Universidade do Minho

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Fundamentos de Sistemas Distribuídos

Distributed Key-Value Store

Diogo Figueiredo Pimenta A75369
Isabel Sofia da Costa Pereira A76550
Maria de La Salete Dias Teixeira A75281

04 de Janeiro de 2019

Conteúdo

1 Arquitetura do Sistema

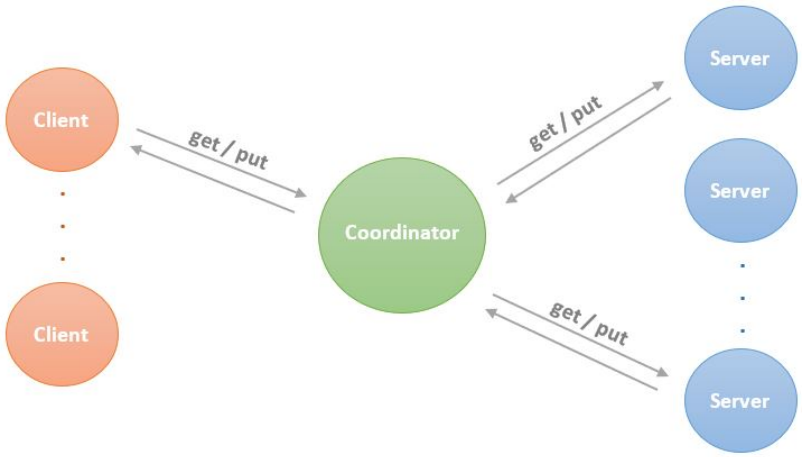

Figura 1: Arquitetura do sistema

Sendo o objetivo deste trabalho o desenvolvimento de um sistema distribuído de armazenamento persistente de pares chave–valor, o grupo optou por criar o sistema seguindo a arquitetura apresentada na figura 1. Deste modo, foi necessário desenvolver um coordenador, clientes e servidores.

O coordenador é responsável por receber os pedidos dos clientes, encaminhar os pedidos para os servidores necessários, receber as respostas dos servidores e reencaminhar estas para os respetivos clientes. Cada servidor armazena uma parte do *Hash Map*, pelo que, para saber que servidores são necessários para dar resposta a um pedido, o coordenador calcula o resto da divisão da chave pelo número de servidores. Para além disso, é também o coordenador o responsável por efetuar o *Two Phase Commit*, que será explicado num dos tópicos que se segue. A comunicação entre o coordenador e os servidores foi isolada num *stub*, dando a ideia ao coordenador que é ele próprio a dar conta dos pedidos dos clientes.

Relativamente aos servidores, tal como já foi referido, cada servidor é responsável por parte do armazenamento persistente dos pares chave-valor. Estes apenas comunicam com o coordenador, dando resposta a pedidos de *put*, *get* e de *Two Phase Commit*. Os servidores correspondem aos participantes no *Two Phase Commit*, sendo que implementam um *log* para garantir a persistência e coerência dos dados.

Por último, tem-se os clientes, que efetuam pedidos de *put* e *get* ao coordenador. Para desenvolver a classe *Client* implementou-se um *stub*, o que permite isolar a parte que envia e recebe os pedidos, ou seja, a parte correspondente

a sistemas distribuídos. Isto dá ao cliente a ilusão de que os pedidos são processados localmente, sem recorrer a outras entidades, dando a ideia de que o *Hash Map* está armazenado na máquina do cliente. Tal como foi sugerido pelo docente, não foi desenvolvida nenhuma interface para o cliente, sendo os testes efetuados na *main* do mesmo.

Os pedidos *get* são respondidos sem qualquer restrição, não sendo tomadas medidas em caso de exceção ou bloqueio. No caso do *put*, sendo este o pedido que leva a alteração dos dados, são tidos mais cuidados, sendo a seguir à resposta a um *put* que se efetua o *Two Phase Commit*.

Todo o tipo de comunicação entre cliente-coordenador e coordenador-servidor foi realizada utilizando *MessagingService*. Para dar resposta aos pedidos dos clientes e ao processo de *Two Phase Commit*, recorreu-se ao *Completable Future* do *Java*, o que faz com que o coordenador consiga aguentar com um largo número de pedidos. Assim, como o coordenador é orientado a eventos, este suspende e só acorda quando recebe um pedido, realizando as operações associadas a este. Desta forma, o coordenador ao receber um pedido de um cliente, pode avançar para outras ações e só responder-lhe quando obtém a resposta dos servidores, ao contrário de ficar bloqueado à espera da resposta.

