Relatório do Projeto 2

Serverless Distributed Service

Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática e Computação

29 de Maio de 2020

Grupo T3G22

Gustavo Nunes Ribeiro de Magalhães	up201705072@fe.up.pt
Nuno Miguel Teixeira Cardoso	up201706162@fe.up.pt
João Francisco de Pinho Brandão	up201705573@fe.up.pt
Tiago Gonçalves da Silva	up201705985@fe.up.pt

Índice

Índice **2**

Visão Geral **3**

Aplicação Cliente **3**

Comunicação da Aplicação Cliente **3**

Aplicação Peer **4**

Comunicação da Aplicação Peer 4

Formato de mensagens 4

Protocolos 5

Backup 5

Restore 6

Delete 8

Reclaim 9

Design de Concorrência **11**

JSSE **12**

Escalabilidade **14**

Implementação 15

Node.java 15

Peer.java 17

Criação da chord 17

Entrada na Chord 18

Manutenção da Chord 20

Tolerância a falhas **22**

Visão Geral

Este projeto tem como objetivo a implementação de um sistema distribuído de backup para Internet. Este serviço executa todas as operações de Backup, Restore, Delete e Reclaim. Foram utilizados Thread Pools para a comunicação entre nós, utilizamos JSSE através de SSL Sockets para garantir uma maior segurança na comunicação, Java NIO para a leitura de ficheiros e, por último, implementou-se o Chord de forma a que o nosso sistema desenvolvido fosse escalável.

Aplicação Cliente

Para que os protocolos implementados possam ser acionados mostra-se necessário haver uma aplicação que consiga executá-los, assim, através da aplicação TestApp é possível efetuar o *backup* de ficheiros, restaurar e eliminar os ficheiros que guardou no sistema previamente e, por fim, definir o espaço que dispõe para guardar ficheiros de outros clientes inseridos no sistema.

Comunicação da Aplicação Cliente

A comunicação da aplicação cliente é feita com a aplicação *Peer* que corra na sua máquina. Esta comunicação é feita com recurso à interface de *Remote Method Invocation* (*RMI*) e permite a um processo a invocação de métodos ou funções de um processo que implemente esta interface. Disponíveis a esta interface estão os métodos *backup*, *restore*, *delete*, *reclaim* e *state* que possuem as seguintes funções:

- Backup – armazenar um ficheiro na rede *peer-to-peer*
- Restore – restaurar um ficheiro previamente enviado para a rede *peer-to-peer.*
- Delete – eliminar um ficheiro previamente enviado para a rede *peer-to-peer.*
- Reclaim – definir o espaço disponível para armazenamento de ficheiros provenientes de outros *peers* na rede.
- State – *output* da informação do estado atual do *peer*

Aplicação Peer

Comunicação da Aplicação Peer

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

Formato de mensagens

As mensagens trocadas pelo sistema são em formato de objeto da classe *Message*, tendo como atributos um *String[ ] header* e um *byte[ ]* body. O atributo *body* representa o conteúdo a ser transmitido quando necessário (*chunks*), enquanto que *header* é o cabeçalho da mensagem, que pode assumir um dos seguinte formatos:

1. *PUTCHUNK <FileId> <ChunkNo> <ReplicationDeg> <SenderAddress> <SenderPort>*
2. *STORED <FileId> <ChunkNo> <SenderAddress> <SenderPort>*
3. *GETCHUNK <FileId> <ChunkNo> <SenderAddress> <SenderPort>*
4. *CHUNK <FileId> <ChunkNo> <SenderAddress> <SenderPort>*
5. *DELETE <FileId> <SenderAddress> <SenderPort>*
6. *REMOVED <FileId> <ChunkNo> <SenderAddress> <SenderPort>*
7. *FINDSUCC <SenderId> <ReqIpAdress> <ReqPort> <ReqId>*
8. *SUCC <SenderId> <SuccId> <SuccAddress> <SuccPort>*
9. *FINDSUCCFINGER <SenderId> <ReqIpAdress> <ReqPort> <ReqId> <FingerId>*
10. *FINGERSUCC <SenderId> <SuccId> <SuccAddress> <SuccPort> <FingerId>*
11. *NOTIFY <SenderId> <ReqIpAdress> <ReqPort>*
12. *FINDPRED <SenderId> <ReqIpAdress> <ReqPort>*
13. *PRED <SenderId> <PredId> <PredAddress> <PredPort>*
14. *CHECKPRED*

Legenda:

FileId - identificador do ficheiro;

ChunkNo - número do *chunk*;

ReplicationDeg - grau de replicação desejado para os *chunks*;

SenderAddress - endereço ip do *peer* transmissor;

SenderPort - porta do *peer* transmissor;

ReqIpAdress - endereço de ip do *peer* que originalmente fez o pedido;

