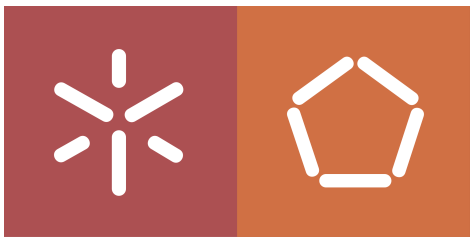
Tolerância a Faltas
Trabalho Prático
-
Mestrado em Engenharia Informática
Universidade do Minho

Grupo nº 8

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

10 de junho de 2020



Conteúdo

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 descrita 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 é abordada a forma como são implementadas as várias vertentes deste trabalho. Será apresentado não só o *middleware* genérico desenvolvido para servir de base a 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 mesma 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 processá-los 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 *checkpoints* 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* atomicamente, 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 atomicamente 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 de fazer isto repetidamente, como é o caso dos *checkpoints*.

3.1.1.3 Comunicação entre servidores

Apesar de já ter sido bastante explicado nas secções anteriores, expõe-se agora a *framework* utilizada para efetuar a comunicação entre os servidores. Esta comuni-

cação é essencial para o funcionamento correto do sistema, visto que, por um lado, é necessário o envio da base de dados quando um servidor novo se junta ao *cluster* (ou, no caso de desconexão, volta a conectar). Por outro, é a razão pela qual a vista global do sistema é consistente em todos os servidores, dado que todos efetuam as operações dos clientes pela mesma ordem.

Para permitir isto, recorremos à *framework Spread*, que foi extensivamente utilizada nas aulas. Esta permite a formação de grupos de servidores, aos quais podem ser enviadas mensagens, tendo a garantia, através das configurações corretas, que as mensagens são entregues pela mesma ordem a todos. Para além disto, existe um tipo especial de mensagens, as mensagens de *membership*. Quando são recebidas, tal como as anteriores, a sua entrega é logicamente simultânea em todos os servidores. As informações que trazem são, por exemplo, a junção de um servidor a um grupo (o que ajuda no envio da base de dados a este), a saída de um servidor de um grupo e a ocorrência de uma partição (o que ajudará a resolver uma das valorizações).

3.1.2 *ClientConnection*

Para o estabelecimento de ligações aos servidores por parte dos clientes foi utilizada a *framework Atomix*. Esta permite o envio de mensagens entre dois utilizadores. O recetor de uma mensagem tem de registar um *handler* para um certo tipo de mensagem, caraterizado por uma palavra. Quem envia, necessita de especificar o destino da mensagem e o tipo da mesma. Quando a mensagem é enviada, esta tem de ser serializada pelo emissor e, aquando da sua receção, desserializada pelo recetor.

Uma das funcionalidades mais importantes desta *framework* é a possibilidade de enviar mensagens assincronamente. Isto permite o envio de uma mensagem assincronamente, adiando a sua receção para uma altura posterior. No caso específico deste trabalho, criámos um método auxiliar que permite enviar uma mensagem e retornar apenas quando é recebida uma resposta. Apesar de já existir um método semelhante na *framework*, os nossos requisitos não eram compatíveis com o que este nos fornecia.

Assim, para inicializar um objeto desta classe, ou seja, para comunicar com um *cluster* de servidores, é necessário passar um endereço onde o cliente estará à escuta e os endereços dos servidores disponíveis no *cluster*. Apenas um método é exposto para o utilizador desta classe, aquele referido acima que envia uma mensagem e recebe uma resposta. Para utilizá-lo, é apenas necessário fornecer o tipo da mensagem e a mensagem serializada, isto é, sob a forma de uma *array* de *bytes*. Quando a resposta é recebida, o método retorna, também, um *array* de *bytes* que terá de ser desserializado. Para obtermos este comportamento, fizemos uso da primitiva *CompletableFuture*. Uma última caraterística deste método é a resiliência à falha de servidores. Para isso, se o envio de uma mensagem falhar, é feito o seu envio para outro servidor. Se nenhum estiver ativo, a execução é terminada com um erro.

