Enhanced Distributed Backup Service

28/05/2018

Estudantes:
Ana Margarida Silva - up201505505@fe.up.pt
Bruno Piedade - up201505668@fe.up.pt
Danny Soares - up201505509@fe.up.pt
Pedro Daniel Reis - up201506046@fe.up.pt

Objetivo da aplicação

O objetivo desta aplicação consiste no melhoramento da aplicação de *backup* distribuído desenvolvida para o primeiro projeto. O aperfeiçoamento que se irá realizar focar-se-á na segurança, autenticação, consistência e usabilidade da aplicação.

Funcionalidades Principais

- *Enhancements* sugeridos na primeira versão do projeto
- Segurança
- Autenticação dos *peers*
- Tolerância a falhas
- Controlo de versões

Controlo de versões

Para este trabalho iremos implementar um sistema simples de controlo de versões. Para este efeito, sempre que um *peer* pretender efetuar um backup, o nome do ficheiro e a data de backup serão armazenados localmente, para que o utilizador consiga mais tarde aceder a essa informação e saber se quer fazer um restore de algum ficheiro, ou eventualmente apagá-lo do sistema. Se a segunda opção for seleccionada, o *peer* tem duas opções: ou elimina todas as cópias de uma determinada versão do ficheiro, ou pode eliminar todas as versões de uma só vez. Independentemente da opção, será eliminado de todos os peers que tenham sido.

Descrição do design e implementação de concorrência

Para gerir a concorrência no nosso projeto, recorremos a uma implementação baseada em *threads* e *mutexes*, por forma a assegurar que é possível executar múltiplos pedidos sem que haja nenhum problema de concorrência. Para permitir a execução concorrente de múltiplos pedidos, foi usada uma pool, da classe *ThreadPoolExecutor*, que gere a execução das diferentes *threads*. Para cada pedido recebido é instanciado um objeto *Runnable* que processa a tarefa e, no caso dos protocolos BACKUP e RESTORE, é

guardada uma referência para o mesmo num *Map*(classe *ConcurrentHashMap*), para permitir posteriores notificações aquando da receção de mensagens relevantes para o pedido que está a ser processado. Para garantir que os diferentes pedidos a ocorrer ao mesmo tempo não acedem ao mesmo objeto ao mesmo tempo, utilizámos *mutexes* da classe *ReentrantLock*, que fazem a gestão do acesso das múltiplas *threads* ao mesmo objeto. No código seguinte, encontra-se um exemplo da utilização desses *mutexes* no nosso trabalho, em que a variável *lock* é uma variável do tipo *ReentrantLock* e verifica que não há outra *thread* a aceder à variável *chunkPeers* no momento em que a thread a executar este código acede.

```
try{this.lock.lock();
        this.chunkPeers.add(new ArrayList<Integer>());
}
finally{
        this.lock.unlock();
}
```

O envio de mensagens é feito através de três *sockets multicast*, partilhados por todos os servidores, cujos acessos competitivos são geridos pela classe *TwinMulticastSocket*, recorrendo também a *locks*, para evitar que várias *threads* escrevam no mesmo *socket* ao mesmo tempo . Estes *sockets* estão associados a três *threads*, presentes nos servidores, que escutam constantemente os canais *multicast* e criam uma nova *thread* para processar qualquer mensagem recebida. Para o MC(multicast control channel) é criada uma *thread* da classe *ControlProtocol* que processa as mensagens do tipo STORED, GETCHUNK, DELETED e REMOVED; para o MDB(*multicast data backup channe*) uma *thread* da classe *StoreChunk* que processa as mensagens do tipo STORECHUNK; para o MDR(*multicast data restore channel*) uma *thread* da classe Chunk que processa as mensagens do tipo GETCHUNK. Estas *threads* realizam o *parsing* do conteúdo da mensagem, processam as mesmas e, caso seja necessário, enviam a resposta correspondente. Para gerir os acessos concorrentes à memória que representa os ficheiros dos servidores, desenvolvimentos a classe FileManager , que guarda os ficheiros e os *chunks* que o seu servidor e funciona com o mecanismo *synchronized*, que garante que não há duas *threads* a aceder ao mesmo ficheiro ao mesmo tempo. Concluindo, o nosso *design* de concorrência, baseado em *threads* e *mutexes*, permite que o sistema funcione corretamente com vários pedidos a ocorrer ao mesmo tempo.

