Universidade do Minho

Mestrado Integrado em Engenharia Informática

Tolerância a Faltas

Serviço tolerante a faltas

Autores:

Bernardo Mota	(A77607)
Filipe Nunes	(A78074)
Luís Neto	(A77763)

30 de Junho de 2019

1 Introdução

Os sistemas distribuídos são cada vez mais utilizados, pois permitem que várias máquinas independentes, ligadas através de uma rede de dados, comuniquem entre si e tomem decisões como se se tratasse de um sistema único e coerente. De entre todos os desafios existentes na implementação deste tipo de sistema, a tolerância a faltas é dos mais importantes, uma vez que a falha de um componente (processo, computador ou rede) não deve afetar o comportamento do serviço. Atualmente, no mundo da tecnologia, são vários os exemplos de sistemas distribuídos, sendo os sistemas de pesquisa, serviços financeiros, jogos online e redes sociais os que mais se destacam.

O presente documento tem como finalidade justificar as escolhas do grupo na realização deste projeto. Este foi proposto na unidade curricular de Tolerância a Faltas e consiste na implementação, em `Java`, de um serviço tolerante a faltas, através de protocolos de replicação, usando o protocolo de comunicação em grupo `Spread`. Mais concretamente, pretende-se implementar um serviço de escalonamento de tarefas distribuído para um intermediário de bolsa de valores, onde dada tarefa representa uma operação de compra ou venda de uma quantidade de ações.

Ao longo do presente documento será apresentado de forma mais detalhada cada um dos componentes do sistema implementado e será descrita a sua função, tecnologias utilizadas e a justificação das mesmas.

2 Estratégia utilizada

De modo a traçar o plano de trabalho, teve de se definir quais as escolhas necessárias a serem feitas. De entre elas, a mais importante foi definir qual o protocolo de replicação que seria utilizado no sistema, sendo escolhido o protocolo de replicação **ativa**. Como mencionado anteriormente, tirou-se partido do *toolkit* de comunicação *Spread* para implementar este protocolo.

Na replicação ativa todas as réplicas do sistema são idênticas e recebem os pedidos do cliente pela mesma ordem, processam-nos, determinam qual o resultado correto e enviam a resposta ao cliente. Desta forma, as principal vantagem que este tipo de replicação apresenta é que as falhas são transparentes para o cliente, sendo adequado para sistemas ininterruptos. Com a escolha deste protocolo também se evita alguns dos problemas existentes no protocolo de replicação passiva, como a escolha da réplica primária e o facto de ter de se lidar com as possíveis falhas desta.

De maneira a otimizar o funcionamento do protocolo de replicação ativa foi necessário definir um *timeout* no cliente, de modo a tolerar possíveis faltas de *timing* que possam ocorrer. Para além disso, as réplicas enviam as respostas ao cliente e este, em vez de aceitar as respostas apenas quando as recebe na totalidade, aceita-as assim que tiver a maioria delas, o que permite ultrapassar possíveis falhas das réplicas.

Foram utilizadas *threads* para que o cliente possa realizar e receber simultaneamente pedidos, de modo a garantir concorrência de pedidos, que pode ser importante devido ao tipo de sistema.

3 Arquitetura do Sistema

A seguir será apresentada a arquitetura do sistema desenvolvido, assim como a descrição de cada componente do sistema e de algumas decisões de implementação.

3.1 Visão Geral

A seguir serão descritos os componentes que compõe o serviço para um intermediário de bolsa de valores.

O ***Client*** é a interface do utilizador com o programa, que comunica com o servidor para fazer pedidos de compra/venda e apresenta as respostas ao utilizador. No ***Server*** cada réplica recebe os pedidos do cliente, processa-os e envia a resposta ao cliente.

A comunicação entre cliente e servidor é estabelecida através da classe ***CommDaemon***, que encapsula as operações de envio e recepção de mensagens do *Spread* e mantêm uma *queue* de mensagens.

Tendo isto em conta, foi idealizada uma possível arquitetura do sistema ilustrada na figura 1.

