U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Project 2 Report

Class 3, Group 22

Clara Martins (up201806528)
Daniel Monteiro (up201806185)
Gonçalo Pascoal (up201806332)
Teresa Corado (up201806479)

Course: MIEIC 3rd Year
Curricular Unit: Distributed Systems (Sistemas Distribuídos)
Year: 2020/21

1. Overview

The aim of this project is the development of a peer-to-peer distributed backup service for the Internet.

Similarly to the first project, the service we implemented supports the **backup**, **restore** and **deletion** protocols, while allowing the peers to retain total control over their own storage space through the **reclaim** protocol. Files are divided into chunks with a maximum size of 64KB. We also included a test application that uses RMI to communicate with peers, in order to facilitate testing of the protocols and inspection of the state of the peers.

For the peer-to-peer backup system, we chose to implement the **Chord** protocol, according to the specification present in the 2001 paper: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*.

To achieve a high level of concurrency, we used both thread pools and Java NIO whenever possible, as well as several data structures from the `java.util.concurrent` package, to avoid issues related to accessing shared data.

JSSE, specifically `SSLEngine`, was used to ensure that any messages between the peers were exchanged through secure and encrypted channels.

We also addressed both of the issues detailed in the project specification: the chord design and the use of thread pools with Java NIO contributes to the scalability of the service, and we also implemented mechanisms that allow the service to maintain stability and operability even after unexpected peer failures (fault tolerance).

Therefore, our project was developed in order to achieve a ceiling grade of 20.

2. Protocols

I. Packages

For a general overview of the project, we present below a brief description of the purpose of each package present in our application.

- **`chord`**: classes related to the Chord protocol implementation
- **`client`**: classes related to the test application (`TestApp`) and RMI interface
- **`jsse`**: classes related to secure communication using JSSE
- **`messages`**: classes related to the different messages exchanged between the peers
- **`protocol`**: classes related to the implementation of the distributed backup service, including the `Peer` main class
- **`utils`**: miscellaneous and utility classes
- **`workers`**: worker threads that perform several tasks related to the distributed backup service

II. Test Application and RMI

The testing application (`client.TestApp`) connects to the RMI registry and looks up the access point given as one of the arguments on the command line. After connecting to the access point, it calls the protocol requested on the command line.

Valid command-line argument combinations are:

- **BACKUP** protocol: `<access_point> BACKUP <file_path> <replication_degree>`
- **RESTORE** protocol: `<access_point> RESTORE <file_path>`
- **DELETE** protocol: `<access_point> DELETE <file_path>`
- **RECLAIM** protocol: `<access_point> RECLAIM <disk_space>`
- **STATE** protocol: `<access_point> STATE`

The replication degree must be a value between 1 and 9 (both inclusive), and the disk space in the **RECLAIM** protocol is specified in KB. We also included a shell script (`test.sh`) to make it easier to run the testing application.

```java
public interface ClientInterface extends Remote {
    void backup(String filePath, int replicationDegree) throws RemoteException;
    void restore(String filePath) throws RemoteException;
    void delete(String filePath) throws RemoteException;
    void reclaim(long diskSpace) throws RemoteException;
    PeerState state() throws RemoteException;
}
```

The previous code snippet (`client.ClientInterface`) showcases the remote interface that the Peer implements. Each function maps to one of the protocols of the test application.

III. Command Line Arguments (`Peer`)

The peer application can be executed using the following command:

```
java protocol.Peer <protocol_version> <peer_id> <service_ap>
<keystore_path> <truststore_path> <password> <addr> <port>
[<chord_addr> <chord_port>]
```

The list below offers a brief explanation of each of the command-line arguments. The first argument was ported from the first project, but is currently not being used:

- **`protocol_version`**: version of the protocol to be used *(unused)*
- **`peer_id`**: id of the peer
- **`service_ap`**: RMI access point of the peer
- **`keystore_path`**: path to the key store (JSSE)
- **`truststore_path`**: path to the truststore (JSSE)

- **`password`**: password to access the keystore and truststore
- **`addr`**: IP address of the node
- **`port`**: port number of the node
- **`chord_addr`**: IP address of the chord node to contact when joining an existing Chord ring *(optional)*
- **`chord_port`**: port number of the chord node to contact when joining an existing Chord ring *(optional)*

We also included two shell scripts to facilitate execution (`peer.sh` and `peer_simple.sh`, which uses default values for some of the arguments).

