Universidade do Minho

Licenciatura em Engenharia Informática

Sistemas Distribuídos

Trabalho Prático - Reserva de voos

Grupo 32

Ana Filipa Ribeiro Murta (A93284)
Ana Paula Oliveira Henriques (A93268)
Jose Miguel Dias Pereira (A89596)
Miguel Ângelo de Sousa Martins (A89584)

12 de janeiro de 2022

1 Introdução

O presente relatório procura apresentar o projeto desenvolvido no âmbito da Unidade Curricular de Sistemas Distribuídos, que suporta o desenvolvimento de uma plataforma de reserva de voos sob a forma de um par cliente-servidor em Java, utilizando *sockets* e *threads*.

Os utilizadores, após se autenticarem, podem reservar viagens constituídas por vários voos, sendo posteriormente informados quando a reserva for feita com sucesso, bem como cancelar uma reserva antes do fim do dia correspondente a essa viagem. Além disto, pode existir um tipo de utilizador especial, os administradores, que são capazes de inserir informação sobre voos, como a sua origem, o seu destino e a sua capacidade.

Assim sendo, a plataforma consiste num servidor que é capaz de comunicar com vários clientes concorrentemente, em que cada cliente comunica com o dito servidor através de sockets TCP. Para esse efeito, foi necessário aplicar diversos conceitos aprendidos nas aulas, nomeadamente o controlo de concorrência, variáveis de condição e a adoção da arquitetura cliente-servidor.

2 Arquitetura da Plataforma

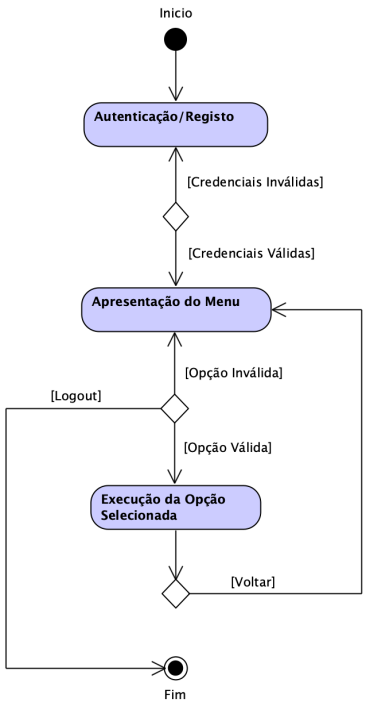

Figura 1: Arquitetura da Plataforma

2.1 Servidor

O servidor incializa um socket na porta `12343`, permitindo a troca de mensagens com o cliente, sendo este socket conhecido por todos os clientes. Após ser estabelecida a conexão, sempre que um cliente realizar um *request*, este será respondido pelo o servidor. Deste modo, ele irá aceder à classe *GestInfo*, que, por sua vez, acede aos métodos e aos dados armazenados em memória. Assim sendo, o servidor trata de satisfazer os pedidos feitos pelos users.

2.2 Cliente

O cliente conecta-se ao servidor através do socket criado com a porta `12343`. Todos os clientes têm associados a si uma conta. Como tal, após estabelecida esta conexão, é apresentado o menu principal ao cliente, onde o mesmo poderá fazer login, registar-se ou, então, abandonar o programa.

Se escolher uma das duas primeiras opções, encontrará um submenu com as várias funcionalidades a que terá acesso caso seja um administrador ou um usuário normal. Ou seja, este submenu será diferente na situação de ser um user normal ou um administrador. Uma das opções é, precisamente, o logout que efetuará, tal como o nome sugere, a saída da conta em que se encontra.

Cada uma destas opções, tanto do menu principal, como do submenu, leva à criação de uma thread de execução. No software, o cliente apenas envia pedidos e recebe as suas respostas.

2.3 Comunicação entre Cliente e Servidor

Para ser possível a comunicação entre servidor e o cliente, é necessário estabelecer uma conexão, estabelecida a partir de um socket, implementando duas classes responsáveis por encaminhar os dados no socket: `Demultiplexer` e `TaggedConnection`.

Cliente $\longrightarrow$ Servidor

Para poder efetuar uma funcionalidade da plataforma, o cliente precisa de introduzir os dados necessários para a sua execução. Estes dados serão enviados para o servidor através do método `send` criado na classe `Demultiplexer`, que, por sua vez, chama o método, de mesmo nome, da classe `TaggedConnection`.

