Mestrado Integrado em Engenharia Informática

Fundamentos de Sistemas Distribuídos

Trabalho Prático - Relatório de Desenvolvimento

Jorge Oliveira (A78660)
José Ferreira (A78452)
Nuno Silva (PG38420)

23 de Abril de 2020

Conteúdo

Capítulo 1

Introdução e Exposição do Problema

O presente relatório serve para expor as abordagens utilizadas bem como a implementação de um sistema distribuído de uma base de dados chave-valor. O funcionamento deste sistema é bastante simples e um utilizador pode efetuar duas operações, uma de **PUT** que serve para que guarde valores associados a uma chave que deseja, e um **GET** onde poderá observar quais os valores associados às chaves que requisitou.

O objetivo do grupo com este trabalho é conseguir aplicar os conhecimentos teóricos aprendidos nas aulas e sermos capazes de os utilizar na prática. Para além do básico, que é implementar a as operações de PUT e GET utilizando o *2-Phase Commit*, houve uma preocupação para que o sistema funcionasse a uma grande escala, fosse modular e que existisse uma ordem. Podemos então apresentar os nossos objetivos de uma forma sucinta:

1. Realizar com sucesso a produção das operações básicas de GET e PUT, utilizando o *2-Phase Commit*
2. Garantir uma ordem das transações, através do *2-Phase Locking*, mas evitando que possa criar gargalos a uma grande escala
3. Tornar imparcial ao nosso sistema as operações que o mesmo tem de realizar, isto é, neste caso de estudo em concreto é para guardar valores, mas caso fosse para efetuar qualquer outro tipo de operação, então o nosso sistema base era capaz de o fazer. Isto implica que as operações sejam escolhidas pela entidade que pretende utilizar o nosso sistema;

Capítulo 2

Desenvolvimento

2.1 Abordagem Escolhida

O cliente do sistema interage com o mesmo através de um **stub**, sendo associado um por cada cliente a aceder ao sistema. As principais responsabilidades do **stub** são receber as invocações por parte do cliente e escolher um servidor com quem comunica para despoletar uma operação distribuída de leitura ou escrita. Quando essa operação termina, o **stub** reencaminha o resultado da mesma para o cliente. O **stub** oferece a seguinte API ao cliente:

- CompletableFuture<Boolean>put(Map<Long,byte[]>values): Escreve um conjunto de pares chave–valor, indicando se teve sucesso.
- CompletableFuture<Map<Long,byte[]>get(Collection<Long>keys): Lê os valores associados a um conjunto de chaves. Devolve uma exceção no *CompletableFuture* caso não consiga obter o get passado um *timeout*.

O cliente pode invocar estes 2 métodos utilizando o **stub**. Exemplos de utilização estão presentes na classe Cliente.

Das duas operações para serem implementadas, o **PUT** requeria uma atenção especial da nossa parte, já que é uma operação que modifica o conteúdo do sistema. Para garantir que era realizado com sucesso, decidimos utilizar transações. Como os valores, associados às diferentes chaves, estão separados por vários servidores, então é necessário que numa operação de **PUT**, onde possivelmente as chaves a modificar estão guardadas em locais diferentes, todas as modificações sejam atómicas, e onde se assegure que ou todos os valores são introduzidos/alterados ou então nenhum é, tendo de efetuar um *rollback* em alguns casos. Para efetuar as transações usamos o protocolo *2-Phase Commit*. Utilizando esta abordagem, tivemos de implementar dois papéis distintos, o **Participante** que tem como principal tarefa executar a ação (no caso em estudo é guardar o par chave-valor), ao passo que o **Coordenador** é responsável por coordenar a transação entre os diferentes participantes, e certificar-se de que todos os Participantes tomem a mesma decisão.

De modo a evitar os gargalos no sistema decidimos não centralizar o nosso Coordenador, podendo qualquer um dos servidores desempenhar este papel durante uma transação. A primeira escolha do Coordenador é feito de forma aleatória, mas de seguida todas as outras é aplicada um algoritmo de *round robin*. De salientar, que na mesma transação um mesmo servidor pode desempenhar o papel de Coordenador e também de Participante. Esta escolha do coordenador é feita pelo *stub* aquando de um pedido *PUT* ou *GET*.

