Distributed Systems Second Assignment

Distributed Backup System for the Internet

Faculdade de Engenharia da Universidade do Porto

Pedro Emanuel de Sousa Pinto - up201806252@fe.up.pt
Pedro Miguel da Costa Azevedo - up201806728@fe.up.pt
João Alexandre Lobo Cardoso - up201806532@fe.up.pt
Tomás Costa Fontes - up201806252@fe.up.pt

Contents

1. Overview

This project implements a distributed backup system for the internet. Our implementation is built on top of the chord protocol and makes use of the *Java Secure Socket Extension (JSSE)* in order to ensure the security of the communication between peers of the network. Furthermore, we utilise thread-pools in order to achieve a higher level of scalability and concurrency. Lastly, fault-tolerance was also considered in this implementation and some features were implemented for this reason, like an operation that informs a peer's neighbours in the network of its files, which will be explained later in this report.

The implemented backup service supports the following operations:

- **backup** - this operation allows a peer to backup one of its files on other peers from the same network. Each file is divided into chunks that have a maximum size of 64000 bytes, and each chunk is propagated through the network until the number of peers that stored the chunk satisfies the desired replication degree for the chunk's file.
- **delete** - the delete operation allows a peer to delete all the chunks from one of the files it had previously backed up. For this, a delete message is propagated through the network and each peer deletes all the chunks they have that belong to the specified file.
- **restore** - the restore operation allows a peer to rebuild a file it had backed up previously. The peer that initiates this protocol sends a message to the other peers of the network that hold chunks from the specified file and receives the corresponding chunks. In the end, the file is rebuilt and placed in the *restored* folder, inside the initiator peer's filesystem.
- **reclaim** - this operation allows a peer to specify how much of its filesystem's space it wants to be available for the storing of chunks. The peer checks if it needs to delete chunks from its filesystem in order to achieve the specified maximum space for storing them and, when deleting chunks during this operation, sends a message to the deleted chunk's original peer, so that it can handle its deletion and trigger a new backup for that chunk if necessary.
- **state** - the state operation displays on the peer's screen information about its filesystem. The displayed information

contains the chunks the peer has stored, the space used by them, the total space available to store chunks, as well as the chunks stored by the peer's neighbours in the network.

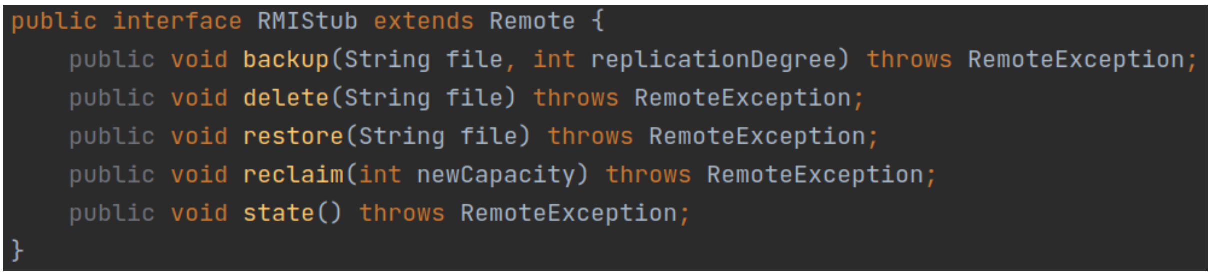
2. Protocols

This section will describe the implemented protocols.

2.1 The Remote Interface

All operations are requested using RMI. Each peer registers itself using an access point received as an argument on the peer's initialization, so that a protocol can be triggered through the remote interface, by the testing/client application.

The remote interface has 5 available functions, represented in the following picture.



```
public interface RMIStub extends Remote {
    public void backup(String file, int replicationDegree) throws RemoteException;
    public void delete(String file) throws RemoteException;
    public void restore(String file) throws RemoteException;
    public void reclaim(int newCapacity) throws RemoteException;
    public void state() throws RemoteException;
}
```

Picture 1: Remote Interface (RMIStub.java file, peers package, lines 9-15

Each method is responsible for triggering the operation described in the method's name.

2.2 Messages

In this section, all the messages used in our implementation will be described. Our implementation of the messages is based on a single *Message* class, which contains a *MessageType* attribute that defines the type of the message. The other attributes of the *Message* class are dependent on the message's type and, depending on it, may be optional or mandatory.

