U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems

Project 2 - Distributed Backup Service for the Internet

Turma: 4 Grupo: 25

2 de junho de 2021

António Cadilha da Cunha Bezerra **up201806854**@fe.up.pt
Gonçalo de Batalhão Alves **up201806451**@fe.up.pt
Inês Oliveira e Silva **up201806385**@fe.up.pt
Pedro Jorge Fonseca Seixas **up201806227**@fe.up.pt

Overview **3**

Protocols **4**
- RMI 4
- TCP 5
- Chord 7
- Backup 8
- Restore 9
- Delete 10
- Reclaim 11
- State 12

Concurrency Design **13**
- Threads 13
- CHORD 14
- Java NIO 15
- ConcurrentHashMap 16

JSSE **17**

Scalability **19**

Fault-tolerance **20**

Overview

The goal of this project was to develop a peer-to-peer distributed backup service for the Internet, which supports the backup, restore and deletion of files. As in the first project, our implementation also allows the reclaiming of a peer's used storage space, so that the participants in the service can retain total control over their own storage.

The distributed backup service we developed implements all the additional features specified in the project description, therefore raising the ceiling of our grade to 20 points.

To assure secure communication, we worked with the Java Secure Socket Extension (JSSE), using the SSLEngine interface, which is more flexible than SSLSockets, allowing for higher concurrency.

Regarding scalability, this issue was addressed both at design level - since we chose to implement a totally distributed design, replicating full files and using the Chord protocol to locate a file's replicas - and at implementation level - by using Java NIO and thread-pools.

To address the fault-tolerance issue, a replication degree is specified when a file is backed up, so that the file has the specified number of replicas, each stored in a different peer. Moreover, to ensure that as little data as possible is lost if a peer crashes and then joins the service again, its information is periodically written to storage.

Protocols

The following command is used to run a peer:

```
java backupservice.peer.Peer <access_point> <host_name> <host_port> [-b]
```

Where each argument means:

- **access_point** - name of the peer's remote interface in the RMI registry.
- **host_name** - IP address of the Chord ring's boot peer (or the peer's own IP address, if -b option is used).
- **host_port** - port number of the Chord ring's boot peer (or the peer's own port number, if -b option is used).
- **-b** - option to indicate this peer is the boot peer and needs to initiate the Chord ring.

The following command is used to run the client application:

```
java backupservice.client.TestApp <peer_ap> <sub_protocol> <opnd_1> <opnd_2>
```

Where each argument means:

- **peer_ap** - the RMI access point of the initiator peer of the desired protocol.
- **sub_protocol** - 'BACKUP', 'RESTORE', 'DELETE', 'RECLAIM' or 'STATE'.
- **opnd_1** - in the case of backup, restore and delete, the file to apply the specified protocol to; in the case of reclaim, the peer's new storage space.
- **opnd_2** - in the case of backup, the replication degree.

Note: The storage of a peer is created, when initialized, in the working directory. The storage contains two folders "files" and "chunks", where the restored files are saved and where the files are backed up, respectively.

RMI

In order for the client to communicate with the peers, the **Peer** class implements the **RMI** interface. When running a client, depending on the *sub_protocol* argument, one of the following functions is called:

- **backup(String filePath, int replicationDegree)** - initiates the backup protocol for the specified file, creating the number of replicas given in *replicationDegree.*
- **restore(String filePath)** - initiates the restore protocol for the specified file, whose backup was initiated by the same peer receiving this instruction.
- **delete(String filePath)** - initiates the delete protocol for the specified file, whose backup was initiated by the same peer receiving this instruction.
- **reclaim(int diskSpace)** - initiates the reclaim protocol, setting the peer's storage space to the value specified in *diskSpace* (deleting stored files as needed).
- **state()** - retrieves the peer's chord information, owned and backed up files and storage information.

TCP

Peer-to-peer communication is implemented using TCP, using the SSLEngine class to take advantage of JSSE secure communication.

When a peer receives a message as a *byte[]*, the **Message** class' *parseMessage()* function is called, returning an object that extends the **Message** class and is specific to that message type. This object is passed to a **MessageReceiver** object, a **Runnable** class that calls the message's *handleMessage()* method in order to trigger the desired actions to deal with the received message (and send a response message back if needed).

Each message has the following structure:

```
<key> <messageType> [<fileID>] [<fileKey>]<CRLF><CRLF>[<body>]
```

Where:

- **key** - Sender CHORD key
- **fileID** - The hashed file ID
- **fileKey** - Chord key associated to the file
- **body** - The message body

The messages used by the protocols are the following:

- **PUTFILE** - `<key> PUTFILE <fileID> <fileKey> <CRLF><CRLF><body>`
  The body of this message has the form: `<owner> <fileSize>`, where **owner** refers to the peer who owns this file, and that initially backed it up, and the **fileSize** is the size of the file in bytes.
  This kind of message is sent to start the transfer of a file to another peer, whether for the backup protocol or for the reclaim protocol - the second referring to when a peer needs to delete a file and tries to transfer it to another peer before deleting it.
  When sending this message there are 3 types of message expected to be received as response: **ReceivedMessage**, if a peer already has the file stored; **NoSpaceMessage**, if the peer has no space available to store this file; **ReadingMessage**, if the peer has enough space to store the file and is ready to receive it.

