SDIS 2021

Distributed Backup Service for the Internet

Relatório - 2º Trabalho laboratorial

Turma 7, Grupo 24

Diana Freitas, up201806230
Eduardo Brito, up201806271
Hugo Guimarães, up201806490
Paulo Ribeiro, up201806505

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Visão Global

Neste relatório, será descrito o design e processo de implementação de um serviço de backup descentralizado, que permite gerir um sistema de ficheiros pela Internet. Serão descritas as principais funcionalidades, os protocolos de comunicação e as funcionalidades adicionais, relacionadas com a concorrência, escalabilidade e tolerância a falhas.

O design do serviço de *backup* implementado tem como principal referência o algoritmo *Chord*, criando um sistema totalmente descentralizado que suporta diversos protocolos, como *Backup*, *Restore*, *Delete*, *Reclaim*, e *State*.

Toda a comunicação é feita através de TCP, com a componente de comunicação segura garantida pela *framework* JSSE, fornecida pelas interfaces SSLSockets/SSLServerSockets.

Em termos de escalabilidade, ao nível do design, todas as funcionalidades que tornam o *Chord* altamente escalável foram implementadas. Ao nível da implementação, a escalabilidade e a concorrência foram garantidas usando múltiplas *threads*, *thread-pools* e funções assíncronas de I/O, i.e. Java NIO.

Em relação à tolerância a falhas, diversas situações que geram pontos de falha foram analisadas e a sua solução, devidamente endereçada. Neste sistema, existe a possibilidade de replicar um ficheiro, sendo distribuídas várias cópias pela rede. Existe ainda a tolerância a saídas e entradas de peers, com mecanismos de manutenção da consistência de todo o sistema.

Protocolos

Esta seção destina-se à descrição dos protocolos implementados no nosso projeto, nomeadamente, ***Backup***, ***Restore***, ***Delete***, ***Reclaim*** e ***State.***

Cada peer implementa um objeto da classe **RemoteInterface,** permitindo utilizar a Interface RMI para comunicar com o cliente, utilizando as seguintes operações:

- **Backup**(String filename, int replicationDegree) - Inicia o protocolo de backup do ficheiro fornecido, distribuindo-o pelo número de peers especificado no argumento *replicationDegree.* Caso não existam peers suficientes para satisfazer o *replicationDegree* fornecido, é guardado um *backup* no número máximo de peers possíveis, sendo o *peer* que chama o protocolo capaz de guardar um *backup* do próprio ficheiro.
- **Restore**(String filename) - Restaura o ficheiro com o nome fornecido, criando uma cópia do mesmo, caso seja encontrado um *peer* na rede que possua uma cópia do ficheiro.
- **Delete**(String filename) - Apaga todos os *backups* do ficheiro de todos os *peers* da rede, desde que os mesmos se encontrem conectados à rede.
- **Reclaim**(long diskSpace) - Limita o espaço máximo de armazenamento de um *peer* ao valor fornecido no argumento, eliminando ficheiros do seu sistema caso necessite de libertar espaço.
- **State**() - Retorna ao cliente informação sobre o sistema de armazenamento do *peer,* nomeadamente, o espaço total, o espaço disponível e todos os *backups* que possui de outros ficheiros.

Execução dos protocolos

A comunicação entre os *peers* e o cliente é efetuada através da classe **TestApp**, a qual é executada na *root* do projeto, da seguinte forma:

- java g24.TestApp <PeerID> <Protocol> <Arguments>*

Execução de cada um dos protocolos:

- java g24.TestApp <PeerID> BACKUP <FilePath> <ReplicationDegree>
- java g24.TestApp <PeerID> RESTORE <FilePath>
- java g24.TestApp <PeerID> DELETE <FilePath>
- java g24.TestApp <PeerID> RECLAIM <DiskSpace>
- java g24.TestApp <PeerID> STATE

Backup

Como mencionado anteriormente, este protocolo permite distribuir cópias de um determinado ficheiro pelo número de *peers* da rede, especificado pelo *replication degree* fornecido.
Este protocolo começa por gerar a *hash* do ficheiro, com base no nome, e, de seguida, calcula o identificador do ficheiro, resultante da operação do resto da divisão inteira pelo número máximo de *peers* possíveis da rede. O valor deste identificador é independente do *peer* responsável por executar o protocolo, o que nos permite efetuar o BACKUP em qualquer *peer* ligado à rede, obtendo o mesmo resultado.
De seguida, é lançada à rede a mensagem FINDSUCCESSOR com o identificador gerado, sendo possível obter o primeiro *peer* responsável por guardar uma cópia do ficheiro, para o qual é

enviada a mensagem BACKUP com o *replication degree* fornecido pelo cliente. De seguida, o *replication degree* é decrementado em uma unidade, é calculado o sucessor do *peer* para o qual foi enviado o último BACKUP e o processo repete-se até que o número de *backups* requisitados seja cumprido, ou até que não existam mais *peers* capazes de guardar uma cópia do ficheiro.
Este método permite que cada *peer,* possuindo um valor diferente de *replication degree* para o mesmo ficheiro, tenha uma noção do tamanho e ordem da sequência de *peers* responsáveis por guardar o ficheiro, o qual será útil para múltiplas funcionalidades do projeto, incluindo o protocolo de *Delete* e o sistema de tolerância a falhas.

Desta forma, ao executar o comando:

```
java g24.TestApp peer1 BACKUP test/large_file.jpg 3
```

o ficheiro será guardado em três peers, com o replication degree distribuído da seguinte forma:

- Peer[id=5] - *replication degree* 3
- Peer[id=8] - *replication degree* 2
- Peer[id=12] - *replication degree* 1

Restore

O protocolo *Restore* foi implementado em conformidade com o protocolo de *Backup*, calculando, de igual forma, o primeiro *peer* responsável por guardar uma cópia do ficheiro.
Desta maneira, é possível invocar este protocolo a partir de qualquer *peer* da rede, pois este cálculo apenas se baseia na *hash* obtida a partir do nome do ficheiro, resultando sempre no mesmo identificador.
De seguida, é enviada a esse *peer* a mensagem RESTORE, esperando por uma confirmação. Caso receba o ficheiro pretendido, recupera-o e termina a execução. Caso não receba a confirmação que deseja, percorre a rede através dos sucessores de cada nó, ora até encontrar o ficheiro, ora até ter percorrido toda a rede.

