Fundamentos de Sistemas Distribuídos

Trabalho Prático

Relatório de Desenvolvimento

Miguel Oliveira	Pedro Moura	César Silva
pg41088	pg41094	pg41842

Universidade do Minho,

3 de Janeiro de 2020

Resumo

Este relatório descreve o desenvolvimento de um projeto no âmbito da UC de Fundamentos de Sistemas Distribuídos, onde, através de ferramentas e bibliotecas apresentadas nas aulas, são aplicadas técnicas para um bom e, sobretudo, fiável funcionamento de um sistema distribuído.

Conteúdo

Capítulo 1

Introdução

Com o objetivo de aplicar os conhecimentos adquiridos nas aulas relativas à UC de Fundamentos de Sistemas Distribuídos, foi-nos proposto o desenvolvimento de um sistema de troca de mensagens com persistência e ordenação, inspirado na rede social *Twitter*. O enunciado apresenta alguns requisitos que devem ser respeitados:

1. O sistema deve incluir um conjunto de servidores, que se conhecem todos entre si. Admite-se a possibilidade de um destes servidores ser reiniciado, devendo garantir que o sistema continua a funcionar depois de todos os servidores estarem novamente operacionais. O servidor não deve ter qualquer interação direta com o utilizador.

2. O sistema deve incluir clientes que se ligam a qualquer um dos servidores. Admite-se que o cliente pode ser reiniciado e ligado a um novo servidor. O cliente deve incluir uma interface rudimentar para interagir com o sistema, que deve ter algumas funcionalidades.

3. Admite-se que tanto os clientes como os servidores podem fazer uso da memória persistente.

4. O conjunto de mensagens obtido por cada cliente em cada operação deve refletir uma visão causalmente coerente das operações realizadas em todo o sistema, por esse ou outros utilizadores.

Neste primeiro capítulo foi feita a contextualização e foram apresentados os objetivos deste projeto. De seguida, apresentamos uma proposta de solução, onde são descritas tanto as abordagens seguidas bem como as decisões tomadas para atingirmos os objetivos. No terceiro capítulo, descrevemos como chegamos à proposta de solução apresentada, com detalhes mais técnicos. Posto isto, no quarto capítulo, apresentamos o resultado final da aplicação acompanhados com alguns testes realizados. Por fim, no quinto e último capítulo, sumarizamos o que foi escrito no relatório, e a nossa satisfação global com o projeto.

Capítulo 2

Proposta de Solução

2.1 Sistema

Muito abstratamente, o sistema é composto por um conjunto de servidores que comunicam entre si através de mensagens (principalmente de estado), e por clientes que se ligam a estes servidores.

Um bom e consistente funcionamento do sistema depende de uma igualmente boa capacidade de organização e coordenação por parte dos servidores, o que requer a utilização de algumas técnicas e propriedades, nomeadamente:

- *Leader election* - é o processo de eleger um servidor como o organizador das tarefas que são distribuídas por todos os servidores. É essencial para o sistema adquirir a capacidade de organização desejada.
- *Causal delivery* - é uma abstração usada para garantir que as mensagens são enviadas numa ordem que respeita a relação *happened-before*, que diz que se um evento deve acontecer antes de outro, o resultado destes eventos também deve refletir isso. Esta propriedade é o que permite o sistema ter informação consistente e ser coordenado.
- *Two phase commit* - é um protocolo usado para coordenar todos os processos que participam numa transação distribuída para os casos de *abort* ou *commit*.

Os servidores comunicam apenas por mensagens, logo é com estas que o sistema tem de conseguir implementar as técnicas referidas. De modo a que seja possível conseguir diferentes fins, é necessária a existência de diferentes tipos de mensagens, como iremos ver a seguir.

Outro aspeto fundamental no sistema é a relação servidor-cliente. Um servidor tem de ser capaz de hospedar diversos clientes, comunicar com eles, e alterar o seu estado consoante os pedidos efetuados por eles.

No caso do *2-phase commit*, é necessária a existência de um processo Coordenador, o qual envia e recebe mensagens dos participantes (servidores) de modo a executar o protocolo.

2.1.1 Mensagens entre servidores

As mensagens enviadas entre servidores têm 2 campos.

- *Vector Clock* - é um vetor de N relógios lógicos (um relógio por servidor). Com este vetor é possível implementar o algoritmo dos *vector clocks* que por sua vez assegura a *causal delivery* referida acima.

- *Conteúdo* - o conteúdo da mensagem em si.

Como foi referido anteriormente, as mensagens têm diferentes funcionalidades e, consoante estas, estão divididas em diferentes tipos.

2.1.1.1 Mensagens de estado

Sempre que um servidor efetua uma alteração no estado, é suposto notificar os outros servidores de modo a que estes se atualizem, e assim se garanta a consistência de informação. As mensagens de estado são as responsáveis por esta notificação, de tal forma que o conteúdo que é enviado é o estado do servidor após as alterações.

Os outros servidores, ao receberem uma mensagem deste tipo, extraem o conteúdo e atualizam o seu próprio estado consoante este conteúdo.

2.1.1.2 Mensagens para a *Leader Election*

No sistema, o líder é o processo com o maior *id*.

Cada servidor começa por enviar uma mensagem a todos os outros com o seu *id*. Quando recebidas todas as mensagens, os servidores pegam no maior *id* recebido, e elegem-no como líder.

O servidor líder está responsável por ler e guardar o estado global. No ínicio de cada execução do sistema, depois de eleito o líder, este lê o estado guardado de execuções anteriores, coloca-o como o seu estado, e envia-o aos restantes servidores.

2.1.1.3 Heartbeat

As mensagens de *heartbeat* servem para saber se algum servidor deixou de funcionar.

