Serviço de Backup Distribuído - Relatório

Mestrado Integrado em Engenharia Informática e Computação

Sistemas Distribuídos

Turma 03 Grupo 1:
Ricardo Silva - up201607780
Luís Oliveira - up201607946

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

14 de Abril de 2019

Este relatório descreve os métodos utilizados para tratar da concorrência do Serviço de Backup Distribuído bem como os melhoramentos adicionados ao protocolo base.

1 Concorrência

A nossa aplicação encontra-se dividida em módulos mais pequenos que correm em threads separadas, que se sabe sempre o que estão a fazer e que, inclusive, podem terminar, se necessário. Processa diferentes mensagens recebidas simultaneamente no mesmo canal, usando *ThreadPools*.

A classe **Peer** usa um *ScheduledThreadPoolExecutor* e um *MessageDispatcher*. O primeiro agenda o arranque posterior de tarefas e inicializa os originadores de cada serviço em novas threads. O segundo é usado para distribuir as mensagens e atribuir-lhes uma thread de acordo com o seu serviço. Sempre que recebe uma mensagem, coloca-a numa *BlockingQueue*, e depois vai processando mensagem a mensagem, retirando-as por ordem dessa fila, atribuindo-lhe uma nova thread do serviço apropriado.Quando não tem qualquer mensagem a processar, liberta os seus recursos para serem usados pelo CPU.

Como este uso de várias threads que lidam com os mesmos dados pode levar a race conditions. Usamos coleções da classe *java.util.concurrenct.** sempre que uma variável possa ser acedida ou modificada pelas diferentes threads que correm simultaneamente.

A classe **Backup** requer muitos acessos ao disco e também um intervalo grande de tempo entre enviar o PUTCHUNK e receber o STORED, e por isso tem uma *ThreadPool* própria. O acesso à **Database**, ondes se guardam os registos necessários, é protegido por métodos *Synchronized*.

Para as tarefas que se querem iniciadas após um certo intervalo de tempo, usamos novamente o *ScheduledThreadPoolExecutor*, permitindo assim também libertar a thread principal e ter a possibilidade de terminar a sua execução se necessidade houver. Este método devolve um objeto **Future**, através do qual temos acesso a cada thread.

2 Melhoramentos

2.1 BACKUP (versão 1.1)

Esta versão evita que o espaço de memória se encha rapidamente em peers que não sejam necessários para guardar chunks com baixo grau de replicação.

Ao invés de guardar logo os chunks ao receberem o PUTCHUNK como na versão normal, com este melhoramento os peers esperam um intervalo de tempo aleatório depois de receberem o PUTCHUNK antes de o guardarem. Se nesse intervalo de tempo receberem mensagens de STORED em número que indique o grau de replicação já foi atingido então não chega a guardar nada.

A interoperabilidade está garantida porque só se o peer replicador tiver esta versão é que a implementa, sendo independente dos outros.

2.2 RESTORE (versão 1.2)

No RESTORE só há um destinatário nos envios dos chunks. Apesar disso, a versão normal utiliza o canal MULTICAST para o RESTORE, correndo o risco de o sobreutilização do canal se os chunks forem grandes.

Como só há um recetor, podemos utilizar ligações TCP para transferir os chunks entre um peer replicador e o recetor. Trata-se de um protocolo orientado à ligação, por isso garante-se que não há perda de dados e no final a união de chunks num ficheiro não corre perigo de ter chunks em falta.

A forma que encontramos de implementar este melhoramento foi de ter o peer iniciador a fazer um broadcast do seu access point através do canal multicast, adicionando um novo campo ao cabeçalho da mensagem GETCHUNK e depois o peer replicador que envia cada chunk estabelece uma ligação TCP diretamente ao peer recetor (iniciador).

`GETCHUNKENH <versao> <n°peer> <file_ID> <chunk_No> <acess_point> <CRLF> <CRLF>`

A interoperabilidade está garantida porque se um peer normal receber um ENH envia por multicast como se tivesse recebido um pedido normal

2.3 DELETE (versão 1.3)

Para evitar que um peer não apague chunks porque a mensagem DELETE foi enviada quando ele não estava ativo, guardamos a identificação dos peers replicadores de um ficheiro numa lista no peer iniciador após este receber cada mensagem STORED. Os replicadores guardam também o nome do ficheiro original numa lista quando recebem o DELETE.

Aquando do BACKUP, o peer iniciador, ao receber os STORED guarda numa lista a identificação dos peers replicadores. Quando há um DELETE também é guardado numa lista o identificador do ficheiro e os peers enviam de volta mensagens HASDELETED quando apagam os chunks. O peer iniciador, ao receber a mensagem HASDELETED, remove o identificador da tabela referente ao ficheiro em causa.

Se um replicador voltar a receber um PUTCHUNK para um ficheiro que já tinha apagado, remove o nome do ficheiro da lista de apagados.

Quando um peer acorda/volta ao ativo envia uma mensagem ACTIVE. Quando um peer iniciador recebe uma mensagem ACTIVE de outro peer que ele identifica como replicador de um ficheiro seu entretanto apagado, porque ainda consta da lista, reenvia a mensagem DELETE.

`ACTIVE <versao> <peer_iniciador> <CRLF> <CRLF>`

`HASDELETED <versao> <peer_iniciador> <file_ID> <CRLF> <CRLF>`

A interoperabilidade está garantida porque só se o peer replicador tiver esta versão é que a implementa, sendo independente dos outros.

[Fim]