

TEC502 - MI - Concorrência e Conectividade

Bilhete Compartilhado

Adlla Katarine Aragão Cruz Passos¹

¹Engenharia de Computação – Universidade Estadual de Feira de Santana (UEFS)
Av. Transnordestina, s/n – Novo Horizonte - 44036-900 – Feira de Santana – BA – Brasil

adllakatarine@hotmail.com

Abstract. *This report presents the resolution of the problem proposed in TEC502 - MI - Competition and Connectivity 2019.1. The use of the RMI programming interface was necessary to develop a distributed and decentralized system. The goal was the development of three airlines where a customer, from any server, can book available flights on other airlines.*

Resumo. *Este relatório apresenta a resolução do problema proposto na disciplina TEC502 – MI – Concorrência e Conectividade 2019.1. Foi exigido o uso da interface de programação RMI para o desenvolvimento de um sistema distribuído e descentralizado. O objetivo era o desenvolvimento de três companhias aéreas onde um cliente, a partir de qualquer servidor pudesse reservar trechos disponíveis em outras companhias conveniadas.*

1. Introdução

O problema consiste no desenvolvimento de um sistema distribuído descentralizado. Este sistema é composto por três companhias aéreas brasileiras (A, B e C), sendo elas os servidores, e seus usuários, o cliente. Cada companhia é responsável pela venda de bilhetes compondo os trechos de partida-chegada de seus estados preestabelecidos e o objetivo final é que um usuário, logado em qualquer uma das companhias, seja capaz de reservar trechos disponíveis nas outras companhias conveniadas.

Assim, um problema pode ocorrer. O que aconteceria caso dois ou mais usuários quisessem comprar um mesmo trecho, e tendo este apenas o último bilhete? Este seria um caso de *deadlock*, onde mais de um processo, os usuários, estariam requisitando um mesmo recurso, o trecho.

Diante disso, duas exigências foram feitas para este problema. A primeira foi o uso do RMI para a implementação da comunicação cliente-servidor e servidor-servidor. A segunda foi tratar o *deadlock* que pode ocorrer caso dois ou mais usuários desejem comprar trechos iguais.

2. Fundamentação Teórica

2.1. RMI

RMI (Remote Method Invocation) é uma interface de programação que permite a invocação de métodos que residem em diferentes máquinas em aplicações desenvolvidas em Java. O JVM pode estar em diferentes máquinas ou podem estar na mesma máquina. Em ambos os casos, o método pode ser executado em um endereço diferente do processo de

chamada. O RMI é um mecanismo de chamada de procedimento remoto orientada a objetos[da Costa].

Uma aplicação RMI é frequentemente composto por um servidor e um cliente. O servidor cria objetos remotos e faz referências a esses objetos disponíveis. Em seguida, ele é válido para clientes invocarem seus métodos sobre os objetos. O cliente executa referências remotas aos objetos remotos no servidor e invoca métodos nesses objetos remotos.

Por fim, o RMI é composto de três propriedades:

- Localização de objetos remotos.
- Comunicação com objetos remotos.
- Carregar os bytecodes de classe dos objetos que são transferidos como argumentos ou valores.

2.2. Deadlock

Deadlock diz respeito a uma situação em que ocorre um impasse, e dois ou mais processos ficam impedidos de continuar suas execuções. Ou seja, os processos competem entre si e ficam bloqueados, esperando uns pelos outros[Rodrigues].

De acordo com Tanenbaum, “Um conjunto de processos estará em situação de deadlock se todo processo pertencente ao conjunto estiver esperando por um evento que somente um outro processo desse mesmo conjunto poderá fazer acontecer”.

Existem cerca de quatro condições para a ocorrência de deadlock(é necessário que todas as condições devem ocorrer para que o deadlock ocorra):

- Condição de exclusão mútua: cada recurso pode estar somente associado a um único processo ou disponível.
- Condição de posse e espera: processos que possuem algum recurso podem requerer outros recursos.
- Condição de não preempção: recursos já alocados não podem ser retirados do processo que os alocou; somente o processo que alocou os recursos pode liberá-los.
- Condição de espera circular: um processo pode esperar por recursos alocados a outro processo.

3. Metodologia

Para o desenvolvimento do projeto, algumas coisas foram inicialmente definidas:

- O projeto seria desenvolvido na linguagem de programação Java e faria uso do padrão MVC(model-view-control), como pode ser visto na Figura 1.
- As companhias A, B e C são responsáveis, respectivamente, pelas regiões Norte e Nordeste, Centro-Oeste, Sul e Sudeste.
- Usuários serão cadastrados com persistência de seus dados.
- Um conjunto de trechos formam uma passagem. Um mesmo usuário poderá comprar quantas passagens desejar.