É também importante referir que, sendo o nosso sistema guiado por um único coordenador, não houve necessidade de implementar *locks*, pois o coordenador processa um pedido de cada vez, não havendo risco de incoerência nos pedidos.

2 Two Phase Commit

O *Two Phase Commit* permite, juntamente com a aplicação *log*, fazer *backup* e gerir as transações presentes no coordenador e nos servidores. Como só a operação *put* é que era requerida ter estado persistente, só as transações associadas a esta é que são guardadas em *log*.

Sendo assim, relativamente ao servidor, quando este recebe um *put*, coloca o id da transação e os valores do *put* em log com o *status* a ¨¨. Este estado representa que o *Two Phase Commit* ainda não foi inicializado para essa transação. Caso este receba um pedido de *prepared*, modifica o *status* para **P** e responde ao coordenador *yes* ou *no*, caso queria abortar. No caso de receber o pedido de *commit* ou *rollback*, este coloca, respetivamente, o *status* a **C** ou **A** e avisa o coordenador que acabou o processo com a mensagem *finished*. No entanto, quando este executa um *rollback*, o servidor atualiza ainda o *map*, anulando a transação em questão e atualizando as chaves contidas neste.

O coordenador, por outro lado, sempre que recebe um *put* de um cliente associa um id de transação a esse pedido e calcula os servidores que serão encarregues dessa transação. Desta forma, obtêm-se os participantes do *Two Phase Commit* para a transação em questão. Com isto, o coordenador guarda em *log* esses dois dados, juntamente com o status a ¨¨, o endereço do cliente e o *request* id do cliente associado a essa transação. Os dois últimos valores são armazenados caso seja feito *restart* do coordenador, fazendo com que os *completable futures* não possam ser utilizados, mas seja possível à mesma responder ao cliente. Após

esta escrita em *log*, o coordenador envia o pedido para os vários participantes. Quando estes todos lhe responderem ao pedido, o coordenador dá inicio ao processo de *2PC*, onde este último devolve um *completable future*. O processo fica então encarregue de gerir todas as massagens relativas ao *2PC*, aplicando o papel de coordenador, e quando termina devolve ao coordenador o *boolean* que este tem de retornar ao cliente. Sendo assim, começa-se por enviar a mensagem de *prepared* aos participantes. Quando estes respondem pode gerar dois casos: um seria quando todos respondem *yes*, o que leva o coordenador a mandar *commit* para todos e mudar o estado para **C**; o outro caso acontece quando pelo menos um dos participantes responde *no*, o que leva o coordenador a mandar *rollback* para todos e mudar o estado para **A**. De seguida, o coordenador espera receber *finished* de todos os participantes para assim dar como terminada a transação, mudando o estado para **F** e devolvendo a reposta associada ao *put* do cliente. O *boolean* irá a *true* se a transação finalizou com *commit* ou a *false* se finalizou com *abort*.

Sempre que o coordenador envia um pedido de *prepared*, *commit* ou *rollback* aos participantes, este começa a contar o tempo que os mesmos têm para lhe responder ao pedido correspondente. Se todos já tiverem respondido dentro do tempo esperado, o processo decorre normalmente. Caso contrário, o coordenador repete o pedido pela última vez. Ao repetir o pedido pode acontecer duas coisas, ou os participantes respondem a tempo nesta segunda tentativa e decorre o processo normal, ou os participantes voltam a não responder a tempo e o coordenador vê-se obrigado a fazer *rollback* e *finished* da transação (*status* passa a **A** e depois a **F**), respondendo imediatamente ao cliente com *put* igual a *false*.

Como o grande objetivo do *log* é pode haver *backups*, também o *Two Phase Commit* apresenta essa opção.

Ora, quando se trata do servidor, este poderá fazer o *restart* e obter transações com *status* igual a ¨¨, **P**, **C** ou **A**. Como este possui um *map*, torna-se necessário restaurar o mesmo aplicado sobre este apenas as transações com *status* igual a ¨¨, **P** e **C**. Os *status* ¨¨ e **P** são considerados pois ainda podem resultar em *commit*, como veremos de seguida. Relativamente ao *restart* das transações, o servidor aplica um *rollback* interno caso o *status* seja ¨¨, passando então este para **A**. Isto acontece porque a transação não foi votada, provavelmente porque perdeu o pedido de *prepared*. Nesta situação, como o servidor nunca mais responde ao pedido, o coordenador fá-lo novamente ou pede o *rollback* caso já tenha passado o tempo estipulado. Para o *status* **P**, o servidor limita-se a esperar a decisão do coordenador, pois já enviou o seu voto. Isto sucedesse quando a sua resposta é perdida (coordenador repete o pedido de *prepared*) ou este não recebe o pedido de *commit* ou *rollback* por parte do coordenador. Nesta última situação, o coordenador executa o pedido correspondente pela primeira ou segunda vez. Por último, para os *status* **C** e **A**, o servidor não tem de aplicar nenhuma ação pois as transações já se encontram supostamente finalizadas. Se não se encontrarem, o coordenador pede a repetição do pedido.