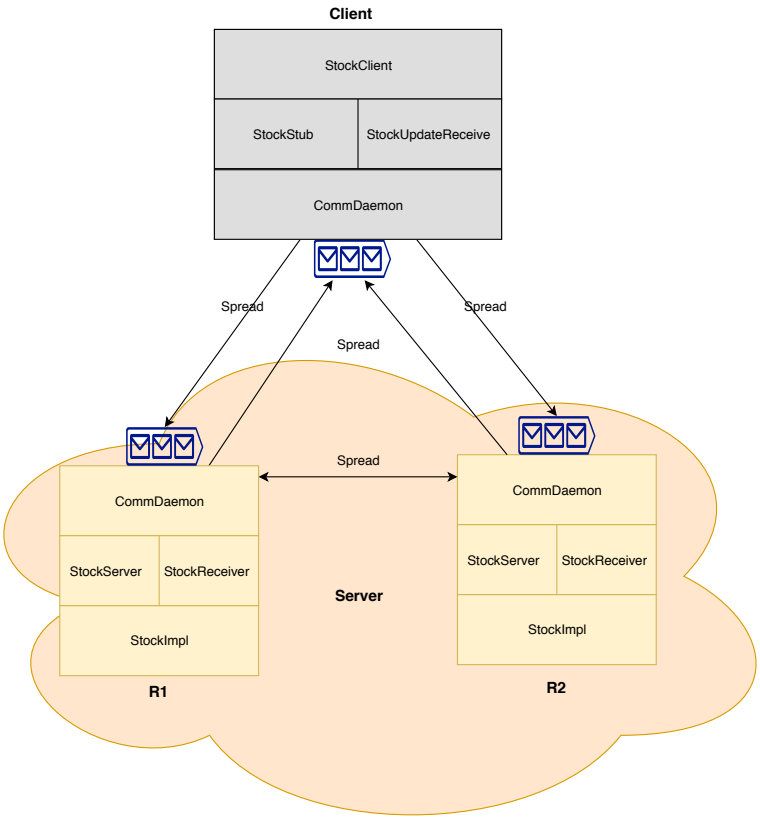

Figura 1: Visão geral do sistema.

3.2 *Client*

O *Client* está dividido em três classes: ***StockClient***, ***StockStub*** e ***StockUpdateReceive***.

A classe ***StockClient*** oferece a interface propriamente dita para o cliente e é nesta que o utilizador seleciona o tipo de pedido, podendo ser de compra ou de venda, que pretende realizar. Esses pedidos são realizados com uma invocação à operação no *StockStub* e o resultado é retornado através de um *CompletableFuture*.

O ***StockStub*** é responsável por enviar as operações aos servidores e receber as respostas. Ao enviar cada pedido através do *multicast* para as réplicas do servidor, mantêm um id local, para mapear o *CompletableFuture* que deve completar, depois de receber as respostas.

Para receber e validar as respostas, é utilizada uma thread da classe ***StockUpdateReceive***, que mantêm dados sobre o estado de cada operação enviada, como o *CompletableFuture*, o número de réplicas ativas e o número de respostas recebidas. O número de réplicas funcionais são enviados em cada resposta do servidor e o número de respostas é incrementado se a resposta for igual às respostas previamente recebidas dessa operação. Se o número de réplicas ativos tiver sido alterado, o cliente atualiza a estrutura com as réplicas existentes.

Para tolerar possíveis falhas de réplicas do servidor, foi estabelecido que o cliente, esperará por, pelo menos, 50 por cento de respostas idênticas das réplicas. Após receber o número de *acks* anteriormente definido, completa os *CompletableFuture* e a operação está concluída.

A utilização de *threads* e de *CompletableFuture* permite a introdução contínua por parte do cliente, recebendo a resposta de maneira assíncrona.

3.3 *Server*

O *Server* está dividido em três classes: ***StockServer***, ***StockReceiver*** e ***StockImpl***.