IV. Messages

The messages exchanged by peers in our program all follow the same generic formatting (CRLF means *carriage return, line feed*):

```
<Header><CRLF><CRLF>[<Body>]
```

These messages can be divided into two groups: the messages related to the Chord protocol and the messages related to the distributed backup service. The following section explains and showcases the format of each message.

Chord Protocol Messages

- **`NOTIFY`**: sent when a node wishes to notify its successor of its existence. This message is part of the stabilization protocol of Chord that runs periodically.

```
<Version> NOTIFY <SenderId> <InitiatorKey> <InitiatorHostname>
<InitiatorPort> <CRLF><CRLF>
```

- **`GET_PREDECESSOR`**: sent when a node wishes to find out the predecessor of another node. This message is also used during the stabilization protocol.

```
<Version> GET_PREDECESSOR <SenderId> <InitiatorKey>
<InitiatorHostname> <InitiatorPort> <CRLF><CRLF>
```

- **`PREDECESSOR`**: sent as a response to a `GET_PREDECESSOR` message. This message may or may not contain information about the predecessor, depending on whether the predecessor is null.

```
<Version> PREDECESSOR <SenderId> [<PredecessorKey>
<PredecessorHostname> <PredecessorPort>] <CRLF><CRLF>
```

- **GET_SUCCESSOR**: sent when a node wishes to find out the successor of another node. It is used to keep a double-ended queue (`ConcurrentLinkedDeque`) of potential successors that will replace the current successor if it fails.

```
<Version> GET_SUCCESSOR <SenderId> <InitiatorHostname>
<InitiatorPort> <CRLF><CRLF>
```

- **NODE_SUCCESSOR**: sent as a response to a `GET_SUCCESSOR` message, it contains information about the node's successor, such as Chord key, address and port.

```
<Version> NODE_SUCCESSOR <SenderId> <SuccessorKey>
<SuccessorHostname> <SuccessorPort> <CRLF><CRLF>
```

- **FIND_SUCCESSOR**: sent when a node wishes to find out the successor of a certain key (that is, the node responsible for it). This message will be forwarded around the chord ring until the successor is known, at which point a `SUCCESSOR` message will be sent to the initiator.

```
<Version> FIND_SUCCESSOR <SenderId> <Key> <InitiatorHostname>
<InitiatorPort> <CRLF><CRLF>
```

- **SUCCESSOR**: sent as a response to a `FIND_SUCCESSOR` message, it contains information about the successor of the key.

```
<Version> SUCCESSOR <SenderId> <Key> <SuccessorKey>
<SuccessorHostname> <SuccessorPort> <CRLF><CRLF>
```

- **ALIVE**: sent periodically to the node's predecessor to verify that it is still operational.

```
<Version> ALIVE <SenderId> <CRLF><CRLF>
```

Distributed Backup Service Messages

- **PUT_CHUNK**: sent during a BACKUP protocol, a RECLAIM protocol or after the reception of a `START_PUT_CHUNK` message; its body contains the data of a chunk that the initiator peer wishes to backup. Information about the chunk, such as the **file id** and **chunk number** is included, as well as the replication degree, which is the remaining amount of copies of the chunk that should be stored (this number decreases as peers store copies of the chunk and the message travels

around the Chord ring). The message also includes the address and port of the initiator peer, so that the peers that store the chunk can contact it.

```
<Version> PUT_CHUNK <SenderId> <FileId> <ChunkNo> <ReplicationDeg>
<InitiatorHostname> <InitiatorPort> <CRLF><CRLF><Body>
```

- **STORED**: sent as a response to a PUT_CHUNK message after the peer has stored the chunk contained in it. Contains the **file id** and the **chunk number** of the stored chunk as well as the address and port of the sender, so that the initiator peer can contact it if it needs to recover the chunk.

```
<Version> STORED <SenderId> <FileId> <ChunkNo> <SenderAddress>
<SenderPort> <CRLF><CRLF>
```

- **GET_CHUNK**: sent during a RESTORE protocol, the initiator peer asks a peer for the chunk with the specified **file id** and **chunk number**. If the peer that receives the message does not have the desired chunk, it will forward the message to its successor.

```
<Version> GET_CHUNK <SenderId> <FileId> <ChunkNo>
<InitiatorHostname> <InitiatorPort> <CRLF><CRLF>
```

