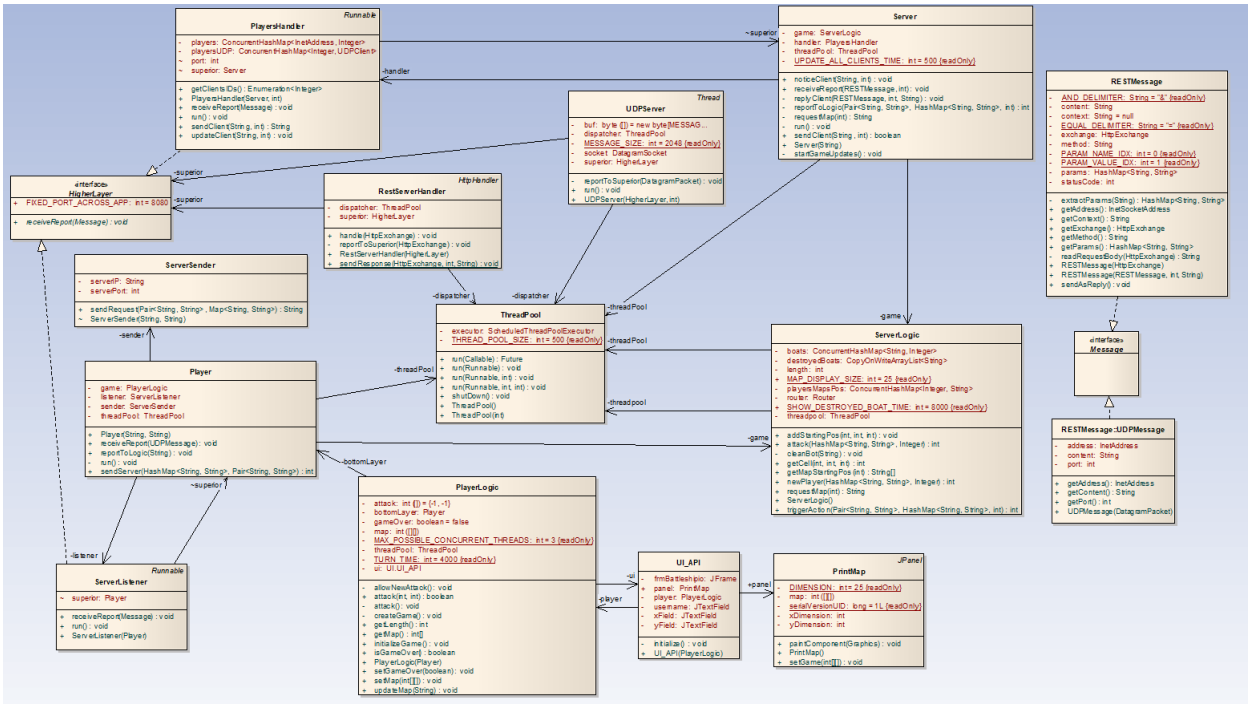

Figura 1: Diagrama de classes da aplicação

2.1 Servidor HTTPS

De maneira a aceder às ações indicadas por URL, foi criado um HashMap de rotas, cuja *Key* é um par que contém o caminho(URL) e o Método HTTP, e o *Value* é a ação em si. O par é obtido através da mensagem REST correspondente.

```
1 HashMap<Pair<String , String>, String> routes = new HashMap<>();
```

O grupo optou por esta abordagem para melhor se aproximar daquilo que são as aplicações que usam REST em hoje em dia. De facto, se analisarmos, por exemplo, a

framework de desenvolvimento *web* - *Laravel* -, podemos verificar que esta terá sempre uma classe chamada *route/web.php* que se encontra responsável por desencadear, recebendo uma rota e um método, uma ação pertencente a um controlador. Como por exemplo:

```
1 Route::get('tags', 'TagsController@showAllTags');
2 Route::post('messages/{id}/vote', 'MessageController@vote');
```

A seguinte lista identifica as rotas que o servidor recebe, assim como as ações correspondentes:

- POST /create - Cria um novo jogador, adicionando-o ao jogo;
- POST /attack - Ataque de um jogador, com as coordenadas como parâmetros;
- POST /move - Movimento de um jogador.

Para cada pedido HTTP, o servidor responde com um dos seguintes códigos:

- **200**, em caso de sucesso;
- **203**, em caso de sucesso, mas com uma conotação negativa associada ao resultado da ação do jogador;
- **404**, caso o caminho esteja incorreto;
- **500**, caso o servidor não consiga aceder à ação pedida.

