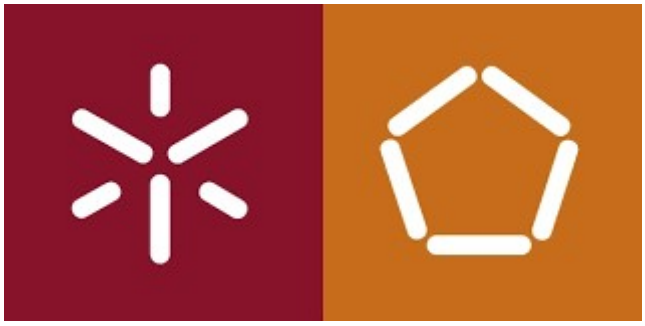

Universidade do Minho

Departamento de Informática

# *Frota de Trotinetes Elétricas*

## Sistemas Distribuídos

## Grupo 11


Afonso Magalhães- A95250

Gonçalo Rodrigues- A90439

Artur Leite- A97027

Diana Teixeira- A97516

## Introdução

Este trabalho consiste num desenvolvimento de um programa capaz de gerir uma frota de trotinetes elétricas. Para tal, será utilizado um par cliente-servidor em Java recorrendo também a *sockets* e *threads.*

Por questões de gestão eficiente, é essencial permitir que os clientes sejam capazes de reservar e estacionar trotinetes em diferentes locais. Estes locais serão determinados num mapa de grelha N*N, onde as coordenadas geográficas são identificadas por um par de inteiros, como (1,6) ou (15,8). Para além disso, com o intuito de incentivar uma boa distribuição de trotinetes pelo mapa, será implementado um sistema de recompensas que irá premiar os clientes que estacionem as trotinetes em locais que não as possuem.

## Cliente

Entidade capaz de comunicar com o Servidor. Pode efetuar registo no sistema, introduzindo username e password. Estes dados são então enviados para o servidor que, depois de os verificar, armazena-os para que possa ser efetuado o *login* posteriormente sem ter que se registar novamente. No processo de *login,* ao receber as credenciais introduzidas, o servidor executa um processo de verificação onde irá averiguar se os dados são válidos, informando o cliente se este foi terminado com sucesso ou não.

Na eventualidade do cliente se autenticar com sucesso, irá desbloquear um conjunto de funcionalidades que requeriam uma conta para poderem ser acedidas. Estas são:

- **Listar localizações com trotinetes a uma distância D:** funcionalidade que pede ao cliente que introduza o X e o Y do ponto que deseja. De seguida, irá verificar todos os pontos presentes num raio de distância D, listando todos aqueles que possuem uma trotinete estacionada;
- **Reserva de trotinete mais próxima a uma distância D:** funcionalidade que pede ao cliente que introduza o X e o Y do ponto que deseja. Após isso, irá verificar a existência de uma trotinete disponível mais próxima, limitado a uma distância D. Caso seja encontrada uma trotinete, a verificação retorna um número inteiro positivo que irá corresponder ao código de reserva. Caso não seja encontrada uma trotinete, a verificação retorna -1 e o cliente é informado do acontecimento.
- **Estacionamento de uma trotinete:** assim que o cliente estacionar a trotinete, é introduzido o código de reserva, juntamente com as coordenadas X e Y do ponto de chegada. O sistema calcula o custo em função da distância percorrida. Finalmente, verifica se a viagem está qualificada para receber uma recompensa de acordo com a lista em vigor no momento do estacionamento.
- **Listar recompensas com origem até distância D de um ponto:** funcionalidade que pede ao cliente que introduza o X e do Y do ponto que deseja. Após isso, irá verificar a existência de recompensas possíveis no raio em questão, sendo os resultados representados em formato de lista para análise do utilizador.
- **Receber notificações de Recompensas com origem até distância D de um ponto:** funcionalidade que os clientes podem ou não aderir, indicando o ponto desejado. A partir deste momento, será criado uma *thread* tanto na parte do servidor como no

cliente, encarregue de, sempre que existir uma alteração a ser notificada, sejam enviados os pontos atualizados e recebidos os mesmos pontos nas *threads* respetivas.

## Servidor

Com o intuito de garantir que todos os comportamentos necessários do servidor estão corretamente implementados, optamos por desenvolver um conjunto de classes java, cada uma descrevendo uma função essencial do programa.

- **Recompensa:** classe que engloba os métodos de *setters e getters* das variáveis referentes às recompensas, nomeadamente ponto de origem, ponto de destino e o valor da recompensa em questão.
- **GestaoRecompensas:** classe que engloba os métodos utilizados para obter os valores das recompensas. Estão aqui presentes as funções que atualizam as Recompensas de acordo com o estado do mapa atual, bem como formas de obter todas estas que estão relacionadas com o percurso atual;
- **Parque:** classe que engloba todas as variáveis referentes à informação no parque das trotinetes, destacando o número total de trotinetes e o conjunto de vizinhos relativos a cada ponto. Engloba também métodos que atualizam estas respetivas informações de acordo com o desenvolvimento do programa, ou seja, quando um cliente reserva ou estaciona uma trotinete.
- **GestaoReservas:** classe que inclui os métodos que definem o comportamento do programa no processamento de registo de utilizadores, bem como os processos de login e logout. Adicionalmente, possui na sua definição métodos que definem o comportamento do programa face a estacionamento de trotinetes e reservas.
- **ServerWorker:** classe que engloba a parte worker do servidor que lida com a conexão com o cliente, através de uma *thread* para que não bloqueie outras conexões com o servidor por parte de outros clientes.
- **User:** classe que engloba os métodos de *getters e setters* das variáveis referentes às informações que correspondem a contas dos utilizadores, nomeadamente nome e palavra passe.
- **Viagem:** classe que engloba os métodos essenciais no processo de uma Viagem por parte de um utilizador, juntamente com os parâmetros das mesmas. Estes incluem ponto de partida, ponto de chegada, tempo do começo, duração da viagem e custo.

