Universidade do Porto

Peer-to-peer backup service for the Internet

Bruno Sousa **up201604145**@fe.up.pt

Catarina Figueiredo **up210606334**@fe.up.pt

Francisco Filipe **up201604601**@fe.up.ptt

Pedro Fernandes **up201603846**@fe.up.pt

Unidade Curricular: Sistemas Distribuídos

Docente: Diana Guimarães

Turma: MIEIC02

Grupo: 8

1 de Junho de 2019

Conteúdo

1.1 Introdução

Este relatório foi desenvolvido no âmbito do segundo trabalho prático da Unidade Curricular de Sistemas Distribuídos. O seu objetivo é clarificar alguns dos aspetos principais da implementação do projeto *"Peer-to-peer backup service for the Internet"*. A sua estrutura é a seguinte:

- **Overview:** Descrição das principais funcionalidades do projeto, incluindo os protocolos suportados pelo serviço de backup, bem como um resumo das funcionalidades extra implementadas(threads, JSSE , Chord e tolerância a falhas).
- **Protocolos:** Esta secção descreve, com recurso a excertos de código, a implementação de todos os protocolos desenvolvidos de uma forma clara e concisa.
- **Design de Concorrência:** Aqui estão presentes os mecanismos utilizados para o suporte de pedidos e acessos concorrentes durante a execução do sistema. Mais uma vez são utilizados alguns pedaços de código para mais facilmente expôr o raciocínio por detrás da implementação.
- **JSSE:** Descrição da implementação de JSEE neste projeto. Nesta secção expõe-se os recursos utilizados neste projeto que adicionam uma camada de segurança ao mesmo. É também discutido em que situações estes são utilizados e explorada (com recurso a código desenvolvido) a implementação em si.
- **Escalabilidade:** Nesta secção está presente a implementação do protocolo Chord. É apresentada, de forma breve, qual a funcionalidade deste, sendo posteriormente abordados alguns dos conceitos principais e implementações desenvolvidas no projeto.
- **Tolerância a Falhas:** Por último é abordada a forma como se previnem falhas que possam surgir ao longo da execução. Nesta secção estão presentes não só a solução pensada como a sua execução.

1.2 Overview

O presente projeto tem como objetivo a implementação de um serviço de backup. Este serviço executa operações de Backup, Delete, Restore e Reclaim. Foi decidido utilizar JSSE para garantir um canal de comunicação seguro, Thread pools para a comunicação entre nós da rede, Java NIO para a leitura dos ficheiros e, por último, decidiu-se implementar o Chord de forma a assegurar que o sistema desenvolvido fosse escalável.

1.2.1 Compilação

Para compilar este projeto é necessário executar o comando seguinte no diretório onde se encontra o projeto.

```
javac *.java
```

1.2.2 Execução

Para facilitar a execução do projeto foram desenvolvidos scripts para a criação dos nós e execução dos protocolos. Caso o sistema operativo seja windows é necessário a utilização de git bash para correr os scripts.

Para executar o projeto é necessário em primeiro lugar, inicializar o *chord ring*. Para isso deve-se correr a seguinte instrução:

```
java "-Djavax.net.ssl.keyStore=keystore"
     "-Djavax.net.ssl.keyStorePassword=sdis1822"
     "-Djavax.net.ssl.trustStore=truststore"
     "-Djavax.net.ssl.trustStorePassword=sdis1822"
      Node $NODE_IP $NODE_PORT

//Script
./ring_node.sh $NODE_IP $NODE_PORT
```

Os argumentos são respetivamente, o IP e a Porta do nó que pretendemos adicionar ao *chord ring*. Caso pretendamos adicionar mais nós ao *chord ring* devemos executar a seguinte instrução:

```
java "-Djavax.net.ssl.keyStore=keystore"
     "-Djavax.net.ssl.keyStorePassword=sdis1822"
     "-Djavax.net.ssl.trustStore=truststore"
     "-Djavax.net.ssl.trustStorePassword=sdis1822"
     Node $NODE_IP $NODE_PORT $RING_NODE_IP $RING_NODE_PORT

//Script
./join_node.sh $NODE_IP $NODE_PORT $RING_NODE_IP $RING_NODE_PORT
```