2.2.1 Chord Messages

The following messages are used by the chord protocol:

- **CHECK**
    - Purpose - to check if a chord node is alive
    - Arguments - does not require any additional arguments
    - Response - requires an ALIVE message

- **ALIVE**
    - Purpose - to tell a chord node that the current node is alive
    - Arguments - does not require any additional argument
    - Response - this is a response to the CHECK message

- **FIND_PREDECESSOR**
    - Purpose - to ask a chord node for its predecessor
    - Arguments - does not require any additional argument
    - Response - requires a PREDECESSOR message

- **PREDECESSOR**
    - Purpose - to tell a chord node the predecessor of the current node
    - Arguments - requires a FingerTableEntry object that represents the node's predecessor
    - Response - this is a response to the FIND_PREDECESSOR message

- **FIND_SUCCESSOR**
    - Purpose - to find the successor of a chord node
    - Arguments - requires an integer, which is the id of the node to find the successor
    - Response - requires a SUCCESSOR message

- **SUCCESSOR**
    - Purpose - to tell a chord node the successor of a specified node
    - Arguments - requires an integer, the id of the node to find the successor, and a FingerTableEntry object that represents the successor found for the target node

  - Response - this is a response to the FIND_SUCCESSOR message

- **NOTIFICATION**
  - Purpose - to tell a chord node that the current node might be its predecessor
  - Arguments - requires a FingerTableEntry object that represents the current node
  - Response - requires a NOTIFIED message

- **NOTIFIED**
  - Purpose - to tell a chord node that the current node has been notified
  - Arguments - does not require any additional argument
  - Response - this is a response to the NOTIFICATION message

- **JOIN**
  - Purpose - to tell a chord node that the current node wants to join its chord ring
  - Arguments - requires a FingerTableEntry object that represents the current node
  - Response - requires a SUCCESSOR message

- **INFOSUCC**
  - Purpose - to send the current node's information to its successor
  - Arguments - requires a SubProtocolsData object that has to hold a vector containing information about the chunks stored on the current peer
  - Response - does not require a response

- **INFOPRED**
  - Purpose - to send the current node's information to its predecessor
  - Arguments - requires a SubProtocolsData object that has to hold a vector containing information about the chunks stored on the current peer
  - Response - does not require a response

2.2.2 Protocols Messages

The following messages are exchanged during the execution of the protocols:

- **PUTCHUNK**
    - Purpose - to tell a peer to store a chunk, if possible
    - Arguments - requires a SubProtocolsData object that has to hold the chunk to be stored and the desired replication degree for the chunk
    - Response - requires a STORED or FAILED message

- **STORED**
    - Purpose - to tell a peer that the current peer has stored a previously received chunk
    - Arguments - requires a SubProtocolsData object that has to hold the chunk to be stored, the current replication degree for the chunk and a vector containing the information of the peers that have stored the chunk
    - Response - this is a response to the PUTCHUNK message

- **FAILED**
    - Purpose - to tell a peer that the current backup protocol has failed
    - Arguments - requires a SubProtocolsData object that has to hold the chunk to be stored, the current replication degree for the chunk and a vector containing the information of the peers that have stored the chunk
    - Response - this is a response to the PUTCHUNK message

- **DELETE**
    - Purpose - to tell a peer to delete the chunks of a certain file
    - Arguments - requires a SubProtocolsData object that has to hold the file id of the file whose chunks will be deleted
    - Response - requires a DELETE message

- **DELETED**
    - Purpose - to tell a peer that the current peer has deleted all the chunks from a certain file
    - Arguments - does not require any additional argument
    - Response - this is a response to the DELETE message

- **GETCHUNK**
    - Purpose - to ask a peer for a certain chunk
    - Arguments - requires a ChunkInfo object that holds the information about the chunk
    - Response - requires a CHUNK message

- **CHUNK**
    - Purpose - to send a chunk to another peer
    - Arguments - requires a SubProtocolsData object that has to hold the chunk to be sent
    - Response - this is a response to the GETCHUNK message

- **DECREASE_REP_DEGREE**
    - Purpose - to tell a peer to decrease the current replication degree of a certain chunk
    - Arguments - requires a ChunkInfo object that holds the information about the target chunk
    - Response - does not require a response

2.3 Backup Protocol

When a backup is triggered, the initiator peer starts by dividing the target file in chunks and generating the file's unique id. After that, for each chunk, a PUTCHUNK message is sent to the peer's successor in the chord network, containing the chunk to be stored. When receiving a PUTCHUNK message, a peer first checks if it already has the received chunk or not. If it already has the chunk or if it can store it, then the peer decreases the chunk's current replication degree. If the new replication degree is zero, then the peer sends back a response , which will be propagated back to the peer that started the backup. In case the peer cannot store the received chunk because it does not have enough space for it, the message will still be propagated to the peer's successor, however, the chunk's current replication degree will not be decreased.

