Universidade do Minho
Escola de Engenharia

Sistema de Gestão de Frota de Trotinetes Elétricas

Trabalho Prático
Sistemas Distibuídos

Grupo 39

Emanuel Lopes Monteiro da Silva - a95114

Gabriela Santos Ferreira da Cunha - a97393

Miguel de Sousa Braga - a97698

Nuno Guilherme Cruz Varela - a96455


a95114


a97393


a97698


a96455

Conteúdo

1 Introdução

Para este projeto, foi-nos proposta a implementação de uma plataforma de gestão de uma frota de trotinetes elétricas, sob a forma de um par cliente-servidor, em Java, utilizando *threads* e *sockets*. O serviço consiste em permitir ao utilizador reservar e estacionar trotinetes em diferentes locais. O sistema permite ainda recompensar os utilizadores que levam as trotinetes estacionadas de um local A para um local B, proporcionando, deste modo, uma melhor distribuição de trotinetes pelo mapa.

2 Arquitetura

A arquitetura seguida pelo grupo pode ser representada pela figura seguinte.

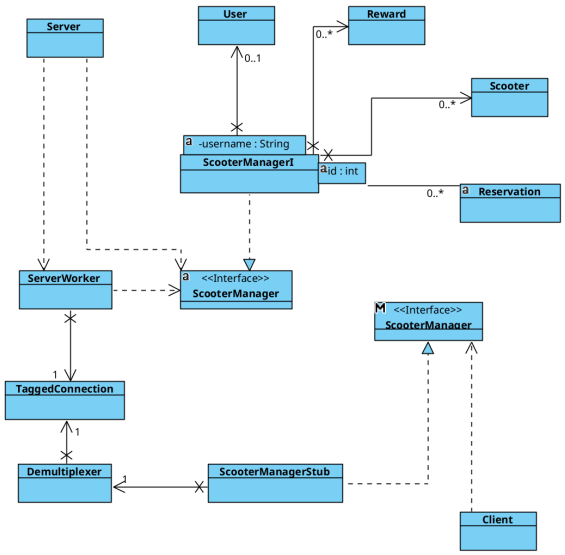

Figura 1: Arquitetura do sistema.

Um dos guias essenciais do nosso trabalho foi a ideia da abstração e da modularidade do código. Como tal, desenvolvemos uma arquitetura assente em camadas e módulos independentes. Podemos notar que o cliente tem acesso a um módulo de *middleware*, que foi desenvolvido na classe "ScooterManagerStub", para conseguir interagir com o servidor de forma abstrata, contribuíndo para a transparência de acesso da aplicação. Este *stub* faz uso das classes "TaggedConnection" e "Demultiplexer" que garantem o encapsulamento da conexão com o servidor. O servidor também faz parte da camada de *middleware* usando, como tal, as classes anteriormente referidas. A classe "ServerWorker" vai ser corrida nas *threads* criadas pelo servidor para cada conexão e é responsável por receber e processar pedidos dos clientes. Esta última serve-se de um estado partilhado, representado pela sua fachada "ScooterManagerI", que garante o processamento concorrente das operações e é totalmente independente.

3 Conexão

Uma vez que este sistema vai suportar diferentes funcionalidades, surge a necessidade de distinguir o comportamento a tomar perante cada pedido. Desta maneira, recorremos à utilização da classe "TaggedConnection" que nos permite adicionar *tags* (identificadores) para os diversos pedidos. Ainda assim, estamos perante um cliente de ambiente *multi-threaded* pelo que precisamos de garantir que os pedidos com uma certa *tag* cheguem às respetivas *threads* no cliente.

De modo a simplificar o processo de envio e receção de mensagens, estas são representadas por *frames* que contêm um *array* de *bytes*, com os dados da mensagem, e uma *tag*, de modo a identificar o tipo de mensagem.

O cliente pode enviar vários pedidos simultaneamente através de várias *threads*. A conexão conta com um desmultiplexador que opera no cliente e é responsável por encaminhar cada *frame* proveniente da conexão para o respetivo *buffer*. Deste modo, a classe assegura que as *threads* "adormecem" e, assim que chega o sinal indicando que a mensagem esperada já foi recebida, "acordam".

4 Protocolo de comunicação

De modo a aumentar a eficiência da transmissão, implementamos o seguinte protocolo de comunicação entre cliente e servidor.

Pedidos

Tag	**Tipo de Mensagem**	**Bytes**	**Descrição**
1	Registo	Variável	username + password
2	Autenticação	Variável	username + password
3	Listagem das trotinetes livres	8	$x + y$
4	Listagem das recompensas	8	$x + y$
5	Reserva de uma trotinete	8	$x + y$
6	Estacionamento de uma trotinete	12	$x + y$ + codReserva
7	Ativar notificações	9	bool + $x + y$
7	Desativar notificações	1	bool

Nota: No pedido com a *tag* 7, o booleano indica se a mensagem tem como objetivo ativar ou desativar os pedidos de notificação, pelo que se tiver o valor *true*, é enviada a posição desejada.

