Fundamentos de Sistemas Distribuídos
Keystore Distribuída

Daniel Fernandes, Maria Helena Poleri, and Mariana Miranda

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal
e-mail: {a78377,a78633,a77782}@alunos.uminho.pt

Resumo O presente documento tem o intuito de descrever as principais decisões de implementação de um sistema distribuído de armazenamento persistente de pares chave-valor. Este foi escrito em Java, utilizando a *framework* para sistemas distribuídos Atomix.

1 Introdução

Neste documento pretende-se descrever a arquitetura e as principais decisões de implementação de um sistema distribuído de armazenamento persistente de pares chave-valor. A implementação foi feita na linguagem de programação Java, utilizando a *framework* para sistemas distribuídos Atomix.

O sistema deve ser constituído por vários servidores responsáveis por armazenar de forma persistente as chaves e os seus respetivos valores. O cliente deverá poder utilizar o serviço de armazenamento de chaves mediante o uso das operações `get` e `put`. A operação `get` é responsável por ler os valores associados a um determinado conjunto de chaves, enquanto que a operação `put` deve escrever um conjunto de pares chave-valor, indicando se teve sucesso. A operação `put` deve ser realizada de forma atómica, garantindo que ou todos os pares chave-valor no pedido são modificados ou então nenhum é, mesmo em caso de falha de algum dos servidores. Quanto à operação `get`, não se colocam nenhumas restrições ao que os leitores concorrentes observam, devendo-se devolver parte das chaves pedidas ou uma exceção em caso de falha dos servidores.

Pretende-se ainda desenvolver uma solução modular e que, no caso de escritas concorrentes, os valores escritos em último lugar nas diferentes chaves provêm da mesma operação.

2 Arquitetura da Solução

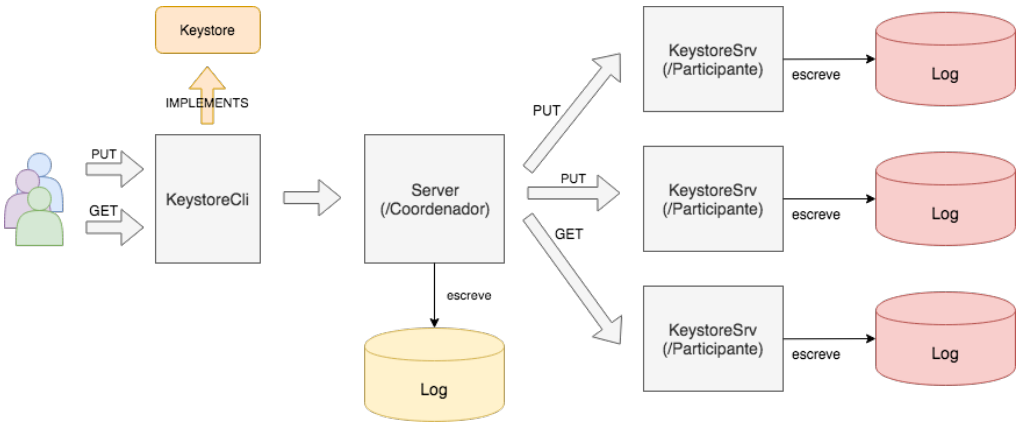

Figura 1. Visão geral da arquitetura da solução

A arquitetura adotada compreende três tipos de entidades principais: *KeyStoreCli*, *Server* e *KeyStoreSrv*, que resumidamente possuem as tarefas de fazer chegar os pedidos do cliente

ao servidor e obter as respostas, concretizar os pedidos estabelecendo comunicação com os participantes e garantir persistência e consistência dos dados (pares chave-valor) armazenados, respetivamente.

Relativamente ao *KeyStoreCli*, podemos apresentá-lo como sendo o *Stub* da aplicação. Este permite abstrair o cliente da distribuição no armazenamento das chaves, simulando a existência local dos recursos através da simples invocação dos métodos neste definidos. Todo o processo de estabelecimento de conexão com o servidor, troca de mensagens necessárias à satisfação dos pedidos, recai sobre este componente.

O *Server* por sua vez é quem recebe os pedidos dos clientes e trata de providenciar os meios necessários para que estes mesmos sejam realizados. Este sabe, por exemplo, que servidores de chaves é que existem e como distribuir os pedidos pelos mesmos de forma a obter uma resposta final. Para o caso dos pedidos `put` onde o protocolo 2PC tem de ser aplicado, este age também como coordenador da transação, mantendo o seu próprio *log*.