- **CHUNK**: contains a chunk that was requested by a GET_CHUNK message.

```
<Version> CHUNK <SenderId> <FileId> <ChunkNo> <CRLF><CRLF><Body>
```

- **DELETE**: sent during a DELETE protocol to the peers that store chunks of the specified file. After receiving this message, the peers will delete any chunks of that file from their file systems.

```
<Version> DELETE <SenderId> <FileId> <CRLF><CRLF>
```

- **REMOVED**: sent during a RECLAIM protocol or as a response to a VERIFY_CHUNK message, informing the initiator peer that the peer that sent this message is no longer storing a copy of the specified chunk.

```
<Version> REMOVED <SenderId> <FileId> <ChunkNo> <SenderHostname>
<SenderPort> <CRLF><CRLF>
```

- **VERIFY_CHUNK**: periodically sent by the initiator peer to a random peer that has stored a copy of a certain chunk. It contains the chunk's identification (**file id** and

chunk number) and the address and port of the initiator peer. It verifies that the peer is alive and still storing the chunk. If the peer is no longer storing the specified chunk, it must send a `REMOVED` message to the initiator peer, using the contact information in the `VERIFY_CHUNK` message.

```
<Version> VERIFY_CHUNK <SenderId> <FileId> <ChunkNo>
<InitiatorHostname> <InitiatorPort> <CRLF><CRLF>
```

- **`START_PUT_CHUNK`**: part of the fault tolerance mechanism, this message is sent to one of the peers that store a copy of a chunk when its replication degree drops below its desired value. It asks the peer to initiate a chunk backup protocol for that chunk, providing the information needed for a `PUT_CHUNK` message.

```
<Version> START_PUT_CHUNK <SenderId> <FileId> <ChunkNo>
<ReplicationDeg> <InitiatorHostname> <InitiatorPort> <CRLF><CRLF>
```

V. Chord Implementation

Before describing the behaviour of the distributed backup service, we shall explain how the Chord protocol that supports it was implemented. We will mention the data structures used, explain how nodes join the network and how information about the network is maintained and updated and describe the tasks that run periodically.

Each node in a Chord ring is identified by a unique *m*-bit key (for our implementation we chose *m* = 10 for simplicity, meaning 1024 possible identifiers). In our implementation, this value can be easily changed to allow the service to scale up. The key of each node is calculated using a consistent hashing function (in our case SHA-1), whose input is the IP address and port of the node. The *m* least significant bits of the hash (which is 160 bits long for SHA-1) are used for the node's key. Using a consistent hashing function is recommended in the Chord paper to minimize the disruption caused by nodes entering and leaving. The `generateKey` function in the code snippet below shows how keys are calculated.

```java
/**
 * Generates an m-bit key to be used for the Chord protocol from a sequence of bytes. The key is
 * generated using a consistent hashing algorithm, in this case SHA-1. The 160-bit SHA-1 has is truncated
 * to a length of m bits, using the least significant bits.
 */
public static long generateKey(byte[] input) throws NoSuchAlgorithmException {
    MessageDigest digest = MessageDigest.getInstance("SHA1");
    byte[] sha1Bytes = digest.digest(input);

    long key = 0;
    // The length of the key in bytes, rounded up (for example, a 20-bit key would need 3 bytes)
    int keyNumBytes = (int) Math.ceil((double) keyBits / 8);

    // Obtain the m least significant bits from the 160-bit SHA-1 hash
    // This is equivalent to obtaining the 160-bit hash modulo 2^m
    for (int i = 0; i < keyNumBytes; ++i) {
        byte b = sha1Bytes[sha1Bytes.length - 1 - i];
        key |= Byte.toUnsignedInt(b) << (8 * i);
    }

    // This bitmask is needed whenever m is not a multiple of 8,
    // since we need to guarantee that the generated keys are always smaller than 2^m
    long mask = (long) Math.pow(2, keyBits) - 1;
    key &= mask;

    return key;
}
```

Each node is responsible for the keys directly preceding it. This means that we also need to calculate the Chord keys for the chunks that are stored by the peers in the backup service. To do this, we use the same process as when calculating the keys for the Chord nodes, except the input provided to the hashing function is: `<file_id>_<chunk_number>`.