De modo a que o servidor possa distinguir o tipo de mensagem enviada, os dados no socket são encaminhados juntamente com uma tag. Cada tag corresponde a uma funcionalidade diferente, tal como se pode observar na tabela apresentada a seguir:

Tag	Tipo de mensagem
1	Efetuação do login
2	Efetuação do registo
3	Efetuação da reserva de uma viagem
4	Efetuação do cancelamento de uma reserva
5	Consulta da lista de voos existentes
6	Inserção de informação sobre um voo
7	Encerramento de um dia
8	Consulta das reservas feitas pelo user atual
9	Execução da main

Para receber os dados, o servidor invoca o método receive da classe `TaggedConection`. Este devolve um objeto da classe `Frame`, contendo a tag e os dados enviados previamente pelo cliente.

Servidor ⟶ Cliente

Houve uma maior atenção na questão do cliente ser multi-thread uma vez que era necessário colocar o servidor a enviar, para as threads, os resultados obtidos com a execução das funcionalidades. Assim sendo, para enviar esses resultados, o servidor recorre ao método `send` da classe `TaggedConection`, que escreve no socket a tag e os dados.

Ao iniciar a execução, o método `start` da classe `Demultiplexer` é chamado, permitindo, assim, organizar as respostas, recebidas no cliente, num buffer que junta todos os dados recebidos (consoante a sua tag) numa fila de espera. Este método vai executar um thread responsável por interromper todas as mensagens enviadas pelo servidor para o socket. Deste modo, o método `receive` da classe `Demultiplexer` será invocado quando uma thread do cliente pretender receber os dados. Isto porque, este procura, na fila de espera anteriormente mencionada, a partir da tag que lhe é passada como argumento, os dados que lhe dizem respeito. Na situação desta fila se encontrar vazia, a thread fica à espera que cheguem os dados a partir do método await.

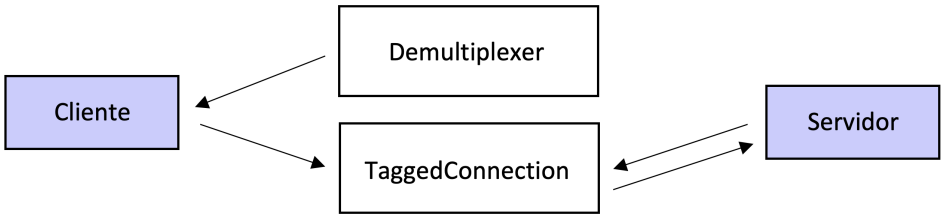

Figura 2: Comunicação entre Cliente e Servidor

3 Funcionalidades Implementadas

Neste capítulo, serão exploradas as funcionalidades implementadas na classe `GestInfo`, que, por sua vez, gera toda a informação necessária para permitir o funcionamento do sistema.

3.1 Funcionalidades Básicas

Os métodos **login**, **signup** e **logout** são responsáveis por autenticar/verificar as credenciais introduzidas, registar um novo utilizador e permitir a saída da sua conta, respetivamente. O utilizador é, no entanto, notificado se, aquando do signup, o username inserido não estiver disponível ou se, aquando do login, o username inserido não existir ou a password estiver incorreta.

O método **reserveFlight** permite registar um voo, devolvendo o código da reserva. Primeiramente, é feito uma escolha entre as duas datas propostas, com o método **pickDate**. Este opta pela data que não coincide com nenhum dia fechado: se ambas coincidirem, o utilizador será notificado; caso contrário, será devolvida a primeira data se esta não for um dia fechado ou, então, a segunda se a primeira o for e a segunda não. Posteriormente, é invocado o método **flightAvailable**, que testa se existe algum voo com o percurso pedido pelo utilizador Na situação desse percurso incluir etapas, como *Londres* $\longrightarrow$ *Bangkok* $\longrightarrow$ *Tóquio*, então é verificado se os voos *Londres* $\longrightarrow$ *Bangkok* e *Bangkok* $\longrightarrow$ *Tóquio* existem. Mesmo que existam, tem de ser verificado se a capacidade máxima desse(s) voo(s) ainda não foi atingida porque, se sim, o utilizador é notificado que o voo está lotado. Caso contrário, se a reserva for permitida, a capacidade do(s) voo(s) é decrementada. No entanto, se os voos não existirem, a reserva é cancelada.

O método **cancelReservation** executa o cancelamento de uma reserva, recebendo, para o efeito, o respetivo código de reserva. Como tal, se o dia de partida não esitver fechado, o cancelamento é bem sucedido e, consequentemente, a capacidade do voo com aquele percurso é incrementada. Na situação deste código não existir ou o dia de partida for fechado, o utilizador será informado e o processo interrompido.

