Universidade do Minho

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Tolerância a Faltas

Bolsa de Valores Distribuída

Isabel Sofia da Costa Pereira A76550
Maria de La Salete Dias Teixeira A75281

30 de Junho de 2019

Conteúdo

1 Introdução

Este trabalho consiste na implementação de uma bolsa de valores distribuída, isto é, os clientes podem consultar os servidores do serviço sobre quais empresas existem e quantas ações possuem para venda, com o intuito de as comprar ou vender as que já possui.

Outro aspeto relevante deste serviço é que este tem de ser tolerante a faltas. Tendo isso em conta implementou-se comunicação em grupo, tirando partido de um dos protocolo de replicação estudados na disciplina.

2 Protocolo de replicação ativa

O protocolo eleito para a resolução do problema foi então o protocolo de replicação ativa, devido ao facto de este utilizar a primitiva *Total Ordem Multicast*, o que permite que exista ordem em todas as mensagens trocadas entre clientes e servidores. Além disso, a falha de um servidor é transparente ao cliente, pois este comunica com o grupo e não com uma réplica especifica, sendo a reintegração de qualquer réplica feita de modo bastante acessível. O único ponto negativo deste protocolo é o facto do cliente ter de esperar que todos os servidores lhe respondam, tarefa dificultada quando esses servidores saem do grupo, no entanto, considerou-se um aspeto menos relevante do que perder a ordem das mensagens e ter de se eleger um novo líder, perdendo-se a transparência.

3 Arquitetura do Sistema

Tal como se pode verificar pela arquitetura apresentada, o serviço desenvolvido é composto por duas entidades principais, sendo estas o **Cliente** e o **Servidor**. O Cliente é composto por duas *threads*, o *ClientStub* e o *ClientStubThread*, e o Servidor, que possui a implementação da bolsa de valores (*StockImp*), idealmente seria composto por um número desejado de *threads Refresher*.

Para além disso, ambas as entidades tiram partido do ***Middleware*** para efetuar a comunicação. Assim, o cliente realiza pedidos ao grupo de servidores pelo *ClientStub*, cada servidor recebe a mensagem e adiciona a uma lista, para uma das suas *threads Refresher* processar e enviar a resposta para o *ClientStubThread*. Para além disso, o próprio servidor envia mensagens aos clientes com informações acerca do grupo de servidores.

Para facilitar a visualização da arquitetura, na imagem apenas se representou um servidor, no entanto, como já foi mencionado, os pedidos dos clientes são encaminhados para o **grupo de servidores** e, consequentemente, recebidos por todos os servidores. Isto é possível graças à utilização do *Spread Toolkit*.

Como será explicado nos tópicos que se seguem, nomeadamente no Multithread e Apreciação Crítica, esta arquitetura não corresponde ao resultado final do serviço, pois apenas foi implementada uma *thread Refresher* por cada servidor.

