



Universidade do Minho  
Departamento de Informática

## Sistemas Distribuídos

### Grupo 26

16 de janeiro, 2022



Diogo Pires  
(a93239)



Gonçalo Soares  
(a93216)



Guilherme Fernandes  
(a93286)



Mariana Rodrigues  
(a93229)

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Descrição do Trabalho</b>	<b>4</b>
2.1	Comunicação cliente/servidor . . . . .	4
2.2	Airport System . . . . .	4
<b>3</b>	<b>Implementação</b>	<b>5</b>
3.1	Common . . . . .	5
3.2	Client . . . . .	6
3.3	Server . . . . .	6
3.3.1	Tratamento de pedidos . . . . .	6
3.3.2	Airport System . . . . .	6
3.4	Funcionalidades básicas . . . . .	7
3.5	Funcionalidades Adicionais . . . . .	7
3.6	Testes . . . . .	8
<b>4</b>	<b>Conclusão</b>	<b>9</b>

# **Capítulo 1**

## **Introdução**

O presente relatório procura descrever e explicar a implementação do trabalho desenvolvido no âmbito da Unidade Curricular de Sistemas Distribuídos.

A temática proposta aborda o desenvolvimento de uma plataforma de reserva de voos capaz de gerir pedidos em simultâneo de vários clientes.

A aplicação deve permitir que vários utilizadores utilizem os seus serviços ao mesmo tempo. E para tal, torna-se crucial implementar métodos de controlo de concorrência, para garantir que todos os pedidos são atendidos atempadamente e corretamente.

No decorrer deste relatório iremos detalhar os diversos componentes da nossa aplicação, e de que modo estes interagem para a execução de todas as funcionalidades propostas.

# Capítulo 2

## Descrição do Trabalho

Este trabalho divide-se em duas grandes partes: **comunicação cliente/servidor** e **Airport System**.

### 2.1 Comunicação cliente/servidor

Na parte de **cliente/servidor** focamo-nos na comunicação entre estas duas entidades, na qual implementamos um **demultiplexador**, para um cliente poder realizar vários pedidos enquanto espera pela resposta de outros. Essa comunicação é feita em binário, tentando minimizar ao máximo o fluxo de dados trocados.

Um dado utilizador ao iniciar a conexão ao servidor, é lhe atribuída uma *thread*. Desta forma, todos os pedidos do mesmo, passam a serem tratados na *thread* atribuída.

Os pedidos realizados pelo cliente são encaminhados para o **Airport System**, e cada um destes terá uma **tag** que indica qual o tipo de pedido. Consoante a **tag**, tanto no servidor como no cliente será chamado o respetivo método que ou tratará o pedido ou receberá a resposta deste, respetivamente.

### 2.2 Airport System

Já o **Airport System** encontra-se responsável pelo tratamento da componente lógica do sistema. Aqui foi fundamental a implementação de *locks* para garantir a atomicidade das operações. Visto que, o servidor pode invocar vários métodos simultaneamente e pretendemos que não hajam interferências indesejadas entre eles.

Este encontra-se responsável por realizar reservas de voos, cancelar dias, obter rotas possíveis entre duas dadas localidades, entre outras operações.

Para isso, organizamos o nosso sistema utilizando os seguintes tipos de informação:

- **Flight** Estes possuem informações sobre a rota a que pertencem, o dia em que são realizados, e as reservas que lhe estão associadas.
- **Reservation** Estes guardam os voos associados a uma reserva de um cliente.
- **User** Este guarda as informações sobre cada um dos utilizadores da aplicação. Aqui encontramos dois tipos de **user**: o **client** e um **admin** (utilizador especial).

Todas estas entidades possuem identificadores únicos, com excepção da **Route**.

# Capítulo 3

## Implementação

A implementação do presente trabalho foi dividido em 3 módulos:

- **Client**: Responsável pela implementação do cliente e efetuar os seus pedidos.
- **Server**: Responsável pela ligação entre os pedidos do cliente e o **Airport System**.
- **Common**: Responsável pelas classes comuns entre os módulos anteriores.

Para conseguirmos concorrência no nosso sistema, dividimos as várias componentes de forma a que os *locks* utilizados bloqueassem apenas o necessário.

