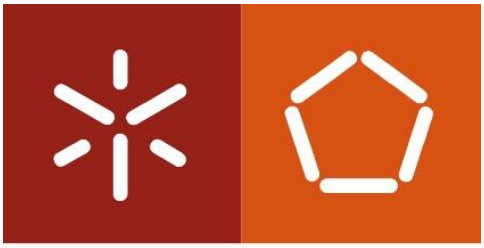
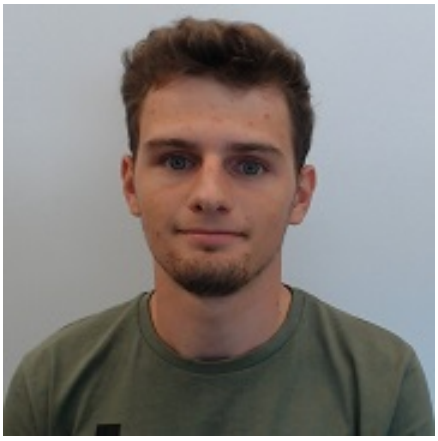


Universidade do Minho
Licenciatura em Engenharia Informática

Sistemas Distribuídos

Serviço de Gestão de Reserva de voos

Grupo 34



Bohdan Malanka
a93300

Diogo Rebelo
a93278

Henrique Alvelos
a93316

Teresa Fortes
a93250

30 de outubro de 2023

Conteúdo

1 Introdução **2**

2 Servidor **2**

3 Clientes **2**
3.1 Cliente Normal 2
3.2 Administrador 3

4 Conexão Cliente <-> Servidor **3**
4.1 TaggedConnection 3
4.2 Demultiplexer 3

5 Funcionalidades **3**
5.1 Autenticação e Registo 4
5.2 Listar todos os voos 4
5.3 Reservar um voo 5
5.4 Cancelar uma reserva 5
5.5 Inserir um voo novo - Admin 5
5.6 Encerrar um dia - Admin 6
5.7 Log-out 6

6 Conclusão **6**

1 Introdução

Neste trabalho prático era proposta a implementação de uma aplicação que tem como objetivo principal implementar e criar uma plataforma de reserva de voos, com o uso de clientes e um servidor que comunicam entre si através de sockets TCP-IP. Para a realização deste projeto foram utilizados vários conceitos aplicados nos guiões das aulas práticas e teóricas como o conceito de concorrência, exclusão mútua, secção crítica, serialização, etiquetamento, entre outros.

2 Servidor

O servidor tem como função processar pedidos provenientes de clientes. Nesse sentido, foi implementado um servidor **Threaded-per-request**, isto é, por cada conexão com cada cliente existe uma thread da classe **ClientHandler** que por sua vez invoca outras threads para cada pedido que o cliente fizer, aumentando assim o nível de concorrência do servidor. Nesse sentido, é possível o servidor estar a atender simultaneamente pedidos de vários clientes e/ou do mesmo cliente.

Embora haja 2 tipos de clientes distintos, nós implementamos apenas um **Listener** no servidor, ou seja, thread cuja funcionalidade é "escutar"o socket, e deste modo permitir o tratamento correto dos pedidos de cada cliente. Esse tratamento é feito através de **workers** que implementam um conjunto de funcionalidades refletidas na classe **ClientHandler**. Assim, esta thread estabelece uma conexão com o cliente através da classe **TaggedConnection** e recebe um pacote do tipo **Frame**. De seguida, invoca uma thread que termina o trabalho, isto é, passa o pacote à classe **ServerStub** para fazer o parse da mensagem recebida e enviando a respetiva resposta de seguida.

3 Clientes

No enunciado foram apresentados 2 tipos de clientes diferentes: cliente normal e cliente especial de administração (admin). Optamos por representá-los pela mesma classe: **Client**. Cada cliente possui funcionalidades particulares de acordo com o seu tipo. Assim sendo, a sua implementação teve como objetivo responder aos requisitos característicos de cada um, considerando a porta 5000 para ambos, visto pertencerem à mesma classe.

3.1 Cliente Normal

Para o cliente normal, considerou-se uma implementação **multi-threaded** refletida através do uso da classe **Demultiplexer**, que irá ser abordada em detalhe posteriormente, de modo a permitir efetuar múltiplos pedidos e receber respostas assíncronas.