When a node joins without specifying contact information in its command-line arguments, it creates a new Chord ring. When a new node wants to join that ring, the IP

address and port of a node already in the ring need to be provided as command line arguments.

The code snippet below (`joinNetwork` function from the `ChordNode` class) shows the join procedure. The node that wants to join the ring asks the contact node to locate its successor, using a `FIND_SUCCESSOR` message (`FindSuccessorMessage` class). This message will travel around the Chord ring, taking advantage of the finger tables as much as possible, and be answered with a `SUCCESSOR` message (`SuccessorMessage` class), which contains the key, IP address and port of the node's successor.

```java
public void joinNetwork(InetSocketAddress contact) {
    // Called when the node is joining a new Chord network
    initializeFingerTable();

    long successorStart = getStartKey( k: 0);

    tasksMap.putIfAbsent(successorStart, new ConcurrentLinkedQueue<>());
    tasksMap.get(successorStart).add((nodeInfo) -> {
            startPeriodicTasks();

            // Update the new node's finger table
            try {
                for (int i = 1; i < keyBits; ++i) {
                    long fingerStart = selfInfo.id + (long) Math.pow(2, i);
                    FindSuccessorMessage message = new FindSuccessorMessage(Peer.version, Peer.id, fingerStart, Peer.address);
                    ClientThread thread = new ClientThread(contact, message);
                    Peer.executor.execute(thread);
                }
            }
            catch (IOException | GeneralSecurityException ex) {
                System.err.println("Error when updating new node's finger table: " + ex.getMessage());
            }
    });

    FindSuccessorMessage message = new FindSuccessorMessage(Peer.version, Peer.id, successorStart, Peer.address);

    try {
        ClientThread thread = new ClientThread(contact, message);
        Peer.executor.execute(thread);
    }
    catch (IOException | GeneralSecurityException ex) {
        System.err.println("Error when sending FIND_SUCCESSOR message: " + ex.getMessage());
    }
}
```

The `FIND_SUCCESSOR` message is used to find the node responsible for a certain key. The initiator peer sends this message to the key's closest preceding node in the finger table. When receiving this message, the peer will have two options: sending a `SUCCESSOR` message, if its successor is the node that the initiator peer wanted to identify, or employing the same procedure of contacting the key's closest preceding node. When the finger tables are correct, this leads to a time complexity of O(log n) for `FIND_SUCCESSOR` queries, where n is the number of nodes in the network.

Afterwards, the node will attempt to update the rest of its finger table (a list with *m* entries of nodes in the Chord ring that speeds up queries to find the successor of a key) and start several periodic tasks related to the Chord protocol.

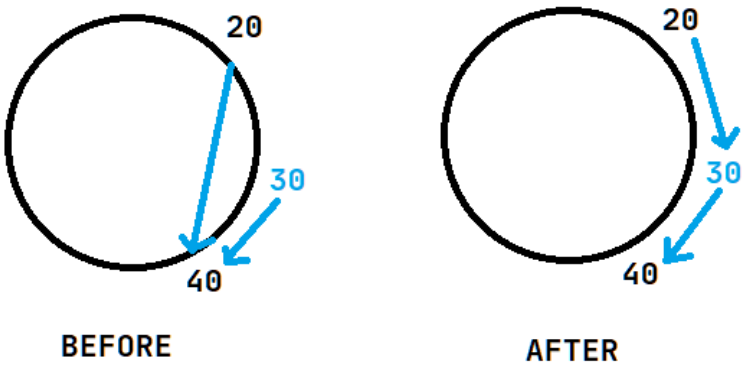
```java
// Schedule FixFingersThread to execute periodically
FixFingersThread fixFingersThread = new FixFingersThread();
Peer.executor.scheduleWithFixedDelay(fixFingersThread, initialDelay: 0, delay: 300, TimeUnit.MILLISECONDS);

// Schedule StabilizationThread to execute periodically
StabilizationThread stabilizationThread = new StabilizationThread();
Peer.executor.scheduleWithFixedDelay(stabilizationThread, initialDelay: 0, delay: 2, TimeUnit.SECONDS);

// Schedule FindSuccessorsThread to execute periodically
FindSuccessorsThread findSuccessorsThread = new FindSuccessorsThread();
Peer.executor.scheduleWithFixedDelay(findSuccessorsThread, initialDelay: 0, delay: 10, TimeUnit.SECONDS);
```

The periodic tasks required by the Chord protocol can be seen in the code snippet above, from the `startPeriodicTasks` function of the `ChordNode` class. The `FixFingersThread` periodically attempts to fix one of the fingers in the finger table, using the previously mentioned `FIND_SUCCESSOR` queries. The next execution of `FixFingersThread` will fix the following finger, and so on, looping back to the start of the finger table once the last finger is fixed.

