Universidade de Évora
Departamento de Informática
Sistemas Operativos II
Ano letivo 2019 - 2020

# Sistema de Partilha de Disponibilidades e Necessidades

Alunos:
Luís Ressonha - 35003
Rúben Teimas - 39868

Docente:
José Saias

26 de Abril de 2020

# Indice

# 1 Introdução

Neste 1º trabalho da *UC* de Sistemas Operativos II foi-nos pedido que implementássemos um Sistema de Partilha de Disponibilidades e Necessidades para uma comunidade local.

Esse sistema deve permitir ao utilizador efetuar 3 operações:

- **Reportar uma disponibilidade**: o utilizador reporta ao servidor em como encontrou um produto $X$ numa loja $Y$;
- **Verificar a disponibilidade de um produto**: o utilizador efetua um pedido ao servidor com o objetivo de verificar se o produto $X$ se encontra disponível. Na resposta, o servidor indica a(s) loja(s), data/hora do ultimo reporte de disponibilidade e em que quantidade o produto $X$ se encontra disponível;
- **Reportar uma necessidade**: o utilizador efetua um pedido ao servidor para que este o notifique caso seja reportada uma nova disponibilidade de um produto $X$;

Para a implementação deste sistema tentámos abstrair-nos ao máximo dos detalhes de comunicação entre o cliente e servidor, utilizando *Java RMI*, uma solução de *Middleware* abordada nas aulas.

Para guardar os dados do servidor de forma persistente utilizámos uma Base De Dados *PostgresSQL*. Para guardar a última informação recebida e enviada pelo cliente considerámos utilizar um repositório embebido, tendo ponderado usar *HyperSQL*, usado na aula, ou *SQLite*, mas acabamos por não usar nenhum deles, dado que não implementámos este *feature*.

# 2 Desenvolvimento

Para uma melhor organização do trabalho optámos por separar as classes em 2 pacotes: o pacote *client* e o pacote *server*.

Os nomes são auto-descritivos, tendo o objeto remoto, a interface remota e o *connector* à BD ficado do pacote *server*. As restantes classes, que dizem respeito às necessidades do cliente, ficaram no pacote *client*, tal como é possivel observar na figura 1.


Figura 1

Para a base de dados do servidor criámos 3 tabelas: a tabela **disponibilidade**, **necessidade** e **produto**. A tabela produto acabou por não ter grande importância no resultado final, dado que o produto não possui mais nenhuma informação senão o seu nome, embora a ideia inicial fosse ter uma tabela **produto** com a descrição detalhada do mesmo, sendo a chave primária um *Id* único que iria ser posteriormente usado nas restantes tabelas.

Uma das primeiras dificuldades com as quais nos depararmos foi a forma como organizar a informação de disponibilidade e como a mostrar ao utilizador. Após alguma discussão optámos por criar uma chave primária complexa para a tabela **disponibilidade** formada pelo par (*produto*, *nome_loja*) sendo que caso fosse introduzido na tabela esse mesmo produto nessa mesma loja, a informação dessa *entry* seria atualizada.

Assim sendo, a forma como optámos por apresentar a disponibilidade de um produto ao utilizador foi retornando todas as linhas que continham esse produto, sendo que o resultado da query possui o *produto*, a *loja*, o *stock*, o *dia* e a *hora*.
Dessa forma o utilizador fica a conhecer em que altura foi reportada a disponibilidade do produto podendo depois avaliar essa informação, dado que é muito difícil garantir informações exatas em serviços geridos por uma comunidade.

Tivemos também algumas dificuldades em enviar notificações ao utilizador a partir do servidor com a arquitetura de *Java RMI*, contudo, após alguma pesquisa encontrá-

mos no livro *Distributed Systems Concepts and Design* o conceito de *Callbacks*, o que se tornou bastante importante.

Usando *callbacks* foi-nos possível enviar ao servidor uma referencia para os clients que comunicam com o objeto remoto. Isto permitiu-nos que ao ser introduzida uma disponibilidade o servidor conseguisse verificar que clientes esperavam que esse produto estivesse disponível e enviar-lhe uma notificação.

De forma a identificar os clientes atribuimos-lhe também um *ID* sendo que estes poderiam identificar-se caso já o possuíssem ou esperar que o servidor lhes atribuísse. O identificador de cliente foi criado a pensar no envio de notificações a 100% e na funcionalidade de guardar num repositório embebido as ultimas informações recebidas pelo cliente.

Para esta ultima funcionalidade, ao ligar-se, o cliente iria obter a ultima informação recebida que estaria guardada na base de dados. Caso não existisse uma base de dados local com o *ID* do cliente, esta seria criada com as respetivas tabelas.

# 3 Instruções de Execução

Para facilitar a execução do trabalho criámos um ficheiro *Makefile*. Para correr este trabalho são necessárias, pelo menos, 3 instâncias do terminal.

Antes de executar qualquer comando é necessário ter uma Base de Dados em *PostgresSQl* onde se deve correr o script *BD.sql*, encontrado na pasta base, que é responsável pela criação das tabelas.

A ordem pela qual os makes devem ser executados é a seguinte:

- make A0=regPort
- make rs A0=regPort A1=regHost A2=dbName A3=user A4=password
- make rc A0=regPort A1=regHost A2=id

Cada um destes comandos deve ser executado numa diferente instãncia de terminal.

O 1º comando compila os ficheiros e executa o serviço de nomes a escutar no porto que será passada ao argumento *A0*.

O 2º comando corre o servidor. O número do porto passado ao servidor deve ser o mesmo passado no comando acima, onde o *binder* está ativo.

O 3º comando corre o cliente. O o porto que lhe é passado deverá ser o mesmo que é passado ao *binder*. O argumento *A2* é opcional pois é o argumento que recebe o *ID* do cliente. Este 3º comando deve ser corrido por cada cliente que se queira executar.

# 4 Conclusão

Ainda que numa 1ª vista o trabalho parecesse bastante simples, rapidamente se revelou trabalhoso.

O trabalho foi parcialmente concluído com sucesso, contudo, ficaram por realizar algumas funcionalidades que com um melhor planeamento teriam sido facilmente implementadas.

Ficou por implementar a funcionalidade que, em caso de falha, a aplicação cliente deve reter os últimos dados de forma persistente. Esta iria permitir o cliente receber todas a notificações que deveria ter recebido enquanto desconectado assim como realizar eventuais pedidos ao servidor que não tenham sido concluídos com sucesso.

A juntar-se ao planeamento deficiente, devido à difícil gestão de tempo com outras UC's, apontamos também o facto de estarmos a trabalhar com uma tecnologia relativamente recente para nós (*Java RMI*) que ainda que tenhamos usado nas aulas práticas e compreendamos o funcionamento da mesma, exigiu uma mudança na forma como programamos habitualmente.

Fazendo um balanço do trabalho consideramos que foi positivo, pois deu para vermos a tecnologia a funcionar não se cingindo unicamente a conhecimento teórico, mas reconhecemos que ficou um pouco aquém das expectativas e do que são as nossas capacidades.

# 5 Referências

https://www.sqlitetutorial.net/sqlite-java/
https://www.postgresql.org/docs/9.5/
https://docs.oracle.com/javase/8/docs/api/