Os argumentos $NODE_IP e $NODE_PORT são respetivamente, o IP e a Porta do nó que pretendemos adicionar. Os dois últimos argumentos, $RING_NODE_IP e $RING_NODE_PORT são o IP e a Porta do nó ao qual nos queremos conectar, este nó tem que pertencer ao *chord ring*. Decidimos passar como argumento o IP do nó ao qual nos pretendemos conectar visto que, como os computadores têm conexões de vários tipos(wifi, cabo, etc), não encontramos uma maneira de através de código o fazer, por isso, decidimos fazê-lo manualmente.

Só depois de iniciarmos os servidores podemos executar os protocolos desenvolvidos. Para invocar qualquer um dos protocolos implementados é necessário correr um comando com o seguinte formato.

```
java "-Djavax.net.ssl.trustStore=truststore"
     "-Djavax.net.ssl.trustStorePassword=sdis1822"
     TestApp $NODE_IP $NODE_PORT $PROTOCOL $OPND_1 $OPND_2

//Script:
./test_app.sh $NODE_IP $NODE_PORT $PROTOCOL $OPND_1 $OPND_2
```

Os dois primeiros argumentos, $NODE_IP e $NODE_PORT são o IP e a Porta do nó ao qual nos pretendemos conectar. O terceiro argumento, $PROTOCOL é o nome do protocolo que pretendemos executar (BACKUP, DELETE, RECLAIM OU RESTORE). Os argumentos seguintes $OPND_1 e $OPND_2 variam de protocolo para protocolo, apenas o protocolo BACKUP utiliza o $OPND_2.

Backup

Para invocar o protocolo Backup os parâmetros deverão ser:

- $PROTOCOL = “BACKUP”
- $OPND_1 = Nome do ficheiro
- $OPND_2 = Desired replication degree

Delete

Depois de realizado o backup podemos eliminar o backup do ficheiro através do comando Delete.

- $PROTOCOL = “DELETE”
- $OPND_1 = Nome do ficheiro

Restore

Caso pretendamos recuperar o ficheiro ao qual fizemos backup podemos fazer restore do mesmo.

- $PROTOCOL = "RESTORE"
- $OPND_1 = Nome do ficheiro

Reclaim

Se pretendermos definir um espaço máximo em disco para o nó ao qual nos queremos conectar podemos executar o comando de Reclaim.

- $PROTOCOL = "RECLAIM"
- $OPND_1 = Espaço maximo em KBytes

1.3 Protocolos

A comunicação entre os nós é feita através da utilização do protocolo TCP com JSSE, desta forma garante-se uma comunicação segura e eficaz.

1.3.1 Backup

```
  for (int i = 0; i < chunks; i++) {
   int length = in.readInt();
   byte[] message = new byte[length];
   in.readFully(message, 0, message.length); // read the message

   String[] header = Utils.getHeader(message);
   byte[] content = Utils.getChunkContent(message, length);

   if (i == 0) {
     String key = header[1];
     BigInteger encrypted = Utils.getSHA1(key);

     ExternalNode successor = this.findSuccessor(this.id, encrypted);

     if (successor.id.equals(this.id)) {
       this.storeKey(this.id, encrypted, chunks + ":" + header[3]);
     } else {
       successor.storeKey(this.id, encrypted, chunks + ":" + header[3]);
     }
   }

   for (int j = 0; j < Integer.parseInt(header[3]); j++) {
     String key = header[1] + "-" + header[2] + "-" + j;
     BigInteger encrypted = Utils.getSHA1(key);

     ExternalNode successor = this.findSuccessor(this.id, encrypted);

     if (successor.id.equals(this.id)) {
       storeChunk(encrypted, key, content);
     } else {
       executor.execute(new ChunkSenderThread(this, successor, encrypted, key,
           content, false));
     }
   }
 }
```

1. Inicialmente é executado um comando no terminal contendo: IP e a porta do nó com o qual se vai comunicar; protocolo; nome do ficheiro;*replication degree*. Este pedido é processado pela TestApp sendo estabelecida uma comunicação com o nó identificado pelo IP e porta inseridos.

