

# [Sistemas Distribuídos] Trabalho Prático

Filipe Felício  
A85983

Luís Pereira  
A96681

Rui Oliveira  
A95254

Tiago Pereira  
A95104

Janeiro 2023

## 1 Introdução

O presente relatório é referente ao trabalho prático da Unidade Curricular de Sistemas Distribuídos da Licenciatura em Engenharia Informática da Escola de Engenharia da Universidade do Minho, no ano de 2022/2023; tendo sido realizado por Filipe Felício (A85983), Luís Pereira (A96681), Rui Oliveira (A95254) e Tiago Pereira (A95104).

O objetivo do trabalho prático é a implementação de um serviço de gestão de uma frota de trotinetes que siga o modelo cliente/servidor em Java. As principais decisões tomadas pelo grupo serão aqui devidamente apresentadas e justificadas.

O relatório começa com um manual de utilização, que explica como interagir com os diferentes componentes do sistema. De seguida, explica-se o modelo comunicacional utilizado entre cliente e servidor. De seguida, o funcionamento do servidor e do cliente é detalhado. Por fim, apresenta-se uma conclusão ao projeto, que inclui potencial trabalho futuro.

### 1.1 Manual de Utilização

#### 1.1.1 Servidor

O servidor necessita de diferentes parâmetros para rodar. Para isso, para rodar o servidor, é necessário passar os seguintes argumentos, por esta ordem.

- N: este número determina o comprimento do lado da grelha quadrangular em que as trotinetes são guardadas
- D: a constante referida no enunciado
- SCOOTERS: o número de trotinetes a ser aleatoriamente colocado pela grelha

Resumidamente, para iniciar o servidor roda-se o comando

```
java -classpath target/classes server.Main <N> <D> <SCOOTERS>
```

Depois de iniciado, o servidor regista na consola clientes a conectar-se e desconectar-se, pedidos recebidos e respetivas respostas enviadas, e a subrotina de geração de recompensas. Não exige qualquer tipo de *input* do utilizador.

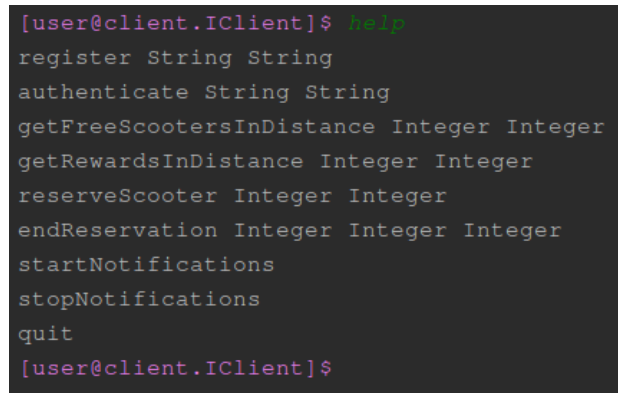
### 1.1.2 Cliente

O cliente não necessita de qualquer argumento para ser executado, ou seja, pode ser rodado através do comando

```
java -classpath target/classes Main
```

Ao rodar este comando, aparece uma interface semelhante a uma *shell*. Para rodar o comando, basta escrever o nome do comando e passar os seus argumentos, da mesma forma que se utiliza, por exemplo, um interpretador de *bash*.

Para listar todos os comandos disponíveis, usa-se o comando *help*. Exemplo de *output* do comando encontra-se na figura 1.



```
[user@client.IClient]$ help
register String String
authenticate String String
getFreeScootersInDistance Integer Integer
getRewardsInDistance Integer Integer
reserveScooter Integer Integer
endReservation Integer Integer Integer
startNotifications
stopNotifications
quit
[user@client.IClient]$
```

Figura 1: Exemplo de *output* do comando *help*

## 2 Comunicação Entre Servidor e Cliente

### 2.1 Mensagens

A comunicação entre cliente e servidor é feita através do envio de objetos de subclasses da classe abstrata *Message*. A serialização / desserialização de mensagens é realizada através de chamadas aos métodos estáticos *Message::serialize* e *Message::deserialize*, que recebem *DataOutputStream* e *DataInputStream*, respetivamente.