ReqPort - porta do *peer* que originalmente fez o pedido;

ReqId - identificador do *peer* que originalmente fez o pedido;
SenderId - identificador do *peer* que faz o envio da mensagem;
*SuccId/*PredId - Identificador do sucessor ou antecessor encontrado;
*SuccAddress/*PredAddress - endereço de ip do sucessor ou antecessor encontrado;
*SuccPort/*PredPort - porta do sucessor ou antecessor encontrado;
FingerId - índice da posição da *fingerTable* do *peer* que originalmente fez o pedido;

Protocolos

Backup

O protocolo de *backup* foi concebido tirando partido da *finger table* fornecida pela implementação do algoritmo *Chord* para a manutenção da rede de *peers*. Quando a ativação deste protocolo é requisitada o *peer* alvo deste pedido, inicialmente, prepara o ficheiro a ser enviado, primeiro verificando se este existe e, de seguida, dividindo-o em pacotes de 64*kb*. Posteriormente, é construída uma mensagem (*PUTCHUNK*) com o formato acima apresentado. Para efetuar o envio, um *peer* percorre a sua *finger table* e envia a mensagem para tantos nós quanto o *replication degree* pedido. Na eventualidade deste número ultrapassar o número de posições da *finger table* a mensagem é novamente enviada para o sucessor o número restante de vezes, servindo-se do procedimento de retransmissão a seguir explicado.

Do lado do recetor, um pedido de backup é resolvido de duas formas alternativas, consoante o resultado das duas seguintes operações: verificação de espaço disponível, verificação de posse prévia do pacote recebido. Assim, caso possua espaço de armazenamento suficiente para guardar o pacote e não tenha ainda guardado o mesmo, então este será armazenado e uma mensagem de confirmação (*STORED*) é enviada para o *peer* que inicialmente enviou a mensagem, caso contrário, a mensagem recebida é transmitida para o seu sucessor, fornecido pelo algoritmo Chord. Deste modo a transmissão apenas cessa quando um *peer* guarda o pacote recebido ou no *peer* que iniciou o protocolo, ao notar que recebeu uma mensagem *PUTCHUNK* com início em si mesmo.

Peer.java

```
public synchronized String backup(String file_path, Integer replication_degree) {
   //File creation
   FileInfo file = new FileInfo(file_path);
   file.prepareChunks(replication_degree);

   //File store
   storage.storeFile(file);

   //Send PUTCHUNK message for each file's chunk
   Iterator<Chunk> chunkIterator = file.getChunks().iterator();
   MessageFactory messageFactory = new MessageFactory();

   while(chunkIterator.hasNext()) {
       Chunk chunk = chunkIterator.next();
       Message msg = messageFactory.putChunkMsg(Peer.getPeer().getAddress(),
Peer.getPeer().getPort(), chunk, replication_degree);

       for (int i = 0; i < chunk.getDesired_replication_degree(); i++) {
           Node destNode;
           if (i < fingerTable.length)
               destNode = fingerTable[i];
           else destNode = fingerTable[0];
           threadExecutor.execute(new SendMessagesManager(msg, destNode.getAddress(),
destNode.getPort()));
           msg.printSentMessage();

           try {
               Thread.sleep(500);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
           String chunkKey = chunk.getFile_id() + "-" + chunk.getChunk_no();
           PutChunkAttempts putChunkAttempts = new PutChunkAttempts(1, 5, msg, chunkKey,
replication_degree, destNode);
           Peer.getThreadExecutor().schedule(putChunkAttempts, 1, TimeUnit.SECONDS);
       }
   }

   return "Backup successful";
}
```

Restore

Na execução do protocolo de *restore*, para que fosse possível obter todos os pacotes pertencentes ao ficheiro requerido, foi utilizado o mapeamento que cada *peer* mantém quando recebe mensagens de confirmação de armazenamento e de remoção de pacotes. Através destas, um *peer* mantém um registo atualizado de todos os *peers* que

detêm os pacotes de todos os ficheiros de que pediu backup, desde que esteja ligado à rede quando estas são enviadas. Com isso, quando o protocolo de *restore* é ativado, um *peer* envia, sequencialmente, um pedido de envio (*GETCHUNK*) dos pacotes que pertencem a um determinado ficheiro para os *peers* que constam no mapeamento que mantém.

Quando recebe uma mensagem *GETCHUNK* um *peer*, após verificar que possui o pacote requerido, constrói uma *CHUNK* e responde ao *peer* que efetuou o pedido.