Delete

O protocolo *Delete* também começa por calcular o primeiro *peer* responsável pelo ficheiro, através da *hash* obtida a partir do seu nome, ao qual envia a mensagem DELETE, indicando ao *peer* que, caso possua o ficheiro, o deve apagar, retornando o *replication degree* que possui, que indica quantos *peers* ainda existem com uma cópia do ficheiro.
Desta forma, ao percorrer os sucessores da rede, o protocolo sabe que deve parar de enviar a mensagem DELETE quando receber como resposta o *replication degree* a zero.
De modo a impedir ciclos infinitos, o protocolo é também interrompido quando é detetado que se está a comunicar de volta com o *peer* inicial.

Reclaim

O protocolo *Reclaim* é responsável por alterar a capacidade de armazenamento do *peer*. Quando a nova capacidade é inferior à soma do tamanho dos ficheiros armazenados, o *peer* inicia um processo de remoção de cópias de ficheiros guardados, até que a sua capacidade efetiva respeite o pedido.

Durante o processo de eliminação das cópias, é iniciado um protocolo de *Backup*, na tentativa de manter o *replication degree* no sistema, sendo o ficheiro enviado para o próximo *peer* capaz de armazenar uma cópia.

State

O protocolo *State* acede à storage do *peer*, exibindo, no ecrã, os seguintes atributos do sistema de armazenamento:

- Espaço total;
- Espaço ocupado;
- Espaço livre;
- *Backups* de ficheiros:
  - Identificador
  - *Replication degree*
  - Tamanho do ficheiro

Mensagens

Durante a execução dos protocolos, os *peers* comunicam entre si através do método **sendMessage,** pelo qual enviam as mensagens BACKUP, RESTORE e DELETE. A leitura do pedido é realizada no **MessageReceiver** de cada *peer*, descrito em detalhe na secção relativa à concorrência. Já o processamento do pedido e envio da resposta é responsabilidade dos *handlers* do *package* **protocol**.

Todas as mensagens, enviadas por TCP, esperam uma resposta por parte do *peer* que as recebe, indicando o sucesso ou insucesso da operação, assim como informação adicional necessária para a execução do protocolo.

A mensagem BACKUP contém o identificador do ficheiro, o *replication degree* desejado e o conteúdo do ficheiro, sendo enviada com o seguinte formato:

BACKUP <FileId> <ReplicationDeg> <Body> <CRLF> <CRLF>

Após um *peer* receber a mensagem, tenta guardar o ficheiro recebido, retornando a mensagem "OK" quando o armazena com sucesso ou quando já possui uma cópia do mesmo. Quando não apresenta espaço livre suficiente para armazenar o *peer*, retorna "NOT OK", informando o *peer* que lhe enviou a mensagem que não conseguiu guardar o ficheiro.

```
public class Backup extends Handler {
    private String fileID;
    private byte[] data;
    private Storage storage;
    private int replicationDegree;

    public Backup(String fileID, int replicationDegree, byte[] data, Storage
    storage) {
        this.fileID = fileID;
        this.data = data;
```

```
        this.storage = storage;
        this.replicationDegree = replicationDegree;
    }

    @Override
    public void run() {
        try {
            FileData newFileData = new FileData(this.fileID, this.data,
            this.replicationDegree);
            byte[] message;
            if(this.storage.store(newFileData)) {
                message = ("OK").getBytes();
            }
            else{
                message = ("NOT OK").getBytes();
            }
            out.write(message, 0, message.length);
            out.flush();
            out.close();
            } catch(Exception e) {
                e.printStackTrace();
            }
    }
}
```

Fig. 1: Backup Protocol

A mensagem RESTORE recebe o identificador do ficheiro, possuindo o seguinte formato:

RESTORE <FileId> <CRLF> <CRLF>

Esse identificador permite-lhe verificar se contém uma cópia armazenada. Caso possua essa cópia, envia uma mensagem com o seguinte formato:

OK <CRLF> <CRLF> <DATA>

Sendo a “data” os bytes do ficheiro requisitado.

Caso não possua uma cópia do ficheiro pedido, retorna:

NOT_FOUND <CRLF> <CRLF>

```
public class Restore extends Handler {
    String fileID;
    Storage storage;

    public Restore(String fileID, Storage storage) {
        this.fileID = fileID;
        this.storage = storage;
    }

    @Override
    public void run() {

        try {
            byte[] message;
```

```
            if (this.storage.hasFileStored(this.fileID)) {
                byte[] body = this.storage.read(this.fileID).getData();
                byte[] header = ("OK" + Utils.CRLF + Utils.CRLF).getBytes();
                message = new byte[header.length + body.length];
                System.arraycopy(header, 0, message, 0, header.length);
                System.arraycopy(body, 0, message, header.length, body.length);
            } else {
                message = ("NOT_FOUND" + Utils.CRLF + Utils.CRLF).getBytes();
            }

            out.write(message, 0, message.length);
            out.flush();
            out.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Fig. 2: Restore Protocol

A mensagem DELETE recebe também o identificador do ficheiro e possui o seguinte formato:

DELETE <FileId> <CRLF> <CRLF>

Através deste identificador, é possível verificar se o *peer* possui uma cópia do ficheiro. Caso exista, é eliminada, retornando a mensagem "OK <left_to_notify>", sendo *left_to_notify* o valor de *replication degree* do ficheiro apagado decrementado em uma unidade, informando ao peer que lhe enviou a mensagem quantos peers ainda necessitam de apagar a sua cópia do ficheiro. Caso não possua o ficheiro pedido, envia a mensagem "NOT_FOUND".