Sendo assim, passaremos à descrição das diferentes partes da implementação:

### 3.1 Common

- **Flight** - Nesta entidade guardámos as informações relativas a um voo. Visto que as operações sobre um voo nem sempre pretendem modificá-lo, como é o caso da consulta de lugares disponíveis, decidimos implementar um *ReadWriteLock*.
- **Route** - Nesta entidade guardámos as informações relativas à ligação entre duas localidades. Após a sua criação, manter-se-á constante durante todo o programa, e por isso não necessita da utilização de um mecanismo de controlo de concorrência.
- **LockObject** - Esta entidade é necessária para generalizar a utilização de locks, evitando repetir código. A sua utilidade é atribuir um *ReadWriteLock* a um objeto de qualquer classe.
- **PossiblePath** - Esta entidade é necessária para implementarmos o pedido de obter os caminhos possíveis entre dois locais. Utilizámo-la de forma a diminuir a quantidade de dados transferidos entre cliente e servidor. Como esta classe só é criada em resposta a um pedido, e não é partilhada, não precisa de qualquer tipo de controlo de concorrência. A sua utilidade será descrita *a posteriori*.
- **Reservation** - Nesta entidade encontra-se toda a informação relativa a uma reserva de um utilizador. Aqui foi implementado apenas um *ReentrantLock* para termos controlo sobre os voos relativos a uma reserva (*Set<Flight> flights*).
- **Users** - Um **User** conterá a informação relativa a cada utilizador como *username* e *password*.

A cada **User** encontra-se associado um *ReentrantLock*, porque os métodos associados a este modificam e atualizam as suas informações. Apenas no caso da obtenção de **username** é que não se usa o *lock*, pois este é constante durante todo o decorrer a aplicação. E visto que, o único método de leitura serve para verificar a *password*, aquando a autenticação, não se justifica o uso de *ReadWriteLock*.

## 3.2 Client

Este módulo é responsável pela realização dos pedidos dos utilizadores da aplicação. Para isso, implementamos um menu que apresenta os vários tipos de pedidos possíveis. Para cada opção do menu, é apresentada uma fase de *input*, onde é pedido ao utilizador os dados que pretende enviar. Após isto, o **client** manda o respetivo pedido ao servidor.

Nesta secção, foi adicionado um demultiplexer para que consigamos guardar vários pedidos a serem enviadas sequencialmente ao servidor. Assim, o utilizador pode dar queue de pedidos, e se estes forem pedidos mais demorados, são recebidos numa *thread* separada quando estiverem prontos. No caso dos pedidos serem rápidos são recebidos de imediato na *thread* principal.

## 3.3 Server

Este módulo encontra-se responsável pelo tratamento de vários pedidos e pelo **Airport System**.

### 3.3.1 Tratamento de pedidos

Sempre que um novo pedido chega ao servidor, e antes de passarmos ao seu tratamento, este será guardado numa *queue*.

- As inserções de pedidos em espera na *queue* é feita utilizando locks;
- No caso de não existirem pedidos em espera, a *thread* ficará em *await* até um novo pedido ser adicionado à *queue*.
- Cada pedido é encaminhado para a *thread* atribuída ao utilizador que fez o pedido.

### 3.3.2 Airport System

Esta secção encontra-se responsável pelo armazenamento da nossa base de dados, bem como da sua manutenção.

- **Map<String, LockObject<Map<String, Route>> connectionsByCityOrig** - Aqui, as rotas são guardadas num Map, cuja key é a cidade de origem. Cada cidade de origem têm correspondência a um conjunto de cidades de destino, onde também guardamos as rotas. Assim, o custo de procura é baixo, diminuindo o tempo de bloqueio de cada consulta.
- **Map<LocalDate, LockObject<Map<Route, Flight>> flightsByDate** - Aqui são armazenados todos os voos por dias, num Map cuja key é o dia do voo. A cada dia, associámos um Map que associa as rotas e voos correspondentes. Quando procuramos informações de voos relativos a um dia, começamos por bloquear toda a estrutura *flightsByDate*, e depois bloqueamos a informação correspondente a esse dia. Mantendo o dia bloqueado, desbloqueámos os *flightsByDate*. Desta forma, podemos procurar informações sobre um dia sem bloquear consultas relativamente a outros dias.
- **Map<String, User> usersById** - Aqui são armazenados todos os *users* por *id*. Optamos por usar um *ReadWriteLock*, visto que grande parte das operações sobre este serão de leitura.
- **Map<UUID, Reservation> reservationsById** - Aqui são armazenados todos as *reservation's* por *id*. Optamos por usar um *ReentrantLock*, visto que grande parte das operações sobre este serão de escrita.
- **Set<LocalDate> canceledDays** - Aqui armazenamos os dias cancelados. Usamos um *ReadWriteLock* porque a quantidade de vezes que os consultamos ao iniciar uma reserva, é muito superior à quantidade de vezes que o vamos modificar.

### 3.4 Funcionalidades básicas

De um modo geral e em todos os pedidos, usámos o "*Two-phase locking*" para não bloquear estruturas de dados em excesso.

