FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Sistemas Distribuídos | Projeto 2

Distributed Backup Service for the Internet

MIEIC 2020/21
2 de junho de 2021

T1 G22

Francisco José Paiva Gonçalves | **201704790**
Luís André Santos Correia Assunção | **201806140**
Pedro Daniel Fernandes Ferreira | **201806506**
Pedro Varandas da Costa Azevedo da Ponte | **201809694**

1. Overview

1.1 Features

O projeto foi desenvolvido usando **Chord** para *data lookup protocols*, *scalability*, *fault-tolerance* e *synchronization*. Para garantir a segurança nos canais de comunicação entre nós do Chord, recorremos ao uso de **JSSE**. Usámos também ***Thread*** e ***ScheduledThreadPoolExecutor*** para permitir e lidar com a concorrência, sendo possível correr vários protocolos ao mesmo tempo.

1.2 Supported Operations

1.2.1 Backup

A operação **backup** permite a um peer do sistema chamar outros *peers* para guardar um ficheiro indicado, de acordo com o *replication degree* desejado. Para concretizar isto, o *backup* divide o ficheiro em *chunks* de 64KB, associando a cada um dos *chunks* IDs na rede do *Chord*, guardando-o na diretoria **backup** do *peer* que é sucessor desse ID e nos *peers* seguintes, de acordo com o *replication degree* (não contando com o peer que faz o pedido).
Nota: o caminho para a diretoria é ***src/build/peer*/backup***

1.2.2 Restore

A operação **restore** vai buscar os *chunks* criados na operação de **backup** para recompor o ficheiro. O **initiator peer** indica o nome do ficheiro que quer restaurar e assim o programa vai procurar os *chunks* e enviar mensagens para os reaver. Após os *chunks* estarem todos obtidos são anexados por ordem, reconstruindo o ficheiro e guardando na diretoria **restore** do *initiator peer*.
Nota: o caminho para a diretoria é ***src/build/peer*/restore***

1.2.3 Delete

A operação **delete** permite a um *peer* fazer um pedido para apagar todas as ocorrências de *chunks* de um ficheiro (especificado pelo mesmo) da rede do *Chord*. Para este efeito, o *initiator peer* envia um pedido para apagar os *chunks* desse ficheiro a todos os *peers* que tenham feito *backup* de algum dos *chunks* pertencentes a este ficheiro.

1.2.4 Reclaim

A operação **reclaim** permite gerir o espaço ocupado por um *peer*, enviando como argumento o novo espaço disponível para esse *peer*. Caso o espaço novo seja menor que o espaço em utilização, o protocolo vai tratar de apagar *chunks*, um a um, até que o espaço utilizado seja menor que o novo espaço máximo. Caso seja usado o reclaim com 0, todos os chunks são apagados desse peer.

1.2.5 State

A operação **state** permite ao utilizador ver as informações sobre os ficheiros que determinado *peer* fez *backup*, os *chunks* que esse *peer* tem guardados assim como a informação sobre a capacidade de armazenamento total em KB e o espaço total ocupado com chunks replicados.

SDIS P2 | T1 G22
Francisco Gonçalves
Luís André Assunção
Pedro Ferreira
Pedro Ponte

1.2.6 Chord

A operação **chord** permite ver o estado atual do nó do anel do *Chord* associado a um determinado *peer*. Através deste comando é possível obter informações sobre o id, o endereço e a porta em que o nó, bem como o seu sucessor e o seu predecessor, estão a correr. É também mostrado o estado da *finger table*.

1.3 Compile and Execute

Para compilar e executar o programa criámos uma makefile que tem entradas para todas as operações necessárias, entre elas, compilar, limpar, iniciar RMI, terminar RMI, correr os protocolos (chamando scripts para cada uma das operações na pasta scripts/). Os ficheiros serão guardados dentro da diretoria ***src/build/peer**** onde * é o nº do peer correspondente.

Para mais detalhes sobre compilação e execução do projeto, as instruções estão bem documentadas no ficheiro **README.md**. Uma maneira breve de correr o projeto seria:

```
make clean  # cleanup build/ directory (optional/just in case)
make        # compile all java files inside src/
make kill   # kill rmi (optional/just in case)
make rmi    # run rmiregistry
make peer1  # runs peer1 with default arguments (try on new shell window)
make peer2  # runs peer2 with default arguments (try on new shell window)
make peer3  # runs peer3 with default arguments (try on new shell window)
make backup # perform operation (backup/restore/delete/reclaim/state/chord)
```

2. Protocols

Na nossa aplicação, os *peers* comunicam entre si através do protocolo **TCP** com **JSSE**, para que as comunicações sejam mais seguras. A comunicação entre o **Client** e os *peers* é feita utilizando **RMI**, tal como no primeiro projeto.

2.1 Messages Format

1. <Version> FINDSUCC <nodeId> <address> <port> <isFinger> (<fingerPos>)* <CRLF><CRLF>
2. <Version> SUCCFOUND <nodeId> <address> <port> <isFinger> (<fingerPos>)* <CRLF><CRLF>
3. <Version> FINDPRED <nodeId> <address> <port> <CRLF><CRLF>
4. <Version> PREDFOUND <nodeId> <address> <port> <CRLF><CRLF>
5. <Version> NOTIFY <nodeId> <address> <port> <CRLF><CRLF>
6. <Version> PREDALIVE <nodeId> <address> <port> <CRLF><CRLF>
7. <Version> ALIVE <nodeId> <address> <port> <CRLF><CRLF>
8. <Version> PUTCHUNK <SenderId> <Address> <Port> <FileId> <ChunkNo> <ReplicationDeg> <CRLF><CRLF> <Body>
9. <Version> STORED <SenderId> <Address> <Port> <FileId> <ChunkNo> <CRLF><CRLF>
10. <Version> REMOVED <SenderId> <Address> <Port> <FileId> <ChunkNo> <CRLF><CRLF>
11. <Version> DELETE <SenderId> <Address> <Port> <FileId> <CRLF><CRLF>
12. <Version> GETCHUNK <SenderId> <Address> <Port> <FileId> <ChunkNo> <CRLF><CRLF>
13. <Version> CHUNK <SenderId> <Address> <Port> <FileId> <ChunkNo> <CRLF><CRLF> <Body>