O *restart* do coordenador faz com que este tenha de atualizar o número da última transação, para que assim as futuras não possuam identificadores

repetidos. Para além disso, este pode conter transações com *status* igual a ¨¨, **C**, **A** ou **F**, tendo de aplicar o devido restauro a cada uma delas. No caso de se apresentar com estado ¨¨, o coordenador executa *rollback* sobre essa transação, colocando o estado a **A**. Isto acontece porque o coordenador pode não ter mandado o pedido de *prepared* para todos os participantes. Quando o estado se encontra a **C** ou **A**, o coordenador repete o pedido correspondente porque os participantes podem nem todos ter executado essa ação. Para o estado **F**, o coordenador não tem de executar nada pois a transação já se encontra finalizada.

Com este mecanismo consegue-se então uma boa gestão das transações.

3 Gestão de Concorrência

Tendo em conta a arquitetura considerada, em que se tem um único coordenador responsável por gerir todos os pedidos, não haverá qualquer problema com pedidos concorrentes. Isto porque, o coordenador recebe os pedidos, mas processa um pedido de cada vez, sendo que não se corre o risco de haver mistura de informações nas escritas.

Para além disso, para aplicar uma ordem aos pedidos, sempre que é recebido um pedido é atribuído ao mesmo um *transaction id*. Já nos servidores, para além do *map* chave-valor necessário, também é guardado um *map* em que a cada chave se associa o id da última transação a alterar o valor da mesma. Este *map* é guardado para permitir que o valor de uma chave seja alterado apenas se o *id* da transação for superior ao *id* da última transação que alterou aquela chave.

Assim, na presença de duas escritas concorrentes e1 = {a=1,b=1,c=1} e e2 = {a=2,b=2,d=2}, o coordenador recebe os pedidos, associa um *transaction id* a cada um e processa cada pedido. Os servidores envolvidos em cada pedido verificam o *id* da última transação que alterou a chave a considerar e, em caso de este ser inferior ao *id* da nova transação, altera o valor associado à chave e atualiza o *id* no *map* das chave-transação. Deste modo, os valores armazenados após o processamento dos pedidos referidos seriam {a=1,b=1,c=1,d=2} ou {a=2,b=2,c=1,d=2}, dependendo de qual o pedido com o maior *transaction id*.

4 Modularidade

Toda a arquitetura do sistema foi, desde o inicio do planeamento, desenhada tendo em conta o objetivo desta ser modular e reutilizável, encapsulando os mecanismos de distribuição do sistema e de garantia de coerência nas escritas efetuadas pelos diferentes servidores. Como principal exemplo, o mecanismo de *Two Phase Commit* está encapsulado em objetos do mesmo nome que lidam com toda a comunicação relativa ao mecanismo, com a gestão das transações e com a sua escrita em *log*. Deste modo, este mecanismo pode ser facilmente adaptado e transferido para outros contextos, sendo apenas necessário alterar o que é escrito no *log*.

5 Escalabilidade

A modularidade especificada na anterior secção, assim como a facilidade de acrescentar ou remover servidores da arquitetura desenvolvida torna este sistema distribuído altamente escalável. Apesar disso, com um número extremamente elevado de clientes e pedidos a ocorrer simultaneamente, é possível existir um *bottleneck* na capacidade deste sistema, na entidade Coordenador. Uma possível mitigação deste gargalo será a atribuição de uma *thread pool* ao Coordenador, aumentando a sua capacidade de tratar pedidos.

6 Casos de Teste

Para a verificação da correta implementação do programa, executaram-se meticulosamente vários cenários de teste. Com intuito de facilitar esta analise, utilizou-se o ficheiro *Teste2PC_Simple* como cliente pois este contem sempre as mesmas transações, enquanto que o ficheiro *cliente* gera *puts* e *gets* aleatórios. Além disso, utilizou-se um tempo máximo de resposta de 1 milissegundo.