The following example shows the execution of this protocol, with 4 nodes in the network (with ids 2, 3, 12 and 25), where node with id 3 triggers a backup for one of its chunks with a replication degree value of 2, assuming all nodes have enough space to store the chunk:

1. Node 3 sends a PUTCHUNK message to node 12;

2. Node 12 stores the chunk and decreases its current replication degree to 1;
3. Node 12 sends a PUTCHUNK message to node 25;
4. Node 25 stores the chunk and decreases its current replication degree to 0;
5. Because the chunk's current replication degree is 0, node 25 sends a STORED response to node 12;
6. Node 12 propagates the received response to node 3;
7. Node 3 receives the response and stores the ids of the peers who stored the chunk;

This protocol is implemented in the file *Backup* inside package *subProtocols*.

2.4 Delete Protocol

When a Delete protocol is triggered, the initiator peer propagates a DELETE message through the network, containing the file whose files should be deleted, telling the other peers to delete their chunks from the target file. In order to do this, the initiator peer sends the DELETE message to its successor, who will delete the chunks corresponding to the target file and will then forward the message to its own successor. The message propagation stops when it reaches the peer who triggered the protocol, which then propagates back the response, a DELETED message.

The following example shows the execution of this protocol, with 3 nodes in the network (with ids 2, 3 and 25), where node with id 3 triggers a delete protocol for a certain file:

1. Node 3 sends a DELETE message to node 25;
2. Node 25 deletes the chunks from the received file that it had previously stored in its filesystem;
3. Node 25 sends a DELETE message to node 2;
4. Node 2 deletes the chunks from the received file that it had previously stored in its filesystem;
5. Node 2 sends a DELETE message to node 3;
6. Because node 3 corresponds to the peer that triggered the protocol, node 3 sends a DELETED response to node 2;
7. Node 2 forwards the DELETED response to node 25;
8. Node 25 forwards the DELETED response to node 3;
9. Node 3 finished the protocol;

This protocol is implemented in the file *Delete* inside package *subProtocols*.

2.5 Restore Protocol

When a restore protocol is triggered, the initiator peer, for each chunk of the file to be restored, checks which peers had stored the chunk when the backup was performed and sends a GETCHUNK message to one of those peers. If that peer responds with a CHUNK message containing the chunk, the initiator stores the received chunk and moves on to the next one. If there is no response from the peer, then the initiator sends a GETCHUNk message to another peer who had stored the chunk and repeats this behaviour until it receives a CHUNK response or until there are no more peers storing the target chunk, in which case the protocol has failed.

The following example shows the execution of this protocol, with 3 nodes in the network (with ids 2, 3 and 25), where node with id 3 triggers a restore protocol for a certain file with 1 chunk and both nodes 2 and 25 have that chunk stored in their filesystem:

1. Node 3 sends a GETCHUNK message to node 25;
2. Node 25, for some reason, fails and doesn't send a response containing the target chunk;
3. Node 3 sends a GETCHUNK message to node 2, the only other node which it knows had stored the file's chunk;
4. Node 2 gets the chunk from its filesystem and sends it back to node 3 in a CHUNK response;
5. Node 3 receives the chunk and restores the file;

This protocol is implemented in the file *Restore* inside package *subProtocols*.

2.6 Reclaim Protocol

When a Reclaim protocol is triggered, the initiator peer starts by checking if it needs to delete any chunk in order to satisfy its new capacity for storing chunks. If there is no need to delete chunks, then the peer updates its capacity for storing chunks and ends the protocol. If the peer detects that it needs to delete chunks in order to satisfy the new capacity, then it starts deleting chunks until the capacity is satisfied. For each chunk it deletes, the initiator sends a DECREASE_REP_DEGREE message to the original peer of the chunk, who will then decrease the chunk's current replication degree and trigger a new backup protocol for that chunk if its replication degree drops below the desired value for it.