De destacar que o grupo optou por usar o método POST em todos os casos por julgar ser o mais adequado, visto que cada uma das ações em cima referidas vão acrescentar algo de novo às estruturas referentes à lógica de jogo.

2.2 Comunicação

Para o desenvolvimento da aplicação foram utilizados dois protocolos de comunicação: o protocolo HTTP e o protocolo UDP.

O protocolo HTTP é usada na comunicação iniciada pelo Cliente com o Servidor. Assim esta comunicação trata de realizar pedidos ao servidor, nomeadamente os pedidos especificados na secção 2.1. No desenvolvimento do projeto, o grupo tentou desenhar uma aplicação que fosse o mais modular possível e que funcionasse através de camadas. Assim, os ficheiros associadas ao envio e receção usando o protocolo HTTP encontram-se no *package* **Communication.REST**. No protocolo HTTP, visto o número de parametros poder variar, é usado um *HashMap¡String, String¿* no qual as *keys* representam o nome do parâmetro e o *value* o seu valor.

O protocolo UDP é usado para o envio constante e uniforme de mensagens por parte do Servidor aos Clientes, que de meio em meio segundo envia uma atualização de um mapa personalizado para cada cliente. O mapa será personalizado de forma a não mostrar ao cliente os outros possíveis barcos na redondeza bem como a só passar informação relativa à percentagem do mapa que o cliente consegue visualizar. Para o envio do mapa este necessita de ser descodificado e codificado sobre a forma de uma *string*. As funções relativas à implementação do protocolo UDP encontram-se presentes na *package* **Communication.UDP**.

Em ambos os protocolos, visto estes serem protoclos de baixo nível, as funções recetoras de mensagens reportam às camadas superiores a informação que lhes foi passada, como é o caso dos ficheiros *ServerListener.java* e *PlayersHandler.java*.

