HighFly: Sistema de Gestão de Reservas de Voos

Beatriz Rodrigues[a93230], Francisco Neves[a93202], Gabriela Prata[a93288], and João Machado[a89510]

Grupo 24
Sistemas Distribuídos
Departamento de Informática
Universidade do Minho, Braga

Resumo Implementação de um sistema de gestão de reservas de voos sobre a forma de um par cliente-servidor, utilizando sockets com conexão TCP e *threads*.

Keywords: TCP · Multithreading · Client · Server.

1 Introdução

Este relatório descreve o resultado do desenvolvimento de uma aplicação de gestão de reserva de voos, efetuado no contexto da unidade curricular de Sistemas Distribuídos, na Universidade do Minho.

O sistema é constituído por um servidor que armazena os dados necessários para o objetivo desejado e, recorrendo a multithreading, permite que múltiplos clientes interajam com ele e solicitem informação ou alterações. A comunicação realizada recorre a sockets TCP.

Foram implementadas todas as funcionalidades base e adicionais pretendidas. Para além disso, foi decidido acrescentar a possibilidade de realizar ações extra, por conveniência e por fazerem sentido no contexto da aplicação.

2 Cliente

Cada instância de um cliente é capaz de interagir com o servidor através de uma interface que utiliza um *BufferedReader* para receber input e *System.out* para lhe transmitir as opções de funcionalidades das quais pode fazer uso e o resultado de ações que tenha realizado. Este pode ser um *admin* ou um utilizador comum. As *passwords* destas entidades são alvo de um processo de encriptação através de AES de forma a salvaguardar a segurança dos dados dos utilizadores.

2.1 Admin

Quando a aplicação é iniciada, o *admin* poderá efetuar *login* através do email *admin@highfly.pt* e da senha *admin*. De seguida, pode escolher realizar qualquer uma das seguintes interações:

- Adicionar um voo (a partir da cidade de origem, destino e número máximo de passageiros);
- Adicionar um dia fechado (impedindo reservas naquele dia e cancelando as reservas que tenham sido realizadas);
- Remover um dia fechado (voltar a possibilitar futuras reservas naquele dia);
- Solicitar uma lista dos dias fechados;
- Solicitar uma lista de todos os voos existentes;

As cidades que a aplicação suporta encontram-se representadas numa *enum*, o que permite restringir a adição de voos apenas às cidades abrangidas pela companhia aérea hipotética à qual pertence o sistema.

2.2 Utilizador

Quando a aplicação é iniciada, o utilizador deverá efetuar *login* com as suas credenciais. Para isso, terá que as ter registado previamente através da opção de registo, que impossibilita que duas contas tenham o mesmo email. De seguida, terá acesso às seguintes funcionalidades:

- Fazer uma reserva (através dos ID dos voos ou através das cidades que constituem a rota);
- Solicitar uma lista dos voos existentes;
- Cancelar uma reserva (através do ID da reserva);
- Solicitar uma lista das reservas que realizou;
- Receber as possíveis rotas (com o máximo de 2 escalas) entre duas cidades;

Foram cumpridas ambas as funcionalidades adicionais: a primeira encontra-se mencionada acima, por outro lado, a segunda (que possibilita que o utilizador execute outras ações enquanto aguarda pela finalização de uma reserva), foi implementada através de uma *priority queue* contida numa classe *Notices* que é atualizada pela *thread* responsável por receber respostas a reservas enquanto o cliente estiver *online*. Para não receber notificações enquanto está a realizar outra ação, perturbando-o, as notificações são reveladas ao utilizador quando ele volta ao seu *Home menu*, caso existam operações de reserva finalizadas.

3 Model

Toda a informação encontra-se contida na classe *Model* que é constituída pelos dados de autenticação dos clientes, pelos voos existentes, pelas reservas efetuadas, pelos *IDs* das reservas mapeados pelo *email* do cliente a que pertencem, pelos dias fechados e pelos *IDs* das reservas mapeados pela data na qual a viagem irá ser efetuada.

De uma forma mais específica, um voo é representado na classe *Flight*, na qual este é identificado por um *ID* único (gerado no modelo, através de incrementos), pelo número máximo de passageiros permitidos no avião, pela cidade de origem e pela cidade de destino.

Para além disso, uma reserva é representada na classe *Reservation*, em que esta é dotada de um *ID* único, o *ID* do cliente que a efetuou, e um *Set* com os identificadores dos voos que esta contém. Para além disso, tem ainda a data para a qual foi efetuada.

É dentro do *Model* que são efetuados a maior partes dos *Locks* que permitem a prevenção de problemas de exclusão mútua. Foi priviligiada a utilização de *ReadWriteLock* uma vez que existiam múltiplas operações de leituras e era pretendido que múltiplos clientes pudessem realizar leituras dos mesmos dados em simultâneo de forma a aumentar a eficiência do programa.

