SDIS Second Assignment Report

Distributed Backup Service for the Internet

Eduardo Ribeiro - up201705421

José Guerra - up201706421

Luís Ramos - up201706253

Miguel Pinto - up201706156

Faculdade de Engenharia da Universidade do Porto

Overview

The implementation allows the **backup** of files in other peers, the **restoration** of those files and the **deletion** of the kept files. For that, the file is divided into smaller chunks that have a maximum size of 64KB. The application also permits the **reclaim** of used storage space in the peer. Any chunks kept by the peer that do not allow the new wanted storage space are deleted; because of that every peer has complete control over its own storage. Each peer can also **exit** the system, passing all its chunks to other peers before leaving.

Our implementation of the project takes into account some of its requirements, in order to achieve a **ceiling of 20 grade points**. Firstly, in order to connect all the peers and to know which peer should keep each chunk, we use the **Chord protocol**. Each peer/node has an ID in the ring, and each chunk needs to be saved also has an ID; the successor of the chunk's ID is the node that will store that chunk (this concept has some modifications due to the existence of a replication degree, that will be explained later in the report).

For secure communication, the implementation uses JSSE, more specifically the **SSLEngine** class to encrypt and decrypt messages sent to / received from other peers.

For scalability, an implementation using **Java NIO and thread-pools** was used, using non-blocking I/O for message sending/receiving, and asynchronous I/O for reading and writing to files.

When it comes to our **fault tolerance** features, the implementation does permit the specification of a **replication degree** when a file is going to be backed up. If there are enough peers online on the Chord ring, each chunk of the file will be saved on the number of peers specified. Furthermore, all information that the peer needs to function (chunks stored, replication degrees, etc) is written regularly and kept in the peer's **file storage**; because of this, if a peer crashes, when it starts up again, its state can be restored with no loss of data.

Protocols

As said before, the operations/protocols that were implemented are the **backup, restore, delete, reclaim** and **exit**. Each of these protocols will be thoroughly explained and described, as well as the underlying transport protocols and how a client app can communicate with a peer, and how a peer can communicate with the rest of the active peers in the Chord ring.

Running a peer or a client

The command needed to run a peer is the following:

java -cp "src/" peer.PeerLauncher <REMOTE_NAME> <IP_ADDRESS> <MC_PORT> <MDB_PORT> <MDR_PORT> <CHORD_PORT> <PROTOCOL_NAME> <SERVER_KEYS> <CLIENT_KEYS> <TRUSTSTORE> <PASSWORD> (<PEER_IP> <PEER_PORT>)

The command line arguments are:

- **REMOTE_NAME** - name of the remote interface for the peer, to be used for RMI communication with the client;
- **IP_ADDRESS** - IP address of the peer;
- **MC_PORT** - Port number for the MC channel;
- **MDB_PORT** - Port number for the MDB channel;
- **MDR_PORT** - Port number for the MDR channel;
- **CHORD_PORT** - Port number for the Chord channel;
- **PROTOCOL_NAME** - Identification of the protocol to be used for encryption/decryption of messages with JSSE and SSLEngine;
- **SERVER_KEYS** - server keys to be used for encryption/decryption of messages with JSSE and SSLEngine;
- **CLIENT_KEYS** - client keys to be used for encryption/decryption of messages with JSSE and SSLEngine;
- **TRUSTSTORE** - truststore to be used for encryption/decryption of messages with JSSE and SSLEngine;

- **PASSWORD** - password to access the keys and truststore;
- **PEER_IP** (opcional) - IP address of another peer that is already in the Chord ring, in order for this peer to join the Chord ring too.
- **PEER_PORT** (opcional) - Chord channel port number of another peer that is already in the Chord ring, in order for this peer to join the Chord ring too.

The command needed to run a client is the following:

```
java -cp "src/" client.TestApp <PEER_REMOTE_NAME>
        STATE
        BACKUP <FILEPATH> <REP_DEGREE>
        RESTORE <FILEPATH>
        DELETE <FILEPATH>
        RECLAIM <NEW_STORAGE_SPACE>
        EXIT
```

Format and meaning of messages (Client-Peer and Peer-Peer)

Client-Peer

Each peer implements a remote interface, with a set of functions that a client can call using RMI. The remote interface is specified in the **RemoteInterface** class, and implemented by the **Peer** class. The remote interface of the peer is the following:

- **backup(String filepath, int replicationDegree)** - starts the backup operation of the file present in the given file path, with the desired replication degree.
- **restore(String filepath)** - restores the file that was originally backed up by the same peer using the given file path.
- **delete(String filepath)** - deletes the chunks kept by other peers relative to the file that was originally backed up by the same peer using the given file path.
- **reclaim(int diskSpace)** - starts the reclaim operation, so that the peer has a new storage space capacity. Eliminates all the chunks that it is storing that make it impossible to achieve that desired disk space.

- **state()** - returns to the client information about the peer's state, including stored chunks, files that it has backed up, Chord information, etc.
- **exit()** - makes the peer exit the Chord ring and shut down. Passes all the chunks that it has stored to other peers before it abandons the system.

Each function can be called using the appropriate command line arguments for a client (explained in the previous section).

Peer-Peer

As it will be explained in a later section of this report, the peers can communicate with each other using JSSE secure communication that uses the SSLEngine class. In order to do that, there are several types of messages that can be sent and received, each one having a different meaning and having different information stored in them. All the message types and their purpose will be explained in this section.

The **Message** and **Header** classes are used for this. Each peer also has a **MessageHandler**, that is in charge of calling the right callback function in the protocol when a message is received. This is done by checking the message's type when it is received.

- **Regular protocol messages:**
  - **PUTCHUNK** - message that contains a chunk, sent by a peer that is initiating a backup protocol, telling the receiving peer to store this chunk if possible. Contains the **fileId** and **chunkNo** of the chunk, the desired **replication** of the chunk, the **ipAddress** and **port** of the peer to contact in order to confirm that the chunk has been stored, and finally, it contains the **body** of the chunk.
  - **GIVECHUNK** - message that also contains a chunk. It is used in the **exit** protocol, when a peer needs to give a chunk that it has stored to another peer, before it leaves the Chord ring. Contains the **fileId**, **chunkNo**, **ipAddress** and

port of the initiator peer, a **barrierId** that is used in order to avoid the message to propagate to other peers unnecessarily, or in other words, to avoid the message to make more than one lap around the Chord ring (this will be explained in detail in a later section). Finally, it of course contains the **body** of the chunk.

