Serverless Distributed Backup Service

3MIEIC05-13: João Esteves, Renato Campos

3 de Abril de 2018

1 Introdução

Foi desenvolvido um serviço de backup distribuído para uma LAN (*Local Area Network*). A ideia é usar o espaço livre em disco dos computadores numa LAN para guardar ficheiros em outros computadores na mesma LAN. O serviço é disponibilizado por servidores num ambiente que é assumido cooperativo (em vez de hostil). Não obstante, cada servidor retem controlo sobre os seus próprios discos e, se necessário, pode recuperar o espaço que fez disponível para guardar ficheiros de outros computadores.

2 Melhoramentos

2.1 Backup

Na versão base, um peer iniciador envia uma mensagem PUTCHUNK para guardar um dado chunk, sendo este aceite por qualquer peer que o receba e confirmado por uma mensagem de confirmação STORED de cada um destes peers. Cada peer também está constantemente a registar as mensagens STOREDs que receba para que tenha uma visão global dos graus de replicação de todos os chunks.
O problema deste esquema é que, caso hajam mais peers que o grau de replicação desejado, os peers extra guardam o chunk desnecessariamente, reduzindo o seu espaço livre e causando um volume extra na rede de mensagens STORED vindas de todos estes peers.
Para ultrapassar esta situação, e aproveitando o registo de STOREDs em todos os peers, cada peer na versão melhorada espera um tempo aleatório entre 10 e 400ms antes de executar o protocolo de receção de um PUTCHUNK e, caso já tenha registado um número de STOREDs para este chunk maior ou igual ao seu grau de replicação desejado, não o guarda nem envia uma mensagem de confirmação. Assim, o grau de replicação é reforçado a ser o realmente desejado e não o máximo possível.

2.2 Restauro

Na versão base, o Iniciador envia um pedido GETCHUNK por Multicast e quem tiver o chunk responde-lhe com uma mensagem CHUNK que contém o chunk, também em Multicast. Como a mensagem contém o chunk esta poderá ser grande, e sendo Multicast todos os Peers irão recebê-la desnecessariamente.

Para corrigir esta questão, o CHUNK é agora enviado por TCP ao peer que o requisitou, ou seja, apenas o peer que o pede é que o recebe. Para permitir esta conexão TCP, o hostname e a porta do peer que requisita o restauro são enviados na mensagem GETCHUNK, na segunda linha do header.

2.3 Delete

O problema deste protocolo na versão base é que, sendo enviada apenas uma mensagem do iniciador para apagar um ficheiro, os peers que se encontrem desligados no momento de envio não irão remover as suas partes do ficheiro. Como melhoramento, corrigimos este comportamento através de respostas ao delete e de uma tarefa periódica para enviar novas mensagens de delete.
Mais concretamente, os Peers a cada 30 segundos verificam se há DELETEs a reenviar caso não tenham recebido um número de confirmações de DELETE igual ao número do real grau de replicação dos mesmo, e há o seguinte flow: Peer P tem um ficheiro guardado; Peer Iniciador envia DELETE desse ficheiro enquanto Peer P está desligado e adiciona esse ficheiro a uma lista de ficheiros a apagar; Iniciador não recebe resposta; P é ligado; Iniciador manda novo DELETE quando é executada a sua tarefa periódica; P apaga ficheiro e envia confirmação; Iniciador recebe confirmação e diminui o seu registo do real grau de replicação desse ficheiro para 0, retirando o ficheiro da lista de ficheiros a reenviar o DELETE.

3 Concurrency Design

O projeto recorre a multithreading e leitura/escrita assíncrona de ficheiros para suportar a execução concorrente de protocolos. Em cada Peer existe uma thread por cada um dos três canais multicast, ver figura 1, cada uma com a sua thread pool, sendo elas *MCListener*, *MDBListener* e *MDRListener*. Cada vez que é recebida uma mensagem num canal, esta é processada por uma das threads da pool. Para além destas 3 threads *listeners*, cada Peer tem métodos que podem ser invocados por RMI para que por ele sejam iniciados os diversos protocolos tais como o backup de algum ficheiro, sendo que cada um destes métodos reencaminham o processamento necessário para uma nova thread. A leitura e escrita assíncrona de ficheiros é feita com uso da biblioteca java.nio.channels.AsynchronousFileChannel.