```
public class Delete extends Handler {
    String fileID;
    Storage storage;

    public Delete(String fileID, Storage storage) {
        this.fileID = fileID;
        this.storage = storage;
    }

    @Override
    public void run() {

        try {
            byte[] message;
            int leftToNotify = 0;
            if(this.storage.hasFileStored(this.fileID) && (leftToNotify =
this.storage.getFile(fileID).getReplicationDegree() - 1) >= 0 &&
this.storage.removeFileData(fileID)){
                message = ("OK " + leftToNotify).getBytes();
            }
            else{
                message = ("NOT_FOUND").getBytes();
```

```
            }

            out.write(message, 0, message.length);
            out.flush();
            out.close();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Fig. 3: Delete Protocol

Existem ainda mensagens que não estão diretamente relacionadas com os protocolos, que serão descritas em secções mais abaixo, em particular na secção do *Chord* e da Tolerância a Falhas:

A mensagem ONLINE, enviada no método **hasFailed** da classe *Chord*, é utilizada para verificar se um *peer* se encontra ativo na rede:

ONLINE - ONLINE <CRLF><CRLF>

A mensagem NOTIFY P, enviada no método **notifySuccessor** da classe *Chord*, é utilizada para avisar o sucessor do nó atual de que este é o seu predecessor:

NOTIFY - NOTIFY P <IP> <PORT> <CRLF><CRLF>

A mensagem NOTIFY L, enviada no método **notifyLeaving** da classe *Chord*, é utilizada para avisar o sucessor de que tem um novo predecessor:

NOTIFY - NOTIFY L <IP> <PORT> <CRLF><CRLF>

A mensagem NOTIFY S, enviada no método **notifyPredecessor** da classe *Chord,* é utilizada para avisar o predecessor de que tem um novo sucessor direto:

NOTIFY - NOTIFY S <IP> <PORT> <CRLF><CRLF>

A mensagem FINDSUCCESSOR, enviada no método **findSuccessor** da classe Chord, é utilizada para encontrar o sucessor de um nó:

FINDSUCCESSOR - FINDSUCCESSOR <NODEKEY> <CRLF><CRLF>

A mensagem GETPREDECESSOR, enviada no método **notifyPredecessor** da classe *Chord,* é utilizada para obter o predecessor de um dado nó:

GETPREDECESSOR - GETPREDECESSOR <CRLF><CRLF>

A mensagem GETKEYS é utilizada quando um peer se junta ao *Chord Ring*, para obter as *keys* que lhe correspondem:

GETKEYS - GETKEYS <IP> <PORT> <CRLF><CRLF>

A mensagem HASFILE é utilizada para verificar se um dado peer guarda uma cópia do ficheiro identificado pelo ID:

HASFILE- HASFILE <FILEID> <CRLF><CRLF>

Design de Concorrência

De modo a suportar a concorrência no tratamento e realização de pedidos dos vários protocolos e também no acesso ao sistema de ficheiros, foram utilizadas threads, um objeto da classe **ScheduledThreadPoolExecutor**, Java NIO e estruturas de dados que suportam o acesso concorrente (**ConcurrentHashMap**).

Primeiramente, de modo a possibilitar a execução simultânea dos diferentes protocolos, cada instância dos protocolos *Backup*, *Delete*, *Restore* e *Reclaim* é executada numa nova thread, utilizando, para esse efeito, as classes do *package* **handler**.

Para além disso, cada *peer* da rede utiliza uma thread **MessageReceiver** para a receção de mensagens, algumas associadas aos protocolos implementados e outras utilizadas na implementação do *Chord*. No **MessageReceiver** de cada um dos *peers*, é criado um único **SSLServerSocket**, cuja função é ouvir e aceitar os múltiplos pedidos, sendo delegada a responsabilidade de os processar ao **MessageHandler**, através da invocação do método **handle**.

```
public class MessageReceiver implements Runnable {
   private SSLServerSocket socket;
   private MessageHandler handler;
   private Chord chord;

   public MessageReceiver(int port, ScheduledThreadPoolExecutor scheduler, Chord
chord, Storage storage) throws IOException {
       this.chord = chord;
       this.handler = new MessageHandler(scheduler, this.chord, storage);

       // Creating Server Socket
       SSLServerSocketFactory ssf =
           (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
       this.socket = (SSLServerSocket) ssf.createServerSocket(port);
       this.socket.setEnabledCipherSuites(Utils.CYPHER_SUITES);
   }

   @Override
   public void run() {
       while(true) {
           try {
               SSLSocket newSocket = (SSLSocket) this.socket.accept();
               newSocket.startHandshake();
               newSocket.setEnabledCipherSuites(Utils.CYPHER_SUITES);
               this.handler.handle(newSocket);
           } catch(Exception e) { Utils.out("ERROR", e.getMessage());}
       }
   }
}
```

Fig. 4: Cada *peer* utiliza uma thread da classe **MessageReceiver** para receber as mensagens enviadas pelos restantes *peers* da rede.

O método **handle** recebe como argumento o **SSLSocket**, a utilizar para comunicar com o *peer* que enviou a mensagem, e inicia, numa nova *thread,* o **Handler** que vai processar o pedido e gerar a respetiva resposta. Para cada tipo de mensagem, é utilizado o **Handler** com o mesmo nome do *package* **protocol**, gerado no método **prepare**.

```
public class MessageHandler {
    …
    public void handle(SSLSocket socket) throws IOException {
        this.scheduler.execute(this.parse(socket));
    }

    private Handler parse(SSLSocket socket) throws IOException {
        …
        Handler handler = this.prepare(result);
        …
        return handler;
    }

    private Handler prepare(byte[] message) {
        …
        // Call the respective handler
        switch(splitHeader[0]) {
            case "BACKUP":
                return new Backup(splitHeader[1],
                    Integer.parseInt(splitHeader[2]), body, this.storage);
            case "DELETE":
                return new Delete(splitHeader[1], this.storage);
            case "RESTORE":
                return new Restore(splitHeader[1], this.storage);
            case "ONLINE":
                return new Online();
            case "NOTIFY":
                return new Notify(splitHeader[1], splitHeader[2],
                    Integer.parseInt(splitHeader[3]), this.chord);
            case "FINDSUCCESSOR":
                return new FindSuccessor(Integer.parseInt(splitHeader[1]),
                    this.chord);
            case "GETPREDECESSOR":
                return new GetPredecessor(this.chord);
            case "GETKEYS":
                return new GetKeys(this.chord, this.storage, splitHeader[1],
                    Integer.parseInt(splitHeader[2]));
            case "HASFILE":
                return new HasFile(this.storage, splitHeader[1]);
            default:
                System.err.println("Message is not recognized by the parser");
                break;
        } …
    }
}
```

Fig. 5: O método **handle** da classe **MessageHandler** inicia numa *thread* a execução do **Handler** do protocolo correspondente à mensagem recebida, determinado no método **prepare**.

