Serviço de Backup Distribuído para a Internet

Segundo Projeto de SDIS

Bruno Dias da Costa Carvalho	up201606517
Fernando Jorge Coelho Barreira Calheiros Alves	up201605270
Sandro Miguel Tavares Campos	up201605947
Sofia Cardoso Martins	up201606033

Faculdade de Engenharia
Universidade do Porto

2018-2019

1 Resumo

Esta aplicação, desenvolvida no âmbito da unidade curricular de *Sistemas Distribuídos*, teve como finalidade a implementação de um sistema do tipo *Peer-to-Peer* para a Internet, permitindo a utilização do espaço livre dos computadores da rede para o *backup* de ficheiros. Com este intuito, o projeto requereu o desenvolvimento de diversas funcionalidades que permitem a partilha de ficheiros através de uma rede composta por múltiplos computadores, que detêm controlo sobre o seu próprio armazenamento. Estas funcionalidades estão integradas nos sub-protocolos **Backup**, **Restore** e **Delete**, bem como nos métodos **Reclaim** e **State**, explicitados com maior detalhe na secção seguinte.

A arquitetura do sistema baseou-se nos seguintes conceitos:

1. JSSE (sem Java NIO)
2. Chord
3. Java NIO para aceder ao sistema de ficheiros
4. *Thread-pools*

As secções que se seguem abordam em maior detalhe cada uma destas características.

1.1 Definições

Seguindo a nomenclatura de [1], um *nó*, ou *node*, é um elemento da rede *Chord*, que corre num processo num certo endereço da Internet. O *address* de um nó é o endereço (incluindo porta) do *server socket* desse nó. Para um nó N, o seu identificador na rede é também N, dependendo do contexto. O parâmetro m é o número de bits nos identificadores dos nós e ficheiros, e um parâmetro essencial para o algoritmo.

2 Protocolos

O Sistema de *Backup* Distribuído desenvolvido é constituído por dois grupos de *sub-protocolos de comunicação*: os sub-protocolos *Chord* e os sub-protocolos *Backup*. O primeiro grupo é utilizado na manutenção da *Hash Table* distribuída, incluindo gestão de responsabilidades de chaves, suporte para *joins* concorrentes, deteção (mas não recuperação) de falhas na rede e

manutenção das *finger tables*. Estes sub-protocolos não lidam com transmissão de dados armazenados na rede e são de uso interno. O segundo grupo serve para a transmissão, recuperação e armazenamento na rede distribuída dos dados propriamente ditos – os ficheiros.

2.1 Sub-protocolos *Chord*

Os sub-protocolos de comunicação *Chord* são:

KeepAlive, **Predecessor**, **Notify**, e **Lookup**.

A utilizar estes protocolos de comunicação existem quatro *algoritmos de manutenção* da rede:

CheckPredecessor, **Stabilize**, **FixFingers**, e **Join**.

Todos estes algoritmos seguem as orientações do artigo original, com pequenas modificações necessárias para uma implementação **recursiva** de *lookups*.

2.1.1 Modo de operação

KeepAlive O nó iniciador N envia uma mensagem *KeepAliveMessage* a outro nó M qualquer, à qual o nó M deve incondicionalmente responder com *AliveMessage*. Nenhuma destas mensagens tem conteúdo propriamente dito.

Predecessor O nó iniciador N envia uma mensagem *GetPredecessorMessage* a outro nó M, à qual o nó deve responder com *PredecessorMessage*. A resposta contém o nó que M julga ser o seu predecessor, e não pode ser nulo – se M não tiver nenhum predecessor quando receber a mensagem, adota N como o seu predecessor.

Notify O nó iniciador N envia uma mensagem *NotifyMessage* a outro nó M *afirmando* que M é o seu predecessor. Este protocolo não instiga uma resposta, ao contrário dos outros. Em princípio, M deverá adotar N como seu predecessor caso não tenha nenhum, ou caso N seja um melhor predecessor do que o que tem atualmente.