- **STORED** - message sent by a peer that just stored a chunk, to the peer that asked the chunk to be stored (or in other words the peer that initiated the backup protocol). It contains the **fileId** and **chunkNo** of the stored chunk.
- **REMOVED** - similar to the **STORED** message, but for the inverse operation, that is, to inform the initiator peer that the sender peer just removed the chunk.
- **GETCHUNK** - message sent by a peer that initiated a restored request for a specific file. It asks the receiving peer for the contents of that file. Contains the **fileId** of the file, as well as the **ipAddress** and **port** of the peer to contact and to send the file's contents.
- **CHUNK** - response to the **GETCHUNK** message; it is sent to a peer that has requested the restore of a specific file; it transports a chunk of that file to the initiator peer, and is sent by a peer that keeps that chunk. Contains the **fileId** and **chunkNo** of the chunk, as well as the actual contents of the chunk, in the **body** of the message.
- **DELETE** - message sent by a peer that initiates a delete protocol for a specific file. Sent to the peers that store chunks of that file. Contains the **fileId** of the file, as well as the **ipAddress** and **port** of the peer to contact and to inform that the chunks of the file were in fact deleted.
- **DELETED** - response to the **DELETE** message. Sent to the initiator peer to inform him that the chunks of the file have been deleted. Contains the **fileId** of the file.

- **Chord related messages:**
  - **FIND_SUCC** - message that is related to the main feature of the Chord protocol: the lookup. The sender node/peer asks the receiver node for the successor of a

given key. Contains the **key** and the **ipAddress** and **port** of the initiator node to contact.

- **RTRN_SUCC** - response to the **FIND_SUCC** message. Contains the information about the **successor** of the key, and the **key** that was asked.
- **GET_PRED** - message used in the **stabilize()** Chord method (that will be explained later). It is sent to the node's successor, in order to get its predecessor, which might be our new successor.
- **RTRN_PRED** - response to the **GET_PRED** message. Returns the predecessor of the node.
- **NOTIFY** - message sent to the peer's successor, warning it that we might be his predecessor.
- **CHECK_ACTIVE** - message sent to the peer's predecessor, to make sure that it is still active and functioning. If the message cannot be correctly sent and received by the predecessor, it is set to null in the send peer.
- **SET_SUCC** - message sent to a peer, with its new successor.
- **SET_PRED** - message sent to a peer, with its new predecessor.

Chord

Firstly, our implementation of the **Chord protocol** is going to be explained, showcasing the data structures used to maintain information about other nodes and the format of the messages sent that are related to Chord.

The main data structures used to maintain information about the nodes in the chord were the **ChordNode** class and the **ChordRingInfo** class. An additional class, named **ChordUtils**, was also created to maintain some utility functions for Chord's procedures. All these classes belong to a package in the src folder of the project named **chord**.

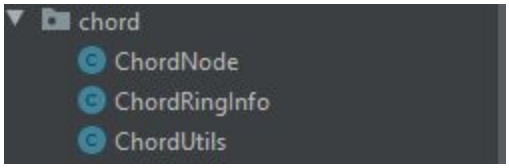

Figure 1. Chord data structures

The **ChordNode** class represents a node/peer in the Chord protocol, and holds all the information relative to it.

The **member variables** in this class and their use are the ones that follow:

- **int id** - ID of the node in the Chord protocol, that identifies that peer in the ring.
- **String ipAddress** - IP address of the node.
- **int portMC** - Control channel port of the node, that is used for message sending/receiving.
- **int portMDB** - Backup channel port of the node.
- **int portMDR** - Restore channel port of the node.
- **int portChord** - Chord channel port of the node, that is used for sending/receiving messages specific to Chord. (those messages will be described later in this section).

It was decided to follow the same approach as the first project and keep three separate channels for the communication between peers, one used for control messages, one for the backing of file chunk data and another for restoring file chunk data. One more channel was added for communication between peers because of chord related messages, the Chord channel. This approach was taken in order to not saturate a specific port with all the messages being sent and received.

The **ChordRingInfo** class represents the distributed hash table used by the Chord protocol. This class contains the information about other peers in the Chord ring such as the Chord's node predecessor, successor and the routing finger table that holds all the nodes that the

node can access directly without having to exchange messages beforehand. This class also accommodates all the functions necessary for creating, entering and managing Chord.

The main structures kept and used in this class are:

- **ChordNode nodeInfo** - contains information relative to the actual peer.
- **ChordNode predecessor** - contains information relative to the predecessor of the node (if any).
- **AtomicReferenceArray<ChordNode> fingers** - a concurrent, thread-safe array used to store information about the node's fingers. The successor of the node is always the first entry of this table (at index 0).

Next is a list of the main functions in this class and their purpose:

- int **generateHash**(String key) (line 115) - generates the SHA-1 hash , making an identifier for the Chord ring. The string key used is unique to each peer, and takes into account the node's IP address and port numbers.
- void **createOrJoin**(string initIpAddress, int initPort) (line 156) - makes the node create a new Chord ring if the initIpAddress is null and initPort is -1, otherwise it tries to join the chord ring by contacting the node with ip address initIpAddress in port initPort. (For a node that is not
- ChordNode **findSuccessor**(int keyHash, String ipAddress, int port, Task task) (line 225) - finds the successor node for a given key keyHash and contacts the node with ip address ipAddress and port port when it knows the response. If the task parameter passed is not null, then executes the task (a given function) after finding the successor.
- void **stabilize**() (line 315) - ran periodically, asks the direct successor of the node for its predecessor in order to know about new nodes that came in the Chord ring.
- void **fixFingers**() (line 367) - ran periodically. Tries to find the successor of the necessary key in order to get the ith finger for the finger table.