A cada segundo, o todos os servidores mandam mensagem aos restantes a perguntar se estão "vivos", e aguardam por resposta. Caso algum servidor não responda, o sistema espera que este reinicie, e reinicia a execução.

2.1.2 Mensagens entre coordenador e participantes

As mensagens enviadas entre o processo coordenador e os nodos participantes são dos seguintes tipos:

2.1.2.1 Coordenador para participantes:

- *prepared* - o coordenador pergunta aos participantes se estão prontos para executar a transação;

- *commit* - o coordenador informa os participantes que podem fazer *commit*;

- *abort* - o coordenador avisa os participantes que não podem efetuar a transação, e estes fazem *roll-back*.

2.1.2.2 Participante para coordenador:

- *ready* - o participante informa o coordenador que está preparado;
- *abort* - o participante avisa o coordenador que não está preparado.

2.2 Servidor

2.2.1 Conexão com o cliente

Os servidores estão responsáveis pela conexão dos clientes. Uma vez realizada a conexão, o servidor tem a capacidade de comunicar com o cliente conetado.

Esta comunicação é essencial porque permite que o servidor envie informação ao cliente. Estas mensagens podem ser avisos (ex.: *Registo efetuado com sucesso*; *Erro: password errada*; *Publicação efetuada com sucesso*), como podem ser respostas a pedidos do cliente (ex.: Enviar a lista de tópicos a que um utilizador está subscrito). Mas como é que um servidor sabe que a *password* está errada, ou que tópicos é que um utilizador está subscrito?

2.2.2 Estado

Cada servidor possui um estado que contém a informação acerca dos utilizadores e das publicações do sistema. Com isto, um servidor não só consegue ter controlo sobre autenticação, como consegue responder a pedidos de consulta de um cliente. Adicionalmente, pode acrescentar nova informação ao sistema.

2.3 Cliente

2.4 Conexão ao servidor

O cliente está responsável por conetar-se a um servidor. Uma vez feita a conexão o cliente consegue comunicar com o respetivo servidor. Isto faz com que o cliente possa enviar (receber) mensagens ao (do) servidor que permitem implementar as funcionalidades a seguir descritas.

2.5 Funcionalidades

Um cliente tem as seguintes funcionalidades:

- Autenticação
- Publicar uma mensagem etiquetada com um ou mais tópicos
- Indicar qual a lista de tópicos subscrita
- Obter as últimas 10 mensagens enviadas para tópicos subscritos

Capítulo 3

Desenvolvimento

3.1 Tecnologia

O projeto foi desenvolvido com a ferramenta `IntelliJ`, que é um IDE utilizado para desenvolver *software*. A linguagem utilizada foi o `Java`, com principal destaque na biblioteca `Atomix`.

O `Atomix` é uma *framework* orientada a eventos destinada a desenvolver sistemas distribuídos. Fornece blocos de construção que resolvem muitos problemas de sistemas deste tipo, como por exemplo, mensagens assíncronas.

3.2 Implementação

O código do sistema está dividido em *packages*, onde cada *package* está responsável por uma funcionalidade.

3.2.1 *net*

O *package* `net` está responsável pela parte distribuída do sistema.

Em principal destaque está a classe `MessageHandler`, que trata da comunicação entre servidores, ou seja, da receção e interpretação das mensagens internas relacionadas com sistemas distribuídos. As mensagens são manuseadas através de um `NettyMessagingService`, que é uma classe do `Atomix`.

É nesta classe que são implementadas a *causal delivery*, *leader election* e os *heartbeats*.

3.2.2 *server*

Neste *package* está a classe `Server`, que representa um servidor do sistema.

A classe `Server` tem várias variáveis de instância, sendo que em destaque estão:

- `mh` - `MessageHandler`, explicado na secção anterior;
- `port` - é a porta do servidor;
- `ssc` - *server socket channel* da classe `FutureServerSocketChannel`;
- `users` - lista dos utilizadores;
- `posts` - lista das publicações.

O `Server` começa por inicializar o `MessageHandler`, fazendo de seguida a eleição de líder. Após a eleição, o servidor recupera o último estado do sistema, que está guardado em `SegmentedJournals`, classe do `Atomix`, e guarda-o em memória no `users` e `posts`, enviando-os depois para os restantes servidores.

De seguida cria o `FutureServerSocketChannel` para esperar conexões de clientes. Uma vez que um cliente esteja conetado, este e o servidor comunicam através de `FutureLineBuffer`s.

Dependendo da mensagem recebida, o servidor pode ou não atualizar o `users` e o `posts`.

3.2.3 *client*

Neste *package* estão todas as classes relacionadas com o cliente, nomeadamente a classe `Client` e os menus da interface.

Um cliente liga-se a um servidor através de um `FutureSocketChannel` e comunica com ele através de um `FutureLineBuffer`.

Após se ligar ao servidor, faz *display* do menu inicial. Consoante as ações que escolher nos menus, mensagens respetivas são enviadas e respostas esperadas. Nos menus são implementadas as funcionalidades descritas no capítulo anterior, na secção do Cliente.

3.2.4 *data*

Package que contém os modelos dos utilizadores e das publicações, bem como as classes dos `SegmentedJournals` que o sistema usa para guardar o seu estado.

Capítulo 4

Conclusão

Neste projeto, descrevemos o desenvolvimento do projeto da disciplina de Fundamentos de Sistemas Distribuídos.

No geral estamos satisfeitos com o resultado, visto que cumprimos os objetivos pedidos.

- Discussão dos resultados
- Pros/contras da abordagem seguida
- Como poderíamos evoluir o que foi construído
- Sempre com enfoque que foi feito tudo o que foi pedido