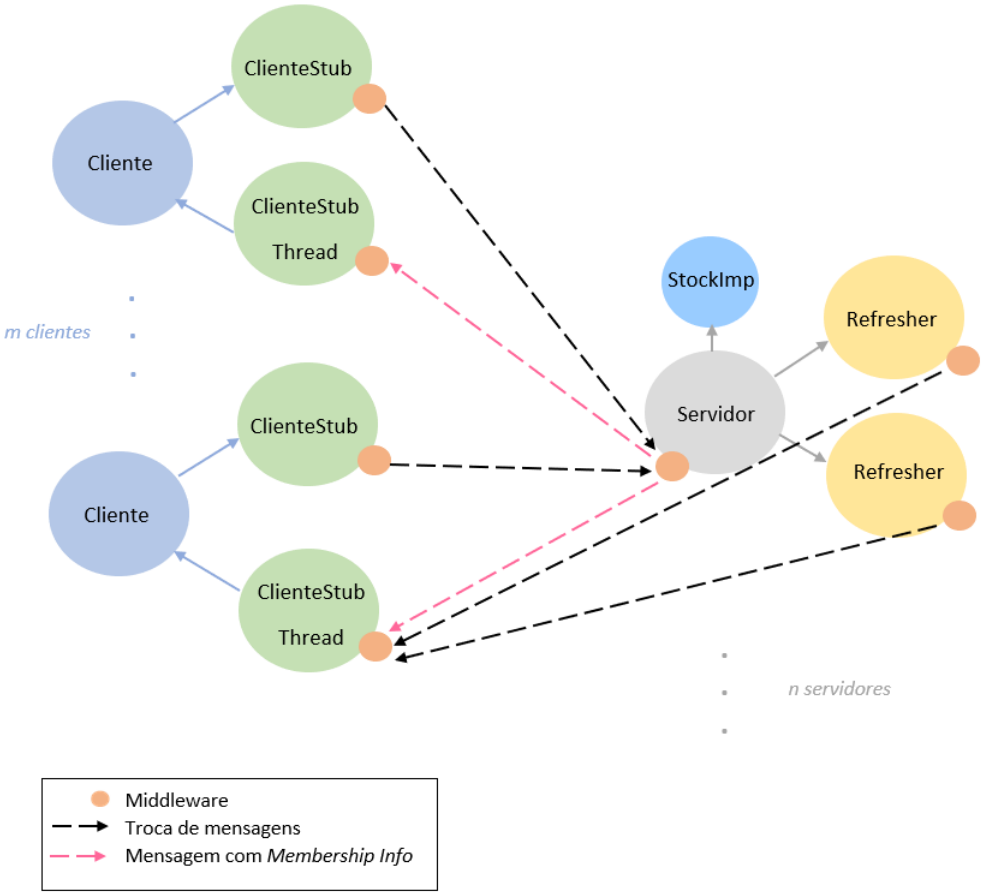

Figura 1: Arquitetura do sistema

4 *Middleware*

O *Middleware* serve como intermédio entre a comunicação dos clientes com os servidores. A necessidade da implementação desta peça na arquitetura surgiu pelo facto de tornar o processo de enviar e receber mensagens, através do protocolo de replicação ativa, mais genérico, podendo ser utilizado para qualquer serviço, aumentando assim a modularidade do código.

Para utilizar esta classe apenas se tem de indicar o nome da conexão e a que grupo se quer juntar.

5 Mensagens

As mensagens possíveis de enviar por parte dos clientes/servidores e que serão processadas são:

- **CompaniesRequest**: onde o cliente requer o *map* com todas as empresas e ações disponíveis nas mesmas. A esta mensagem responde-se com um **CompaniesReply**.
- **ActionsRequest**: onde o cliente requer a informação de quantas ações estão disponíveis na empresa Y. A esta mensagem responde-se com um *ActionsReply*.
- **BuyRequest**: onde o cliente compra X ações da empresa Y. A esta mensagem responde-se com um **BuyReply**, que contém um *boolean* para indicar se foi ou não possível realizar a compra.
- **SellRequest**: onde o cliente vende X ações da empresa Y. A esta mensagem responde-se com um **SellReply**, que contém um *boolean* para indicar se foi ou não possível realizar a venda.
- **MembershipInfoRequest**: onde o cliente reequer a informação sobre o grupo dos servidores, isto é, quantos servidores estão ativos e quais os nomes da sua conexão. A esta mensagem responde-se com um **MembershipInfoReply**.
- **EstadoRequest**: onde um servidor pede o estado aos servidores ativos, sendo que este estado corresponde às mensagens na lista a processar a partir de uma posição X, inclusive. A esta mensagem responde-se com um **EstadoReply**.

6 Cliente

A classe Cliente toma o papel das ações que um cliente real poderia aplicar sobre o sistema. Sendo assim, este possui uma estrutura onde armazena quantas ações possui sobre cada empresa e aplica determinadas funcionalidades sobre a bolsa (que é gerida pelos servidores). Estas funcionalidades correspondem às mensagens CompaniesRequest, ActionsRequest, BuyRequest e SellRequest.

Como esta classe representa apenas um teste para o funcionamento do programa, inicialmente este consulta as ações que cada empresa possui para venda e, de seguida, requer 5 vezes uma das outras 3 funcionalidades, de forma aleatória. Também o número de ações que compra e que vende é aleatório, sendo apenas verificado se pode de facto vender o número indicado.

6.1 *Stub*

Para esconder a logística da comunicação ao cliente, dando a ilusão que tudo acontece localmente, utiliza-se um *stub*. Esta entidade fica então responsável

por enviar os pedidos do cliente ao grupo dos servidores, utilizando assim a classe *middleware* para tal.