- void **startCheckPredecessor**() (line 393) - ran periodically. Contacts the predecessor of the node in order to check if it is still active and functioning correctly. If that is not the case, set predecessor to null.
- void **exit**() (line 418) - function to be called when the local peer wants to exit the ring and shut down. Sends messages to the predecessor and successor in order to connect them, so that the local peer can exit the ring without causing problems in other peers.

NOTE: The Chord protocol implementation is heavily based on the Chord paper provided by the professor and that was present in the subject's website. (https://dl.acm.org/doi/pdf/10.1109/TNET.2002.808407)

Another class connected to the Chord protocol that should be mentioned and explained is the **TaskManager** class. Firstly, and as it was already mentioned, a **Task** is an interface with a **complete**(**ChordNode successor**) method. By calling task.complete(successor), we can run any callback function. Tasks are used to run functions after the peer receives information about the successor of a key it asked.

When a peer wants to know the successor of a given key, two scenarios can happen:

- The peer has enough local information to know what the successor of the key is, without having to contact other peers. In this case, the successor is calculated and the task is completed immediately after that.
- The peer does not have enough information to know what is the successor of the key. In order to know that, it has to contact other peers, which will respond to the original peer with the key's successor, or forward the request to other peers if they also don't know the answer (this is done using the finger tables and according to the Chord protocol). When this happens, the task cannot be completed immediately because the peer needs to wait for the response in order to know the key's successor.

When this second scenario happens, the task is kept in the task manager. The task manager has a data structure keeping all the pending tasks, **ConcurrentHashMap<Integer, ConcurrentLinkedQueue<Task>> pendingTasks**. The key of the hash map is an integer

containing the value of the key to which we want to know the successor. The value is a queue containing all the tasks relative to that key. When a response is sent to a peer, containing the successor of a given key, the peer completes all the pending tasks that it had that were related to that key. This can be seen in line 78 of the **MessageHandler** class (it calls the **completeTasks** method of the **TaskManager** class, at line 17). After the tasks are completed/consumed, they are deleted.

The main operation that can be done with Chord is the **lookup** of the **successor** of a key. This way, we can know which node/peer should have to store a certain chunk. The lookup of a successor is also used for other operations related to Chord. Our implementation of this functionality is as follows:

When a peer needs to know the successor of a key, it calls the **startFindSuccessor** method (**ChordRingInfo** class, line 190). It receives the key to which we want to know the successor, and a task that should be executed after the peer knows the successor of the key. The peer will check if the key is between its own ID and its successor's ID; if that is the case, then the peer's successor is the key's successor. If not, the peer will check the nodes stored on its finger table in order to check if any of them is the successor. If that is not the case, then it will check if the actual peer is the key's successor.

If every of these verifications are not successful, then the peer does not have enough local information, and needs to contact other peers in order to know the response. For that, it sends a **FIND_SUCC** message to the closest preceding node relative to the key's value.

When a node receives a **FIND_SUCC** message, it runs the **handleFindSuccessor** method (**ChordRingInfo** class, line 200); it tries to find the successor for that key, and the process is repeated: it tries to find the successor based on its local information; if that is not enough then it sends a **FIND_SUCC** message to the closest preceding node.

When a node discovers the successor of a key locally, it returns the response to the first, initiator peer (the one that called **startFindSuccessor**), using a **RTRN_SUCC** message.

The following diagram illustrates what a find successor call can look like on the distributed hash table implemented by the Chord protocol:

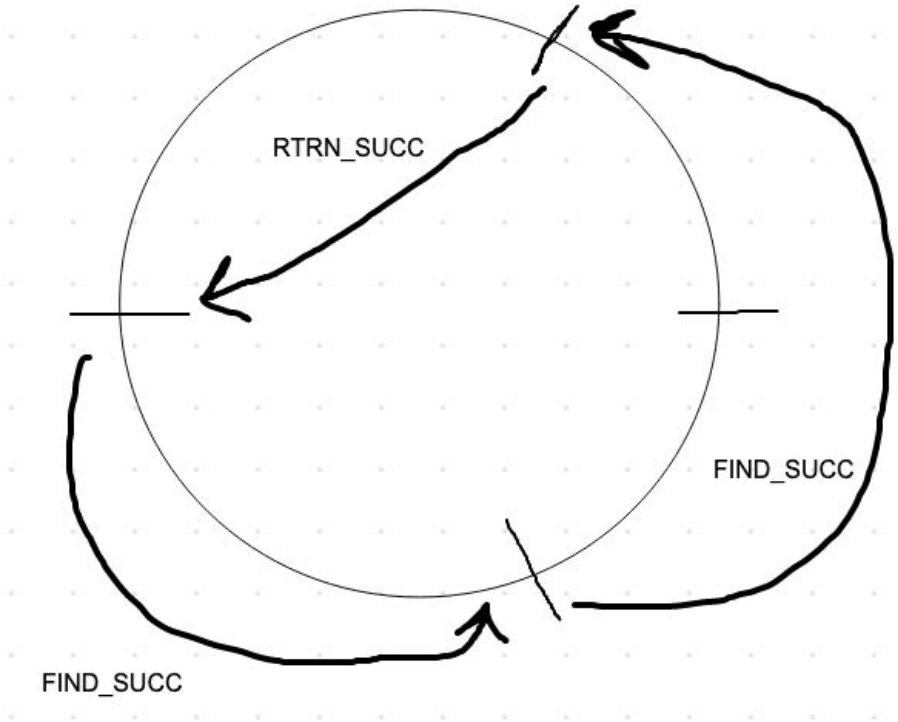

Figure 2. Diagram showing Chord lookup functionality

Backup

The backup protocol starts by calculating the identifier of the file to be backed up, known as the **fileId** (**startFileBackup** method, in **Protocol** class, line 73). The file is divided into chunks, and for each chunk, a chord ID is generated, using the string "**fileId_chunkNo**" as a unique identifier (**initiateBackup** method, in **Protocol** class, line 94). The successor of the chunk's key is requested by the peer and, after the successor is know, a **PUTCHUNK** message is sent to it, so it can backup the chunk (**backupChunk** method, in **Protocol** class, line 139).

