U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Serviço Distribuído de Backup

SISTEMAS DISTRIBUÍDOS 20/21
MIEIC 3º ANO

TURMA 4 - GRUPO 6

Carlos Lousada, up201806302
José David Rocha, up201806371

1 Sumário

Este trabalho foi elaborado no âmbito da unidade curricular de Sistemas Distruibuídos e teve como objetivo desenvolver um sistema distribuído de *backup*.

Com efeito, o trabalho foi concluído com sucesso - o sistema distribuído foi implementado com sucesso e a aplicação desenvolvida foi capaz de executar todos os protocolos referidos no enunciado.

Sendo assim, este relatório tem como objetivo explicar detalhadamente as melhorias implementadas nos protocolos utilizados bem como descrever o design escolhido para a execução simultânea dos mesmos e sua respetiva implementação.

Índice

2 *Backup*

A melhoria do protocolo de *Backup* focou-se na garantia de que o *Replication Degree* desejado correspondia ao *Replication Degree* real.

Sendo assim, foi implementada uma classe ***Storage***, de forma a poder guardar um registo daquilo que é enviado/recebido através dos canais *Multicast*. Assim, através do uso de um *ConcurrentHashMap* ***chunkOcurrences*** é possível guardar o número de ocurrências de um determinado *Chunk* em vários *Peers*. Cada entrada deste *ConcurrentHashMap* usa como chave uma junção do seu *File ID* com o seu respetivo número de *Chunk*. O valor respetivo a uma determinada chave representa o número de Peers que possuem armazenado um determinado *Chunk*. Este valor é baseado na quantidade de mensagens do tipo *STORED* que um *Peer* recebe.

Na prática, sempre que um *Peer* recebe uma mensagem do tipo *PUTCHUNK*, e após ter esperado um tempo aleatório entre 0 e 400 *ms*, é consultada a tabela ***chunkOcurrences*** de modo a comparar o valor respetivo ao *Replication Degree* atual com o valor desejado, que é enviado na mensagem *PUTCHUNK*. Assim, caso o valor presente na tabela seja inferior ao valor desejado, o *Peer* procede à escrita do ficheiro, atualiza a sua estrutura de dados e envia uma mensagem do tipo *STORED*. Caso contrário, o *Chunk* é descartado.

Com efeito, esta solução revelou ser bastante eficiente, no sentido em que permitiu reduzir drasticamente a probabilidade do *Replication Degree* ser diferente do desejado. Contudo, é de notar que pode ocorrer casos em que este seja superior ao desejado, que, na sua maioria, ocorre quando dois Peers acedem ao mesmo tempo aos seus *ConcurrentHashMaps*, antes que um deles tenha tempo de enviar uma mensagem do tipo *STORED*.

3 *Delete*

A melhoria do protocolo de *Delete* foca-se na garantia de que, caso um *Peer* não esteja conectado no momento em que é feito o pedido de *Delete* de um determinado ficheiro, o espaço para ele alocado não seja inevitavelmente perdido.

Para tal, foram adicionadas duas mensagens de controlo à nova versão do protocolo (*Protocol Version 1.9*): *LOGGEDIN* e *DELETED*. Um *Peer* que utilize esta versão, antes de enviar uma mensagem do tipo *DELETE*, irá também guardar uma lista de *"Deleted Files"*, de forma a controlar quais ficheiros ainda não foram totalmente apagados. Para fazer esta verificação, utiliza-se o *ConcurrentHashMap* ***chunkOcurrences*** para verificar qual o número de *chunks* que deverão ser apagados.

Relativamente aos *Peers* em geral, ao utilizar esta versão, logo após ser feita a conexão, envia uma mensagem do tipo *LOGGEDIN* de estrutura: $< Version > LOGGEDIN < SenderId >< CRLF >< CRLF >$. Desta forma, o initiator peer irá analisar a sua lista de *Deleted Files* e enviar os respetivos *DELETE* que considerar necessário.