2.2 Backup

Quando um *peer* (classe **Peer**) recebe um pedido para fazer *backup* de um ficheiro, começa por verificar se o ficheiro existe mesmo e se anteriormente já não fez *backup* desse ficheiro, retornando caso alguma destas condições não seja cumprida. De seguida, o ficheiro original é dividido em *chunks*, cada um com um tamanho máximo igual a 64KB. Tendo os *chunks*, agora é necessário enviá-los. Para isso, primeiro constrói-se a mensagem do tipo PUTCHUNK, através de uma chamada ao método ***constructPutChunkMessage()*** da classe **MessageBuilder**. O passo seguinte é enviar esta mensagem para que os nós possam guardar os *chunks*. Para enviar as mensagens, recorremos à *finger table* do nó associado ao *peer* que faz *backup*, enviando cada mensagem de PUTCHUNK para tantos *peers* quanto a replicação desejada.

```
222         ArrayList<Chunk> fileChunks = fileManager.getFileChunks();
223
224         for(int i = 0; i < fileChunks.size(); i++) {
225             // <Version> PUTCHUNK <SenderId> <FileId> <ChunkNo> <ReplicationDeg> <CRLF><CRLF><Body>
226             try {
227                 Chunk chunk = fileChunks.get(i);
228
229                 MessageBuilder messageBuilder = new MessageBuilder();
230                 byte[] message = messageBuilder.constructPutChunkMessage(this, fileIDNew, chunk);
231
232                 if(!(storage.hasRegisterStore(fileManager.getFileID(), chunk.getChunkNo()))) {
233                     storage.createRegisterToStore(fileManager.getFileID(), chunk.getChunkNo());
234                 }
235
236                 for(int j = 0; j < replication; j++) {
237                     NodeInfo receiver = chordNode.getFingerTable().get(j);
238
239                     // send threads
240                     threadExec.execute(new ThreadSendMessages(receiver.getIp(), receiver.getPort(), message));
241                     threadExec.schedule(new ThreadCountStored(replication, fileManager.getFileID(), i, message, receiver), 2, TimeUnit.SECONDS);
242                 }
243             }
```

Peer | Line 222

SDIS P2 | T1 G22
Francisco Gonçalves
Luís André Assunção
Pedro Ferreira
Pedro Ponte

As mensagens são enviadas criando-se uma *thread* onde irá correr a classe **ThreadSendMessage**, responsável por enviar a mensagem para o destinatário utilizando o canal TCP anteriormente criado. É também agendada para começar a ser executada 2 segundos após uma nova *thread* onde vai correr a **ThreadCountStored**.

Um *peer*, ao receber uma mensagem do tipo PUTCHUNK (classe **PutChunkMessageThread**), vai verificar se não foi ele mesmo a enviar a mensagem e, caso seja, vai descartar a mensagem. De seguida, verifica se já guardou o *chunk* que está a ser enviado.

Caso já o tenha feito, então vai responder ao *initiator peer* com uma mensagem do tipo STORED, para garantir que o outro sabe que ele já fez *backup* do *chunk* e de seguida encaminha a mensagem de PUTCHUNK recebida para o seu sucessor, obtido através da sua *finger table*.

Deste modo, a mensagem PUTCHUNK vai sendo sempre reencaminhada até que algum nó guarde o *chunk* ou então até chegar ao próprio *peer* que iniciou o *backup*.

```
34  @Override
35  public void run() {
36      if(this.address.equals(Peer.getChordNode().getNodeInfo().getIp()) && this.port == Peer.getChordNode().getNodeInfo().getPort()) {
37          System.out.println("Same peer as initiator. Can't backup chunk.");
38          return;
39      }
40
41      System.out.println("RECEIVED: " + this.protocolVersion + " PUTCHUNK " + this.senderId + " " + this.address + " " + this.port + " " + this.fileId + " " + this.chunkNo + " " + this.replication_d
42      System.out.println();
43
44      //check if the peer already has stored this chunk
45      // in this case, forwards the message to its successor
46      if(Peer.getStorage().hasChunk(this.fileId, this.chunkNo) == true) {
47          System.out.println("Already has chunk");
48          System.out.println();
49
50          Random r = new Random();
51          int low = 0;
52          int high = 400;
53          int result = r.nextInt(high-low) + low;
54
55          MessageBuilder builder = new MessageBuilder();
56          byte[] toSend = builder.constructStoredMessage(Peer.getChordNode().getNodeInfo().getIp(), Peer.getChordNode().getNodeInfo().getPort(), this.fileId, this.chunkNo);
57          Peer.getThreadExec().schedule(new ThreadSendMessages(this.address, this.port, toSend), result, TimeUnit.MILLISECONDS);
58
59          // necessario enviar mensagem de stored?
60
61          MessageBuilder messageBuilder = new MessageBuilder();
62          byte[] message = messageBuilder.constructPutChunkMessage(this.address, this.port, this.fileId, this.chunkNo, this.replication_degree, this.body);
63
64          NodeInfo receiver = Peer.getChordNode().getFingerTable().get(0);
65
66          Peer.getThreadExec().execute(new ThreadSendMessages(receiver.getIp(), receiver.getPort(), message));
67
68          return;
69      }
```

PutChunkMessageThread | Line 34

Caso o *peer* ainda não tenha guardado o chunk, vai verificar se tem espaço suficiente para o fazer. Caso não tenha, encaminha a mensagem PUTCHUNK para o seu sucessor.

```
71      // checks if the peer has free space to save the chunk
72      if(!(Peer.getStorage().checkIfHasSpace(this.body.length))) {
73          System.out.println("Doesn't have space to store chunk " + this.chunkNo);
74          System.out.println();
75
76          MessageBuilder messageBuilder = new MessageBuilder();
77          byte[] message = messageBuilder.constructPutChunkMessage(this.address, this.port, this.fileId, this.chunkNo, this.replication_degree, this.body);
78
79          NodeInfo receiver = Peer.getChordNode().getFingerTable().get(0);
80
81          Peer.getThreadExec().execute(new ThreadSendMessages(receiver.getIp(), receiver.getPort(), message));
82
83          return;
84      }
```