O método **insertInfo** possibilita, a todos os administradores do sistema, a inserção de um voo, indicando a sua origem, destino e capacidade.

O método **closeDay** guarda um determinado dia indicado como um dia fechado, proibindo a efetuação de reservas e cancelamentos nessa data.

Finalmente, os métodos **flightsList** e **reservationsList** permitem consultar a lista de todos os voos existentes e a lista de reservas efetuadas pelo utilizador atual, respetivamente.

4 Teste

Na figura abaixo, temos dois clientes a comunicar com o servidor. O cliente 1 é um administrador que, após se autenticar, insere informações sobre voos. Também conectado ao servidor ao mesmo tempo que o cliente 1, temos o cliente 2, um usuário normal, que reserva uma viagem já que existe um voo com o percurso indicado e este ainda não atingiu a capacidade máxima.

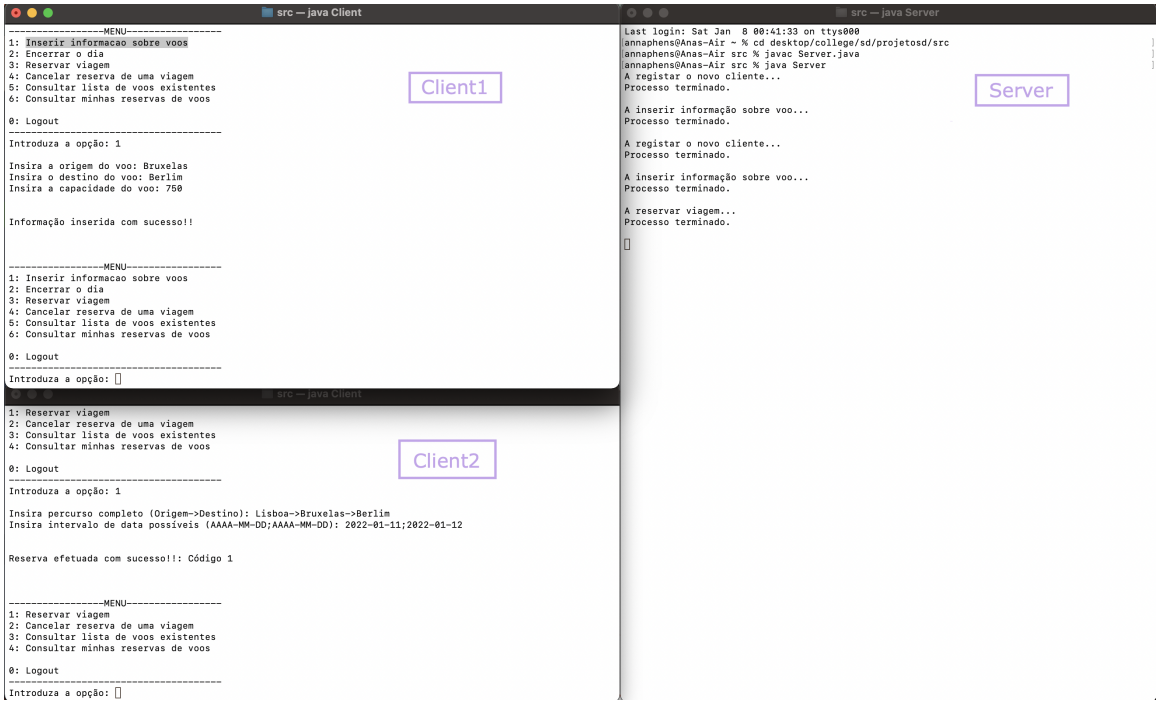

Figura 3: Teste

5 Conclusão

Para a realização deste trabalho prático, foi utilizado o controlo da concorrência para impedir haver erros ou incoerências de dados, uma vez que a plataforma pode ser acedida por diferentes utilizadores ao mesmo tempo. Para além disto, também foi necessário utilizar a programação com sockets, como pedido no enunciado, de modo a enviar pedidos do cliente para o servidor.

Este projeto permitiu-nos consolidar os conhecimentos adquiridos em Sistemas Distribuídos, nomeadamente os conhecimentos no desenvolvimento de software em ambientes multi-thread.

Por último, o grupo considera que o resultado do trabalho foi positivo, visto que todas as funcionalidades básicas requiridas, e outros requisitos, foram implementadas com sucesso.