Sistemas Distribuídos

·Grupo T5G11 ·
·Anabela Silva - up201506034 ·
·Beatriz Baldaia - up201505633 ·

5 de Outubro de 2018

Resumo

Neste projeto foi desenvolvido um serviço de backup distribuído para uma rede local (LAN) usando-se o espaço livre dos computadores que dela fazem parte. O serviço é fornecido por servidores num ambiente que é considerado cooperativo. No entanto, cada servidor retem o controlo sobre os seus próprios discos e, se necessário, pode recuperar o espaço disponibilizado para fazer o backup de ficheiros de outros computadores. Foram implementados todos os subprotocolos pedidos (*Backup, Restore, Delete e Reclaim Space*) mais os melhoramentos aos mesmos, garantindo-se, também, uma execução concorrente do processamento destes.

1 Subprotocolo de Backup

Inicialmente este protocolo não garante que o *replication degree* seja igual ao desejado, podendo por isso ser superior, ocupando espaço de disco desnecesário. Isto porque os recetores da mensagem PUTCHUNK guardam o seu contéudo em disco e esperam um período aleatório entre 0 e 400 ms para enviar a confirmação.

De forma a impedirmos tal situação invertemos a ordem das operações, ou seja, quanto é recebida a mensagem PUTCHUNK o *peer* espera na mesma entre 0 a 400 ms e depois verifica se o *replication degree* atual do chunk em questão já igual ao pedido. Caso não seja, guarda o chunk em disco, atualiza o *replication degree* atual do chunk e envia uma mensagem de confirmação. O *replication degree* atual de cada chunk vai sendo atualizado, mesmo quando um *peer* espera o tempo aleatório falado, pois, numa thread diferente, ao receber por um outro canal multicast (o MC) a mensagem STORE de um outro *peer* que acabou de guardar o chunk, incrementamos o *replication degree* atual desse chunk e guardamo-lo em memória volátil.

2 Subprotocolo de Restore

Inicialmente a implementação deste subprotocolo pode não ser a desejada para *chunks* grandes visto que apenas o *peer* que pediu o *restore* é que está à espera de receber esse *chunk* e, no entanto, como a mensagem CHUNK é enviada usando um *multicast channel*, todos os *peers* da LAN vão recebê-lo.

Desta forma, se o cliente estiver a usar a versão com *enhancement* a mensagem CHUNK, enviada pelo *peer* que irá transferir um *chunk*, terá no seu *body*, em vez dos dados do *chunk*, o *local host address* do emissor. Antes desta mensagem ser enviada, é criado um *Serversocket* para onde o *peer* escreverá os dados do *chunk*. Usamos o construtor *ServerSocket(int port)* com *port* igual a 0, o que significa que automaticamente uma porta disponiveis é alocada para o socket criado. O que fez o pedido *restore*, ao receber o *host address* na mensagem CHUNK, abre uma conecção TCP e começa a ler bytes (corpo do chunk) do socket.

3 Subprotocolo de Delete

Inicialmente este protocolo não garante que os ficheiros apagados num *peer* sejam apagados noutro *peer* que não se encontre ativo no momento em que a mensagem DELETE é enviada.

Nesse sentido criamos duas novas mensagens:

- DELETED, enviada como resposta ao DELETE:

  `DELETED <version> <senderID> <fileID> <CRLF><CRLF>`

- CHECKDELETE, enviada na iniciação de um peer, se ele tiver *chunks*:

  `CHECKDELETE <version> <senderID> <fileID> <CRLF><CRLF>`

Sempre que um *peer* recebe uma mensagem DELETE, apaga os *chunks* correspondentes e a informação armazenada sobre eles. De seguida envia uma resposta DELETED com o seu id e com o identificador do ficheiro apagado. O *peer* que fez o pedido DELETE diminui o *replication degree* dos *chunks* desse ficheiro guardados pelo *peer* que enviou a resposta. Assim, o *peer* que funciona como *initiator* só apaga a referência ao ficheiro quando o grau de replicação de todos os seus *chunks* é zero.

A mensagem CHECKDELETE é enviada pelo canal *MC* quando um *peer* se ativa, avisando os *peers* dos ficheiros aos quais está a fazer *backup*, a fim de receber uma confirmação de se tais ficheiros já foram ou não apagados durante o seu período de inatividade. Ou seja, se os identificadores dos ficheiros enviados na mensagem CHECKDELETE estiverem, em algum dos *peers* da LAN, notificados como apagados, o *peer* que tem tal informação reenvia uma mensagem DELETE com a versão com *enhancement*.