PutChunkMessageThread | Line 71

Caso tenha espaço livre para fazer *backup* do *chunk* e ainda não o tenha guardado, vai guardá-lo, enviando para o *initiator peer* uma mensagem STORED.

```
89          Peer.getStorage().addChunk(chunk);
90
91          // create the chunk file in the peer directory
92          String dir = "peer_" + Peer.getPeerId();
93          String backupDir = "peer_" + Peer.getPeerId() + "/" + "backup";
94          String file = "peer_" + Peer.getPeerId() + "/" + "backup" + "/" + this.fileId + "_" + this.chunkNo;
95          File directory = new File(dir);
96          File backupDirectory = new File(backupDir);
97          File f = new File(file);
98
99          try{
100             if (!directory.exists()){
101                 directory.mkdir();
102                 backupDirectory.mkdir();
103                 f.createNewFile();
104             }
105             else {
106                 if (directory.exists()) {
107                     if(backupDirectory.exists()) {
108                         f.createNewFile();
109                     }
110                     else {
111                         backupDirectory.mkdir();
112                         f.createNewFile();
113                     }
114                 }
115             }
116
117             FileOutputStream fos = new FileOutputStream(f);
118             fos.write(this.body);
119             fos.close();
120
121         } catch(Exception e) {
122             System.err.println(e.getMessage());
123             e.printStackTrace();
124         }
125
126
127         Random r = new Random();
128         int low = 0;
129         int high = 400;
130         int result = r.nextInt(high-low) + low;
131
132         MessageBuilder builder = new MessageBuilder();
133         byte[] toSend = builder.constructStoredMessage(Peer.getChordNode().getNodeInfo().getIp(), Peer.getChordNode().getNodeInfo().getPort(), this.fileId, this.chunkNo);
134         Peer.getThreadExec().schedule(new ThreadSendMessages(this.address, this.port, toSend), result, TimeUnit.MILLISECONDS);
135     }
```

PutChunkMessageThread | Line 89

O *initiator peer*, ao receber as mensagens STORED, incrementa o registo de mensagens STORED recebidas associado ao *chunk* em questão, assim como adiciona as informações sobre o *peer* que fez *backup* ao *chunk* à *hash table* **backupChunksDistribution** (classe **StoredMessageThread**).

A classe **ThreadCountStored** agendada para executar quando as mensagens PUTCHUNK são enviadas, vai verificar se a replicação desejada já foi atingida ou se é necessário reenviar as mensagens PUTCHUNK para tentar atingir a replicação que era desejada pelo cliente. Este procedimento é repetido 4 vezes, e caso não se atinja a replicação desejada, então será imprimida no terminal do *peer* uma mensagem a dizê-lo. Caso contrário, será imprimida uma mensagem de sucesso.

```
23      @Override
24      public void run() {
25          ConcurrentHashMap<String, ArrayList<InetSocketAddress>> distribution = Peer.getStorage().getBackupChunksDistribution();
26          String chunkId = this.fileId + "_" + this.chunkNo;
27          int storedReplications = 0;
28          if(distribution.containsKey(chunkId)) {
29              storedReplications = distribution.get(chunkId).size();
30          }
31
32          if(storedReplications < this.replication && this.tries < 4) {
33              // <Version> PUTCHUNK <SenderId> <Address> <Port> <FileId> <ChunkNo> <ReplicationDeg> <CRLF><CRLF> <Body>
34              Peer.getThreadExec().execute(new ThreadSendMessages(this.receiver.getIp(), this.receiver.getPort(), this.message));
35              String[] messageArr = (new String(this.message).toString()).split(" ");
36              System.out.println("SENT: "+ messageArr[0] + " " + messageArr[1] + " " + messageArr[2] + " " + messageArr[3] + " " + messageArr[4] + " " + messageArr[5] +
37              this.time = this.time * 2;
38              this.tries++;
39              Peer.getThreadExec().schedule(this, this.time, TimeUnit.SECONDS);
40          }
41
42          if(this.tries >= 4) {
43              System.out.println("Minimum replication not achieved");
44              if(storedReplications == 0) {
45                  Peer.getStorage().deleteFile(this.fileId);
46              }
47              return;
48          }
49          else if(storedReplications >= this.replication) {
50              System.out.println("Replication completed: " + storedReplications);
51              System.out.println();
52              return;
53          }
54      }
```

ThreadCountStored | Line 23

2.3 Restore

Ao receber um pedido de ***restore*** vindo do **Client**, o *peer* vai inicialmente verificar se alguma vez iniciou o backup de tal ficheiro. Caso tenha efetuado o *backup* desse ficheiro, calcula o número de *chunks* que o ficheiro tem e para cada *chunk* envia uma mensagem GETCHUNK para todos os peers que têm o *chunk* em questão guardado em backup, através de uma chamada ao método ***constructGetChunkMessage()*** da classe **MessageBuilder.** De seguida, cria uma nova *thread* e envia a mensagem.

```
282         ArrayList<Chunk> chunks = file.getFileChunks();
283         for(Chunk chunk : chunks) {
284             try {
285                 MessageBuilder messageBuilder = new MessageBuilder();
286                 byte[] message = messageBuilder.constructGetChunkMessage(peer.getAddress().getHostAddress(), peer.getTcpPort(), fileID, chunk.getChunkNo());
287 
288                 //send delete message
289                 for(ConcurrentHashMap.Entry<String, ArrayList<InetSocketAddress>> set : getStorage().getBackupChunksDistribution().entrySet()) {
290                     if(set.getKey().equals(fileID + "_" + chunk.getChunkNo())) {
291                         for(int j = 0; j<set.getValue().size(); j++)
292                             threadExec.execute(new ThreadSendMessages(set.getValue().get(j).getAddress().getHostAddress(), set.getValue().get(j).getPort(), message));
293                     }
294                 }
295             } catch(Exception e) {
296                 System.err.println("Caught exception while restoring");
297                 e.printStackTrace();
298             }
299         }
```

Peer | Line 282

Quando um *peer* recebe uma mensagem GETCHUNK vai verificar se tem o *chunk* pedido guardado e, se tiver, vai responder para o *peer* inicial com uma mensagem CHUNK que contém no *body* o conteúdo do *chunk* (classe **GetChunkMessageThread**).

```
23      @Override
24      public void run() {
25          String path = "peer_" + Peer.getPeerId() + "/backup/" + this.fileId + "_" + this.chunkNo;
26 
27          System.out.println("RECEIVED: " + this.protocolVersion + " GETCHUNK " + this.senderId + " " + this.address + " " +
28          System.out.println("PATH: " + path);
29 
30          String chunkId = fileId + "_" + chunkNo;
31          ConcurrentHashMap<String, Chunk> chunksStored = Peer.getStorage().getChunksStored();
32 
33          // checks if this peer has the chunk stored
34          if(!(chunksStored.containsKey(chunkId))){
35              System.out.println("Don't have chunk " + chunkNo + " stored");
36              System.out.println();
37              return;
38          }
39 
40          byte[] body = chunksStored.get(chunkId).getChunkMessage(); // chamada do message builder e depois envia
41 
42          MessageBuilder messageBuilder = new MessageBuilder();
43          byte[] message = messageBuilder.constructChunkMessage(
44              Peer.getChordNode().getNodeInfo().getSocketAddress().getAddress().getHostAddress(),
45              Peer.getChordNode().getNodeInfo().getPort(), this.fileId, this.chunkNo,
46              body
47          );
48 
49          Peer.getThreadExec().execute(new ThreadSendMessages(this.address, this.port, message));
50      }
51  }
52 
```