Para o cliente não ficar à espera dos pedidos que executa, aplicou-se *CompletableFuture* com a propriedade de depois avisar o cliente quando se obtivesse resposta. Desta forma, cada mensagem é associada a um identificador único, para se associar esta ao seu *CompletableFuture*.

6.2 *Stub Thread*

Como o *stub* tem de estar sempre disponível para receber novos pedidos do cliente e os enviar, este não pode utilizar o método *receive* do *Spread Toolkit* para receber mensagens pois esse método bloquearia a classe fazendo com que o cliente só pudesse fazer um pedido ao *stub*. Para resolver esse inconveniente, foi necessário criar uma *thread* para o *stub*, sendo esta responsável por esperar por mensagens novas, recebidas através do *Middleware*, as processar e mostrar o resultado ao cliente.

6.3 Ordem e espera pelos servidores ativos

Como ambos o *Stub* como o *Stub Thread* utilizam o *Middleware*, pode-se afirmar que cada cliente representa um grupo, estando nesse grupo sozinhos.

A conveniência destes pertencerem a um grupo de replicação ativa provém de dois fatores. O primeiro é o facto de estes receberem assim as mensagens por ordem, podendo estas ser apresentadas ao cliente pela mesma ordem que este as executou, mesmo com a utilização de *CompletableFuture*. Tal não se verificaria se se utilizasse outros métodos como o *messaging service* utilizado no semestre passado ou *sockets*. O segundo, e mais relevante, é pelo facto de, na ativa, o cliente ter de esperar que todos os servidores lhe respondam para efetivamente ler a resposta, verificando que, de facto, os servidores se encontram coerentes.

Para aplicar a segunda funcionalidade é necessário que o *Stub* e o *Stub Thread* partilhem uma estrutura que indique quais os servidores ativos naquele momento, e outra que associe por que servidores uma determinada mensagem está à espera. Além disso, o *Stub Thread* necessita de mais duas estruturas, uma que lhe indique de quem é que determinada mensagem já recebeu, e outra que guarde efetivamente a mensagem recebida. Desta forma, sempre que uma mensagem é enviada, o *Stub* acrescenta à estrutura referida uma entrada para a nova mensagem, onde a chave é o identificador único desta e o valor igual aos servidores ativos naquele momento. Como é necessário então que antes de enviada a primeira mensagem já se saiba a lista dos servidores ativos, a primeira função que o *Stub Thread* executa é perguntar ao grupo de servidores, através da mensagem MembershipInfoRequest, quem está nesse grupo. Estes, por sua vez, respondem com a mensagem MembershipInfoReply.

Após isso, a mensagem MembershipInfoReply é recebida sempre que houver uma mudança no grupo dos servidores, fazendo com que, ao recebê-la, o *Stub Thread* tenha de atualizar todas as estruturas referidas anteriormente. Caso determinada mensagem estivesse à espera de um servidor que já não está nos

ativos, esse servidor é removido dos servidores que ela estava à espera e verificado se já recebeu de todos os outros a que enviou. Se sim, a mensagem armazenada é apresentada ao cliente, caso contrário, essa mensagem tem de continuar à espera.

Caso receba uma mensagem dita "normal", o *Stub Thread* apenas têm de atualizar que determinada mensagem já recebeu também daquele servidor e verificar se já recebeu de todos os que estava à espera. Se sim, a mensagem é apresentada ao cliente. Se não, a mensagem é armazenada.

Como se pode verificar pela explicação acima, se a mensagem de MembershipInfoReply não fosse enviada com uma ordem implicita relativa às outras, não se poderia ter a certeza que de facto os servidores ativos naquele momento eram os incluidos no corpo da mensagem quando a recebesse, porque a mensagem poderia-se atrasar. Assim, justifica-se mais uma vez a relavância do cliente pertencer a um grupo.

Para tudo isto ser possível, os servidores têm de saber que clientes é que se encontram a comunicar com estes, fazendo com que os clientes tenham então de enviar o nome do seu grupo em todas as mensagens.

7 Servidor

Cada servidor tem uma instância da classe *StockImp*, onde se encontra a lógica de negócio, e uma *thread Refresher*.

Esta última é responsável pela processamento dos pedidos, alterando os dados do *StockImp*, e envio dessas respostas aos clientes.