The peer, after it receives a **PUTCHUNK** message (**handleBackup** method, in **Protocol** class, line 185), can have one of various behaviours: if it already has the chunk or if it can't save it (for example, does not have enough space), it forwards the message to its successor. If it can save it, it will do that, and propagate the **PUTCHUNK** message, decrementing the replication degree on the message by 1 (**redirectBackup** method, in **Protocol** class, line 618). That replication degree then indicates the **number of nodes that still need to backup that chunk in order for the desired replication degree to be met**. If the replication degree of the message decreases to 0, then the message is not propagated. Also, if the initiator peer receives that **PUTCHUNK** message it simply redirects it to the successor, in order to avoid the chunk to be saved by the initiator peer. The message is also only propagated if the **key of the chunk is not between the current peer's ID and the next peer's ID** (**canPropagate** method, in **Protocol** class, line 657). This is done to make sure that the message does not pass through a peer more than once, that is, that it does not make more than one lap around the Chord ring. If that would happen, then it would mean that there are not enough peers on the ring that are able to store the chunk, and so the message can stop being propagated.

A peer, after it saves a chunk, shall send a **STORED** message to the initiator peer. This way, the peer that ordered the file backup knows exactly which peers have backed up the chunks of the file.

Another important aspect of the backup protocol is that when a file that was already backed up is modificated and asked to be backed up again, the initiator peer will first start a **deletion protocol** for the old version of the file, and then a **backup protocol** for the new file (**deleteIfOutdated** method, in **Protocol** class, line 164).

The following diagram better illustrates the functioning of the backup protocol.

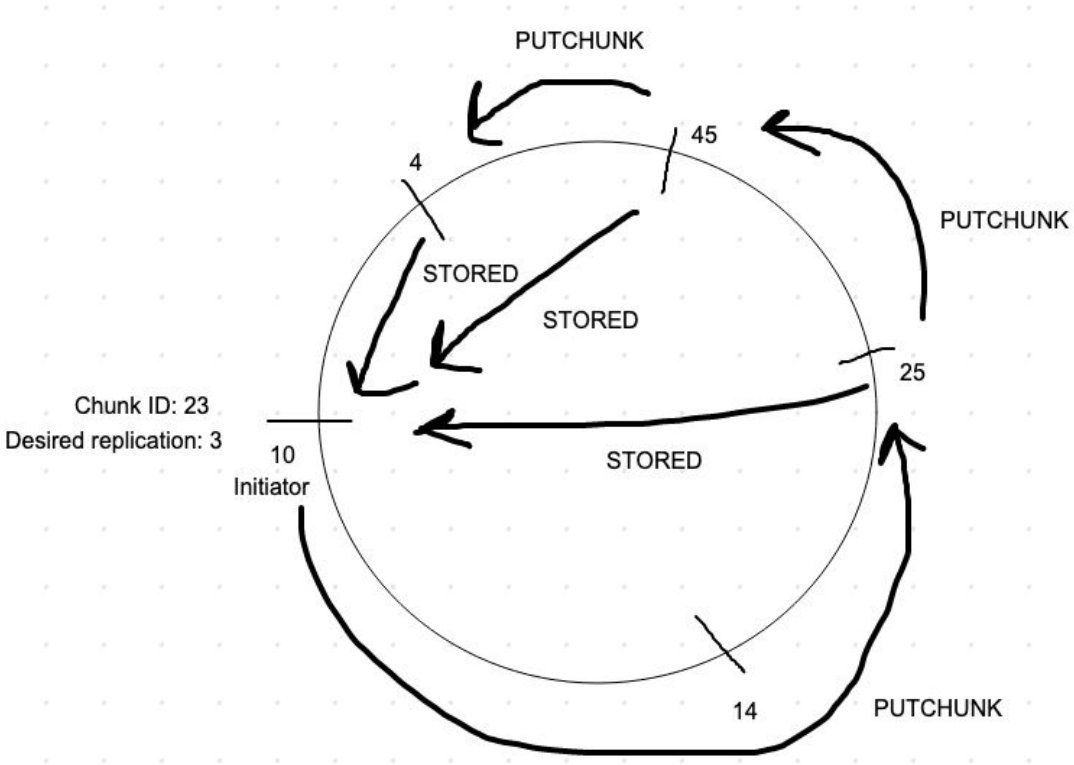

Figure 3. Diagram showing backup protocol functioning

Restore

The restore protocol takes advantage of the fact that the initiator peer knows who stored the chunks of a file, this is done by saving all the peers' IDs that sent a **STORED** message. The IDs are saved sequentially in the order that the **STORED** message was received and since the chunk stored is equal between all, the protocol starts by sending a **GETCHUNK** message to the first peer that sent the **STORED** message (call to **sendRestore** method with index 0, in **Protocol** class, line 409). If however the peer that received the **GETCHUNK** message does not currently have the chunk requested, the peer will propagate the message to his successor, this occurs until the **key of the chunk is between the current peer's ID and the next peer's ID** (**canPropagate** method, in **Protocol** class, line 657). The peer that receives a **GETCHUNK** message and has the chunk then sends the chunk in the **CHUNK** message to the initiator peer and stops the

propagations of the **GETCHUNK** message. It is important to note that should the message transmission between Initiator peer and the known storer of that chunk then the service seeks the next known storer in his records (call to **sendRestore** method with next storer's index, in **Protocol** class, line 423).