2. Esta, inicialmente envia o protocolo e o número de *chunks* do ficheiro ao nó com o qual está a comunicar. De seguida abre o ficheiro correspondente e envia os seus *chunks* para o nó com o qual estabeleceu a comunicação. A mensagem com os *chunks* tem o seguinte formato:

```
BACKUP <filename> <chunkNo> <replicationDegree> <CRLF><CRLF>
    <chunkContent>
```

   Termina assim, após o envio de todos os *chunks*, a execução da TestApp.

3. Este nó, que inicialmente já se encontra à escuta de pedidos (através da *thread ClientRequestListenerThread*), ao fazer a leitura do protocolo, faz o tratamento desse pedido executando uma chamada à função respetiva (neste caso a função*backup*).

4. Nesta função o nó lê o valor do número de *chunks* que irá receber executando posteriormente um ciclo para o tratamento dos chunks. Na primeira iteração é também necessário guardar informação sobre o ficheiro em si, sendo guardada uma chave encriptada (gerada a partir do nome do ficheiro) associada ao número de *chunks* e ao *replication degree*.

5. Em cada iteração deste último ciclo, cada *chunk* é guardado tantas vezes quanto o *replication degree* previamente estabelecido. Para isso executa-se um novo ciclo que a cada iteração gera uma chave a partir da concatenação do número de *chunks* e o *replication degree* atual. A partir desta chave e da chamada à função *findSuccessor*, descobre-se o nó onde se deve guardar esse *chunk*.

6. De seguida, cria-se uma *thread ChunkSenderThread* responsável pelo envio desta chave e da informação ao sucessor anteriormente descoberto.

7. O nó a receber esta informação encontra-se já a executar uma outra *thread* de tratamento de pedidos entre nós. Desta forma, ao receber o pedido de BACKUP do *chunk* atual, cria uma última *thread ChunkReceiver Thread* que guardada a informação recebida.

8. Este processo é executado para todos os *chunk*, e em cada um destes é relizado tantas vezes quanto o *replication degree* estabelecido.

1.3.2 Restore

```
    for (int i = 0; i < numChunks; i++) {
     String key = fileName + "-" + i + "-0";
     BigInteger chunkID = Utils.getSHA1(key);
     keys.add(chunkID.toString());
     successor = this.findSuccessor(this.id, chunkID);

     if (successor.id.equals(this.id)) {
       storage.addRestoredChunk(chunkID.toString(), fileName,
           storage.readChunk(chunkID));
     } else {
       executor.execute(new ChunkRequestThread(this, successor, chunkID, fileName));
       executor.schedule(new CheckChunkReceiveThread(this, successor, key, keys,
           fileName, Integer.parseInt(args[1])),
           5, TimeUnit.SECONDS);
     }
    }
```

1. Inicialmente é executado um comando no terminal contendo: IP e a porta do nó com o qual se vai comunicar; protocolo; nome do ficheiro ao qual se pretende fazer restore; Assim como no pedido anterior, este pedido é processado pela TestApp sendo estabelecida uma comunicação com o nó identificado pelo IP e porta inseridos.

2. Esta, envia o protocolo e o nome do ficheiro ao nó com o qual está a comunicar e fica à espera de receber as chunks para mais tarde restaurar o ficheiro.

3. Este nó, que inicialmente já se encontra à escuta de pedidos na *thread ClientRequestListenerThread* ao fazer a leitura do protocolo, faz o tratamento desse pedido executando uma chamada à função respetiva (neste caso a função restore).

4. Nesta função o nó lê o nome do ficheiro a que se pretende fazer restore e guarda-o numa chave encriptada (gerada a partir do nome do ficheiro). A partir da chave e da chamada à função *findSuccessor* descobre-se um dos nós que fez backup desse ficheiro. A partir deste nó descobre-se o número de chunks desse ficheiro.

