

Sistemas Distribuídos
Relatório de Trabalho Prático
Grupo 23



A82418 José Gomes



A69858 João Ribeiro

1 Introdução

Neste projeto foi implementado um serviço de reserva e gestão de voos. Deste modo, foi implementado um *Servidor* que executa todas as operações de *Backend*, um *Administrador* para gestão da informação no sistema e um *Cliente* que pode fazer reservas de voos.

2 Classes

Nesta secção apenas referimos as classes que achamos mais importantes para compreender a maneira como organizamos o projeto.

2.1 Reserva

Unidade utilizada para armazenar uma reserva de uma determinada viagem por um determinado utilizador. A classe *Reserva* contém os aeroportos por ordem pelos quais a viagem passa, o utilizador que a reservou e o dia da viagem.

```
public class Reserva {  
  
    private ArrayList<String> aeroportos;  
    private String user;  
    private LocalDate dia;  
}
```

Figure 2: Variaveis da Classe Reserva.png

2.2 Voo

Unidade utilizada para representar um voo. Esta classe é composta por *Origem*, *Destino* e *CapacidadeMax*.

```
public class Voo {  
  
    private String origem;  
    private String destino;  
    private int capacidade_Maxima;  
}
```

Figure 3: Variaveis da Classe Voo.

2.3 Servidor

Classe que aceita conexões vindas dos objectos da classe *Cliente* e as atribui a um *ServerSocketHandler*.

```
while(true){  
    Socket socket = ssocket.accept();  
  
    Thread thread = new Thread(new ServerSocketHandler(socket, capacidadeMap, reservaMap, userMap));  
    thread.start();  
}
```

Figure 4: Variaveis da Classe Servidor.

2.4 ServerSocketHandler

Classe principal do *BackEnd* que executa todas as funcionalidades do programa. Para tal, contém também, toda a informação do sistema, contida em 3 estruturas, exploraremos essas estruturas no capítulo a seguir.

```
public class ServerSocketHandler implements Runnable {  
    private Socket socket;  
    private String user;  
    private MapReservas reservaMap;  
    private MapUsers userMap;  
    private ListaVoos listaVoos;
```

Figure 5: Variaveis da Classe ServerSocketHandler.

2.5 Cliente

Classe que comunica com um objecto da classe *ServerSocketHandler*, através da qual o *Administrador* e o *Cliente* interagem com o sistema, e exploram as funcionalidades do programa.

3 Estruturas

Neste capítulo vamos aprofundar sobre as estruturas escolhidas para desenvolver este projeto.

3.1 ListaVoos

A Classe *ListaVoos* contém um *ArrayList<Voo>*, uma lista de todos os Voos existentes no sistema e um *ReentrantReadWriteLock*, utilizado para controlar o acesso a esta *classe/lista*, porque achamos expectável que a estrutura seja mais frequentemente consultada do que alterada.

3.2 MapCapacidades

A Classe *MapCapacidades* contém um *HashMap<"Origem-Destino", ListaCapacidades>* e um *ReentrantLock*.

A classe *ListaCapacidades*, contém um *ArrayList<Capacidade>*, que armazena as capacidades diarias para um determinado Voo e um *ReentrantLock*, este *Lock* interior é utilizado apenas quando se acrescenta novos dias à lista de capacidades, deste modo, listas de capacidades de vários voos podem ser acedidas e alteradas simultaneamente.

O *lock* desta classe é apenas utilizado quando se acrescenta novos voos, pois é preciso adicionar uma entrada ao *HashMap*.

```
public class ListaCapacidades {  
    private ArrayList<Capacidade> listaCapacidades;  
    private final int maxCapacidade;  
    private ReentrantLock lock;
```

Figure 6: Exemplo de classe com implementação Lock.

3.3 MapReservas

A Classe MapReservas contém um *HashMap*<Código, Reserva>, um MapCapacidades, um contador de reservas, o dia atual e *ReentrantLock*. O *HashMap* permite um acesso instantâneo as reservas efetuadas.

Temos nesta classe o MapCapacidades pois ao cancelar uma reserva, tem de ser atualizado o número de lugares disponíveis nessa viagem.

O *Lock* é utilizado quando se acrescenta ou remove reservas, de modo a impedir acessos enquanto esta classe está incompleta ou com informação incorreta.

3.4 MapUsers

Contém apenas um *HashMap*<Username, Password>. Utilizamos novamente o *HashMap* pelo fácil e rápido acesso na autenticação, tanto do utilizador como do administrador. Como não está implementada a expansão ou contração do *HashMap* não é necessária a utilização de *Lock*.