GetChunkMessageThread | Line 23

Quando o *initiator peer* recebe uma mensagem CHUNK, verifica se o chunk recebido já tinha sido enviado por outro *peer*. No caso de não ter sido, então guarda-o. Caso contrário, ignora a mensagem (classe **ChunkMessageThread**).

```
23      @Override
24      public void run() {
25          String chunkID = this.fileId + "_" + this.chunkNo;
26          if(!Peer.getStorage().hasRegisterToRestore(chunkID)) {
27              Peer.getStorage().addChunkToRestore(chunkID, this.body);
28
29          }
30          else {
31              System.out.println("Chunk " + chunkNo + " not requested or already have been restored");
32              System.out.println();
33          }
34      }
```

ChunkMessageThread | Line 31

Depois de enviar as mensagens a pedir os *chunks*, o *initiator peer* criam uma nova *thread* onde vai ser corrida a classe **ManageRestoreThread,** a qual é responsável por tentar juntar os vários *chunks* recebidos de modo a obter o ficheiro que anteriormente se tinha feito *backup*. Dentro desta classe, vão ser executados 10 ciclos intervalados por 1 segundo em que o *peer* vai verificar se já recebeu todos os *chunks* necessários para obter o ficheiro. Se receber os chunks todos previstos então vai guardar o ficheiro, já com os chunks todos concatenados num único ficheiro, numa pasta /restore na raiz da pasta do peer.

```java
@Override
public void run() {
    int chunksNumber = this.fileManager.getFileChunks().size();
    ConcurrentHashMap<String, byte[]> fileChunks;
    int n=0;
    // checks if all file chunks are restored
    while(true) {
        ConcurrentHashMap<String,byte[]> allChunks = Peer.getStorage().getChunksRestored();
        fileChunks = new ConcurrentHashMap<String, byte[]>();

        for(String key : allChunks.keySet()) {
            if((key.split("_")[0]).equals(this.fileManager.getFileID())) {
                fileChunks.put(key, allChunks.get(key));
            }
        }


        if(fileChunks.size() != chunksNumber) {
            if(n >= 10) {
                Peer.getStorage().deleteFileRestored(this.fileManager.getFileID());
                Peer.getStorage().deleteChunksRestored(this.fileManager.getFileID());
                System.out.println("Could not restore all chunks. Exiting...");

                break;
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            n++;
        }
        else {
            System.out.println("All chunks restored, going to create file");
            break;
        }
    }

```

ManageRestoreThread | Line 16

2.4 Delete

Ao receber um pedido de ***delete*** vinda do **Client,** um *peer* vai inicialmente verificar se alguma vez iniciou o *backup* de tal ficheiro. Caso o tenha feito,, envia 5 mensagens DELETE para todos os *peers* que têm algum *chunk* do ficheiro em questão guardado durante o *backup*, através de uma chamada ao método *constructDeleteMessage()* da classe **MessageBuilder**.

Quando um *peer* recebe uma mensagem DELETE, vai verificar se tem algum *chunk* do ficheiro que vai ser apagagado guardado e, para cada um desses chunks, remove-o da estrutura FileStorage do Peer e de seguida apaga o *chunk* do disco, finalizando o protocolo de *delete*.

```java
public DeleteMessageThread(Message message) {
    this.message = message;
    String[] header = this.message.getHeader();
    this.senderId = Integer.parseInt(header[2]);
    this.address = header[3];
    this.port = Integer.parseInt(header[4]);
    this.fileId = header[5];
    this.protocolVersion = header[0];
}

@Override
public void run() {
    System.out.println("RECEIVED: " + this.protocolVersion + " DELETE " + this.senderId + " " + this.address + " " + this.port + " " + this.fileId);

    ConcurrentHashMap<String, Chunk> chunks = Peer.getStorage().getChunksStored();

    for(String key : chunks.keySet()) {
        Chunk chunk = chunks.get(key);
        if(chunk.getFileId().equals(this.fileId)) {
            Peer.getStorage().deleteChunk(key);
            String path = "peer_" + Peer.getPeerId() + "/backup/" + key;
            //System.out.println("PATH: "+ path);
            File filename = new File(path);
            filename.delete();
        }
    }
}
}
```

DeleteMessageThread | Line 23

2.5 Reclaim

Ao receber um pedido de *reclaim*, um *peer* vai verificar se necessita de apagar alguns dos *chunks* que tem guardados ou se com a nova capacidade que lhe foi atribuída ainda consegue manter todos os *chunks* que replicou. Caso seja necessário remover *chunks*, então vai ordenar todos os *chunks* que possui por ordem decrescente de tamanho, começando a remover os maiores em primeiro lugar. Ao apagar um *chunk*, o *peer* envia uma mensagem REMOVED para o *peer* que iniciou o backup do ficheiro a que esses *chunks* estão associados.

```
370     public void reclaim(int maximum_disk_space) {
371         // <Version> REMOVED <SenderId> <FileId> <ChunkNo> <CRLF><CRLF>
372         int max_space = maximum_disk_space * 1000;
373 
374         storage.setCapacity(max_space);
375 
376         int occupiedSpace = storage.getPeerOccupiedSpace();
377 
378         int spaceToFree = occupiedSpace - max_space;
379 
380         if(spaceToFree > 0) {
381             ConcurrentHashMap<String, Chunk> chunksStored = getStorage().getChunksStored();
382             ArrayList<Chunk> chunks = new ArrayList<>();
383 
384             for(String key : chunksStored.keySet()) {
385                 chunks.add(chunksStored.get(key));
386             }
387 
388             // descendant ordered list to start delete biggest chunks first
389             Collections.sort(chunks, Comparator.comparing(Chunk::getSize));
390             Collections.reverse(chunks);
391 
392             for(int i = 0; i < chunks.size(); i++) {
393                 String chunkId = chunks.get(i).getFileId() + "_" + chunks.get(i).getChunkNo();
394                 storage.deleteChunk(chunkId);
395 
396                 MessageBuilder messageBuilder = new MessageBuilder();
397                 byte[] message = messageBuilder.constructRemovedMessage(this, chunks.get(i).getFileId(), chunks.get(i).getChunkNo());
398 
399                 try {
400                     threadExec.execute(new ThreadSendMessages(chunks.get(i).getIp(), chunks.get(i).getPort(), message));
401                 } catch (Exception e) {
402                     System.err.println(e.getMessage());
403                     e.printStackTrace();
404                 }
405 
406                 spaceToFree -= chunks.get(i).getSize();
407 
408                 String name = "peer_" + peerId + "/backup/" + chunkId;
409 
410                 File filename = new File(name);
411 
412                 filename.delete();
413 
414                 if(spaceToFree <= 0) {
415                     return;
416                 }
```

