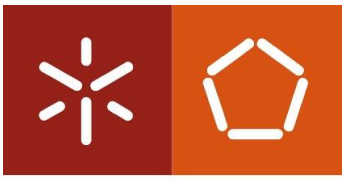

Universidade do Minho

Departamento de Informática

Fundamentos de Sistemas Distribuídos
Sistema de troca de mensagens

Eduardo Barbosa (a83344) João Vilaça (a82339)

5 de Março de 2021

Conteúdo

1 Introdução

Ao longo deste projeto, realizado no âmbito da Unidade Curricular de Fundamentos de Sistemas Distribuídos, pretende-se criar e analisar programas distribuídos que manipulam um estado global coerente recorrendo a transações.

Para isso foi proposta a implementação de um sistema distribuído de troca de mensagens com persistência e ordenação onde serão identificados e discutidos os mecanismos transacionais que integrarão o middleware de distribuição, baseado em programação por eventos acente na *framework* **Atomix**.

2 Descrição do problema

2.1 Servidores

O servidores deverão compor um sistema distribuído, capaz de se auto-gerir. Estes deverão manter uma visão causalmente coerente das operações realizadas em todo o sistema por todos os clientes, através uma interface de comunicação para a realização de operações que deverá ser disponibilizada. Deverá ser ainda possível, em caso de falha de um dos servidores, a sua atualização de estado e a recuperação do sistema, quando este é novamente inicializado.

2.2 Cliente

O cliente será o serviço que fará a ponte entre os utilizadores deste sistema e os servidores que o compõe e, para isso, deverá ser capaz de se ligar e comunicar com qualquer um dos servidores. Terá então de ser capaz de receber pedidos dos utilizadores e "comunicá-los"a um servidor.

2.3 Utilizador

Deve ser oferecida ao utilizador uma interface na qual seja possível, executar as funcionalidades básicas de um sistema de troca de mensagens, nomeadamente publicar uma mensagem com um ou mais tópicos, subscrever a uma lista de tópicos e obter as últimas 10 mensagens relativas aos tópicos subscritos.

2.4 Assunções

A implementação final teve em conta as seguintes assunções:

- Qualquer mensagem no sistema tem no mínimo um tópico;
- Os servidores são conhecidos *a priori*;
- Qualquer servidor pode ser reiniciado;
- O sistema deve continuar a funcionar após todos os servidores estarem operacionais;
- Cada cliente pode ligar-se a qualquer servidor;
- Qualquer cliente pode ser reiniciado;
- Quando um cliente é reiniciado, este pode ligar-se a um servidor diferente do original.

3 Arquitetura

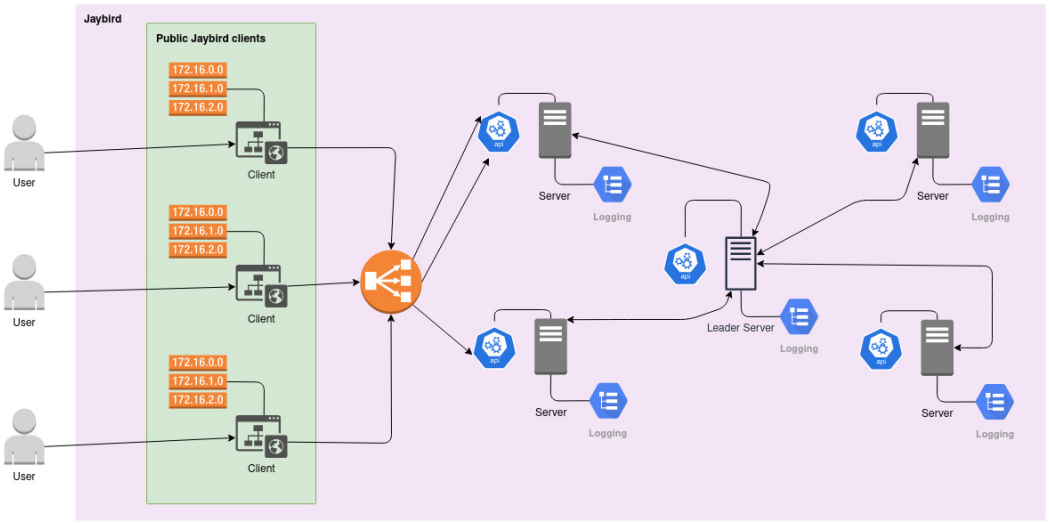

Figura 1: Arquitetura Geral

O sistema é constituído por um número fixo de **servidores** e de **clientes**. O conjunto de clientes e de servidores são conhecidos *a priori*, sendo que *host discovery* não se encontrava no âmbito deste projeto. Cada **utilizador** liga-se a um cliente através de uma interface Java.

