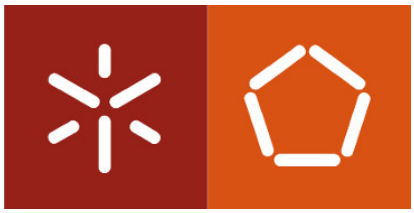


Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistemas Distribuídos - Trabalho Prático
Ano Letivo 2021/2022

Grupo 6

Gonçalo Braz (a93270) Simão Cunha (a93262)
Tiago Silva (a93277) Gonçalo Pereira (a93168)

25 de janeiro de 2022

1 Introdução

Motivado pela importância de plataformas de gestão nos dias de hoje e pela iniciativa em implementar conceitos da UC de Sistemas Distribuídos, foi-nos proposto para este trabalho prático o desenvolvimento de uma plataforma de reservas de voos. Esta funciona num modelo servidor-cliente onde vários clientes podem-se conectar através de *threads* e *sockets* TCP.

No nosso sistema, permitimos aos clientes que se registem na plataforma, iniciem sessão (e terminem), ver o catálogo de voos existentes na aplicação, ver as suas reservas efetuadas, possibilitar a reserva num voo numa certa data e cancelar reservas. Além disso, permitimos a existência de um cliente do tipo *Administrador* que pode registar novos *admins* na plataforma, adicionar mais voos no programa e encerrar um dia (i.e. não permitir novas reservas para esse dia assim como o cancelamento das já existentes).

O presente relatório descreverá em detalhe a arquitetura da aplicação, a implementação dessa arquitetura e as funcionalidades da plataforma.

2 Arquitetura

2.1 Cliente

Esta entidade representa a componente da aplicação responsável por interagir com o utilizador e é responsável por enviar pedidos (funcionalidades descritas na secção `Introdução`) do utilizador para o servidor assim como as respostas aos mesmos.

O cliente trata de enviar o pedido que consiste, em primeiro lugar, enviar um inteiro que identifica o comando, seguido da informação necessária para execução desse comando no lado do servidor, ficando à espera da resposta do servidor.

Opcode	Funcionalidade associada	Send	Receive
0	**Encerrar conexão**	-	-
1	**Registar no sistema**	(0 ou 1) e 2 Strings	boolean
2	**Login**	2 Strings	1 ou 2 booleans
3	**Logout**	-	-
4	**Listar voos**	-	Vooslist
5	**Listar reservas**	-	ReservasList
6	**Efetuar reserva**	2 Strings	boolean
7	**Cancelar reserva**	int	boolean
8	**Encerrar dia**	String	boolean
9	**Adicionar voo ao sistema**	2 Strings e int	boolean

O cliente oferece ainda um suporte visual de menu que trata de facilitar a vida do utilizador, bem como, gerir o seu próprio funcionamento, já que praticamente todas as funcionalidades necessitam que o utilizador esteja logado no sistema. Existe, ainda, algumas funcionalidades que são apenas visíveis aos administradores. Assim sendo, este começa por mostrar o menu de registo/login, e, assim que o utilizador esteja logado, apresenta o respetivo menu baseado nas suas permissões.

Mais à frente no relatório será explicada na secção de implementação, o que exatamente é enviado para o servidor, e o que este retorna.

2.2 Servidor

Quando o servidor é iniciado, é instanciado um objeto da classe `VoosManager` para onde são carregados para a memória informação sobre os utilizadores, os voos, as reservas e as datas encerradas.

Após esta iniciação, o servidor está sempre à escuta de uma nova conexão por parte de um cliente. Quando este conecta-se ao servidor, é invocada uma *thread* com um *Handler* que é responsável por atender pedidos, permitindo, assim, atender múltiplos clientes. Na classe `Handler` estão definidos os comandos pedidos pelo cliente e onde estes são executados. Esta classe recebe, a socket, o VooManager e ainda o id do cliente (para mera ilustração visual nos logs que o servidor gera no terminal).

2.3 Classe Utilizador

Esta classe é identificada por um nome (String) e possui uma password (String) e um *boolean* que representa se o utilizador/cliente é um administrador ou *"normal"*.

2.4 Classe Reserva

Esta classe representa a reserva de um voo. Para tal, será necessário saber a origem (String) e o destino (String), o nome do utilizador que a efetuou (String), um código que a identifica (int), uma lista com os *ID*'s dos voos das escalas (array de Strings) e a data da reserva (LocalDate). Esta classe implementa métodos de *serialize* e *deserialize*.