Peer | Line 370

Quando o *initiator peer* recebe a mensagem de REMOVED, vai adaptar os seus registos e de seguida vai verificar se com a remoção desse *chunk* o nível de replicação dele continua a ser igual ou superior ao nível de replicação desejado. Caso isto se verifique, então não é necessário desencadear mais nenhuma ação. Caso contrário, então é necessário o *peer* iniciar um novo processo de replicação do *chunk*, enviando novamente mensagens do tipo PUTCHUNK, originando-se aqui um novo processo de *backup* igual ao que foi acima explicado (classe **RemovedThreadMessage**) .

3. Concurrency design

Para garantir a concorrência entre os diferentes protocolos, de modo a que um *peer* possa lidar com as diversas mensagens que são enviadas e recebidas simultaneamente e de forma a poder enviar e receber diversos *chunks* ao mesmo tempo, recorremos à implementação de *multithreading*. Para atingir a concorrência, recorremos à classe da API do Java ***java.util.concurrent.ScheduledThreadPoolExecutor***, que permite agendar um gestor de "tempo limite", sem usar qualquer *thread* antes que o tempo limite expire. Desta forma, evitamos recorrer ao uso de ***Thread.sleep()*** que pode provocar a existência de demasiadas *threads* ao mesmo tempo, utilizando-se assim demasiados recursos e limitando-se a escalabilidade. Com a utilização desta classe, não necessitamos de nos preocupar com o número de *threads* existente e a correr simultaneamente, uma vez que este é limitado e é gerido pela API do Java.

```
39        threadExec = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(10000);
```

Peer | Line 39

De forma a garantirmos a concorrência entre as várias *threads* no acesso, de forma segura, às tabelas onde são guardadas as informações sobre os ficheiros dos quais foi feito backup por um determinado *peer*, *chunks* guardados, distribuição dos *chunks* pelos vários peers existentes, entre outras, recorremos à classe ***java.util.concurrent.ConcurrentHashMap***.

Quando um *peer* é iniciado (“Peer.java”), criam-se uma *thread* que está associada ao canal TCP que os peers utilizam para enviar e receber mensagens. Neste canal, é feita a receção das mensagens enviadas pelos peers. Quando é recebida uma mensagem (“ChannelController.java”, método ***run()***), cria-se uma nova thread - **ManageReceivedMessages** - que é responsável por processar a mensagem recebida. Desta forma, permite-se que seja possível processar diversas mensagens ao mesmo tempo.

Através da utilização das ***ScheduledThreadPoolExecutor***, conseguimos atingir a concorrência dentro do sistema, permitindo que simultaneamente cada nó possa tratar da manutenção do processo associado ao *Chord* e, simultaneamente, possa estar e executar um protocolo de *Backup*, *Delete*, *Reclaim* ou *Restore*.

4. JSSE

Todos os *peers* comunicam entre si através de *SSL Sockets* para que as comunicações entre eles sejam seguras.

As mensagens trocadas entre *peers* para fazerem *backup*, *restore*, *delete* ou *reclaim*, assim como as mensagens trocadas para garantir a manutenção do *Chord*, utilizam todas *SSLSockets*. Excetuam-se a esta utilização as mensagens enviadas pela aplicação de teste para os *peers*, que recorrem ao RMI, tal como no primeiro projeto.

Quando se inicializa um *peer*, este cria uma *thread* onde vai correr um objeto da classe **ChannelController**. Dentro desta classe, fica a correr até que o *peer* seja desligado, uma função que fica à espera de receber alguma mensagem proveniente de algum dos outros *peers*. Quando recebe uma mensagem, cria uma nova *thread* onde vai correr o **ManageReceivedMessages**, onde se vão desencadear as respetivas ações dependendo do seu conteúdo.

```
95      public void run() {
96          //System.out.println("INSIDE RUN");
97
98          SSLServerSocketFactory ssf = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
99          SSLServerSocket serverSocket;
100
101         try {
102             serverSocket = (SSLServerSocket) ssf.createServerSocket(this.port);
103             //System.out.println("CREATED SERVER");
104
105             while(true) {
106                 SSLSocket socket = (SSLSocket) serverSocket.accept();
107                 ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
108                 Message message = (Message) ois.readObject();
109                 Peer.getThreadExec().execute(new ManageReceivedMessages(message));
110                 //socket.close();
111             }
112         } catch (Exception e) {
113             System.err.println(e.getMessage());
114             e.printStackTrace();
115         }
116     }
```

ChannelController | Line 95

Dentro desta classe existe também uma função que permite ao *peer* enviar mensagens. Assim, quando o *peer* desejar enviar alguma mensagem, basta invocar este método.

```
38      public void sendMessage(byte[] message) {
39          // send request
40          //System.out.println("INSIDE SEND MESSAGE");
41
42          SSLSocketFactory ssf;
43          SSLSocket socket;
44
45          ssf = (SSLSocketFactory) SSLSocketFactory.getDefault();
46
47          int i;
48          for(i = 0; i < message.length; i++) {
49              if(message[i] == 0xD && message[i + 1] == 0xA && message[i + 2] == 0xD && message[i + 3] == 0xA) {
50                  break;
51              }
52          }
53
54          byte[] h = Arrays.copyOfRange(message, 0, i);
55          String[] header = new String(h).split(" ");
56
57          try {
58              socket = (SSLSocket) ssf.createSocket(this.ipAddress, this.port);
59              ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
60              out.flush();
61              Message toSend = new Message(message);
62              out.writeObject(toSend);
63              //System.out.println("SEND MESSAGE"); // sera necessario fazer Thread.sleep()?
64
65              try {
66                  Thread.sleep(1000);
67              } catch(Exception e) {
68                  System.err.println(e.getMessage());
69                  e.printStackTrace();
70              }
71
72              socket.close();
```