3.2 Servidor - *Supermarket*

A classe `Supermarket` representa o comportamento de cada um dos servidores, sendo nesta classe que se estabelece as ligações à base de dados e aos outros servidores, através do *middleware*.

A ligação à base de dados é feita de vários pontos e sempre através de *skeletons*. Assim, existe um *skeleton* para cada carrinho de compras e ainda um outro para o catálogo. Desta forma, invoca-se apenas o *skeleton* necessário a cada ação, sem que seja necessário ter total conhecimento dos seus atributos.

3.2.1 Funcionamento

A classe `Supermarket` é responsável pela criação inicial da base de dados, pelo que cria sempre três tabelas:

- `cart` - tabela com todos os carrinhos de compras, identificados unicamente.
- `product` - tabela com todos os produtos disponíveis; cada produto tem um identificador, nome, descrição, preço e quantidade.
- `cartProduct` - tabela com os produtos num carrinho; cada entrada é identificada pelo par (*idCart*, *idProduct*), sendo que cada uma destas componentes é uma referência para uma das outras tabelas; para além disto, contém ainda a quantidade do produto naquele carrinho.

Esta classe recebe ainda todas as comunicações dos clientes e, por isso, tem registados todos os *handlers* necessários, sendo que, a cada pedido, o servidor dirige o mesmo para o *skeleton* respetivo (de um carrinho ou do catálogo).

Dependendo do tipo de pedido recebido, o servidor envia as informações para o *cluster*, em caso de necessidade.

3.2.2 *CartSkeleton*

Esta classe é uma das pontes entre o servidor e a base de dados, modificando, maioritariamente, as tabelas `cart` e `cartProduct`. De realçar que é criado um *skeleton* por cada carrinho de compras criado.

O `CartSkeleton` implementa as *queries* necessárias para responder aos pedidos que envolvam um carrinho de compras:

- consultar os produtos num carrinho;
- adicionar/remover um produto a/de um carrinho;
- apagar um carrinho;
- realizar o *checkout* - retirar, se possível, os produtos do carrinho em *checkout* da tabela `product` e, de seguida, apagar o carrinho.

3.2.3 *CatalogSkeleton*

Esta classe é a outra ponte entre o servidor e a base de dados, modificando, exclusivamente, a tabela `product`.

O `CatalogSkeleton` implementa as *queries* necessárias para responder aos pedidos que envolvam alterações a um produto:

- consultar todos os produtos existentes;
- consultar as informações de um produto do catálogo - nome, descrição, preço e quantidade;
- consultar o preço ou quantidade de um produto;
- adicionar/remover um produto do catálogo disponível;
- adicionar uma quantidade a um produto do catálogo (reposição de *stock*);
- retificar/atualizar as informações de um produto do catálogo - nome, descrição ou preço;

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 realizaram todas as que foram propostas:

1. separação do código relativo ao *middleware* genérico de replicação do código alusivo à aplicação;

   Esta valorização foi atingida pela criação das duas classes `ServerConnection` e `ClientConnection`, que podem ser utilizadas por qualquer aplicação que necessite de um *cluster* de servidores capaz de gerir uma base de dados consistente e capaz de responder a um elevado número de clientes. Para o caso específico deste trabalho, implementámos as classes `Supermarket` e `Client` que fazem uso destas.

2. garantia do tratamento concorrente de varias operações;

   Ao ser permitido que vários clientes enviem pedidos aos servidores concorrentemente, sendo que os `Get` são respondidos imediatamente e apenas as operações que alteram a base de dados terem a necessidade de ser sincronizadas entre servidores (para mantermos a consistência da base de dados), esta valorização foi atingida.

3. suporte de partições do grupo na ferramenta computacional *Spread*;

   Através do tratamento das mensagens de *membership* emitidas pela *framework Spread*, mais especificamente as relativas à ocorrência de partições na rede, temos que este objetivo foi concluído. De uma forma resumida, quando uma destas mensagens é recebida, é verificado, em cada um dos servidores, se este ainda faz parte da partição maioritária. No caso de isto não ser verdade, o servidor é desconectado.