2.5 Classe Voo

Esta classe representa um voo no sistema. Um voo é identificado por um id (String), por uma origem (String), por um destino (String) e pela capacidade do voo (int). Conta ainda com um map [data (LocalDate),lotaçao (int)] para gerir as lotações. Esta classe implementa métodos de *serialize* e *deserialize*, e ainda, suporte de concorrência de threads, através da utilização do lock; suporte necessário, devido à alteração de lotação (adicionar ou remover passageiros),

2.6 Classe ReservasList

Esta classe estende *ArrayList<Reserva>* e implementa os métodos *serialize* e *deserialize*.

2.7 Classe VoosList

Esta classe estende *ArrayList< Voo>* e implementa os métodos *serialize* e *deserialize*.

2.8 classe VoosManager

Esta é a classe que manuseia e gere toda a informação do sistema. Esta, quando é construída, recebe de parâmetros 4 Strings, que são equivalentes aos nomes do ficheiros (de utilizadores, de voos, de reservas e dias encerrados) donde vai carregar/salvar a informação. A classe conta com 3 maps (um de utilizador, um de voos e um de reservas) e 1 lista (de dias encerrados) para guardar a sua informação. A classe suporta também concorrência de *threads*, uma vez que tem *ReadWriteLocks* para cada *collection* (`utilizadoresLock`, `voosLock`, `reservasLock` e `datasLock`), e faz os devidos locks e unlocks, quando estas são acessadas. De notar que para não ter que acumular datas encerradas na lista excessivamente, o manager guarda ainda a data de arranque, e assume que todas as datas que ocorram antes desta estão encerradas por "default".

Esta classe oferece os seguintes métodos:

1. public void load()
Método que carrega as informações dos ficheiros.

2. public boolean updateUtilizador(Utilizador u)
Método que adiciona um utilizador ao manager. Faz writeLock.lock() no utilizadoresLock antes de inserir no map o novo utilizador e writeLock.unlock() depois. De notar, que apenas regista se ainda não existir um utilizador com aquele nome.

3. public boolean updateVoo(Voo v)
Método que adiciona um voo ao manager. O manager cria um id para o voo, somando o nome da origem com o nome do destino, caso já exista um voo, ou vários, com essa mesma origem e destino, então é adicionado ao id um '#' e o número equivalente à quantidade de voos com essas carateristicas, exemplo: Ao inserir um voo de Porto para Lisboa, se já existir no sistema, o novo voo ficará com o id: PortoLisboa#2. Faz writeLock.lock() no voosLock antes de inserir no map o novo voo e writeLock.unlock() depois.

4. public VoosList getVoos()
Método que retorna uma lista de todos os voos do sistema, este faz readLock.lock() no voosLock e depois readLock.unlock().

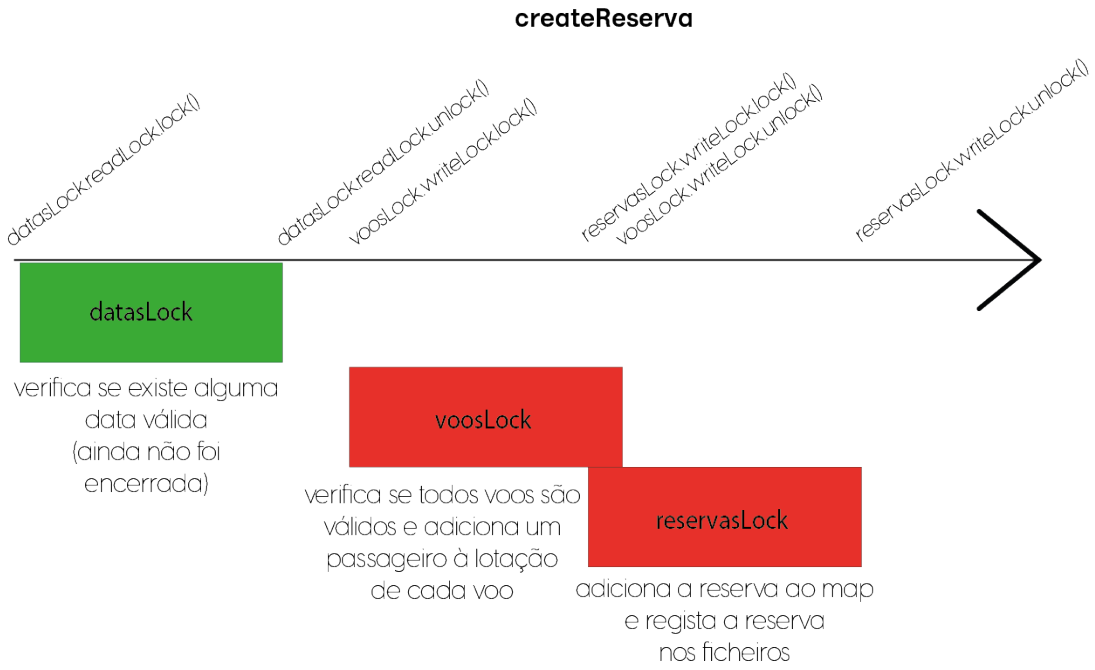
5. public ReservasList getReservas(String nomeUser)
Método que retorna uma lista de todos as reservas de um determinado utilizador do sistema, este faz readLock.lock() no reservasLock e depois readLock.unlock().