5. De seguida é executado um ciclo que percorre o número total de chunks. Para cada *chunk* é criada uma chave encriptada (gerada a partir do número do *chunk* , do nome do ficheiro e do *replication degree*).

6. Com a utilização desta chave e da chamada à função *findSuccessor*, descobre-se o nó onde se deve ir buscar o *chunk*. Se for o próprio, invoca-se uma função que lê o *chunk* do disco e o guarda em memória temporária. Caso contrário, abre-se uma ligação com o sucessor e pede-se que ele envie o *chunk*, guardando-0 também em memória temporária.

7. Na eventualidade de uma possível falha do nó em questão, é pedido o mesmo *chunk* com um novo *replication degree* recorrendo para isso à *thread CheckChunkReceiveThread*.

8. Este processo é repetido para todos os *chunks* do ficheiro. O nó que recebe os *chunks*, após o envio de todos os pedidos, bloqueia a execução da função restore até receber os *chunks* todos.

9. No final é enviado para a TestApp o número total de *chunks* e o conteúdo de cada *chunk*. antes de se enviar cada *chunk* é enviado o seu tamanho. Esta lê toda a informação dos *chunks* e restaura o ficheiro.

1.3.3 Delete

```
 for (int i = 0; i < chunks; i++) {
  for (int j = 0; j < repDegree; j++) {
    String key = fileName + "-" + i + "-" + j;
    BigInteger chunkID = Utils.getSHA1(key);
    ExternalNode chunkSuccessor = this.findSuccessor(this.id, chunkID);
    if (chunkSuccessor.id.equals(this.id)) {
      deleteChunk(this.id, chunkID);
    } else {
      chunkSuccessor.deleteChunk(this.id, chunkID);
    }
  }
}
```

1. Em primeiro lugar, é executado um comando no terminal contendo: IP e a porta do nó com o qual se vai comunicar; protocolo; nome do ficheiro. Tal como anteriormente, este pedido é processado mais uma vez pela TestApp, sendo estabelecida uma comunicação com o nó identificado pelo IP e porta inseridos.

2. Esta, envia o protocolo e o nome do ficheiro ao nó com o qual está a comunicar e termina a sua execução.

3. Este nó, que inicialmente já se encontra à escuta de pedidos através da *thread ClientRequestListenerThread* ao fazer a leitura do protocolo, faz o tratamento desse pedido executando uma chamada à função respetiva (neste caso a função delete).

4. Nesta função o nó lê o nome do ficheiro e guarda-o numa chave encriptada (gerada a partir do nome do ficheiro).A partir da chave e da chamada à função *findSuccessor* descobre-se um dos nós que fez backup do ficheiro.

5. De seguida é executado um ciclo em que, através da função *findSucessor* e de uma chave encriptada(gerada através do nome do ficheiro, do número do *chunk* e do *replication degree*) descobre os nós que executaram backup desse *chunk* e assim eliminam-se os *chunks* e chaves desses *chunks*.

6. Este ciclo repete-se até que todos os *chunks* do ficheiro fornecido sejam eliminados.

1.3.4 Reclaim

```
for (File file : chunks.listFiles()) {
    String fileName = file.getName();
    BigInteger key = new BigInteger(fileName.split("\\.")[0]);
    long fileSize = file.length();

    if (spaceToReclaim <= 0)
      break;
    deleteChunk(this.id, key);
    spaceToReclaim -= fileSize;
  }
```

1. Tal como em qualquer outro protocolo, é executado um comando no terminal contendo: IP e a porta do nó com o qual se vai comunicar; protocolo; tamanho máximo; Assim como nos pedidos anteriores, este pedido é processado pela TestApp sendo estabelecida uma comunicação com o nó identificado pelo IP e porta inseridos.

2. Esta, envia o protocolo e o espaço máximo ao nó com o qual está a comunicar e termina a sua execução.

3. Este nó, que mais uma vez já se encontra à escuta de pedidos dentro da *thread ClientRequestListenerThread*, ao fazer a leitura do protocolo, realiza o tratamento desse pedido executando uma chamada à função respetiva (neste caso a função reclaim).