Este tipo de cliente dispõe de diversas funcionalidades, como o momento de autenticação/registo até ao momento em que abandona o programa (através do *log-out*). Após a autenticação, o cliente tem acesso às funcionalidades do programa que serão processadas pelo servidor.

3.2 Administrador

O cliente com autorização especial também é um cliente **multi-threaded**, uma vez que este implementa mais que uma funcionalidade assíncrona. O que diferencia este utilizador de um utilizador normal é a sua conta, que é única. Para aceder às funcionalidades do admin deve-se autenticar com o username: "**ADMIN**"e palavra-passe: "**admin123**". Mal o servidor é iniciado, esta conta é automaticamente acrescentada à lista de utilizadores. Este utilizador tem funcionalidades diferentes do cliente normal, que serão explicadas mais à frente. Caso este cliente pretenda aceder às restantes funcionalidades terá de se autenticar no sistema como cliente normal com uma conta diferente.

4 Conexão Cliente <-> Servidor

Como forma de implementar a conexão entre o cliente <-> servidor, foram utilizadas duas classes: **TaggedConnection** e **Demultiplexer**. A figura seguinte ilustra a conexão feita entre o cliente <-> servidor.

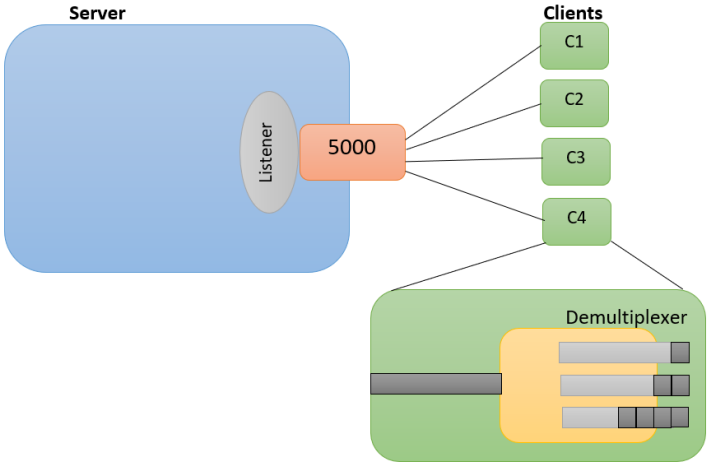


Figura 1: Conexão Cliente <-> Servidor

A partir da imagem seguinte, conseguimos visualizar a conexão entre o servidor e os vários clientes, conectados na porta 5000. A respetiva porta está a ser "escutada" por uma *thread* que vai criar as *threads* por cada nova conexão. Do lado do cliente (c1, c2, c3, c4) está implementado um *demultiplexer* que tem como uma função "escutar" o *socket* de *input* como forma de distribuir as respostas do servidor pelas *threads* respetivas de acordo com a *tag* da mensagem.

4.1 TaggedConnection

A classe **TaggedConnection** tem como função a gestão da troca de mensagens entre cliente e servidor, e vice-versa, recorrendo a recursos de etiquetamento e serialização de mensagens. Deste modo, estão implementados métodos de envio e receção de mensagens para o socket, recorrendo à serialização e escrita em binário (DataInputStream e DataOutputStream). Assim sendo, são trocados objetos serializados da classe **Frame** que tem o intuito de armazenar a informação recebida pelo servidor e cliente, respetivamente. Tanto para o cliente como para o servidor, os métodos são os mesmos e o tipo de pacote(Frame) é sempre o mesmo. Deste modo, um objeto do tipo Frame possui 3 campos: **tag**(int), **username**(String) e **data**(byte[]).

4.2 Demultiplexer

A classe *Demultiplexer* tem como objetivo agrupar as respostas provenientes do servidor de acordo com a sua tag, de forma a distribuí-las pelas *threads* que enviaram o pedido respetivo. Para cumprir com esse objetivo, existe uma *thread* que está à "escuta" de novas mensagens do socket de *input* do lado do cliente, que após a sua receção, irá "acordar" as *threads* que estejam à espera de resposta da *tag* da mensagem recebida.

5 Funcionalidades

Para a implementação deste programa foram consideradas as funcionalidades dos 2 tipos de clientes distintos, tal como referido anteriormente. Cada um destes clientes possui ações distintas.

O cliente normal pode: efetuar autenticação/registo, pedir a lista de todos os voos, reservar um voo, cancelar a sua reserva e, por fim, efetuar *log-out* da aplicação.