The following example shows the execution of this protocol, with 3 nodes in the network (with ids 2, 3 and 25), where node with id 3 triggers a reclaim protocol for a new capacity which leads to it having to delete one of its chunks that belonged to a file from peer 2 that had been backed up with a replication degree of 1:

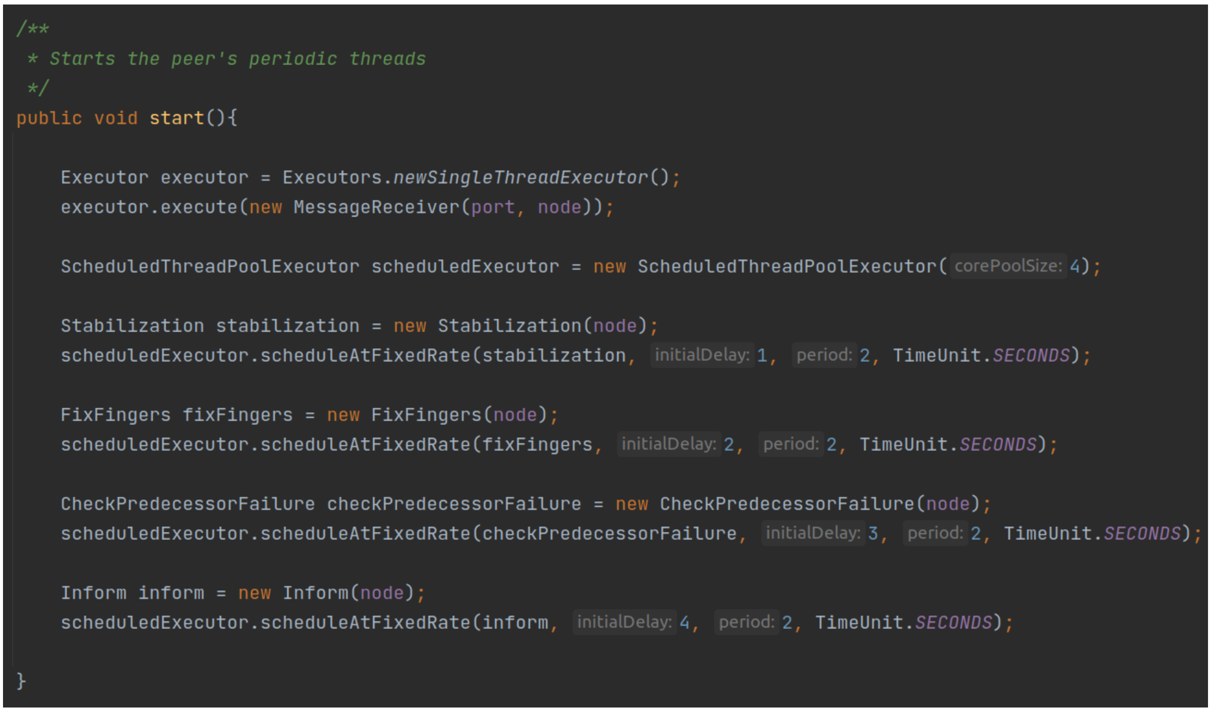
1. Node 3 checks if it needs to delete chunks in order to satisfy the new capacity and detects it has to delete one chunk;
2. Node 3 deletes the chunk and sends a DECREASE_REP_DEGREE message to node 2, the peer who triggered the backup for this chunk;
3. Node 2 receives the message and decreases the chunk's current replication degree, which drops below the desired value for it;
4. Node 2 starts a backup protocol for this specific chunk, which will then be stored in node 25;

This protocol is implemented in the file *Reclaim* inside package *subProtocols*.

3. Concurrency Design

Due to the high number of service requests that could appear simultaneously, the concurrency design was a critical part of the implementation of this project. In our case, concurrency is handled, mostly, using thread-pools.

Firstly, the chord protocol requires three operations to be run periodically, the *Stabilization* routine, the *Check Predecessor* routine and the *Fix Fingers* routine. We also implemented an extra routine, *Inform*, used for fault-tolerance features that will be explained later in this report. A peer also needs to be constantly listening for incoming connections. For this, we created a method called *start()*, responsible for starting all the threads required for these operations.