Lookup Este protocolo é recursivo e corresponde a pedidos *Successor* no artigo original. O nó iniciador S (*source*) pretende encontrar (i.e. abrir um canal de comunicação com) o nó responsável por uma chave k. Para isso, envia uma mensagem *LookupMessage* a outro nó M_0, que é o predecessor mais próximo de k que encontra na sua *finger table*. Ao receber a mensagem,

M_0 vai procurar na sua *finger table* o predecessor mais próximo de k que conhece, e reencaminhar a mensagem para esse predecessor M_1, etc. Este processo continua até que um node M_i não encontre na sua *finger table* nenhum predecessor de k. Nesse caso, se M_i tiver um sucessor R, então reencaminha a mensagem para R. O nó R, ao receber o pedido, reconhece que é o responsável pela chave k (dado que esta se encontra entre M_i, o seu predecessor, e si próprio), e termina o pedido enviando ao nó S uma mensagem *ResponsibleMessage* afirmando que é responsável pela chave k.
Nem sempre este pedido é bem sucedido. Se uma ou várias secções da rede estiverem instáveis devido à entrada de um ou mais novos nós, um nó M_j pode não encontrar um nó M_{j+1} a quem reencaminhar a mensagem, ou então pode verificar que aquela mensagem **já passou por** M_j e pode entrar num *loop* na rede. Nestes casos a mensagem é **descartada**, e o pedido nunca é respondido.[1]

2.1.2 Algoritmos de manutenção

CheckPredecessor Este algoritmo é periodicamente executado em cada nó N da rede, e inicia uma instância do protocolo *KeepAlive* com o seu predecessor P. Se a ligação com P falhar ou P não devolver uma resposta *AliveMessage* a N, o nó P é purgado no nó N. Se o algoritmo correr quando N não tiver predecessor, não faz nada.

Stabilize Este algoritmo é periodicamente executado em cada nó N da rede. O seu objetivo é manter tanto o sucessor de N como o *predecessor do sucessor de* N atualizados e corretos. N inicia uma instância do protocolo *GetPredecessor* com o seu sucessor S, e recebe como resposta um nó P que S julga ser o seu predecessor. Se P for um melhor sucessor para N do que S, então N adota P como seu sucessor. Após isso, o nó N notifica o seu sucessor (S ou P) através do protocolo *Notify*.

FixFingers Este algoritmo é periodicamente executado em cada nó N, atuando sobre cada *finger* $f_1, f_2, \ldots, f_m$ de cada vez.[2] O seu objetivo é manter a *finger table* atualizada. Para isso, atuando sobre f_i, o nó realiza

[1] Seria possível devolvermos a S uma resposta de insucesso, do género *Try again later*, mas isto não leva em conta a possibilidade de comunicação entre nós não ser bem sucedida. A nossa solução é mais simples: S desiste de esperar pela resposta passado um *timeout* configurável t

[2] A primeira execução executa sobre f_1 (o sucessor), a segunda sobre f_2, ..., a m-ésima sobre f_m, e depois do início novamente: a $m+1$-ésima sobre f_1, ...

um *Lookup*$(N + 2^{i-1})$. A resposta a este pedido será o sucessor S que será o novo valor de f_i, incondicionalmente.

Join Este algoritmo é executado uma vez, quando um nó N entra na rede com conhecimento de um nó I já dentro da rede. O funcionamento é simples: N faz um pedido *Lookup*(N). Se receber uma resposta, S, então S é o seu sucessor imediato. N aceita S como seu sucessor e inicia os seus outros algoritmos, correndo *Stabilize* imediatamente para informar S da sua entrada. Se não receber resposta, tenta outra vez após uma pequena pausa, esperando que a rede estabilize.

2.2 Sub-protocolos DBS

Os sub-protocolos DBS são: **Backup**, **Restore** e **Delete**.

As mensagens dos protocolos descritos nesta secção encontram-se na *package* ***dbs.chord.messages.protocol***, e os respetivos *Observers* encontram-se na *package* ***dbs.chord.observers.protocols***.

2.2.1 Backup

Este protocolo encarrega-se de dar *backup* ao ficheiro especificado pelo utilizador, com o grau de replicação pedido. Para executar um pedido de *backup*, o comando a executar na *interface* da *TestApp* é o seguinte:

java -cp bin dbs.TestApp <remote_obj> BACKUP <file_path> <rep_deg>

Para implementarmos o grau de replicação dos ficheiros que foram sujeitos a *backup*, calculamos a *hash* do nome do ficheiro, e depois calculamos R *offsets* para esse valor, em que R é o grau de replicação pedido. Decidimos utilizar esta abordagem para que as réplicas do ficheiro estivessem distribuídas de forma mais ou menos uniforme pela rede, mesmo que por vezes aconteça que alguns nós guardem o mesmo ficheiro múltiplas vezes. Esses *offsets* são calculados pela seguinte fórmula:

$$baseId + \frac{i \cdot 2^m}{R}$$

O *baseId* corresponde à *hash* original do nome do ficheiro e i varia entre 0 e (R-1).

