1º Projeto de SDIS

Project 1 – Distributed Backup Service

Engenharia Informática e Computação

Regente: Pedro Alexandre Guimarães Lobo Ferreira Souto

Turma 7

Grupo2

- **Duarte Nuno Esteves André Lima de Carvalho– up201503661- up201503661@fe.up.pt**
- **Tomás Nuno Fernandes Novo – up201604503 - up201604503@fe.up.pt**

1 - Introdução

No âmbito da unidade curricular Sistemas Distribuídos foi-nos proposto como 1ª projeto do trabalho prático a implementação de um serviço de *backup* distribuído para uma LAN.

Este relatório tem o fim de explicar com maior detalhe a implementação da execução concorrente dos protocolos ***backup, delete, restore*** e ***reclaim***.

2 – Execução Simultânea de Protocolos

Com o objetivo de permitir a execução simultânea de protocolos, foi indispensável ter em conta diversos aspetos.

Principiando pelas estruturas de dados, priorizámos o uso das tabelas ***ConcurrentHashMap*** devido à sua segurança, maleabilidade e facilidade de manipulação no ambiente ***multi-thread,*** além de que possui um desempenho adequado quando existem mais ***threads*** a ler do que a escrever. Estas tabelas encontram-se na classe **Cloud.**

```
/**
 * stores received and backedUp chunks
 * key = <fileID>
 */
    private ConcurrentHashMap<String, Chunk> storedChunks;

    /**
 * stores the number of chunks a file was splited into
 * key = <fileID>
 */
private ConcurrentHashMap<String, Integer> numberOfFileChunks;

/**
 * registers the peers that have stored chunks sent by this peer
 * key = <fileID>:<ChunkNo>
 */
private ConcurrentHashMap<String, HashSet<Integer>> colaborativePeers;

/**
 * stores the number of times a chunk was stored
 * key = <fileID>:<ChunkNo>
 */
private ConcurrentHashMap<String, Integer> numberOfChunksConfirmations;
```

```
/**
 * key = <fileID>:<ChunkNo>
 */
private ConcurrentHashMap<String, Integer> chunksToRestore;

/**
 * Holds chucks restored by restore protocol
 * key = <fileID>:<ChunkNo>
 */
private ConcurrentHashMap<String, Chunk> recoveredChunks;

/**
 * stores the number of chunks a file was splited into
 * key = <fileID>
 */
private ConcurrentLinkedQueue<String> numberOfFileChunksToBeRestored;
```

Com a **serialização de um objeto,** um dos muitos mecanismos que a linguagem ***JAVA*** nos oferece, é possível armazenar um objeto numa sequência de bytes que contêm não só os dados e tipo do respetivo objeto como o tipo e dado dos seus dados armazenados. Este objeto é escrito num ficheiro com extensão "***.ser***" e pode ser usufruído com o fim de recriar o objeto em memória.

Assim, utilizando este mecanismo, como cada peer tem um objeto do tipo **Cloud,** foi-nos possível armazenar o estado de cada *peer* com a serialização da **Cloud,** armazenando dados importantes (as ***ConcurrentHashMap***'s) no diretório "../database/peerID" onde ID é o ID do respetivo **peer**, armazenando os dados da **Cloud** de cada **peer** no seu respetivo diretório. A serialização destes dados é realizada no método ***saveToDisk()*** e a sua de-serialização na função ***loadFromDisk()***.

As tabelas referidas em encontram-se presentes na classe ***Cloud,*** que também possui um atributo por canal *multicast*: **MC**, **MDB** e **MDR.** Estes canais estendem a classe **Channel** que permite que estes sejam executados como uma *thread*.

```
public ControlChannel controlRoom;
public BackupChannel backupCh;
public RestoreChannel restoreCh;
```

A classe **Peer** possui um objeto do tipo **Cloud** e no seu método **main** os canais são inicializados.

```
cloud.createControlRoom(args[3], args[4]);
cloud.createBackupChannel(args[5], args[6]);
cloud.createRestoreChannel(args[7], args[8]);

cloud.activateChannels();
```

Desta forma, quando um **Peer** é inicializado, os canais também o são.

Objetivando um melhor acesso aos recursos em comum que as várias *threads* usam, utilizámos um mecanismo bastante útil fornecido pela linguagem Java: a **sincronização.** Este mecanismo permite que de todas as *threads* que têm acesso a um recurso apenas uma de cada vez pode aceder a esse recurso. Para ativar este mecanismo foi apenas necessário a inserção do atributo ***synchronized*** nas funções pretendidas, sendo elas os protocolos **backup, delete, restore, reclaim, state** e as funções responsáveis pelo envio e receção de mensagens por parte dos canais *multicast*.

Abordando a implementação dos protocolos em si, cada protocolo está definido na classe **Peer,** que estende a interface **RMI**.

Por exemplo, no protocolo **backup**, o ficheiro é dividido em **chunks** de **64000 bytes**, sendo estes posteriormente usados, através de uma **Thread Pool** de tamanho fixo, para criar mensagens que são enviadas pelos canais *multicast* **MC** e **MDB** de forma faseada e de acordo com o exigido no protocolo, esperando um segundo com o fim de averiguar se o *replication degree* foi correspondido, caso contrário aumenta o tempo de espera e volta a averiguar, até um máximo de 5 vezes.

O protocolo em si encontra-se no diretório "/protocols/" , sendo que um objeto deste protocolo é criado na função definida no **Peer** e este protocolo é executado em simultâneo com os **peers** que também estejam a correr.

```
            ExecutorService WORKER_THREAD_POOL;
            List<Callable<Object>> callables = new ArrayList<Callable<Object>>();
            List<Future<Object>> futures = null;

                                                    ...

futures = WORKER_THREAD_POOL.invokeAll(callables);

WORKER_THREAD_POOL.shutdown();

if(WORKER_THREAD_POOL.awaitTermination(35, TimeUnit.SECONDS)) {
        for(Future<Object> p: futures) {
                if(!(boolean)p.get()) {
                        System.out.println("One of the chunks could not be stored.");

                        delete(file_path); //order successfully stored chunks to be deleted

                        return "Backup failed preparing chunks";
    }
        }
}
```

```
public Object call() throws Exception {

        int numberOfTries = 0;
        int multiplier = 1;

        if(storage.insertAwaitingStoreFile(builtChunkMessage.getHeader())) {
                while(numberOfTries < Constants.MAX_PUTCHUNK_MESSAGES) {
                        storage.sendBackupMessage(builtChunkMessage);
                        System.out.println("Trying " + multiplier + " seconds");

                        Thread.sleep(1000 * multiplier);

                        if(storage.wasTheRepDegreeMatched(objective)) {
                                storage.addChunk(objective);
                                System.out.println(objective.getChunkNumber() + " replication degree matched!");
                                return true;
                        }

                        multiplier *= 2;
                        numberOfTries++;
                }
        }

        System.out.println("ERROR: " + objective.getChunkId() + " replication degree not matched!");
        return false;
}
```

Foi também criada a função ***waitRandomTime()*** que cria um número aleatório **result** entre 0 e 400ms, chamando-o em ***Thread.sleep(result)***. Como o uso desta função pode gerar um aumento do número de *threads* coexistentes, a realização de um *sleep* no *thread* com um número aleatório permite que o *thread* onde é chamado esta função não fique bloqueado. Esta função é chamada no protocolo Backup e Restore.

```
public void waitRandomTime() throws InterruptedException {
        Random r = new Random();
        int low = 1;
        int high = 401;
        int result = r.nextInt(high-low) + low;

        Thread.sleep(result);
}
```