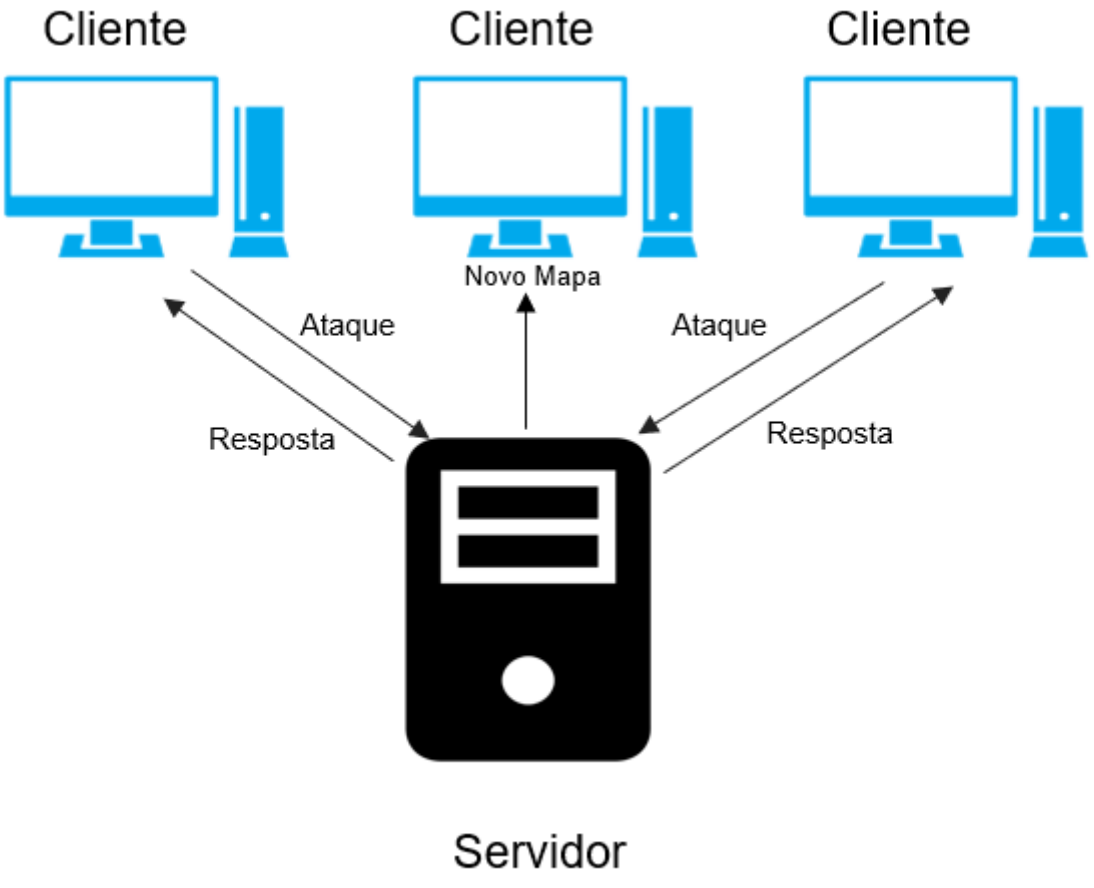

Figura 2: Exemplo de mensagens trocadas. O novo mapa é enviado para todos os jogadores

2.3 Interface de Utilizador

Foi também desenvolvida uma Interface de Utilizador para tornar a aplicação esteticamente mais apelativa para os jogadores.

Nesta UI, os jogadores conseguem alterar os seus *usernames* e a cor dos barcos, pressionando o botão **Edit Profile**. Ao pressionar o botão **Play**, serão conectados ao servidor, sendo-lhes apresentado o mapa do jogo mais recente.

Os botões na barra de navegação também permitem editar o perfil e jogar, assim como ver a opção **Help**, que permite obter algumas informações.

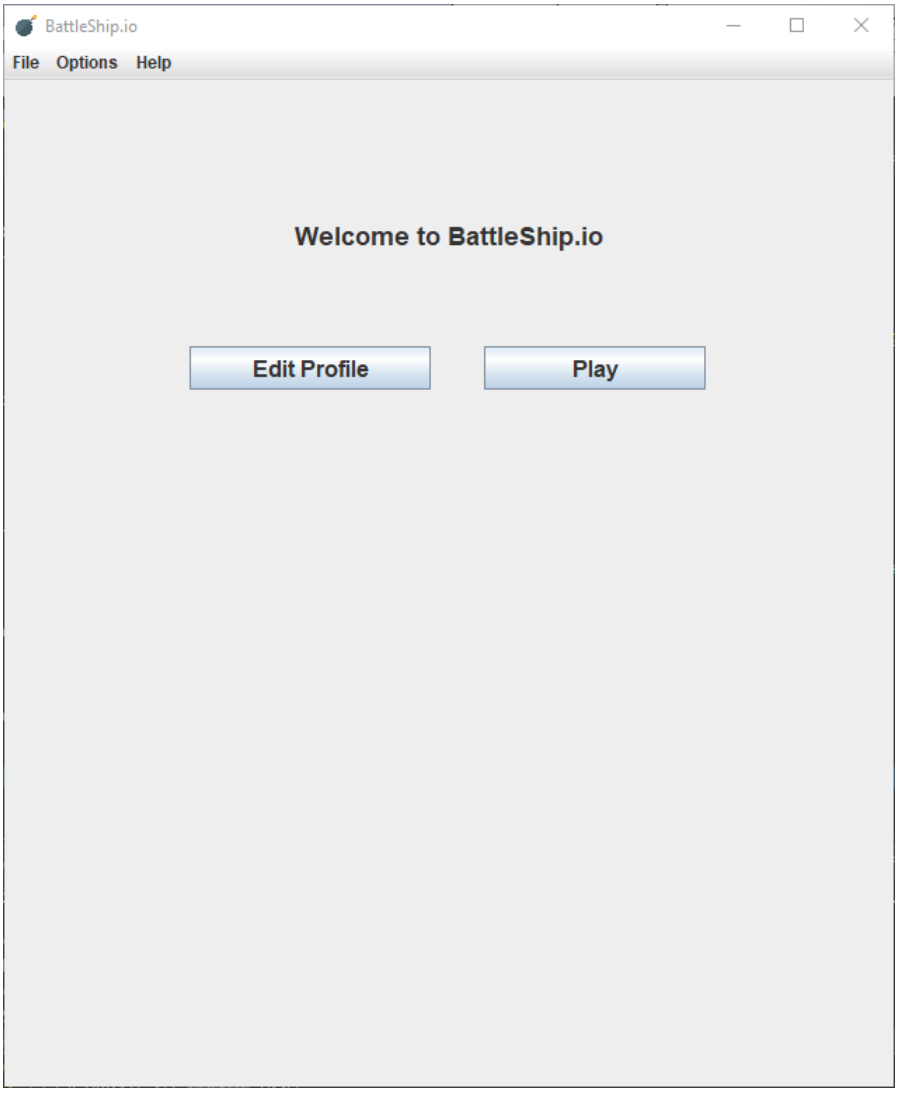

Figura 3: Menu no início do programa

Na interface referente ao jogo, um jogador pode visualizar o seu barco, bem como posições que foram atingidas por disparos. Estes disparos podem também ser de outros utilizadores, fazendo assim com que todos os utilizadores possam ter *feedback* visual sobre os disparos de todos os utilizadores. No entanto, este *feedback* (ou um barco em chamas, ou um tiro falhado) irão desaparecer ao fim de cerca de dez segundos.