O nó iniciador do protocolo *backup* envia uma mensagem do tipo *lookup* com o valor de cada um dos *offsets* calculados, para determinar os nós que são responsáveis pelas réplicas do ficheiro. Após saber quais são esses nós,

irá enviar para cada um deles uma mensagem do tipo **BackupMessage**, que contém o *id* do ficheiro, o seu conteúdo, e uma variável booleana chamada *overwrite* que determina o comportamento do nó responsável caso já tenha dado *backup* a um ficheiro com o mesmo *id*. Cada um dos nós responsáveis, após ter guardado o ficheiro enviado pela mensagem *BackupMessage*, irá enviar uma mensagem de retorno ao nó iniciador do protocolo do tipo **BackupResponseMessage**, que contém o *id* do ficheiro guardado e um código do tipo *ResultCode*, que indica o resultado da operação.

2.2.2 Restore

Este protocolo encarrega-se de dar *restore* a um ficheiro especificado pelo utilizador, que tenha sido sujeito a *backup* pelo nó iniciador. Para executar um pedido de *restore*, o comando a executar na *interface* da *TestApp* é o seguinte:

java -cp bin dbs.TestApp <remote_obj> RESTORE <file_path>

Para obtermos o ficheiro a restaurar, primeiro precisamos de saber com que grau de replicação ele foi guardado inicialmente, pois esse valor é necessário para calcular os *offsets* que representam as diferentes cópias de um ficheiro.

Em cada pedido de *backup*, esse valor é guardado num *HashMap* chamado *replicationMap*, cuja chave é o *id* original do ficheiro e o valor é o grau de replicação do pedido de *backup* previamente efetuado. De forma semelhante ao *backup*, prosseguimos a enviar mensagens do tipo *lookup* com os valores dos *offsets* calculados, para determinar quais são os nós que potencialmente terão o ficheiro a restaurar. Após sabermos quais são esses nós, prosseguimos a enviar uma mensagem do tipo **RestoreMessage** a cada um deles, até algum devolver o ficheiro. Esta mensagem contém apenas o *fileId* do ficheiro a restaurar. Se um nó que receba esta mensagem tiver o ficheiro especificado, irá enviar ao nó iniciador uma mensagem do tipo **RestoreResponseMessage**, que contém o *fileId* do ficheiro que vai enviar, o seu conteúdo, e um código do tipo *ResultCode*, com o valor *ResultCode.OK*.

2.2.3 Delete

Este protocolo encarrega-se de apagar um ficheiro especificado pelo utilizador, que tenha sido sujeito a *"backup"*. Para executar um pedido de *delete*, o comando a executar na *interface* da *TestApp* é o seguinte:

java -cp bin dbs.TestApp <remote_obj> DELETE <file_path>

Para apagarmos o ficheiro de todos os nós que o guardaram, primeiro precisamos de saber com que grau de replicação ele foi guardado inicialmente, pois esse valor é necessário para calcular os *offsets* que representam as diferentes cópias de um ficheiro. Tal como já foi mencionado, esse valor é guardado num *HashMap* chamado *replicationMap*. De forma semelhante ao *backup*, prosseguimos a enviar mensagens do tipo *lookup* com os valores dos *offsets* calculados, para determinar quais são os nós que potencialmente terão o ficheiro a apagar. Após sabermos quais são esses nós, prosseguimos a enviar uma mensagem do tipo **DeleteMessage** a cada um deles. Esta mensagem contém apenas o *fileId* do ficheiro a apagar. Se um nó que receba esta mensagem tiver o ficheiro especificado, irá apagá-lo, e proceder a enviar ao nó iniciador uma mensagem do tipo **DeleteResponseMessage**, que contém o *fileId* do ficheiro apagado, e um código do tipo *ResultCode*, com o valor *ResultCode.OK*.

2.3 User Interface

Esta secção irá abordar os métodos **Reclaim** e **State** que o utilizador pode chamar usando a *TestApp*, mas como não envolvem comunicação com outros nós na rede, não incluímos na secção anterior.

2.3.1 Reclaim

O método *Reclaim* serve para o utilizador poder decidir qual é o tamanho máximo que pode ser ocupado por ficheiros de *backup* num dado nó. Este método pode ser chamado usando o seguinte comando na *interface* da *TestApp*:

java -cp bin dbs.TestApp <remote_obj> DELETE <max_space>

Quando um nó recebe este comando (via RMI), ele irá determinar o tamanho atualmente ocupado no seu diretório de ficheiros de *backup*. Se este tamanho for superior ao *max_space* especificado, então irá começar a apagar ficheiros, até o espaço ocupado ser inferior ou igual a esse valor. O nó irá dar prioridade a ficheiros de maior dimensão ao escolher quais apagar.