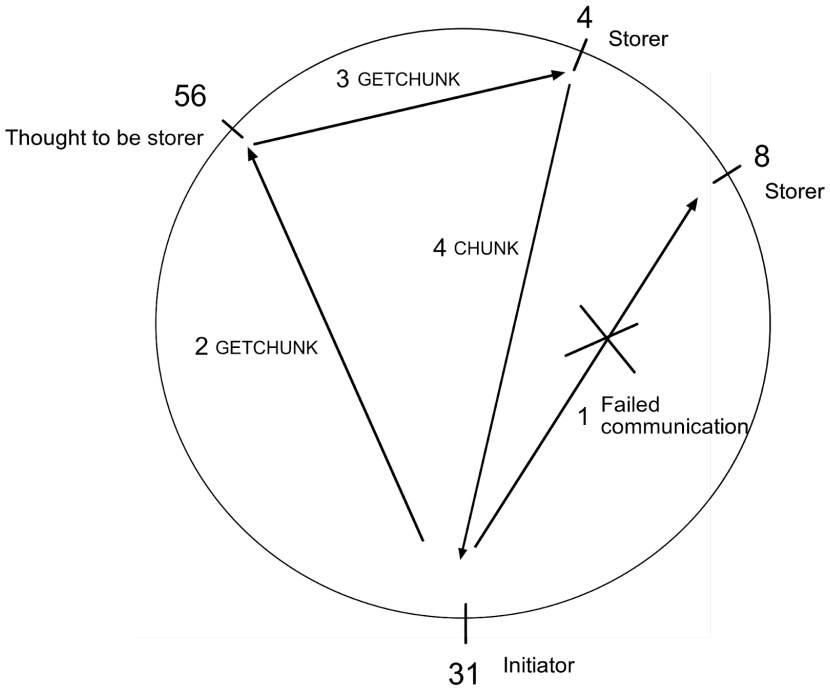

Figure 4. Diagram showing the Restore protocol functioning

Delete

Because of the use of the data structure that keeps information about all the peers that have backed up a specific chunk, all the delete protocol needs to do is to send a **DELETE** message to all the peers that keep at least 1 chunk of the specified file to be deleted. All local information about the file in the initiator peer is deleted (**initiateDelete** method, in **Protocol** class, line 431).

In order to tolerate failures of the nodes/peers that have the chunks backed up, and in order to make sure that every piece of information about the file is in fact deleted in every peer, the initiator peer keeps a **FileDeleter** instance, with the **DELETE** messages to send for a specific

peer. In the case when a peer is not available or is down when the initiator sends the **DELETE** message, when that peer comes back up, if it sends any message to the initiator peer, it will know that that peer is up and running again, so it will get the **FileDeleter** instance connected to that peer and it will send all the **DELETE** messages that it had cached previously.

When a peer receives a **DELETE** message (**delete** method, in **Protocol** class, line 477), it will remove all chunks from the file that it has stored, and will respond with a **DELETED** message. The initiator, after receiving a **DELETED** message (**receiveDeleted** method, in **Protocol** class, line 524), has confirmation that the other peer has in fact deleted the chunks of the file, so it can delete the **FileDeleter** instance associated with it.

The following diagram better illustrates the functioning of the delete protocol.

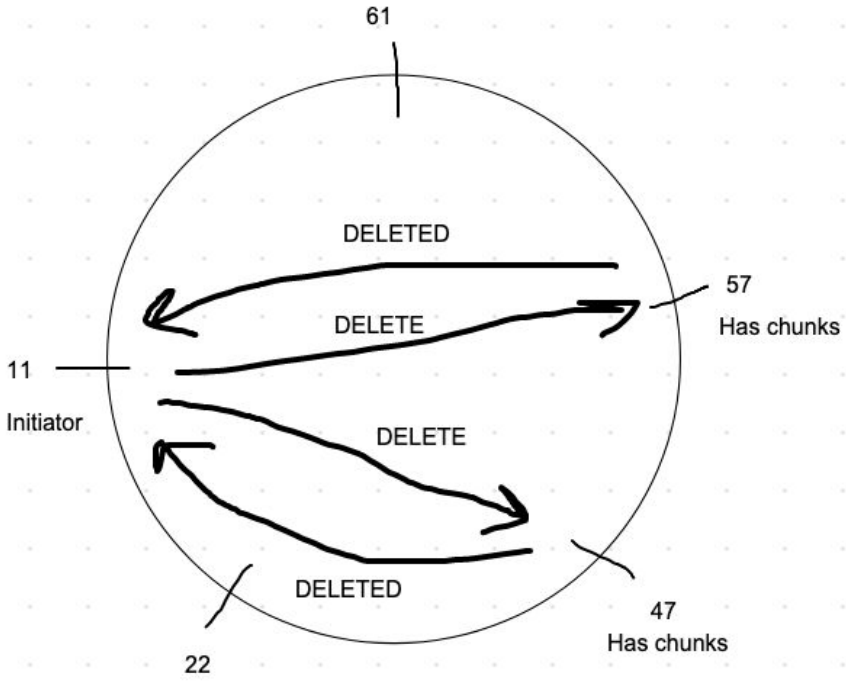

Figure 5. Diagram showing delete protocol functioning

Reclaim

When a peer wishes to reclaim some of its space it does so by deleting some chunks it has stored. When deleting a chunk, the peer sends a **GIVECHUNK** message to its successor containing the chunk (**reclaim** method, in Protocol class, line 559) . The successor has two options, it either saves the chunk (**handleBackup** method, in **Protocol** class, line 208) and sends the **STORED** message to the initiator peer of that chunk (**handleBackup** method, in **Protocol** class, line 211), or retransmits the message to its successor (call to **redirectBackup** method, in **handleBackup** method in **Protocol** class, line 196, line 246). This retransmission might happen if the peer has no available space, the peer is the initiator peer for that chunk or some error happened when saving the chunk to its storage.

After sending the **GIVECHUNK**, the peer reclaiming space sends yet another message, this time to the initiator peer of the chunk, but not before deleting the chunk content and all the metadata related to it (**removedChunk** method, in **Protocol** class, line 582). The message sent is the **REMOVED** one (**removeChunk** method, in **Protocol** class, line 587), signaling to the initiator peer that the peer from where that message originated from is about to no longer hold the file in its storage. The initiator peer then removes the peer, who no longer holds the chunk, from the list of peers holding the chunk (**removed** method, in **Protocol** class, in line 610).