Cada cliente consiste num *website*, público, que expõe uma *RESTful API*, de forma a interagir com o(s) servidor(es). É utilizado um *load balancer* em *round-robin* de forma a aliviar a carga de cada servidor. Por sua vez, cada servidor expõe uma *API REST* que é utilizada pelos clientes. A escolha de que servidor utilizar é aleatória e completamente transparente ao utilizador.

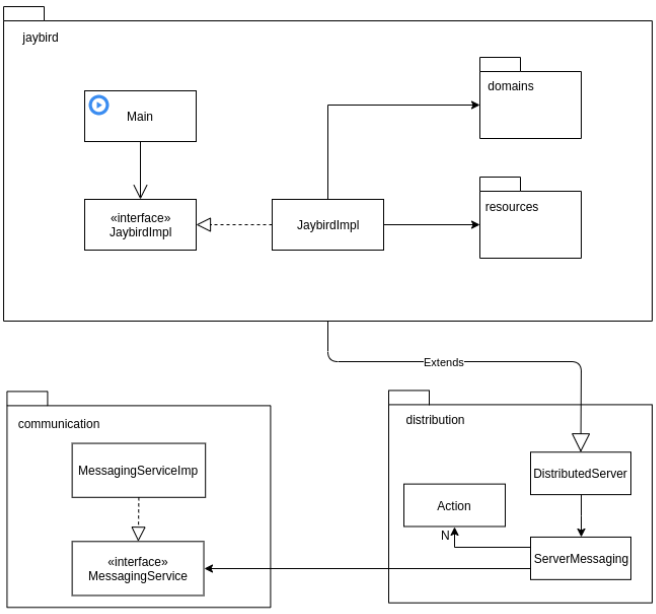

Figura 2: Packages do lado do servidor

4 Middleware de Comunicação

O middleware de comunicação, responsável pela troca de mensagens entre 2 ou mais componentes do sistema, é baseado na framework *Atomix*, mais propriamente no serviço *ManagedMessagingService*. A encapsulação implementada oferece através do interface *MessagingService* métodos para adicionar "Hosts", enviar mensagens para todos os "Hosts"conhecidos, para todos excepto para o próprio e para um "Host"em específico. Este serviço disponibiliza ainda operações sobre um fila de mensagens recebidas, nomeadamente consultar o primeiro elemento da fila e remover o primeiro elemento da fila.

5 Middleware de Distribuição

Sobre este módulo de comunicação está então implementado o middleware de distribuição. Para ser usado, este módulo oferece ao utilizar uma "abstract class"que disponibiliza o método "start"para iniciar o servidor, o método "send"para distribuir os dados por todos os servidores e obriga a implementação do método "objectReceivedCallback"que é invocado quando é recebido um novo dado.

A orquestração do servidor e das suas tarefas é feita a partir da classe "ServerMessaging". Esta classe lança várias *Actions* e faz a ponte entre o sistema distribuído e a lógica de negócio a ser implementada pelo programador.

5.1 Actions

Todas as ações usadas pelo "ServerMessaging"são baseadas em algumas classes do package "ActionHandler". Neste package podemos encontrar a "abstract class Action"que define um método "run"que adquire um lock, executa a ação definida na sua concretização ("startAction"). Quando é invocado o método "finishAction"definido em "Action", é informado o "ServerMessaging"e o lock é libertado. Na concretização de "Action"é possível fazer uso da classe "Timeout"disponível no package acima referido, para limitar o tempo de espera, e em que, caso se dê efetivamente o timeout, é executado o método "timeOut"definido na concretização. Por fim, cada "Action"tem associada a si um "Receiver", uma Thread, que fica à escuta na fila disponibilizada pelo módulo de comunicação de mensagens que devem ser tratadas por essa "Action".