## Implementação de Locks

Para evitar problemas de exclusão mútua, usamos locks em algumas classes. Isto deve-se ao facto destas classes se tratarem de instâncias onde diversos métodos fazem operações de leitura e de escrita de forma concorrente.

Uma das estratégias que achamos importante foi utilizar um ReadWriteLock para o mapa de users. O facto de se tratar de um ReadWriteLock e não um lock normal, deve-se ao facto de que na maior parte dos métodos implementados na lógica de negócio nunca alteravam o respetivo mapa, à exceção do registo de utilizadores.

Outra decisão importante, foi não incluir um lock ao nível dos parques, visto que estes só eram acedidos quando efetuávamos um lock da gestão de recompensas. Isto deve-se ao facto de se tratarem de operações que devem ser atómicas, ou seja, uma atualização das recompensas nunca deve ocorrer ao mesmo tempo que um estacionamento/reserva num parque do mapa.

Adicionalmente, decidimos reduzir ao máximo o número de classes cujas instâncias poderiam ser modificadas, ou seja, constantes, levando desta forma a uma redução da necessidade de locks. A titulo de exemplo, a classe Ponto.

## Sistema de Recompensas

Para o sistema de recompensas, decidimos incluir uma classe mencionada anteriormente, a GestaoRecompensas, encarregue de efetuar atualizações e gestão de notificações das recompensas.

Para o primeiro caso, optamos possuir uma *thread,* que está constantemente à espera de alterações no mapa, para que verifique possíveis alterações nas recompensas atuais do sistema. Com o intuito de reduzir o número de pontos verificados, decidimos incluir uma lista de pontos vizinhos em cada parque, para que numa alteração, apenas estes sejam verificados, visto que a mudança num certo ponto a mais de D de distância não afetará a presença de recompensas no ponto alterada.

Tomando agora em conta o segundo caso das notificações, decidimos incluir um mapa de ponto para uma variável de condição na qual as *threads* encarregues de esperar por notificações num ponto possam invocar o método await na mesma, fazendo com que, desta forma, as *threads* acordadas serão apenas aquelas que estão à espera de recompensas nesse ponto e nos seus vizinhos, semelhante ao caso de atualizações, reduzindo o número de *threads* acordadas.

## Conexão Servidor-Cliente

Para uma boa conexão entre os elementos essenciais do nosso programa, implementamos a utilização de *sockets* TCP e *threads.* Através dos sockets, os clientes são capazes de estabelecer conexão com o servidor, podendo então ser efetuados todos os pedidos. Isto porque assim que a conexão é estabelecida, torna-se possível a troca de mensagens utilizando os *streams* de entrada e saída do *socket.* Adicionalmente, com o intuito de garantir que diversos clientes possam ser atendidos em simultâneo, utilizamos as *threads,* onde cada uma vai executar os pedidos efetuados por um cliente, relativo sempre a um cliente e aos seus pedidos específicos.

## Frame

Para simplificar o processo de enviar e receber mensagens, aliado ao facto de possuirmos um cliente *multi-threaded,* havendo a necessidade de redirecionar os pedidos para a *thread* correta, estas são representadas por *frames.* Uma frame contém uma *tag,* um identificador do pedido do cliente que enviou a respetiva mensagem e própria mensagem em si. Neste programa, a *tag* é um valor numérico inteiro. A seguinte tabela demonstra o que cada valor *tag* representa:

| Tag | Tipo de Mensagem |
|---|---|
| 1 | Registo de utilizador |
| 2 | Login de Utilizador |
| 3 | Listagem de pontos com trotinetes a D de distância |
| 4 | Reserva de uma trotinente a menos de D de distância de um determinado local |
| 5 | Estaciona uma trotinete num determinado local |
| 6 | Listagem de recompensas com origem até D de distância de um ponto |
| 7 | Receber atualizações de recompensas com origem a menos de D de distância de um ponto |
| 8 | Deixar de receber atualizações de recompensas com origem a menos de D de distância de um ponto |
| 10 | Atualização recebida |

## Conclusão

Este trabalho permitiu-nos consolidar os conhecimentos obtidos nas aulas de sistemas distribuídos de forma interessante e interativa. Conseguimos averiguar a forma como aplicações como a *Bolt* ou *Circ* funcionam de forma eficiente recorrendo ao uso de implementações semelhantes às nossas, tomando em consideração é claro, numa escala muito maior. Devido a este facto, acreditamos que este conhecimento possa vir a ser extremamente útil no nosso futuro académico e principalmente profissional.

Para além disso, acreditamos que o sistema beneficiava imenso de uma implementação de um método de localização GPS, de forma a otimizar a precisão e coerência do sistema de recompensas e de reservas. Para tal, seria necessário um local de armazenamento para esta quantidade exorbitante de informação, algo que seria muito interessante numa implementação em larga escala.

Concluindo, acreditamos ainda que fomos capazes de implementar um sistema eficiente e conciso, capaz de atender a todos os requisitos pedidos no enunciado. Esperamos que a implementação de recompensas esteja bem conseguida, visto que foi a parte aonde encontramos mais dificuldades.