

# Tolerância a Faltas

## Trabalho Prático

-

Mestrado em Engenharia Informática  
Universidade do Minho

### Grupo nº 8

---

PG41080	João Ribeiro Imperadeiro
PG41081	José Alberto Martins Boticas
PG41091	Nelson José Dias Teixeira

1 de Junho de 2020



# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Descrição do problema e requisitos</b>	<b>4</b>
<b>3</b>	<b>Implementação</b>	<b>5</b>
3.1	<i>Middleware</i> genérico . . . . .	5
3.1.1	<i>ServerConnection</i> . . . . .	5
3.1.1.1	Replicação . . . . .	6
3.1.1.2	Temporizadores . . . . .	6
3.1.1.3	Comunicação entre servidores . . . . .	7
3.1.1.4	Comunicação com clientes . . . . .	7
3.1.2	<i>ClientConnection</i> . . . . .	7
3.2	Servidor - <i>Supermarket</i> . . . . .	7
3.2.1	Funcionamento . . . . .	7
3.2.2	<i>CartSkeleton</i> . . . . .	7
3.2.3	<i>CatalogSkeleton</i> . . . . .	7
3.3	Cliente . . . . .	7
3.3.1	Funcionamento . . . . .	7
3.3.2	<i>CartStub</i> . . . . .	7
3.3.3	<i>CatalogStub</i> . . . . .	8
<b>4</b>	<b>Valorizações</b>	<b>9</b>
<b>5</b>	<b>Conclusão</b>	<b>10</b>
<b>A</b>	<b>Observações</b>	<b>11</b>

# Lista de Figuras

# Capítulo 1

## Introdução

Toda a gente, de uma forma ou de outra, já esteve em contacto com uma loja *online*. Muitos podem mesmo dizer que já dependem deste tipo de serviços para efetuar as suas compras. Por não estarem diretamente relacionadas com uma localização física, estas lojas estão disponíveis para todos, independentemente de onde se encontre fisicamente no mundo.

Assim, levantam-se alguns problemas relacionados com a implementação destes tipos de serviços, como por exemplo a oferta de um serviço com bom desempenho para todos os clientes, independentemente da sua localização física. Isto leva à necessidade de não depender de apenas um servidor central, distribuindo a disponibilização do serviço por diversos servidores. Ora, isto leva a que seja necessário ter cuidados extra nas interações com os clientes, como a manutenção da consistência entre os servidores, para que, no caso em que um destes servidores falhe, o cliente não seja afetado negativamente.

Posto isto, é-nos proposta a implementação, em **Java**, de uma versão simplificada de um supermercado *online* distribuído por vários servidores, que por sua vez seja o mais tolerante a faltas possível. Para isso, entre outras ferramentas, será utilizado o protocolo *Spread* para comunicação em grupo, à semelhança do que se sucedeu ao longo das aulas da componente prática desta unidade curricular.

Relativamente à estrutura do presente documento, é descrito de forma mais detalhada a proposta deste projeto, fazendo-se referência aos requisitos intrínsecos à mesma. De seguida, são exibidos todos os aspetos referentes à implementação deste trabalho. Nesta fase, são especificados todos os tópicos alusivos ao cliente, servidor e, ainda, ao *middleware* genérico da aplicação. Posteriormente, são evidenciadas todas as valorizações tomadas em consideração no desenvolvimento do mesmo. Por fim, são extraídas algumas conclusões da realização deste trabalho, sumariando, globalmente, os objetivos alcançados.

## Capítulo 2

# Descrição do problema e requisitos

Tal como foi mencionado no capítulo introdutório deste documento, o objetivo principal deste projeto prático consiste na implementação em **Java**, usando o protocolo de comunicação em grupo *Spread*, de um serviço tolerante a faltas.

Este serviço diz respeito a um supermercado *online* que disponibiliza algumas funcionalidades aos seus clientes. Entra elas destacam-se:

- criar uma encomenda;
- iniciar uma compra;
- consultar o preço e disponibilidade de um produto;
- acrescentar um produto a uma determinada encomenda;
- confirmar uma encomenda, indicando se foi concretizada com sucesso.

O serviço guarda um catálogo contendo uma descrição de cada produto e a quantidade disponível. Cada encomenda inclui um ou mais produtos, sendo que só pode ser concretizada com sucesso se todos os produtos estiverem disponíveis. Admite-se que existe um tempo limite **TMAX** para a concretização de uma encomenda. Caso esse tempo seja esgotado e a encomenda ainda não tenha sido efetuada, é cancelada. Embora seja indesejável, admite-se também que uma encomenda pode ser cancelada unilateralmente pelo sistema.

Quanto aos requisitos da aplicação são impostos os seguintes:

- par cliente/servidor da interface descrita, replicado para tolerância a faltas;
- permitir o armazenamento persistente do estado dos servidores na base de dados *HSQLDB*;
- transferência de estado para permitir a reposição em funcionamento de servidores sem interrupção do serviço;
- implementação de uma interface simplificada para o utilizador, de forma a testar o serviço em causa;

Tendo em conta todos os pontos referidos acima, procede-se agora à implementação da proposta inerente a este projeto prático.