No caso do cliente autorizado, este tem capacidade para: efetuar autenticação/registo, pedir a lista de todos os voos, inserir um voo novo, fazer o encerramento de um dia e, por fim, efetuar *log-out* da aplicação.

```
final static int LOGIN = 1;
final static int ISLOGIN = 2;
final static int SIGNUP = 3;
final static int LOGOUT = 4;
final static int LIST = 5;
final static int ADDNEWFLIGHT = 6;
final static int BOOKFLIGHT = 7;
final static int CANCELBOOK = 8;
final static int CLOSEDAY = 9;
```

Figura 2: Tags respetivas a cada funcionalidade

5.1 Autenticação e Registo

Cliente: No lado do cliente, o utilizador escolhe iniciar sessão (opção 1 do menu inicial) com uma conta já existente ou com uma nova conta (opção 2 do menu inicial). Em cada método é pedido o nome do utilizador que deverá ser único e uma palavra passe que serão enviados para o servidor.
Servidor: No servidor, caso o utilizador tenha escolhido iniciar sessão, é validada a correspondência entre a palavra passe e o nome de utilizador e é verificado também se o respetivo cliente já tem a sessão iniciada ou não. Posteriormente, é enviada uma mensagem de confirmação ou de erro caso não haja correspondência. Para o caso de registar um novo utilizador é verificada a unicidade do nome do utilizador e caso se verifique é armazenado com a correspondente palavra passe. De seguida, é enviado para o cliente uma mensagem de confirmação ou de erro.
Cliente: De volta ao lado do cliente, este recebe a mensagem e apenas faz o print dela. O que determina se o cliente iniciou a sessão é a variável de controlo do ciclo de autenticação que é atualizada com o envio de uma nova mensagem ao servidor perguntando se o cliente iniciou a sessão ou não ao que o servidor responde com um "OK"se sim ou "KO"se não.

```
---- Flight Company by Group 34 21/22 ----
Choose an option:
1 - Login
2 - Register
0 - Quit
> 1
Username: Antonio
Password: pass123
Welcome, Antonio
```

(a)

```
---- Flight Company by Group 34 21/22 ----
Choose an option:
1 - Login
2 - Register
0 - Quit
> 2
Username: Antonio
Password: pass123
Successfully registered!
```

(b)

Figura 3: (a) Cliente inicia a sessão, (b) Cliente cria uma conta nova

5.2 Listar todos os voos

Cliente: O cliente envia o pacote com a respetiva tag, indicando que pretende obter a lista dos voos e de seguida coloca-se em modo escuta para receber os ditos voos.

Servidor: O worker responsável por lidar com este cliente vai ao servidor buscar o map dos voos. Primeiramente, envia o número total dos voos e de seguida faz **serialize** de cada voo, enviando um de cada vez ao cliente.

Cliente: Recebe todos os pacotes sequencialmente e reconstrói o map dos voos fazendo o **deserialize**. Por fim, usa o método `toString()` da classe *Flights* imprimindo a respetiva tabela. Nesta parte, o programa é bloqueado pois aparece um menu secundário com duas opções: (0) voltar ao menu principal e (1) obter a lista de todos os voos possíveis entre dois pontos origem e destino limitados a duas escalas se o utilizador é normal e aparece apenas a opção de voltar se for o admin. Se o cliente normal escolher a opção 1 não são feitas trocas com o servidor, pois ele já possui a lista dos voos, apenas imprimindo os caminhos usando a função `listPossibleRoutes(o, d.)` da classe *Flights*. Nesta função adaptamos a versão recursiva do algoritmo **DFS** limitado a duas escalas, adaptando ao nosso cenário.

