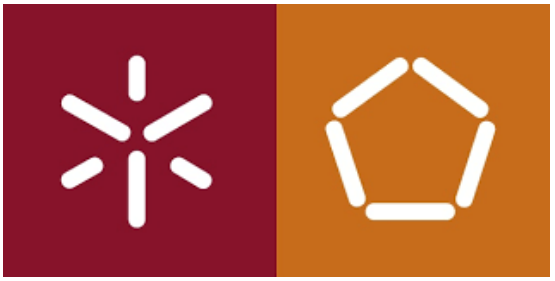
Universidade do Minho



Sistemas Distribuídos

Trabalho Prático

Gestão de Frota

Ano letivo 2022/2023

a96106, Miguel Silva Pinto

a97755, Orlando Palmeira

a97613, Pedro Miguel Castilho Martins

1. Introdução

Este trabalho foi elaborado no âmbito da unidade curricular de **Sistemas Distribuídos** onde nos foi proposto a implementação de uma plataforma de **gestão de uma frota** de trotinetes elétricas, sob a forma de um par cliente-servidor em Java utilizando sockets e threads.

O serviço permitirá aos utilizadores reservar e estacionar trotinetes em diferentes locais do mapa. Este mapa é uma matriz de N × N locais, sendo as coordenadas geográficas pares discretos de índices. Para a distância entre dois locais considera-se a distância de Manhattan. De maneira a que haja uma boa distribuição das trotinetes pelo mapa, existe um sistema de recompensas em que os utilizadores são premiados por levarem trotinetes estacionadas num local A para um local B. Em cada momento existe uma lista de recompensas, atualizada ao longo do tempo, sendo cada recompensa identificada por um par origem-destino, com um valor de recompensa associado, calculado em função da distância a percorrer.

2. Arquitetura do sistema

2.1. Servidor

O servidor inicializa um socket TCP com o número de porta “12345” conhecido por todos os clientes. De seguida o servidor guarda em memória toda a informação inicial de que precisa para a gestão do mapa, bem como outras estruturas de dados que o auxiliarão na gestão dos pedidos provenientes dos clientes, como por exemplo, informação sobre as contas registadas, geração de códigos de reserva, etc.

A cada cliente que estabelece uma conexão com o servidor, é criada uma thread que irá estar encarregue de responder aos pedidos desse cliente.

Mapa de trotinetes: A implementação do mapa de trotinetes foi feita através de uma matriz de objetos Localizacao que representam uma coordenada do mapa e contêm informação acerca do número de trotinetes que existem naquele local.

Na **gestão de concorrência do mapa**, tomamos a decisão de ter **ReadWriteLocks** em cada Localizacao do mapa. Decidimos não colocar um lock relativo ao mapa inteiro, uma vez que existem várias operações que não precisam de aceder ao mapa todo, apenas precisando de aceder a certas posições. Decidimos usar ReadWriteLocks, uma vez que prevemos muitas mais operações de leitura do que de escrita. Prevemos isto pois uma reserva/estacionamento é uma operação de escrita que irá aceder a uma só posição e devido ao mecanismo de geração de recompensas, cada uma destas escritas leva a uma operação de leitura que irá aceder a várias localizações (atualização da lista de recompensas em vigor no mapa). Além disso, ainda há a ter em conta eventuais pedidos de listagem de trotinetes numa área que aumenta a proporção de operações de leitura relativamente às operações de escrita.

Geração de recompensas: Para a geração de recompensas, é criada uma thread que corre em background no servidor que irá determinar as recompensas em vigor no momento. Para tal, ela analisa o mapa de trotinetes e gera uma lista de todas as recompensas em vigor. Esta análise é feita sempre que ocorre uma mudança no mapa, ou seja, uma reserva ou estacionamento de trotinete. Na prática este sistema é implementado através do uso de **Condition’s** em que após a análise inicial do mapa, a thread entra em

bloqueio até receber um sinal, que indica a reserva ou estacionamento de uma trotinete, voltando a fazer a análise para determinar a nova lista de recompensas do momento.

2.2. Cliente

O cliente conecta-se ao servidor através do socket aberto pelo servidor com a porta “12345”. Após a conexão com o servidor, um menu é apresentado para autenticação do cliente, apresentando as opções de “Registar nova conta” ou “Iniciar sessão”.

Com a autenticação concluída com sucesso, é apresentado ao cliente um outro menu que apresenta todas as funcionalidades do programa. Na aplicação cliente, estas funcionalidades apenas se baseiam em enviar pedidos ao servidor, e receber a sua resposta, apresentando-as devidamente ao cliente.