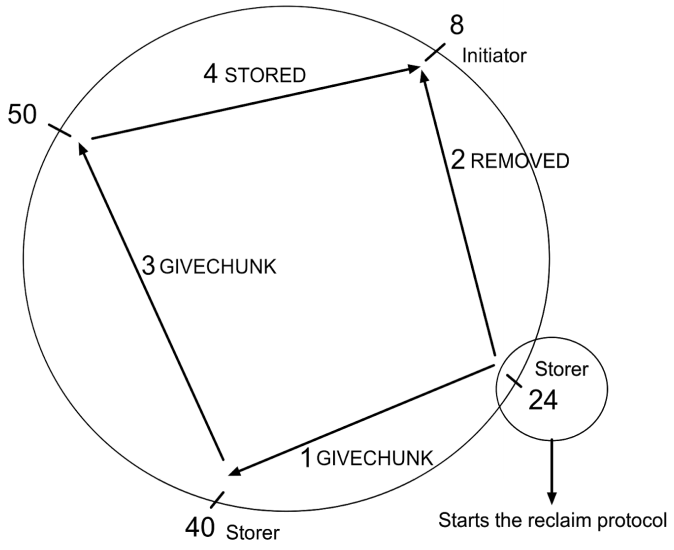

Figure 6. Diagram showing the reclaim protocol working

State

The state protocol does not interact with other peers; rather, it compiles information about the local peer's state, and returns it to the client. It can be used by the client for test purposes and to know the state of the peer in a specific instant (**state** method, in **Protocol** class, line 729). The information about the peer returned to the client in the state protocol is the following:

- Files backed up by other peers (for each one of them):
    - The path name;
    - The file ID;
    - Chunks of the file (for each one of them):
        - Chunk ID;
        - Perceived replication degree;
        - IDs of the peers that store that chunk.
- Files stored (for each one of them):
    - The file ID;
    - IP address and port number of the initiator peer;
    - Chunks of the file (for each one of them):
        - Chunk ID;
        - Chunk size (in KB).
- Maximum storage capacity (in KB);
- Available storage capacity (in KB);
- Chord information:
    - Peer/node ID;
    - Predecessor ID;
    - Successor ID;
    - Finger table IDs.

Exit

This protocol is called when a client tells a peer that it should leave the system. It comprises several operations made to make sure that the exit of this peer does not produce errors in the functioning of other peers and in the Chord ring in general (**exit** method, in **Protocol** class, line 705).

Firstly, all the Chord tasks that are ran periodically (**stabilize**, **fixFingers**, **checkPredecessor**) are stopped.

Then, all chunks that the peer has stored need to be transferred to other peers, so there is no loss of information. This situation can be treated as a **reclaim protocol call, where the new desired disk space is 0.** This means that no new special function for this case needs to be created, and the function **reclaim(0)** can be called.

After the peer is not storing any chunks, it needs to make sure that its exit does not affect the Chord ring and the other peer's functioning and inter-communication. For that, the peer sends a **SET_SUCC** message to its predecessor, with information about the peer's successor, and a **SET_PRED** message to its successor, with information about the peer's predecessor.

After that, the peer stops receiving and sending messages, and can safely terminate its program.

Concurrency design

The concurrency in this project is achieved by the use of threads, thread_pools and Java NIO. There are two specific classes that were created to allow for concurrency to be achieved. They are called **ReceiverThread** and **SenderThread** . Both these classes extend an abstract class called **SSLThread** that contains the foundations for a secure SSL communication between peers. The instances of these classes are intended, as the name of the class implies, to be executed by a thread, in light of that both of them implement the **Runnable** interface from package java.lang All these classes are found in the package jsse, which in itself belongs to package peer, present in the src folder of the project.

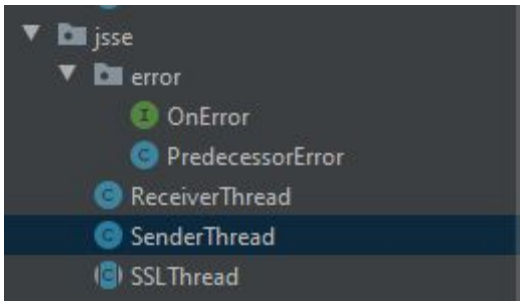

Figure 7. Package jsse classes

The SSLThread contains a function called **perfomHandshake** that implements the handshake protocol between two peers, required for the establishment of the SSL/TLS connection. This function will be described with more care in the next section of the report but it is important to mention it here since both the ReceiverThread and SenderThread use it to establish a connection.

The **SenderThread** is used whenever a node wants to send a message to another. It can be interpreted as a SSL/TLS client that connects to a peer, using its IP address and port and sends it a message. Describing the process in more detail, when a node wants to send a message to another it uses a private static and final thread pool (Figure 3) created in the **SenderThread** class to execute a new instance of this class in another thread. This is done in the public static void function called **sendMessage** also present in the SenderThread class, as can be seen below in figure 4.

```
/**
 * Executor service to send messages
 */
private static final ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);
```

Figure 11. Thread pool in the SenderThread class

```java
/**
 * Static method for message sending.
 * @param remoteAddress IP address to send the message
 * @param port Port to send the message
 * @param message Message to send
 * @param onError Error function to run if error occurs
 */
public static void sendMessage(String remoteAddress, int port, Message message, OnError onError) {
    try {
        if (!exit.get())
            service.execute(new SenderThread(remoteAddress, port, message, onError));
    } catch (Exception e) {
        System.err.println("Error creating sender thread");
        e.printStackTrace();
    }
}
```

Figure 12. sendMessage function in the SenderThread class

It is this new thread's responsibility to establish a connection to the receiver peer of the message by creating a socket channel, connecting it to the receiver's correspondent socket with their ip address and port and finally performing the handshake protocol between the two peers using the function **performingHandshake.** This process is all done at the start of thread in the function **connect.**

```java
@Override
public void run() {
    if (message == null)
        return;
    message.setSenderId(senderId);
    try {
        if(!connect()){
            System.err.println("Error (connection) when sending message: " + message.getHeader());
            if (onError != null)
                onError.errorOccurred();
            return;
        }
```