Ao jogador é ainda possível escolher as coordenadas onde quer disparar, premindo depois o botão *Send*, que através de processos internos, enviará o pedido de ataque ao servidor.

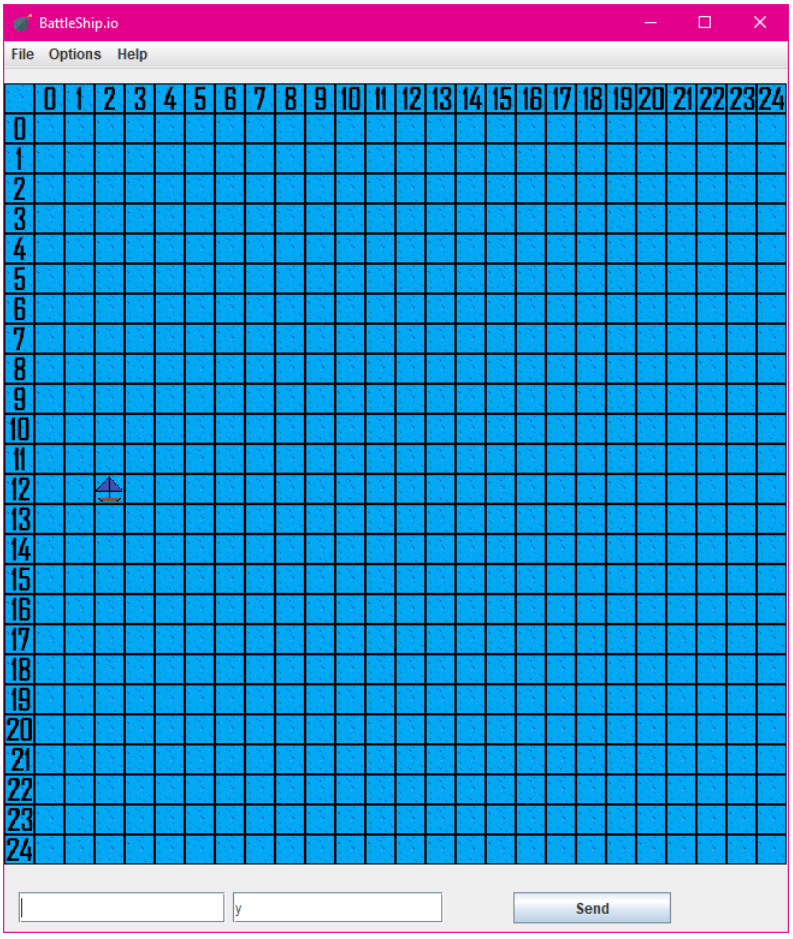

Figura 4: Imagem dentro do jogo

NOTA: Todos os *sprites* utilizados são originais.

3 Implementação

3.1 Detalhes Gerais

Para o servidor HTTP, foi usada a biblioteca Java **com.sun.net.httpserver**, pois é essencial para as respostas dadas aos pedidos dos clientes e para o sucesso das respostas.

O serviço REST foi implementado para a possível expansão da aplicação para browser.

A User Interface foi desenvolvida usando **Java Swing** devido à facilidade de implementação e à familiaridade com a biblioteca.

Também foi implementado Serialization, na camada **GameLogic** para guardar informações acerca dos barcos e o tamanho do mapa, no ficheiro *ServerLogic.java*.

A seguinte lista identifica as bibliotecas de Java utilizadas ao longo do projeto:

- **Reflect**, para obter métodos para as notificações REST;
- **ProcessBuilder**, para criar processos/*threads*;
- **Concurrent**, para resolver concorrência (explicado na secção seguinte).

3.2 Concorrência

Ao longo do desenvolvimento deste projeto, foram implementadas inúmeras técnicas com o objetivo final de garantir, ao máximo, a existência de concorrência, pois, sendo uma aplicação distribuída, é crucial a existência de várias ações em simultâneo.