Uma vez que a aplicação cliente tem de receber notificações e receber respostas do servidor, o cliente é **multi-threaded**, com uma thread responsável por fazer pedidos e receber respostas do servidor e uma thread que esteja à escuta de notificações, e a lógica de gestão destes dois tipos de mensagem é feito pela classe **Demultiplexer** que é explicada em melhor detalhe na secção seguinte que explica a comunicação entre cliente e servidor.

2.3. Comunicação entre cliente e servidor

Na comunicação entre estas duas entidades são usados **sockets TCP** que são as vias de comunicação para que a informação seja transmitida de um componente para o outro.

Para esta parte da transmissão de informação, foram criadas as seguintes classes e interfaces:

- A nossa interface **Serializavel** é implementada pelos objetos que são enviados entre servidor e cliente através dos métodos serialize() e deserialize() que, respetivamente, serializa e desserializa os objetos para serem transmitidos por um DataStream. Um exemplo destes objetos enviados, que implementam esta interface, são objetos da classe **Pair** que contém as coordenadas de uma posição (para verificar a existência de trotinetes nas redondezas, para reserva de uma trotinete naquele local, etc).
- A classe **Frame** é constituída por uma **tag** e um **objeto que implementa a interface Serializavel**. A tag é um inteiro que serve para etiquetar a mensagem de maneira a que seja direcionada para o seu correto processamento. O objeto que implementa a interface Serializavel é o objeto que foi transmitido pelo emissor e que juntamente com a tag dá sentido à mensagem, como por exemplo, a receção de uma Frame com a tag 2 e um Pair que indica a coordenada (2,1) indica ao servidor que se trata de um pedido de listagem dos locais onde existem trotinetes livres nas redondezas da coordenada.
- A classe **Connection** é uma classe que contém os métodos send(Frame) e receive() que abstraem a troca de mensagens entre o servidor e o cliente.
- A classe **Demultiplexer** encapsula uma Connection e, para além de delegar o envio de mensagens para esta, disponibiliza a operação receive(int tag), que bloqueia a thread invocadora até chegar uma mensagem com a etiqueta especificada,

devolvendo então o conteúdo. Esta classe é particularmente útil no cliente para que este consiga distinguir entre notificações e respostas de pedidos ao servidor.

▪ **Cliente → Servidor**

Quando um cliente quer enviar dados para o servidor invoca o método send() da classe Demultiplexer, que por sua vez chama também o método send() da classe Connection. Com estes métodos, os dados vão ser encaminhados no socket juntamente com a sua tag, para que o servidor possa identificar o tipo de mensagem que recebe.

Para o servidor receber os dados apenas invoca o método receive da classe Connection que retorna um objeto da classe frame que inclui a tag e os dados enviados pelo cliente.

▪ **Servidor → Cliente**

Para enviar respostas para o cliente, o servidor invoca o método **send()** da classe **Connection**, que escreve no socket, uma **tag** e os **dados da mensagem**. Uma vez que o cliente é **multi-threaded**, pois irá estar à escuta de **notificações** e de **respostas** do servidor, foi preciso organizar o recebimento de mensagens no cliente. No início da sua execução, é invocado o construtor da classe **Demultiplexer**, que vai criar uma thread responsável por capturar todas as mensagens enviadas pelo servidor para o socket e organizá-las num **mapa** de inteiros(tags) para objetos que contêm uma variável de condição e uma Queue onde serão depositadas as mensagens com uma determinada tag/etiqueta.

Assim, quando uma thread do cliente quer receber os dados relativos a uma determinada tag, basta invocar o método **receive()** da classe Demultiplexer que vai procurar os dados no mapa que tem todos os dados recebidos organizados. Se a fila estiver vazia, a thread vai ficar à espera que cheguem dados através do método await da Condition relativa à tag.

Deste modo, o cliente consegue distinguir **notificações,** de **respostas** provenientes do servidor. O servidor para indicar que a mensagem é uma **resposta** utiliza a **tag 0**, para indicar que é uma **notificação** usa a **tag 30**.

3. Funcionalidades

3.1. Autenticação e registo de utilizador

Quando a aplicação cliente é inicializada, é estabelecida uma conexão ao servidor e aparece ao utilizador um menu com as seguintes opções: **(1) Registar nova conta** e **(2) Iniciar sessão**.

```
*** LOGIN ***

Pretende:
1) Registar nova conta.
2) Iniciar sessão.

Opção:
```

Figura 1 - Menu de Autenticação.