- **READING-BACKUP** - `<key> READING-BACKUP <fileID> <CRLF><CRLF>`
  This message is sent by a peer, when it is ready to read a file to back it up.
  When a peer receives this message, it will load the requested file and send it to the sender.

- **READING-RESTORE** - `<key> READING-RESTORE <fileID> <CRLF><CRLF>`
  This message has the same functionality as **READING-BACKUP**. It is sent by a peer, when it is ready to read a file.
  When a peer receives this message, it will load the requested file from the peer storage and send it to the sender.

- **STORED -** `<key> STORED <fileID> <fileKey> <CRLF><CRLF>`
  This message is sent by a peer that stored a file to the peer that initiated that file's backup protocol, creating a **StoredMessage** object.
  When a peer receives this message, it increases its perceived replication degree for the specified file, saving the fileKey received to posteriorly use it to restore the file, if needed.

- **GETFILE -** `<key> GETFILE <fileID> <CRLF><CRLF>`
  This message is sent by the restore protocol initiator peer.
  When a peer receives this message, it creates a **GetFileMessage** and, if the peer that received it has the specified file backed up, it sends back a **FileMessage** with the requested contents.

- **FILE -** `<key> FILE <fileID> <CRLF><CRLF>`
  This message is sent by a peer that has the specified file stored to the peer that requested it, creating a **FileMessage**.
  When a peer receives this message, it will start the restore process by sending a **ReadingMessage**.

- **DELETE -** `<key> DELETE <fileID> <CRLF><CRLF>`
  This message is sent by the delete protocol initiator peer to the peers that are storing replicas of the specified file.
  When a peer receives this message, it creates a **DeleteMessage** and the file is removed from the peer's storage.

- **RECEIVED -** `<key> RECEIVED <fileID> <CRLF><CRLF>`
  This message is sent by a peer when it: has already backed up the requested file; has just finished backing up a file; has just finished restoring a file.
  When a peer receives this message, it will close the connection between the two peers.

- **REMOVED -** `<key> REMOVED <fileID> <fileKey> <CRLF><CRLF>`
  This message is sent by the peer executing the reclaim protocol when it eliminates a file from storage to free up space.
  When a peer receives this message, it decreases its perceived replication degree for the file, if it is the owner of the specified file.

- **NO-SPACE -** `<key> NO-SPACE <fileID> <fileKey> <CRLF><CRLF>`
  This message is sent by a peer when it has received a request to store a file and does not have enough space.
  When a peer receives this message, if it is the original owner of the file, it will decrease the replication degree of the file. If not, it will simply terminate the connection.

Chord

- **JOIN -** `<address>:-1 JOIN <CRLF><CRLF>`
  This message is sent to the **boot peer** by a peer that wants to connect to a CHORD ring. When the boot peer receives this message, it will create a ring ID for this new peer and send it back to the requesting peer.

- **LOOKUP-PRED -** `<key> LOOKUP-PRED <CRLF><CRLF>`
  This message is sent by a peer when it requests the predecessor of a peer. When received, the peer will respond with its predecessor peer.

- **LOOKUP-SUCC -** `<key> LOOKUP-PRED <CRLF><CRLF><search>`
  This message is sent by a peer when it requests a peer to find the successor of the **search** key. When received, a peer will begin the lookup process to get that key's successor. If it finds the successor, it will respond with a **SUCCESSOR** message, else it will send a new **LOOKUP-SUCC** message to the closest preceding peer.

- **NEWNODE -** `<key> NEW-NODE <CRLF><CRLF><id> <succ>`
  This message is sent by the boot peer, to a peer who wants to join the CHORD ring. When received, the joining peer will set its CHORD ID to the received ID and initialize its **Finger Table** with the given successor.

- **NOTIFY -** `<key> NOTIFY <CRLF><CRLF>`
  This message is sent by a peer to notify another peer that it may be its predecessor. When received, the peer will check if its predecessor is null or if the potential predecessor is between its current predecessor and its own key, setting the potential predecessor as its true predecessor.

- **PREDECESSOR -** `<key> PREDECESSOR <CRLF><CRLF><predecessor>`
  This message is sent in response to the **LOOKUP-PRED** message, containing the predecessor of a peer.

- **REPLACE-PRED -** `<key> REPLACE-PRED <CRLF><CRLF><predecessor>`
  This message is sent by a peer who is leaving, to its successor, telling him to replace its predecessor with the leaving peer's predecessor. When received, a peer will replace its predecessor with the new predecessor.

- **REPLACE-SUCC -** `<key> REPLACE-SUCC <CRLF><CRLF><successor>`
  This message is sent by a peer who is leaving, to its predecessor, telling him to replace the leaving peer's entry with the leaving peer's successor. When received, a peer will replace all occurrences of the leaving peer, in the finger table, with the new successor.