Face à possibilidade de estarem a ocorrer em simultâneo múltiplas transações, tivemos de implementar um sistema de *lock* para garantir a ordem. De forma a garantir que um Participante não fique bloqueado apenas numa transação, decidimos aplicar esta estratégia ao nível das chaves. Desta forma garantimos, que duas transações com chaves distintas mas que partilhem o mesmo Participante podem ser executadas em simultâneo. Este *lock* é implementado utilizando o protocolo do *2-Phase Locking*, já que é útil para quando queremos utilizar *locks* distribuídos.

Poderíamos optar por duas estratégias, efetuar primeiro o *Two-Phase Locking* e depois o *Two-Phase Commit* ou então utilizar o *Two-Phase Locking* implicitamente com o *Two-Phase Commit*. Optámos pela segunda apesar de não garantirmos total modularidade, que conseguíamos alcançar com a primeira já que a utilização dos dois protocolos seria totalmente independente, mas assim conseguimos que a chave que queremos alterar tenha uma taxa de utilização útil (se usássemos a primeira estratégia enquanto estávamos na fase do *Two-Phase Locking* não iríamos estar a alterar

a chave) e ainda reduzimos o número de mensagens existentes durante uma transação, conseguindo que a rede fique menos sobrecarregada, podendo mais uma vez evitar gargalos e garantir que funciona a uma larga escala.

Para tornar a nossa implementação do *Two-Phase Commit* genérica, decidimos ocultar nos Participantes a ação que os mesmo têm de realizar. Desta forma garantimos, se caso fosse necessário alterar o conceito da aplicação só era necessário modificar apenas uma parte e continuar a usar o *Two-Phase Commit* de igual forma.

Já no caso do **GET**, é uma operação muito mais simples visto não alterar os dados existentes e ser apenas uma operação de consulta. No entanto, é necessário saber em que servidores se encontram armazenadas as chaves incluídas no pedido e comunicar com esses mesmos servidores para recuperar esses valores. Essa comunicação pode ser interrompida por alguma falha e reinício dos servidores. Caso isso aconteça, o grupo deliberou que o sistema deveria devolver uma exceção ao cliente que invoca a operação de leitura.

2.2 Implementação

2.2.1 Operação Get

A implementação da operação **Get** foi bastante mais simples que a operação **Put** por não serem necessárias transações. Um dos servidores terá o papel de coordenar a operação. Este receberá o conjunto total de chaves associadas ao pedido **Get** e utilizando o mecanismo usado para a divisão das chaves por servidor, divide o conjunto inicial num novo conjunto de chaves a enviar a cada um dos servidores participantes. Envia a estes o conjunto respetivo e espera pelas respostas. Quando um servidor recebe esse pedido vindo de servidor "coordenador", vai ao seu mapa e envia de volta um novo mapa com as chaves pedidas e os seus respetivos valores. Quando todos os participantes da operação tiverem informado o servidor "coordenador"das suas respostas, estes junta todos estes mapas num só e devolve-o ao **stub**. Caso a operação não termine num período definido, é devolvida uma exceção no CompletableFuture enviado ao cliente, avisando-o que a operação excedeu o tempo limite. As mensagens a serem trocadas pelos intervenientes são:

- "get- mensagem vinda do stub para o coordenador da operação, contendo a coleção de chaves associadas à operação de leitura;
- "getCoordenador- mensagem vinda do coordenador para um servidor participante, contém a coleção de chaves para as quais este deve devolver os valores que possui;
- "getResposta- mensagem do coordenador para o **stub** que vai acompanhada do mapa resultado da operação;
- "getExcecao- mensagem do coordenador para o **stub** que avisa o mesmo que a operação excedeu o tempo limite;

2.2.2 *Two-Phase Commit*

A nossa implementação do *Two-Phase Commit* é muito parecida à tradicional, tendo apenas as seguinte diferenças:

- As mensagens trocadas entre coordenadores e participantes, devido a um mesmo servidor poder desempenhar ambos os papeis, têm de ter assuntos diferentes de um lado para o outro, para garantir que a mesma chega ao local correto.
- É dada uma tentativa à transação, para esta se realizar e após um *timeout* é verificado o resultado da mesma. Se o resultado ainda for **I** (nem todos os participantes deram uma resposta), então a transação é concluída com o resultado **A**(*abort*).