Em ambos os casos são pedidos um username e password para que o utilizador se autentique. O utilizador escolhe “Registar nova conta” se não tiver nenhuma conta registada previamente, caso contrário, escolhe a opção “Iniciar sessão” introduzindo as suas devidas credenciais. Caso um utilizador queira iniciar sessão numa conta que já está a ser utilizada por outro utilizador, o sistema impede a sua entrada.

Para indicar ao servidor um pedido de registo de nova conta, o cliente envia uma Frame com a **tag 0**, e para um pedido de início de sessão leva a **tag 1**. Juntamente com a tag, estes pedidos levam também um objeto **AccountInfo** que contém o nome de utilizador e a respetiva password, que ao chegarem no servidor, verifica se a operação é possível. Após ser feita a verificação, o servidor responde com um objeto da classe Mensagem que consiste apenas num byte que indica o sucesso ou os diferentes tipos de insucesso que ocorreram nesta operação.

Caso a mensagem que recebe indique insucesso na autenticação, é apresentado ao utilizador o motivo da falha e o utilizador é questionado novamente se pretende registar nova conta ou iniciar sessão. No caso do cliente receber um indicativo de sucesso, é permitido ao utilizador aceder ao resto das funcionalidades, apresentando um novo menu com as restantes funcionalidades.

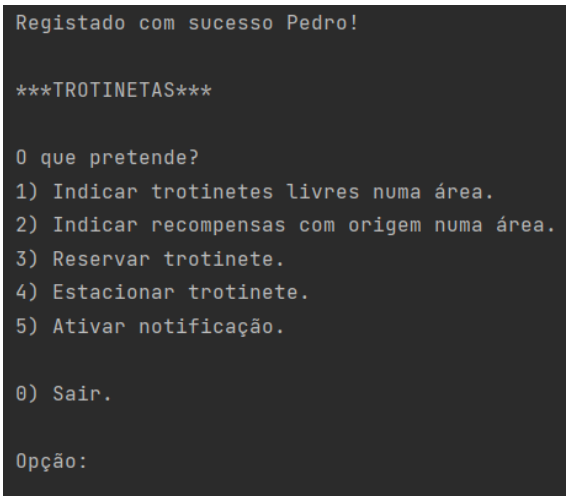


Figura 2 - Menu de funcionalidades.

3.2. Listagem das trotinetes numa área

O utilizador, ao selecionar a opção 1, é-lhe pedido que forneça um par de números discretos, indicando as coordenadas centrais da área que pretende saber da existência de trotinetes livres. O utilizador fornece as coordenadas que vão ser transformadas num objeto da classe Pair, que será enviada para o socket numa Frame com a **tag 2**, indicando que se trata de um pedido de probing de trotinetes numa área. O servidor recebe essa frame com a tag 2 e o Pair indicando as coordenadas que o cliente pretende procurar por trotinetes e procede a obter essa informação ao chamar o método *TrotinetesArround(Pair p)* que retorna um objeto da classe **PairList** que é uma classe que extende a classe “ArrayList<Pair>” e implementa a interface ***Serializavel*** para que seja possível o envio do objeto para o cliente. Este PairList vai chegar ao cliente, contendo uma lista dos sítios onde existem trotinetes livres, sendo mostrados esses valores ao utilizador através do standard output.

3.3. Listagem das recompensas numa área

Na opção 2, é novamente pedido o fornecimento de um par de números discretos, indicando as coordenadas do local que quer obter a informação. O utilizador fornece essas coordenadas que vão ser transformadas num objeto da classe Pair, que será enviada para o socket numa **Frame** com a **tag 3**, indicando que se trata de um pedido de probing de recompensas numa área. O servidor recebe essa frame com a tag 3 e o Pair indicando as coordenadas que o cliente pretende procurar por recompensas e procede a obter essa informação à lista de recompensas em vigor do momento e retorna um objeto da classe **RecompensaList** que é uma classe que extende a classe “HashSet<Recompensa>” e implementa a interface ***Serializavel*** para que seja possível o envio do objeto para o cliente. A classe **Recompensa** é uma classe que tem três atributos, um Pair origem, um Pair destino e um inteiro que representa o valor da recompensa. O cliente vai receber esse objeto RecompensaList, que contém o conjunto das recompensas com origem num raio de 2 unidades do local definido pelo utilizador e indica os resultados obtidos ao utilizador.

3.4. Reserva de trotinete

Para esta funcionalidade, será pedido ao cliente um par de coordenadas que indique um local onde o cliente pretende reservar uma trotinete num raio de 2 unidades.

Com as coordenadas fornecidas, é enviada uma Frame para o cliente com a **tag 4** e com as respetivas coordenadas (no objeto Pair), indicando ao servidor que o cliente pretende fazer a reserva de uma trotinete.