4 Execução Concorrente

Todos os subprotocolos usam um canal *multicast*: canal de controlo (MC), canal para transmissão de dados (MDB) e canal de recuperação de dados (MDR). Existe uma thread para cada um destes canais que os vai estar a escutar infinitamente, assim garantimos que podemos estar a receber diferentes tipos de mensagens ao mesmo tempo.

Uma abordagem linear, mas limitante, do processamento das diferentes mensagens seria o seu processamento sequêncial. A forma de suportarmos um processamento concorrente é criando uma thread para cada nova mensagem recebida. É uma abordagem simples, mas introduz uma sobrecarga ao sistema com a criação e termino das threads. Uma melhor opção que foi a que implementamos no nosso projeto, é o uso de uma *thread pool* recorrendo à classe **java.util.concurrent.ThreadPoolExecutor** . Assim, para diferentes serviços existem diferentes protocolos. No nosso trabalho cada subprotocolo é um objeto *runnable*. Quado uma certa mensagem é recebida, criamos um desses objetos e colocamo-lo na *thread pool* que os executará.

Certos protocolos exigem uma espera, como o *random delay* exigido antes do envio das mensagens CHUNK e STORED e antes do começo de um *backup* necessário numa situação de *reclaiming* e o tempo de espera pelas confirmações ao serviço *backup*. Se fosse aplicado Thread.sleep() para todos estes casos encorreriamos à coexistência de diferentes thread que estaria a consumir recursos, limitando a escalabilidade do programa. Desta forma, usamos a classe **java.util.concurrent.ScheduledThreadPoolExecutor** que nos permite "agendar"um *"timeout"handler* sem que para tal seja necessário o uso de uma thread antes de tal tempo limite ser esgotado. Isto é, enviamos na mesma os subprotocolos *runnable* já referidos para a *thread pool* mas pedimos para estes só começarem a ser executados passado t tempo. Tal é possível com a invocação do método *schedule(Runnable command, long delay, TimeUnit unit)*.

A informação recebida das mensagens e resultante do processamento das mesmas deve ser guardada numa estrutura de dados que é frequentemente visitada para o tratamento lógico dos subprotocolos. Como seria de prever, diferentes threads estarão a aceder a essa estrutura por se tratar de memória partilhada. De forma a evitarmos conflitos e corrupção da informação por acessos simultâneos a essa estrutura, usamos a classe **java.util.concurrent.ConcurrentHashMap**. Esta classe suporta acessos simultâneos e ajusta a concorrência esperada para atualizações. Mesmo que todas as operações sejam *thread-safe* (o código manipula estruturas de dados partilhadas assegurando que o comportamento das threads é o previsível e desejado, sem intereções não intencionais), operações de recuperação não implicam bloqueio e não há forma de bloquear a tabela por completo para se impedir o acesso à mesma. Métodos de obtenção de dados, como o *get*, geralmente não são bloqueadas, por isso podem se sobrepor às operações de atualização, como *put* e *remove*. O objetivo da ConcurrentHashMap é aumentar a taxa de transferência, com concorrência, permintindo-se leituras/escritas simultâneas na tabela sem bloquear a tabela inteira. Assim, temos uma classe LocalState com uma ConcurrentHashMap onde a chave é o identificador do ficheiro e o valor é um Objeto BackupFile. Este BackupFile também tem uma ConcurrentHashMap onde a chave é o número de um *chunk* e o valor o *chunk* em si (objeto Chunk). De facto, sempre que queremos guargar um novo chunk, corremos a instrução

da classe LocalState *backupFiles.compute(fileID, (k,v)->computeSaveChunk(k, v, pathName, serviceID, replicationdeg, chunk))* que usa o método compute da ConcurrentHashMap. Este método processa a entrada na tabela para a chave k e o seu valor atual v (pode ser nulo se esta entrada não existir na tabela) executando a função passada como segundo argumento, no nosso caso, a função computeSaveChunk. Enquanto esta função se encontra em execução, operações de atualização da entrada em questão que forem tentadas por outras threads serão bloqueadas, o que nos garante que os dados não serão corrompidos por acessos simultâneos a memória partilhada.

A fim de removermos qualquer bloqueio possível no momento de acesso ao sistema de ficheiros, usamos a classe **java.nio.channels.AsynchronousFileChannel**. Um canal assíncrono é criado quando um ficheiro é aberto usando um dos métodos definidos por esta classe. Não existe uma posição atual no ficheiro. A posição no mesmo é especificada a cada leitura e escrita. Para estas duas operações é necessário especificar um *CompletionHandler* que é invocado aquando da operação I/O. Um *AsynchronousFileChannel* está associado uma *thread pool* à qual são submetidas tarefas para processarem eventos I/O. Deste modo, outras operações podem estar a ocorrer ao mesmo tempo, visto que estes eventos não requerem uso do processador.