As mensagens trocadas no sentido dos coordenadores para os participantes são:

- "preparedCoordenador- denota um início da transação, pergunta ao participante se está preparado para a realizar
- "abortCoordenador- indica que a transação que vai na mensagem foi abortada e por isso não deve sofrer alterações
- "commitCoordenador- informa que a transação que vai na mensagem é para ser realizada, alterando os valores

As mensagens trocadas no sentido dos participantes para os coordenadores são:

- "prepared- indica que participante está preparado para realizar a transação
- "abort- informa o coordenador que o participante não pode realizar a transação
- "ok- indica que o resultado da transação está confirmado e que a mesma se encontra terminada para o participante

A nossa implementação do *Two-Phase Commit* contempla uma implementação separada para os coordenadores e para os participantes, sendo os mesmos independentes.

2.2.3 Garantia de ordem

Para a garantia de ordens das escritas tínhamos algumas soluções, como por exemplo:

- *Replicated State Machine*, no entanto esta envolvia uma troca de mensagens constantes com todos os coordenadores pelo que decidimos não a utilizar
- *Locking* com um coordenador central, onde o *lock* é mantido por esse coordenador, no entanto ficamos com um *overhead* muito grande nesse mesmo coordenador de locks
- Utilizar uma variante de *Two-Phase Locking*, no qual o *Client* e o *Manager* são o mesmo interveniente, sendo o coordenador da transação. Este coordenador sabe à partida todos os participantes envolvidos na mesma, o que também é uma simplificação.

A estratégia que nos pareceu mais adequada e simples de implementar foi uma variante de *Two-Phase Locking*. Para implementar esta estratégia decidimos que a mensagem de *prepared* seria um *lock* implícito e que a mensagem de *abort/commit* seria um *unlock* implícito.

Definimos o critério de *locking* como sendo ao conjunto de chaves pedidas na transação, não sendo apenas ao participante total. Isto faz com que a granularidade do *lock* seja um pouco mais fina, permitindo que um participante possa ter várias transações ao mesmo tempo, desde que elas não envolvam as mesmas chaves. Isto permite algumas vantagens como a realização de mais do que uma transação, mas também traz desvantagens que serão abordadas futuramente.

Cada participante mantém uma fila dos *locks* atuais, bem como uma fila de espera dos *locks* requeridos para o mesmo, estando esta organizada por ordem de chegada. Para além disto contém também uma lista de valores a serem alterados no momento. O participante executa as seguintes operações ao receber as seguintes mensagens:

preparedCoordenador Invoca o método *lock(Lock)*, que verifica se os valores a serem alterados pela transação são disjuntos dos valores a serem alterados de momento. Caso sejam disjuntos adiciona os valores alterados pela transação aos valores a serem alterados no momento. Caso contrário adiciona o *lock* à fila de espera.

abortCoordenador Invoca o método *unlock(Lock)*, que verifica se o *lock* está contido nos atuais. Caso esteja, retira os valores que aquele *lock* alterava dos valores a serem alterados e retira o mesmo da fila de dos atuais, verificando de seguida se pode adquirir mais algum dos *locks* em fila de espera. Caso contrário retira apenas o *lock* passado da fila de espera.

commitCoordenador Pela nossa implementação, só pode chegar uma mensagem de *commit* ao participante quando este tenha respondido *prepared*. Isto implica que ao receber a mensagem de *commit* tenhamos certeza que temos o *lock* adquirido para essa transação. Por isso o participante altera o que tem a alterar e invoca o método *unlock(Lock)* referido anteriormente, com a garantia que o *lock* está na lita dos atuais.

Para além do descrito acima convém referir que antes de adquirir ou remover qualquer *lock* o participante verifica se já tem uma resposta para essa transação e em caso afirmativo apenas responde, não realizando mais nada. Isto está correto pelo facto de que se existir um resultado para uma transação então é porque o *lock* já foi liberto.

Convém também referir que cada *lock* é identificado pelo identificador da transação.

Coordenador

Para além disso definimos uma ordem nos participantes para a obtenção de *locks*, sendo que apenas passamos ao próximo aquando da obtenção do anterior. Como são todos garantidos pela mesma ordem, existe sempre um que os garante e os outros ficam à epera não existindo deadlocks.

Para garantir mesmo que não existem *deadlocks* e garantir a ausência de *starvation* a transação tem um *timeout* pelo que o coordenador eventualmente vai acabar com a mesma no estado *abort*. Após a transação ter o resultado *abort* existe um evento que faz com que os outros *"locks"* que estavam à espera se dissipem.