Plane	Origin -> Destiny	DateLeaving	DateArrival	Places
A10	Porto -> Lisbon	13/01/2022	13/01/2022	(0/1)
A11	Lisbon -> São Paulo	14/01/2022	14/01/2022	(0/1)
A12	Lisbon -> New York	14/01/2022	15/01/2022	(0/1)
A13	Moscow -> Porto	12/01/2022	12/01/2022	(0/1)
A14	Roma -> Dubai	15/01/2022	15/01/2022	(0/1)
A15	Paris -> Roma	12/01/2022	12/01/2022	(0/1)
A16	Berlin -> Moscow	11/01/2022	11/01/2022	(0/1)
A17	London -> Porto	15/01/2022	16/01/2022	(0/1)
A18	Warsaw -> Porto	16/01/2022	16/01/2022	(0/1)
A19	Porto -> Hawaii	13/01/2022	14/01/2022	(0/1)
A1	Porto -> Madrid	10/01/2022	10/01/2022	(1/1)
A2	Madrid -> London	11/01/2022	11/01/2022	(1/1)
A3	Porto -> London	10/01/2022	10/01/2022	(0/1)
A4	Porto -> Paris	12/01/2022	12/01/2022	(0/1)
A5	Roma -> Berlin	12/01/2022	12/01/2022	(0/1)
A6	Madrid -> Berlin	11/01/2022	11/01/2022	(0/1)
A7	Paris -> London	11/01/2022	11/01/2022	(0/1)
A8	Berlin -> London	12/01/2022	12/01/2022	(0/1)
A9	Madrid -> Roma	11/01/2022	11/01/2022	(0/1)
A20	Faro -> Porto	13/01/2022	13/01/2022	(0/1)

```
Your reservations: [ANTONIO-A1-A2]
1 - List all possible routes
0 - Back
>
```

A seguinte figura ilustra uma exemplo desta funcionalidade. Note que as linhas verdes significam que é possível fazer mais reservas para esses voos. As linhas a amarelo significam que o avião já não tem espaço para mais clientes pois atingiu a capacidade máxima. As linhas a vermelho significam que esse dia foi encerrado por parte do Administrador.

5.3 Reservar um voo

Para que esta funcionalidade possibilite o cliente executar outras operações enquanto espera pela conclusão duma reserva, incluindo novas operações de reserva, é iniciada uma *thread* para executar este passo. No entanto, como a nossa *View* do projeto é em modo texto, ficava muito confuso, pois a *main* do cliente voltava a imprimir o menu principal e esperava uma nova opção e a *thread* imprimia as informações relativas à reserva e também esperava parâmetros do utilizador. Portanto, para evitar este problema, decidimos implementar esta funcionalidade em *Java Swing*. Assim, o programa continua normalmente e sempre que se escolhe esta funcionalidade aparece uma janela para fazer a reserva do voo, podendo ser várias ao mesmo tempo. Nesse sentido, importamos um biblioteca externa, que se encontra na pasta "lib", para facilitar a escolha da data que é a *JDateChooser* na janela de *Java Swing*, portanto é preciso incluir no **IDEA** para poder correr ou então também disponibilizamos os executáveis do servidor e cliente na pasta "out/artifacts".

Cliente: Assim sendo, no campo da janela é pedido o caminho completo com todas as escalas ao cliente, não havendo limite nesta parte, e o intervalo de tempo em que pretende que a sua reserva seja feita. Depois é enviado ao servidor todos os dados numa String, na qual ele se encarrega de fazer o *parse* e validar os dados.
Servidor: Recebe a *Frame* do cliente com a respetiva *tag* de reservar um voo. Faz o *parse* dos campos e verifica os dados. Para esta função é tido em conta se existem voos entre cada ponto, se a data de arranque está dentro do limite, que é encurtada à medida que percorremos as escalas, pois o dia de chegada de um avião é o limite inicial para o próximo, se o avião tem espaço e se o dia não está encerrado. Processando tudo, é gerado um código de reserva com o seguinte padrão **USERNAME-IDVOO1-IDVOO2...** que é enviado ao cliente se tudo correr bem ou então uma mensagem de erro do respetivo problema.
Cliente: Apenas recebe a mensagem e mostra-a numa janela menor. Se for afirmativa, encerra ambas as janelas, senão o cliente pode alterar os campos e tentar outra vez ou sair

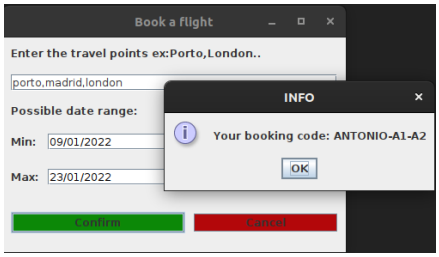


Figura 4: Cliente efetua uma reserva

5.4 Cancelar uma reserva

Cliente: Se o cliente executar a função de listar os voos (opção 1 do menu principal) no fundo da tabela aparecem os seus códigos de reserva, se se tratar de um utilizador normal. Assim, na funcionalidade de cancelar uma reserva (opção 3) é necessário fornecer um código de reserva que é enviado ao servidor com a respetiva *tag* da funcionalidade.
Servidor: Recebe o código de reserva e faz *parse*, pois este possui o *id* do utilizador e os *ids* dos respetivos voos reservados, se for escalonado ou de um só voo. Valida os dados e cancela a reserva desocupando o lugar nos voos, enviando uma mensagem afirmativa ao utilizador se tudo correr bem.
Cliente: Apenas imprime a mensagem recebida.