Peer.java

```
public synchronized String restore(String file) {
   boolean file_exists = false;
   Storage peerStorage = this.storage;

   for (int i = 0; i < peerStorage.getStoredFiles().size(); i++) {
       FileInfo fileInfo;

       if (peerStorage.getStoredFiles().get(i).getFile().getName().equals(file)) {
           file_exists = true;
           //Get previously backed up file
           fileInfo = peerStorage.getStoredFiles().get(i);
           //Get file chunks
           Set<Chunk> chunks = peerStorage.getStoredFiles().get(i).getChunks();
           Iterator<Chunk> chunkIterator = chunks.iterator();

           //For each chunk
           while(chunkIterator.hasNext()){
               Chunk chunk = chunkIterator.next();
               //Prepare message to send
               MessageFactory messageFactory = new MessageFactory();
               Message msg = messageFactory.getChunkMsg(Peer.getPeer().getAddress(),
Peer.getPeer().getPort(), fileInfo.getFileId(), chunk.getChunk_no());

               String chunkKey = fileInfo.getFileId()+'-'+chunk.getChunk_no();
               if
(Peer.getPeer().getStorage().getPeers_with_chunks().containsKey(chunkKey)) {
                   for (int j = 0; j <
getStorage().getPeers_with_chunks().get(chunkKey).size(); j++) {
                       String[] destPeer =
getStorage().getPeers_with_chunks().get(chunkKey).get(j);

                       //Send message
                       Peer.getThreadExecutor().execute(new SendMessagesManager(msg,
destPeer[0], Integer.parseInt(destPeer[1])));
                       msg.printSentMessage();
                   }
               }
               else return "Chunk was not backed up previously";
           }
```

```
            Peer.getThreadExecutor().execute(new RestoreChunks(file,
fileInfo.getChunks().size()));
            break;
        }
    }

    if (file_exists)
        return "Restore successful";
    else return "File was not backed up previously";
}
```

Delete

O protocolo de *DELETE* tem como função remover um ficheiro especificado, e todos os seus bocados, do sistema. Este protocolo é pedido ao *peer* portador do ficheiro, que ficará responsável por contactar os restantes de *peers* do sistema, com vista na remoção de todos os *chunks* anteriormente copiados através do protocolo de *BACKUP*.

O *peer* portador do ficheiro armazena na sua *Storage* uma correspondência entre os *peers* e os *chunks* que estes armazenam, através de uma *ConcurrentHashMap*. Deste modo, no momento de executar o protocolo de *DELETE*, o *peer* sabe exatamente a quais *peers* enviar a mensagem de remoção de *chunks* (*DELETE*) relativos ao ficheiro que se pretende apagar. Ao receber uma mensagem deste tipo, o *peer* recetor apaga os *chunks,* correspondentes ao ficheiro indicado pelo *fileId,* da sua *Storage*.

Do lado do *peer* transmissor resta agora remover da *ConcurrentHashMap* todas as entradas relativas ao ficheiro e, por fim, apagá-lo também do seu armazenamento.

Peer.java

```
public synchronized String delete(String file_path) {
    boolean file_exists = false;
    FileInfo fileInfo;
    for (int i = 0; i < storage.getStoredFiles().size(); i++) {
        if (storage.getStoredFiles().get(i).getFile().getName().equals(file_path)) {
            file_exists = true;

            //Get previously backed up file
            fileInfo = storage.getStoredFiles().get(i);

            //Get file chunks
            Set<Chunk> chunks = fileInfo.getChunks();
            Iterator<Chunk> chunkIterator = chunks.iterator();
            MessageFactory messageFactory = new MessageFactory();
```

```
            //For each chunk
            while (chunkIterator.hasNext()) {
                Chunk chunk = chunkIterator.next();

                //Prepare message to send
                Message msg = messageFactory.deleteMsg(Peer.getPeer().getAddress(),
Peer.getPeer().getPort(), chunk);

                String chunkKey = fileInfo.getFileId()+'-'+chunk.getChunk_no();
                if
(Peer.getPeer().getStorage().getPeers_with_chunks().containsKey(chunkKey)) {
                    ArrayList<String[]> peers_with_chunk =
Peer.getPeer().getStorage().getPeers_with_chunks().get(chunkKey);
                    for (int j = 0; j < peers_with_chunk.size(); j++) {
                        String[] destNode = peers_with_chunk.get(j);
                        //Send message
                        Peer.getThreadExecutor().execute(new SendMessagesManager(msg,
destNode[0], Integer.parseInt(destNode[1])));
                        System.out.println("To "+destNode[0]+":"+destNode[1]);
                        msg.printSentMessage();
                    }
                    storage.remove_entry_peer_chunks(chunkKey);
                }
            }
            //Delete file
            storage.deleteFile(fileInfo);

            break;
        }
    }

    if (file_exists)
        return "Delete successful";
    else return "File does not exist";
}
```

Reclaim

No sistema em questão, os *peers* participantes mantêm total controlo da sua *Storage*. Como tal, o protocolo de RECLAIM funciona como uma salvaguarda que permite aos *peers* a gestão do armazenamento. Ao executar o protocolo de RECLAIM é especificado um número máximo de *KBytes* a poder ser usado para cópias de segurança naquele *peer*. Caso o valor especificado seja inferior ao espaço já em uso no *peer*, este remove *chunks* até que o valor de armazenamento seja atingido. De cada vez que um *peer* remove um *chunk* por execução deste protocolo, o *peer* encarrega-se de enviar uma mensagem *REMOVED* ao *peer* dono do ficheiro.

