Relatório

Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática e Computação

(Abril de 2018)

Projeto 1 - *Distributed Backup Service*

Grupo T3G02

André Filipe Pinto Esteves up201606673@fe.up.pt

Joana Sofia Mendes Ramos up201605017@fe.up.pt

I. *Backup Enhancement*

Para o *enhancement* do protocolo de *backup* ao receber uma mensagem *PUTCHUNK* a *thread* que a processa:

1. Guarda **temporariamente**, **internamente** e **localmente** o *chunk*, se não tiver iniciado o seu backup, sem mandar *STORED* (este "*backup* temporário" serve para manter conta de quantos *peers* guardam este *chunk* e quais);

```
Chunk chunk = new Chunk(fileID, chunkID, chunkByte, chunkByte.length, rd);
if(this.storage.storeChunk(key, chunk, this.peerID)) {
    this.storage.decSpaceAvailable(chunk.getSize());
    chunk.storeChunk(peerID);
    Peer.savesInfoStorage(this, this.storage);
}
```

2. Agenda o envio da mensagem *STORED*, com um *delay* entre 0 e 400 ms, fazendo uso do método *schedule* da classe *ScheduledThreadPoolExecutor*. Este método retorna um objeto do tipo *ScheduleFuture*, que é guardado num *ConcurrentHashMap* em que a chave corresponde a um par <ID do ficheiro, ID do *chunk* a guardar> e o valor a esse objeto do tipo *ScheduleFuture*;

```
this.waitingOnStoreds.put(key, this.scheduledExecutor.schedule(sendStored, delay, TimeUnit.MILLISECONDS));
```

3. Se se receber uma mensagem *STORED* é verificado no *ConcurrentHashMap* se existe algum envio de uma mensagem *STORED* pendente para esse ID do ficheiro e *chunk* e obtém-se o seu valor. Se existir, após atualizar o *replication degree* observado verifica-se se ele é maior ou igual ao *replication degree* exigido, e, se for, é cancelada a thread que envia *STORED* desse *chunk* e é anulado o seu "*backup* temporário" e removida do *ConcurrentHashMap*. Isto é feito através do método *cancel* da classe *ScheduleFuture*;

```
ScheduledFuture storeChunk = this.waitingOnStoreds.get(key);
if(Float.compare(this.version, 1.0F) != 0 && storeChunk != null) { // enhanced version
    if(chunk.getObservedRD() - 1 >= chunk.getDesiredRD()) {
        storeChunk.cancel(true);
        this.storage.deleteChunk(key, this.peerID);
    }
}
```

Note-se que é necessário fazer -1 porque localmente está-se a guardar o *chunk*.

4. Se se receber uma mensagem *STORED* de um *chunk* pendente e a mensagem for do próprio *Peer*, significa que já passou o *delay* e o *Peer* passou então a guardar efetivamente e para toda a rede aquele *chunk*, podendo-se retirar o objeto do *ConcurrentHashMap*.

II. Concorrência

Uma vez que este projeto se trata de um sistema distribuído, é crucial a existência de ocorrências de várias ações a serem executadas ao mesmo tempo. Como tal, é necessária atender o máximo de pedidos em simultâneo, de forma a maximizar a utilização do serviço, mas ao mesmo tempo garantir que não há sobrecarga da rede.

Assim, na implementação deste projeto usufruímos dos executores de Java, para que, de tal modo, os diversos processos pudessem ser executados em simultâneo deixando a *main thread* sempre disponível para receber novos pedidos a serem também lançados por novas *threads*. Utilizamos dois tipos de executores: *ThreadPoolExecutor* e *ScheduledThreadPoolExecutor*.

Cada *Peer* tem o atributo de cada canal *multicast* (**MC**, **MDB** e **MDR**). O *Peer* começa por lançar 3 *threads*, uma por canal, que ficam à escuta de mensagens no respetivo canal.

```java
@Override
public void run() {

    try {
        byte[] receiveData = new byte[66000];

        MulticastSocket socket = new MulticastSocket(this.portNumber);
        socket.joinGroup(this.address);

        while(true) {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            socket.receive(receivePacket);
            byte[] copy = Arrays.copyOf(receiveData, receivePacket.getLength());
            this.peer.getExec().execute(new ReceiveMessage(this.peer, copy));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Em cada canal, sempre que é recebida uma nova mensagem, *socket.receive(receivePacket);* , é feita a cópia da mesma para assim não haver perda (sobreposição) de informação aquando de uma sobrecarga de mensagens e também a execução de uma nova thread que processa a mensagem recebida (classe *ReceiveMessage*), para que, assim possibilite o canal estar sempre à escuta de novas mensagens.

Em relação à escolha das estruturas de dados, acabamos por utilizar *ConcurrentHashMap<>(),* em vez de *HashMap<>()* , pois é o mais adequado quando é

necessário ter uma elevada concorrência e também é *thread safe*. Para a chave do mapa usamos um *AbstractMap.SimpleEntry<String, Integer>*, em que a chave é o fileID (com *hash*) e o valor é o ID do respectivo *Chunk*.

```
private ConcurrentHashMap<AbstractMap.SimpleEntry<String, Integer>, InitiatedChunk> initiatedChunks;
private ConcurrentHashMap<AbstractMap.SimpleEntry<String,Integer>, Chunk> storedChunks;
private ConcurrentHashMap<AbstractMap.SimpleEntry<String, Integer>, byte[]> restoredChunks;
private ArrayList<FileManager> storedFiles;
```

O valor do *ConcurrentHashMap<>()* pode ser:

- *InitiatedChunk:* possui a informação acerca do *replication degree* observado e desejado do Chunk e também os IDs dos peers que o possui;
- *Chunk*: possui toda a informação relativa a um *Chunk*;
- *byte[]*: array que contém a informação de um *Chunk* para restaurar um ficheiro.

Para além disso, todos os métodos que modificam dados partilhados por várias *threads* são *synchronized*, isto assegura que cada *thread* acede aos dados sem que outras acedam ao mesmo tempo, podendo fazer alterações atomicamente.