O servidor fica então responsável pela receção de mensagens, que são inseridas numa lista para posteriormente serem processadas pelo *Refresher*. Assim, sempre que o servidor recebe uma mensagem, verifica se esta deve ser guardada na lista de mensagens para processar, se deve ser ignorada ou se deve ser guardada na lista de mensagens em espera. O critério a ter em conta para realizar esta escolha é explicado no tópico Recuperação do estado de forma parcial. Além disso também é esta que envia aos clientes informações sobre mudanças no grupo de servidores quando recebe uma mensagem de *membership*. Para tal, armazena-se o grupo dos clientes de quem recebe mensagem para posteriormente os poder então avisar.

Sendo assim, em cada servidor são criadas duas instâncias da classe *Middleware*, sendo uma utilizada para receber pedidos e a outra para enviar mensagens.

7.1 *StockImp*

A classe *StockImp* contém a lógica de negócio da bolsa de valores. É nesta que estão definidas as empresas disponíveis, armazenadas num *HashMap*, às quais os clientes podem consultar, comprar ou vender ações. Todas as empresas são iniciadas com 500 ações, sendo que não há limite máximo para a venda de ações

por parte dos clientes, havendo apenas um limite máximo para a compra de ações, que corresponde ao número de ações disponíveis.

7.2 *Refresher*

Na *thread Refresher* são processadas as mensagens recebidas por parte de clientes ou de outros servidores e envidas as respostas necessárias. Assim, sempre que houver uma mensagem na lista de mensagens a processar, é verificado o tipo da mensagem (por exemplo, *BuyRequest*, *CompaniesRequest* ou *EstadoRequest* para realizar a ação necessária para esse tipo e criar a mensagem de resposta (por exemxplo, *BuyReply*, *CompaniesReply* ou *EstadoReply*) sendo esta, de seguida, enviada.

Antes de ser processada a próxima mensagem da lista, primeiro é verificado se esta efetivamente pode ser processada. Isto é, caso o servidor esteja à espera de receber o estado por parte de outro, a mensagem apenas é processada caso corresponda a um *EstadoReply* digirido ao servidor em questão. Caso contrário, qualquer mensagem pode ser processada, sendo ignoradas mensagens do tipo *EstadoReply* e os próprios *EstadoRequest*.

É também importante referir que sempre que se processa uma mensagem, é incrementada a variável *nextMsg*, que indica a posição da próxima mensagem a processar na lista, sendo também utilizada para definir a partir de que posição é preciso recuperar estado (tópico Recuperação do estado de forma parcial). Esta variável é utilizada porque não se remove as mensagens já processadas da lista, pois todo o estado deve ser armazenado e estas podem vir a ser necessárias para outros servidores ficarem coerentes com os presentes.

7.3 Mensagens *membership*

No grupo de servidores optou-se por ativar a opção *membership* do *Spread Toolkit*. Com isto, sempre que há uma alteração no grupo, isto é, entrada ou saída de servidores, cada servidor recebe uma mensagem de *Membership Info*, onde são comunicados da mudança e onde podem tirar determinadas informações, tal como os servidores ativos na nova vista do grupo.

Recorreu-se a esta funcionalidade para enviar essa informação aos clientes e auxiliar a recuperação de estado dos servidores, tal como é aprofundado no próximo tópico.

7.4 Recuperação do estado de forma parcial

Os servidores desenvolvidos armazenam estado e acedem ao estado atual do sistema, caso se reintegrem neste, através de outros servidores ativos, a partir do envio das mensagens armazenadas.

Sempre que é processada uma mensagem no servidor ou no *Refresher*, o estado deste é armazenado através de *Serializable*, tanto em ficheiro TXT como em binário. Assim, sempre que um servidor se conecta, verifica se existem

dados armazenados para carregar, sendo que no caso de não existirem inicia-se um servidor vazio.

Depois do servidor se ligar e se juntar ao grupo, recebe uma mensagem de *membership info*, onde obtém informações sobre os servidores ativos. Caso o servidor seja o único no grupo, sabe que não precisa de pedir o estado atual do serviço e procede para a receção e processamento de pedidos. Caso existam mais servidores no grupo, envia uma mensagem de *EstadoRequest*, onde pede aos outros servidores as mensagens recebidas por estes a partir da última posição conhecida, sendo 0 no caso de o servidor não ter carregado estado anterior. Esta posição é determinada a partir da variável *nextMsg* mencionada anteriormente.