Tal como já foi mencionado, os *KeyStoreSrv* são as entidades onde as chaves se encontram armazenadas. Estes garantem que o seu estado se mantém consistente, mesmo na eventualidade de ocorrência de uma falha. Dão resposta aos pedidos `get` e agem como participantes do protocolo *Two-Phase Commit* nos pedidos `put`. Implementam o protocolo *Two-Phase Locking* para coerência dos dados.

2.1 Modularidade do código

Um dos principais desafios encontrados, centrou-se em conseguir com que o código seja modular, isto é, independente desta aplicação em concreto e que possa ser reutilizável noutros contextos. Para tal criou-se um *package* que apenas lida com o *Two-Phase Commit Protocol* e desconhece qual é o tipo de operação que está a ser efetuada, ou seja, que se trata de um sistema que armazena chaves do tipo `Long` a que estão associado valores do tipo `byte[]`. Dentro deste *package* encontram-se todos os recursos necessários para que seja possível efetuar na sua totalidade o 2PC, contendo as seguintes classes:

- *Coordinator* - implementa o coordenador do protocolo;
- *Participant* - implementa o participante;
- *Log* - permite o registo das alterações no sistema;
- *Phase* - define as fases de uma transação;
- *TwoPCProtocol* - protocolo de comunicação entre o coordenador e os participantes;
- *TwoPCTransaction* - regista os dados da transação;
- *SimpleTwoPCTransaction* - versão simplificada do `TwoPCTransaction` que contém apenas os dados que são de interesse para guardar no *log*.

Assim o *Server* apenas tem que declarar um objeto da classe *Coordinator*, e o *KeyStoreSrv* da *Participant*, e passamos a ter um protocolo de 2PC funcional. Por exemplo, o *Server*, ao receber um pedido de `put`, separa as chaves pelos participantes e invoca o início do protocolo na instância da classe *Coordinator* passando-lhe um *map* do tipo *<Integer , T>*, em que a *key* é o identificador do participante e o *value* um *map* com os pares chave-valor para aquele participante. Este, usando o *TwoPCProtocol*, envia os pares e realiza todas as interações com os participantes características do 2PC.

No entanto, deparamo-nos com um problema. Quando já sabemos se a transação aconteceu com sucesso ou não, é necessário enviar uma resposta ao *KeyStoreCli* e, como o objetivo é que este *package* apenas trate de operações do 2PC e seja genérico, foi preciso no momento de criação do *Coordinator* passar um `BiConsumer` definido pelo *Server*, que recebe o resultado e a transação e perante estes dados responde ao cliente.

O mesmo acontece do lado do participante, dado que existe código que é chamado pelo *Participant*, mas precisa de ser efetuado no *KeyStoreSrv*, tais como a aquisição de *locks* ou efetuar a transação. Para tal, foram definidos no *KeyStoreSrv* e passados ao *Participant* três comportamentos a tomar quando ocorre uma dada ação, sendo estes *prepare*, *abort* e *commit* e que são invocados pelo *Participant* consoante a situação.

Conseguimos assim ter um módulo completamente independente deste contexto e que pode ser reutilizado noutros situações.

3 Implementação

3.1 Operação `put`

Dado que é necessário que a operação `put` seja realizada de forma atómica e que ou todas as alterações sejam realizadas ou nenhuma é, para implementar esta operação recorreu-se ao *Two-Phase Commit Protocol* que, tal como explicitado no ponto 2.1, se encontra implementado de forma independente deste contexto.

O *Two-Phase Commit Protocol* é um algoritmo normalmente utilizado em bases de dados distribuídas e lida com transações atómicas distribuídas de forma a que exista um consenso entre todos os intervenientes de uma transação, isto é, ou todas ou nenhuma alteração é realizada.

Para tal, um dos processos é designado de coordenador e trata de gerir as transações e onde estas se irão efetuar são designados por participantes. Quer o coordenador, quer os participantes, suportam falhas temporárias no sistema, sendo que para tal fazem uso de *logs* onde vão registando as alterações que acontecem no sistema. Em caso de falha, o sistema lê do seu *log*, e aplica as regras definidas nos pontos 3.1.3 e 3.1.4.

Este algoritmo é composto por duas fases: a *voting phase*, na qual o coordenador pergunta aos participantes se estão preparados para a transação e a *completion phase* na qual, perante as respostas, o coordenador determina se a transação deve ser ou não efetuada.

3.1.1 Voting phase

Ao receber um pedido de `put`, o *Server* separa os pares chave-valor para cada um dos participantes, recorrendo a uma função de *hash* e passa ao coordenador para este iniciar o protocolo. Este, por sua vez, regista no seu *log* informações sobre este pedido e a marca de `STARTED` a simbolizar que a transação começou. De seguida, envia para os respetivos participantes cada conjunto de pares de chave-valor numa mensagem a perguntar se estão preparados para efetuar a transação.