## Capítulo 3

# Implementação

De seguida serão abordadas a forma adotada para implementar várias vertentes deste trabalho. Será apresentado o middleware genérico desenvolvido para servir de base para este trabalho, bem como o uso deste num par cliente/servidor que atinge os objetivos cumpridos.

### 3.1 *Middleware* genérico

Se for excluída a lógica de negócio relacionada com o funcionamento do supermercado, temos na essência deste trabalho um conjunto de servidores ligados entre si que gerem uma base de dados em HSQLDB (por recomendação do enunciado). Para além disto, todos os servidores são réplicas perfeitas uns dos outros, ou seja, todos aplicam as mesmas alterações às suas bases de dados, o que significa que, na prática, é obtida uma base de dados distribuída por eles. Os papéis dos servidores são simétricos, sendo que qualquer um deles poderá receber pedidos de clientes interessados no serviço disponibilizado.

Posto isto, foi tomada a decisão de implementar esta arquitetura genérica em classes independentes da lógica de negócio propriamente dita, dando origem a duas classes: *ServerConnection* e *ClientConnection*. A primeira é responsável por gerir o conjunto de servidores, sendo iniciada uma instância desta em cada um destes. A segunda é utilizada por clientes para interagir com os servidores de forma transparente.

Seguidamente, será detalhada cada uma destas classes.

#### 3.1.1 *ServerConnection*

Como referido atrás, a classe *ServerConnection* é responsável pela coordenação entre os servidores, gerindo a base de dados de cada um deles e recebendo pedidos de clientes. Para iniciar um destes servidores é necessário indicar a porta onde este estará à espera de pedidos de clientes, o que fazer após ser estabelecida a ligação à base de dados, uma lista de tabelas a ser criada na base de dados, os handlers para lidar com pedidos do cliente e o processamento das atualizações à base de dados.

Quando é iniciado, o servidor conecta-se ao daemon Spread e prepara-se para a receção de mensagens vindas dos restantes servidores do cluster. Após isto, coloca-se num de estado de espera, o qual pode ser interrompido de duas formas:

- se for o primeiro servidor a juntar-se ao cluster, inicializa a base de dados e cria as tabelas especificadas pelo utilizador do middleware. Depois, executa uma função que é fornecida pelo utilizador, dando-lhe oportunidade de inicializar objetos que dependam de uma ligação à base de dados;

- caso contrário, fica à espera de receber uma vista da base de dados de um dos servidores que já estão no cluster, o que será detalhado posteriormente. Este servidor não é escolhido ao acaso, dado que é assumida a noção de um servidor primário que, na prática, é o mais antigo do cluster. Enquanto não recebe esta informação, vai acumulando todos os pedidos enviados pelos outros servidores, de forma a poder quando esta é recebida.

Em ambos os casos é inicializada a forma de comunicação com os clientes que, no nosso caso, recorre à framework Atomix, à qual são passados os handlers fornecidos pelo utilizador, os quais estão relacionados com a lógica de negócio.

Voltando ao envio do conteúdo da base de dados para os novos membros, são feitos checkpoint's periódicos da base de dados, de forma a que, no caso de um dos servidores se desconectar e conectar passado algum tempo, ser possível enviar apenas as alterações feitas desde o último checkpoint, em vez da base de dados completa. Isto torna-se mais relevante em bases de dados de maior dimensão.

#### **3.1.1.1 Replicação**

Quanto ao tipo de replicação escolhido para resolver o problema em mãos, acabou por ser adotada a replicação passiva, dado que, por um lado, o cliente só necessita de comunicar com um servidor (o que facilita a sua implementação) e, por outro, a sincronização necessária é muito simplificada pelo uso da framework Spread.

Quanto à implementação em si, é necessário que o utilizador do middleware forneça os handlers que serão chamados quando chega um pedido do cliente e o que fazer quando chega uma atualização para a base de dados.

Quando chega um pedido de um cliente, é chamado o respetivo handler e é devolvido um HandlerRes pelo utilizador, que informa o middleware se a informação é para ser enviada diretamente para o cliente (por exemplo, no caso de um GET) ou, no caso de ser necessário fazer alterações à base de dados, se é para passar pelo mecanismo de replicação, isto é, enviada a todos os membros do cluster atonicamente, utilizando o Spread.

Depois disto, quando esta informação é entregue aos servidores, é invocada a função fornecida pelo utilizador correspondente àquele pedido. Esta função poderá efetuar qualquer operação, mas o mais normal será enviar uma atualização para a base de dados. Pelo uso do Spread, é garantido que esta atualização ocorrerá "em simultâneo" em todos os servidores.

Por fim, é enviada uma resposta ao servidor, se o utilizador assim pretender, e esta resposta será enviada pelo servidor que recebeu o pedido correspondente àquela atualização.

