Explicação dos melhoramentos dos subprotocolos

Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática e Computação

Turma 1 Grupo 13

Edgar de Lemos Passos - up201404131@fe.up.pt

Nuno Miguel Mendes Ramos - up201405498@fe.up.pt

Neste relatório iremos explicar ao pormenor os quatro melhoramentos implementados nos principais protocolos definidos pelo primeiro trabalho da unidade curricular de sistemas distribuídos:

1. BACKUP;
2. RESTORE;
3. DELETE;
4. RECLAIM;

1. BACKUP

O *enhancement* do ***Chunk backup subprotocol*** tem como objetivo poupar memória nos *peers.* A nossa ideia, no melhoramento deste protocolo, foi utilizar informação que já estava armazenada no *peer* (variável *chunkMap* da classe ChunkManager, que guarda para cada *chunk,* quantos *peers* a têm armazenada).

Basicamente, quando chega uma mensagem PUTCHUNK em vez de executar as operações normais, ou seja, atualizar as estruturas de dados, escrever o ficheiro para a memória e bloquear durante um período aleatório entre 0 e 400 ms (ChunkManager.java :: storeChunk()). Alteramos a ordem das operações, começamos por bloquear o peer durante um período aleatório entre 0 e 400 ms, após este tempo, o *peer* verifica se o número de mensagens STORED recebidas para aquele *chunk,* de um dado ficheiro, já é igual ou superior ao *replication degree* desejado. Se já for, o *peer* descarta a *chunk*. Se não for, o *peer* atualiza as estruturas de dados e escreve o ficheiro em memória (ChunkManager.java :: storeChunk()).

Esta solução permitiu alcançar excelentes resultados. A probabilidade de um *chunk* estar com um *replication degree* superior ao desejado é, aproximadamente, igual a 1/400 (0,25%). Para executar este melhoramento, basta iniciar os *peers* com a versão 2.0.

2. RESTORE

O *enhancement* do ***Chunk restore subprotocol*** visa tornar o envio das mensagens CHUNK mais eficientes, não enviando para os *peers* que não desejam receber**.** Neste protocolo, rapidamente nos apercebemos que a solução para este melhoramento seria recorrer a conexões TCP.

Aquando da chegada de um pedido de *restore*, o *initiator peer* inicia um *ServerSocket* no porto utilizado pelo canal *multicast* de *restore* (PeerService.java :: tcpServer()), desta forma todos os *peers* sabem qual o porto onde o *initiator peer* vai esperar pelas *chunks*. Um *peer* que contenha uma *chunk* desejada e sabendo que o protocolo é a versão melhorada do *restore,* executa tudo da mesma forma da versão normal, excepto quando for enviar a informação. Nessa altura, o *peer* cria um *Socket* com endereço igual ao do pacote recebido na mensagem GETCHUNK*,* e com o porto igual ao explicado em cima (PeerService.java :: sendChunk()). Durante este período, o *initiator peer* está a aceitar pedidos de conexão no *Server Socket* criado. Aquando da chegada de uma mensagem *CHUNK,* o *initiator peer* faz um pequeno pré-processamento da mesma (PeerService.java :: processRestoreMessage()), visto que é um bocado diferente o processamento de um pacote TCP em comparação a um pacote UDP. Após este pré-processamento, é chamado o *handler* das mensagens (PeerService.java :: *messageHandler*()) e a partir daqui o melhoramento é igual à versão normal do protocolo.

Esta implementação teve o efeito desejado, ou seja, apenas o *initiator peer* recebia mensagens CHUNK. Para executar este melhoramento, basta iniciar os peers com a versão 2.0.

3. DELETE

O *enhancement* do ***File Deletion Subprotocol*** funciona com duas novas mensagens, **DELETED** e **AHOY**.

Quando um *peer* chama a versão *enhanced* do protocolo, o *Initiator Peer* copia a entrada correspondente do campo *chunkMap* para um novo *HashMap<String,ArrayList<Integer>>* (ChunkManager.java :: markForDeletion()). Este *HashMap* guardará agora os *peers* que contém uma cópia do *chunk*.

Quando um *peer* recebe a mensagem DELETE, deve responder, após apagar as *chunks* pertencentes ao ficheiro, com a mensagem DELETED, cujo *header* é:

DELETED <protocol_version> <peer_id> <file_id> <chunk_no> <CRLF><CRLF>

Ao receber uma destas mensagens, o *Initiator Peer* irá atualizar este *HashMap.*

No caso de um *peer* estar *offline* quando a mensagem inicial é enviada, não irá apagar as suas cópias das *chunks* do ficheiro.

Para este caso, faz-se com que os *peers* que suportam este *enhancement* passem a enviar, quando se ligam, a mensage AHOY (PeerService.java :: sendGreeting()), cujo *header* é:

AHOY <protocol_version> <peer_id> <CRLF><CRLF>

Quando um *peer* recebe esta mensagem, irá consultar o novo *HashMap,* de forma a verificar que o *peer* que se ligou está presente no *HashMap*, isto é, que o *peer* não tem chunks por apagar. Se isto se verificar, o *peer* irá reiniciar o *File Deletion Subprotocol*, fazendo com que o novo *peer* apague finalmente os *chunks*.

4. RECLAIM

O *enhancement* do ***Space Reclaiming Subprotocol*** visa tornar o protocolo de *backup* de *chunks* **tolerante a falhas**. Isto é alcançado dando aos *peers* a possibilidade de verificar que o *Initiator Peer* do protocolo falhou.

Quando um *peer* recebe uma mensagem PUTCHUNK, coloca um identificador da *chunk* num *HashMap <String, Integer>*, em que o value é o número de vezes que o backup da *chunk* foi pedido. Após isto, o *peer* lança uma *Thread* que vai seguir o *backup protocol*. (PeerService.java :: trackBackup())

Esta thread espera 35 segundos (correspondente ao tempo máximo que pode durar um protocolo de *backup*), e verifica o *HashMap* anteriormente criado.

Se o valor que corresponde a este protocolo for igual ou superior a 5, o *peer* considera que o protocolo falhou sem incidentes, isto é, o *time out* foi atingido normalmente. No entanto, se o valor for inferior a 5, considera-se que o *peer* falhou. Desta forma, o *peer* espera um intervalo aleatório entre 0 e 400 ms para iniciar o *Chunk Backup Subprotocol*. Se verificar que outro *peer* já o iniciou, aborta o protocolo.