O participante ao receber o pedido tenta adquirir *lock* das chaves envolvidas na transação, e quando o conseguir verifica se entretanto não recebeu nenhuma mensagem de abortar a transação. Caso não seja o caso, então envia a confirmação para o coordenador de que está preparado e regista no seu *log* os pares chave-valor e a marca `PREPARED`. Caso contrário, liberta os *locks* entretanto adquiridos.

3.1.2 Completion phase

Se todos os participantes responderam 'OK' ao coordenador, então este escreve no seu *log* `PREPARED` e envia uma mensagem para cada um dos participantes para efetuar a transação e aguarda a confirmação dos participantes.

Ao receber o pedido de *Commit* o participante efetua a transação, liberta os *locks* para as chaves envolvidas, regista no seu *log* que a transação foi realizada com marca `COMMITTED` e envia a confirmação ao coordenador.

Se, passado algum tempo, um dado participante ainda não tenha confirmado o *Commit*, então o coordenador envia de novo a mensagem de *Commit* ao participante, e repete de novo o processo até o *Commit* ser confirmado. Ao receber *Committed* de todos os participantes, então o coordenador escreve no *log* `COMMITTED`, sendo esta transação assim finalizada.

Caso algum dos participantes, passado um dado período de tempo, ainda não tenha respondido ao pedido de *Prepared*, então o coordenador cancela a operação, registando no seu *log* a marca `ABORT` e envia para todos uma mensagem para abortar a operação.

Ao receber o pedido de cancelamento, o participante liberta os *locks*, marca no seu *log* `ROLLBACKED` e envia confirmação ao coordenador. Tal como para o *Commit*, o coordenador verifica se cada uns dos participantes já confirmou o cancelamento e, se tal não acontecer, envia de novo o pedido de cancelamento até receber a confirmação. Ao receber confirmação de cancelamento de todos, então o coordenador pode finalmente escrever no *log* `ROLLBACKED`, finalizando assim a transação.

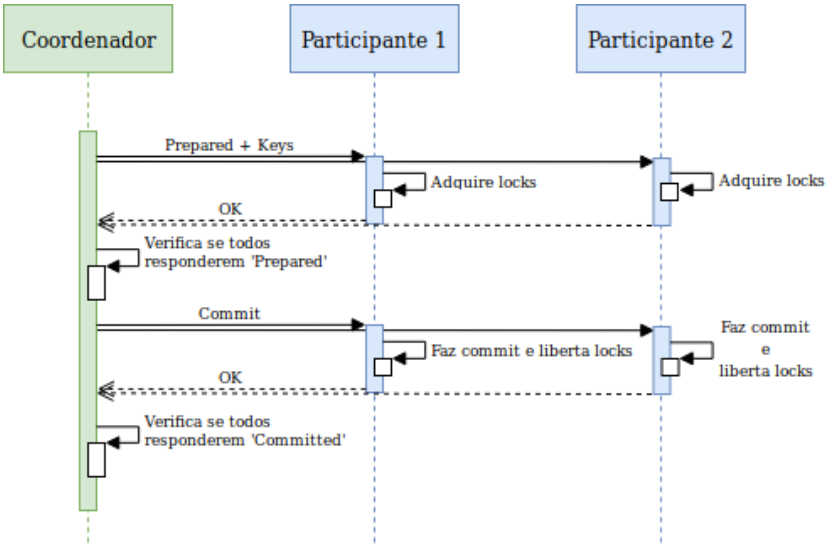

Figura 2. Transação efetuada com sucesso

3.1.3 Recuperação de falhas - Coordenador

Em caso de falha do coordenador este vai ler do seu *log* e dependendo da última fase registada para uma dada transação, toma uma dada decisão.