- **SUCCESSOR -** `<key> SUCCESSOR <CRLF><CRLF><successor>`
  This message is sent in response to the **LOOKUP-SUCC** message, containing the successor of a peer.

Backup

The backup protocol, as in the previous project, stores a file in *n* peers (n being the replication degree). First, the peer hashes the file name and verifies if there wasn't a previous backup of this file. It then generates a random set of CHORD keys, higher than the replication degree assuming that not every key will lead to a different peer successor, representing the potential peers that will store the file. For each generated key, a PutFileMessage is sent to start the backup.

A peer that receives this message will firstly check if it hasn't the file already in storage, in which case it ends the backup process right away with a ReceivedMessage. The peer will then try to reserve space to store the file and if it doesn't have enough space available, it will send a NoSpaceMessage and will end the backup process for that peer. If it can reserve space for the file, the peer will then send a ReadingMessage, to signal that it is ready to receive the file. It then reads the data incoming from the server, storing in the file, and sends a ReceivedMessage, to signal the end of the connection.

From the perspective of the "owner peer": when it receives a NoSpaceMessage or ReceivedMessage, it will terminate the connection; when it receives the ReadingMessage, it will send the contents of the file to a peer.

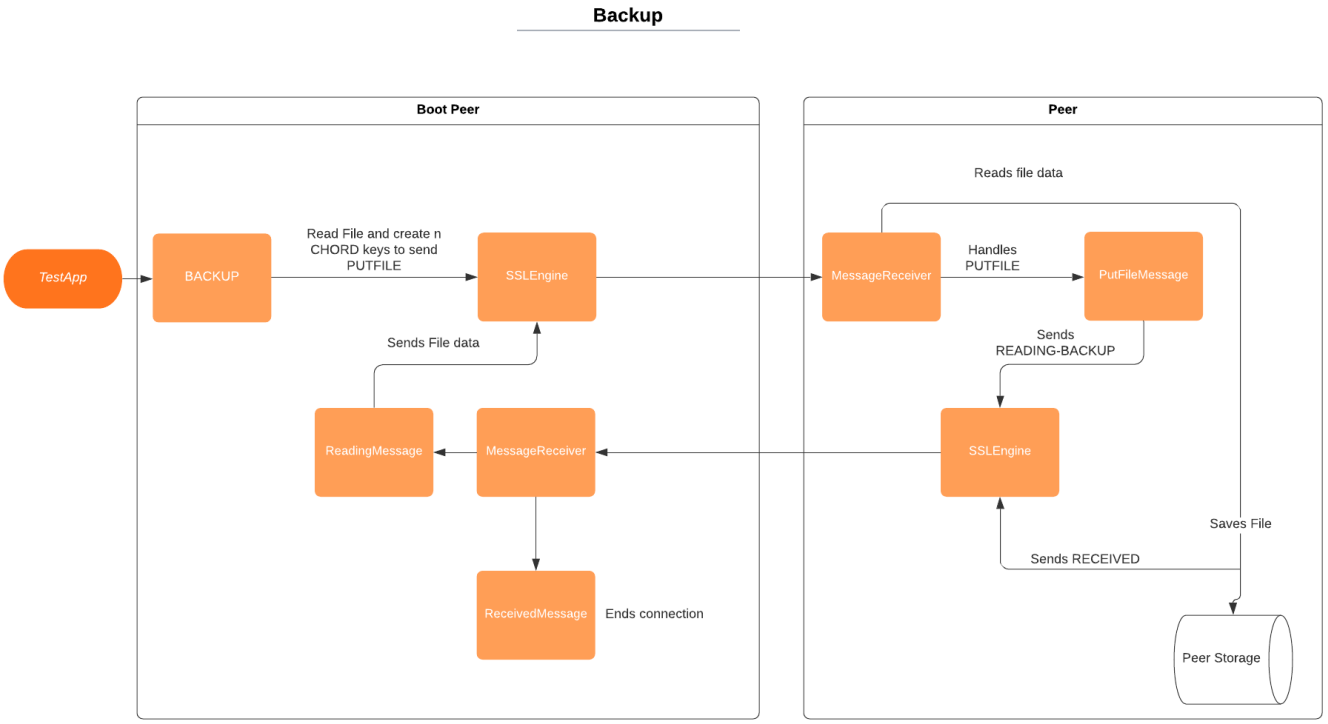


Fig 1. Backup UML diagram

Restore

The restore protocol, as in the previous project, requests a file from a peer. First, the peer retrieves the file's generated CHORD keys, representing the peers that stored the file. A **GetFileMessage** is sent to the successor of each key, one at the time, until one of them returns a **FileMessage**, which will mean that it has the file ready to be restored. Usually the first key is enough, it only needs to check other keys if the first peer who had the file goes down unexpectedly and is not able to send a **RemovedMessage** in order to update the stored keys table.

A peer that receives this message will firstly check if it has the file. If it doesn't, it ends the restore process right away. Otherwise, it will respond with a FileMessage. The peer will then receive a ReadingMessage, signalling that it can send the file. It then writes the file data to the initial peer. Finally, it receives a ReceivedMessage, closing the connection.