4. realização de uma análise de desempenho;

   A análise de desempenho foi feita numa única máquina virtual, com 4 *cores* (processador i7-8565U) e 8 *GB* de memória. Por isto, não existe a latência que existiria numa situação real em que existiriam várias máquinas diferentes, separadas fisicamente. De qualquer das formas, esta análise tem como objetivo perceber qual o máximo de pedidos a que um servidor consegue responder.

   Foram feitos testes com apenas um servidor, de forma a perceber os limites do sistema. Para pedidos de `Update`, o servidor estabilizou na resposta a cerca de 4000 pedidos por segundo. Para pedidos de `Get`, o servidor estabilizou na resposta a cerca de 9000 pedidos por segundo.

Pelo grande objetivo deste trabalho ser a resiliência a faltas, faz sentido testar o sistema com mais do que um servidor. Por isso, foram feitos testes para um *cluster* de 3 servidores. Para pedidos de `Update`, o cluster estabilizou em cerca de 3300 pedidos por segundo. A quebra em relação aos testes com apenas um servidor pode ser explicada pelo facto de ser necessária a comunicação entre servidores. Para pedidos de `Get`, a intuição é a de que, como este tipo de pedidos não implica a comunicação entre servidores, o limite de pedidos deve ser individual a cada servidor, pelo que neste caso, o limite deverão ser 27000 pedidos por segundo (9000×3).

A limitação de 4 processos disponíveis não permitiu o teste a um *cluster* com 5 servidores, uma vez que a necessidade de partilhar *CPU* entre os mesmos terá impacto (significativo) na *performance* individual de cada servidor e coletiva do sistema. Apesar disto, a expectativa é que um *cluster* com mais servidores não apresentará quebras muito significativas quando comparado com um de 3 servidores.

5. minimização das encomendas canceladas como consequência de faltas ou do funcionamento do mecanismo de replicação;

   Como resultado dos testes efetuados e do conhecimento da implementação deste trabalho, temos muita confiança de que as encomendas nunca serão canceladas unilateralmente pelo sistema devido a faltas ou ao funcionamento do mecanismo de replicação. Temos esta confiança, pela forma como é feita a replicação. Sendo que todos os servidores têm a mesma informação e, no nosso caso, todos executam os temporizadores relativos ao fim das encomendas (enviando uma mensagem de fim do tempo ao *cluster*), as encomendas são apenas canceladas ao fim do tempo previsto, `TMAX`. A única forma de uma encomenda ser cancelada será se todos os servidores do *cluster* forem desconectados, mas isto já é um problema intratável.

6. atualização oportuna do estado dos servidores com recurso ao sistema de base de dados, diminuindo, sempre que possível, o volume da informação copiada.

   Quanto às atualizações dos servidores, optamos por serializar o objeto correspondente pela atualização e enviá-lo a todos os servidores. Na altura de executar esta alteração, é escrita a *query SQL* responsável pela alteração.

   No caso de um servidor se conectar pela primeira vez, efetuamos um *checkpoint* e enviámos-lhe toda a base de dados. No caso de o servidor já ter feito parte do *cluster* e a sua informação for minimamente recente, é apenas enviado o *log* de alterações, que será muito menor em termos de espaço ocupado (isto para bases de dados de grande dimensão). Temos ainda a efetuação de *checkpoints* periódicos, de forma a diminuir ainda mais a informação enviada no caso de um servidor se desconectar e voltar a ligar.

Capítulo 5

Conclusão

Este trabalho permitiu a aplicação dos conhecimentos adquiridos ao longo desta unidade curricular relativos à implementação de um sistema distribuído tolerante a faltas.

No geral, os objetivos foram atingidos, uma vez que foram aplicados todos os requisitos do enunciado inicial, para além de terem sido implmentadas todas as valorizações propostas no enunciado. A implementação foi pensada no sentido de simplificar todos os processos, sem descurar a robustez exigida, nomeadamente no que toca a tolerância a faltas. Desta forma, temos um sistema muito robusto que permite a sua auditoria por qualquer pessoa interessada.

A análise de desempenho e restantes testes levados a cabo indicam que o sistema é eficiente, para além de resiliente, cumprindo os dois principais objetivos à partida.

No conto geral, os resultados são satisfatórios e a evolução do grupo de trabalho evidente.

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/