4. Nesta função o nó lê o espaço máximo disponível e calcula o *spaceToReclaim*. Está variável é menor que 0 caso o espaço máximo disponível seja maior que o espaço utilizado pelo nó.

5. De seguida é executado um ciclo que percorre todas as chunks que o nó guardou. Caso a variável *spaceToReclaim* seja maior que 0, elimina os chunks até que a variável tenha uma valor menor ou igual a 0.

1.4 Design da Concorrência

Na implementação de mecanismos que lidassem com a concorrência de pedidos e acessos a ficheiros recorreu-se a:

- *thread pools* na comunicação entre nós.
- *Java NIO* para a leitura dos ficheiros.

```
    ScheduledExecutorService executor;
...
executor = Executors.newScheduledThreadPool(100);
executor.execute(new NodeThread(this));
executor.scheduleAtFixedRate(new StabilizeThread(this), 15, 15, TimeUnit.SECONDS);
```

O excerto de código acima apresenta um exemplo de implementação das *threads pools* neste projeto. Durante a execução do programa algumas *threads* são imediatamente executadas quando são chamadas enquanto que outras (como é o caso deste exemplo) são agendadas para apenas realizarem uma execução periódica após um certo intervalo de tempo.

```
    FileOutputStream out = null;
    try {
      out = new FileOutputStream(fileName);
    } catch (FileNotFoundException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    FileChannel fout = out.getChannel();
    ByteBuffer byteBuffer = ByteBuffer.allocate(chunkSize);
```

Este último pedaço de código apresenta a utilização de Java NIO na leitura dos ficheiros. A utilização deste mecanismo impede que um ficheiro bloqueie quando acedido por múltiplas *threads* simultaneamente.

1.5 JSSE

Nesta implementação está também presente a utilização de JSSE. Este é utilizado em qualquer tipo de comunicação entre os nós de forma a garantir uma troca segura de mensagens (desde pedidos e respostas de peers a transferências de ficheiros). Desta forma protege-se a identificação de cada peer (IP e porta), tornando a comunicação mais segura. Por este motivo, não é possível indicar um protocolo específico em que se use este mecanismo pois ele está presente em todos.

Dos recursos disponibilizados pelo JSSE optou-se pela utilização da classe javax.net.ssl.SSLSocket. Esta classe é instanciada sempre que é necessário o envio de uma nova mensagem entre nós, ou seja:

- **Comunicação entre Peers:** troca de mensagens que indicam pedidos a realizar entre *peers* (p.e backup de um *chunk*, restore de um *chunk*, encontrar um sucessor, entre outros - presentes na função *interpretMessage* da classe *NodeThread*).
- **Comunicação entre TestApp e Peers:** realização do *"handshake"* e troca de mensagens que indicam os protocolos a realizar.

```
SSLSocketFactory factory = (SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket) factory.createSocket(ip, port);

DataOutputStream out = new DataOutputStream(socket.getOutputStream());
BufferedReader in = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));

String message = "FINDSUCCESSOR " + requestId + " " + id + " \n";
System.out.println("[Node " + requestId + "] " + message);
out.writeBytes(message);

String response = in.readLine().trim();
socket.close();
```

O excerto de código acima apresenta a implementação deste socket do lado do emissor (aquele que envia a mensagem). A implementação é relativamente simples e muito semelhante à de um socket desprovido de SSL. Apenas é necessário instanciar uma *SSLFactory* contendo a "default factory" e criar um *SSLSocket* com o ip e porta respetivos. De seguida é instanciada a stream de output (*DataOutputStream*) , encarregue de enviar a mensagem para outro nó. É também neste momento que se instancia um *BufferedReader* responsável pela leitura da resposta. Posteriormente constrói-se a mensagem a enviar pela stream de output. Na penúltima instrução, bloqueia-se a thread atual, que fica assim à espera de uma resposta por parte do recetor. Uma vez recebida esta resposta é fechado o socket. É importante referir que isto é apenas um exemplo de uma aplicação de um *SSLSocket*.

```
SSLServerSocketFactory ssf = (SSLServerSocketFactory)
    SSLServerSocketFactory.getDefault();
SSLServerSocket listenSocket = (SSLServerSocket) ssf.createServerSocket(node.port);

while (true) {
  SSLSocket connection = (SSLSocket) listenSocket.accept();
  node.executor.execute(new Runnable() {
  public void run() {
       try {
          interpretMessage(connection);
       } catch (IOException e) {
       }
        }
   });
 }
```