Um exemplo dessa utilização é na inserção de uma **Route**. Começamos por bloquear o *lock* de escrita da estrutura que armazena todas as **Routes** por local de origem, e após isso, também bloqueámos o *lock* de escrita do **LockObject<Map<String, Route>**, *map* associado a todas as rotas que partem dessa cidade de origem. Com isto, desbloqueámos a estrutura principal, inserimos a nova rota e desbloqueámos o **LockObject**. Assim, reduzimos o tempo que mantemos bloqueada a estrutura que armazena todas as **Routes**.

Agora vamos descrever algumas particularidades de *queries* que consideramos importantes:

- **Reserva de voos** - Para reservar um voo começamos por saber quais as rotas associadas aos locais indicados pelo utilizador. Depois procuramos a partir da data de início se é possível reservar os voos associados a essas rotas. Se for possível, isto é, o dia não se encontra cancelado e existirem lugares livres, bloqueámos esse voo. Fazemos isto para cada conexão indicada, e caso seja possível, fazemos a reserva. Essa reserva é depois guardada nos voos, e estes são desbloqueados. Caso não seja possível fazer todos os voos, desbloqueámos os voos que foram bloqueados até esse ponto, e avisamos o utilizador.
- **Cancelar um dia** - Ao cancelar um dia por parte de um administrador, deve deixar de ser possível reservar nesse dia e os voos desse dia devem ser cancelados. Por outro lado, caso um voo seja de conexão, os lugares associados a esse voo também são cancelados. Exemplificando, um cliente compra uma reserva com dois voos, e o dia do primeiro é cancelado, o sistema cancela automaticamente o voo do segundo, mesmo que seja realizado noutro dia.

Fora isso, também implementamos:

- Registo e autenticação de utilizadores.
- Inserção e obtenção de informações sobre voos.
- Cancelar reservas.

### 3.5 Funcionalidades Adicionais

- **Rotas possíveis entre dois locais** - Obtenção de uma lista com todos os percursos possíveis entre 2 locais, limitados a duas escalas.

Para minimizar a quantidade de dados transferida por este pedido, decidimos criar a classe **PossiblePath**. Com esta classe, conseguimos guardar várias rotas com menos informação, em que por exemplo "A:[B:[C],C]" representa as conexões: "A->B->C", e "A->C".

- **Alterar password** - Possibilidade de alteração da **password** de um dado **user** autenticado.

- **Consultar reservas** - Obtenção de todas as reservas de um dado utilizador autenticado.

- **Consultar notificações** - Obtenção de todas as notificações de um dado Utilizador.

Aqui foi decidido que qualquer notificação destinada a um **User** que não está autenticado será reservada para o mesmo assim que ele volte a dar login no sistema.

- **Demultiplexer** - Com isto, um utilizador da aplicação é capaz de realizar várias operações enquanto espera pelo resultados de outros pedidos.

### 3.6 Testes

Para ser possível estudar a coerência do sistema à medida que modificávamos e adicionávamos funcionalidades ao trabalho, foram desenvolvidos testes que garantissem isso.

Para tal, decidimos criar *três* grupos de testes:

- **AirportSystem** - Neste conjunto de testes verificámos se o aeroporto funciona corretamente em termos de lógica.  
Aqui foram criados testes para cada método da interface **IAirportSystem**, e exceções associadas.
- **Concorrência no AirportSystem** - Para testar a concorrência não fizemos testes tão meticulosos como os anteriores, mas procuramos testar métodos que originassem erros. Para isso, criámos um teste geral que realiza várias operações ao mesmo tempo, nomeadamente: inserir rotas; cancelar dias; registar clientes e fazer reservas. Com estes testes, conseguimos descobrir erros que tínhamos no programa.
- **Testes da conexão do cliente ao servidor** - Para testar a conexão entre o cliente e servidor foram efetuados testes em que foram criados múltiplos clientes simultaneamente. Estes conectam-se ao servidor e efetuam vários pedidos. Nos testes realizados, apenas testamos a inserção de rotas e o processamento de reservas. Por outro lado, este módulo de teste permite uma fácil adição de novos testes.

Depois de efetuar os testes, concluímos que tanto os clientes como o servidor funcionam de forma correta.

## **Capítulo 4**

### **Conclusão**

Com a realização deste trabalho constatamos a importância de programação concorrente, e gestão de variáveis partilhadas. Por outro lado, também ficamos a entender melhor o funcionamento da comunicação entre clientes e servidor. Com isto, conseguimos consolidar os conhecimentos adquiridos nas aulas leccionadas.

Dito isto, achamos que conseguimos desenvolver um programa que vai de encontro não só aos requisitos pedidos como de funcionalidades adicionais.