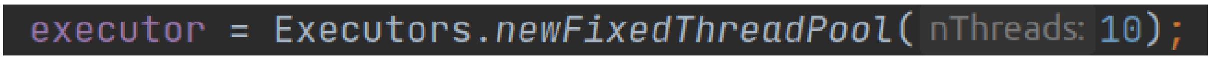


```
/**
 * Starts the peer's periodic threads
 */
public void start(){

    Executor executor = Executors.newSingleThreadExecutor();
    executor.execute(new MessageReceiver(port, node));

    ScheduledThreadPoolExecutor scheduledExecutor = new ScheduledThreadPoolExecutor( corePoolSize: 4);

    Stabilization stabilization = new Stabilization(node);
    scheduledExecutor.scheduleAtFixedRate(stabilization,  initialDelay: 1,  period: 2, TimeUnit.SECONDS);

    FixFingers fixFingers = new FixFingers(node);
    scheduledExecutor.scheduleAtFixedRate(fixFingers,  initialDelay: 2,  period: 2, TimeUnit.SECONDS);

    CheckPredecessorFailure checkPredecessorFailure = new CheckPredecessorFailure(node);
    scheduledExecutor.scheduleAtFixedRate(checkPredecessorFailure,  initialDelay: 3,  period: 2, TimeUnit.SECONDS);

    Inform inform = new Inform(node);
    scheduledExecutor.scheduleAtFixedRate(inform,  initialDelay: 4,  period: 2, TimeUnit.SECONDS);

}
```

Picture 2: start() method of the Peer class, (Peer.java file, peers package, lines 89-108)

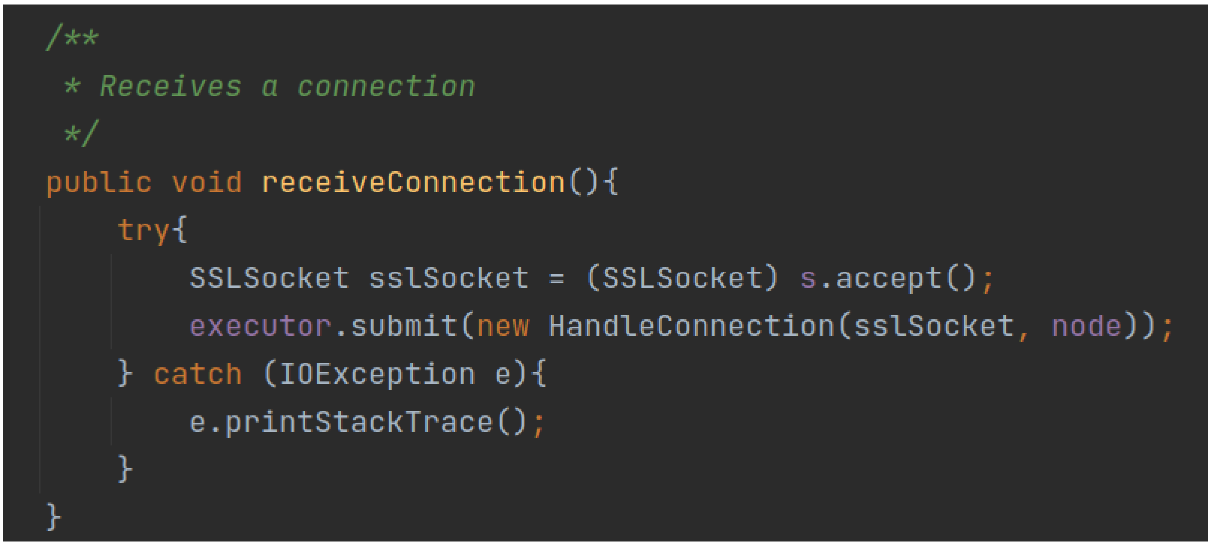
The picture shows the use of a *ScheduledThreadPoolExecutor* object with four threads, one for each operation described previously, that are run every two seconds. We also use a *SingleThreadExecutor* to run the thread that listens for incoming connections.

When a connection is received, it is important to assign it a new thread, so that the handling of this connection does not stop the job of listening for new incoming ones. To solve this problem, the *MessageReceiver* class, which is the one that listens for incoming connections, has a fixed size thread-pool that is utilised to execute the handling of connections. After a connection is received, the method *receiveConnection()* of the *MessageReceiver* is called, which submits to the thread-pool the task of handling the received connection (*HandleConnection* class), so that we can still listen for connections while handling the ones already received.



```
executor = Executors.newFixedThreadPool( nThreads: 10);
```