Por fim, nesta classe existem vários processos de validação que impedem operações que não possam ser executadas. Alguns cuidados tomados foram os seguintes:

- Não é possível marcar voos para dias fechados;
- Não é possível fechar um dia que já tenha passado;
- Não é possível cancelar um dia fechado que já tenha passado;

– Não é possível efetuar reservas para dias fechados;
– Não é possível efetuar uma reserva através do nome das cidades a visitar se não for disponibilizado um intervalo de datas válido (por exemplo, 17/01/2022 - 15/01/2022);
– Não é possível efetuar uma reserva para uma data que já tenha passado;
– Não é possível registar mais do que uma conta com o mesmo email;
– Não é possível cancelar uma reserva para uma data que já tenha passado;
– Entre outros...

4 Servidor e conexão

O servidor permite gerir interações de múltiplos clientes em paralelo com a informação do sistema que armazena (numa interface *ModelFacade* implementada pelo *Model* previamente referido), através de *threads*. Para isto, é utilizada comunicação por *sockets* através de **TCP**, recorrendo a um *demultiplexer* e a uma *tagged connection*. Os dados são enviados/recebido na forma de *frames* que posteriormente são desserializadas para o formato pretendido.

Foi decidido que seria benéfico minimizar a transfência da informação. Desta forma, existe sempre uma validação inicial dos dados inseridos pelo cliente antes de ser enviado o pedido para o servidor. Por exemplo, se o cliente não preencher um dos campos necessários para efetuar uma dada ação, não há utilidade nenhuma em comunicar ao servidor para tentar efetuar a funcionalidade correspondente. Para além disso, o cliente e o servidor enviam apenas *strings* convertidas para *arrays* de *bytes* para garantir simplicidade e eficência tanto no envio dos dados como na deserialização da informação.

4.1 Frame

Uma *frame* é constituída por uma *tag*, pelo *email* do utilizador com quem o servidor está a interagir e os dados trocados em formato de *arrays* de *bytes*.

Cada *tag* tem associada a seguinte funcionalidade:

– 0 : Login;
– 1 : Registo;
– 2 : Adição de um voo;
– 3 : Fechar um dia;
– 4 : Fazer uma reserva através dos IDs do(s) voo(s);
– 5 : Fazer uma reserva através das cidades a atravessar;
– 6 : Cancelar uma reserva;
– 7 : Listar os voos existentes;
– 8 : Receber uma lista de rotas entre 2 cidades;
– 9 : Listar as reservas feitas por um utilizador;
– 10 : Remover um dia fechado;
– 11 : Listar os dias fechados;

De acordo com a *tag*, são efetuadas as alterações correspondentes ao *Model*.

4.2 Demultiplexer e Tagged Connection

A classe *Tagged Connection* permite a troca das mensagens através de um *socket*, a partir do qual são abertos o *DataInputStream* e o *DataOutputStream*, de forma a receber e enviar informação, respetivamente.

A classe *Demultiplexer* encapsula a classe referida acima e é utilizada do lado de cliente para, quando existem múltiplas threads, bloquear a *thread* invocadora até chegar uma mensagem com a *tag* especificada, caso esta tenha que aguardar por uma resposta.

4.3 Persistência da informação

Sempre que se acrescenta ou altera informação do sistema, a informação é serializada para o ficheiro *model.ser*, de forma a ser possível ter um *backup* dos dados para carregar no início da próxima execução do servidor. Para garantir problemas de exclusão mútua, foi usado um *Reentrant Lock*, uma vez que não se justifica a utilização de *ReentrantReadWriteLocks* neste contexto, recorrendo-se ao invés disso a esta opção menos dispendiosa.

5 Logs

Como outra funcionalidade adicional, foi decidido que seria interessante existir um registo de todas as operações efetuadas por clientes com o servidor durante a sua execução. Para isso, foi criada uma classe *Logs* que escreve para o ficheiro *.logs.txt* as alterações efetuadas ao sistema, por exemplo, se um cliente efetuou uma reserva e a sua identificação.

6 Conclusão e Trabalho Futuro

Foi concluído que o trabalho foi desenvolvido com sucesso, uma vez que cumpre todos os objetivos pretendidos, bem como possui ainda outras funcionalidades consideradas importantes no contexto da aplicação.

Consideramos ainda que, numa fase futura, este projeto poderia possuir novas funcionalidades, tais como a possibilidade de um Cliente receber uma notificação assim que uma das suas reservas for cancelada, por exemplo, quando um dia é fechado. Esta funcionalidade permitiria que, mesmo sem o Cliente estar autenticado no momento da eliminação da reserva, quando este se autenticasse fosse informado do ocorrido.

Além disso, este projeto possibilitou uma melhor compreensão acerca da utilidade de *threads*, *sockets* e *locks*.