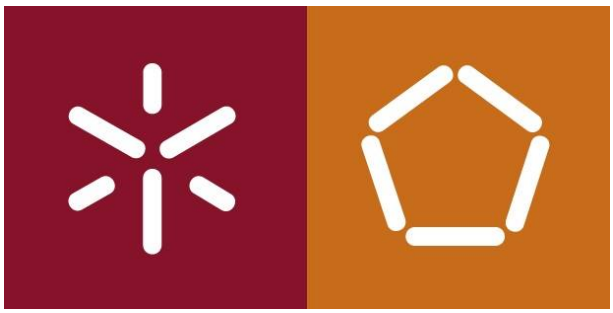

Universidade do Minho

Mestrado em Engenharia Informática

Fundamentos de Sistemas Distribuídos

Trabalho Prático

Grupo 6

Flávio Martins (a65277)
Pedro Costa (a85700)
Ana Ferreira (pg44412)
Bruno Santos (pg44414)

1 de fevereiro de 2021

Conteúdo

1 Introdução

O presente trabalho prático foi realizado no âmbito da unidade curricular de Fundamentos de Sistemas Distribuídos e consistiu no desenvolvimento de um sistema distribuído de armazenamento em memória de pares chave-valor.

O sistema implementado deve assumir um sistema assíncrono e possuir um conjunto fixo e previamente conhecido de servidores destinados ao armazenamento de subconjuntos disjuntos de chaves às quais são associados valores. Os clientes deve ter a possibilidade de realizar duas operações: a operação PUT que serve para guardar um conjunto de pares chave-valor e a operação GET que permite a leitura de valores associados às chaves solicitadas. A distribuição das chaves pelos servidores deve ser fixa recorrendo a uma função de *hashing*.

O sistema deve, também, garantir que um cliente observa as escritas realizadas por si próprio anteriormente e que quando dois clientes escrevem concorrentemente os mesmos itens, os valores que persistem para serem lidos depois das operações terminarem são todos provenientes do mesmo cliente.

Este relatório tem como finalidade a exposição da abordagem utilizada na concretização deste projeto e explicação do funcionamento do sistema.

2 Desenvolvimento

2.1 Abordagem utilizada

Um dos requisitos do presente projeto era a utilização de vários servidores para armazenar subconjuntos disjuntos de chaves (`Long`) às quais são associados valores (`byte[]`). Deste modo, decidiu-se elaborar um *cluster* constituído por um número variável de servidores fixos e previamente conhecidos e um coordenador. Em cada servidor são armazenados, para além do par chave-valor, a versão do correspondente valor.

Para manter a consistência do nosso sistema, optou-se por desenvolver um coordenador centralizado que usa um relógio global escalar. Este coordenador possui um *map* de todas as chaves do sistema com as respetivas versões e um *map* de todos os servidores ativos com o estado de cada versão, concluída ou não. No sentido de não sobrecarregar o coordenador, apenas as operações PUT incrementam o relógio global. Com o propósito de adotar modularidade no sistema, todas as mensagens trocadas entre a API, o coordenador e cada um dos servidores são encapsuladas através da classe parametrizável `GenericMessage<T>`, cujas variáveis de instância são o tipo de mensagem, o valor e o *clock*. Esta classe oferece métodos para fazer a serialização e a deserialização de mensagens. Com a mesma finalidade, na classe Protocol inseriu-se duas funções genéricas de escrita e de leitura num *socket*.

O cliente interage com o sistema (*cluster*) através de uma API, cujo propósito é oferecer suporte aos pedidos dos clientes. São duas as operações que disponibiliza: PUT que permite a escrita de um conjunto de chave-valor e GET para a leitura de valores associados a um conjunto de chaves.

- `CompletableFuture<Void> put(Map<Long,byte[]> values)`
- `CompletableFuture<Map<Long,byte[]> get(Collection<Long> keys)`

A API possui uma função de *hashing*, `int HashFunction(Long key)`, que distribui as chaves pelos servidores de forma equitativa. A cada cliente que pretende aceder ao sistema é associada uma API. Esta apresenta um *map* com todos os `FutureSocketChannel` do sistema, incluindo o do coordenador.

De forma a conceber um código não bloqueante e de forma a que as operações sejam executadas de forma assíncrona, utilizamos `CompletableFuture`. Sempre que possível recorremos às *lambda functions* de forma a obter um código mais conciso.