To understand the purpose of the `StabilizationThread`, it is important to note that each node also keeps a pointer to its predecessor. This pointer initially starts out as being null. The `StabilizationThread` performs several functions: first, it asks the node's successor for its predecessor *p* (using a `GET_PREDECESSOR` message, from the `GetPredecessorMessage` class) and decides whether or not *p* should be the node's successor instead (this is done when handling the `PREDECESSOR` message, from the `PredecessorMessage` class). This procedure is shown in the diagram below.



Secondly, the `StabilizationThread` also notifies the successor of the node of its existence, so that it can update its predecessor pointer. This is done through a `NOTIFY` message (`NotifyMessage` class), which has no reply. Finally, an `ALIVE` message (`AliveMessage` class) is sent to the node's predecessor, if it exists, to verify that it is still operational. If the message isn't transmitted successfully, the predecessor pointer is set to null.

The FindSuccessorsThread class is explained in detail in the fault tolerance section of this report.

VI. Backup

When a backup protocol is initiated, the specified file is divided into chunks and each chunk is backed up separately. A worker thread from the ReadChunkThread class is started for each chunk of the file, and an AsynchronousFileChannel from the Java NIO package is opened to allow several threads to read concurrently from the file. Since the threads all read distinct and disjunct sections of the file, there should be no concurrency issues related to file I/O.

The thread will then calculate the Chord key of the chunk from the file id and the chunk number (as described previously) and then attempt to find the node responsible for this key, usually by sending a FIND_SUCCESSOR message with the chunk's key (unless the node responsible for the chunk is the peer's successor in the Chord ring).

When the peer knows the successor of the chunk's key, it will send a PUT_CHUNK message (PutChunkMessage class) to that node, containing the chunk's data, identification and desired replication degree (number of copies that should be stored in the system).

When receiving a PUT_CHUNK message, a peer will try to store a copy of the chunk in its storage system if it has enough space available (see the reclaim protocol for more information). When the chunk is successfully stored, it will send a STORED message (StoredMessage class) to the initiator peer, which will keep this information in its state, so that it knows which peers have stored a copy of the chunk. If the replication degree hasn't been reached (meaning its value is still positive) and the chunk hasn't traveled around the entire Chord ring, the peer will forward the PUT_CHUNK message to its successor, reducing the replication degree by 1 if it has stored a copy of the chunk.

A chunk will be stored in the following path in the peer's file system (the peer<id> folder allows several peers to run in the same machine):

```
peer<id>/<file_id>/<chunk_number>
```

It is important to note that the initiator peer will never store copies of its own chunks, therefore when it receives a PUT_CHUNK message of one of its chunks, it will simply forward the message to its successor.

VII. Restore

Since files are stored as chunks in the backup service, in order to restore a file, we need to obtain each chunk separately and reconstruct the file on the initiator peer.

Since the initiator peer has the IP addresses of the peers that have stored copies of each chunk, and the peer that initiates a restore protocol should be the peer that originally asked for the backup of the file, we can send a `GET_CHUNK` message (`GetChunkMessage` class) directly to one of the peers that should have the chunk stored.

Before sending these messages, the initiator peer will also open an `AsynchronousFileChannel` from the Java NIO package for the restored file.

If the peer that receives the `GET_CHUNK` message has the requested chunk in its storage, it will read the chunk's data and forward it to the initiator peer in a `CHUNK` message (`ChunkMessage` class). If the peer doesn't have a copy of the requested chunk or if it fails to send the `CHUNK` message, it will forward the `GET_CHUNK` message to its successor.

When the initiator peer tries to send a `GET_CHUNK` message and notices that the recipient of the message has failed, it will send the message to the next peer that has stored the chunk, which makes this protocol more resilient to node failures.