Para a criação das *threads,* foi utilizado um objeto da classe **ScheduledThreadPoolExecutor**, criado na classe **Peer**. A sua utilização, para além de permitir evitar a criação sucessiva de novas *threads*, permitiu agendar a execução de *threads* periódicas, utilizadas, por exemplo, para atualização dos dados armazenados em memória não volátil e também para a implementação do *Chord*, tal como detalhado na secção referente à Escalabilidade.

```
private ScheduledThreadPoolExecutor executor=new
ScheduledThreadPoolExecutor(250);
```

Fig. 6: Declaração e inicialização do **ScheduledThreadPoolExecutor.**

Foi ainda necessário garantir o processamento concorrente dos dados do sistema, em particular o acesso às informações dos ficheiros armazenados. Para isso, foram utilizadas estruturas de dados do tipo **ConcurrentHashMap.** Estas estruturas suportam conjuntos de operações em sequência, assim como em paralelo e com grande volume, sendo desenhadas para serem seguras quando usadas concorrentemente por diferentes *threads*.

```
// Files stored in this peer file system
private ConcurrentHashMap<String, FileKey> storedFiles;
```

Fig. 7: **ConcurrentHashMap** da classe **Storage**, utilizado para guardar as chaves dos ficheiros armazenados no *peer*.

Por fim, para as chamadas ao sistema de ficheiros, necessárias para a escrita e leitura de ficheiros e para atualizar as informações constantes no espaço de armazenamento dos diferentes *peers*, foram utilizados objetos da classe **AsynchronousFileChannel**, eliminando assim as *blocking-calls* que restavam. A sua utilização pode ser verificada nos métodos **read**, **store** e **storeRestored** da classe **Storage** e ainda na classe **AsynchronousStorageUpdater**.

```
// Used by a peer to store a file in non-volatile memory.
public boolean store(FileData file) throws IOException {
    if(this.hasFileStored(file.getFileID())) {
        this.getFile(file.getFileID()).setReplicationDegree(
            file.getReplicationDegree());
        return true;
    }
    if (file.getSize() + this.occupiedSpace > this.totalSpace) return false;

    String fileDir = this.path + "/backup";
    Files.createDirectories(Paths.get(fileDir));
    Path path = Paths.get(fileDir + "/file-" + file.getFileID() + ".ser");
    Set<OpenOption> options = new HashSet<OpenOption>();
    options.add(StandardOpenOption.CREATE);
    options.add(StandardOpenOption.WRITE);

    AsynchronousFileChannel channel = AsynchronousFileChannel.open(path, options,
       this.executor);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(file);
    oos.flush();

    ByteBuffer buffer = ByteBuffer.wrap(baos.toByteArray());
    Future<Integer> operation = channel.write(buffer, 0);
    while (!operation.isDone()){}

    this.storedFiles.put(file.getFileID(), file.getFileKey());
    channel.close();
    oos.close();
    baos.close();
    this.occupiedSpace += file.getSize();
    return true;
}
```

Fig. 8: Exemplo de utilização de um **AsynchronousFileChannel** para a escrita de ficheiros.

JSSE

Com o objetivo de estabelecer uma ligação TCP segura entre os *peers* da rede, foi utilizada a framework *JSSE - Java Secure Socket Connection;* para a troca de mensagens e envio de ficheiros.

Como tal, foi necessário gerar chaves/certificados *keystore* e *truststore* para a autenticação dos *peers,* através do comando *keytool,* e passar a cada *peer* a informação relativa à sua localização no sistema de ficheiros e *password,* através de propriedades do sistema.

```
PROPERTIES = -Djavax.net.ssl.keyStore=server.keys \
             -Djavax.net.ssl.keyStorePassword=sdisg24 \
             -Djavax.net.ssl.trustStore=truststore \
             -Djavax.net.ssl.trustStorePassword=sdisg24 \
peer1:
   java ${PROPERTIES} g24.Peer peer1 0.0.0.0 9050
```

Fig. 9: Invocação de um **Peer** e definição das propriedades do sistema necessárias para a autenticação usando chaves/certificados *keystore* e *truststore.*

O estabelecimento da ligação TCP recorre à utilização de um **SSLServerSocket**, criado no **MessageReceiver** de cada peer e utilizado para receber os pedidos. São também utilizados, após ter sido aceite a conexão, sockets da classe **SSLSocket**, retornados pela função **SSLServerSocket.accept(),** para a troca de dados entre servidor e cliente nos vários **Handlers**, que, tal como referido na secção anterior, são gerados no método **prepare** do **MessageHandler**, e retornados no método **parse**, cuja responsabilidade é ler, em bytes, a mensagem do *socket*.

Do lado do *peer* emissor, são, mais uma vez, utilizados os **SSLSocket** no método **sendMessage** da classe **Chord** para enviar as mensagens a um *peer* e esperar a resposta. Este método recebe o IP e porta do *peer* para o qual seguirá a mensagem, a informação a ser transmitida, particularmente o cabeçalho e os dados, e pode ainda receber um *timeout* a utilizar para evitar que o *peer* fique indefinidamente à espera de uma resposta, bloqueando a leitura apenas durante o tempo estabelecido.

Antes de ser realizada a transferência dos dados, são estabelecidos os parâmetros a utilizar na conexão segura através do protocolo *Handshake*, invocado explicitamente através do método **startHandshake** da classe **SSLSocket**. Para a configuração dos parâmetros, é utilizado o método **setEnabledCipherSuites**, que permite escolher as possíveis combinações de algoritmos criptográficos suportados na ligação.

Por fim, para a transferência de dados entre *peers* através do **SSLSocket,** são utilizados *streams* da classe **DataInputStream** e **DataOutputStream**, para leitura e escrita, respetivamente.

```
public class MessageReceiver implements Runnable {
   private SSLServerSocket socket;
   private MessageHandler handler;
   private Chord chord;

   public MessageReceiver(int port, ScheduledThreadPoolExecutor scheduler, Chord
chord, Storage storage) throws IOException {
```

```
        this.chord = chord;
        this.handler = new MessageHandler(scheduler, this.chord, storage);

        // Creating Server Socket
        SSLServerSocketFactory ssf =
            (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
        this.socket = (SSLServerSocket) ssf.createServerSocket(port);
        this.socket.setEnabledCipherSuites(Utils.CYPHER_SUITES);
    }

    @Override
    public void run() {
        while(true) {
            try {
                SSLSocket newSocket = (SSLSocket) this.socket.accept();
                newSocket.setEnabledCipherSuites(Utils.CYPHER_SUITES);
                newSocket.startHandshake();
                this.handler.handle(newSocket);
            } catch(Exception e) { Utils.out("ERROR", e.getMessage());}
        }
    }
}
```