No *package* ExampleUsage encontra-se um exemplo de utilização do sistema. Neste exemplo são concebidos quatro clientes, três dos quais fazem uma operação de PUT e um faz uma operação de GET. As operações são realizadas de forma concorrente. Em cada operação de

PUT é enviado para o sistema um conjunto de 500 pares chave-valor, as chaves são sempre numeradas de 0 até 499. Na operação GET, é requisitado o valor que se encontra armazenado no sistema nas chaves numeradas de 0 até 499.

Na figura seguinte, apresentamos graficamente, a arquitetura desenvolvida do presente projeto.

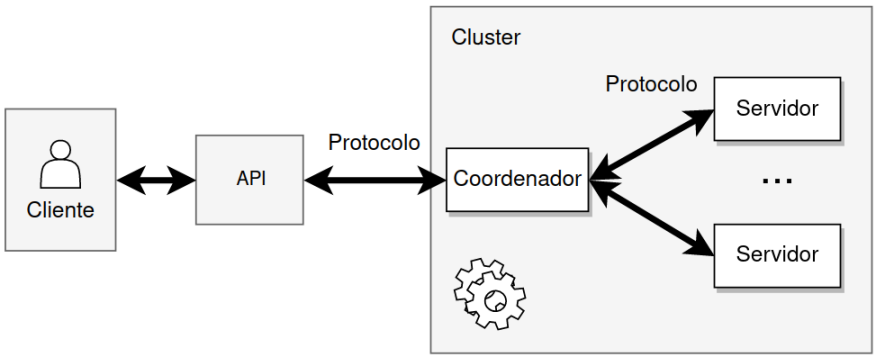

Figura 1: Arquitetura do sistema desenvolvido.

2.2 Implementação

Nesta secção será feita a explicação das componentes principais do sistema, necessárias para a implementação de todas as suas funcionalidades, consoante as decisões referidas na secção anterior.

2.2.1 Protocolo

De modo a possibilitar o envio e receção de mensagens, sem a utilização da biblioteca Atomix, desenvolveu-se um protocolo modular, que permite escrever e ler qualquer tipo de mensagem serializada num *socket*. Para tal, criou-se a classe Protocol que possui dois métodos genéricos para escrita e leitura de mensagens, entre quaisquer duas entidades que troquem mensagens. Criou-se, também, a classe parametrizável GenericMessage com o objetivo de encapsular cada mensagem trocada e cujas variáveis de instância são o tipo de mensagem, o valor e o *clock*.

O protocolo define poucas operações, mas foram suficientes para criar todos os requisitos do sistema. Foram disponibilizados os seguintes tipos de mensagem:

- Entre o cliente e o coordenador:

  - GET_CLOCK - Utilizado na operação PUT, na fase inicial, em que o cliente pede ao coordenador o seu relógio.
  - GET_SS - Utilizado para pedir o snapshot ao coordenador.

- Entre o cliente e os servidores:
  - PUT - Utilizado sempre que queremos inserir dados.
  - GET - Utilizado para pedir dados a um servidor.

- Entre os servidores e o coordenador:
  - PUT_DONE - Indica ao coordenador que o servidor em causa acabou o trabalho que estava a fazer.

Para respostas que não precisam de ter o seu tipo verificado criou-se o tipo `REPLY` e para lidar com erros tem-se o tipo `ERROR`.

2.2.2 Coordenador de versões

Na implementação das funcionalidades do sistema foi necessário desenvolver um mecanismo capaz de evitar conflitos entre operações realizadas por diferentes clientes, de modo a garantir a consistência do sistema. Para resolver esse problema utilizou-se um coordenador de versões centralizado, constituinte do *cluster*, cujo objetivo é armazenar todas as versões dos valores escritos para cada uma das chaves e coordenar as operações por todos os clientes.

Neste coordenador é guardado um *map*, onde para cada chave é associado uma lista com as versões existentes dessa chave e um outro *map*, onde para cada servidor se verifica o estado de cada versão nesse servidor, isto é, se foi completada ou não. Neste coordenador existe, ainda, um *clock* global, sendo este o *clock* que irá distinguir as várias versões.