Ao receber uma mensagem deste género, o *peer* decrementa o grau de replicação atual do *chunk* e remove da sua *Storage* a entrada correspondente da *ConcurrentHashMap* referida no protocolo anterior. De seguida, caso o grau de replicação atual do *chunk* seja agora inferior ao grau de replicação desejado, o *peer* portador do ficheiro deverá iniciar um novo processo de BACKUP do *chunk*. O *peer* envia então uma nova mensagem *PUTCHUNK* ao seu sucessor, para que este execute uma de duas tarefas: ou armazene o novo *chunk,* caso não o tenha e possua espaço livre suficiente*,* ou a encaminhe para outro *peer*.

Peer.java

```
public synchronized String reclaim(Integer max_space) {
   System.out.println("Space Requested: "  + max_space);
   double spaceUsed = storage.getOccupiedSpace();
   System.out.println("Space Used: "  + spaceUsed);

   double spaceClaimed = max_space * 1000; //The client shall specify the maximum disk
space in KBytes (1KByte = 1000 bytes)
   System.out.println("Space Claimed: "  + spaceClaimed);

   double tmpSpace = spaceUsed - spaceClaimed;

   if (tmpSpace > 0) {
       double deletedSpace = 0;
       Iterator<Chunk> chunkIterator = storage.getStoredChunks().iterator();
       MessageFactory messageFactory = new MessageFactory();

       while (chunkIterator.hasNext()) {
           Chunk chunk = chunkIterator.next();
           System.out.println("Deleted Space: " + deletedSpace);
           System.out.println("Temp Space: " + tmpSpace);

           if (deletedSpace < tmpSpace) {
               deletedSpace += chunk.getChunk_size();

               Message msg = messageFactory.reclaimMsg(Peer.getPeer().getAddress(),
Peer.getPeer().getPort(), chunk);

               // Send reclaim message to chunk owner
               Node destNode = chunk.getOwner();
               //Send message
               Peer.getThreadExecutor().execute(new SendMessagesManager(msg,
destNode.getAddress(), destNode.getPort()));
               msg.printSentMessage();

               String filepath = storage.getDirectory().getPath() + "/file" +
chunk.getFile_id() + "/chunk" + chunk.getChunk_no();
               File file = new File(filepath);
```

```
                file.delete();
                storage.deleteDirectory(chunk.getFile_id());

                String chunkKey = chunk.getFile_id() + "-" + chunk.getChunk_no();
                storage.decrementChunkOccurences(chunkKey);

                chunkIterator.remove();
            }
            else {
                break;
            }
            System.out.println(" ");
        }

        storage.reclaimSpace(max_space);

    }

    return "Reclaim successful";
}
```

Design de Concorrência

No sistema todos os peers utilizam *threadPools* tanto para leitura e escrita de pedidos como para manutenção do chord. Para isso todos os peers possuem um objeto *threadExecutor* do tipo *ScheduledThreadPoolExecutor*, onde qualquer thread que o peer tenha de correr será agendada/evocado a partir daíi, em baixo apresentamos um fragmento de código exemplificando a sua evocação.

Peer.java

```
//Create executor
threadExecutor = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(200);
threadExecutor.execute(new SSLConnection(ipAddress,port));
```

Todos os peers quando se juntam ao sistema colocam a correr a thread *SSLconnection*, indefinitivamente para ler mensagens recebidas, assim encaminhando-as para a thread *ReceivedMessagesManager*, que lendo o header da mensagem recebida decidirá qual o comportamento a adotar para esse pedido. Com este design conseguimos assim fazer leituras a todas a mensagens recebidas em cada peer, nunca havendo bloqueios dado cada mensagem gerar uma nova thread para tratamento de mensagem.

Outra thread aberta em cada Peer é a *ChordManager* que corre sempre a uma frequência fixa, executando todas as funções de manutenção do chord.

Quanto à leitura de ficheiros optamos pelo fazer recorrendo a Java NIO, permitindo assim evitar que o ficheiro bloqueia quando é acedido por múltiplas threads ao mesmo tempo. Podemos assim observar no fragmento de código abaixo como foi implementado.