Esta solução garante uma API consistente entre os diferentes tipos de mensagem, particularmente aquando da desserialização de mensagens, em que o tipo de mensagem não é conhecido *a priori*. Cria, no entanto, um problema, que consiste no seguinte: como é que a superclasse consegue desserializar objetos que são instâncias de subclasses suas? A resposta é simples: não sabe. O que a classe *Message* sabe é delegar essa tarefa para a subclasse respetiva. Para isso ser possível, cada subclasse *regista-se* perante a superclasse *Message*.

O registo consiste em chamar o método de classe *Message::registerSubClass* no construtor estático da subclasse. Isto originou problemas relativamente ao momento em que o construtor era executado, dado que poderia ser preciso desserializar um objeto da respetiva subclasse antes de esta ter sido alguma vez instanciada. Para dar a volta a este problema, foi implementado um método que utiliza reflexão, que instancia uma mensagem de cada tipo no início do programa (cliente e servidor).

O registo é efetuado com um inteiro constante fixado no código-fonte<sup>1</sup>. Esse identificador é enviado sempre antes do conteúdo da mensagem. Assim, ao serializar / desserializar, a superclasse apenas precisa de ler/escrever esse identificador, e delegar a tarefa à subclasse correspondente àquele inteiro. Para garantir que todas as subclasses definem métodos de serialização / desserialização, criaram-se métodos de instância abstratos protegidos responsáveis pela serialização / desserialização da subclasse.

## 2.2 Conexões

Para garantir o devido funcionamento do serviço em clientes com várias *threads* (veremos em diante o porquê deste requisito), foi utilizada uma implementação de TaggedConnection do lado do servidor, e Demultiplexer semelhante àquela descrita nas aulas teórico-práticas da UC.

De forma a minimizar o impacto de clientes lentos no servidor, cada cliente é atendido na sua *thread* particular.

## 3 Servidor

### 3.1 Estrutura do servidor

#### 3.1.1 Dados

A camada de dados do servidor consiste principalmente nos objetos correspondentes às entidades do sistema e às suas coleções. O controlo de concorrência, como será explicado mais a seguir, é feito ao nível das coleções, pelo que as classes que definem entidades do sistema não precisam de estar preocupadas com esse aspeto. Apesar disso, algumas classes (User, Location e Reward) foram criadas de forma a serem imutáveis.

Existem várias coleções de objetos que expõem a principal funcionalidade associada às entidades do sistema. A forma como foram implementadas, nomeadamente ao nível da exclusão mútua, será explicada mais à frente. As coleções que existem são

- UserCollection
- RewardCollection
- ReservationCollection
- ScooterCollection

#### 3.1.2 Fachada

Toda a funcionalidade que o servidor oferece está exposta através da classe ServerFacade. Desta forma, qualquer alteração efetuada ao estado interno do servidor deve ser feita através desta classe. Todos os métodos que esta classe disponibiliza garantem exclusão mútua, o que torna a sua API bastante fácil de utilizar.

---

<sup>1</sup>Para simplificar o processo de escolha do identificador, e para evitar colisões acidentais, está a usar-se o *timestamp* UNIX do momento da implementação da classe

### 3.1.3 Comunicação

A comunicação é tratada pelas classes `Server`, que escuta por pedidos de conexão TCP, e delega o tratamento das conexões de clientes à classe `ClientHandler`, que roda numa *thread* separada por cliente, para mitigar o impacto de clientes lentos no serviço. Cada `ClientHandler` gere ainda a sua *thread* de notificações, que são separadas dos restantes pedidos para poder ler atualizações do servidor independentemente dos pedidos do cliente.

### 3.1.4 Tratamento de Pedidos

De forma análoga à serialização de mensagens, foi desenvolvido um sistema modular para processamento de mensagens. Definiu-se uma interface funcional `IMessageHandler`, que deve ser implementada pelas classes que processam mensagens. Cada classe deve apenas processar um e um só tipo de mensagem, que deve ser indicado aquando do seu registo perante a classe `ClientHandler`. Desta forma, para adicionar processamento a um novo tipo de pedido, basta criar uma classe nova para o processar e este passa automaticamente a ser processado.