Já este último pedaço de código apresenta a implementação deste mesmo socket do lado do recetor (aquele que recebe a mensagem). Aqui é instanciada uma *SSLServerFactory* onde vai estar presente um *SSLServerSocket*. Este último, a cada pedido que recebe, cria uma nova conexão através de um *SSLSocket* que vai estar responsável por esse pedido. Por último é feita uma chamada à função *interpretMessage* onde, de acordo com o pedido executado, realiza as operações necessárias e envia uma resposta ao emissor através de uma *DataOutputStream*. Mais uma vez isto é apenas uma aplicação de *SSLSocket* do lado do recetor.

1.6 Escalabilidade

Para assegurar que o sistema desenvolvido é escalável foi implementado o Chord, um protocolo escalável para pesquisas num sistema peer-to-peer dinâmico (com entradas e saídas frequentes de nós e com número altamente variável de nós). Este fornece suporte apenas para uma operação: dada uma chave, mapeia essa chave num nó. Esse nó pode ser responsável por armazenar um valor associado à chave.

A utilização do Chord como protocolo torna o sistema escalável pois o custo de uma pesquisa cresce logaritmicamente com o aumento do número de nós (log(N) em que N é o número de nós na rede). Como tal, até mesmo sistemas muito grandes são alcançáveis.

1.6.1 Implementação do Chord

Esta secção centra-se apenas na implementação dos conceitos principais do protocolo Chord. O seu objetivo é esclarecer um pouco estes conceitos e falar da abordagem utilizada na implementação dos mesmos.

Chaves

No protocolo Chord toda a informação guardada é encriptada. Para isso é utilizado o algoritmo de encriptação *SHA-1*. Este algoritmo gera chaves de 160 bits. Desta forma, este é o número máximo de entradas presente na *finger table* de cada nó, pois representa o valor mínimo de nós necessário para conseguir chegar a qualquer outro nó de uma rede de N nós com um custo de log(N).

Existem 2 tipos de chaves: file-info e chunk.

- **File-info:** chave contendo informação acerca do nome do ficheiro, o número de *chunks* que esse mesmo ficheiro contém e o *replciation degree* deste.
- **Chunk:** chave contendo informação sobre o nome do ficheiro, o número do *chunk* atual e o valor do *replication degree* atual.

Os identificadores dos nós também são encriptados, estes contém os valores de IP e porta do nó respetivo.

Chord Ring

Dentro do Chord, para que um nó consiga chegar a todos os outros nós e de forma a que estes cheguem a ele, é necessário guardar, para cada nó, o seu antecessor, o sucessor e uma *finger table* (que será abordada na secção seguinte). O conjunto de todos os nós de um sistem organizado desta forma é comumente designado por *ring*. A implementação desta caraterística no sistema desenvolvido foi realizada através da criação de 3 atributos na classe *Node*, sendo estes:

- **predecessor:** Nó que antecede o nó atual.

- **successor:** Nó que sucede o nó atual.
- **fingerTable:** Nós que estão ligados ao nó atual.

Finger Table

Para evitar a pesquisa linear, o Chord implementa um método de pesquisa mais rápido, exigindo que cada nó mantenha uma tabela contendo até m entradas, lembrando que m é o número de bits na chave encriptada por *SHA-1*. A primeira entrada da *finger table* é, na verdade, o sucessor imediato do nó. Sempre que um nó pretende pesquisar uma chave, apenas necessita de consultar o sucessor ou predecessor mais próximo na sua *finger table*, até que um nó descubra que a chave está armazenada no seu sucessor imediato.