From the perspective of the "owner peer": when it receives a **FileMessage**, it will verify if it really is the owner of the file, in which case it will send a **ReadingMessage**. It then reads the incoming data and composes the file. Lastly, it will send a **ReceivedMessage** to terminate the connection.

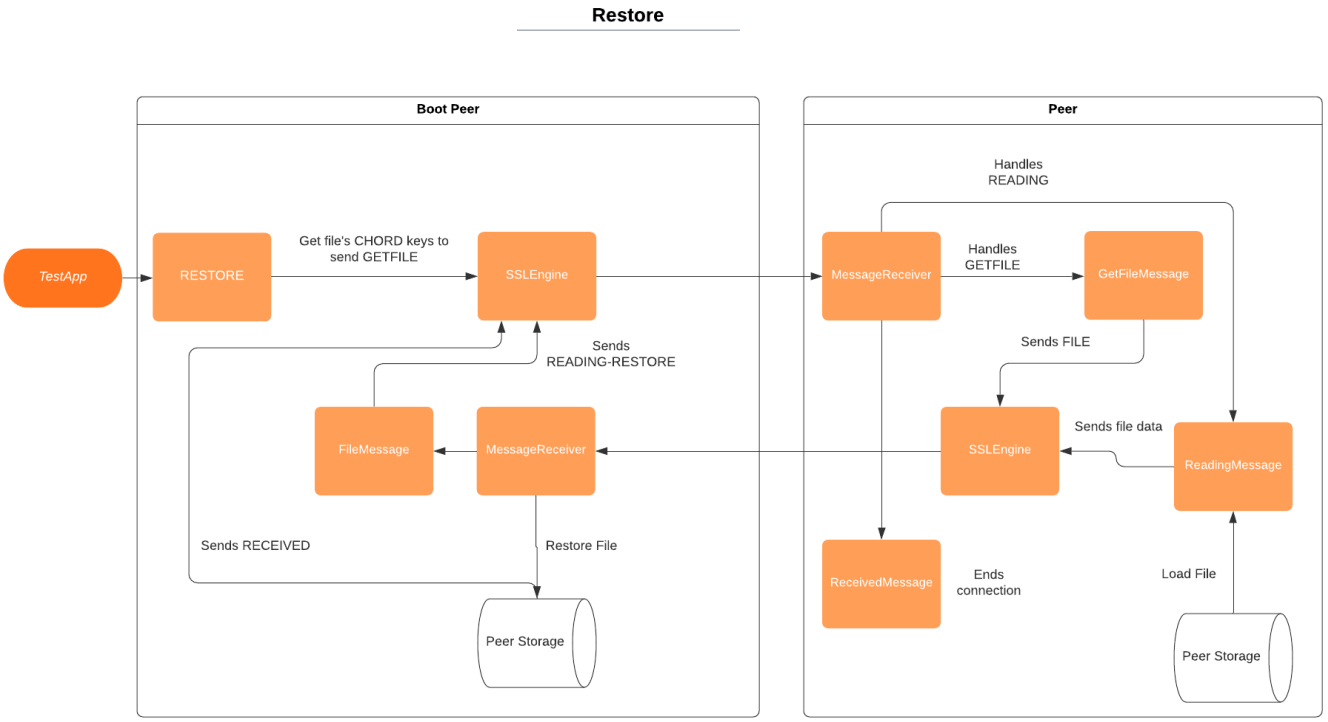


Fig 2. Restore UML diagram

Delete

The delete protocol, as in the previous project, deletes a file from the network. First, the peer retrieves the file's generated CHORD keys, representing the peers that stored the file. For each key, a **DeleteMessage** is sent.

A peer that receives this message will check if it has the file, in which case it will delete the file and all its registries.