Enquanto aguarda por uma *EstadoReply* com as mensagens que necessita processar antes de poder atender pedidos, o servidor pode receber mensagens dos clientes ou de outros servidores. No entanto, como está em estado de espera, tem que determinar se deve ignorar as mensagens recebidas ou se as deve adicionar a uma outra lista de mensagens em espera. Assim, quando o servidor recebe a própria mensagem de *EstadoRequest* e como as mensagens são recebidas com ordem total, o servidor ignora todas as mensagens que recebe antes de receber o próprio pedido de estado e adiciona à lista de espera as mensagens que receber a seguir ao seu pedido. Isto porque, as mensagens que antecedem o pedido de estado do servidor foram também já processadas pelos servidores que atenderem ao seu pedido, logo o servidor não necessita de as guardar.

Quando recebe a mensagem de resposta com o estado do sistema por parte de algum servidor, realiza o processamento das mensagens incluídas nessa resposta, adicionando-as à lista principal de mensagens e incrementando o *nextMsg* para ter atualizada a próxima posição de mensagem a processar. Depois de lidar com todas as mensagens recebidas no estado, adiciona-se à lista principal as mensagens que se guardou na lista de espera, para de seguida se dar o processamento destas. A partir deste ponto, o servidor passa a ignorar todas as mensagens de *EstadoReply* e procede com o funcionamento normal.

Assim, para além de se armazenar os dados, os servidores têm também a recuperação de estado de forma parcial, pois não é necessário enviar tudo desde o início do programa para um servidor desatualizado, sendo apenas necessário enviar a partir do estado conhecido pelo servidor.

8 *Multithread*

Para aumentar o processamento das mensagens do lado dos servidores, seria necessário aumentar o número de *Refreshers* que cada um possui. No entanto, com a introdução de concorrência, alguns cuidados extras teriam de ser aplicados como no acesso à lista das mensagens para não processarem a mesma mensagem. Para tal, sempre que se acedesse à lista teria de se obter o *lock* desta e, só depois, o *lock* do *nextMsg* para aumentar essa variável, assim, quando se fizesse *unlock* de ambas, o outro *Refresher* acedesse à próxima posição, para assim processar a mensagem seguinte. Outra questão é caso a mensagem a processar fosse de escrita ou leitura. Se fosse de leitura então o *Refresher* poderia ler do *StockImp*

para responder, enquanto outro *Refresher* também lia desse mesmo objeto. No entanto, caso fosse de escrita, mais nenhum *Refresher* poderia ler ou escrever. Desta forma, se ao obter a mensagem da lista esta fosse de leitura, fazia-se uma cópia do *StockImp* e só depois fazia-se *unlock* do *nextMsg* e da lista das mensagens, sendo essa cópia utilizada para responder ao cliente depois dos *unlocks*. Caso fosse de escrita, só no fim de alterar o *StockImp* e responder ao cliente se poderia fazer *unlock* do *nextMsg* e da lista das mensagens. Desta forma, manteria-se a coerência das respostas ao cliente e do próprio *StockImp*, permitindo leituras simultâneas e, consequentemente, aumentar o processamento das mensagens.

9 Apreciação Crítica

Na realização deste trabalho foi possível aplicar vários dos conhecimentos adquiridos na cadeira, como a aplicação do protocolo de replicação ativa utilizando o *Spread Toolkit* que foi apresentado nas aulas práticas.

Além disso, esta escolha recaiu sobre uma reflexão entre os dois protocolos estudados e, até mesmo, tirando-se partido de características vantajosas presentes no protocolo de replicação passiva, como o uso das mensagens *membership*. Também pelo uso, por exemplo, da espera, por parte dos clientes, pelos vários servidores e pela recuperação do estado de forma parcial, considerámos que fomos além do que foi aplicado durante as aulas práticas.

O único ponto que gostaríamos de ter implementado era a questão de utilização de vários *Refreshers* nos servidores, para aumentar o processamento das mensagens, e não só apenas um. No entanto, consideramos que o facto do estudo teórico desse ponto já é por si um aspeto positivo, ficando o passo de implementação deste tópico com trabalho futuro deste serviço. Além de que se teria de estudar a quantidade ótima de *Refreshers* a aplicar consoante o tipo de ações que os clientes mais procuram do serviço. Caso fossem apenas escritas, a aplicação de mais *Refreshers* não apresentaria uma melhoria no processamento das mensagens.