After receiving a `CHUNK` message, the initiator peer will create a worker thread from the `RestoreChunkThread` class, which will write the message body to the file that is being restored, at the correct offset, calculated using the chunk number (`offset = 64000 * chunkNumber`).

The advantage of using Java NIO for this task is that the chunks can be written in any order: if the file is too small for the designated offset, it will be automatically enlarged. This approach improves the concurrency of the protocol.

The restored file will be present in the following path:

```
peer<id>/restored/<file_id>/<file_name>
```

VIII. Delete

When a delete protocol is initiated, the peer that initiated the backup of the file will send a `DELETE` message (`DeleteMessage` class) to all the peers that have stored at least one of the chunks of that file.

Upon receiving this message, the peers will then delete every chunk they have stored from that file.

IX. Reclaim

The reclaim protocol allows the peers in the network to retain full control of their storage. When the reclaim protocol is initiated, the maximum storage space that can be used for backing up chunks is set. If the space occupied by the chunks the peer is currently backing up exceeds this value, the peer will begin to delete chunks in order to

free up space, starting with the largest ones (this ensures that the fewest amount of chunks will be deleted).

Before removing a chunk from its storage, the peer needs to ensure that the replication degree of that chunk is maintained. Therefore, the peer will first read the contents of the chunk and initiate a chunk backup protocol, with a replication degree of 1. This procedure is similar to the one described above in the backup protocol. After removing the chunk from its storage, the peer will send a `REMOVED` message (`RemovedMessage` class) to the peer that initiated the backup of the chunk. These actions are performed by a worker thread from the `RemoveChunkThread` class.

Upon receiving a `REMOVED` message, the initiator peer will remove this peer from the list of peers who have stored the specified chunk.

To avoid any synchronization issues with the chunk backup protocol, the selection of chunks to remove is performed inside a `synchronized` block, temporarily preventing any `StoreChunkThread` instances from storing more chunks. In order to reduce the overhead caused by this synchronization, only the high level data structures that contain information about the stored chunks are changed; the actual file I/O is done afterwards.

3. Concurrency design

Concurrency is an important part of performance reliant distributed systems. In this section of the report, we will describe the steps taken to increase concurrency and minimize synchronization issues.

For each message that the peer wants to send, an instance of the `ClientThread` class from the `jsse` package is created and executed. This thread establishes a secure connection using JSSE and transmits the desired message. If an error occurs during the transmission of the message, the program assumes that the destination peer has failed, and initiates fault tolerance procedures based on the recipient of the message. This allows multiple messages to be sent concurrently.

Each message that the peer receives is forwarded to a thread from the `HandleReceivedMessageThread` class, which divides the message into header and body, parses the message into one of the classes that extend `Message` and then executes any required tasks depending on the message type.

```
private void handleChunkMessage(ChunkMessage message) {
    RestoreChunkThread thread = new RestoreChunkThread(message);
    Peer.executor.execute(thread);
}
```

As an example, the previous code snippet shows one of the methods that can be called: whenever a valid `CHUNK` message is received, a `ReceivedChunkThread` is created and executed (this thread will add the received chunk to the restored file). This

allows several messages to be processed concurrently or new messages to be received whilst others are being processed.

Some critical sections, such as deciding whether or not to store a chunk when the space reserved for chunk backup is limited (see `StoreChunkThread` class), had to be executed inside `synchronized` blocks. We strived to make these blocks as short as possible and to take advantage of thread-safe data structures whenever possible.

To avoid synchronization issues when accessing data that was shared between threads, choosing the right data structures was an important factor.

```
// For each file ID + chunk number, this hash map stores a set with the addresses and ports of all peers who have
// backed up that chunk. The actual replication degree of the chunk, therefore, is the size of the set.
public ConcurrentHashMap<ChunkIdentifier, Set<InetSocketAddress>> chunkReplicationDegreeMap = new ConcurrentHashMap<>();

// Hash map containing information about the files that this peer has initiated the backup of
public ConcurrentHashMap<String, FileInformation> backupFilesMap = new ConcurrentHashMap<>();

// Maps file IDs to the desired replication degree of their chunks
public ConcurrentHashMap<String, Integer> desiredReplicationDegreeMap = new ConcurrentHashMap<>();

// Hash map containing information about chunks this peer is backing up
public ConcurrentHashMap<ChunkIdentifier, ChunkInformation> storedChunksMap = new ConcurrentHashMap<>();
```