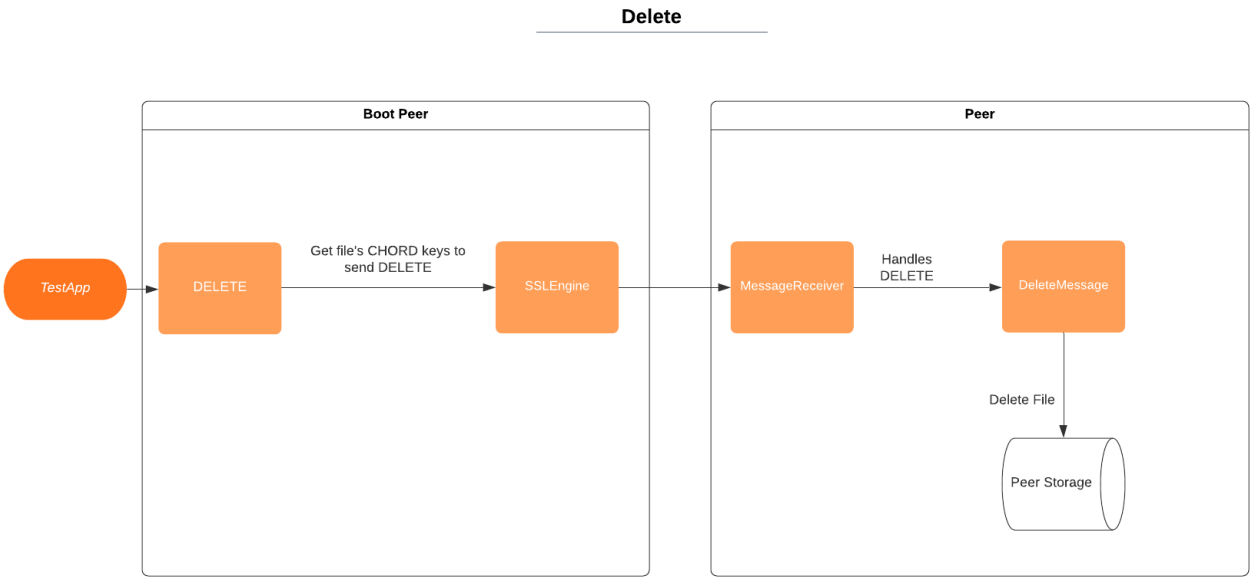


Fig 3. Delete UML diagram

Reclaim

The reclaim protocol, as in the previous project, updates the storage size of a peer and, consequently, deletes files. After updating its storage space, the peer will check if the used space has exceeded the storage space. In this case, the peer will begin to delete files, until the previous situation is no longer true.

When deleting a file, the peer will send a **RemovedMessage** to the owner of the file, to notify of this event. It will then initiate a similar process to that of backup, generating a random peer key, so that it can move the file.

This is done because every peer assumes that all the files stored are essential, or, in other words, if a file is deleted, the desired replication that a client wanted for that file is not met any more. With this, when deleting a file, a peer tries to guarantee that the file isn't lost, so it finds another peer to store it and moves the file.

From the perspective of the "owner peer": when it receives a **RemovedMessage**, it will decrease the replication degree of the file.

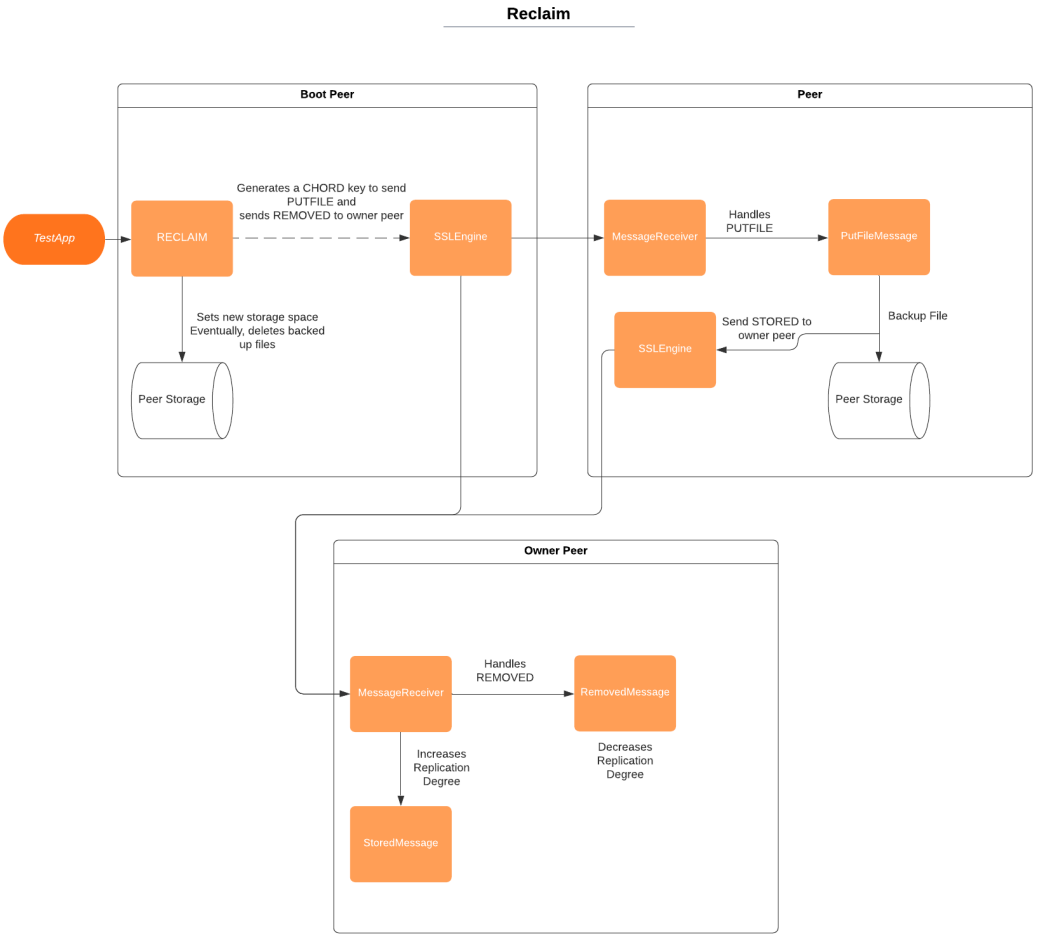


Fig 4. Reclaim UML diagram (the backup process is identical to that above, so it was not represented)

State

The state protocol, as in the previous project, shows the state of a peer. Besides showing the owned files, backed up files and storage information, it now shows the peer's CHORD ID, predecessor and finger table.