Com essa *finger table*, o número de nós que devem ser contactados para localizar um sucessor na rede de N nós é O(log N).

Na implementação desta *finger table* apenas se criou um atributo na classe *Node* que consiste num array de *External Node*. Este atributo está constantemente a ser alterado pela *thread StabilizeThread*, que periodicamente modifica esta estrutura de dados de forma a ficar o mais atual possível.

Find Successor

Esta é a operação central deste protocolo. A sua função é, dada uma chave, retornar o nó onde esta chave deve ser guardada. Para isso implementou-se a seguinte função:

```
public ExternalNode findSuccessor(BigInteger requestId, BigInteger id) {
    if (id.equals(this.id))
      return this;

    if (idBetween(id, this.id, this.successor.id))
      return this.successor;

    ExternalNode n0 = closestPrecedingNode(id);

    if (n0.id.equals(this.id))
      return this;

    return n0.findSuccessor(this.id, id);
  }
```

Esta função está presente na classe *Node* e consiste num *override* à função da classe mãe *External Node*. A sua finalidade é retornar o nó onde deverá ser guardada a chave *id* (segundo parâmetro da função acima apresentada). Caso essa chave seja igual ao nó atual, então é neste que deve ser guardada. Senão, caso esta se encontre entre o nó atual e o seu sucessor, deverá ser guardada neste mesmo sucessor. Caso nenhuma destas condições seja cumprida, recorrendo à *finger table* é executada uma chamada a esta mesma função no nó que se

encontrar mais próximo da chave (*id*) e que a antecede. Isto vai gerar um pedido que vai ser enviado a esse nó (através da função *findSuccessor* do *ExternalNode*). Este último, ao receber tal pedido, volta o pedaço de código acima apresentado. Este processo repete-se até que seja encontrado o sucessor da chave em questão, ou seja, o *id*.

Estabilização

Para garantir pesquisas corretas, todos os sucessores devem estar atualizados. Portanto, um protocolo de estabilização é executado periodicamente em segundo plano, o que atualiza as *finger tables* e sucessores.

O protocolo de estabilização funciona da seguinte maneira:

- **stabilize():** N pergunta ao seu sucessor pelo predecessor P e decide se P deve ser o sucessor de N ou não (isto acontece caso P tenha entrado recentemente no sistema).

```
public void stabilize() {
    ExternalNode x = this.successor.getPredecessor(this.id);

    if (x != null && !this.id.equals(x.id)
        && (this.id.equals(this.successor.id) || idBetween(x.id,
            this.id, successor.id))) {
    this.successor = x;
    fingerTable[0] = x;
    }

    this.successor.notify(this.id, this);
}
```

- **fixFingers():** Realizar um pequeno *update* à *finger table*. Visto que podem haver entradas de novos nós, e saídas de outros, convém ter uma versão o mais atualizada possível da *finger table*.

```
private BigInteger calculateFinger(int i) {
    return ((this.id.add(new BigInteger("2").pow(i))).mod(new
        BigInteger("2").pow(fingerTable.length)));
}

public void fixFingers() {
    for (int i = 1; i < fingerTable.length; i++) {
    BigInteger fingerID = calculateFinger(i);
    fingerTable[i] = findSuccessor(this.id, fingerID);
    }
}
```

- **checkPredecessor():** Notifica o sucessor de N da sua existência para que este possa alterar o seu antecessor para N.

```
public void checkPredecessor() {
    if (this.predecessor != null &&
        this.predecessor.failed(this.id))
      this.predecessor = null;
}
```

1.7 Tolerância a Falhas

Como forma de prevenir que saídas ou encerramentos inesperados de alguns nós do sistema causem a perda da informação contida nesse nó (chaves ou chunks), implementou-se um protocolo de tolerância a falhas. Esse protocolo tem por base a utilização de *replication degree*, semelhante ao já implementado no primeiro projeto desta Unidade Curricular.

