3 de Janeiro de 2016

Universidade do Minho
Escola de Engenharia

Sistemas Distribuídos

Trabalho Prático

Gestor de Táxis

Trabalho realizado por (grupo 66):

Adriano Dias Teixeira - a67663

Filipe Manuel Gonçalves de Oliveira - a67686

João Miguel Gonçalves Fernandes - a67727

João Simões Farinha - a69302

Introdução

As aplicações distribuídas lidam normalmente com vários utilizadores, e como tal devem estar preparadas para responder aos vários pedidos por eles feitos simultaneamente.
Assim, neste tipo de sistemas, o controlo de concorrência torna-se essencial para que o sistema tenha um bom desempenho, mantendo ao mesmo tempo a consistência dos dados.

Neste trabalho procuramos explorar as funcionalidades do modelo cliente-servidor com comunicação orientada à conexão via TCP, através do desenvolvimento de um servidor multi-threaded que funciona como intermediário na comunicação entre clientes, utilizando a linguagem de programação Java para implementar tanto o servidor como o cliente.

O tema desenvolvido consiste na implementação de uma aplicação distribuída que permite a gestão de um serviço de taxi, com a qual interagem dois tipos de clientes, os passageiros que solicitam viagens ao sistema, e os condutores que informam o sistema quando estão disponíveis e no final da viagem, o preço da deslocação.

Estrutura da aplicação

A aplicação é constituída pelas seguintes classes principais:

Cliente: Esta classe implementa uma interface de utilizador, comunicando com o servidor através de sockets TCP, enviando-lhe pedidos (input) do utilizador e recebendo as respetivas respostas. Esta classe faz uso das classes “BufferedReader” e “PrintWriter” para trocar (enviar e receber) mensagens com o servidor, bem como ler input do utilizador.

Servidor: Esta classe implementa um servidor, que recebe conexões de clientes, mantendo em memória toda a informação relativa aos utilizadores e respetivas viagens, sendo esta informação partilhada por várias threads (uma por cada cliente), inicializadas com instâncias de “ServidorRunnable”.

ServidorRunnable: De cada vez que um cliente se tenta ligar ao servidor, é criada uma nova thread (do “lado” do servidor), inicializada com uma nova instância desta classe, que será responsável por gerir a ligação estabelecida com esse cliente e satisfazer os pedidos do mesmo. Para além do referido, existe também nesta classe informação acerca do cliente com o qual se mantém a conexão e o seu estado. Tal como a classe “Cliente”, também esta faz uso das classes “BufferedReader” e “PrintWriter” para trocar (enviar e receber) mensagens com os respetivos clientes.

Utilizador: Esta classe representa o tipo de utilizador mais básico existente no sistema, cuja informação abrange o seu nome, a sua password e, também, a indicação de que está online/offline. Caso um cliente seja passageiro, ele será representado através da classe "Utilizador", sendo que a classe "Condutor" extende esta.

Utilizadores: Esta classe possui uma coleção de utilizadores, organizados por nome, de forma alfabética, e fornece operações para atuar sobre esse conjunto, mais especificamente dois métodos para adicionar utilizadores (um para passageiros e um para condutores), bem como um método para efetuar login e outro para efetuar logout de um dado utilizador.

Condutor: Esta classe é uma especialização da classe "Utilizador", criada para representar os clientes que são condutores. Para além dos dados herdados da sua superclasse, esta possui ainda informações referentes ao carro do condutor, nomeadamente a sua matrícula e o seu modelo.

Viagem: Esta classe representa a associação entre um passageiro e um condutor, contendo os nomes de ambos, bem como as respetivas coordenadas. Sempre que um condutor anuncia disponibilidade para conduzir e não há passageiros disponíveis é criada uma nova instância desta classe, utilizando o método construtor para condutores, que recebe a informação relativa ao condutor. Quando um passageiro solicita uma viagem e não há condutores disponíveis é criada uma nova instância de Viagem, utilizando o método construtor para passageiros, que recebe a informação relativa ao passageiro.

Viagens: Esta classe possui duas coleções de objetos da classe Viagem, sendo que uma das coleções serve para guardar as viagens criadas por condutores que, aquando do anúncio da disponibilidade para conduzir, não possuíam passageiros disponíveis para seleção e, portanto, ficaram em espera; analogamente, a outra coleção guarda as viagens criadas por passageiros que, aquando da solicitação de uma viagem, não possuíam condutores disponíveis para seleção e, portanto, ficaram em espera.