ChannelController | Line 38

Quando esta classe é inicializada, é também passada ao sistema a chave criada.

```
public ChannelController(String ipAddress, int port) {
    this.port = port;
    this.ipAddress = ipAddress;

    //set the type of trust store
    System.setProperty("javax.net.ssl.trustStoreType","JKS");

    //set the password with which the truststore is encripted
    System.setProperty("javax.net.ssl.trustStorePassword", "123456");

    //set the name of the trust store containing the server's public key and certificate
    System.setProperty("javax.net.ssl.trustStore", "../../keys/truststore");

    //set the password with which the client keystore is encripted
    System.setProperty("javax.net.ssl.keyStorePassword","123456");

    //set the name of the keystore containing the client's private and public keys
    System.setProperty("javax.net.ssl.keyStore","../../keys/keystore");
}
```

ChannelController | Line 95

5. Scalability

Para que o nosso sistema fosse escalável decidimos recorrer à implementação do *Chord*.
Chord é um protocolo e também um algoritmo *peer-to-peer* escalável, composto por vários nós distribuídos.. Utiliza uma *hash-table* (composta por pares chave-valor), atribuindo chaves aos diferentes *peers* (conhecidos como nós). Cada nó vai guardar os valores para todas as chaves pelas quais é responsável.
Graças a este *design*, os *peers* podem-se juntar livremente à rede, até que se atinja o limite máximo definido. O número máximo de *peers* pode ser alterado mudando-se o valor de **M** dentro da classe **ChordNode**.
A nossa implementação do *Chord* encontra-se distribuída pelos ficheiros *ChordNode.java* e *NodeInfo.java*.
Na classe **ChordNode** encontram-se os diversos métodos que permitem a um *peer* juntar-se a um *chord* já existente ou então criar um novo. É nesta classe que também se faz a manutenção do *chord*, recorrendo aos métodos ***stabilize()***, ***fix_fingers()*** e ***check_predecessor()***. Esta classe guarda também as informações sobre o sucessor e predecessor de cada nó, bem como o estado atual da sua *finger table*.
A classe *NodeInfo* serve para guardar as informações de cada um dos nós, como o seu *id*, *address* e *port* para lhe aceder.
Quando um *peer* é criado, cria um objeto do tipo **ChordNode** e de seguida, dependendo dos argumentos que lhe são passados através da linha de comandos, chama uma de duas funções do **ChordNode**:

- ***create()***, no caso de ser o primeiro nó a ser ativado, criando um novo anel. Nesta situação, a mensagem enviada para iniciar o *peer* é do tipo: “java Peer <protocol_version> <peer_id> <service_access_point> <ip_address> <TCP_port>”.
- ***join()***, no caso de se juntar a um anel já criado anteriormente. Neste caso, é necessário, aquando da criação do *peer*, passar o endereço e a porta de *peer* que se encontre ativo e no anel. Neste caso, a mensagem para criar o *peer* é: “java Peer <protocol_version> <peer_id> <service_access_point> <ip_address> <TCP_port> <ip_address_of_other> <TCP_port_of_other>”

```java
    // first node to enter in the ring needs to create that
    public void create() {
        this.fingerTable.set(0, this.nodeInfo);
        // call threads that will search/ actualize successors, fingers, predecessor, etc...
        maintainer();
    }


    public void join(String address, int port) {
        // send message to find successor
        MessageBuilder messageBuilder = new MessageBuilder();
        byte[] message = messageBuilder.constructFindSuccessorMessage(this.nodeInfo, false, -1);
        Peer.getThreadExec().execute(new ThreadSendMessages(address, port, message));

        try {
            this.joinCountDownLatch.await();
        } catch (Exception e) {
            System.err.println(e.getMessage());
            e.printStackTrace();
        }

        this.predecessor = this.successor;

        maintainer();
    }
```

ChordNode | Line 136

Dentro do método ***join()***, constrói-se uma mensagem do tipo FINDSUCC que vai ser enviada pelo *peer* para o *peer* cujo endereço e a porta foram passados através da linha de comandos quando foi criado. Quando o *peer* recebe a resposta do outro, passa a conhecer o seu sucessor a *thread* desbloqueia-se, sendo que inicialmente se atribui ao predecessor o valor do seu sucessor. Este valor depois será atualizado através das funções de manutenção que são chamadas dentro do método *maintainer*. O *peer* recetor de uma mensagem do tipo FINDSUCC vai chamar o método ***findSuccessor()*** da sua classe **ChordNode**. Neste caso, como se procura encontrar um sucessor para um nó, então no interior deste método será chamada uma outra função, a ***findSuccessorNode()***.

```java
public NodeInfo findSuccessorNode(BigInteger nodeId, String ipAddress, int port) {
    if(this.nodeInfo.getNodeId().equals(this.fingerTable.get(0).getNodeId())) { // existe apenas um nó, por isso o sucessor é igual ao próprio nó
        if(!nodeId.equals(this.nodeInfo.getNodeId())) { // o nó que procura o sucessor não é o mesmo que o que recebe a mensagem
            this.predecessor = new NodeInfo(nodeId, new InetSocketAddress(ipAddress, port));
            this.successor = new NodeInfo(nodeId, new InetSocketAddress(ipAddress, port));
            this.fingerTable.set(0, this.successor);
        }
        return this.nodeInfo; // o sucessor do nó que o procura é este, assim como o sucessor deste nó é o nó que estava a procura de sucessor
    }

    if(compareNodeIds(this.nodeInfo.getNodeId(), nodeId, this.fingerTable.get(0).getNodeId())) { // se estiver compreendido entre este nó e sucessor
        return this.fingerTable.get(0);
    }

    else {
        NodeInfo n = closestPrecedingNode(nodeId);

        if(this.nodeInfo.getNodeId().equals(n.getNodeId())) {
            return this.nodeInfo;
        }

        MessageBuilder messageBuilder = new MessageBuilder();
        NodeInfo newNode = new NodeInfo(nodeId, new InetSocketAddress(ipAddress, port));
        byte[] message = messageBuilder.constructFindSuccessorMessage(newNode, false, -1);
        Peer.getThreadExec().execute(new ThreadSendMessages(n.getSocketAddress().getHostName(), n.getSocketAddress().getPort(), message));

        return null;
    }
}
```

