U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Backup Service

Sistemas Distribuídos
MIEIC 3º ANO

TURMA 7 - GRUPO 11

Diogo Almeida, up201806630
Miguel Silva, up201806388

Índice

1 Sumário

Este trabalho foi desenvolvido no âmbito da unidade curricular de Sistemas Distribuídos e teve como objetivo desenvolver e implementar um sistema distribuído de *backup* de dados. O trabalho foi concluído com sucesso, sendo a aplicação desenvolvida capaz de executar todos os protocolos referidos no enunciado - backup, restore, delete e reclaim.

Este relatório tem então como propósito explicar os *enhancements* desenvolvidos bem como descrever o design escolhido que permite a execução simultânea dos protocolos e explicar a sua implementação.

2 *Backup Enhancement*

Em relação à melhoria do protocolo de *Backup*, o seu objetivo foi garantir que o *replication degree* desejado corresponda ao *replication degree* real, o que consequentemente leva a uma poupança de memória.

Assim, por forma a manter um registo daquilo que é recebido/enviado através dos *multicast channels* para auxílio na análise e gestão de informação foi implementada a classe ***PeerStorage***. Nesta classe é usado um *ConcurrentHashMap*, ***chunkMap***, o qual guarda o número de vezes que a mensagem STORED foi recebida em relação a um *chunk* específico, ou seja, o número de ocurrências de um *chunk* nos *Peers*. Esta estrutura de dados tem como chave uma *string* de combinação do seu *fileID* com o seu *chunkNo* e o seu valor contém uma lista com os *IDs* dos *peers* onde esse *chunk* está guardado (representando o comprimento desta lista o grau de replicação real).

Deste modo, sempre que é recebida uma mensagem do tipo PUTCHUNK, o *peer* espera um tempo aleatório entre 0 a 400 ms e, após esse tempo, consulta a tabela ***chunkMap*** verificando se o grau de replicação atual do *chunk* é maior ou igual ao desejado. Caso o seja, a escrita é cancelada e o *chunk* descartado. Se, pelo contrário, o grau de replicação for menor ao desejado o *peer* atualiza a estrutura de dados, escreve o *file* e envia uma mensagem do tipo STORED.

Esta solução provou-se eficiente, reduzindo a probabilidade de o grau de replicação real de um *chunk* ser superior ao desejado. Este último caso só ocorre caso dois *peers* acedam aos seus ***chunkMaps*** ao mesmo tempo, sem que um deles tenha primeiro enviado uma mensagem do tipo STORED.

```
1   //..
2   Random random = new Random();
3       try {
4           Thread.sleep(random.nextInt(401));
5       } catch (InterruptedException e) {
6           e.printStackTrace();
7       }
8
9       //Verify if this chunk already has the desired replication degree -
    ↪ backup protocol enhancement
10      if (Peer.peerStorage.timesStored(chunkKey) >= castMsg.getRepDegree() &&
    ↪ castMsg.getProtocolVersion().equals("1.1") && Peer.version.equals("1.1"))
    ↪ {
11          return;
12      }
13  //..
```

3 *Delete Enhancement*

Em relação à melhoria do protocolo de *Delete* este teve como objetivo garantir que, mesmo que um *peer* que contém *chunks* de um determinado ficheiro a apagar não esteja *online*, o espaço alocado para este *chunk* não seja perdido, mas sim recuperado quando o *peer* em questão se conectar.

Para implementar este *enhancement*, foi criado um novo tipo de mensagem: *ONLINE*. Na classe ***PeerStorage*** - já referida anteriormente - é usado um *ConcurrentHashMap*, ***filesDeleted***, o qual tem como chave uma *string fileID* e o seu valor é uma instância da classe *FileHandler* referente a um ficheiro que já foi apagado.