Para além das classes referidas, há ainda cinco classes de exceção, cujos nomes permitem facilmente deduzir o seu propósito: UtilizadorNaoExisteException, UtilizadorJaExisteException, UtilizadorOnlineException, UtilizadorOfflineException e PasswordIncorretaException.

Implementação das funcionalidades pretendidas

Registo de utilizador

O registo é feito da forma “registar <nome> <password>“, caso se pretenda criar um passageiro, ou “registar <nome> <password> <matricula> <modelo>“, caso se pretenda criar um condutor. Em qualquer das situações, o servidor interpreta os comandos recebidos, chamando o método “adicionarUtilizador” correspondente (pois há um método para adicionar passageiros e outro para adicionar condutores) na classe Utilizadores, que verifica se o nome já existe na coleção de utilizadores e, caso não exista, cria um novo Utilizador ou Condutor (dependendo da situação), inserindo-o na coleção.
Ambos os métodos “adicionarUtilizador” da classe Utilizadores são synchronized, já que invocações simultâneas destes métodos poderiam ser conflituosas, visto que duas threads poderiam ambas confirmar a inexistência, na coleção de utilizadores, de um mesmo nome que elas pretendem inserir e ambas criarem um utilizador, inserindo-os na coleção, mas ocorrendo um “overwrite” de tal forma que apenas um novo utilizador fará parte da coleção, nomeadamente o último a ser inserido. Desta forma, torna-se necessário obter o lock (neste caso, optou-se pelo lock implícito) da instância da classe Utilizadores antes de efetuar a operação de registo.

Login

A autenticação no sistema é feita da forma “login <nome> <password>“. Quando um cliente envia o referido comando ao servidor, este último chama o método “logIn” da classe Utilizadores, que, por sua vez, caso verifique a existência de um utilizador com o nome referido, chama o método “logIn” da classe Utilizador sobre esse utilizador, que verifica o estado do utilizador e se a password está correta, procedendo à atualização do estado da ligação.
O método “logIn” da classe Utilizador é synchronized, já que duas threads podem tentar autenticar-se com o mesmo nome, sendo que ambas confirmam a existência do utilizador na coleção através do método “logIn” da classe Utilizadores e, nesse caso, é necessário impedir atualizações simultâneas do estado desse utilizador, que é conseguido obtendo o lock implícito do objeto que representa o utilizador em questão.

Solicitar uma viagem

A solicitação de uma viagem é realizada da forma “solicitar <x1> <y1> <x2> <y2>”, sendo que x1/y1 representam as coordenadas do local de partida e x2/y2 representam as coordenadas do ponto de destino. Quando um cliente solicita uma viagem, o servidor chama o método “escolherCondutor” da classe Viagens, que faz uso do lock existente nessa classe,

para que não ocorra a situação em que várias threads escolhem o mesmo condutor. O referido método, após obter o lock associado, verifica se existe algum condutor disponível, através do método “haCondutorDisponivel”, que faz uso do mesmo lock (tal é possível pois o lock é reentrante), verificando se existe alguma viagem sem passageiro associado, que tenha sido criada por um condutor que se encontre em espera.
Caso haja uma viagem (condutor) disponível, é então complementada a informação da viagem escolhida (conforme o cálculo da menor distância) com a informação relativa ao passageiro e os locais de partida e destino, através do método “atribuirPassageiro” da classe Viagem.
No caso de não haver qualquer condutor disponível, ou seja, nenhum condutor anunciou disponibilidade para conduzir ou, de entre os que o fizeram, já todos têm um passageiro associado, o método “escolherCondutor” retorna o valor nulo, que é interpretado no servidor como a inexistência de condutores disponíveis, sendo criada uma nova instância de Viagem, com apenas a informação relativa ao passageiro, e adicionada à coleção respetiva no objeto Viagens através do método “adicionarViagemPassageiro”, que pode então ser selecionada por uma thread de um condutor que pretenda realizar uma viagem. De seguida, é chamado o método “esperarAtribuicaoCondutor” sobre a viagem criada, que faz uma espera (não ativa) com “await()” na Condition “semCondutor” da classe Viagem, que serve para esperar pela atribuição de um condutor. A thread fica então em espera até que uma thread de um condutor selecione a mesma viagem e chame sobre ela o método “atribuirCondutor”, que introduz as informações do condutor no objeto Viagem e avisa as threads à espera na Condition “semCondutor”.