ChordNode | Line 184

A primeira verificação que se faz neste método é se o nó recetor da mensagem tem-se a si próprio como sucessor. Caso seja verdadeiro, significa que até este momento o anel apenas tinha um nó. Nesta situação, verifica-se se o nó que enviou a mensagem é o mesmo que a recebe e, caso não seja, então existe um novo nó no anel e o recetor atualiza o seu sucessor, predecessor assim como a primeira posição da sua *finger table* (que é sempre igual ao sucessor). Caso o emissor e o recetor da mensagem sejam o mesmo, então continua a existir apenas um nó. Em ambos os casos, o nó vai sempre responder à mensagem recebida com a sua própria informação.
Caso já existam vários nós no anel, então o recetor vai verificar se o *id* do nó que pretende descobrir o seu sucessor se encontra compreendido entre o seu *id* e o *id* do seu sucessor. Caso isto se verifique, então o sucessor do outro nó é igual ao sucessor do nó que recebe a mensagem.
Por último, quando nenhuma das situações anteriores se verifica, então o nó faz uma chamada ao método ***closestPrecedingNode()***, que percorre a *finger table* e retorna o nó com maior *id* que será predecessor do nó que procura o seu sucessor. Caso o ***closestPrecedingNode()*** seja o próprio nó que recebe a mensagem, então retorna-se a ele próprio. Caso contrário, encaminha o pedido de sucessor para o ***closestPrecedingNode()***, que irá repetir todo este processo.

```
248     // search the local table for the highest predecessor of nodeId
249     public NodeInfo closestPrecedingNode(BigInteger nodeId) {
250         for(int i = M - 1; i >= 0; i--) {
251             if(this.fingerTable.get(i) == null) {
252                 continue;
253             }
254 
255             if(compareNodeIds(this.nodeInfo.getNodeId(), this.fingerTable.get(i).getNodeId(), nodeId)) {
256                 return this.fingerTable.get(i);
257             }
258         }
259 
260         return this.nodeInfo;
261     }
```

ChordNode | Line 248

Para a manutenção do *chord* foram criadas 3 funções que cada um dos nós vai chamando periodicamente, correndo cada uma delas numa *thread* diferente.

```
public void maintainer() {
    Peer.getThreadExec().scheduleAtFixedRate(new ChordMaintainer(this, "stabilize"), 3, 3, TimeUnit.SECONDS);
    Peer.getThreadExec().scheduleAtFixedRate(new ChordMaintainer(this, "fix_fingers"), 3, 5, TimeUnit.SECONDS);
    Peer.getThreadExec().scheduleAtFixedRate(new ChordMaintainer(this, "check_predecessor"), 3, 4, TimeUnit.SECONDS);
}
```

ChordNode | Line 265

Começando pelo método ***stabilize()***, quando o nó *n* o corre, pergunta ao seu sucessor, enviando uma mensagem do tipo FINDPRED, qual é o seu antecessor *p* e decide se *p* deve passar a ser o seu sucessor ou não. Esta situação ocorrerá, por exemplo, caso o nó *p* se tenha juntado recentemente ao anel. Este método também notifica, através de uma mensagem do tipo NOTIFY, o sucessor de *n* acerca da sua existência, de forma a que este possa saber da existência de *n* e atualizar o seu predecessor. O sucessor apenas faz esta atualização caso não conheça nenhum antecessor mais próximo do que *n*.

```
283     public void stabilize() {
284         if(this.nodeInfo.getNodeId().equals(this.fingerTable.get(0).getNodeId()) && this.nodeInfo.getIp().equals(this.fingerTable.get(0).getIp()) && this.nodeInfo.getPort() == this.fingerTable.get(0).getPort()) {
285             this.succPred = this.predecessor;
286         }
287         else {
288             MessageBuilder messageBuilder = new MessageBuilder();
289             byte[] message = messageBuilder.constructFindPredecessorMessage(this.nodeInfo);
290             Peer.getThreadExec().execute(new ThreadSendMessages(this.fingerTable.get(0).getIp(), this.fingerTable.get(0).getPort(), message));
291 
292             try {
293                 this.stabilizeCountDownLatch.await();
294             } catch (Exception e) {
295                 System.err.println(e.getMessage());
296                 e.printStackTrace();
297             }
298 
299             this.stabilizeCountDownLatch = new CountDownLatch(1);
300         }
301 
302         if(this.succPred != null) {
303             if(compareNodeIds(this.nodeInfo.getNodeId(), this.succPred.getNodeId(), this.fingerTable.get(0).getNodeId()) || this.nodeInfo.getNodeId().equals(this.fingerTable.get(0).getNodeId())) {
304                 this.successor = this.succPred;
305                 this.fingerTable.set(0, this.succPred);
306             }
307         }
308 
309         MessageBuilder messageB = new MessageBuilder();
310         byte[] m = messageB.constructNotifyMessage(this.nodeInfo);
311         Peer.getThreadExec().execute(new ThreadSendMessages(this.fingerTable.get(0).getIp(), this.fingerTable.get(0).getPort(), m));
312     }
```

ChordNode | Line 272

```
354 public void notify(NodeInfo node) {
355     if(this.predecessor == null || compareNodeIds(this.predecessor.getNodeId(), node.getNodeId(), this.nodeInfo.getNodeId()) || !this.predecessor.getNodeId().equals(this.nodeInfo.getNodeId())) {
356         if(node.getNodeId().equals(this.nodeInfo.getNodeId())) {
357             return;
358         }
359 
360         if(this.predecessor == null) {
361             this.predecessor = node;
362             //System.out.println("PREDECESSOR3: " + this.predecessor);
363 
364 
365             if(this.successor.getNodeId().equals(this.nodeInfo.getNodeId())) {
366                 changeSuccessor(node);
367                 //System.out.println("SUCCESSOR3: " + this.successor);
368             }
369         }
370         else if(!node.getNodeId().equals(this.predecessor.getNodeId())) {
371             this.predecessor = node;
372             //System.out.println("PREDECESSOR4: " + this.predecessor);
373 
374 
375             if(this.successor.getNodeId().equals(this.nodeInfo.getNodeId())) {
376                 changeSuccessor(node);
377                 //System.out.println("SUCCESSOR4: " + this.successor);
378             }
379         }
380     }
381 }
```