De cada vez que um cliente pretende fazer um PUT, este pede ao coordenador uma versão e o coordenador devolve o valor do *clock* atual, incrementando o mesmo de seguida. Desta forma, é possível reconhecer as versões mais recentes pelo *clock* superior. Antes de confirmar a versão do PUT ao cliente, o coordenador vai adicionar essa versão a todas as chaves que irão ser atualizadas pela mesma e, ainda, adicionar essa versão a todos os *peers*, sendo que a marca como *false* para os *peers* que participam nela, ou seja, como não terminada nos mesmos, e como *true* aos que não vão participar, visto que estes não precisam de trabalhar na mesma para esta ser válida. Desta forma, uma operação PUT com várias chaves, mesmo que estas estejam distribuídas por vários servidores, apenas será dada como concluída, e como tal, aceitável para os pedidos do tipo GET, após todos os pares chave-valor desta operação terem sido adicionados nos vários servidores.

Quando um cliente pretende fazer um GET, vai enviar o pedido primeiro ao coordenador, que irá responder com um *snapshot* seu, este *snapshot* irá conter o *map* de versões para cada chave, e o *map* de conclusão das versões nos diferentes servidores. Desta forma, o cliente poderá ver qual a versão válida mais recente para as chaves que procura. Essa seleção é feita no cliente, de modo a não aumentar ainda mais a latência do uso de uma solução centralizada.

O último tipo de mensagem que o coordenador pode receber vem dos servidores, indicando a conclusão de uma versão. Cada vez que um servidor termina uma operação PUT, é enviada uma notificação ao coordenador para que este marque a mesma como completa no mapa de conclusão de versões por servidor.

2.2.3 Operação PUT

Como supracitado, a presente operação tem o objetivo de inserir um conjunto de chave-valor no sistema. Para manter a consistência do sistema e o controlo de versões, em cada PUT, é necessário ter acesso ao valor atual do relógio escalar presente no coordenador. Para tal, é executado o método `getClock(Map<Long, byte[]> map)`. Este método envia uma mensagem, serializada, ao coordenador do tipo `PUT`, que retorna o valor atual do relógio.

De seguida, no método PUT da API, na estrutura de dados `Map<Integer, Map<Long, byte[]» peerMap` e através da função de *hashing*, são distribuídas as chaves a serem armazenadas por cada servidor ativo, sendo que a variável do tipo `Integer` corresponde ao servidor e o `Map<Long,byte[]> >` ao conjunto de chave-valor a ser armazenado nesse servidor.

Distribuídas as chaves pelos servidores, para cada elemento da variável `peerMap`, é executado o método `putOne`. Este tem como objetivo, para cada servidor que participará na presente operação PUT, construir uma mensagem do `put`, com o valor que é o conjunto de chave-valor a ser armazenado nesse servidor e o relógio atual, devolvido momentos antes pelo coordenador. De seguida, a mensagem serializada envia para o servidor respetivo. Este método é, então, executado de forma assíncrona e não bloqueante, para cada servidor que participará no presente processo PUT.

Para garantir que todas as chaves e valores do conjunto enviado pelo cliente eram inseridas, utilizou-se o método `CompletableFuture.allOf()` para que no método PUT se esperasse pela conclusão do envio de todas as chave-valor para os respetivos servidores. Concluída esta etapa, o método PUT retorna um `CompletableFuture<Void>` significando que o conjunto chave-valor enviado pelo cliente foi inserido com sucesso.

2.2.4 Operação GET

A operação GET tem como desígnio obter um conjunto de valores para um dado conjunto de chaves. Esta operação encontra-se codificada na classe API no método `get`, cuja sintaxe

encontra-se na secção 2.1.

O método `get` recebe como parâmetro um conjunto de chaves. De seguida executa o método `getFromCoordinator(List<Long> keys)`. No método `getFromCoordinator` é construída uma mensagem do tipo `GET_SS` cujo valor é o conjunto das chaves. Esta mensagem é enviada para o coordenador através do método `write` do protocolo implementado.

Como a mensagem é do tipo `GET_SS`, significa que o coordenador ao receber a presente mensagem irá construir um *snapshot*, ou seja, o coordenador verifica qual é a versão mais consistente das respetivas chaves. As variáveis de instância `keyVersions` e `completedVersions` do coordenador, respetivamente o *map* de todas as chaves e as suas versões e o *map* todos os servidores e se a versão nesse servidor já foi concluida, são encapsulados num objeto do tipo `Snapshot`. O coordenador constrói uma mensagem do tipo `REPLY` com o objeto referido e envia de volta para a API.