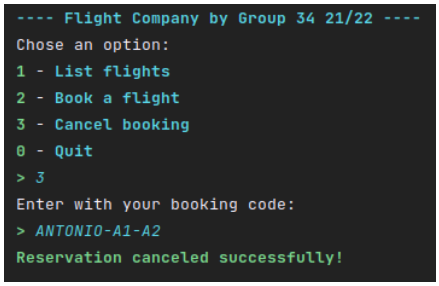


Figura 5: Cliente cancela uma das suas reservas

5.5 Inserir um voo novo - Admin

Cliente: Para executar esta função, são pedidas as respetivas informações sobre o novo voo, como origem, destino, data de partida e data de chegada. A capacidade não é necessária, pois consideramos que todos os aviões da empresa têm a mesma

capacidade e por isso é uma variável global. Posto isto, é enviada a *Frame* que contém todos os dados em uma String e o servidor fica responsável de interpretar.
servidor: Como foi referido, faz o *parse* da mensagem, valida os dados, nomeadamente se o formato das datas está correto e se a ordem também, e coloca o novo voo no *map* dos voos. Por fim, retorna uma mensagem afirmativa ou de erro.
Cliente: Apenas imprime a resposta recebida pelo servidor.

Figura 6: Admin adiciona um novo voo

5.6 Encerrar um dia - Admin

Cliente: O Administrador fornece uma data para encerrar um dia e é enviado ao servidor a relativa mensagem.
Servidor: Recebe a data e itera todos os voos colocando a variável booleana *isClosed* a *true*, se a data de partida for equivalente à data fornecida. Por fim, retorna a mensagem afirmativa se tudo correr bem ou com erro se o formato da data estiver errado.
Cliente: Recebe a mensagem e imprime-a.

Figura 7: Admin faz o encerramento de um dia

5.7 Log-out

Cliente: É enviado ao servidor uma pacote com a *tag* relativa ao terminar sessão, apenas para informar que o utilizador pretende terminar sessão, sendo o campo dos dados irrelevante.
Servidor: Recebe o pedido e muda a variável *logged* do utilizador a *false*, envia a última mensagem de despedida e quebra o ciclo infinito para a escuta pois já não vai receber mais mensagens deste cliente e a execução do *worker* termina.
Cliente: Imprime a resposta recebida e faz o `close()` do *Demultiplexer* e do *socket*, encerrando assim a execução do programa.

6 Conclusão

Dado por concluído o projeto Serviço de Gestão de Reserva de voos, consideramos necessário efetuar uma reflexão crítica do trabalho realizado e dos resultados obtidos. Deste modo, é importante fazer um balanço do trabalho realizado, tendo em conta tanto os aspetos positivos como possíveis melhorias na implementação.

Por um lado, sublinhamos o facto de todas as funcionalidades estarem implementadas (básicas e adicionais). Além disso, implementámos um servidor *threaded-per-request* ao invés de *threaded-per-connection*, ou seja, para além de se criar uma thread responsável por atender um cliente, criam-se outras a partir dessa, mas para cada pedido.

Por outro lado, poderíamos fazer melhorias que enriqueceriam o nosso programa. Seria mais benéfico se implementássemos um cliente do mesmo género, ou seja, para cada funcionalidade do cliente normal, seria enviado um pedido ao servidor por uma *thread* específica, de modo a permitir respostas assíncronas e a evitar o bloqueio do programa. No entanto, como as nossas interfaces de menu são em modo texto, o utilizador não iria perceber o que se passava pois os *prints* das mensagens não estariam sequenciais. A única solução que encontramos foi como fizemos para a funcionalidade de reservar voos, ou seja, numa janela à parte feita em *Java Swing*.

Para finalizar, consideramos que obtivemos um balanço positivo na globalidade do trabalho.