

Serviço distribuído de cópias de segurança

Relatório do projeto 1



Mestrado Integrado em Engenharia Informática e
Computação

Sistemas Distribuídos - FEUP-EIC0036

Grupo T2G02:

Daniel Ribeiro de Pinho - 201505302

Francisco Tuna de Andrade - 201503481

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

2 de Abril de 2018

Conteúdo

1	Introdução	3
2	Concorrência no programa	4
3	Backup Enhancement	5

1 Introdução

No âmbito da unidade curricular de Sistemas Distribuídos foi-nos proposta a implementação de um sistema distribuído destinado à execução de cópias de segurança de ficheiros. Estes ficheiros são armazenados em bocados que vão até aos 64000 bytes, sendo transmitidos através de pares (ou *peers*, sendo neste documento usado o termo em português), usando o protocolo UDP.

O sistema inclui várias funcionalidades que são acedidas pelo cliente, incluindo a armazenagem e recuperação de ficheiros, para além de existir a possibilidade de recuperar espaço de disco nos peers e de remover ficheiros do sistema. A nossa implementação ainda inclui uma melhoria ao processo de armazenamento (*backup*) de ficheiros.

Neste documento segue-se a descrição da implementação de concorrência no programa e da melhoria ao processo de armazenamento.

2 Concorrência no programa

Tendo em conta o âmbito do projeto, em que é preciso ler e escrever dados de forma simultânea, respondendo a vários pedidos (dos clientes e dos peers), é importante que exista um mecanismo de concorrência no que toca às várias partes do programa.

A implementação presente segue um modelo em que cada canal de comunicação (MC, MDR e MDB) é monitorizado por uma *thread*, de forma a que haja uma correspondência unívoca; estas *threads* estão presentes nas classes `ThreadMC`, `ThreadMDR`, e `ThreadMDB`. Estas classes estendem a classe abstrata `MulticastThread`, que implementa a interface `Runnable`.

Como o seu nome indica, `ThreadMC` monitoriza o canal *multicast* dedicado à troca de mensagens gerais, recebendo e atendendo aos pedidos de pacotes com headers `STORED`, `PUTCHUNK`, `DELETED` e `REMOVED`. Após a receção de um pacote destes tipos, a *thread* chama uma função de forma a processar o pedido recebido, voltando a escutar o canal quando estiver disponível.

De forma análoga, `ThreadMDB` monitoriza o canal *multicast* dedicado à troca de mensagens de armazenamento de dados, lidando com pacotes `PUTCHUNK`. Ao receber um pacote deste tipo, a *thread* analisa a situação e guarda o conteúdo do pacote, caso seja oportuno.

Finalmente, temos `ThreadMDR`, que monitoriza o canal *multicast* dedicado à troca de mensagens de recuperação de dados, com pacotes `CHUNK`. Esta interage com o `Peer` de forma a assegurar que está a receber o pacote que pretende.

Adicionalmente, os protocolos iniciados pelos pares são delegados a somente uma única *thread*. Este modelo é baseado no ponto 3 do documento anexado ao enunciado deste projeto (uma *thread* por canal, uma instância de protocolo). Cada *initiator* é chamado por uma função que implementam métodos definidos por uma interface do Java, permitindo assim que o cliente comunique com os peers através de RMI. Existem um total de 4 *initiators*: `Backup`, `Restore`, `Delete` e `Reclaim`.

O *initiator* `Backup` inicia o *Backup* do protocolo. Este *initiator* chama várias *threads*, sendo que cada uma delas inicia o *Backup* de um chunk do ficheiro.

O *initiator* `Restore` é responsável por enviar as mensagens de `GETCHUNK` dos vários chunks do ficheiro que ele quer restaurar e é também depois responsável por juntar o ficheiro através das mensagens `CHUNK` que recebe como resposta.

O *initiator* `Delete` envia a mensagem `DELETE` aos restantes peers, não recebendo qualquer mensagem como resposta.

Por fim, o `Reclaim` envia a mensagem `REMOVED` aos restantes peers, não recebendo, tal como é o caso de `Delete`, qualquer mensagem de resposta.

As 3 *threads* que monitorizam a chegada das mensagens aos canais e os 4 *initiators* necessitam de aceder a certas estruturas de dados comuns a todas elas. Para tal definiu-se uma classe principal `Peer.java`, onde todos os atributos são *static* e acessíveis através de métodos *get*. Existem dois atributos nesta classe `Peer.java` que necessitam de estar acessíveis a múltiplas *threads*, sendo eles o atributo `chunksInPeer` e o atributo `fileStores`.

O `chunksInPeer` mantém o registo dos *chunks* guardados por um determinado peer, sendo guardado em memória não volátil num ficheiro. Como este atributo necessita de ser acedido e modificado por várias *threads* foi declarado como um `ConcurrentHashMap`, com o intuito de evitar *race conditions*:

```
ConcurrentHashMap<String , ArrayList<Integer>> chunksInPeer
```

A *String* do *hashMap* corresponde ao identificador de um ficheiro, enquanto que o *ArrayList < Integer >* corresponde ao conjunto de *chunks* desse ficheiro que o determinado peer em questão guardou.

Por outro lado, o **fileStores** regista para cada mensagem **STORED** que recebeu, o id do peer que a enviou e o **chunk** a que essa mensagem corresponde, sendo a estrutura responsável por registar a *perceived replication degree* de um determinado peer, algo que será usado no algoritmo de libertação de espaço do protocolo **RECLAIM**. Tal como o atributo descrito anteriormente, o **fileStores** necessita de ser acedido e modificado por várias **threads** ao mesmo tempo, pelo que também foi declarado como um **ConcurrentHashMap**:

```
ConcurrentHashMap<String , ChunkStoreRecord> fileStores
```

A *String* deste *hashMap* corresponde ao identificador de um ficheiro, enquanto que a classe **ChunkStoreRecord** foi uma classe por nós implementada e cujos atributos principais são mostrados abaixo:

```
public class ChunkStoreRecord implements Serializable {
    public ConcurrentHashMap<Integer , ArrayList<Integer>> peers;
    private int replicationDeg;
    private int peerInit;
    private String fileName;
    ...
}
```

É de chamar atenção ao atributo **peers** desta classe que como necessita de ser acedido e modificado por várias *threads* foi também declarado como um *ConcurrentHashMap*. Neste atributo o *Integer* refere-se ao número de um determinado **chunk**, enquanto que *ArrayList < Integer >* se refere à lista dos peers que guardaram esse **chunk**.

3 Backup Enhancement

No nosso projeto, nós implementamos um *enhancement* ao protocolo **Backup** com o objetivo de que o sistema não faça muito mais cópias de um ficheiro do que aquilo que o seu grau de replicação exige. Para tal, de cada vez que um peer recebe uma mensagem **STORED**, ele regista na estrutura **fileStores** já referida na secção anterior o peer que a enviou e o **chunk** correspondente.

Quando um peer recebe uma mensagem de **PUTCHUNK**, antes de fazer store ele espera um tempo aleatório entre 0 e 400ms e verifica que o número de peers que guardaram um determinado **chunk** é estritamente inferior ao seu grau de replicação e só nessa situação é que guarda esse **chunk**.