Figure 13. Start of the method run, executed when a new instance of SenderThread starts running on a thread.

```java
/**
 * Opens a socket channel to communicate with the configured server and tries to complete the handshake protocol.
 * @return True if client established a connection with the server, false otherwise.
 * @throws Exception when the connection fails
 */
protected boolean connect() throws Exception {
    socketChannel = SocketChannel.open();
    socketChannel.configureBlocking(false);
    socketChannel.connect(new InetSocketAddress(remoteAddress, port));
    while(!socketChannel.finishConnect());
    engine.beginHandshake();
    return performHandshake(socketChannel, engine);
}
```

Figure 14. function connect in SenderThread class

If the connection doesn't fail then it is time to effectively send the message to the receiver. To send the message a function called **writeToPeer** is used.

```java
        writeToPeer(socketChannel, engine);

try {
    ReceiverThread receiverThread = new ReceiverThread(messageHandler, protocol, serverKeys, trustStore, password, N_THREADS);
    receiverThread.addServer(ipAddress, portMC);
    receiverThread.addServer(ipAddress, portMDB);
    receiverThread.addServer(ipAddress, portMDR);
    receiverThread.addServer(ipAddress, portChord);

    //starting the receiver thread
    new Thread(receiverThread).start();

} catch (Exception e) {
    System.err.println("Could not initiate receiver thread");
    e.printStackTrace();
}
```

Figure 15. Receiver thread configuration and execution in the constructor for Peer class

When it comes to the ReceiverThread class, it starts running in a thread at the start of the peer and it stays active until the peer's shutdown. This thread is started in the constructor of the peer class where it is also previously configured, as can be seen in figure 8. The use of this thread is to receive incoming messages from other peers. It can be seen as a sort of SSL/TLS "server" that accepts and reads messages from other peers.

This receiver thread uses Java.NIO **Selector** class to serve all connections made by other peers to the peer on the 4 channels available. This is particularly efficient since it isn't necessary to have 4 extra threads listening for connections, one per channel available, instead a single thread can be used for managing multiple channels, and thus multiple network connections.

```
/**
 * A part of Java NIO that will be used to serve all connections to the server in one thread.
 */
private final Selector selector;
```

Figure 16. Java.NIO selector in ReceiverThread class

To add the channels for the selector to manage, function **addServer** is utilized with the parameters ipAddress and port of the channel.

```
/**
 * Adds a server to the server pool of the selector.
 * @param ipAddress The IP address that of the new server
 * @param port The port number of the new server
 * @throws IOException
 */
public void addServer(String ipAddress, int port) throws IOException {
    ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
    serverSocketChannel.configureBlocking(false);
    serverSocketChannel.socket().bind(new InetSocketAddress(ipAddress, port));
    serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
}
```

Figure 17. function addServer in the ReceiverThread class

The ReceiverThreads blocks in the selector.select() until at least one channel is ready for an operation. Then, when the condition is met, the receiver thread starts by calling selector.selectedKeys(), a set of SelectionKey objects is returned, each key represents a registered channel which is ready for an operation. After this, the set is iterated over and for each key, it is checked if it is a valid and acceptable one, if it isn't the key is discarded.

```java
@Override
public void run() {
    while (!exit.get()) {
        try {
            selector.select();
        } catch (IOException e) {
            continue;
        }
        Iterator<SelectionKey> selectedKeys = selector.selectedKeys().iterator();
        while (selectedKeys.hasNext()) {
            SelectionKey key = selectedKeys.next();
            selectedKeys.remove();
            if (!key.isValid()) {
                continue;
            }
```

Figure 18. run method in the ReceiverThread class

After verifying the validity of the key, then it is accepted and the handshake with the sender peer is made using the performingHandshake function. From this function results in a Socket where the receiver will receive the message. In order to maintain concurrency, this task of receiving the message and also processing it, is attributed to another thread. For that, a thread pool is created in the ReceiverThread, in its constructor, as can be seen in figure 12 and 13.

```java
/**
 * ExecutorService responsible for threads.
 */
private final ExecutorService service;
```

Figure 19. ExecutorService service in ReceiverThread class

```java
public ReceiverThread(MessageHandler messageHandler, String p
    this.service = Executors.newFixedThreadPool(nThreads);
    this.messageHandler = messageHandler;
```

Figure 20. Assigning a number of threads to the thread pool in the ReceiverThread's constructor

The function readFromPeer is used to read the message and then the processing of the message is done by the function called **process .**

```
} else if (key.isValid() && key.isReadable()) {
    key.cancel();
    // execute in thread pool the receiving of the data and the
    // composing and processing of the message
    this.service.execute(() -> {
        SocketChannel channel = (SocketChannel) key.channel();
        SSLEngine engine = (SSLEngine) key.attachment();
        ByteBuffer message;

        try {
            message = readFromPeer(channel, engine);
        } catch (IOException e) {
            System.err.println("Error while trying to read a message");
            e.printStackTrace();
            return;
        }

        this.messageHandler.process(message);

        try {
            closeConnection(channel, engine);
        } catch (IOException ignored) {
            System.err.println("Error closing the socket after reading a message");
        }
```

Figure 21. Executing thread to read the message and process it

JSSE

The JSSE services are utilized in this project when sending messages between peers. For this effect, we used the SSLEngine class in order to achieve a higher degree of security when transmitting data through the TCP connections that are established throughout the execution of the protocols.