Fig. 10: Criação do **SSLServerSocket** no **MessageReceiver**, configuração dos *cipher suites* permitidos para a ligação e invocação do protocolo *Handshake*.

```
private Handler parse(SSLSocket socket) throws IOException {
    DataOutputStream out = new DataOutputStream(socket.getOutputStream());
    DataInputStream in = new DataInputStream(socket.getInputStream());
    byte[] response = new byte[Utils.FILE_SIZE + 200];
    byte[] aux = new byte[Utils.FILE_SIZE + 200];
    int bytesRead = 0;
    int counter = 0;

    int total = in.readInt();
    while(counter != total) {
        bytesRead = in.read(response);
        System.arraycopy(response, 0, aux, counter, bytesRead);
        counter += bytesRead;
    }

    byte[] result = new byte[counter];
    System.arraycopy(aux, 0, result, 0, counter);

    Handler handler = this.prepare(result);
    handler.setSocket(socket, out, in);

    return handler;
}
```

Fig. 11: O método **parse** do **MessageHandler** é utilizado para ler as mensagens em bytes do *output stream*. Após a leitura, utiliza o método **prepare** para gerar o **Handler** que vai processar o pedido Este irá utilizar o *socket* e os respetivos *input* e *output stream* para o envio da resposta, definidos com a chamada do método **setSocket**.

```
public byte[] sendMessage(String ip, int port, int timeout, byte[] body,
String...
   headerString) {
   …
   try {
      SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
      SSLSocket socket = (SSLSocket) factory.createSocket(ip, port);
      socket.setEnabledCipherSuites(Utils.CYPHER_SUITES);
      socket.startHandshake();

      if (timeout > 0) {
         socket.setSoTimeout(timeout);
      }

      DataOutputStream out = new DataOutputStream(socket.getOutputStream());

       out.writeInt(message.length);
       out.write(message, 0, message.length);
       out.flush();

       DataInputStream in = new DataInputStream(socket.getInputStream());
       byte[] response = new byte[Utils.FILE_SIZE + 200];
       byte[] aux = new byte[Utils.FILE_SIZE + 200];
       int bytesRead = 0;
       int counter = 0;

       while((bytesRead = in.read(response)) != -1) {
          System.arraycopy(response, 0, aux, counter, bytesRead);
          counter += bytesRead;
       }

       in.close();
       out.close();
       socket.close();

       byte[] result = new byte[counter];
       System.arraycopy(aux, 0, result, 0, counter);

       return result;
    }catch (Exception e) {Utils.out("ERROR", e.getMessage());}

    return new byte[0];
}
```

Fig. 12: Criação de um **SSLSocket**, configuração dos *cipher suites* permitidos para a ligação, invocação do protocolo *Handshake*, envio de uma mensagem e receção da respetiva resposta no método **sendMessage**.

```
public static final String[] CYPHER_SUITES =  new String[]{
   "TLS_RSA_WITH_AES_128_CBC_SHA", "TLS_DHE_RSA_WITH_AES_128_CBC_SHA",
   "TLS_DHE_DSS_WITH_AES_128_CBC_SHA", "TLS_DH_anon_WITH_AES_128_CBC_SHA"
};
```

Fig. 13: *Cipher Suites* suportados. São utilizados nas chamadas ao método **setEnabledCipherSuites**.

Escalabilidade

Para garantir a escalabilidade do sistema desenvolvido, optamos por implementar o algoritmo *Chord*, que se trata de um protocolo *peer-to-peer* utilizado para encontrar dados, altamente escalável e com um baixo custo de comunicação. Para além disto, o seu estado é mantido pela comunicação logarítmica de cada nó com os restantes nós da rede. Consiste numa computação distribuída rápida de uma função *hash* para mapeamento de chaves para os nós, e a sua alta performance deve-se ao facto de cada nó não precisar de armazenar muitas informações sobre os outros nós dentro da rede, conhecendo somente informações necessárias para o seu mapeamento.

Chaves

O algoritmo *Chord* requer que a cada nó esteja associada uma chave. Esta chave é obtida usando uma função de *hashing* (SHA-1), aplicada sobre uma *string* no formato "ip:port", a partir do IP e da porta utilizada pelo *peer* que foi inicializado. Após isto, é calculado o id a ser atribuído ao nó, que é um número compreendido entre 0 e 2^m-1, resultante da *hash* obtida, sendo "m" o número de *bits* usados para o id de cada nó.

Anel do Chord

Para que um nó consiga alcançar qualquer outro na rede, será necessário armazenar o seu predecessor, sucessor e uma tabela designada "*finger table*". O conjunto de todos os nós da rede pode ser representado num círculo que ilustra bem a sua estrutura, que se designa anel do *Chord* (*Chord Ring*). Cada nó é armazenado na classe **Identifier**, que contém os seus dados (IP, porta e id), o predecessor e o sucessor.

- **predecessor:** nó na posição imediatamente anterior à do nó atual;
- **sucessor:** nó na posição imediatamente posterior à do nó atual;
- **fingerTable**: conjunto de nós conectados ao nó atual, que permitem a procura logarítmica na rede, aumentando a performance do algoritmo;

Funções periódicas

É necessária a invocação periódica de algumas funções, que permitem manter o estado da rede atualizado. A configuração destas atualizações é realizada no método **initialize** da classe **Peer**, chamada quando o *peer* é inicializado, com recurso a um **ScheduledThreadPoolExecutor**, que permite definir o *delay* inicial antes de estas serem chamadas pela primeira vez, e o *delay* entre invocações dessas funções, que em todas elas é de 500 milissegundos.

```
private void initialize(int port) throws IOException {
   this.storage = new Storage(this.chord.getId(), this.executor);
   this.executor.scheduleWithFixedDelay(new AsyncStorageUpdater(this.storage),
      5000, 5000, TimeUnit.MILLISECONDS);
   Runtime.getRuntime().addShutdownHook(new Thread(
      new AsyncStorageUpdater(this.storage)));
   this.receiver = new MessageReceiver(port, this.executor, this.chord,
      this.storage);
   this.executor.execute(this.receiver);
   Runtime.getRuntime().addShutdownHook(new Thread(() -> this.notifyLeaving()));
   this.executor.scheduleWithFixedDelay(new Thread(() ->
      this.chord.checkPredecessor()), 1000, 500, TimeUnit.MILLISECONDS);
   this.executor.scheduleWithFixedDelay(new Thread(() ->
this.chord.fixFingers()),
      1000, 500, TimeUnit.MILLISECONDS);
```

```
   this.executor.scheduleWithFixedDelay(new Thread(() ->
this.chord.getSummary()),
      1000, 500, TimeUnit.MILLISECONDS);
   this.executor.scheduleWithFixedDelay(new Thread(() ->
this.chord.stabilize()),
      1000, 500, TimeUnit.MILLISECONDS);
   this.executor.scheduleWithFixedDelay(new Thread(() ->
      this.chord.checkSuccessor()), 3000, 500, TimeUnit.MILLISECONDS)
}
```

Fig. 14: Configuração das funções periódicas.