1.7.1 Implementação da Tolerância a Falhas

De forma a tornar esta explicação mais clara será dado como exemplo a implementação e importância do *replication degree* nos dois protocolos mais relevantes: **BACKUP** e **RESTORE**. Os protocolos DELETE e RECLAIM não são aqui referidos pois a existência ou não de um mecanismo de tolerância a falhas não afeta estes protocolos.

- **BACKUP:** Neste protocolo, caso não existisse *replication degree*, ao realizar-se um *backup* de um *chunk* apenas seria gerada uma chave com o nome do ficheiro e número do *chunk* atual. Esta chave, após encontrado o seu sucessor, seria guardada no respetivo nó. O problema desta implementação baseia-se no facto de que, caso este último nó inesperadamente sofra uma falha (p.e. abandonar o sistema), a informação desse *chunk* perde-se por completo, comprometendo assim o bom funcionamento de todo o sistema.

```
for (int j = 0; j < Integer.parseInt(header[3]); j++) {
    String key = header[1] + "-" + header[2] + "-" + j;
    BigInteger encrypted = Utils.getSHA1(key);

    ExternalNode successor = this.findSuccessor(this.id, encrypted);

    if (successor.id.equals(this.id)) {
    storeChunk(encrypted, key, content);
    } else {
    executor.execute(new ChunkSenderThread(this, successor,
        encrypted, key, content, false));
    }
 }
```

Com a introdução deste *replication degree* a cada *chunk* são geradas várias chaves (tantas quanto o valor do *replication degree* - header[3] no código acima) utilizando o nome do ficheiro, o número do *chunk* atual e o valor do *replication degree* atual. Apesar de, num determinado *chunk*, apenas variar o *replication degree*, esta mudança já é o suficiente para que sejam geradas chaves distintas para *replication degrees* distintos. Desta forma, cada cópia desse *chunk* será aleatoriamente espalhada pelo sistema, diminuindo a probabilidade de que, ao falhar um nó, haja perda imediata de um *chunk*.

- **RESTORE:** Este protocolo está encarregue de pesquisar o sistema por *chunks* de um ficheiro para restaurar no seu nó. Caso não se tivesse em conta esta tolerância a falhas, no momento de BACKUP de um ficheiro cada *chunk* seria apenas guardado uma única vez. Desta forma, no momento do pedido RESTORE desse ficheiro, caso algum nó, por motivo de falha, não se encontrasse ligado ao sistema, tornar-se-ia impossível restaurar o ficheiro em questão.

```
        for (int i = 0; i < numChunks; i++) {
            String key = fileName + "-" + i + "-0";
            BigInteger chunkID = Utils.getSHA1(key);
            keys.add(chunkID.toString());
            successor = this.findSuccessor(this.id, chunkID);

            if (successor.id.equals(this.id)) {
                storage.addRestoredChunk(chunkID.toString(), fileName,
                    storage.readChunk(chunkID));
            } else {
                executor.execute(new ChunkRequestThread(this, successor,
                    chunkID, fileName));
                executor.schedule(new CheckChunkReceiveThread(this,
                    successor, key, keys, fileName,
                    Integer.parseInt(args[1])),
                    5, TimeUnit.SECONDS);
            }
        }
```

Com a introdução do *replication degree*, é gerada uma chave contendo, tal como no protocolo BACKUP, o nome do ficheiro, o número do *chunk* atual e o valor do *replication degree* atual. Com esta chave e recorrendo à função *findSuccessor* descobre-se o nó contendo este chunk. Caso este nó não seja ele próprio, é criada uma *thread* responsável pelo envio do pedido RESTORE ao nó correspondente. Para além disso, é criada ainda uma outra *thread* responsável por verificar se o *chunk* pedido já foi recebido ou não. Caso não tenha sido recebido (por falha do nó ao qual se pediu o *chunk*), é pedido o mesmo *chunk* mas com o *replication degree* imediatamente acima do anterior.

Desta forma é possível verificar que, uma falha num nó do sistema já não compromete por completo o bom funcionamento do sistema visto que, com um *replication degree* associado, a probabilidade de um ficheiro ficar incompleto no sistema reduz substancialmente.