2.3.2 State

O método *State* serve para o utilizador puder consultar o estado de um nó. Este método pode ser chamado usando o seguinte comando na *interface* da *TestApp*:

java -cp bin dbs.TestApp <remote_obj> STATE

Quando recebe este comando (via RMI), o nó irá mostrar (no terminal da *TestApp*) o seu *ChordId*, o seu endereço e respetiva porta, e finalmente, a lista dos ficheiros a que deu *backup* noutros nós, com o respetivo *replication degree*.

3 Concorrência

3.1 Concorrência na comunicação

De acordo com o suprarreferido, o projeto envolveu a implementação de um sistema de *Hash Table* distribuída para localização de ficheiros na rede. Este permite que novos nós se possam juntar à rede, sendo portanto necessário que um nó já pertencente a esta lide com este tipo de pedido. Para tal recorreu-se a uma *thread-pool* do tipo **ScheduledThreadPoolExecutor**, inicializada na classe **ChordDispatcher**, de forma a que um nó possa processar múltiplos pedidos simultaneamente. Uma segunda *thread-pool*, pequena, existe internamente na classe **Node** para correr os algoritmos **CheckPredecessor**, **Stabilize**, **FixFingers**, e **Join**, com uma periodicidade fixa definida pelas constantes *STABILIZE_PERIOD*, *FIXFINGERS_PERIOD* e *CHECK_PREDECESSOR_PERIOD*, respetivamente. Com o intuito de verificar o estado de cada nó (a sua *finger table*), esta *thread-pool* realiza também, de forma periódica, a execução da tarefa **Dump**.

3.2 Concorrência interna

O serviço é capaz de receber pedidos concorrentemente, tendo por isso havido a necessidade de recorrer novamente a uma arquitetura *multithreading* para gestão dos mesmos, mas recorrendo-se apenas a uma *thread* para acesso ao sistema de ficheiros de forma a tirar partido de **Java NIO**. Representada por uma instância de **FileManager**, essa mesma *thread* detém a responsabilidade de efetuar operações de escrita e leitura em disco, fornecendo para tal uma API dedicada à execução das operações necessárias aos diversos sub-protocolos, constituída pelos métodos *launchBackupReader*, *launchBackupWriter*, *launchRestoreReader*, *launchRestoreWriter* e *launchEraser*. Estes métodos recorrem a um dos seguintes tipos de executável, **Writer**, **Reader** ou **Eraser**, lançados em execução a partir de uma *thread-pool* de tamanho fixo pré-configurada. Estes constroem os pedidos respetivos e colocam-nos numa fila que o **FileManager** tem disponível e que é processada pelo mesmo por ordem sequencial.

Para manipular os ficheiros de forma assíncrona faz-se uso de canais do

tipo **AsynchronousFileChannel**, que permitem efetuar leituras ou escritas de ou para um ficheiro, associadas a *buffers* do tipo **ByteBuffer**. Essas operações são realizadas de forma particionada, isto é, considerando a existência de *chunks* de tamanho máximo de 64KB, que são posteriormente agregados para constituir o ficheiro original. Desta forma, o **FileManager** recebe para um mesmo ficheiro um conjunto de pedidos, identificados pelo *fileId* e pelo número do *chunk* respetivo, podendo ser processados alternadamente com outros, não havendo possibilidade de bloqueio. Assim sendo, e visto que cada nó da rede tem a possibilidade de executar simultaneamente múltiplos sub-protocolos, o serviço assegura um elevado grau de concorrência.

4 JSSE

De modo a assegurar a comunicação segura entre os nós da rede, foi utilizada a *framework Java Secure Socket Extension (JSSE)*, nomeadamente o protocolo SSL.

Neste contexto, é de destacar a utilização dos canais seguros que a *framework* JSSE oferece - **SSLServerSocket** e **SSLSocket** - em operações de todos os sub-protocolos: *Backup*, *Restore*, *Delete* e *Reclaim*. Embora tivesse sido dada liberdade para optar pelo uso de canais seguros apenas num subconjunto das operações suportadas pela rede, para assegurar a integridade da mesma e até a coerência entre os sub-protocolos, foi estipulado que todas as operações se fariam por intermédio de canais deste tipo.