The information returned is:

- CHORD information:
  - Peer's key and address
  - Predecessor's key and address
  - Finger table with the key and the address of each finger
- For each file whose backup it has initiated:
  - The file name
  - The file path
  - The file size (in Bytes)
  - The backup service id of the file
  - The desired replication degree
  - The list of the file keys
- For each file it stores:
  - Its id
  - Its size (in Bytes)
  - Its file key
  - Its owner
- The peer's storage capacity, i.e. the maximum amount of disk space that can be used to store files, and the amount of storage (both in KBytes) used to backup the files.

Concurrency Design

In this section we'll cover the key aspects of the concurrency design of our implementation.

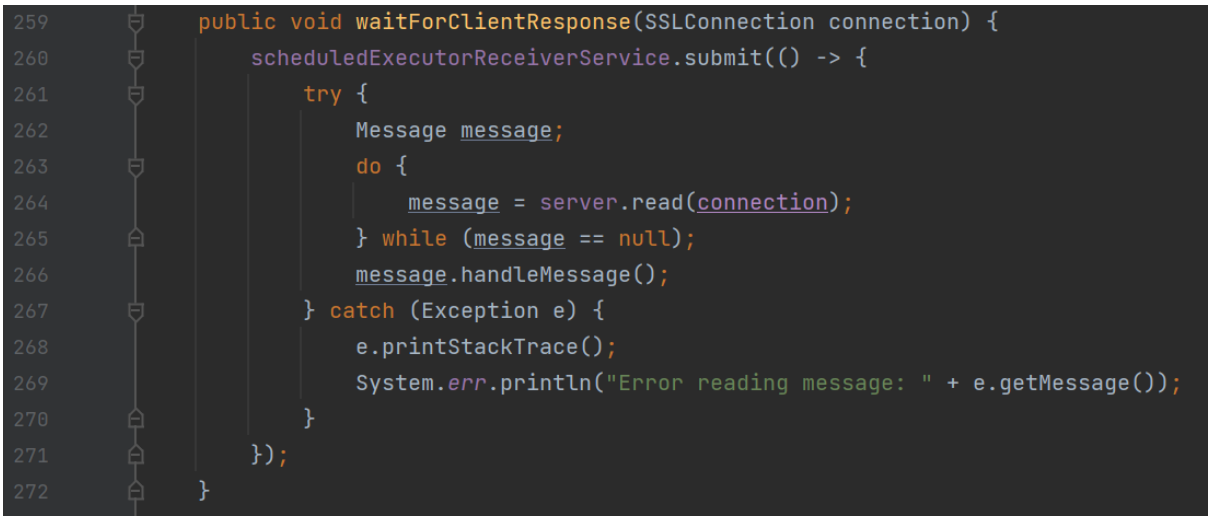
Threads

In order to run specific parts of the service in parallel, thus avoiding blocking calls from delaying the entire program's execution, we used Threads. In particular, it was crucial to parallelize the sending and receiving of messages, as well as the processing required for each operation. To efficiently manage the creation and execution of threads, we decided to use Java Thread Pool Executors. Since there are tasks like reading/writing to files that take longer and may overlap significantly at times, we had to ensure we were able to handle a large number of concurrent requests. To achieve this, we used the **ScheduledThreadPoolExecutor** class.

When a **Peer** sends a message that requires a response, a new thread is created to read the reply message and handle it, which is submitted to the **ScheduledExecutorService**. This is done both when the peer is the server or the client.

```
243    public void waitForServerResponse(SSLConnection connection, boolean stop) {
244        scheduledExecutorReceiverService.submit(() -> {
245            try {
246                byte[] response = client.read(connection);
247                if (response != null) {
248                    Message message = Message.parseMessage( peer: this, response, response.length, connection);
249                    message.handleMessage();
250                }
251                if (stop) client.stop(connection);
252            } catch (Exception e) {
253                e.printStackTrace();
254                System.err.println("Error reading message: " + e.getMessage());
255            }
256        });
257    }
```

Fig 5. Peer.java



```
259    public void waitForClientResponse(SSLConnection connection) {
260        scheduledExecutorReceiverService.submit(() -> {
261            try {
262                Message message;
263                do {
264                    message = server.read(connection);
265                } while (message == null);
266                message.handleMessage();
267            } catch (Exception e) {
268                e.printStackTrace();
269                System.err.println("Error reading message: " + e.getMessage());
270            }
271        });
272    }
```

Fig 6. Peer.java

When a message is received by an **SSLServer**, a new thread is created to handle it, which is submitted to the **ScheduledExecutorService**.

```
74    public void handleKey(Message message, SelectionKey key) {
75        if (message instanceof PutFileMessage || message instanceof GetFileMessage) key.cancel();
76        scheduledExecutorReceiverService.submit(() -> {
77            try {
78                message.handleMessage();
79            } catch (Exception e) {
80                e.printStackTrace();
81                System.out.println("Could not parse message. Reason: " + e.getMessage());
82            }
83        });
84    }
```