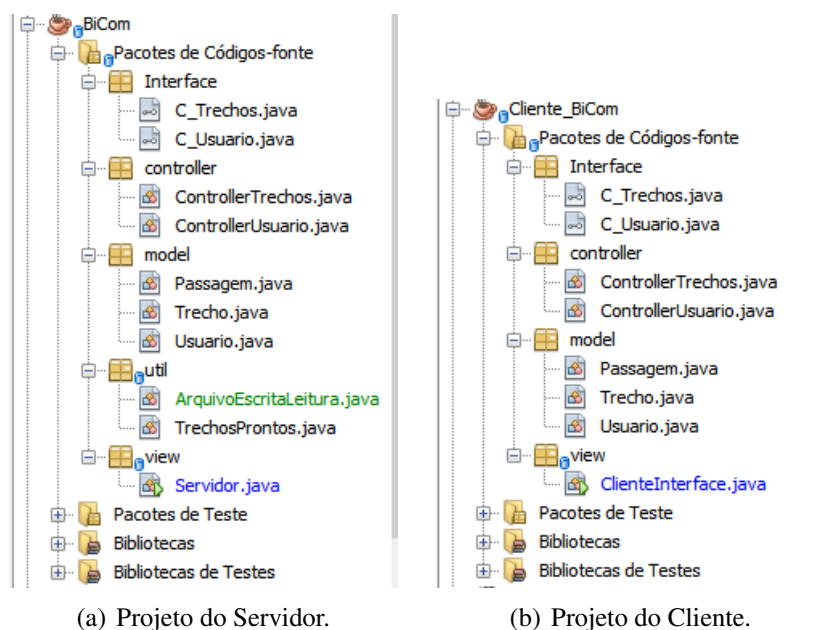


Figura 1. Projetos do servidor e do cliente no padrão MVC.

Primeiramente, para a aplicação do RMI, foram definidas duas **interfaces**, uma para o cadastro e login do usuário e outra para tratar a compra dos bilhetes; foi escolhido trabalhar com duas interfaces para uma melhor divisão de responsabilidades de cada uma. Logo após, no **controller**, a classe **ControllerUsuario** implementa a interface referente ao cadastro-login(**C_Usuario**), enquanto que o **ControllerTrechos** implementa a interface **C_Trechos**. O mesmo ocorre com o cliente, com a diferença é claro, da implementação dos métodos em cada um.

No **model**, três classes foram criadas. A primeira é a classe **Trechos**, contendo, dentre as informações mais importantes, local de partida, local de chegada e dois ID, sendo um que identifica de qual companhia aérea o local de partida pertence e outro em relação ao local de chegada. A segunda é a classe **Passagem**, tendo ela uma lista de trechos, valor total da passagem e um booleano que indicará se a passagem já foi vendida ou não. A terceira classe é o **Usuario**, que possui seus dados pessoais e uma lista de passagens. Todas implementam *Serializable*.

No servidor, a **util** contém duas classes, uma para a persistência dos usuários em arquivo .ser e a outra que instancia automaticamente todos os trechos de cada Companhia Aérea.

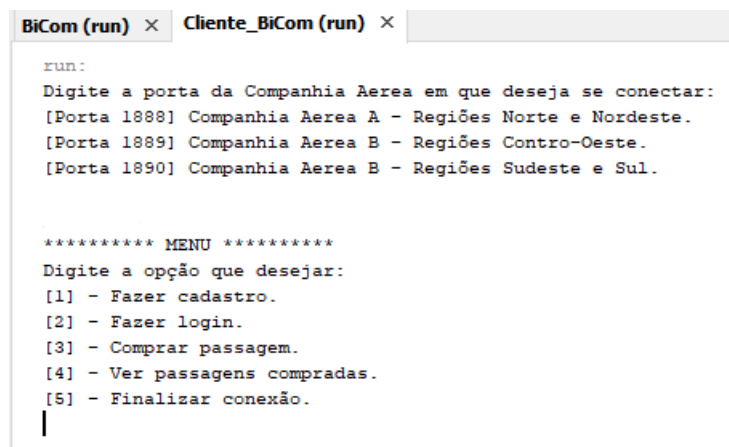
Por fim, a view. Cada uma foi desenvolvida com interação com usuário via console. No servidor, foram usadas as classes *LocateRegistry* e *Registry*. Com a *LocateRegistry*, foi usado o método *createRegistry* que cria e exporta uma instância do *Registry* no host local que aceita solicitações na port especificada[Registry-Documentation]. Do *Registry*, foi usado o método *bind* que define um endereço para um objeto remoto (foram criados dois, um para o usuário e outro para os trechos). No cliente, foi usada a classe *Naming* que fornece métodos para armazenar e obter referências a objetos remotos em um registro de objeto remoto, e seu método *lookup* que recebe o IP, a porta e o endereço de objeto remoto do servidor[Naming-Documentation].