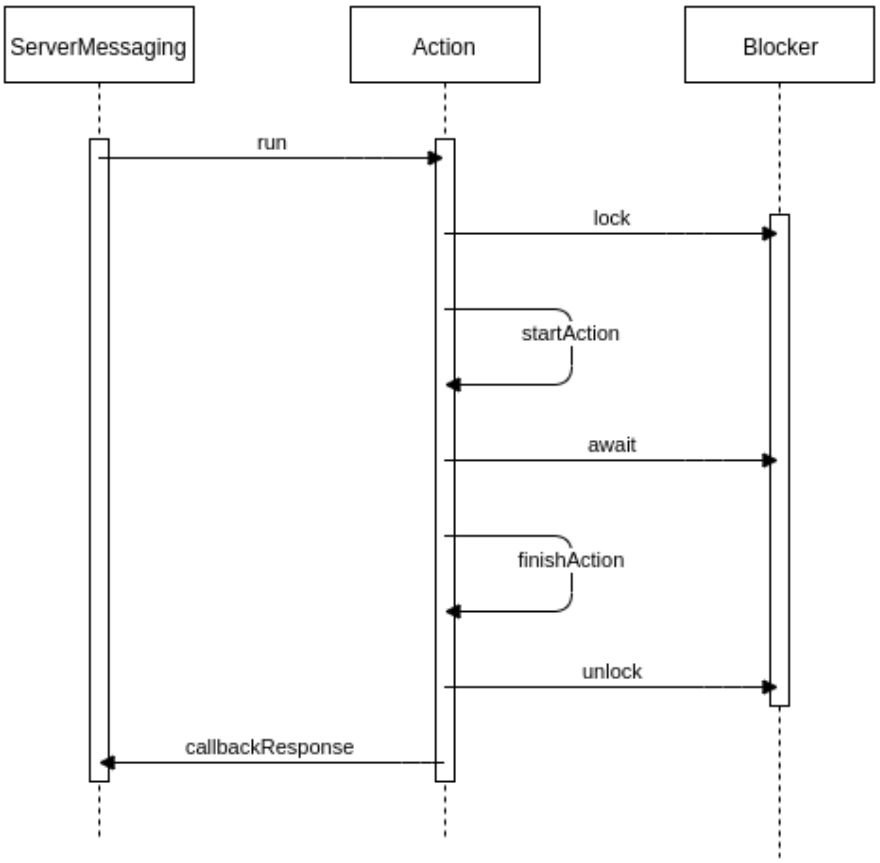

Figura 3: Estrutura de "Action"

5.1.1 LeaderDutiesAction

Quando um servidor é eleito líder, o "ServerMessaging"inicia esta tarefa onde são realizadas todas as tarefas da responsabilidade do líder, isto é, ficar à escuta, através do "Receiver"de pedidos de atualização do estado e início do protocolo de Two Phase Commit (2PC).

No caso dos pedidos de atualização do estado, o servidor simplesmente lê as mensagens armazenadas e faz o seu envio para a origem do pedido.

No caso de receber um pedido de início do protocolo de 2PC, começa por enviar em *broadcast* a indicação de que se deve fazer o stage da mensagem e inícia o mecanismo de timeout. Este fica então há espera que todos os servidores confirmem a mensagem dentro do tempo limite. Caso todos confirmem, emite em broadcast indicação para que seja feito o commit da mesma, ou em caso contrário emite indicação para que seja cancelada.

5.1.2 CommitMessageAction

Esta ação é responsável por todos os processos de envio de novas mensagens durante toda a execução do programa. Assim que o "ServerMessaging"é iniciado o "Receiver"associado a esta tarefa fica sempre à escuta de mensagens relacionadas com o processo de 2PC, nomeadamente à espera de indicação do líder para dar stage de uma mensagem, à quem responde com um okay, e, em seguida, caso tudo corra bem recebe do líder a indicação para dar commit dessa mensagem ou então, em caso de erro, para dar abort. O método "run"nesta tarefa é invocado para comunicar ao líder um mensagem nova para ele dar início ao processo de 2PC. Nesta "startAction"é usado o mecanismo de timeout falado anteriormemte, ativado caso o líder não confirme a receção da

mensagem, e que inicia uma nova eleição de líder.