2.2.4 Recuperação

Temos sempre de ter em consideração que tanto o coordenador como o participante podem ficar *down* e que no momento do seu reinicio é necessário carregar de novo todas as transações para que o seu estado, antes deste *restart*, seja preservado.

Na recuperação do estado do coordenador era carregado para a memória os resultados das transações registadas no seu *log*. De salientar, que se no caso existir um *prepared* ao qual já contêm uma consequência do mesmo, quer seja um *commit* ou então um *abort* ou finalizada, o *prepared* é apagado da memória e é mantido o consequente, para que depois possa ser efetuada uma análise. Após o carregamento das transações é necessário executar uma ação tendo em conta o estado atual da transação, se for *prepared* é enviado uma mensagem, com o mesmo assunto, para todos os participantes daquela transação e é iniciado um *timeout* para que aquela transação passe para a próxima fase. No caso de ser *abort* ou *commit* é bastante parecido com o caso anterior mas por conseguinte não é iniciado o *timeout* (já que neste caso temos de ter a certeza que todos respondem) é enviada uma mensagem com o mesmo assunto para todos os participantes envolvidos. Por fim se tiver o estado de **finalizada** então quer dizer que já não é preciso executar qualquer ação para esta transação.

Na recuperação do estado do participante seguimos uma lógica idêntica à do coordenador, onde efetuámos primeiro um carregamento de todas as transações para a memória e depois é efetuado um reenvio das mensagens para o coordenador correspondente. Para além disto, é necessário guardar os valores das respetivas chaves, no caso da transação ter ocorrido com sucesso. Faltava ainda carregar também o estado dos *locks* do participante, que é efetuado aquando o aparecimento de uma transação que apenas ainda se encontra com o estado *prepared*, no caso de um *commit* ou *abort* já há certezas de que o *lock* foi libertado. Como o LOG já se encontra ordenado, não temos qualquer problema para garantir a ordem dos *locks*, já que existe a certeza de que se o lock A aparecer primeiro que um lock B no LOG, então essa foi mesmo a ordem de pedido dos *locks* antes de serem guardados no LOG.

2.3 Modulação e independência

Para garantir uma independência entre o *Two-Phase Commit* e a nossa aplicação decidimos que tanto as transações como as entradas no *log* guardavam valores o mais gerais possíveis, ou seja um Objeto. Para garantir também que o *Two-Phase Commit* sabe como efetuar as operações necessárias para o seu bom funcionamento definimos as seguintes interfaces:

```
import java.util.HashMap;
public interface InterfaceControlador {
    HashMap<Address,Object> distribuiPorParticipante(Object o); //funcao que da os participantes
    // de uma transacao

    Object juntaValores(HashMap<Address,Object> o); //funcao inversa à anterior

    HashMap<Address,Object> participantesGet(Object o); //funcao que devolve
    // os participantes de um get
}
```

Figura 2.1: Interface guardada no controlador

```java
public interface InterfaceParticipante {

    public Object devolveValores(Object chavesInput); //funcao que devolve os valores
    // consoante as chaves, importante para o get

    public Object atualizaValores(Object novos); //funcao que atualiza os valores

    public Object getValores(); //funcao que devolve os valores atuais

    public LockGlobal novoLock(TransactionID xid, Object o); //funcao que devolve um novo lock
}
```

Figura 2.2: Interface guardada no participante

Com estas interfaces conseguimos tornar o código independente da aplicação visto que a definição daquilo que estamos a alterar na transação e a forma como é alterado é responsabilidade de quem usa o *Two-Phase Commit*. Conseguimos mais alguma modularidade, para além de separarmos o participante do coordenador, visto que o *Two-Phase Commit* pode ser utilizado por qualquer aplicação. De referir apenas que o *Two-Phase Commit* requer que uma transação esteja associada a um pedido, algo que não conseguimos separar e que nos parece bastante usual.

Para além disto conseguimos obter também alguma independência nos *locks*, visto termos a seguinte classe abstrata:

```java
public abstract class LockGlobal{
    public CompletableFuture<Void> obtido = new CompletableFuture<>();
    public TransactionID xid;
    public int lockID;

    public LockGlobal(TransactionID xid, int lockid) {
        this.xid = xid;
        this.lockID = lockid;
    }

    abstract Object items();
}
```

Figura 2.3: Classe que define um lock

Esta classe representa o que um *lock* tem de ter no mínimo, no entanto deixa que seja da responsabilidade de quem usa o *Two-Phase Commit* que defina como são efetuados os métodos que dão os items a serem utilizados naquele *lock*, por exemplo.