The SSLEngine is created using the class SSLContext. This class is instanced with the protocol passed as an argument in initialization of the application and then the init method is called with an array of KeyManager, an array of TrustStore and a SecureRandom which provides a cryptographically strong random number generator.

```
SSLContext context = SSLContext.getInstance(protocol);
context.init(createKeyManagers(clientKeys, password, password), createTrustManagers(trustStore, password), new SecureRandom());
this.engine = context.createSSLEngine(remoteAddress, port);
this.engine.setUseClientMode(true);
```

Figure 22. Creation of SSLEngine class in SenderThread (line 126)

```
SSLEngine engine = context.createSSLEngine();
engine.setUseClientMode(false);
engine.beginHandshake();

// try to perform handshake
if (performHandshake(socketChannel, engine)) {
    socketChannel.register(selector, SelectionKey.OP_READ, engine);
} else {
    socketChannel.close();
    System.err.println("Connection closed due to handshake failure.");
}
```

Figure 23. Creation of SSLEngine and beginning of handshake process in ReceiverThread (line 96)

When the SSLEngine is created and the connection has been successfully established the handshake process starts. This process requires the exchange of messages between the intervening parties, i.e. the client and the server. This process must be started and fully fulfilled before any application data can be passed. The messages passed by the SSLEngine are as follows:

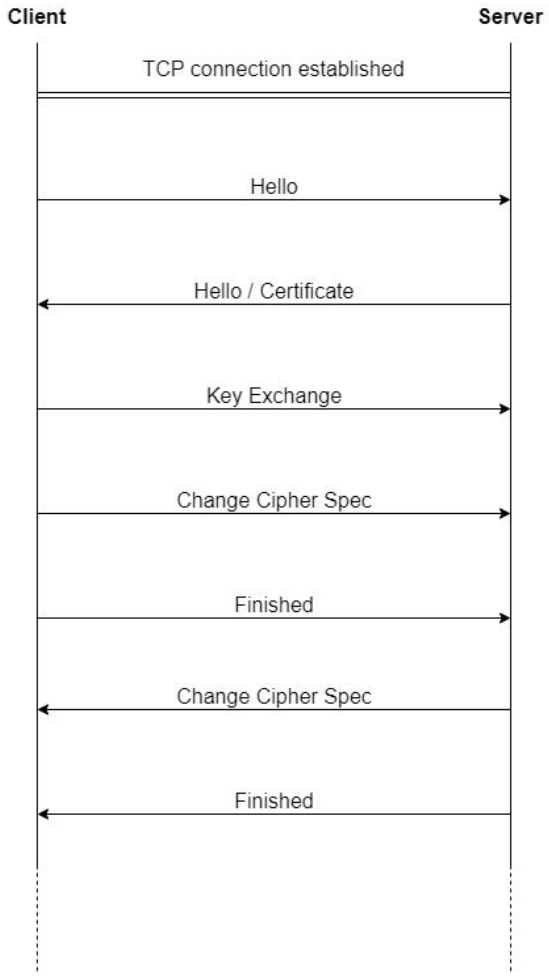

Figure 24. Handshake process of the SSLEngine class

Scalability

The project ensures scalability by two main aspects: using the Chord protocol, on the design level; and using thread-pools and asynchronous I/O, more concretely, Java NIO, on the implementation level.

The Chord protocol was designed with the intent of being a scalable peer-to-peer lookup service for internet applications, so implementing it as described in its original scientific article description contributes greatly to the project's scalability. Some changes were made to the protocol, in order to ensure maximum compliance of the project's specification, while still maintaining the level of scalability. By keeping information about only a few nodes (around log(N) nodes, given that N corresponds to the total number of nodes in the network) and with automatic finger table adjustment on node entry and exit, through the use of specialized messages, good performance and reliability with a big number of working nodes is ensured. The in-depth explanation is in the previous **Protocols** section.

The use of Java NIO, on both the receiver and sender threads, also contributes to the scalability of the application, by allowing multiple message sending and receiving without significant overlay. Its in-depth explanation is in the previous **JSSE** section.

Thread pools are used throughout the application; for instance, to send messages, to process received messages, to run certain Chord tasks periodically and to start protocol instances ordered by a client. All thread polls are used in a controlled manner in order to increase the concurrency of the system but at the same time to keep control on the number of threads used in order to maintain the scalability of the program.

Fault-tolerance

By implementing fault-tolerance, the goal is to avoid single-points of failure. The group decided to implement three important fault-tolerance features.

One way that the system could fail is if there is only ever one copy of a file being stored. If the node in charge of storing it shuts down unexpectedly, then that file is lost, at least while that peer is unavailable. Attempts to restore it by the initiator peer would fail. In order to reduce this possibility, a replication degree associated with each file was implemented. In this case, if one of the peers shuts down, then the initiator peer can simply ask for one of the other peers that also holds a replicated chunk of the file.

Another fault-tolerance feature is saving the peer's state in memory. This way, in case of a peer shutting down unexpectedly, all of its importation information is available on disk, as updated as possible, so that it can be quickly loaded back when the peer returns to the system. More specifically, the information that's stored relates to: the chunks of the files stored, the replication degrees of the nodes that store the given peer's file and the peers that have been offline when a delete was issued, so they must be warned when they come back online (this relates to the next point). An example of this information saving:

```
/**
 * Writes to files in the directory to save the information present on the tables.
 */
synchronized private void saveToDirectory() {
    // Saving perceived replication table
    try {
        FileOutputStream percRepFileOut = new FileOutputStream( name: this.directory + perceivedReplicationInfo);
        ObjectOutputStream percRepObjOut = new ObjectOutputStream(percRepFileOut);
        percRepObjOut.writeObject(this.perceivedReplicationTable);
        percRepObjOut.close();
        percRepFileOut.close();
    } catch (Exception ignore) {
    }

    // Saving file deletion list
    try {
        FileOutputStream fileDelFileOut = new FileOutputStream( name: this.directory + fileDeletionInfo);
        ObjectOutputStream fileDelObjOut = new ObjectOutputStream(fileDelFileOut);
        fileDelObjOut.writeObject(this.fileDeletionList);
        fileDelObjOut.close();
        fileDelFileOut.close();
    } catch (Exception ignore) {
    }
}
```

Figure 25. Saving important information to memory in the ChunkManager class

As the last fault-tolerance feature, there's the specific case when a peer goes offline unexpectedly the moment before receiving a delete order by the initiator peer of one of the files

it’s storing. In this case, the initiator peer places that node on a queue, knowing that it must remind it of deleting the file when it comes back online. The peer knows that it has come back online when it receives any message from the disappearing peer. After that detection, the disappearing peer is warned and it promptly deletes the file, as initially intended.