4 Implementação

Neste projeto utilizamos diversos *ReentrantLock* para garantir que não existem acessos a estruturas que estão a ser alteradas, tanto por estarem a ser expandidas ou contraídas. Deste modo evitamos acesso a informação incorreta, mas também mantemos livre o acesso a informação independente.

```
public boolean cancelarReserva(int codigo, String user){
    try{
        lock.lock();
        Reserva reserva = reservas.get(codigo);
        if(reserva != null)
            if(reserva.getUser().equals(user))
                if(diaActual.compareTo(reserva.getDia()) <= 0){
                    ArrayList<String> aeroportos = reserva.getAeroportos();
                    for(int i = 0; i < aeroportos.size()-1; i++){
                        String oriDest = aeroportos.get(i)+'-'+aeroportos.get(i+1);
                        capacidades.get(oriDest).get(reserva.getDia()).decrementa();
                    }
                    reservas.remove(codigo);
                    return true;
                }
            return false;
    } finally {
        lock.unlock();
    }
}
```

Figure 7: Exemplo de utilização de Lock ao cancelar uma reserva.

Relativamente à comunicação entre *Cliente* / *Servidor* tudo o que o Cliente vê no seu terminal é enviado pelo Servidor, sendo que o Cliente apenas imprime essa informação e envia as respostas às *queries* ao Servidor.

O Servidor executa todas as operações e funcionalidades sobre o sistema.

Utilizamos uma espécie de variável de controlo para sabermos que tipo de informação o Servidor está a enviar. Se a variável de controle não existir o servidor espera uma resposta ao pedido recebido. Se o controle for 0, significa que o servidor não espera resposta, é apenas informação para o utilizador ver. Esta também pode ser 1, se assim for o servidor também não espera resposta pois o que o cliente recebeu é a lista completa de todos voos possíveis. E por último, se a mensagem recebida for *LogOff* o cliente recebeu confirmação que a sua sessão foi encerrada com sucesso e fecha o terminal de comunicação.

A seguinte imagem mostra o método do Cliente que comunica com o Servidor.

```
private static void communicating(DataOutputStream outStrm, DataInputStream inStrm) throws IOException {
    Scanner input = new Scanner(System.in);
    String info;
    String[] received;
    Boolean communication = true;

    while(communication){
        received = inStrm.readUTF().split( regex: "\\|");

        switch (received[0]) {
            case "0":
                System.out.println(received[1]);
                break;
            case "1":
                ListaVoos lVoos = ListaVoos.deserialize(inStrm);
                System.out.println(lVoos);
                break;
            case "LogOff":
                communication = false;
                break;
            default:
                System.out.println(received[0]);

                info = input.nextLine();

                outStrm.writeUTF(info);
                break;
        }
    }
}
```

Figure 8: Método do Cliente que comunica com o Servidor.

Criamos ainda a Classe *Menu* de modo a tentar separar ao máximo o texto de *interface* do código. A seguinte imagem mostra um método dessa classe.

```
public static String menuCliente(){
    return "----- Menu Cliente -----\\n" +
        "1) Reserva de Voos\\n" +
        "2) Cancelar Reserva\\n" +
        "3) Obter Lista de Voos\\n" +
        "4) Calcular Viagem\\n" +
        "5) Sair";
}
```

Figure 9: Texto do Menu acedido pelo Cliente.

5 Funcionalidades

Neste projeto concluímos todas as funcionalidades requeridas com a execução das adicionais.

O Cliente tem as seguintes funcionalidades:

- Reservar Voos
- Cancelar Reserva
- Obter Lista de Voos

O Administrador tem as seguintes funcionalidades:

- Inserir Novo Voo
- Encerrar Dia

6 Conclusão

Após completar este pequeno projecto, sentimos que conseguimos evoluir mais um pouco o nosso conhecimento sobre *Sistemas Distribuídos*, e que compreendemos melhor as dificuldades que podem surgir nas implementações destes mesmos. Sentimos, portanto, que este projecto conseguiu atingir o seu propósito, e que conseguimos ultrapassar grande parte das nossas dificuldades. Vemos também potencial não aproveitado no nosso trabalho, que não foi possível realizar devido à falta e gestão de tempo, para estudar melhor outras matérias e implementá-las, mas esperamos conseguir explorar no nosso futuro académico e/ou profissional.