Distributed Backup Service

Mestrado Integrado em Engenharia Informática e Computação

Sistemas Distribuídos

Grupo T6G06 :
Alexandra Isabel Vieites Mendes - 201604741
Diogo Filipe da Silva Yaguas - up201606165

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

14 de Abril de 2019

Execução Simultânea de Protocolos

De forma a que seja possível a execução simultânea de protocolos, utilizámos várias estratégias e mecanismos disponíveis na linguagem Java.

As estruturas de dados que usámos para guardar as informações pretendidas foram importantes para este fim. Em vez de usarmos HashMaps ou HashTables, foi nos recomendado o uso de **ConcurrentHashMaps** que são mais seguros quando usados em contextos onde existem várias threads e a sincronização e escalabilidade do programa são importantes.

Começámos por implementar na Classe **Peer** um atributo para cada tipo de canal : **ChannelControl MC**, **ChannelRestore MDR**, **ChannelBackup MDB**, que são inicializados quando um peer é inicializado e para os quais criamos uma thread na função **main**. Garantindo assim que só há uma thread por canal *multicast*.

Para além disso, tirámos partido das funcionalidades do **ScheduledThreadPoolExecutor**, que nos permite implementar um *timeout* que não bloqueie a thread que esteja a correr no momento pois este método admite um tempo limite em que não corre nenhuma outra thread até este terminar.

```
1     private static ChannelBackup MDB;
2     private static ChannelControl MC;
3     private static ChannelRestore MDR;
4     private static ScheduledThreadPoolExecutor exec;
5
6       private Peer() {
7
8         [...]
9
10        exec = (ScheduledThreadPoolExecutor) Executors.
              newScheduledThreadPool(250);
11
12        // communication channels initialization
13        MC = new ChannelControl(MCAddress, MCPort);
14        MDB = new ChannelBackup(MDBAddress, MDBPort);
15        MDR = new ChannelRestore(MDRAddress, MDRPort);
16    }
17
18
19     public static void main(String[] args) {
20
21       [...]
22
23        exec.execute(MC);
24        exec.execute(MDB);
25        exec.execute(MDR);
26
27    }
```

A *thread* responsável pelo processamento das mensagens é a classe **MessangeHandler** e a responsável pelo envio das mensagens de confirmação é a classe **MessageForwarder**.

Adicionalmente, fizemos uso de dois mecanismos inerentes à linguagem Java : a **sincronização** e a **serialização**. A primeira porque nos permite que recursos partilhados sejam apenas acedidos por uma thread pois sincronização em Java baseia-se na capacidade de controlar o acesso de múltiplas threads a recursos

partilhados. Assim, usamos nas funções que achámos pertinentes o método **synchronized**.

```
1       /**
2        * Decrease replication degree
3        *
4        * @param fileID  File ID
5        * @param ChunkNr Chunk Number
6        */
7       synchronized void decreaseRepDegree(String fileID, int ChunkNr)
             {
8
9           String key = fileID + '_' + ChunkNr;
10          int total = this.storedReps.get(key) - 1;
11          this.storedReps.replace(key, total);
12      }
13
14      /**
15       * Increase replication degree
16       *
17       * @param fileID  File ID
18       * @param ChunkNr Chunk Number
19       */
20      synchronized void increaseRepDegree(String fileID, int ChunkNr)
             {
21
22          String key = fileID + '_' + ChunkNr;
23          int total = this.storedReps.get(key) + 1;
24          this.storedReps.replace(key, total);
25      }
```

Por último, a **serialização** é um conceito em Java que consiste em representar objetos como sequências de bytes guardando todas as informações acerca do mesmo. Depois de serializado e escrito num ficheiro, o objeto pode ser lido a partir daí e os dados podem ser usados para guardar o objeto em memória. Isto permitiu-nos portanto resguardar as estruturas de dados que tínhamos com as suas informações, guardando assim o estado atual da aplicação. Este método foi utilizado na classe **Storage**, que funciona como a base de dados de cada peer, que guarda e contém a informação disponível em cada um destes. É criada para cada peer na função *main*.

```
1   /* Peer storage, where the peer's state is kept */
2
3   class Storage implements java.io.Serializable {
4
5       /* Peer's files that are backup */
6
7       private HashMap<String, File> files;
8       private HashMap<String, Integer> desiredReplicationDegree;
9       private HashMap<String, ArrayList<Chunk>> fileChunks;
10
11      private ArrayList<Chunk> storedChunks;
12      private ConcurrentHashMap<String, Integer> storedReps;
13
14      [...]
15
16      };
```