6. public Utilizador getUtilizador(String nomeUser)
Método que retorna o utilizador pelo seu nome, este faz readLock.lock() no utilizadoresLock e depois readLock.unlock().

7. public boolean checkCredentials(String nomeUser, String password)
Método que verifica se as credenciais (nome e password) de um utilizador correspondem com o que está no manager. Este método também faz readLock.lock() no utilizadoresLock e depois readLock.unlock().

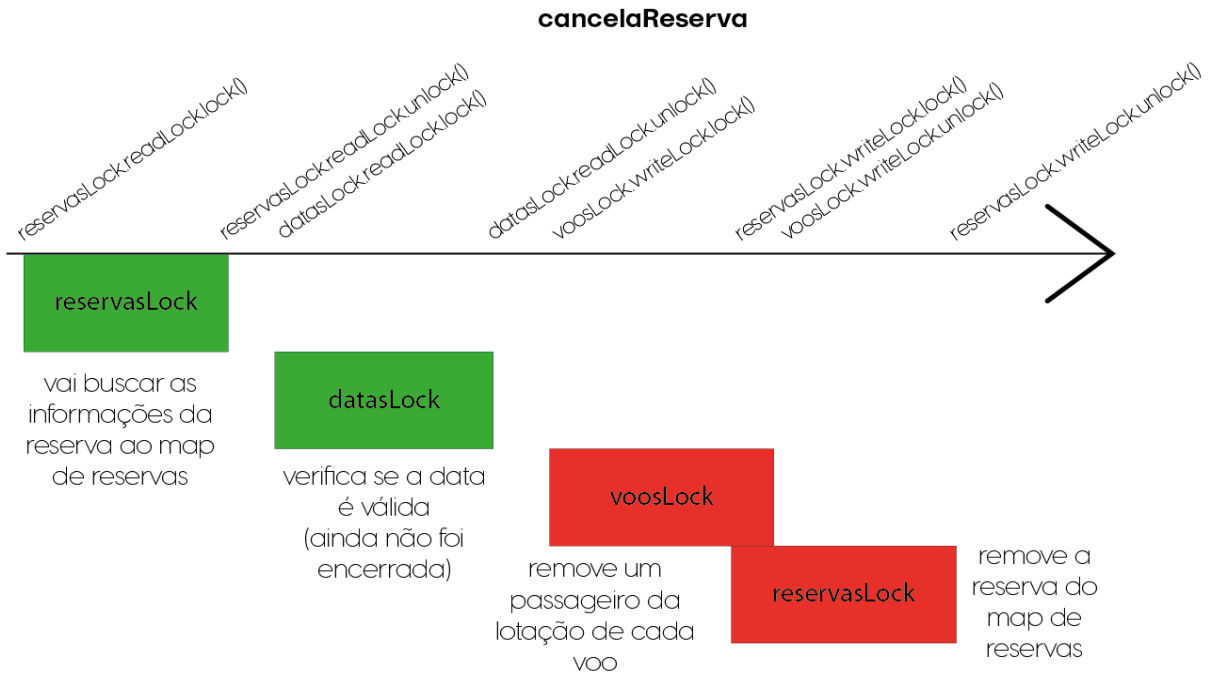
8. public boolean encerraDia(LocalDate data)
Método que adiciona uma data à lista de datas encerradas do manager. Este primeiro verifica se a data já não conta na lista, e se não se passa antes da data de Arranque, se isso se verificar, adiciona à lista. Retorna um booleano a confirmar, ou não, o sucesso da adição. O método faz writeLock().lock() no datasLock e depois writeLock.unlock().