Deste modo, sempre que um *Peer* se conecta, este envia uma mensagem do tipo *ONLINE* através do *multicast channel* de controlo para todos os outros peers, com a estrutura: <Version>ONLINE<SenderId><CRLF><CRLF>. Após receber esta mensagem, cada *Initiator Peer* irá iterar sobre a sua estrutura de dados ***filesDeleted***, enviado uma mensagem do tipo DELETE por cada ficheiro que lá se encontra. Desta maneira, todos os *peers* que se encontravam *offline* até ao momento e que enviaram a mensagem do tipo ONLINE irão executar o sub-protoco *delete* por cada uma das mensagens *DELETE* recebidas.

Esta solução não é a mais eficiente pois mesmo que um *peer* já tenha realizado o sub-protoco de *delete* para um determinado ficheiro, este volta a receber a mensagem de *DELETE* para esse ficheiro sempre que um *peer* se conecta (e também para ficheiros cujos *chunks* pode nem conter). No entanto, implenta com sucesso o *enhancement* pedido no enunciado, usando apenas uma mensagem extra.

```
1  public class OnlineTask extends Task{
2
3      public OnlineTask(Message message){
4          super(message);
5      }
6
7      @Override
8      //start - runs an Online Task: sends the peer's deleted files in DELETE
       ↪ messages
9      public void start() {
10         if (!message.getProtocolVersion().equals(Peer.version)){
11             return;
12         }
13         OnlineMessage castMessage = (OnlineMessage) message;
14         System.out.println("[Peer " + Peer.id + "] Received ONLINE from " +
       ↪ castMessage.getSenderID() + " - A surprise to be sure, but a welcome one.
       ↪ ");
15
16         for (FileHandler file: Peer.peerStorage.getFilesDeleted().values()){
17             DeleteMessage msg = new DeleteMessage(Peer.version, Peer.id, file.
       ↪ getFileID());
18             Peer.threadPool.submit(new ControlChannelSender(msg));
19         }
20     }
21 }
```

4 Execução simultânea de protocolos

Relativamente ao design implementado, este permite a execução simultânea de vários protocolos através do uso de *threads*. Cada *peer* utiliza uma *FixedThreadPool* com 64 *threads* para executar as várias tarefas. Se todas as *threads* estiverem a ser utilizadas em determinado momento, a próxima tarefa é colocada numa fila de espera até alguma *thread* voltar a ficar disponível.

Para iniciar os 3 canais de comunicação: **MC**, **MDB** e **MDR**, são instanciados 3 objetos da classe ***MulticastChannel***, que implementa a interface ***Runnable***, em cada *peer*. Estes 3 canais são depois postos a correr em simultâneo em 3 *threads* distintas (que não pertencem à *thread pool* referida anteriormente), ficando em *loop* infinito a receber mensagens.

```
    //..
    public static MulticastChannel MC;
    public static MulticastChannel MDB;
    public static MulticastChannel MDR;
    public static ExecutorService threadPool;
    public static PeerStorage peerStorage = new PeerStorage();

    private Peer(String[] args) throws IOException {
        //...
        MC = new MulticastChannel("Multicast Control", args[3], Integer.parseInt(
    ↪ args[4]));
        MDB = new MulticastChannel("Multicast Data Backup", args[5], Integer.
    ↪ parseInt(args[6]));
        MDR = new MulticastChannel("Multicast Data Recovery",args[7], Integer.
    ↪ parseInt(args[8]));
        threadPool = Executors.newFixedThreadPool(64);
        loadSerializedStorage();
    }
    //..
```

Quando é necessário enviar uma mensagem para qualquer um dos canais **MC**, **MDB** ou **MDR** é criada uma *thread* ***ControlChannelSender***, ***BackupChannelSender*** ou ***RestoreChannelSender*** respetivamente, de modo a que o envio das mensagens não interrompa a execução de outras tarefas, principalmente no caso do envio de mensagens PUTCHUNK, em que poderão haver várias tentativas de envio.

```
public void backup(String filePath, int repDegree) throws RemoteException {
            //..
            while ((bytesRead = bufferedIS.read(buffer)) > 0) {
                //..
                Chunk newChunk = new Chunk(file.getFileID(), count, repDegree,
    ↪ sizedArray);
                Message msg = new PutchunkMessage(version, id, file.getFileID(),
    ↪ count, repDegree, sizedArray);
                threadPool.submit(new BackupChannelSender(msg));
                newChunk.removeBody();
                file.addChunk(newChunk);
                count++;
                Thread.sleep(50);
            }
```

```
13            //..
```