As funcionalidades da *JSSE* que foram integradas no projeto compreendem o protocolo de *Handshake* e a autenticação do cliente. O primeiro é utilizado para a negociação dos parâmetros a serem usados no canal seguro, aquando do estabelecimento de uma ligação entre os *SSLSockets* e antes da troca de dados que deverão ser protegidos. Para esse efeito, é usado o método **setEnabledCipherSuites** da classe SSLSocket passando-lhe como parâmetros as *cipher suites* adquiridas através do método **getSupportedCipherSuites**. Por outro lado, para garantir a autenticação do cliente, é invocado o método **setNeedClientAuth(true)** a partir de um objeto da classe **SSLServerSocket**. Destacam-se, neste contexto, os métodos **open** e **run** (da classe **Accepter**), incluídos no ficheiro **SocketManager.java**.

Além disto, é de destacar que as chaves partilhadas são geradas automaticamente pelo protocolo de *Handshake* a partir de números aleatórios gerados tanto pelo cliente como pelo servidor, e que as chaves requeridas pelo processo de autenticação são obtidas em separado por cada um dos intervenientes na comunicação. Os ficheiros relevantes para este processo podem ser encontrados na pasta *cert*. Outros métodos considerados relevantes incluem:

setupClientContext, **setupServerContext** e **join**, localizados no ficheiro **Dbs.java**. Por último, e para efeitos de esclarecimento, é importante referir que não foi necessário utilizar a classe **SSLEngine**.

5 Detalhes de implementação de *Chord*

Assumimos que a rede onde o *Chord* corre é colaborativa e não hostil.

A nossa implementação dos protocolos de comunicação utiliza um modelo *Notify-Subscribe*[3] assente em trios do tipo **(Key,Message,Observer)**, que abreviaremos (K, M_K, O_K). Para cada *tipo* de mensagem $M = M_K$ trocada entre dois nós, existe uma chave K associada a M (não confundir com chaves k da rede *Chord*) que identifica M do lado do recetor. Um nó N que queira receber mensagens do tipo M_K (com chave K) regista um *observer* do tipo O_K (com chave K). Quando uma mensagem M_K for recebida o observador O_K é notificado.

As chaves são instâncias de **ChordMessageKey** e normalmente são compostas apenas por um texto identificativo (por exemplo `'LOOKUP'`). Algumas respostas contêm também uma chave de *Chord* k.

As mensagens são instâncias de subclasses de **ChordMessage** (localizadas em `dbs.chord.messages`). As mensagens são enviadas *as-is* através da rede utilizando a serialização nativa de Java e *ObjectStreams*.

Os observadores são instâncias de subclasses de **ChordObserver**, (localizados em `dbs.chord.observers`). Podem ser *permanentes* ou *temporários*.

Os primeiros são instâncias da classe **PermanentObserver**, os outros são instâncias de **TimeoutObserver**. Os observadores permanentes têm uma subscrição permanente, desde o momento em que o nó é criado até ao momento em que morre, e são notificados de todas as mensagens do tipo que subscreveram. Já os observadores temporários subscrevem por um período de tempo finito, e só são notificados de uma mensagem. Se a mensagem subscrita não chegar durante esse período, o observador é *cancelado*.

As funcões de notificação dos observadores correm em *threads* de uma *thread pool* interna da classe **ChordDispatcher**, portanto há total paralelismo entre observadores. No entanto, é raro (mas não impossível) existir mais do que um observador para uma chave K.

Para se adicionar um novo sub-protocolo a este sistema de comunicação, tudo o que é preciso fazer é criar um ou mais trios (K, M_K, O_K) de chaves, mensagens e observadores respetivamente.

[3] semelhante ao *Observer pattern*.

5.1 Escalabilidade

Todos os parâmetros da rede são configuráveis (ver/editar classe **Chord** em `dbs.chord`), nomeadamente, para toda a rede, a função de *hashing* e o parâmetro m. Para cada nó também é possível editar o período dos algoritmos de manutenção, máximos tempos de espera, e tamanho das *thread-pools*.

Um **chord id** é representado por um *BigInteger*, permitindo que m tome valores superiores a 64.

A manutenção da rede foi testada duramente com vários valores do parâmetro m até 32, com uma escala de 31 nós e de 10 *joins* concorrentes com os mesmos sucessores. Dada a instabilidade da rede após um *batch* de *joins*, é normal alguns nós precisarem de um segundo ou terceiro *lookup* para se juntarem à rede (ver *Lookup* em 2.1), mas a rede estabiliza facilmente após no máximo duas rondas do algoritmo *FixFingers* sem nenhum *join*.

Referências

[1] David Karger M. Frans Kaashoek Hari Balakrishnan Ion Stoica, Robert Morris. Chord: A scalable peer-to-peer lookup protocol for internet applications.