Thereby, we made extensive use of the `ConcurrentHashMap`, as shown in the code snippet above, which provides a thread-safe hash map. We also used the `newKeySet` method from this class to create thread-safe sets, given that the use of sets instead of lists improves performance while, at the same time, ensuring that all values are unique.

```
// AtomicReferenceArray is used to ensure thread safety
public AtomicReferenceArray<ChordNodeInfo> fingerTable = new AtomicReferenceArray<>(keyBits);

public final ConcurrentHashMap<Long, Queue<ChordTask>> tasksMap = new ConcurrentHashMap<>();
```

To represent the finger table which is part of the Chord's protocol, we used an `AtomicReferenceArray`, since it allows us to update fingers atomically and since the size of the finger table is constant and known at compile time. The `tasksMap` maps a Chord key to a queue of `ChordTask` instances. When a SUCCESSOR message is received, each task in the queue will be executed in a separate thread, with the information provided by the message (identification of the node that is the successor of the requested key).

```
public static final int MAX_THREADS = 25;
public static ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(MAX_THREADS);
```

The previous code snippet illustrates the use of thread pools, more specifically, `ScheduledThreadPoolExecutor`. Using thread pools reduces the overhead associated with starting new threads and allows us to schedule a thread to be executed

periodically, which was used both in the context of the Chord protocol and fault tolerance mechanisms.

To reduce the overhead associated with file I/O and to allow threads to read or write concurrently to files, we used the Java NIO package, specifically the `AsynchronousFileChannel` class, as was already described in the sections related to the backup and restore protocols.

4. JSSE

To ensure secure and encrypted communication between peers, we used JSSE, specifically the `SSLEngine` class. This interface offers more flexibility than standard SSL sockets and allows for a higher level of concurrency. In our implementation, all messages exchanged between peers are sent through secure channels. The protocol used was TLS and we used the default algorithms for `KeyManagerFactory` and `TrustManagerFactory`. The following code snippet from the `SSLThread` class shows how the key and trust managers are obtained.

```
private KeyManager[] getKeyManagers(String keyStorePath, String password) throws GeneralSecurityException, IOException {
    KeyManagerFactory factory = KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());

    File clientKeys = new File(keyStorePath);
    KeyStore keyStore = KeyStore.getInstance(clientKeys, password.toCharArray());

    factory.init(keyStore, password.toCharArray());
    return factory.getKeyManagers();
}

private TrustManager[] getTrustManagers(String trustStorePath, String password) throws GeneralSecurityException, IOException {
    TrustManagerFactory factory = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());

    File trustStoreFile = new File(trustStorePath);
    KeyStore trustStore = KeyStore.getInstance(trustStoreFile, password.toCharArray());

    factory.init(trustStore);
    return factory.getTrustManagers();
}
```

In order to use `SSLEngine`, we first need to instantiate an `SSLContext`, as shown in the code snippet below.

```
context = SSLContext.getInstance(protocol);
context.init(getKeyManagers(keyStorePath, password), getTrustManagers(trustStorePath, password), new SecureRandom());
```

This is done in the constructor of the `SSLThread` class, an abstract superclass of all the threads that utilize secure sockets, such as `ClientThread` or `ServerThread`. The chosen protocol (for example, TLS) is passed to the `getInstance` function and the key and trust managers (obtained through the `keyStorePath`, `trustStorePath` and `password` command-line arguments) are used to initialize the context.

After this initialization is complete, we can obtain `SSLEngine` instances using the `createSSLEngine` function from `SSLContext`.

```java
SSLEngine engine = context.createSSLEngine(destinationAddress.getAddress().getHostAddress(), destinationAddress.getPort());
engine.setUseClientMode(true);

try {
    SocketChannel socketChannel = SocketChannel.open();

    socketChannel.connect(destinationAddress);
    socketChannel.finishConnect();
    socketChannel.configureBlocking(false);

    doHandshake(socketChannel, engine);
    sendMessage(socketChannel, engine);
}
```

The code snippet above shows how a connection between a client and the peer it wishes to contact is established: first, an instance of `SSLEngine` is created using the address and port of the destination peer, a `SocketChannel` is created and set up to exchange data, the SSL/TLS handshake is performed (`doHandshake` function) and finally, the client can forward its message in a secure and encrypted way.