ChordNode | Line 351

O método ***fix_fingers()*** atualiza periodicamente as entradas da ***finger table*** e o índice do do próximo finger a ser atualizado. Para atualizar a ***finger table***, recorremos a uma variável global *next* que guarda a posição da próxima entrada da ***finger table*** a ser atualizada. O valor da posição *next* da ***finger table*** será igual ao nó sucessor do finger com id igual a $n + 2^{next+1}$, sendo n o id do nó atual. Para se encontrar o sucessor de um finger, recorremos a uma função bastante semelhante à utilizada para encontrar o sucessor de um nó.

```
393     public void fix_fingers() {
394         this.next += 1;
395 
396         if(this.next > M) {
397             this.next = 0;
398         }
399 
400         BigInteger p = BigDecimal.valueOf(Math.pow(2, this.next)).toBigInteger();
401         BigInteger s = p.add(this.nodeInfo.getNodeId());
402         BigInteger module = BigDecimal.valueOf(Math.pow(2, M)).toBigInteger();
403         BigInteger finger = s.mod(module);
404 
405         NodeInfo node = findSuccessor(this.nodeInfo.getIp(), this.nodeInfo.getPort(), finger, true, this.next);
406 
407         if(node != null) {
408             this.fingerTable.set(this.next, node);
409         }
410     }
```

ChordNode | Line 385

Finalmente, o método ***check_predecessor()*** tem como objetivo verificar se o predecessor de um nó ainda se encontra no anel ou se já saiu. Para isso, o nó *n* envia uma mensagem do tipo PREDALIVE para o seu predecessor. Caso ele responda com uma mensagem do tipo ALIVE, então o antecessor continua no anel. Caso não obtenha nenhuma resposta ao fim de 500ms, então muito provavelmente o seu predecessor já deixou o anel, pelo que o valor da variável *predecessor* passa a ser nulo, até que através do método periódico ***stabilize()*** descubra qual o seu novo predecessor.

```
418     public void check_predecessor() {
419         if(this.predecessor != null && !checkPredecessorAlive()) {
420             this.predecessor = null;
421         }
422 
423     }
424 
425     public boolean checkPredecessorAlive() {
426         Boolean alive = false;
427         this.answerAlive = null;
428         MessageBuilder messageBuilder = new MessageBuilder();
429         byte[] message = messageBuilder.constructPredAliveMessage(this.nodeInfo);
430         Peer.getThreadExec().execute(new ThreadSendMessages(this.predecessor.getIp(), this.predecessor.getPort(), message));
431 
432         try {
433             Thread.sleep(500);
434         } catch (Exception e) {
435             System.err.println(e.getMessage());
436             e.printStackTrace();
437         }
438 
439         if(this.answerAlive != null) {
440             alive = true;
441         }
442 
443         return alive;
444     }
```

ChordNode | Line 413

6. Fault-tolerance

Para que o nosso sistema seja tolerante a falhas, contribuem muito as funções de manutenção do *Chord* anteriormente explicadas. Através delas, os nós conseguem recuperar de falhas que possam ocorrer, como o facto de um predecessor/sucessor sair do anel.

No caso de o sucessor de um nó *n* sair do anel, então *n* iria ficar preso no ***CountDownLatch*** existente no interior do método ***stabilize()***, pois iria enviar a mensagem para saber o predecessor do seu sucessor, mas o envio desta iria falhar pois o endereço e a porta do sucessor já não estão ativos. Para evitar lançar uma exceção e bloquear o *peer* recorremos à seguinte estratégia dentro do método ***sendMessage()*** da classe **ChannelController**:

```
57          try {
58              socket = (SSLSocket) ssf.createSocket(this.ipAddress, this.port);
59              ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
60              out.flush();
61              Message toSend = new Message(message);
62              out.writeObject(toSend);
63              //System.out.println("SEND MESSAGE");
64
65              try {
66                  Thread.sleep(1000);
67              } catch(Exception e) {
68                  System.err.println(e.getMessage());
69                  e.printStackTrace();
70              }
71
72              socket.close();
73          } catch (Exception e) {
74
75              switch(header[1]) {
76                  case "FINDPRED":
77                      Peer.getChordNode().recoverStabilize();
78                      break;
79                  case "CHECKPRED":
80                      Peer.getChordNode().setPredecessor(null);
81                      break;
82                  default:
83                      /*System.err.println(e.getMessage());
84                      e.printStackTrace();
85                      break;*/
86                      System.out.println("FAILED");
87                      break;
88              }
89          }
```

ChannelController | Line 57

Desta forma, caso o envio da mensagem FINDPRED falhe, é chamado o método ***recoverStabilize()*** do **ChordNode** de forma a desbloquear o ***CountDownLatch*** e a atualizar a *finger table* e a informação do sucessor e assim recuperar da saída do sucessor.

Podemos também verificar que caso o envio da mensagem CHECKPRED falhe, existe também uma recuperação a fazer, atribuindo-se à variável *predecessor* do **ChordNode** o valor nulo.

Nos restantes casos, é imprimida uma mensagem no terminal a dizer FAILED e o *peer* continua a sua execução.

Através da garantia de que o mesmo nó não pode guardar o mesmo *chunk* mais do que uma vez, então, para os casos em que a replicação de um ficheiro é igual ou superior a 2, existem pelo menos 2 *peers* com o mesmo *chunk* guardado. Assim, desta forma, caso haja um peer que não esteja disponível ou cujos ficheiros se tenham corrompido/ perdido, existe outro *peer* que pode fornecer esse *chunk* para o *restore*.

A manutenção do *replication degree* é mantida mesmo quando há um *peer* que durante o processo de *reclaim* necessitou de apagar um *chunk* para conseguir atingir a sua nova capacidade máxima, pois ao apagar o *chunk* envia uma mensagem REMOVED para o *peer* que iniciou o processo de *backup* e este *peer* vai atualizar os seus registos e verificar se o nível de replicação atual ainda é superior ou igual à replicação desejada. Caso não seja, então este *peer* vai enviar novas mensagens de PUTCHUNK para tentar que outros nós que ainda não tenham este *chunk* guardado o possam guardar, garantindo assim a manutenção do nível de replicação.