Na API, recebido o *snapshot* e retornando ao método GET, é criada a estrutura de dados `peerMap`. Através da função de *hashing*, é possível conhecer qual o servidor cujo par chave-valor está armazenado. Posto isto, na variável `peerMap`, para cada servidor que irá participar na atual operação de GET, é associado o conjunto de chaves e a sua versão consistente pertencentes a esse servidor. Para cada elemento de `peerMap`, servidor a participar no atual processo GET, é executado o método cuja sintaxe é a seguinte:

- `CompletableFuture<Void> getOne(Integer peer, Map<Long, Integer> value, Map<Long, byte[]> map)`

O método `getOne` recebe a porta de um servidor (*peer*), o conjunto de pares chave e a sua versão consistente (*value*) e uma referência a um `HashMap<>` (*map*). De seguida uma mensagem é construída do tipo `GET` com os pares chave-versão. Esta mensagem é enviada para o servidor através do método `write` do protocolo implementado. O servidor processa a mensagem e envia os valores das respetivas chaves de volta para o presente método. Os pares chave-valor recebidos do servidor é armazenado no `HashMap<> map`. A variável `map` tem como objetivo armazenar todos os pares chave-valor de todas as chaves pedidas no atual processo de GET. Como o método `getOne` é executado de forma assíncrona e concorrente, sempre que um par chave-valor é armazenado na variável `map`, é necessário antecipadamente fazer o *lock* e de seguida o *unlock*, para evitarmos as *race conditions*.

Utiliza-se um `CompletableFuture.allOf` para esperar que todos os gets terminem. Quando todos os servidores, que participam no atual processo GET, enviam os respetivos pares chave-valor, colocados no `HashMap<> map`, o método `GET` da API retorna um `CompletableFuture<Map <Long, byte[]>`, constituído por todos os pares chave-valor requisitados inicialmente no conjunto de chaves.

Foi ainda implementado um *garbage collector* que apaga versões antigas e inutilizadas.

3 Conclusão

O presente trabalho prático consistiu no desenvolvimento de um sistema distribuído de armazenamento em memória de pares chave-valor. Com a sua realização foi possível consolidar a aprendizagem na unidade curricular de Fundamentos de Sistemas Distribuídos, nomeadamente na criação e análise de programas distribuídos que manipulam o estado global coerente recorrendo ao tempo lógico e a transações.

Na realização do projeto foi possível implementar as funcionalidades básicas do sistema, isto é, as operações GET e PUT, com facilidade. Porém, o maior desafio encontrado foi a decisão do mecanismo a utilizar para a garantia da consistência destas operações, quando executadas por vários clientes em simultâneo. Após algumas tentativas optou-se por implementar um coordenador de versões no sistema, uma vez que este guarda todas as versões dos valores das escritas para todas as chaves, permitindo que quando vários clientes escrevem na mesma chave e efetuam depois leituras dessa chave, o valor observado é o mesmo para todos os clientes. O coordenador antes de confirmar a versão de cada escrita ao cliente, adiciona essa versão a todas as chaves que irão ser atualizadas e partilha essa versão com todos os servidores. Assim, é também garantido que quando uma operação é marcada como completa, a escrita está mesmo terminada e não em trânsito. No entanto, considera-se que a implementação de um coordenador centralizado apresenta uma limitação, uma vez que, como este é responsável por coordenar todas as operações efetuadas, podem surgir gargalos no sistema.

Para além dos requisitos propostos, teve-se em conta as valorizações do trabalho, sendo que se procurou tornar o projeto modular, maximizando o código que é independente desta aplicação e adequado à utilização em grande escala. Ao implementar o coordenador de versões foi possível cumprir outra valorização, uma vez que este garante que leituras concorrentes de itens que estão a ser modificados por outros clientes não observam uma escrita parcial. De modo a não utilizar a biblioteca Atomix, foi desenvolvido um protocolo de envio e receção de mensagens, o que permitiu o cumprimento de mais uma valorização. É de salientar que apesar de termos utilizado nove servidores no sistema desenvolvido, o modo de implementação utilizado permite a sua escalabilidade, sendo possível aumentar ou diminuir o número de servidores, desde que as portas ocupadas sejam sucessivas.

Em suma, considera-se que foi possível aplicar todo o conhecimento adquirido ao longo do semestre e desenvolver um sistema distribuído que manipula o estado global coerente, mantendo a sua consistência.