9. public int createReserva(String nomeUser, String[] pontos, String[] datas)
Método que cria uma nova reserva no manager. Este começa por verifica se existe alguma data válida (que ainda não tenha sido encerrada), caso exista passa para a verificação dos pontos, que consiste em ver se existe voos para a passagem de todos os pontos pretendidos, e caso exista, adiciona um passageiro à lotação de cada voo. Por fim regista então a nova reserva no sistema. Como este método mexe com várias colections do manager, é normal que faça mais de um lock, por isso, para facilitar a compreensão da estratégia de locks neste método, segue um diagrama do fluxo do método que exemplifica o funcionamento dos locks.



Legenda: Os locks a verde significam os readLocks, e os vermelhos significam os writeLocks.

10. public boolean cancelaReserva(int codReserva, String nomeUser)
Método que cancela uma reserva de um determinado utilizador. Esta começa por ir buscar a reserva, através do código de reserva (codReserva) e verificar se está associada ao utilizador (nomeUser). Em seguida, verifica se a data da reserva ainda não foi encerrada, e caso não tenha sido, procede ao cancelamento, que nada mais é do que, remover um passageiro da lotação de cada voo, e remover a reserva do map de reservas. À semelhança do método anterior, segue em seguida a explicação dos locks em formato de diagrama.



2.9 Conexão e comunicação

A conexão cliente-servidor é estabelecida através de *sockets* TCP. A comunicação é através de *DataInputStream*'s e *DataOutputStream*'s, permitindo, assim, o fluxo de dados.

2.10 Implementações das funcionalidades do programa

2.10.1 Registar no sistema

Cliente: envia opcode 1, seguido de 0 ou 1 (1 se for registo de um admin, 0 se não); uma String com o nome do utilizador e uma String com a password.

ServerHandler: utiliza do método nos VoosManager, updateUtilizador, e envia ao cliente um boolean (true caso haja tenha havido sucesso no registo, false se não).

2.10.2 Login

Cliente: envia opcode 2, uma String com o nome do utilizador e uma String com a password.

ServerHandler: utiliza do método nos VoosManager, checkCredentials, e envia ao cliente um boolean (true caso haja tenha havido sucesso no login, false se não).

2.10.3 Listar voos

Cliente: envia opcode 4.

ServerHandler: utiliza do método nos VoosManager, getVoos, e envia ao cliente a lista de voos serialized.

2.10.4 Listar reservas

Cliente: envia opcode 5.

ServerHandler: utiliza do método nos VoosManager, getReservas, e envia ao cliente a lista de reservas serialized.

2.10.5 Efetuar reservas

Cliente: envia opcode 6, seguido com uma string com os pontos da viagem dividida por ';' e uma string com as datas em que o utilizador está disponível, também esta dividida com ';'.

ServerHandler: utiliza do método nos VoosManager, createReserva, e envia ao cliente um boolean (true se a reserva tiver sido feita com sucesso, false caso não).

2.10.6 Cancelar reservas

Cliente: envia opcode 7, seguido com um inteiro que identifca o código da reserva.

ServerHandler: utiliza do método nos VoosManager, cancelaReserva, e envia ao cliente um boolean (true se a reserva tiver sido cancelada com sucesso, false caso não).

2.10.7 Encerrar dia

Cliente: envia opcode 8, seguido com a uma String da data a ser encerrada.

ServerHandler: utiliza do método nos VoosManager, encerraDia, e envia ao cliente um boolean (true se o dia tiver sido encerrado com sucesso, false caso não).

2.10.8 Adicionar voo

Cliente: envia opcode 9, seguido com a uma String da origem do voo, uma String do destino, e um int que transmite a capacidade do voo.

ServerHandler: utiliza do método nos VoosManager, updateVoo, e envia ao cliente um boolean (true se o voo tiver sido adicionado com sucesso, false caso não).

3 Conclusão

Este trabalho prático permitiu consolidar os conhecimentos obtidos nas aulas de Sistemas Distribuídos de uma forma útil e interessante. Cremos ter um projeto bem definido, embora apenas tenhamos implementado as *queries* básicas, que, no entanto, foram bem sucedidas.

Verificamos que não utilizamos conceitos como o *Demultiplexer* ou *Frammed Connection*, uma vez que esses inserem-se nas *queries* opcionais.

Tendo em conta que este projeto foi escrito com a linguagem de programação JAVA, poderíamos ter optado por uma arquitetura por camadas par, como o modelo MVC (*Model, View* e *Controller*), tal como nos já ensinaram em UC's passadas do nosso percurso académico. Contudo, devido à complexidade exigida, decidimos utilizar os modelos semelhantes aos utilizados nos guiões das aulas práticas.