FileInfo.java

```
FileInputStream fis = null;
      try {
         fis = new FileInputStream(this.file);
      } catch (FileNotFoundException e) {
          e.printStackTrace();
      }

      FileChannel fc = fis.getChannel();
      ByteBuffer bb = ByteBuffer.allocate(chunk_size);
```

JSSE

Toda a comunicação executada entre todos os nossos peers é realizada através de *SSL Sockets*, implementação esta que oferece uma maior segurança na comunicação que o uso de um simples *socket TCP*.

Todas a mensagens trocadas, usam portanto este tipo de socket, conclui-se assim que todos os protocolos implementados usam *SSLsocket*. Excetua-se o *TestApp* que comunica por *RMI* para comunicar com o seu *peer*, dado assumirmos que o *TestApp* comunica sempre com o peer que está ativo dentro da própria máquina em que é executado. Posto isto, qualquer *peer* pode enviar e receber mensagens tanto a nível do protocolo, mas também qualquer mensagem que envolva a entrada e manutenção no chord.

Todos os peers quando são inicializados necessitam de ser evocados sempre com *flags* com referências à chave por nós criada, para isso os peers devem ser evocados sempre

“java -Djavax.net.ssl.keyStore=keystore -Djavax.net.ssl.keyStorePassword=sdis1920 -Djavax.net.ssl.trustStore=truststore -Djavax.net.ssl.trustStorePassword=sdis1920 Peer <argumentos>”.

Sempre que o peer é inicializado cria a thread *SSLConnection*, que fica a correr continuamente em paralelo lendo qualquer mensagem que lendo seja enviada, reencaminhado-a. Representada no fragmento de código abaixo.

SSLConnection.java

```
public void run() {

       SSLServerSocketFactory sslServerSocketFactory = (SSLServerSocketFactory)
SSLServerSocketFactory.getDefault();
       SSLServerSocket serverSocket;

       try {

           serverSocket = (SSLServerSocket)
sslServerSocketFactory.createServerSocket(this.port);

           while (true) {

               SSLSocket clientSocket = (SSLSocket) serverSocket.accept();
               Peer.getThreadExecutor().execute(new
ReceivedMessagesManager(clientSocket));
           }

       } catch (IOException e) {
           System.out.println("Server - Failed to create SSLServerSocket");
           e.getMessage();
           return;
       }

   }
```

Posteriormente sempre que um peer queira enviar uma mensagem apenas cria um objeto do tipo *SSLConnection* com o ip e porta a quem quer enviar, e evoca o método *send* com a mensagem a enviar, representado abaixo.

SSLConnection.java

```
public void send(Message msg) {
       //Create socket
       try {
           SSLSocket sslSocket = (SSLSocket)
```

```
SSLSocketFactory.getDefault().createSocket(this.ipAddress, this.port);

            dos = new ObjectOutputStream(sslSocket.getOutputStream());
            dos.flush();
            dos.writeObject(msg);

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            sslSocket.close();

        } catch (IOException e) {
            if (msg.getHeader()[0].equals("CHECKPRED")) {
                Peer.predNode = null;
                System.out.println("Server - Failed to connect to predecessor");
            }
            else if (
                    msg.getHeader()[0].equals("PUTCHUNK") ||
                    msg.getHeader()[0].equals("STORED") ||
                    msg.getHeader()[0].equals("GETCHUNK") ||
                    msg.getHeader()[0].equals("CHUNK") ||
                    msg.getHeader()[0].equals("DELETE") ||
                    msg.getHeader()[0].equals("REMOVED") ||
                    msg.getHeader()[0].equals("NOTIFY") // Prevent blocking when
successor isn't found
            ) {
                System.out.println("Error - Failed to connect with peer");
            }
            else {
                Peer.unlockStabilize();
                System.out.println("Error - Failed to connect with peer\nStabilizing");
            }
        }
    }
```

Escalabilidade

Para que o nosso sistema fosse escalável, decidimos optar pela sua implementação usando o Chord, para a sua implementação não só nos baseamos nos slides da unidade curricular, mas mais que isso, no artigo fornecido pelo professor Pedro Souto no website da unidade curricular (https://dl.acm.org/doi/pdf/10.1109/TNET.2002.808407?download=true).

Esta implementação torna-se quase ótima para sistemas de larga escala dado a sua pesquisa de nós ter um aumento logarítmico, e mais que isso pelo facto de se conseguir manter a Chord estabilizada com entradas e saídas no sistema.

Podemos explorar esta implantação generalizando a sua implementação e observando os seus comportamentos de inicialização, ligação de novo nó e estabilização.

Implementação

O nosso chord encontra-se essencialmente implementado em duas classes o Node, sendo um nó da corda e no Peer sendo um filho da classe node com a exclusividade de suportar o método main fazendo a criação ou entrada do chord, também é o Peer que suporta os algoritmos para as movimentações feitas no chord.