1. Finger Table

Para aumentar a sua eficiência, o algoritmo recorre à *finger table* para efetuar a procura de nós na rede em tempo logarítmico (log(N), em que N é o número total de nós ativos). Cada nó apresenta uma tabela com **m** entradas, sendo que **m** é o número de bits do id de cada nó. A atualização da *finger table* é feita no método **fixFingers** da classe *Chord*. A cada invocação, atualiza uma dada posição da tabela, e quando chega ao fim da tabela volta à segunda entrada, pois a primeira, que corresponde ao sucessor do nó atual, é atualizada na função **stabilize**, que descreveremos no ponto 3. Assim, a procura de um dado nó na rede tem uma complexidade temporal de O(log N).

```
public void fixFingers(){
   try {
      Utils.log("PERIODICALLY", "FIX FINGERS");
      this.next++;
      if (this.next > Utils.m)
         this.next = 2;
      Identifier finger = findSuccessor(this.id.getNext(this.next));
      this.fingerTable.put(this.next, finger);
      Utils.log("FIX FINGERS", "PUT ( " + this.next + " , " + finger.toString()
+ " )");
   }
   catch(Exception e){
      e.printStackTrace();
   }
}
```

Fig. 15: Atualização da *finger table*.

2. Find Sucessor

Esta função trata-se da principal operação deste protocolo, que consiste em encontrar o nó em que uma dada chave deverá ser armazenada. Caso o valor dessa chave pertença ao intervalo]id,sucessor], então o nó a retornar será o sucessor do nó atual. Caso contrário, o nó atual deverá procurar na *finger table* o nó cujo id precede essa chave com maior proximidade, e invocar o **findPredecessor** dessa chave a partir desse nó. Neste último caso, será necessário enviar uma mensagem FINDSUCCESSOR a esse nó, que terá como resposta o nó correspondente àquela chave.

```
public Identifier findSuccessor(Identifier nextNode, Identifier newNode) {
   byte[] response = sendMessage(nextNode.getIp(), nextNode.getPort(), 3000,
null,
      "FINDSUCCESSOR", newNode.toString());
```

```
   String s = new String(response);

   if (s.equals("NOT_FOUND") || response.length == 0)
      return this.id;

   String[] splitResponse = s.split(" ");
   return new Identifier(splitResponse[0], Integer.parseInt(splitResponse[1]));
}

public Identifier findSuccessor(Identifier newNode) {
   Identifier successor = this.id.getSuccessor();
   if (newNode.between(this.id, successor) || newNode.equals(successor)) {
      return successor;
   } else {
      Identifier nextNode = this.closestPrecedingNode(newNode);
      if (nextNode.equals(this.id))
         return this.id;
      return this.findSuccessor(nextNode, newNode);
   }
}
```

Fig. 16: Procura do *peer* responsável por uma dada key.

3. Stabilize

Para que as operações de pesquisa na rede funcionem, é preciso que os sucessores de cada nó estejam atualizados. Isto é garantido pela função **stabilize** da classe *Chord*, que começa por perguntar ao seu sucessor qual é o seu predecessor (método **getPredecessor**), através do envio de uma mensagem GETPREDECESSOR, recebendo o IP e a porta deste. Com base na resposta, verifica se ele próprio continua a ser o predecessor do seu sucessor ou se existe algum nó entre ambos. Neste segundo caso, o nó atual deverá então atualizar o seu sucessor, que passará a ser o nó retornado pelo **getPredecessor** do sucessor, e alterar a primeira entrada da finger table. Por último, notifica o seu sucessor através da mensagem NOTIFY P, para que este possa atualizar o seu predecessor.

```
public void stabilize() {
   Utils.log("PERIODICALLY", "STABILIZE");
   Identifier x = this.getPredecessor(this.id.getSuccessor());
   if (!x.equals(new Identifier()) && x.between(this.id,
this.id.getSuccessor()))
   {
      this.id.setSuccessor(x);
      this.fingerTable.put(1, x);
      Utils.log("STABILIZE","NEW SUCCESSOR " +
this.id.getSuccessor().toString());
   }
   this.notifySuccessor();
}
```

Fig. 17: Confirmação periódica do sucessor

4. Check Successor / Check Predecessor

Uma outra função periódica é a **checkSuccessor** da classe *Chord*, que atualiza a lista **successorList**, sobre a qual falaremos na secção sobre tolerância a falhas. Isto é feito através do

envio de uma mensagem ONLINE que verifica se o nó sucessor está ainda ativo na rede, no método **hasFailed**.

```
public boolean hasFailed(Identifier node) {
   byte[] response = new byte[0];
   if (!node.equals(new Identifier()))
      response = sendMessage(node.getIp(), node.getPort(), 500, null, "ONLINE");
      return response.length == 0;
   }

public void checkSuccessor() {
   Utils.log("PERIODICALLY", "CHECK SUCCESSOR");
   if(!this.id.getSuccessor().equals(this.id)&&
this.hasFailed(this.id.getSuccessor())) {
      this.id.setSuccessor(this.successorList.get(0));
      Utils.log("CHECK SUCCESSOR", "HAS FAILED");
   }
   Identifier successor = this.id.getSuccessor();

   for (int i = 0; i < Utils.m - 1; i++) {
      try {
         successor = findSuccessor(new Identifier(successor.getId() + 1));
         this.successorList.put(i, successor);
      } catch(Exception e){
         i--;
         Utils.out("CAUGHT LEAVING", successor.toString());
      }
   }
   Utils.log("SUCCESSOR LIST", this.successorList.toString());
}
```

Fig. 18: Atualização periódica da lista de sucessores

O mesmo é feito para verificar se o predecessor do nó atual continua ativo, na função **checkPredecessor** dessa mesma classe.

```
public void checkPredecessor() {
   Utils.log("PERIODICALLY", "CHECK PREDECESSOR");
   if (this.hasFailed(this.id.getPredecessor())) {
      this.id.setPredecessor(new Identifier());
      Utils.log("CHECK PREDECESSOR", "HAS FAILED");
   }
}
```

Fig. 19: Confirmação periódica do predecessor

Consistência e Tolerância a falhas

Com o objetivo de evitar possíveis pontos de falha, sendo o nosso sistema descentralizado e desenhado com base no algoritmo *Chord*, foram implementados alguns dos aspetos que garantem a consistência e a tolerância a falhas comuns, assegurando o funcionamento normal da rede e a capacidade de atender, a qualquer momento, qualquer pedido de clientes.