Fig 7. SSLServer.java

Additionally, the **SSLPeer** class has an **ExecutorService** that receives **Runnable** tasks for the SSLEngine during the handshake process, since its operations may require the results of operations that block or may take an extended period of time to complete

```
123                case NEED_TASK:
124                    Runnable task;
125                    while ((task = engine.getDelegatedTask()) != null) {
126                        executor.submit(task);
127                    }
128                    handshakeStatus = engine.getHandshakeStatus();
129                    break;
```

Fig 8. SSLPeer.java

CHORD

In order to run CHORD tasks, namely **StabilizeTask**, **CheckPredecessorTask** and **FixFingersTask**, the **ScheduledExecutorService** is used with a core pool size of 3, one for each task. These tasks are called periodically with a random delay between 10 and 20 seconds for the **Stabilize** and **CheckPredecessor** tasks, and a delay between 5 and 15 seconds for the **FixFingersTask**. This last has a smaller delay due to the higher need of execution, since it only fixes one of the 8 fingers at a time.

Java NIO

In the **SSLServer** constructor, a **Selector** is created in order to have a mechanism for monitoring the multiple non-blocking socket channels of the clients and recognizing when they become available for data transfer. While the server is running, it is continuously getting **SelectionKey** sets from the selector and checking if a new connection can be started or messages can be read from previously established connections.

```
45      public void run() {
46          while (active) {
47              Iterator<SelectionKey> selectedKeys = selectKeys().iterator();
48              while (selectedKeys.hasNext()) {
49                  SelectionKey key = selectedKeys.next();
50                  selectedKeys.remove();
51                  if (!key.isValid())
52                      continue;
53                  if (key.isAcceptable()) {
54                      try {
55                          startConnection(key);
56                      } catch (Exception e) {
57                          System.err.println("[ERROR] Could not start connection! Reason: " + e.getMessage());
58                      }
59                  }
60                  else if (key.isReadable()) {
61                      Message message = null;
62                      try {
63                          message = this.read((SSLConnection) key.attachment());
64                      } catch (Exception e) {
65                          System.err.println("[ERROR] Server could not read message! Reason: " + e.getMessage());
66                      }
67                      if (message != null)
68                          this.handleKey(message, key);
69                  }
70              }
71          }
72      }
```

Fig 9. SSLServer.java

The **ServerSocketChannel** is opened in the **SSLServer** constructor, configured to be non-blocking and the **ServerSocket** is bound to the peer's address. The selector is then registered in the socket channel.

```
33          selector = SelectorProvider.provider().openSelector();
34          ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
35          serverSocketChannel.configureBlocking(false);
36          serverSocketChannel.socket().bind(address);
37          serverSocketChannel.register(this.selector, SelectionKey.OP_ACCEPT);
```

Fig 10. SSLServer.java

When the **SSLClient** function *connectToServer()* function is called, a **SocketChannel** is opened, configured to be non-blocking and connected to the server's **InetSocketAddress**. The handshake process then begins when the connection is established.

```
23          SSLConnection connection = new SSLConnection(SocketChannel.open(), context.createSSLEngine(address, port), blocking);
24          SocketChannel channel = connection.getChannel();
25          connection.getEngine().setUseClientMode(true);
26
27          channel.configureBlocking(false);
28          channel.connect(new InetSocketAddress(address, port));
29          while (!channel.finishConnect()) {
30              // do something until connect completed
31          }
```

Fig 11. SSLClient.java

ConcurrentHashMap

The **Peer** class tracks the files being backed up by the service, storing this information in hashmaps. If multiple threads access a hashmap concurrently and at least one of the threads modifies the map structurally, it must be synchronized externally to avoid an inconsistent view of the contents. Because of that, we chose to use **ConcurrentHashMap**, a hash table supporting full concurrency of retrievals and high expected concurrency for updates, that ensures all operations are thread-safe.

```
36      private final ConcurrentHashMap<String, BackedUpFile> myFiles;
37      private final ConcurrentHashMap<String, BackedUpFile> backedUpFiles;
```

Fig 12. SSLPeer.java

JSSE

The backup service we developed uses JSSE, namely the SSLEngine class, when transmitting messages between peers as the protocols are executed.

Each object of the **Peer** class has a **SSLClient** and a **SSLServer**. Both these classes implement the **SSLPeer** class, which we created to implement a peer that communicates using JSSE. In the peer's constructor, the **SSLContext** is initialized by the *initializeSSLContext()* function and passed to both these objects' constructors. The authentication uses self-signed certificates and **TLSv1.2** is used to create the SSLContext.

```
211 @    public static SSLContext initializeSSLContext(String keystoreFile, String truststoreFile, String keystorePass,
212                                                    String truststorePass, String keyPass) throws Exception {
213          KeyStore keyStore = KeyStore.getInstance("JKS");
214          KeyStore trustStore = KeyStore.getInstance("JKS");
215
216          keyStore.load(new FileInputStream(keystoreFile), keystorePass.toCharArray());
217          trustStore.load(new FileInputStream(truststoreFile), truststorePass.toCharArray());
218
219          KeyManagerFactory kmFactory = KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
220          kmFactory.init(keyStore, keyPass.toCharArray());
221
222          TrustManagerFactory tmFactory = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
223          tmFactory.init(trustStore);
224
225          SSLContext sslCtx = SSLContext.getInstance("TLSv1.2");
226          sslCtx.init(kmFactory.getKeyManagers(), tmFactory.getTrustManagers(),  random: null);
227          return sslCtx;
228      }
```