De notar que dentro da classe Peer estão definidos atributos que permitem cada peer no Chord consiga chegar a todos os outros, são esses:

- *succNode* - guarda o nó sucessor ao peer, ou seja, o nó ativo imediatamente ao lado do peer no chord;
- *predNode* - assemelhando-se ao *succNode*, mas aqui guardando o nó antecessor;
- *fingerTable* - guardando a informação de todos os nós que estão ligados ao nó atual no chord.

Estes 3 atributos são os atributos que são constantemente manipulados e atualizados nas funções e algoritmos a frente explanados, para que assim todo o chord se mantenha sempre atualizado, permitindo comunicação com todo o sistema.

Node.java

Esta classe é usada para criação dos nós, é importante no sentido em que agiliza a criação de nós externos ao peer para que este assim consiga guardar a informação necessario para eventualmente estabelecer contacto com os nós na chord. Assim esta classe possui um construtor que recebendo um ip e uma porta, os guarda e gera o ip do nó. Este id será resultante da encriptação com SHA-1 da string “ip:port” posteriormente convertido para BigInteger tomando sempre valores positivos.

Node.java

```
public class Node implements java.io.Serializable {

   private int port;
   private String address;
   private BigInteger id;

   public Node(String address, int port) {
       this.port = port;
       this.address = address;
       try {
           byte[] sha1 =  Utility.sha1(this.address+":"+this.port);
          // Integer val = Utility.convertToInt( sha1);

           //ByteBuffer bb= ByteBuffer.wrap(sha1);
           //return bb.getInt();

           this.id = new BigInteger(1,sha1);
       } catch (NoSuchAlgorithmException e) {
           e.printStackTrace();
       } catch (UnsupportedEncodingException e) {
           e.printStackTrace();
       }
   }
```

Para além do construtor aqui estão também definidos vários métodos que geram mensagens a criar para operações a realizar na chord reencaminhar a mensagem para o peer correto.

Node.java

```
public Node requestFindSucc(BigInteger msgId, String ip, int port, BigInteger id) {

       //Create socket
       MessageFactory messageFactory = new MessageFactory();
       Message message = messageFactory.findSuccMsg(msgId,ip,port,id);
       Peer.getThreadExecutor().execute(new SendMessagesManager(message, this.address,
this.port));

       return null;
   }

   public Node requestFindPred(BigInteger msgId, String address, int port){
       //Create socket

       MessageFactory messageFactory = new MessageFactory();
       Message message = messageFactory.findPredMsg(msgId,address,port);
       Peer.getThreadExecutor().execute(new SendMessagesManager(message, this.address,
this.port));

       return null;
```

```
    }

    public Node requestFindSuccFinger(BigInteger msgId, String ip, int port, BigInteger
id, int fingerId) {

        //Create socket
        MessageFactory messageFactory = new MessageFactory();
        Message message = messageFactory.findSuccFingerMsg(msgId,ip,port,id,fingerId);
        Peer.getThreadExecutor().execute(new SendMessagesManager(message, this.address,
this.port));

        return null;
    }

    public Node requestNotify(BigInteger msgId, Node node){

        //Create socket
        MessageFactory messageFactory = new MessageFactory();
        Message message =
messageFactory.notifyMsg(msgId,node.getAddress(),node.getPort());
        Peer.getThreadExecutor().execute(new SendMessagesManager(message, this.address,
this.port));

        return null;
    }
```

Peer.java

Diferencia-se do *Node.java* por conter os algoritmos necessários ao chord que serão explorados à frente, mas também pela evocação da função main, criando sempre os peers a entrar no chord ou o peer inicializador.

Criação da chord

Para que futuros peers se consigam conectar ao chord, é sempre necessário que pelo menos um peer esteja a correr. Para que isto acontece desenhamos um comportamento diferente para o primeiro peer a entrar no sistema, este peer irá sempre ser evocado com 3 argumentos, <RMIAccessPoint> <IPAddress> <Port>, para que assim inicialize o chord, não se conectado a nenhum nó.

Quando o peer inicializador é então evocado na função main, que após inicializar a sua thread de leitura para mensagens, irá criar um objeto do tipo *Peer* para inicializadores apenas, correndo o seguinte fragmento de código.

Peer.java

```
   public Peer(String ipAddress, int port) {

        //create a new chord ring
        super(ipAddress, port);
        succNode = this;

        System.out.println("\nCreated with Id: "+this.getNodeId());

        Arrays.fill(fingerTable, this);

        System.out.println("Init Peer - "+this.getAddress()+":"+this.getPort());

        this.printFingerTable();

        threadExecutor.scheduleAtFixedRate( new ChordManager(this), delay, delay,
TimeUnit.SECONDS);
}
```

O que acontece aqui é que estando o peer sozinho a sua fingerTable terá todas a entradas a apontar para si, ele será o seu próprio sucessor e não terá antecessor. Após preencher estas estruturas, colocará a thread de manutenção a correr para que se possa atualizar caso algum peer se junta a este na chord.