## 3.2 Controlo de Concorrência

Para efeitos de controlo de concorrência, duas estratégias diferentes foram adotadas: uma usada para as recompensas, utilizadores e reservas; e outra utilizada em exclusivo para as trotinetes.

### 3.2.1 Trotinetes

A estratégia de controlo de concorrência de trotinetes é diferente das restantes, uma vez que estas não foram implementadas como um objeto explícito. Em vez disso, representou-se o mapa como uma grelha  $N \times N$ , que é dividida em quadrados mais pequenos - denominados de *chunks* -, com tamanho  $2D \times 2D$ <sup>2</sup>.

Desta forma, minimiza-se a contenção sempre que se pretende procurar por trotinetes próximas de uma dada posição no mapa. Como a distância máxima de procura  $D$  é fixa, esta divisão garante que, sempre que se quer procurar uma trotinete a partir de um dado ponto  $(x, y)$ , apenas é necessário bloquear no máximo 4 *chunks*, em vez de bloquear o mapa todo. Como  $N$  é muito superior a  $D$ , isto é muito vantajoso. De forma a eliminar *deadlocks* associados a uma má ordem de obtenção de *locks*, convencionou-se que estes seriam adquiridos / libertados começando pelo canto superior esquerdo e avançando para a direita e para baixo (semelhante à forma como a língua portuguesa é lida).

Os *locks* em questão são `ReentrantReadWriteLocks`, o que permite ter várias operações de procura de trotinetes a correr simultaneamente, desde que não haja tentativas de reserva / colocação de uma trotinete nos respetivos *chunks*. Para adquirir os *locks* necessários a uma operação relativa a uma dada posição, definiu-se o método `ScooterCollection::lockLocation`, assim como os métodos para libertar o *lock*, e métodos para bloquear / desbloquear toda a grelha (usado para geração de recompensas). Estes devem ser invocados antes da operação pretendida e oferecem uma API opaca relativamente ao funcionamento do controlo de concorrência interno da coleção.

### 3.2.2 Restantes Entidades

A estratégia de controlo de concorrência das restantes entidades do sistema é bastante mais simples. É utilizado um `ReentrantReadWriteLock` para a coleção, não havendo *locks* individuais

---

<sup>2</sup>Por motivos de simplicidade, assumiu-se que  $N$  era múltiplo de  $2D$ , de forma a não haver *chunks* incompletos.

nos objetos. Dado isto, e para evitar que seja possível duas *threads* obterem o *lock* de leitura, e ambas modificarem o objeto devolvido, criando, deste modo, uma corrida; as coleções devolvem os objetos sempre por composição, ou seja, é devolvida uma cópia do objeto ao invés da instância guardada na coleção. Isto obriga a que qualquer alteração aos objetos tenha de passar pela API da coleção, o que garante assim exclusão mútua.

### 3.3 Geração de Recompensas

A geração de recompensas é realizada por uma *thread* dedicada, que inicia e roda inicialmente quando o servidor começa. Quando ainda não houve nenhuma alteração à grelha desde a última geração, esta *thread* encontra-se bloqueada à espera de uma dada condição. Quando alguma alteração é realizada, o método que a provocou deve invocar `RewardGenerator::setAwake`, que acorda a *thread* das recompensas, garantindo que estas são geradas novamente.

O algoritmo de geração de recompensas é o seguinte:

1. Determina todas as posições do mapa que não têm nenhuma trotinete
2. Determina todas as posições do mapa que têm duas ou mais trotinetes
3. Para cada uma das posições com mais do que uma trotinete, escolhe aleatoriamente uma posição vazia, e um valor monetário. Gera uma recompensa com esses valores

Desta forma, geram-se  $M$  recompensas, em que  $M$  é o número de posições da grelha com duas ou mais trotinetes. Gerou-se apenas uma recompensa por posição cheia uma vez que, para gerar uma por cada par vazio-cheio, seria necessário um tempo de processamento muito maior - o algoritmo passaria a rodar em tempo quadrático no número de posições de cada um dos conjuntos - e seria, na ótica do grupo, excessivo gerar tantas recompensas.