Ao receber uma mensagem, o *peer* verifica inicialmente se esta foi enviada por si próprio. Se não for o caso, então esta é imediatamente enviada para uma nova *thread*: ***Messenger***, de modo a não interromper a receção de mensagens. É nesta *thread* que é realizada a interpretação e o processamento da mensagem, resultando na criação de um objeto da classe ***Message*** e na execução da tarefa associada.

```
1  public void run() {
2      System.out.println("[Peer " + Peer.id + "] " + this.name + " Channel: Ready
       ↪ to receive");
3      while (true) {
4          byte[] buffer = new byte[65000];
5          DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
6          try{
7              this.receive(packet);
8          } catch (Exception e){
9              e.printStackTrace();
10         }
11         byte[] sizedArray = Arrays.copyOf(buffer, packet.getLength());
12         if (this.needsMessage(sizedArray)) {
13             Peer.threadPool.submit(new Messenger(sizedArray));
14         }
15     }
16 }
```

Para a escolha das estruturas de dados mais adequadas, decidimos usar *ConcurrentHashMaps* ao invés de *HashMaps* dado estes serem mais apropriados para ambientes *multi-threaded*, como é o caso deste projeto. Isto acontece pois, em comparação com um *HashMap*, um *ConcurrentHashMap* é sincronizado (*Thread Safe*), mais seguro, mais escalável e possui um melhor desempenho.

Finalmente, a classe ***Peer*** contém também a classe ***PeerStorage***, que guarda todos os *ConcurrentHashMaps* utilizados e que implementa a interface ***Serializable***. A **serialização** é uma técnica que permite transformar o estado de um objeto numa sequência de bytes. Depois de um objeto ser serializado este pode ser guardado num ficheiro, sendo possível, mais tarde, ser desserializado para recriar o objeto em memória. No programa, os métodos do ***Peer*** que tratam destes processos são: ***saveSerializedStorage***, que é chamado através de um *ShutdownHook*, que corre numa *thread* paralela, guardando o estado do Peer sempre que encerra, e ***loadSerializedStorage***, o qual é chamado no ínicio da execução de cada *Peer*, de modo a tentar recuperar o estado anterior deste, se existir.

```
1      public class PeerStorage implements Serializable {
2          private long capacity = 1000000000;
3          private final ConcurrentHashMap<String, List<Integer>> chunkMap = new
           ↪ ConcurrentHashMap<>();
4          private final ConcurrentHashMap<String, FileHandler> filesBackedUp = new
           ↪ ConcurrentHashMap<>();
5          private final ConcurrentHashMap<String, Chunk> chunksStored = new
           ↪ ConcurrentHashMap<>();
6          private final ConcurrentHashMap<String, Integer> chunkRestoreMap = new
           ↪ ConcurrentHashMap<>();
```

```
7         private final ConcurrentHashMap<String, Integer> putchunkMap = new
      ↪ ConcurrentHashMap<>();
8         private final ConcurrentHashMap<String, Chunk> restoredFileChunks = new
      ↪ ConcurrentHashMap<>();
9         private final ConcurrentHashMap<String, FileHandler> filesDeleted = new
      ↪ ConcurrentHashMap<>();
10        //..
```

```
1  public class Peer implements RemoteObject {
2      //..
3      private Peer(String[] args) throws IOException {
4          //..
5          loadSerializedStorage();
6      }
7
8      public static void main(String[] args) {
9
10             Runtime.getRuntime().addShutdownHook(new Thread(() -> {
11                 try {
12                     Thread.sleep(200);
13                     System.out.println("[Peer " + Peer.id + "] Shutting Down...")
       ↪ ;
14                     saveSerializedStorage();
15                 } catch (InterruptedException e) {
16                     Thread.currentThread().interrupt();
17                     e.printStackTrace();
18                 }
19             }));
20             //..
```