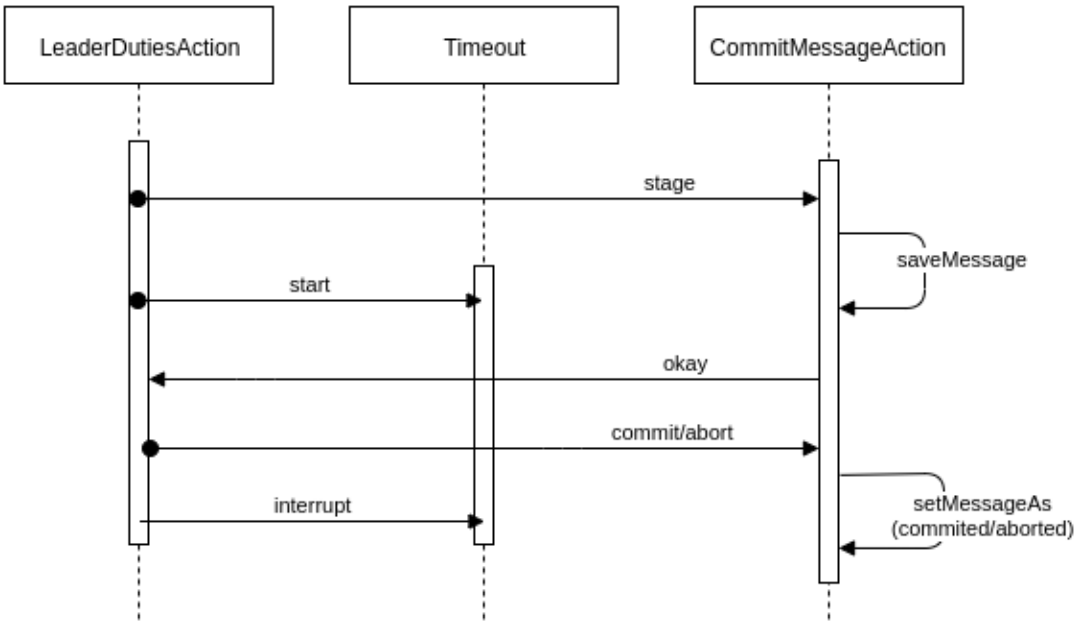

Figura 4: Two Phase Commit

5.1.3 LeaderElectionAction

Sempre que o "ServerMessaging"é iniciado, pede a todos os servidores que conhece para o informarem quem é o líder, caso ninguém responda (estão todos em baixo ou a iniciar), ele toma a iniciativa de começar um eleição de líder. Para além disso, qualquer servidor que receba uma mensagem e a envie ao líder através da "CommitMessageAction", caso se aperceba que o líder não está saudável, informa o "ServerMessaging"que inicia uma eleição de líder.

```
if (node P realizes(leader crash))
    sends an ELECTION message to all nodes

if (no node contest leader election)
    select P as LEADER
    &
    send broadcast message to indicate that he is alive and will take over
else
    leader election started on another node
```

5.1.4 UpdateStateAction

Quando um servidor é iniciado e recebe a indicação de quem é o líder, esta ação é iniciada para que este servidor fica com o estado partilhado mais recente. O servidor envia uma mensagem ao líder a pedir que lhe envie todas as mensagens que compõe o estado, e, em seguida, ordenadamente o servidor faz o seu carregamento para o estado.

5.2 Serialização

A serialização obriga a que todos os "domains"que o programador queira distribuir pelo sistema implementem "java.io.Serializable", isto permite que, na abstração do sistema distribuído,

se possa referir a este objetos como da classe "Serializable"e assim permite ser agnóstico a quais objetos são criados pelo programador, tornando este módulo verdadeiramente independente, e ainda simplifica a transmissão desses mesmo objetos possibilitando o uso de bibliotecas como "SerializationUtils"que convertem o objeto para bytes e vice-versa.

6 Conclusões

O desenvolvimento deste projeto foi um sucesso, a equipa implementou um sistema de troca de mensagens com persistência e ordenação, utilizando Java e Atomix, que está funcional. Todos os requisitos para este sistema foram considerados como cumpridos.

Este sistema de troca de mensagens, está baseado em 2 middlewares desenvolvidos para este projeto, um responsável pela comunicação e outro pelo sistema distribuído. Este 2 módulos são totalmente independentes desta aplicação em concreto e podem ser utilizados em qualquer outro projeto de grande ou pequena escala, sem qualquer limitação.

Destacam-se 2 grandes decisões efetuadas ao longo do projeto. A primeira, a definição da arquitetura, uma proposta que fosse otimizada para um elevado número de clientes e servidores, mas que garantisse a uma visão causalmente coerente de todas as operações efetuadas por todos os utilizadores. Assim sendo, optou-se por implementar o protocolo de Two Phase Commit para garantir a atomicidade e a exclusão mútua, apesar dos custos em termos de atraso causados pelo peso desta operação. A segunda, a abstração do conceito ação, que permite facilitar o processo de gestão de tarefas relacionadas com o sistema distribuído, criando uma interface comum a todas elas, simplificando e otimizando a trabalho efetuado pelo "ServerMessaging".