#### **3.1.1.2 Temporizadores**

Como referido acima, os servidores efetuam checkpoints periódicos. Para isso, necessitam de uma primitiva que lhes permita enviar uma mensagem atonicamente para o servidor ao fim de um determinado período de tempo. Para isso, é criada uma thread em cada servidor, que no final do tempo indicado pelo utilizador, envia uma mensagem para todos os membros do cluster. Note-se que isto é apenas feito pelo servidor primário, ou seja, é garantido que ocorre apenas uma vez. Há ainda a possibilidade fazer isto repetidamente, como é o caso dos checkpoints.

#### 3.1.1.3 Comunicação entre servidores

#### 3.1.1.4 Comunicação com clientes

### 3.1.2 *ClientConnection*

## 3.2 Servidor - *Supermarket*

### 3.2.1 Funcionamento

### 3.2.2 *CartSkeleton*

### 3.2.3 *CatalogSkeleton*

## 3.3 Cliente

O cliente desta aplicação é responsável por interagir com o *cluster* de servidores, invocando ao mesmo pedidos relacionados com operações do supermercado. Esta entidade permite, de certa forma, testar o programa desenvolvido como um todo, verificando, desta forma, a validade das funcionalidades propostas neste projeto. Para proceder à conceção e implementação do cliente, foram criados vários menus na aplicação por forma a cobrir todos os casos intrínsecos às funcionalidades da mesma.

### 3.3.1 Funcionamento

O cliente inicialmente, no momento da conexão ao *cluster*, especifica as portas referentes ao próprio e, também, as relativas aos servidores, sendo necessário indicar pelo menos uma das portas destes últimos. A informação relativa à porta de cada servidor é guardada numa lista para o caso de haver falhas no estabelecimento da conexão entre o cliente e este último. Assim, havendo a indisponibilidade momentânea de um dos servidores, é atribuído outro para o efeito.

Uma vez estabelecida a conexão entre o cliente e um dos servidores presentes no *cluster*, é apresentado à primeira entidade um menu relativo ao supermercado. Nele estão contidas todas as opções indispensáveis para efetuar as operações descritas no enunciado deste trabalho prático. Destas operações destacam-se a consulta do catálogo da aplicação, a criação de um carrinho de compras e ainda a verificação da disponibilidade de um determinado produto. É de realçar também que existe um menu alusivo ao carrinho de compras onde se pode concretizar o *checkout* dos produtos selecionados. Para além destas, o grupo oferece uma opção extra relativa ao menu do administrador, onde se possibilita a invoção de métodos associados à atualização do catálogo do supermercado.

### 3.3.2 *CartStub*

No que toca ao *stub* do carrinho de compras do presente programa disponibilizam-se 3 métodos distintos, nomeadamente *updateProduct*, *checkout* e *getProducts*. Tal como os nomes dos mesmos transparecem, estas funções dizem respeito à atualização de um determinado produto, ao *checkout* dos mesmos e, ainda, à obtenção dos produtos presentes no carrinho. Todos estes partilham o mesmo objeto *serializer*, responsável por serializar as mensagens enviadas para um dos servidores da aplicação. Para além disso, estão também associados a esta classe **Java** um identificador referente ao carrinho de compras em causa e um objeto relativo à conexão do cliente ao *cluster* de servidores (*ClientConnection*).



### 3.3.3 *CatalogStub*

Quanto ao *stub* do catálogo, este assemelha-se ao do carrinho de compras a nível de variáveis de instância. Esta classe contém igualmente o serializador de mensagens referido na secção anterior e, ainda, o objeto correlacionado à conexão do cliente ao conjunto de servidores. Contudo, apesar destas semelhanças, esta classe Java possui, obviamente, funções distintas da mencionada anteriormente. O *stub* do catálogo apresenta métodos inerentes não só à sua consulta como também à observação de informações intrínsecas aos produtos contidos no próprio (como por exemplo, o preço e a quantidade). Mais, por forma a atualizar os dados dos produtos em questão, existe também a possibilidade de invocar métodos alusivos à adição e remoção dos mesmos.

## Capítulo 4

# Valorizações

Relativamente ao que é referido no enunciado deste trabalho, são recomendadas algumas valorizações que beneficiam a nota final do mesmo. Das valorizações mencionadas, os elementos que compõem este grupo optaram por realizar as seguintes:

1. separação do código relativo ao *middleware* genérico de replicação do código alusivo à aplicação;
2. garantia do tratamento concorrente de varias operações;
3. suporte de partições do grupo na ferramenta computacional *Spread*;
4. realização de uma análise de desempenho;
5. minimização das encomendas canceladas como consequência de faltas ou do funcionamento do mecanismo de replicação;
6. atualização oportuna do estado dos servidores com recurso ao sistema de base de dados, diminuindo, sempre que possível, o volume da informação copiada.

## Capítulo 5

## Conclusão

# Apêndice A

## Observações

- Documentação *Java* 8:  
`https://docs.oracle.com/javase/8/docs/api/`
- *Maven*:  
`https://maven.apache.org/`
- *Spread toolkit*:  
`http://www.spread.org/index.html`
- *Atomix*:  
`https://atomix.io/`
- *HSQLDB*:  
`http://hsqldb.org/`