O servidor verifica no seu mapa e retorna um objeto da classe **CodigoReserva** que tem como atributos um inteiro que é o **código de reserva** da trotinete reservada e um **Pair** que indica a localização de onde foi reservada a trotinete. Se num **raio de 2 unidades** do local indicado não existir nenhuma trotinete, nenhuma trotinete é reservada para o cliente e é enviado o código de reserva **-1** indicando o insucesso na reserva da trotinete. Caso contrário, é reservada a trotinete mais próxima do local indicado.

Após a informação chegar, o cliente indica se a reserva de uma trotinete foi bem sucedida e indica o código de reserva para ser utilizado no estacionamento da trotinete.

3.5. Estacionamento de trotinete

No estacionamento de trotinete é pedido o local de estacionamento e o código de reserva da trotinete que foi reservada. O cliente envia num objeto **CodigoReserva** estas informações com a **tag 5** para o servidor tratar de estacionar a trotinete com aquele código e naquela localização. Após o servidor processar os dados recebidos, este envia um objeto da classe **InfoViagem** que contém informação sobre a viagem como a origem, o destino, a distância, a duração, o custo e uma eventual recompensa associada à viagem que é verificada na altura do estacionamento.

3.6. Pedidos de notificação

Um cliente quando pede para ser notificado quando aparecem recompensas com origem a menos de uma distância fixa de 2 unidades de determinado local, é-lhe pedido um par de números discretos indicando a coordenada do local onde quer receber notificações.

O cliente envia uma Frame com a **tag 6** e com a localização fornecida no formato de um objeto **Pair**. O servidor ao receber este pedido verifica se a posição é válida e se o cliente já não pediu uma notificação nesta localização e no caso de passar todas as verificações, o servidor cria uma nova thread que se encarrega de enviar notificações para o cliente ao enviar uma Frame com a **tag 30**(tag utilizada para notificações) com uma **RecompensaList** indicando um conjunto das novas recompensas com origem na área que o cliente indicou.

A thread responsável por enviar a notificação primeiramente guarda uma lista das recompensas em vigor na altura em que o pedido de notificação ocorreu e de seguida bloqueia a thread ao chamar o método await() na Condition do objeto Localizacao que representa a coordenada no mapa. Esta thread fica então à espera de receber um sinal para que depois envie a notificação para o cliente, e este sinal virá do mecanismo de geração de recompensas. Este mecanismo de geração de recompensas corre em background no servidor e sempre que há uma alteração no mapa, este cria uma nova lista de recompensas. Quando este mecanismo deteta o aparecimento de uma nova recompensa, ele envia sinais para as redondezas num raio de 2 unidades da origem da nova recompensa, ou seja, invoca o método signalAll() nos objetos Localizacao referentes a estas coordenadas.

Após receber o sinal, a thread acorda e vai verificar as novas recompensas e trata de enviar ao cliente uma frame com a tag 30 e um objeto RecompensaList com todas as recompensas novas que surgiram na área e volta a entrar em bloqueio à espera do sinal de novas recompensas.

Gostaríamos de sublinhar que esta parte das notificações poderia ser mais otimizada na medida em que é criada uma nova thread para cada pedido de notificação válido. Uma possível maneira de reduzir o número de threads criadas, seria ao criar uma única thread adicional por cada conexão estabelecida (ou noutras palavras, por cliente) que iria ser a gestora de todas as notificações que o cliente pedisse, sendo esta thread responsável ter uma fila de notificações a enviar para o cliente em que o mecanismo de geração de recompensas escreveria as novas recompensas na fila e acordaria a thread, para que enviasse o que estivesse na sua fila. Desta maneira, continuávamos a impedir que threads oriundas de outros clientes escrevessem no socket de outro cliente e reduzíamos o número de threads utilizadas. Não implementamos esta técnica uma vez que nos faltou tempo e a maneira que implementamos continua a satisfazer os requisitos.

4. Conclusão

Este trabalho permitiu-nos familiarizar com os conceitos aprendidos nas aulas, tendo sido utilizadas todas as ferramentas ensinadas nas aulas, desde a gestão de concorrência entre threads, até a programação em sockets para estabelecer uma conexão entre cliente e servidor.

Neste relatório procuramos explicar o nosso projeto, descrevendo a arquitetura do nosso sistema, bem como os seus requisitos e funcionalidades foram implementados.

Em suma, apesar de haver alguns aspetos que gostaríamos de otimizar, acreditamos que o nosso projeto cumpre todas as funcionalidades pedidas no enunciado estando satisfeitos com o resultado final.