De maneira a libertar o processo principal do peso de processamento inerente ao jogo, durante a execução deste são criadas novos processos, cada um com o seu objetivo. Para tal, recorreu-se, a *Threadpools*[1] para criar *threads* separadas e geri-las automaticamente. Estas *Threadpools* são utilizadas em várias situações, tanto do lado do servidor como do lado do cliente. No entanto, a criação das *threads* não permite a execução em intervalos de tempo definidos. Para evitar este problema, foram utilizadas *Scheduled Threads*, que permitem agendar a criação de novas threads sem que seja obrigatório reter recursos, bloqueando-os durante o tempo que se pretende esperar.

Do lado do Servidor, a *Threadpool* é utilizada para mandar as respostas aos pedidos dos clientes, com mensagens REST.[2] Também é usada para enviar mensagens UDP a cada cliente[3], para os manter atualizados com o mapa.

Do lado do Cliente, a Threadpool é usada para enviar pedidos HTTP ao servidor, que serão respondidos.[4].

É assim fácil de concluir que o uso de *threads* é realizado sempre que se represente conveniente, garantindo assim que seja possível uma aplicação extremamente concorrente.

Naturalmente, com uso de concorrência é também necessário o uso de estruturas que o suportem. Assim, o grupo fez um uso maioritário de duas estruturas que suportam concorrência: ***ConcurrentHashMap***, representado uma implementação concorrente de um *HashMap* e ***CopyOnWriteArrayList***, representado uma implementação concorrente de um *ArrayList*. O grupo não sentiu necessidade usar primitivas atómicas (com por exemplo o *AtomicLong*), pois estas primitivas eram sempre acedidas dentro de estruturas que já se garantiam concorrentes.

[1] Package Utils, ficheiro Threadpool.java
[2] Server.java, linha 54
[3] Server.java, linha 60
[4] Player.java, linha 39

4 Aspetos Relevantes

Durante o desenvolvimento do trabalho, foram desenvolvidos alguns tópicos que, devido à sua importância, merecem ser mencionados e explicados nesta secção.

4.1 Consistência

Ao longo do jogo, só existe um mapa, que é concorrente, pois é usado um *ConcurrentHashMap* para guardar a posição dos jogadores. Nesta estrutura, é guardado a informação que mapeie um utilizador ao seu barco. O grupo optou por esta abordagem ao ao invés de manter a totalidade do mapa pois julgou ser benéfico em termos de ocupação de memória. Usando um exemplo como suporte do argumento, no caso de uma mapa de tamanho mínimo de 25 células de altura por 25 células de largura, que só tenha um jogador, a abordagem adotada pelo grupo permite que só seja guarda uma entrada no *ConcurrentHashMap* ao invés de serem guardadas as 125 células do mapa. Esta opção também se apresenta benéfica em mapas de elevado tamanho, pois o mapa que será enviado ao cliente será sempre um mapa de 25 células por 25 células que é gerado personalizadamente para cada jogador.

Além disto, é enviado, a cada meio segundo, uma mensagem UDP a todos os jogadores, de modo a atualizar-lhes o mapa, na função *startGameUpdates*[5].

4.2 Escalabilidade

De maneira a assegurar escalabilidade, o mapa do jogo aumenta conforme o número de jogadores presentes, permitindo uma maior escalabilidade no jogo.[6].

Além disso, no Servidor, em vez de guardar um *Array* Bi-dimensional de inteiros para representar o mapa, é utilizado um *HashMap* a ligar a posição ao barco correspondente, pois os barcos só tem tamanho 1. Assim, impede-se o desperdício de memória e garante-se alguma concorrência[7], como já visto na secção **Consistência**.

[5] Server.java, linha 58
[6] ServerLogic.java, linhas 62 e 63
[7] ServerLogic.java, linha 19

4.3 Notificações com REST

Para as notificações com REST, foi implementado uma forma de *routing*, na camada **GameLogic**.

Neste *routing*, as funções/ações são invocadas quando recebe uma rota e método HTTP que conheça, utilizando *Java Reflection*.

Em arquiteturas web que implementem REST, este tipo de *routing* é bastante comum. Por exemplo, em Laravel[8], basta definir a rota e o método HTTP e a ação dentro do controlador definido. Nesta aplicação, a camada de Lógica de Jogo controla estes encaminhamentos.[9]

4.4 Segurança

Para garantir segurança no uso da aplicação, foram utilizados certificados e canais seguros por SSL.

Dentro da camada **SecurityAPI**, a função *generateCertificate* gera um certificado auto-assinado, caso não exista um dentro da pasta **Keystore**, com recurso à ferramenta *keytool*, já incluída no JDK.