Funcionalidades

Existirão utilizadores com privilégios admin (administradores do sistema), que poderão fazer RECLAIM de ficheiros, sendo isto a gestão dos ficheiros. Esta operação apenas poderá ser utilizada se a chave que o utilizador providenciar no comando no terminal coincidir com a chave do administrador daquele *peer*. Utilizadores sem privilégios de administrador poderão fazer BACKUP, DELETE, RESTORE e LIST dos seus ficheiros. No caso do DELETE, um utilizador apenas pode apagar ficheiros que o próprio fez backup.

Segurança

De modo a proteger a rede de ataques *Man in the Middle,* todas as mensagens que são transmitidas são encriptadas com uma chave que é partilhada por todos os peers. Depois desta ser desencriptada, o conteúdo de cada *chunk* (no caso de um backup, por exemplo) está encriptada com a chave privada de cada utilizador. Isto permite-nos manter uma rede segura, sem risco de outros utilizadores tentarem aceder a ficheiros de outros sem a sua informação privada.

Tolerância a falhas

Para evitar a possibilidade de informação ser perdida, todos os *chunks* podem estar repartidos por múltiplos *peers*, com um número que o utilizador pretender (*replication degree*). Assim, em caso de algum *peer* ir abaixo por motivos anormais, a informação permanece na rede.

De modo a que os *peers* tenham uma noção mais exata da quantidade de dados que há na rede, todos os peers mantém os metadados atuais em memória volátil que vão sendo atualizados de acordo com as operações que decorrem na rede.

(existe um ficheiro de metadados que todos os *peers* têm que armazena informação dos pedidos que já foram efetuados).

Para garantir a consistência dos metadados quando um novo peer é adicionado à rede inicializa um protocolo que envia um pedido *multicast* para a rede para recolher os metadados atuais que estão presente nos outros *peers* (*UpProtocol*) de forma a sincronizar-se com o estado atual do sistema. Esta recolha é feita de forma semelhante ao protocolo RESTORE na medida em que a informação é transferida em blocos não superiores a 64000 *bytes*. Após recebida, o *peer* avalia a metadata atual em relação ao que contém no seu disco, ou seja o *chunks* que tinha guardado, e caso um *chunk* ainda exista nos metadados envia para a rede a mensagem STORED de forma a notificar os restantes. Caso este protocolo falhe sem recuperar os metadados do sistema com sucesso (condição que se verifica através de um timeout na resposta da rede), o *peer* assume que não existe mais nenhum outro *peer* na rede e portanto apaga todo o conteúdo do seu disco. De forma semelhante é executado um protocolo na terminação normal de um *peer* que trata de enviar as mensagens REMOVED para todos os *chunks* que contém de forma a manter a sincronização dos metadados nos diferentes *peers* da rede.

Esta abordagem assume que a falha de um peer é aceitável apesar de se perder a consistência completa dos dados, porém uma vez que o peer falhou será expectável que irá ser reinicializado posteriormente e neste caso a consistência dos dados é recuperada. Todavia, caso isto não ocorra, a redundância de dados permite manter operacionais as funcionalidades do sistema.

(Para forçar a sincronização deste ficheiro, foi implementado um protocolo que é chamado sempre que um *peer* novo é criado. Este pede por *multicast* para que outro peer que tenha o ficheiro de metadados lhe envie. Deste modo, a informação é consistente por toda a rede.)

A maior vantagem desta abordagem reside na aproximação à realidade. Um exemplo simples que demonstra a necessidade deste método é a que se segue. Existem num determinado momento três *peers* a correr em simultâneo, o *peer1, peer2,* e o *peer3*. Um pedido de backup de um ficheiro do *peer1* é recebido, com um grau de replicação de três. No final desta ordem, o *peer2* e o *peer3* têm uma cópia de todos os chunks do ficheiro, e o grau não foi atingido. Agora estes dois ficheiros são terminados normalmente, e um novo *peer* é ativado, o *peer4*. Se o *peer1* tentasse efetuar um backup do mesmo ficheiro de antes, este pedido era executado, e tem um grau de replicação percepcionado de 3. No entanto, o *peer2* e o *peer3* pensam que tem um grau de replicação de 2, e o novo *peer4* pensa que é 1! Após uma operação tão simples, literalmente toda a percepção dos *peers* estava incorreta. Aplicando agora o novo método, quando o *peer4* é ativado, ele requisita uma cópia dos metadados do sistema. O *peer1* envia-lho, com a informação que o *replication degree* do ficheiro que foi feito *backup* é de 2. Agora este novo *peer* já sabe que na próxima ordem, o grau passaria a ser 3, e não 1.

Target Platforms

- Java standalone application for PC