Anunciar disponibilidade para conduzir

O anúncio de disponibilidade para conduzir realiza-se através de um comando que segue o formato “anunciar <x> <y>”, em que x/y representam as coordenadas da localização do condutor. Quando um cliente anunciar disponibilidade para conduzir, o servidor chama o método “escolherPassageiro” da classe Viagens, que faz uso do lock existente nessa classe, para que não ocorra a situação em que várias threads escolhem o mesmo passageiro. O referido método, após obter o lock associado, verifica se existe algum passageiro disponível, através do método “haPassageiroDisponivel”, que faz uso do mesmo lock (tal é possível pois o lock é reentrante), verificando se existe alguma viagem sem condutor associado, que tenha sido criada por um passageiro que se encontre em espera.
Caso haja uma viagem (passageiro) disponível, é então complementada a informação da viagem escolhida (conforme o cálculo da menor distância) com a informação relativa ao condutor, através do método “atribuirCondutor” da classe Viagem.
No caso de não haver qualquer passageiro disponível, ou seja, nenhum passageiro solicitou uma viagem ou, de entre os que o fizeram, já todos têm um condutor associado, o método “escolherPassageiro” retorna o valor nulo, que é interpretado no servidor como a inexistência de passageiros disponíveis, sendo criada uma nova instância de Viagem, com apenas a informação relativa ao condutor, e adicionada à coleção respetiva no objeto Viagens através do método “adicionarViagemCondutor”, podendo então ser selecionada por uma thread de um

passageiro que pretenda solicitar uma viagem. De seguida, é chamado o método “esperarAtribuicaoPassageiro” sobre a viagem criada, que faz uma espera (não ativa) com “await()” na Condition “semPassageiro” da classe Viagem, que serve para esperar pela atribuição de um passageiro. A thread fica então em espera até que uma thread de um passageiro selecione a mesma viagem e chame sobre ela o método “atribuirPassageiro”, que introduz as informações do passageiro no objeto Viagem e avisa as threads à espera na Condition “semPassageiro”.

Simular uma viagem

Após a atribuição de um passageiro/condutor ao objeto Viagem criado por uma thread de um condutor/passageiro, inicia-se então a troca de informação entre os clientes (intermediada pelo servidor). O servidor envia à thread do condutor as informações do passageiro e à thread do passageiro as informações do condutor. De seguida, a thread que representa o passageiro, chama o método “esperarChegadaDoCondutor”, que fica à espera na Condition “condutorNaoChegou”, até que a thread do condutor chame o método “anunciarChegadaDoCondutor” (mediante input do cliente, nomeadamente “ok”), que sinaliza a thread do passageiro que está à espera nessa mesma Condition. Quando o método “esperarChegadaDoCondutor” retorna, a thread do passageiro envia uma mensagem ao cliente passageiro, para que este saiba que tal aconteceu. A seguir, a thread do passageiro chama o método “esperarChegadaAoDestino”, que fica à espera na Condition “viagemNaoAcabou”, até que a thread do condutor chame o método “anunciarChegadaAoDestino” (mediante input do cliente, nomeadamente “fim”), que sinaliza a thread do passageiro, que, por sua vez, envia a informação ao cliente. Após o referido, o servidor envia a informação de que a chegada ao destino já aconteceu, bem como o preço a pagar.

Conclusão

Durante o desenvolvimento deste trabalho pudemos pôr em pratica muito do que estudámos durante este semestre. Usando os mecanismos de controlo de concorrência do java conseguimos assim criar uma aplicação distribuida capaz de garantir o bom funcionamento de um serviço de taxis tal como esperado.

Surgiram algumas duvidas durante o desenvolvimento como, por exemplo, se deveriamos usar uma/várias variaveis de condição para ter em consideração o estado de um pedido de viagem. Apesar disso terminámos com uma aplicação funcional que cumpre os requisitos propostos e por isso achamos que este trabalho foi bem sucedido e nos foi muito útil para melhor interiorizar a matéria lecionada.