As propriedades do sistema *-Djavax.net.ssl* não foram utilizadas[10], portanto foi necessário criar um SSLContext, com a função *getSSLContext*. Aqui, é carregado o certificado criado anteriormente e, com recurso ao *KeyManagerFactory* e *TrustManagerFactory*(objetos provenientes do Java), é criado o SSLContext.

Como foi mencionado anteriormente, para implementar o Servidor HTTPS, na camada **Communication.REST**, a biblioteca **com.sun.net.httpserver.HttpsServer** foi usada. Dentro do servidor, é necessário ativar o HTTPS Configurator, usando o SSLContext obtido anteriormente.

Para um cliente fazer um pedido, este precisa de um *SSLSocket*. Para isso, é necessário obter um SSLContext para criar o seu SocketFactory[11]. Ao iniciar a comunicação, são indicadas, ao socket, as *cipher suites* que podem ser utilizadas na conexão. No fim, é feito um SSL Handshake, porque é necessária a utilização de chaves encriptadas para a comunicação.[12]

[8]Laravel - The PHP Framework For Web Artisans

[9]Router.java

[10]As propriedades que não foram utilizadas são: keyStore, keyStorePassword, trustStore e trustStorePassword.

[11]HTTPRequest.java, linha 37

[12]HTTPRequest.java, linha 47

Se tudo correr conforme o esperado, a conexão pode prosseguir. Caso contrário, serão lançadas exceções no programa do utilizador.

4.5 Tolerância a falhas

Para aumentar a tolerância a falhas da aplicação, é guardado em memória virtual um mapeamento entre o *IP* de um jogador e o seu identificador no jogo. Esta solução representa-se ótimo para aumentar a robustez da aplicação, devido a dois fatores:

1. Permite aos utilizadores desconectarem-se em totalidade do jogo (como por exemplo caso a sua aplicação falhe ou a Internet vá a baixo), sendo que depois se conseguem reconectar e retomar o jogo, como o seu barco na mesma posição, claro está, se este não tiver sido destruído. Esta funcionalidade só é possível pois o mapeamento é mantido, não se perdendo a associação do identificador do barco ao *IP*.

2. A permanência dos mapeamentos *IP*-identificador não pode ser permanente. Como tal, foi implementado um gestor de conexões que tem como função averiguar se foi recebida ação por parte dos utilizadores que se encontra presente no mapeamento previamente referido. Assim, este gestor, verifica quais os jogadores que desde a sua última verificação não efetuaram qualquer jogada e remove estes do mapeamento, mas sem antes mandar um mapa que represente o fim de jogo para esses jogadores. Assim, com verificações que correm de 60 em 60 segundos, são analisados quais os jogadores inativos que devem perder o acesso ao jogo. A implementação desta funcionalidade é realizada no ficheiro **Server.ConnectionChecker.java**.

De destacar ainda, que foi ainda implementada a Serialização do Servidor e da sua lógica, tal como referido na secção 3.1 Detalhes Gerais. Esta implementação tem como objetivo corrigir possíveis colapsos do servidor, tornando possível que este reinicialize sem que perca a informação que tinha imediatamente antes do colapso. Para tal, existe uma *thread* que de 3 em 3 segundos guarda a informação do Servidor e sua lógica num ficheiro *Server.ser*.

Caso o Servidor não tenha recebido notificações de um jogador durante 60 segundos, este será removido do jogo, de modo a dar a possibilidade a outros jogadores, assim como reduzir o número de jogadores a gerir.

5 Conclusões

Com o desenvolvimento desta aplicação, conseguimos, de uma forma mais prática e mais intuitiva, aplicar os vários conceitos lecionados nesta unidade curricular, aprendendo a verdadeiramente criar um sistema distribuído, resistente a falhas, seguro, consistente e escalável.

Todavia, devido aos vários trabalhos que temos em mãos, não nos foi possível desenvolver a aplicação ao seu máximo, pelo que desejávamos ter acabado com um jogo de complexidade ligeiramente mais elevada. Contudo, pensamos que, o que neste projeto apresentamos, é uma boa representação do conhecimento por nós adquirido e ficámos orgulhosos do resultado final.

5.1 Melhoramentos

A movimentação dos barcos teve de ser descartada devido à falta de tempo. Esta ação tem as estruturas necessárias(como mencionado no relatório), mas foi decidido não lhe dar continuidade, em prol do melhoramento da estruturas de comunicação e a robustez do projeto.