Em relação as comunicações, não foi implementada a comunicação entre os servidores, pois estava dando erros que não foi possível resolver. Sendo assim, a responsabilidade dos servidores de finalizar a compra do usuário foi passada para o cliente. O cliente verifica o ID de local de partida de cada trecho que deseja comprar, se conecta ao servidor que está vendendo este bilhete. No servidor, ele verifica se ainda há bilhetes à venda, se sim, ele retorna um array nulo de bilhetes esgotados. Caso não haja mais algum bilhete, ele retorna este array para o cliente, que faz o tratamento para a venda dos restantes ao usuário.

4. Resultados e Discussões

Inicialmente, o usuário deverá escolher uma porta referente à região que preferir, como mostra a Figura 2. Logo depois, o menu exibe 5 opções. Vamos supor que o usuário queira fazer o cadastro e ele precisará informar nome, CPF, email, user e senha e após o cadastro já poderá começar a escolher seus bilhetes, partindo de algum estado da região do seu servidor. Caso o local de chegada do bilhete escolhido seja o da Companhia C, o usuário, automaticamente irá se conectar a este servidor para ver os bilhetes disponíveis desta companhia e assim por diante até que se deseje finalizar sua compra(ele ficará logado ao último servidor ao qual foi conectado dependendo de seu último trecho requisitado).

Ao tentar finalizar a compra, o cliente irá verificar se há bilhetes disponíveis, assim como já explicado anteriormente. Caso sim, o usuário concluirá a compra com todos os trechos escolhidos. Caso não, sua compra será finalizada mesmo assim com os trechos escolhidos que restaram. Por fim, o usuário será redirecionado ao menu novamente e, dentre as opções, poderá visualizar todas as suas passagens compradas, com todos os trechos.



```
BiCom (run) x Cliente_BiCom (run) x
run:
Digite a porta da Companhia Aerea em que deseja se conectar:
[Porta 1888] Companhia Aerea A - Regiões Norte e Nordeste.
[Porta 1889] Companhia Aerea B - Regiões Centro-Oeste.
[Porta 1890] Companhia Aerea B - Regiões Sudeste e Sul.

***** MENU *****
Digite a opção que desejar:
[1] - Fazer cadastro.
[2] - Fazer login.
[3] - Comprar passagem.
[4] - Ver passagens compradas.
[5] - Finalizar conexão.
|
```

Figura 2. Menu do cliente.

5. Conclusão

O problema proposto foi importante para o aprendizado da interface *RMI*, bem como a sua implementação. Contudo, alguns requisitos não foram cumpridos no desenvolvimento do código, como:

- Comunicação entre os servidores, pois estava dando muitos erros que não foi encontrada solução. Portanto, tais responsabilidades dos servidores foram passadas para o cliente.

-
- Tratamento do *deadlock*. Apesar de ter sido usado o *synchronized* nos métodos de compra dos trechos, sua real implementação e funcionamento não foi muito bem entendido.
 - O usuário pode montar suas passagens sem que seu local de Chegada do trecho anterior seja o seu local de Partida do próximo trecho a ser comprado.
 - A persistência dos usuários só são feitas no servidor em que o usuário fez o cadastro e seus dados não são replicados aos outros servidores.

Referências

da Costa, D. G. Uma introdução ao rmi em java. Disponível em: <https://www.devmedia.com.br/uma-introducao-ao-rmi-em-java/28681>. Acesso em Agosto de 2019.

Naming-Documentation. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/rmi/naming.html>. Acesso em Agosto de 2019.

Registry-Documentation, L. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/rmi/registry/loc>. Acesso em Agosto de 2019.

Rodrigues, H. A. Introdução ao deadlock. Disponível em: <https://www.devmedia.com.br/introducao-ao-deadlock/24794>. Acesso em Agosto de 2019.