3.1 Backup de um Ficheiro

Começa com uma thread a executar SendStoreFile que contém a sua própria thread pool. A classe lê as várias "chunks"do ficheiro de forma assíncrona com tamanho máximo 64KB e envia cada "Future"para uma thread a executar StoreChunk. Por sua vez, a StoreChunk ao criar o pacote, executa o método get no "Future"para obter os dados. De seguida, a thread StoreChunk envia um pacote PUTCHUNK pelo MDB e vai lendo o número de confirmações obtidas no MC. Ver 3.2.2.
Quando o Peer recebe um PUTCHUNK no canal MDB, este executa numa thread da pool o PutChunkReceive que, conseguindo guardar o chunk ou este já estiver guardado, envia uma mensagem STORED pelo MC.
O listener do MC vai detetar o STORED e adicionar esta confirmação a um objeto partilhado, sendo daqui que é obtido o número de confirmações necessário para a confirmação do envio do PUTCHUNK no StoreChunk.

3.2 Código

3.2.1 Criação de uma pool

```
private ExecutorService pool =
    Executors.newCachedThreadPool();
```

3.2.2 Envio PUTCHUNK e leitura de confirmações dum objeto partilhado

```
...
mdbSocket.send(chunkPacket); //ENVIO PUTCHUNK
...
numConfirmations = repStatus.getNumConfirms(fileId,
    chunkNo); //OBJETO PARTILHADO
...
```

4 Figuras

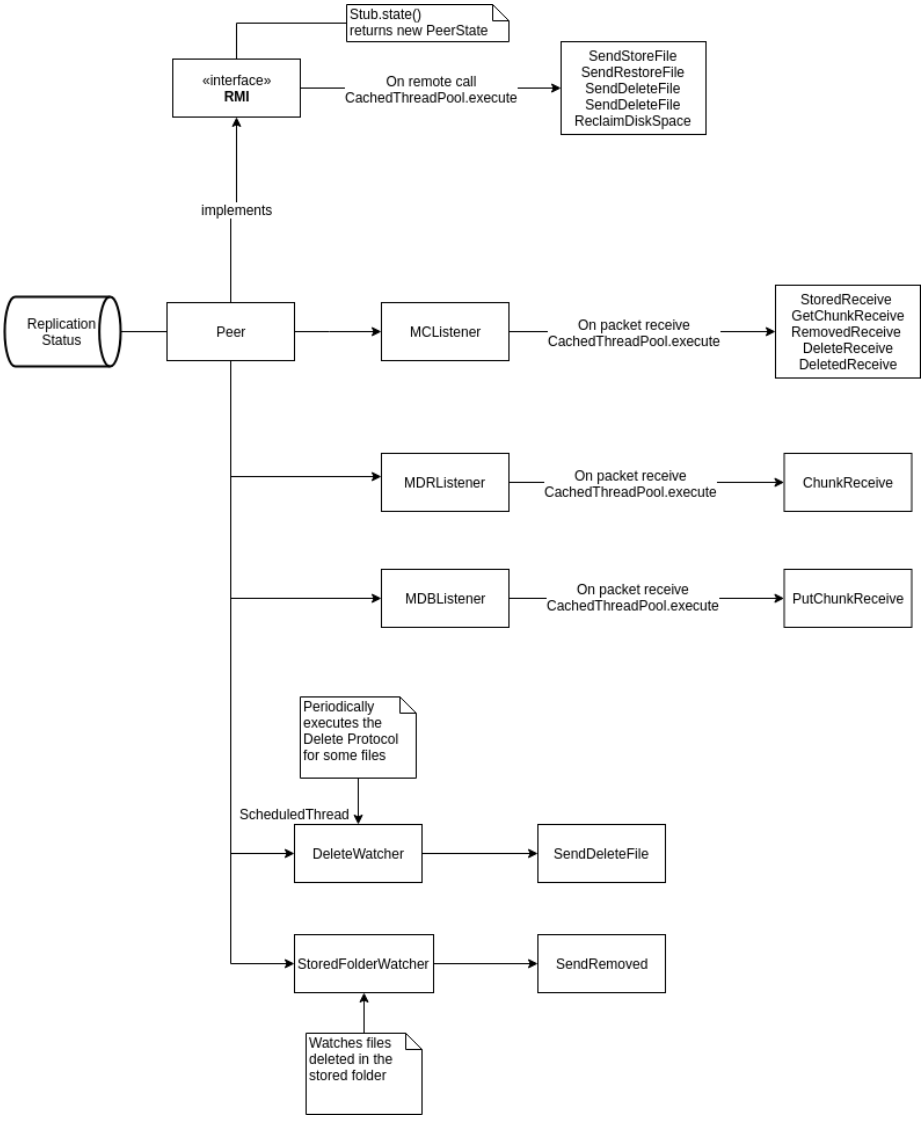

Figura 1: UML describing concurrency