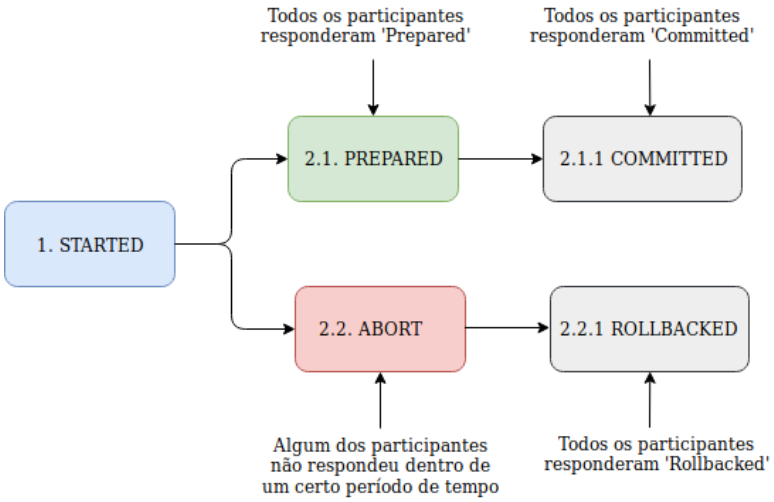

Figura 3. Fases de uma transação no Coordenador

- **Phase STARTED** - Se quando o coordenador falhou a transação se encontrava em fase `STARTED`, isto é, ainda não tinha recebido a confirmação do pedido *Prepared* de todos os participantes, nem decidiu cancelar a operação, então o coordenador reinicia a fase, isto é, envia pedidos de *Prepared* para os participantes, mesmo que já o tenha feito previamente.
- **Phase PREPARED** - Caso a transação se encontre em fase `PREPARED`, ou seja, o coordenador tenha recebido um resposta positiva ao pedido de *Prepared* de todos os participantes, então reinicia a fase, enviando *Commit* para todos os participantes.
- **Phase ABORT** - Se a última fase da transação é `ABORT`, significa que o coordenador decidiu cancelá-la, contudo ainda não recebeu confirmação de que todos os participantes cancelaram a operação. Assim, envia para todos os participantes a mensagem de *Abort* para aquela transação.

- **Phase COMMITTED** - Se a transação já se encontra `COMMITTED`, representa que a transação já acabou, então o coordenador não faz nenhuma operação.
- **Phase ROLLBACKED** - De forma similar à anterior, `ROLLBACKED` significa que a transação já terminou, então o coordenador ignora-a.

3.1.4 Recuperação de falhas - Participante

Tal como o coordenador, em caso de falha, dependendo da última fase registada no *log* para uma dada transação sabe que comportamento assumir.

- **Phase PREPARED** - O participante já respondeu *Prepared* ao coordenador, pelo que coloca a transação nas pendentes e adquire os *locks* das chaves.
- **Phase COMMITED** - Adiciona as alterações aos dados do sistema, uma vez que esta operação já se desenrolou com sucesso.
- **Phase ROLLBACKED** - Ignora a transação, dado que esta foi cancelada.

3.1.5 Garantias de Ordem nas Escritas

Caso dois clientes dessem início a uma operação de `put` sobre as mesmas duas ou mais chaves concorrentemente, poderia acontecer que, quer se encontrassem as chaves em servidores distintos, quer não, os últimos valores armazenados não proviessem da mesma operação. De forma a garantir que tal não acontecia, isto é, que os últimos valores fossem escritos pela mesma operação, foi necessário estabelecer ordem nas escritas que se intersetam. O método utilizado na implementação desta funcionalidade dá-se pelo nome de *Two-Phase Locking*. Seguindo os pressupostos deste protocolo foi necessário, antes de proceder à escrita de um qualquer conjunto de chaves, adquirir os *locks* para as chaves em questão, tendo o cuidado para que os pedidos aos participantes sejam sempre enviados na mesma ordem.

Dado que não se pode recorrer a *ReentrantLocks* na programação por eventos, foi necessário gerir uma fila de pedidos em espera e recorrer a *CompletableFuture*s. Assim, para cada entrada no *map* de chaves armazenadas por um servidor participante, existe, portanto, uma entrada para aquela chave em 2 outros *map*s. Um deles guarda se uma chave está ocupada e o outro contém uma fila dos pedidos de "*lock*" em espera para uma chave, tal como representado na Figura 4.

