SDIS 2019/2020

Project 1 -- Distributed Backup Service

Mestrado Integrado em Engenharia Informática e Computação

Grupo T7G06

Henrique Ferreira - up201605003
Hugo Fernandes - up201909576

Introdução

Este relatório tem como objetivo explicar o design e a simultaneidade dos protocolos desenvolvidos e da comunicação entre os *Peers* pelos canais *Multicast*. Como os protocolos DELETE e STATE são de certa forma simples na parte de design e simultaneidade, estes não foram considerados no desenvolvimento deste relatório.

Message Management

A receção de mensagens é feita pela leitura contínua da chegada de *packets* que possuem a mensagem em questão. Na obtenção de cada *packet* é criado um *thread MessageManagement* responsável por lidar com a mensagem tendo em conta o tipo desta podendo ser: PUTCHUNK, GETCHUNK, CHUNK, STORED, DELETE ou REMOVED.

```
public void run() {
   byte[] rbuf;
   while (true) {
       try {
           rbuf = new byte[MAX_MESSAGE_SIZE];
           DatagramPacket packet = new DatagramPacket(rbuf,
rbuf.length);
           this.socket.receive(packet);
           MessageManagement messageManagement = new
MessageManagement(packet);
           new Thread(messageManagement).start();
       } catch (Exception e) {
           e.printStackTrace();
       }
   }
}
```

Figura 1 - Função *run* dos canais

O envio de mensagens é feita através do *Channel* com escolha do do tipo de canal.

Protocols

Backup Protocol

Este protocolo cria uma *thread ResponseManagement* por cada *chunk* do ficheiro especificado como argumento na chamada desta função. Desta forma as mensagens enviadas por cada chunk serão geridas independentemente umas das outras.

```
@Override
public void run() {
   int i = 100;
   int chunkTimesInSystem = 0;
   do{
       try {
           Thread.sleep(i);
       } catch (InterruptedException e) {
           e.printStackTrace();
       }
       chunkTimesInSystem =
Peer.getStorage().getReplicationsInSystem(this.tag);
       if(chunkTimesInSystem >= repDegree ){
           return;
       }
       String s = new String(msg);
       Peer.getChannel(Channel.ChannelType.MDB).sendMessage(msg);

       i = i*2;
       numberOfAtemtps++;
   }while(chunkTimesInSystem < repDegree || numberOfAtemtps != 5);
}
```

Figura 2 - **Função *run* na classe ResponseManagement**

Restore Protocol

Este protocolo envia uma mensagem GETCHUNK por cada chunk do ficheiro especificado como argumento na chamada deste. Cada chunk lido é guardado na storage. A criação do ficheiro final é efetuada na classe *Util* na função *createFile* com uso dos dados guardados na variável *receivedChunks* em *Storage*.

Reclaim Protocol

Este protocolo procede com a verificação de espaço na *Storage* tendo em conta o valor passado como argumento. Os Peers poderão receber a mensagem REMOVED procedendo com threads *ReponseManagement* (**Figura 2**) para repor os seus chunks a partir de mensagens PUTCHUNKS tal como o protocolo de *Backup*

Additional Info

Cada protocolo é chamado através da interface *RMIInterface* na classe *TestApp* e esta por sua vez chama as funções dos protocolos na classe *Peer* através desta interface.

A geração de um identificador do ficheiro é efetuada na classe *Util* a partir do *SHA256* convertendo esse identificador numa *string* e retornando-o.

Todas as mensagens processadas são escritas na linha de comandos do *Peer* que as processa ignorando os dados dos chunks nas mensagens que os possuem.

Existe um protocolo extra chamado *Save Protocol* que está responsável por guardar a informação do objeto *storage* do *Peer* num ficheiro dentro do diretório deste. Inicialmente tentámos fazer disto uma *thread* que procedia com o save na invocação do comando *ctrl+c* mas esta situação causava que o objeto storage fosse anulado antes que a função *saveStorage* chegasse ao fim. Desta forma, sempre que quisermos guardar a informação da *storage* do *Peer* temos de chamar o protocolo *Save.*