Relatório

Project 1 - Distributed Backup Service

Grupo T1G01:
João Gomes - up201403275
Pedro Costa - up201403291

Protocol Enhancements

1. BACKUP

Esta melhoria pede para mantermos o *replication degree* de um *chunk* na rede igual ao *desired*. Ou seja, guardar apenas o número de *chunks* necessários para garantir o grau de replicação mínimo.

Por forma a implementar esta melhoria, o peer antes de guardar o *chunk* pedido pela mensagem PUTCHUNK, caso não tenha o chunk, espera um tempo random até 1,5 segundos. Durante este intervalo recolhe as mensagens STORED recebidas para este *chunk*. Ao fim deste intervalo, se o grau de replicação não estiver satisfeito então guarda o *chunk* e envia a mensagem STORED. Caso contrário ignora o pedido PUTCHUNK *(função "storeChunkEnhanced" linha 99 do ficheiro src/Subprotocols/Backup.java)*.

Para além desta alteração, alteramos o envio da mensagem STORED, para enviar logo mal é pedido em vez de esperar um tempo random de 400ms como pedido no enunciado. Isto porque já não é necessário esperar este tempo *random*, pois já o foi feito antes e garantimos que mal é guardado o chunk, também é enviada a mensagem de STORED *(função "deliverStoredMessageEnhanced" linha 177 do ficheiro src/Subprotocols/Backup.java)*.

Conseguimos implementar com uma boa eficácia esta melhoria, no entanto, não é interoperável com peers que não a implementam, pois toda a verificação de se deverá ou não guardar o chunk está do lado de quem recebe o chunk e não do peer que envia.

.

2. RESTORE

Nesta melhoria é nos pedido para não usar o *multicast* como forma de *restore* de um ficheiro, isto porque apenas um *peer* está a pedir os *chunks*, logo não precisam todos de receber.

Por forma a solucionar isto, realizamos uma ligação TCP entre o *peer* que faz GETCHUNK e o *peer* que envia a mensagem CHUNK. De modo a funcionar com peers que estejam a correr a versão 1.0 do programa, o peer envia para o *multicast control channel* o pedido GETCHUNK e fica a ouvir respostas do MDR - *multicast data recovery channel* - (para peers com a versão 1.0) e lança um *runnable* que cria uma *ServerSocket* na porta do MDR, que enquanto não tiver os chunks todos fica num loop a aceitar ligações a esta porta. E lança um *runnable* para cada ligação nova que aceita. Nesta *thread* recebe os *bytes* que

contêm a mensagem CHUNK. Do lado do *peer* que envia, quando é chamada a função que envia a mensagem CHUNK *(função "deliverChunkMessage", linha 203 do ficheiro src/Subprotocols/Restore.java)*, caso este esteja a usar a versão 1.1 e quem fez o pedido também, e caso não tenha feito o pedido de ligação TCP, ele faz o pedido e conecta-se. De seguida envia a mensagem pelo canal TCP. Caso esteja a usar a versão 1.0 enviar para o *multicast*.

Desta forma, esta melhoria é interoperável com *peers* que não o implementam.

3. SPACE RECLAIM

Neste ponto é nos pedido para, salvaguardar o caso em que um *peer* falha a meio do protocolo de *backup*, que é iniciado após a ocorrência de um pedido de *space reclaim* que leve a que o grau de replicação de algum(ns) *chunks* desça abaixo do desejado. Lançam nos o desafio de verificar isto quer quando é iniciado o *backup* pela primeira vez ou não.

Por forma a implementar isto, criamos uma classe *Tasks*. Estas classe representa tarefas que estão a ser realizadas. Assim quando um *peer* começa a fazer *backup* de um *chunk*, ele cria esta tarefa *(linha 124 do ficheiro src/Subprotocols/Backup.java)* e guarda como não completa, e após receber o STORED do mesmo, marca a tarefa como completa *(linha 282 do ficheiro src/Peer/Peer.java)*. Esta informação é guardada em memória no disco e lida sempre que um *peer* é iniciado. Desta forma se um *peer crashar* a meio do protocolo, quando este se volta a ligar, verifica que tem uma tarefa pendente e realiza-a novamente *(linha 115 do ficheiro src/Peer/Peer.java)*.

A partir da mesma linha de pensamento, implementamos isto para o *backup* original de um ficheiro. Sendo que desta vez a tarefa não será fazer o *backup* de um *chunk* mas sim de um ficheiro *(linha 206 do ficheiro src/Subprotocols/Backup.java)*. Assim se um peer crashar a meio do primeiro *backup*, quando se volta a ligar envia uma mensagem de DELETE do ficheiro, e inicia o *backup* de novo. Só quando recebe as mensagens de STORED de todos os *chunks* do ficheiro, é que marca a tarefa como completa *(linha 249 do ficheiro src/Subprotocols/Backup.java)*.

Esta melhoria é interoperável com *peers* que não a implementam, pois todo o tratamento de tarefas é feito do lado do *peer* que faz o *backup*.

4. DELETE

Na especificação do projeto é sugerido que nós encontremos uma solução para o caso de um *peer* que contém *chunks* de um ficheiro, não esteja a correr quando o initiator peer manda uma DELETE message, uma vez que o espaço desses *chunks* nunca irá ser reclamado.

Após debatermos sobre o problema, decidimos implementar um sistema capaz de capaz de garantir que as mensagens DELETE chegam a todos os peers mesmo que este não estejam a correr no momento que o *initiator peer* manda a mensagem. A nossa resolução consiste apenas numa mensagem adicional que um peer envia pelo MC Channel, quando um *peer* começa a correr, com o formato ALIVE <Version> <SenderId> <CRLF><CRLF>. Em adição a isto, cada *initiator peer* guarda num *HashMap* todas as

messages DELETE enviadas este. Se este *peer* fizer o protocolo BACKUP, deve apagar do *HashMap* as mensagens DELETE desse ficheiro.

Com esta implementação garantimos que todos os ficheiros que devem estar apagados, são realmente apagados quando um peer começa a correr.