Réplicas dos ficheiros

A primeira funcionalidade que assegura a tolerância a falhas no acesso a ficheiros é a possibilidade de cada ficheiro ser replicado e distribuído pela rede, conforme o *replication degree* especificado pela pessoa que pede o *backup.* Esta funcionalidade assegura a existência do ficheiro em pontos diferentes do sistema, com uma particularidade relevante que é a garantia de consistência na localização do ficheiro, permitindo não só ser replicado, mas também ser localizado com a mesma rapidez com que se procura um sucessor, através da função **findSuccessor** do algoritmo *Chord*. Para isto ser possível, tal como foi descrito anteriormente, o protocolo *Backup* gera uma chave para o ficheiro, que é mapeada para um identificador. Este id existe na rede como se de um nó se tratasse, podendo ser usado para descobrir a sua localização, ou ser entregue para ser armazenado, como pertencente ao intervalo de chaves de um dado peer. Isto garante que qualquer ficheiro possa ser guardado e localizado na rede, independentemente das máquinas que estão ligadas e também da máquina à qual se acede para comunicar com o sistema, já que qualquer nó com capacidades comunicativas consegue, necessitando apenas do nome do ficheiro, gerar a *hash* e o identificador correspondente e, com isso, pesquisar na rede os *peers* responsáveis por guardar o ficheiro.

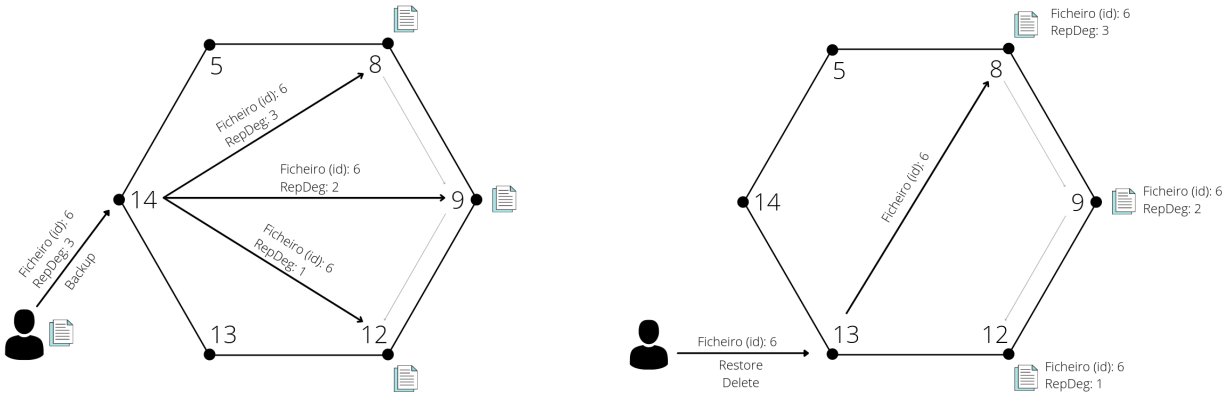

Fig. 20: Todo o processo de Backup garante a replicação distribuída do ficheiro pelo sistema, tornando o processo independente do *peer* ao qual é feito o pedido. Os *peers* responsáveis por guardar as réplicas obedecem à ordem do intervalo onde o identificador do ficheiro se insere. Na imagem, é atribuído ao ficheiro o id 6, o que torna o nó 8 o seu sucessor imediato. Este primeiro nó guarda o ficheiro com *replication degree* 3, o seu sucessor, com *replication degree* 2 e por aí em diante. Este processo permite restaurar, ou eliminar, o ficheiro com um simples *lookup*, além de permitir manter sempre a consistência de todo o sistema, como explicado de seguida.

Saídas e entradas de peers

Uma possível falha do sistema, a um nível estrutural, é a saída de *peers* e a consequente interrupção do anel de comunicação. Tendo isto em conta, a nossa abordagem focou-se em duas situações distintas. A primeira situação, representada pela possibilidade de um dado *peer* ter a intenção explícita de sair do sistema, apelidada de *Graceful Exit*, foi resolvida introduzindo um mecanismo de *Shutdown* do processo e um conjunto de novas mensagens que são enviadas apenas nesta situação. Resumidamente, quando um dado *peer* deseja sair da rede, deve notificar o seu sucessor da sua saída, indicando-lhe o contacto do seu novo predecessor, bem como o seu predecessor, avisando-o, de que o sucessor mudou. Esta funcionalidade permite reatar o anel e manter a rede estabilizada. Junto com esta ação, o *peer* é obrigado, também, a transferir as chaves que possui, iniciando o protocolo de *Backup* a partir do seu sucessor, e contabilizando o número de réplicas com base no *replication degree* que lhe foi atribuído inicialmente. Esta ação garante e permite manter toda a consistência referida no ponto anterior.

```
public void notifyLeaving() {
   Identifier id = this.chord.getId();
   Identifier successor = id.getSuccessor();
   Identifier predecessor = id.getPredecessor();

   this.chord.sendMessage(successor.getIp(), successor.getPort(),
      500, null, "NOTIFY", "L", predecessor.getIp(),
      Integer.toString(predecessor.getPort()));

   this.chord.notifyPredecessor();

   ConcurrentHashMap<String, FileKey> storedFiles =
this.storage.getStoredFiles();

   for(String key : storedFiles.keySet()) {
      FileData fileData;
      try {
         fileData = this.storage.read(key);

         new BackupHandler(this.chord, this.storage, fileData,successor,
            fileData.getReplicationDegree()).run();

      } catch (ClassNotFoundException | IOException e) {
         e.printStackTrace();
      }
   }
}
```

Fig. 21: O método **notifyLeaving** é registado como **ShutdownHook** do processo do *peer*, aquando da sua criação, e chamado quando o peer termina controladamente a sua execução. Este método fica responsável por enviar a referida mensagem para o sucessor, indicando-lhe que deve atualizar o seu predecessor, e para o predecessor, indicando-lhe a existência de um novo sucessor. Além disso, este *peer* inicia, também, o *Backup* das chaves que mantém, começando no seu sucessor imediato, permitindo manter a suposta sequência de réplicas no sistema.