Fig 13. SSLPeer.java

A thread is then created to run the **SSLServer.** When the server's **Selector** selects an acceptable **SelectionKey**, the server calls the *startConnection()* function, which creates the **SSLEngine** and sets the client mode to false. The SSLEngine is then moved into the initial handshaking state and a non-blocking **SocketChannel** is created to be used as the transport mechanism for the connection to use.

```
96  @    private void startConnection(SelectionKey key) throws Exception {
97           SSLEngine engine = sslContext.createSSLEngine();
98           engine.setUseClientMode(false);
99
100          engine.beginHandshake();
101
102          SocketChannel socketChannel = ((ServerSocketChannel) key.channel()).accept();
103          socketChannel.configureBlocking(false);
104
105          SSLConnection connection = new SSLConnection(socketChannel, engine);
106
107          if (handshake(connection)) {
108              socketChannel.register(selector, SelectionKey.OP_READ, connection);
109          } else {
110              socketChannel.close();
111          }
112      }
```

Fig 14. SSLServer.java

When a peer needs to communicate with another one, it calls its **SSLClient**'s *connectToServer()* function, creating a new SSLEngine in client mode. It then waits for its channel to connect to the server, moves the SSLEngine to handshaking state and calls the *handshake()* function.

```
22    public SSLConnection connectToServer(String address, int port, boolean blocking) throws Exception {
23        SSLConnection connection = new SSLConnection(SocketChannel.open(), context.createSSLEngine(address, port), blocking);
24        SocketChannel channel = connection.getChannel();
25        connection.getEngine().setUseClientMode(true);
26
27        channel.configureBlocking(false);
28        channel.connect(new InetSocketAddress(address, port));
29        while (!channel.finishConnect()) {
30            // do something until connect completed
31        }
32        connection.getEngine().beginHandshake();
33        handshake(connection);
34        return connection;
35    }
```

Fig 15. SSLClient.java

The client and the server then exchange messages until the handshake process is done and data can be passed. The **SSLConnection** class, created by us, is used to encapsulate not only the **SocketChannel** and the **SSLEngine**, but also the four needed buffers for the communication:

- **OutAppBuffer** - Buffer with the application data to send
- **OutNetBuffer** - Buffer with the encrypted data to send
- **InNetBuffer** - Buffer with the received encrypted data
- **InAppBuffer** - Buffer with the received application data

Scalability

The backup service we developed ensures scalability both at design level, by using the Chord protocol, and at implementation, by using thread-pools and asynchronous I/O, namely Java NIO.

In a simpler implementation, every peer would store information on how to communicate with every other peer in the network. This solution has very low scalability since every new peer would have to be added to the other's hash tables. To avoid this we developed a network with a Distributed Hash Table by using Chord, a protocol that provides a scalable peer-to-peer lookup service. In our implementation, each peer only needs to know information about 8 other peers to be able to communicate with all the peers in the ring. This is a faster and more efficient way of making our service more scalable.

Moreover, the use of thread-pools to send messages, handle received ones, run SSLEngine and Chord's tasks also contributes to the scalability of our implementation, ensuring that the peers are able to handle multiple tasks at once.

Finally, the use of non-blocking communication channels from the Java NIO library reduces the time it takes to transport messages and therefore allows for a higher number of peers to communicate in the network, since it allows to send and receive multiple messages from different peers at the same time.

Fault-tolerance

Our service includes multiple fault-tolerance provisions, in order to avoid single points of failure. In this section we explain what they are, how they are implemented and why we think they are important to avoid unexpected failures.

The first provision we took was to implement a replication degree desired by the client who will backup a file. When backing up a file, the client should specify the desired replication degree. The peer will then generate *3 * N* random file keys, *N* being the replication, and find those keys' successors, with the goal to find at least *N* different peers to send the file. We assume that we will find one peer that does not have the file yet for every three keys. The file is then backed up and sent to all found peers, even if there are more than *N* peers. With this provision we ensure that if one of the peers who has the file stored unexpectedly shuts down, there are other peers to request that file from.

In our service, and taking into account that there is a replication degree implemented, each peer assumes that every file it has stored is essential to the client. This means that, whenever it needs to delete a file, it will first try to transfer it to another peer before deleting it. This way, the desired replication degree is most of the time maintained, which results in a less vulnerable service.

We also implemented the CHORD features that prevent failures, such as the **CheckPredecessor** task. This task is periodically running to check if the predecessor of a peer is still active. This way we can find out if a peer has crashed and readjust the CHORD ring.

With these provisions, we can ensure that our service is adequately fault-proof.