Entrada na Chord

Para que um nó entre no chord, será necessário que o peer seja evocado com um ip e uma porta do nó ao qual se irá conectar, para além do seu ip e porta, assemelhando-se portanto a algo deste género “java Peer <RMIAccessPoint> <IPAddress> <Port> <IPAddressToJoin> <PortToJoin>”.

Quando o peer é então evocado correndo a função main, o comportamento tomado será o mesmo que na criação do chord no entanto, quando o objeto Peer é criado é imediatamente processado numa função de join, que o irá conectar ao chord e só depois de conectado irá colocar a sua thread de manutenção.

A função *Join*, representada no trecho de código abaixo irá portanto receber como argumento o nó ao qual o nosso peer irá usar como entrada para o *Chord*.

Peer.java

```
public void join(Node initNode) {
```

```
        succNode = initNode.requestFindSucc(this.getNodeId(),
this.getAddress(),this.getPort(), this.getNodeId());

        //open thread to wait for the response

        try {
            latchJoin.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        predNode = succNode;

        System.out.println(this.getNodeId()+": "+getFinger(0));
        for (int i = 0; i < fingerTable.length; i++) {
            if (fallsBetween(getFinger(i), this.getNodeId(), succNode.getNodeId()))
                fingerTable[i] = succNode;
            else
                fingerTable[i] = this;
        }

    }
```

Posto isto, irá fazer um pedido ao seu ponto de entrada para que lhe informe qual será o seu sucessor, e imediatamente irá bloquear até que tenha um sucessor atribuído para que assim se junte ao sistema. Esta função atribui ao seu antecessor o valor do sucessor descoberto e irá atualizar toda sua finger Table da seguinte forma, caso se numa entrada, o id dessa entrada estiver entre o seu id ou do sucessor colocar o sucessor como caminho na tabela, caso contrário coloca-se a si mesmo.

Quanto à forma como se procuram sucessores, isso é sempre feito na função *findSucc*(abaixo) que após um nó receber um pedido para procura de sucessor para um dado id de nó irá verificar se o seu sucessor é igual a si próprio e retornar-se a si próprio, no entanto caso esta condição se verifique irá também tentar confirmar se o seu id é diferente do que recebeu, porque caso isto aconteça significa que o chord só tinha um peer até ao momento devendo assim portanto guardar como seu sucessor o nó que envio o pedido de procura para sucessor.

Peer.java

```
  public Node closestPrecedNode(BigInteger preservedId) {
        for(int i = fingerTable.length - 1; i >= 0; i--  )
            if(fingerTable[i] != null && fallsBetween(fingerTable[i].getNodeId(),
this.getNodeId(), preservedId))
                return fingerTable[i];
        return this;}
```

Após esta verificação o nó irá verificar se o id pedido se encontra entre si e o seu sucessor, pois caso isso aconteça poderá retornar o seu sucessor, caso não aconteça irá procurar o nó que tem mais próximo do id pedido na sua finger table através de *closestPrecedNode* e caso esse nó mais próximo seja igual ao seu id, significa que o sucessor é ele próprio, retornando se a si mesmo, senão encaminha o pedido de procura de sucessor para esse nó mais próximo, para esse tente procurar e caso encontre comunica ao peer que deu join na corda.

Peer.java

```
public Node findSucc(String address, int port, BigInteger id){

        //case its the same id return itself

        if(succNode.getNodeId().equals(this.getNodeId())){
            if(!id.equals(this.getNodeId())){
                succNode = new Node(address,port);
                predNode = new Node(address,port);
                fingerTable[0] = succNode;
            }

            return this;
        }

        if( fallsBetween(id,this.getNodeId(),succNode.getNodeId())){
            return succNode;
        }else{
            Node newNode = closestPrecedNode(id);

            if(newNode.getNodeId().equals(this.getNodeId()))
                return this;

            return newNode.requestFindSucc(this.getNodeId(),address, port,id);
        }

    }
```

Manutenção da Chord

Para que os valores da finger table, sucessores e antecessores dos nós sejam corretos e estejam constantemente atualizados, há necessidade de fazer manutenção com uma dada frequência para que esses valores sejam atualizados.

Assim recorrem-se a 3 métodos diferentes que ocorrem sempre dentro do período de manutenção, são eles:

- *stabilize()* - O nó irá perguntar ao seu sucessor qual o seu antecessor, decidindo se o seu sucessor irá mudar ou não. Encerra notificando o seu sucessor para este possa atualizar o seu antecessor.