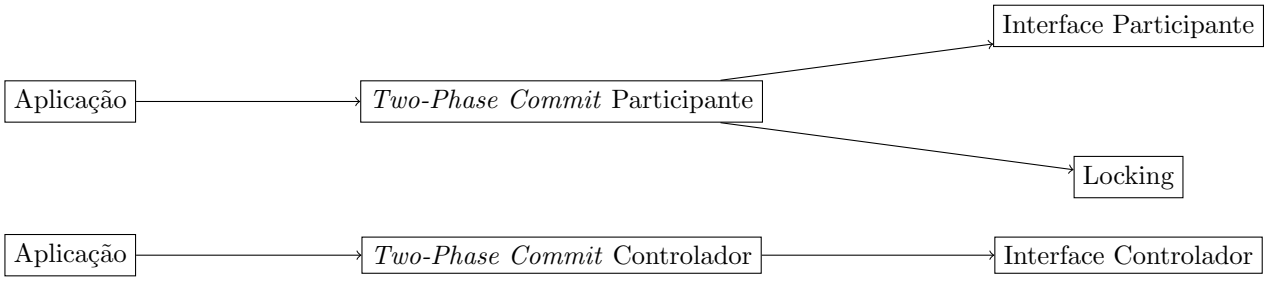
Definimos também a seguinte interface:

```java
public interface Locking {
    CompletableFuture<Void> lock(LockGlobal l);
    void unlock(LockGlobal l);
}
```

Figura 2.4: Interface que define o locking

Esta interface é responsável pela manutenção dos *locks*, bem como o seu correto funcionamento. O participante necessita desta classe para funcionar corretamente com os mecanismos de *locking*, sendo a mesma da responsabilidade do utilizador.

Conseguimos então a seguinte modularidade:



Capítulo 3

Discussão dos Resultados

Como era pretendido, conseguimos cumprir com sucesso a parte básica, tendo desenvolvido um sistema distribuído que permite as operações de GET e PUT, garantindo que as escritas são atomicamente realizadas. Isto foi conseguido utilizando o *Two-Phase Commit*, como referido anteriormente. Como utilizamos a estratégia de implementar um servidor que tanto funciona como participante como coordenador, para o *Two-Phase Commit* e os pedidos são distribuídos aleatoriamente a cada servidor evitamos alguns gargalos e congestionamento por coordenador (pois se apenas existisse um ele receberia todos os pedidos e todas as mensagens das transações). O *Two-Phase Commit* é independente do tipo de operações que queremos efetuar, utilizando objetos o mais genéricos possíveis, tornando-o independente da aplicação. Para além disto conseguimos implementar um mecanismo que garante a ordem das escritas nos participantes, ou seja a última escrita em cada participante é sempre referente ao mesmo pedido.

Porém o nosso sistema não é perfeito, tendo algumas limitações:

- A forma como o *lock* está implementado, para garantir ordem nas escritas, pode levar a que por vezes possa existir *starvation* para pedidos *PUT* com muitas chaves. Um pedido que peça para alterar os valores das seguintes chaves: 0,3,6 poderá ser ultrapassado por múltiplos pedidos que pedem as chaves 0, 3 e 6 separadamente;
- Apesar do *lock* estar global, não é completamente independente da aplicação por ser implícito;
- Os nossos servidores são tanto coordenadores como participantes, sendo que poderiam desempenhar esses papéis também de forma exclusiva;

A maior dificuldade encontrada pelo grupo foi a tentativa de separação entre os módulos que implementam *Two-Phase Commit* e a lógica da aplicação a ser suportada, porque sempre nos pareceram dependentes um do outro pelo facto de ser na ação de *commit* do *Two-Phase Commit* que o servidor deve atualizar os seus valores. Outra dificuldade foi decidir qual o mecanismo a utilizar relativamente à garantia de ordem das escritas.

Em suma, considera-se que conseguimos aplicar todo o conhecimento adquirido na componente teórica, onde apesar de a ideia da implementação ser simples, apresenta pequenos pormenores que tornam mais complicada a sua materialização em sistemas reais.