The SSL/TLS handshake involves an exchange of several messages between the two parties, which relate to cipher suite negotiation, key exchanges and authentication. In order to implement this, we adapted the code snippet provided in the JSSE documentation.

It is important to note that instances of `SSLEngine` are unusable after the connection is closed, therefore a new instance must be created for each connection. Also, the peer who reads the last application-related message must be the one to close the connection (in our implementation, this peer is the server). This ensures that the connection isn't closed while a peer is still receiving a message.

5. Scalability

Scalability is an important part of distributed services. In our program, this is ensured by the Chord protocol implementation (since Chord was designed with scalability in mind) and through the use of thread pools (specifically the `ScheduledThreadPoolExecutor` class) and Java NIO.

Using thread pools reduces the overhead of creating new threads, limits the total amount of threads at any given time (since having a very large number of threads executing concurrently can be detrimental to performance) and allows scheduling tasks to run periodically (for example, the stabilization and finger fixing procedures in the Chord specification, shown in the code snippet below).

```java
private void startPeriodicTasks() {
    // Schedule FixFingersThread to execute periodically
    FixFingersThread fixFingersThread = new FixFingersThread();
    Peer.executor.scheduleAtFixedRate(fixFingersThread, initialDelay: 0, period: 250, TimeUnit.MILLISECONDS);

    // Schedule StabilizationThread to execute periodically
    StabilizationThread stabilizationThread = new StabilizationThread();
    Peer.executor.scheduleAtFixedRate(stabilizationThread, initialDelay: 0, period: 2, TimeUnit.SECONDS);
}
```

Java NIO (specifically `AsynchronousFileChannel`) was used in the **backup** and **restore** protocols to allow multiple threads to read from or write to the same file concurrently, as explained in the protocols section of the report. This reduces some of the overhead associated with file I/O and contributes to a higher degree of concurrency.

6. Fault tolerance

The goal of fault tolerance is to avoid single-points of failure and to ensure that the service remains stable and operational whenever peers fail in an unexpected way. To achieve this, we implemented several mechanisms relating to both the Chord protocol and the backup service.

Since the Chord protocol relies on the successor pointers being correct, each node also keeps a list of n successors after its own successor (for our implementation, we chose a value of 2, but this value can be easily increased for more robustness). If the node detects that its successor has failed, it will replace any entries of that node in the finger table with the next successor in the list.

In addition, a thread (`FindSuccessorsThread` in the `chord` package) runs periodically in the background. If the list of successors isn't full, the thread will send `GET_SUCCESSOR` messages (`GetSuccessorMessage` class), which ask a node for its successor. When the node replies with a `NODE_SUCCESSOR` message (`NodeSuccessorMessage` class), that successor is stored in the list and the process is repeated with the new successor until either the successor list is full or a full trip around the Chord ring is completed.

```java
// Fault tolerance: in addition to its successor, the peer keeps the addresses of n successors, so that it can
// continue operating if its successor fails
public static final int numSuccessors = 2;
public final ConcurrentLinkedDeque<ChordNodeInfo> successorDeque = new ConcurrentLinkedDeque<>();
```

As shown in the code snippet above (`ChordNode` class), we decided to use a double-ended queue since this ensures that the first successor to be inserted in the queue would be used if the successor of a node failed (*first in, first out*), but also allows us to obtain the element at the back of the queue, in order to ask it for its successor.

In order to maintain the desired replication degree of a chunk when one of the peers storing it fails, the `VerifyChunksThread` periodically runs in the background and

sends `VERIFY_CHUNK` messages (`VerifyChunkMessage` class) for each chunk backed up by the initiator peer to a random peer that stores that chunk. If the communication fails or the peer replies with a `REMOVED` message, the initiator removes that peer from the list of peers storing the chunk.

The `CheckReplicationDegreeThread`, which also runs periodically, checks if the perceived replication degree of any chunk has dropped below its desired value. If it has, the initiator peer asks one of the peers that are storing the chunk to initiate a chunk backup protocol through a `START_PUT_CHUNK` message (`StartPutChunkMessage` class), since after the initial backup of a file there is no guarantee that the initiator peer will still have a copy of that file.

These mechanisms ensure that the replication degree of chunks remains relatively consistent in the event of peer failures, assuming that at least one of the peers that store the chunk remains operational.