### 3.4 Notificações

Uma *Notification*, enquanto objeto, é simplesmente um conjunto de *Rewards* que foram geradas simultaneamente pela geração de recompensas. De forma a simplificar a gestão de notificações, estas são guardadas junto das *Rewards* dentro da *RewardCollection* como uma *queue*. Esta *queue*, no entanto, deve suportar operações multi-threaded: cada *ClientHandler* pode ler da *queue* a um ritmo diferente, e alguns podem não ler de todo. Para além disso, só devem ser armazenadas as notificações relevantes para garantir um uso eficiente de recursos.

Para solucionar estes problemas foi implementada a classe *SubscribableQueue*. Esta classe suporta duas operações com controlo de concorrência: colocar novos elementos na *queue* e subscrever a *queue*. Subscrever devolve um objeto *Subscription* que pode ser iterado para obter assincronamente os elementos colocados na *queue* desde o momento da subscrição, e cancelado para terminar a iteração em *threads* paralelas e libertar os recursos associados. Notificações desnecessárias (que já foram lidas por todas as subscrições não-canceladas) são automaticamente apagadas pelo *garbage collector*.

Cada *ClientHandler* obtém uma subscrição quando recebe um pedido de notificações do servidor, e cria uma *thread* auxiliar para ler da subscrição e enviar as notificações para o cliente. Quando recebe um pedido de cancelamento das notificações, o *ClientHandler* cancela a subscrição e a *thread* auxiliar termina normalmente.

Esta solução permite separar elegantemente os aspetos de geração, armazenamento e obtenção de notificações e gere o controlo de concorrência junto dos dados, permitindo uma maior clareza do código nas camadas superiores do programa.

## 4 Cliente

### 4.1 Biblioteca

Para simplificar a implementação do cliente, e abstrair as chamadas ao servidor, criou-se uma biblioteca para utilizar do lado do cliente, completamente independente da implementação do cliente.

Essa camada de abstração é definida pela interface `IClient`, que declara todos os métodos que expõem a funcionalidade que o servidor oferece.

Esta camada está dividida em dois *threads*, ambos iniciados mal o cliente arranca. Um deles é o fio principal de execução: é este o *thread* responsável por ler o *input* do utilizador, fazer os pedidos remotos ao servidor, processar e imprimir a resposta.

A segunda *thread* é responsável por receber as notificações de recompensas do servidor. Porquê a necessidade de esta funcionalidade se encontrar na sua *thread* separada? Porque o programa cliente não estar bloqueado enquanto espera por novas notificações, uma vez que isso tornaria a sua utilização impossível, dado que esse seria sempre o caso.

A necessidade de mais do que uma *thread* no cliente levou à necessidade de utilização de uma `TaggedConnection` do lado do servidor e de um `Demultiplexer` do lado do cliente. Todos os pedidos que não correspondem a notificações levam 1 como a sua *tag* (à exceção do pedido para começar a ser notificado, que leva 2). As notificações do servidor vêm com *tag* igual a 2.

### 4.2 Programa Cliente

O programa cliente consiste numa aplicação simples com interface via terminal, em que o utilizador pode inserir os comandos que pretende executar, não sendo guardado qualquer tipo de estado ou informação.

A implementação deste programa foi uma adaptação de um projeto<sup>3</sup> desenvolvido anteriormente por um dos elementos do grupo para outra Unidade Curricular. Resumidamente, é utilizada reflexão sobre a interface que define a funcionalidade do servidor - neste caso `IClient` - para gerar todos os comandos passíveis de ser executados, assim como para imprimir as respostas do servidor. Não sendo o foco principal deste projeto, os detalhes mais finos da sua implementação não serão aqui discutidos.

## 5 Conclusão e Trabalho Futuro

De forma geral, o grupo está bastante satisfeito com o resultado do trabalho, apesar da inevitável existência de aspetos que poderiam ser melhorados em iterações futuras do projeto, nomeadamente a interface com o cliente. O algoritmo de geração de recompensas é bastante lento a executar e, por isso, poderá ser otimizado futuramente.

Para concluir, o grupo achou particularmente valiosa a experiência adquirida sobre o controlo de concorrência em projetos complexos, nomeadamente em formas de o encapsular dentro das camadas de dados.

---

<sup>3</sup>Link para o repositório do projeto