Respostas

Tag	**Tipo de Mensagem**	**Bytes**	**Descrição**
1	Registo	4	response code
2	Autenticação	4	response code
3	Listagem das trotinetes livres	$4 + n \times 8$	comprimento + position[n]
4	Listagem das recompensas	$4 + n \times 16$	comprimento + reward[n]
5	Reserva de uma trotinete	$4 + 8$	codReserva + posicao
6	Estacionamento de uma trotinete	8	custo / recompensa
7	Receber notificações	$5 + n \times 16$	bool + comprimento + reward[n]

Nota: Na resposta com a *tag* 7 foi adicionado um booleano que indica se o cliente tem ainda as notificações ativadas. Isto é útil para sabermos, do lado do cliente, quando o servidor não vai enviar mais notificações relativas a um pedido anterior do cliente, para pararmos a leitura de dados vindos do servidor.

5 Implementação

5.1 Estado partilhado

De modo a dar resposta às funcionalidades pedidas, criamos a classe "ScooterManagerI" que satisfaz as diversas operações que um cliente pode executar. Após um estudo do enunciado do trabalho prático, concluimos que seriam necessárias estruturas para guardar toda a informação do sistema, como o estado dos utilizadores, das trotinetes, das recompensas e das reservas. Portanto, são criadas quatro estruturas, cada uma com o seu respetivo *lock* de modo a garantir a exclusividade mútua.

5.2 Estado de sessão

De modo a evitar que o cliente, após ter sido devidamente autenticado, envie novamente o seu *username* para o servidor em futuras invocações remotas, adicionamos uma String "clientUsername". Esta é inicializada quando o cliente regista-se ou autentica-se e mais tarde utilizada em operações como reservar trotinetes, ou receber notificações. Consideramos que esta é uma constante (a escrita acontece apenas uma vez e no início), pelo que não foi necessário usar primitivas de exclusão mútua no acesso à mesma.

5.3 Registo e autenticação do utilizador

Os processos de registo e autenticação do utilizador no servidor são muito semelhantes. Para suportar estas duas operações, temos, no estado partilhado, um mapa de utilizadores, em que cada utilizador é identificado por um *username* e uma *password*. De modo a garantir a exclusivade mútua na estrutura, recorremos à utilização de um *lock*. Assim, para a operação de registo, começa-se por adquirir o *lock*, adicionando-se, de seguida, um novo utilizador à coleção, caso este não exista. A operação de autenticação é uma simples operação de consulta na estrutura. Caso o utilizador já exista e a *password* seja igual à fornecida pelo cliente, a operação terá sucesso. Em casos de erros nestas operações são lançadas exceções que irão ser tratadas pelo servidor e, posteriormente, transmitidas ao cliente.

Primeiramente, partimos para uma solução mais básica e recorremos a um *ReentrantLock* para garantir a exclusividade na estrutura. No entanto, após várias considerações decidimos utilizar um *ReadWriteLock*, uma vez que à partida temos mais operações que necessitam de ler da estrutura do que escrever. Deste modo, as operações de *login* passam a poder ser executadas em simultâneo.

5.4 Listagem e reserva de trotinetes livres

O estado partilhado do servidor é ainda constituído por uma lista de trotinetes atualmente disponíveis. Cada uma destas trotinetes guarda a sua posição, o seu estado (livre ou ocupada) e um *lock*. A operação de listagem é uma simples travessia pela lista, verificando quais as trotinetes livres que estão num raio D da posição indicada. A operação de reserva de uma trotinete livre num raio D de p começa por calcular a trotinete mais próxima de p, dentro de um raio D. O *lock* de cada trotinete verificada é adquirido e, consequentemente, libertado, caso esta não seja a trotinete escolhida. Isto assegura que dois clientes que executam concorrentemente não reservam a mesma trotinete. Após ter sido escolhida a trotinete mais próxima, é criada uma nova reserva, sendo armazenada no servidor até a mesma ser estacionada. A *thread* associada à geração de recompensas é também notificada no fim desta operação.

5.5 Estacionamento de uma trotinete

Ao ser recebido o código de reserva e a posição final de uma trotinete, o sistema atualiza a posição e o estado da trotinete que foi estacionada (assegurando o lock respetivo) e calcula o valor total a pagar. Para além disso, é verificado se existe alguma recompensa com as mesmas posições de origem e destino da viagem agora terminada. Para isso, são adquiridos os *locks* das reservas e das recompensas. Assim, garantimos que não há dois clientes a tentarem concluir a reserva ao mesmo tempo, independentemente de essa reserva ser ou não uma recompensa. O lock das recompensas permite-nos ainda sinalizar o estacionamento de uma nova trotinete no mapa.