Peer.java

```
 public void stabilize(){
       //get predecessor

       stabilizeX =
succNode.requestFindPred(this.getNodeId(),this.getAddress(),this.getPort());

       try {
           latchStabilize.await();
       } catch (InterruptedException e) {
           e.printStackTrace();
       }

       latchStabilize =  new CountDownLatch(1);

       //check if X falls between (n,successor)
       if(stabilizeX != null
               && ! this.getNodeId().equals(stabilizeX.getNodeId() )
               && ( fallsBetween(stabilizeX.getNodeId(), this.getNodeId(),
succNode.getNodeId()) || this.getNodeId().equals(succNode.getNodeId()) )
       ){
           fingerTable[0] = stabilizeX;
           succNode = stabilizeX;
       }

       succNode.requestNotify(this.getNodeId(), this);
   }
```

- *fixFingers()* - Atualiza as entradas na finger table, percorrendo todos os seus nós fazendo um pedido do sucessor do valor do nó da tabela, através do findSuccFinger, cujo comportamento é igual ao do findSucc. Caso seja encontrado um nó essa entrada será atualizada pelo retorno.

Peer.java

```
 public void fixFingers(){

       for (int i = 1; i < fingerTable.length; i++) {
           Node n = findSuccFinger(this.getAddress(),this.getPort(),getFinger(i),i);

           if(n!= null)
```

```
                updateFinger(n,i);
        }
}
```

- *checkPred()* - Neste método testa-se a conexão com o antecessor, onde caso não seja possível conectar com o seu antecessor este deverá ser colocado a nulo, indicando que um nó saiu da corda.

Peer.java

```
public void checkPred(){

        MessageFactory messageFactory = new MessageFactory();

        Message msg = messageFactory.checkPredMsg();

        Peer.getThreadExecutor().execute(new SendMessagesManager(msg,
predNode.getAddress(), predNode.getPort()));
    }
```

Tolerância a falhas

De modo a que a aplicação permaneça continuamente em execução, mesmo quando ocorrem erros, é necessário serem implementados procedimentos que operem perante o surgimento destes.

Com isto em vista, para além de terem sido implementados os métodos de manutenção do algoritmo Chord que, por si só, corrigem incompatibilidades geradas pela saída ou indisponibilidade de um *peer*, foram desenvolvidos mecanismos, ao nível dos subprotocolos de comunicação, que lidam com as falhas que se prevêem poder ocorrer.

Assim, nos subprotocolos backup e reclaim foi introduzido o *replication degree* que, juntamente com as estratégias que o preservam, previne a ocorrência de falhas no sistema.

O funcionamento deste em cada subprotocolo, ainda que já tenha sido brevemente abordado anteriormente, encontra-se explicado seguidamente.

- Backup

  No subprotocolo de *backup*, para que o sistema se defenda de falhas, é exigido o armazenamento dos pacotes gerados a partir de um ficheiro em tantos *peers* quanto

o número de *replication degree* pedido. Para isso, o peer que pretende guardar um ficheiro envia os pacotes para todos os peers que consegue alcançar e, caso esse número não seja suficiente para garantir a replicação desejada, envia novamente para o seu sucessor que, ao verificar que já possui o pacote, transmite a mensagem para o seu sucessor. Nesta mensagem constam a identificação e os dados do pacote enviado e o endereço e porta do peer que efetuou o pedido, para que qualquer peer que armazene o pacote consiga enviar uma mensagem de confirmação (*STORED*) para o anterior e, no momento em que desejar remover o pacote, notificar o dono do mesmo com uma mensagem *REMOVED* (algo que será visto de seguida, no subprotocolo *reclaim*).

Na eventualidade de não existirem *peers* suficientes para traduzir a replicação desejada na replicação real o peer que iniciou o pedido de *backup* para a transmissão por detetar que recebeu um pedido de armazenamento com início em si próprio.

- Reclaim

  O subprotocolo *reclaim* é aquele que, sempre que acionado, desencadeia um procedimento que visa em manter o grau de replicação desejado. Quando um peer elimina um pacote o primeiro envia uma mensagem para o dono do último. Na receção dessa mensagem é verificado se o grau de replicação real é inferior ao valor desejado e, caso isso se confirme, é iniciado o subprotocolo *backup* que visa em repor o estado anterior.

Concluindo, o *replication degree* previne possíveis falhas visto que não compromete o sistema na totalidade apenas porque um peer se encontra indisponível. Em qualquer momento todas as operações se mantêm em serviço. No entanto, ainda que o sistema mantenha a execução caso algum peer se torne indisponível, é possível que qualquer pedido que ocorra antes do processo de estabilização da *chord* não seja concluído com sucesso. Ainda assim, uma vez que estas situações foram também antecipadas, o projeto desenvolvido garante que o sistema continue em operação nestes momentos.