A classe ***StockImpl*** contêm a implementação da lógica da bolsa de valores, mantendo um *Map* com as empresas com ações associadas e o número de ações que o servidor disponibiliza. Foi decidido que o serviço aceita sempre a venda de ações, acrescentando estas ao número de ações disponíveis, e que apenas aceita a compra de ações se o número de ações disponíveis não for menor do que as ações que o cliente pretende comprar.

A classe ***StockServer*** engloba todo a lógica do servidor, incluindo a receção de mensagens de maneira ordenada, o processamento destas dependendo do seu tipo, a resposta ao cliente, o mantimento de vistas do grupo e log e a transferência de estado.

Quando o objecto *StockServer* é criado, é feita a recuperação de estado, que consiste no envio de um pedido *JoinRequest*, ao qual os servidores corretos respondem com uma resposta *JoinResponse*, contendo o *log* das operações executadas em cada réplica correta.

Após recuperar o estado, o *StockServer* recebe operações de forma ordenada através do *CommDaemon* e processa dependendo do tipo da operação. No caso de operações de compra e venda de ações, invoca a *StockImpl* e envia a resposta para o cliente. No caso de uma *JoinRequest*, envia o seu log em segmentos de 100KB para a réplica em recuperação

de estado. Todas as operações de compra e venda são acrescentadas ao log, garantindo a consistência de estado entre as réplicas no processo de recuperação de estado.

3.4 *Protocol*

O modulo ***Protocol*** contêm todas as classes necessárias para a serialização dos dados. Este foi criado para que fosse possível converter os dados em bytes e permitisse a utilização do *Spread Toolkit* para trocar mensagens entre clientes e servidores. Para este efeito foi o utilizada a biblioteca *Atomix* e foram definidas as seguintes classes:

- JoinRequest
- JoinResponse
- Operation
- OperationResponse
- ProtocolMessage

As classes ***JoinRequest*** e ***JoinResponse*** são utilizadas para pedir e responder com *log* de operações efetuadas até ao momento, respetivamente.

As classes ***Operation*** e ***OperationResponse*** são utilizadas para enviar os dados necessários para a compra ou vendas de ações e para responder ao cliente se a operação foi um sucesso ou não. Para além disto, a classe ***OperationResponse*** identifica quantas replicas aparentam estar corretas para que o cliente possa saber quando é que recebeu a maioria das respostas e prosseguir com a sua execução.

3.5 *CommDaemon*

Esta classe foi criada com a finalidade de estabelecer um protocolo de comunicação entre maquinas com o auxilio do *Spread Toolkit*. Como esta classe é utilizada por uma aplicação *multithreaded* foram tomadas as devidas precauções para garantir que esta fosse *thread-safe*.

O ***CommDaemon*** oferece uma interface simples que permite enviar, receber e entregar mensagens. Estas duas últimas funcionalidades são necessárias para que seja possível receber novos pedidos enquanto outro é processado. Para que isto fosse possível, foi utilizada uma *queue* onde são armazenadas as mensagens e retiradas quando necessárias.

Devido ao protocolo de replicação ativa foi criada uma segunda *queue* para que as respostas com os dados do *log* sejam armazenadas de forma independente. Para além disto, os pedidos só são armazenados a partir do momento em que a réplica que pediu o *log* de operações recebe o seu próprio pedido.

4 Conclusão

O desenvolvimento deste projeto permitiu, através da sua implementação, reunir e consolidar os conceitos sobre a tolerância a faltas em sistemas distribuídos, neste caso, acerda do protocolo de replicação ativa. Permitiu também um maior contacto com o protocolo de comunicação `Spread`, mostrando-se uma mais valia para a implementação do sistema.

Conclui-se que este projeto foi bem sucedido, uma vez que se conseguiu implementar tudo aquilo que estava proposto. Apesar disso, existiram algumas dificuldades na sua elaboração, como o facto de ter de se lidar com as réplicas lentas, tendo sido definido um *timeout* com o tempo máximo que o cliente espera. Para além disso, lidar com a concorrência também apresentou alguns entraves na implementação do sistema.