5.6 Geração e listagem das recompensas

O mecanismo de geração de recompensas é realizado por uma *thread daemon* a executar dentro do servidor. Esta faz uso de uma variável de condição "rewardsCond" associada ao *lock* das recompensas. Esta *thread* é adormecida até que alguma trotinete seja reservada ou estacionada. Como tal, estas duas últimas operações devem chamar a operação "signal", tendo adquirido o *lock* das recompensas. Havendo zonas demasiado congestionadas e zonas com pouca densidade de trotinetes, são criadas recompensas das zonas mais densas para zonas menos densas, levando a uma melhor distribuição da frota. Ao serem geradas novas recompensas, devem também sinalizadas as *threads* de gestão de notificações dos utilizadores, pelo que também é adquirido o *lock* das notificações.

5.7 Pedido de notificação de recompensas

Ao solicitar ao servidor o envio de notificações, um cliente cria indiretamente uma nova *thread* no servidor que espera, numa variável de condição associada ao *lock* das notificações, que novas recompensas sejam geradas em torno da posição fornecida. Esta *thread* vai correr infinitamente, até que o utilizador desligue as notificações. O processo de envio de novas recompensas ao cliente tem ainda em conta as recompensas que este já recebeu anteriormente, pelo que as recompensas

que cada utilizador já recebeu terão de ser guardadas. Assim, só são enviadas novas notificações assim que uma recompensa nova é gerada.

6 Teste das funcionalidades no servidor

De modo a testar a exclusividade mútua e a ausência de *deadlocks* no estado partilhado do servidor, criamos um módulo de testes à parte com o intuito de testar, de uma forma simulada, o envio concorrente de pedidos dos clientes.

```
int threadNum = 10;
Thread t[] = new Thread[threadNum];

for (int i=0; i<threadNum; i++){
    t[i] = new Thread(() -> {
        Reservation r=null;
        try{
            r = sm.activateScooter(new Position( x: 1, y: 1),  username: "miguel");
        } catch(Exception e){
            System.out.println(e.getMessage());
        }
        if (r != null) System.out.println(r.toString());
    });
    t[i].start();
}

for(int i=0; i<threadNum; i++){
    t[i].join();
}

System.out.println("Terminei");
```

Figura 2: Exemplo de teste criado.

Por exemplo, o código acima testa a reserva simultânea de várias trotinetes. Sabemos que o programa não entrou em nenhum *deadlock* se todas as *threads* terminarem, ou seja, se for imprimido "Terminei" no ecrã. Foram criados vários testes para cada uma das operações permitidas e um último teste que gera operações aleatórias concorrentemente. É importante realçar que o facto de que um destes programas terminar não é sinónimo de controlo de concorrência no nosso programa. Para esse teste, é preciso uma análise mais profunda dos resultados obtidos, com observação de possíveis estados incoerentes.

7 Trabalho futuro

Neste trabalho, a estrutura utilizada para guardar as trotinetes disponíveis foi uma lista. Isto traz algumas desvantagens em termos de desempenho para algumas operações, como por exemplo a de listagem das trotinetes livres num raio D de uma posição p, pois é necessário percorrer a lista toda a fim de encontrar as trotinetes com esse critério. Uma solução mais eficiente passaria pelo uso de uma matriz que em cada posição guardasse o número de trotinetes estacionadas naquele local. No

entanto, esta alteração iria requerer a alteração das primitivas de exclusão mútua anteriormente existentes, sendo uma possível solução o uso de um lock global para a matriz.

Uma outra otimização que poderia ser feita encontra-se no processo de geração de recompensas. Neste momento, quando ativo, este gera uma nova lista de recompensas com base no estado atual do mapa. Seria possível, pois, que, através da sinalização de uma só mudança no estado do mapa, fossem atualizadas apenas as recompensas relativas à vizinhança dessa atualização.

8 Conclusão

Em suma, neste trabalho, procuramos aplicar os conhecimentos de programação concorrente e sistemas distribuídos adquiridos nas aulas para desenvolver uma aplicação de gestão de uma frota de trotinetes elétricas. Acima de tudo, procuramos garantir a correção do programa, tentando que este apresente em todos os momentos um estado coerente. Tentamos também, em alguns momentos, melhorar a eficiência do mesmo, diminuindo o tempo passado em secções críticas ou, por exemplo, diminuindo o número de *threads* "acordadas" por um *signal*. Fazemos, por isso, uma avaliação positiva do trabalho desenvolvido, na medida em que este foi um bom complemento para o estudo e aprofundar de Sistemas Distribuídos.