Primeiramente, testou-se questões que podem acontecer em todos os cenários, nomeadamente, as escritas concorrentes, a atualização do *map* dos servidores quando estes fazem *abort* de uma transação, a execução normal do *2PC* e a repetição de pedidos por parte do coordenador quando os participantes demoram a responder. Para o primeiro teste, criou-se um caso especifico (*TesteEscritas-Concorrentes*) de forma a forçar o servidor a receber primeiro a transação com maior id e só depois a de menor id. Relativamente aos três últimos testes, estes podem ser conjugados quando se aplica um tempo muito curto para os servidores responderem, forçando assim a repetição. Para forçar o *abort* foi necessário colocar os servidores a dormir 5 segundos quando recebem a mensagem *commit*, por exemplo.

De seguida, elaboraram-se testes para o *restart* dos servidores quando estes se encontram com *status* igual a **C** ou **A**, **P** e ¨¨. Para testar os dois primeiros estados, apenas foi necessário correr normalmente o ficheiro mencionado e fazer *restart* no fim deste ser executado. No caso do estado **P**, colocou-se os servidores a dormir 10 segundos quando recebem o pedido *commit* ou *abort*, para se saber quando aplicar o *restart*. Para o teste do estado ¨¨, aplicou-se a mesma ideia do anterior, isto é, colocou-se os servidores a dormir 10 segundos quando recebem a mensagem *prepared*. Considerando estes dois últimos casos, utilizou-se um tempo de resposta de 15 segundos para o coordenador não fazer imediatamente o *rollback* das transações.

Por último, realizaram-se testes ao *restart* do coordenador quando este se encontra com *status* igual a **F**, **C** ou **A**, e ¨¨. Para testar o primeiro estado, apenas foi necessário correr normalmente o ficheiro mencionado e fazer *restart* no fim deste ser executado. No caso do estado **C** ou **A**, colocou-se o coordenador a dormir 5 segundos quando recebesse a mensagem de *finished*, para se saber quando aplicar o *restart*. Para o teste do estado ¨¨, aplicou-se a mesma ideia do anterior, isto é, colocou-se o coordenador a dormir 5 segundos antes de enviar

a mensagem *prepared*.

Com isto, verificou-se que:

1. O servidor não aplica às chaves, que já possuem valores da transação com id maior, os valores da transação com id menor, porque o servidor deve considerar sempre a transação válida mais recente;
2. O *map* do servidor apaga as entradas que são abortadas, ficando com os valores da transação mais alta válida para cada chave;
3. O coordenador e os participantes executam o processo normal de *2PC* como explicado na secção **2**;
4. O coordenador executa as repetições esperadas quando os servidores demoram a responder podendo até levar ao *rollback* forçado;
5. No *restart* dos servidores, estes efetuam o *restart* do *map* e das transações, sendo que os próprios e o coordenador executam as operações esperadas. Estas operações encontram-se esclarecidas na secção **2**;
6. No *restart* do coordenador, este aplica corretamente as ações explicitadas na secção **2** e os servidores respondem aos pedidos efetuados;
7. O cliente recebe *true* quando as transações são *commited* e *false* quando são *aborted*.

Todos estes testes podem ser encontrados na pasta *Testes-Resultados*, juntamente com os seus resultados.

7 Apreciação Crítica

Na realização deste trabalho foi possível aplicar vários dos conhecimentos adquiridos na cadeira, tendo-se usado *Messaging Service*, *Completable Future*, *logs*, *Two Phase Commit*, entre outros. Mesmo a matéria não aplicada, como por exemplo o *Causal Delivery*, consideramos que foi estudada e melhorada a sua aprendizagem durante o projeto, pois compreendemos a razão de não ser necessária a sua utilização e, consequentemente, as vantagens que trariam o uso desses conceitos.

Como trabalho futuro poderíamos tentar uma melhoria na arquitetura, pois neste momento consideramos que a resolução proposta tem um gargalo, por todos os pedidos terem que passar por um único coordenador, no entanto, não consideramos ser um gargalo demasiado grave que invalide o sistema.

No geral, não apontamos grandes dificuldades na resolução do trabalho, sendo que a maior dificuldade que tivemos que ultrapassar foi no caso de falha do coordenador, pois quando o mesmo faz *restart* perdia-se os *Completable Future* para dar resposta aos clientes. No entanto, conseguiu-se dar a volta ao problema, tal como está descrito no tópico *Two Phase Commit*.

Tendo-se concluído o trabalho, consideramos que a sua resolução foi bem sucedida e que está de acordo com o esperado.