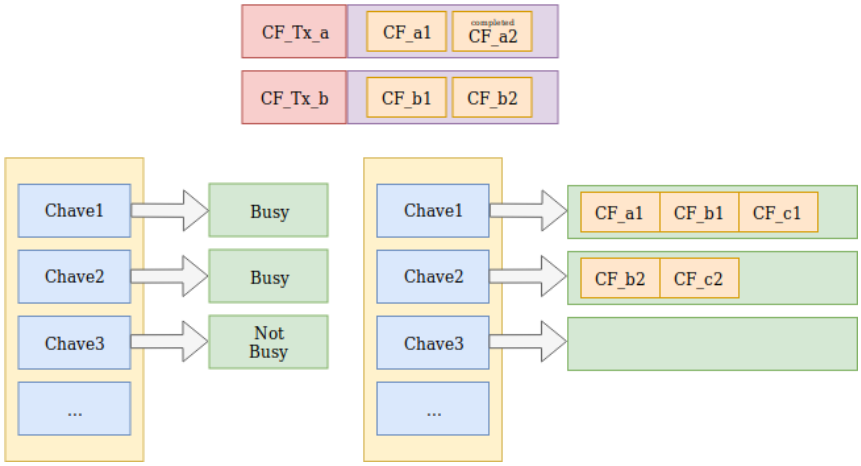

Figura 4. Funcionamento do sistema de locks

Na presença de um `put`, os identificadores das chaves envolvidas são percorridos ordenadamente, de forma a evitar *deadlocks*, tendo-se para isso utilizado um *TreeSet*. É criado ainda um *array* com um *CompletableFuture* para chave da transação, sabendo-se que foi adquirido o *lock* para uma dada chave, quando esse foi completado. Conseguimos saber ainda que todos os *locks* precisos para aquela transação foram adquiridos quando todos os *CompletableFuture*s do *array* foram completados.

Assim, para cada chave é verificado se esta está livre (*Not Busy*) e, caso esteja, coloca-a *Busy* e completa o respetivo *CompletableFuture*. Caso esteja ocupada coloca um *CompletableFuture* na fila de espera para aquela chave, que irá ser completado quando finalmente conseguir ter *lock* da chave.

Quando uma transação é abortada ou completada é preciso fazer *unlock* das chaves adquiridas, pelo que para cada uma das chaves envolvidas verifica-se se a fila de espera está vazia. Caso seja o caso, marca a chave como *Not Busy*, caso contrário, faz *complete* da primeira chave em fila de espera e retira-o da fila de espera.

É de notar que a fase de *locking* acontece pois, de forma implícita, como resposta ao pedido de *Prepared* e que na situação de a chave a inserir ser nova, não se encontrando no mapa de *locks*, é criada uma nova entrada nos mapas sendo o *lock* à partida adquirido. Uma vez todos os *locks* adquiridos e a operação não tenha sido cancelada entretanto, uma resposta ao pedido de *Prepared* é devolvida ao coordenador.

3.2 Operação `get`

Para a operação `get` não existia nenhuma restrição ao que o leitor observava, pelo que não foi praticado nenhum protocolo para garantir consistência. Assim, o *Server* ao receber um pedido de `get` separa as chaves para cada um dos *KeyStoreSrv* e envia o pedido com as chaves. O servidor de dados, ao recebê-lo, consulta as suas chaves guardadas e devolve as que possuiu. Ao receber a resposta de todos, o servidor devolve os pares chave-valor ao leitor. Se ao fim de um certo período de tempo não obter a resposta de todos os servidores de dados, então este devolve as chaves que conseguiu obter até ao momento e descarta qualquer resposta posterior.

Nos casos de falha do servidor principal, e o pedido de `get` não conseguiu ser efetuado, o cliente verifica passado um período de tempo se obteve resposta. Se não foi o caso, então é enviada uma exceção.

4 Conclusões e Trabalho Futuro

Neste trabalho prático foi implementado um sistema distribuído de armazenamento persistente de pares chave-valor. Este providencia uma API que permite ao programador de aplicações realizar pedidos sobre este, tais como operações de `put` e `get`. A operação do `put` foi implementada com sucesso de forma a que fosse atómica, da mesma forma que uma transação numa base de dados relacional. Esta implementação, adicionalmente, garante ainda que caso existam escritas concorrentes nas mesmas duas ou mais chaves, os valores escritos em último lugar nas diferentes chaves provêm da mesma operação.

Este permitiu-nos assim aprofundar o nosso conhecimento no contexto de programação por eventos, assim como consolidar os algoritmos de *Two-Phase Commit* para consistência e *Two-Phase Locking* para controlo de concorrência em sistemas distribuídos. Assim, julgamos ter construído uma *keystore* distribuída em bom funcionamento com as operações mais essenciais, capaz de ser extensível a outras. Sendo que conseguimos ainda uma implementação genérica do algoritmo 2PC, será ainda possível reutilizar este código noutros contextos.

Como trabalho futuro, seria de interesse testar o protocolo de *Two-Phase Commit* implementado noutros contextos de forma a comprovar a modularidade deste, assim como estender a API disponível a outras operações. Para além disso, implementar uma arquitetura que no caso de falha do servidor principal efetuasse a escolha de um novo líder, de modo a proporcionar uma maior disponibilidade ao sistema. Outra possível melhoria consistia em usar um servidor de base de dados para guardar os pares chave-valor já *committed*, de forma a que não fosse necessário uma leitura do *log* inteiro no momento de recuperação.