No caso de um *peer* sair do sistema abruptamente e sem capacidade de notificar nenhum outro elemento, existe outra funcionalidade que garante a continuidade do funcionamento do sistema. Esta funcionalidade resume-se à existência de uma lista de sucessores diretos, guardada por cada *peer*, sempre atualizada e com funcionamento semelhante ao da *finger table*. Esta lista, de tamanho **m**, permite uma tolerância a falhas de até **m** *peers* seguidos, restabelecendo a ligação e estabelecendo o contacto com o próximo sucessor, sempre que detetar a falha, sem aviso, do seu sucessor direto.

```
public void checkSuccessor() {
   Utils.log("PERIODICALLY", "CHECK SUCCESSOR");
   if (!this.id.getSuccessor().equals(this.id) &&
      this.hasFailed(this.id.getSuccessor())) {
      this.id.setSuccessor(this.successorList.get(0));
      Utils.log("CHECK SUCCESSOR", "HAS FAILED");
   }

   Identifier successor = this.id.getSuccessor();
   for (int i = 0; i < Utils.m - 1; i++) {
      try{
         successor = findSuccessor(new Identifier(successor.getId() + 1));
         this.successorList.put(i, successor);
       } catch(Exception e){
          i--;
       }
   }
   Utils.log("SUCCESSOR LIST", this.successorList.toString());
}
```

Fig. 22: O método **checkSuccessor** funciona de maneira semelhante ao método **checkPredecessor**, verificando periodicamente se o sucessor falhou, substituindo-o e mantendo a lista de sucessores atualizada com os peers consecutivos que se encontram ligados.

No caso de entradas de novos *peers*, o cuidado a ter está relacionado com a manutenção da consistência de toda a informação, para que os pressupostos anteriores relativos aos vários protocolos sejam assegurados e a rede continue a funcionar corretamente. A cada nova entrada, as chaves correspondentes ao intervalo de identificadores pertencente ao novo *peer* são-lhe enviadas para serem armazenadas. Assim que o *peer* se estabelecer no seu verdadeiro lugar, pede ao seu sucessor as chaves que lhe pertencem, ou seja, com identificadores pertencentes ao seu intervalo circular, ou chaves que venham de uma sequência anterior de réplicas, com um dado *replication degree*. Estas chaves são transferidas e atualizadas, garantindo a consistência e os pressupostos estabelecidos para a sua localização na rede inteira. Depois deste processo terminar, o sistema pode ser percecionado como tendo apenas mais um *peer*, porque tudo o resto continua consistentemente no seu devido lugar.

```
public void moveKeys(Identifier successor) {
    byte[] response = sendMessage(successor.getIp(), successor.getPort(), 3000,
       null, "GETKEYS", this.id.getIp(), Integer.toString(this.id.getPort()));
}

public class GetKeys extends Handler {
...
for (String fileId : this.storage.getStoredFiles().keySet()) {
    FileData fileData = this.storage.read(fileId);
```

```
    Identifier fileKey = new Identifier(Utils.generateHash(fileId));
    Identifier predecessor = this.chord.getId().getPredecessor();
    boolean toSend = false;
    if(fileKey.between(predecessor, this.node) || fileKey.equals(this.node)) {
          toSend = true;
    }
    else {
          Utils.out("GETKEYS","ASKING predecessor " + fileKey.toString());
          byte[] response = this.chord.sendMessage(predecessor.getIp(),
                  predecessor.getPort(), 1000, null,
                    "HASFILE", fileData.getFileID());
         String s = new String(response);
         if (s.equals("OK")) {
            toSend = true;
         }
     }
...
```

Fig. 23: O método **moveKeys** é executado pelo novo *peer*, pedindo as chaves ao seu sucessor. O sucessor tem o trabalho de percorrer as chaves que tem armazenadas e verificar quais pertencem ao novo *peer*, procedendo, depois, ao seu envio.

Bibliografia

- “Chord: A Scalable Peer-To-Peer Lookup Protocol for Internet Applications: IEEE/ACM Transactions on Networking: Vol 11, No 1,” *IEEE/ACM Transactions on Networking (TON)*, 2021 <https://dl.acm.org/doi/pdf/10.1109/TNET.2002.808407?download=true> [accessed 31 May 2021]
- “SDIS: JSSE,” *Fe.up.pt*, 2008 <https://web.fe.up.pt/~pfs/aulas/sd2021/doc/jsse_intro_en.html> [accessed 31 May 2021]
- “SDIS 2020/2021: Project -- Distributed Backup Service for the Internet,” *Fe.up.pt*, 2020 <https://web.fe.up.pt/~pfs/aulas/sd2021/projs/proj2/proj2.html > [accessed 31 May 2021]
- “SDIS 2017/2018: Hints for Concurrency in Project 1,” *Fe.up.pt*, 2017 <https://web.fe.up.pt/~pfs/aulas/sd2021/projs/proj1/concurrency_hints.html> [accessed 31 May 2021]
- Souto, Pedro, *Name Resolution in Flat Name Spaces Distributed Hash Tables (DHTs)* (, 2021) <https://web.fe.up.pt/~pfs/aulas/sd2021/at/9dhts.pdf>
- Stoica, Ion, Robert Morris, David Karger, M Kaashoek, and Hari Balakrishnan, *Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications* () <https://people.csail.mit.edu/karger/Papers/chord.pdf> [accessed 31 May 2021]
- Santos, Lucas, “Chord: Um Protocolo de Pesquisa Peer-To-Peer Escalável Para Aplicativos de Internet,” *Medium* (Medium, 2017) <https://medium.com/@lucascodejs/chord-um-protocolo-de-pesquisa-peer-to-peer-escal%C3%A1vel-para-aplicativos-de-internet-7c2354469701> [accessed 30 May 2021]
- “Java Secure Socket Extension (JSSE) Reference Guide,” *Oracle Help Center* (Security Developer’s Guide, 2018) <https://docs.oracle.com/en/java/javase/16/security/java-secure-socket-extension-jsse-reference-guide.html#GUID-F069F4ED-DF2C-4B3B-90FB-F89E700CF21A> [accessed 31 May 2021]