Para além disso, cada peer, após receber um *DELETE* e proceder, caso seja necessário, à sua remoção, envia também uma mensagem do tipo DELETED, de estrutura: $< Version > DELETED < DestinationId >< FileId >< NoOfChunksDeleted >$. O *Peer* com *PeerId* igual a *DestinationId* irá assim atualizar o número de *chunks* que deverão ser apagados, subtraindo ao número atual o *NoOfChunksDeleted*. Desta forma, esta solução, apesar de utilizar duas mensagens extra ao protocolo proposto, terá uma eficiência consideralmente alta, visto não ser enviada uma mensagem de *DELETE* para cada um dos ficheiros que foram apagados durante a utilização do serviço, mas sim apenas uma por cada ficheiro que ainda não tenha sido totalmente removido de todos os sistemas.

4 *Execução Simultânea de Protocolos*

O design implementado ao longo do programa permite que vários protocolos executem ao mesmo tempo. Para tal, foi usada a classe *ScheduledThreadPoolExecutor* com uma *Thread Pool* de 128, permitindo assim a implementação de um *timeout* que não bloqueie uma dada *thread* que esteja a correr, dado que este método admite um tempo limite em que não ocorre nenhuma outra *thread* até esta terminar.

A classe ***Peer***, fulcral para o funcionamento do programa, possui um atributo para cada canal: ***MCChannel MC***, ***MDBChannel MDB***, ***MDRChannel MDR***, de modo a que cada canal estivesse apenas associado a uma *thread*.

```
1  public class Peer implements RemoteInterface{
2      // ...
3      private static MCChannel MC;
4      private static MDBChannel MDB;
5      private static MDRChannel MDR;
6      private static ScheduledThreadPoolExecutor exec;
7      private static Storage storage;
8
9      private Peer(String MCAddress, int MCPort, String MDBaddress, int MDBport,
        String MDRAddress, int MDRPort) throws IOException, ClassNotFoundException {
10         exec = (ScheduledThreadPoolExecutor)Executors.newScheduledThreadPool(50);
11         MC = new MCChannel(MCAddress, MCPort);
12         MDB = new MDBChannel(MDBaddress, MDBport);
13         MDR = new MDRChannel(MDRAddress, MDRPort);
14     }
15     // ...
16 }
```

Relativamente às estruturas de dados utilizadas, foi feito uso da classe *ConcurrentHashMap*, já referido por diversas vezes neste relatório. A razão para tal decisão prende-se com o facto de esta ser adequada para ambientes *multithreaded*. Comparativamente a um *HashMap*, um *ConcurrentHashMap* é sincronizado, *Thread Safe* e possui melhor *performance* e escalavilidade em ambientes deste género.

Este *ConcurrentHashMap* é armazenado na classe ***Storage***, que implementa um método *Serializable*. A **serialização de um objeto** consiste em representar um objeto de uma classe como uma sequência de *bytes*, que depois são armazenados num ficheiro do tipo *.ser*. Este mecanismo permite armazenar o estado de um sistema num ficheiro, que pode ser deserializado sempre que o sistema volta a estar ativo. Para tal são usadas as funções ***serialize*** e ***deserialize***.

```
public class Storage implements Serializable {
    private final int peer_id;
    private final ArrayList<Chunk> backedChunks;
    private final ArrayList<FileC> backedFiles;
    private ArrayList<Chunk> receivedBackedChunks;
    private ConcurrentHashMap<String, Integer> chunkOcurrences;
    private ConcurrentHashMap<String, Integer> reclaimedChunks;
    private final int capacity;
    private int spaceavailable;
    private ArrayList<FileC> deletedFiles;

    public Storage(int peer_id) throws IOException, ClassNotFoundException {
        this.peer_id = peer_id;
        this.backedChunks = new ArrayList<>();
        this.backedFiles = new ArrayList<>();
        this.deletedFiles = new ArrayList<>();
        this.receivedBackedChunks = new ArrayList<>();
        this.chunkOcurrences = new ConcurrentHashMap<>();
        this.reclaimedChunks = new ConcurrentHashMap<>();
        this.capacity = 1000000;
        this.spaceavailable = this.capacity;
    }
}
```

Finalmente, o processamento de todos as mensagens recebidas é feita por uma *thread* específica, ***ReceivedMessagesHandler***. Esta *thread* processa a mensagem recebida e lança novas *threads* com base no tipo de protocolo presente no *header* da mensagem.