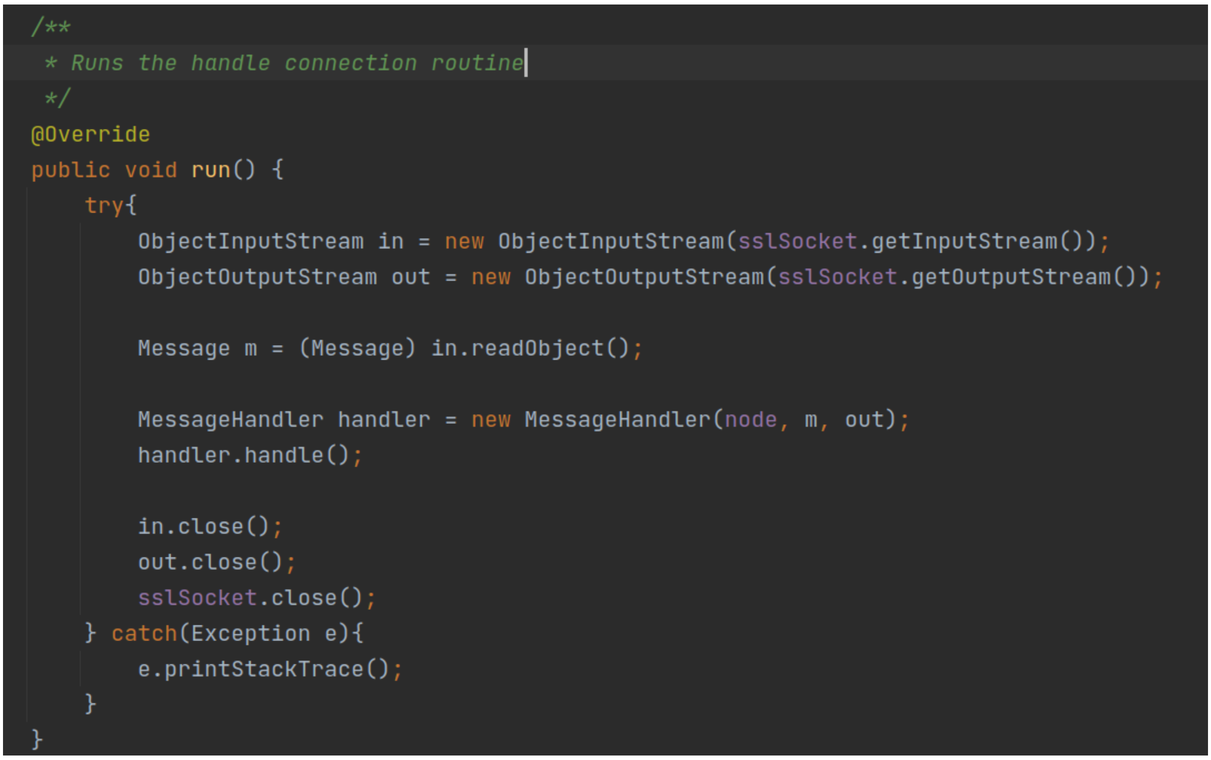
Picture 3: fixed size thread-pool of the MessageReceiver class (MessageReceiver.java file, messages package, line 37)



```java
/**
 * Receives a connection
 */
public void receiveConnection(){
    try{
        SSLSocket sslSocket = (SSLSocket) s.accept();
        executor.submit(new HandleConnection(sslSocket, node));
    } catch (IOException e){
        e.printStackTrace();
    }
}
```

Picture 4: method receiveConnection() of the MessageReceiver class (MessageReceiver.java file, messages package, lines 44-51)

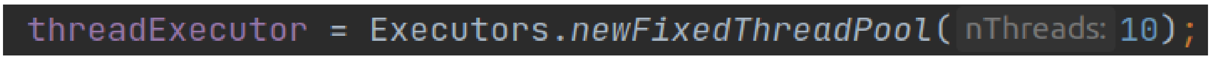
The handling of an incoming connection is then performed by the *HandleConnection* class, which reads the received message and creates a handler (*MessageHandler* class) for it.



```java
/**
 * Runs the handle connection routine
 */
@Override
public void run() {
    try{
        ObjectInputStream in = new ObjectInputStream(sslSocket.getInputStream());
        ObjectOutputStream out = new ObjectOutputStream(sslSocket.getOutputStream());

        Message m = (Message) in.readObject();

        MessageHandler handler = new MessageHandler(node, m, out);
        handler.handle();

        in.close();
        out.close();
        sslSocket.close();
    } catch(Exception e){
        e.printStackTrace();
    }
}
```

Picture 5: method run() of the HandleConnection class, (HandleConnection.java file, messages package, lines 32-50)

Furthermore, the *Node* class also has a thread-pool where the protocols tasks are executed. This allows for the concurrent execution of protocols from the same peer.



```
threadExecutor = Executors.newFixedThreadPool( nThreads: 10);
```

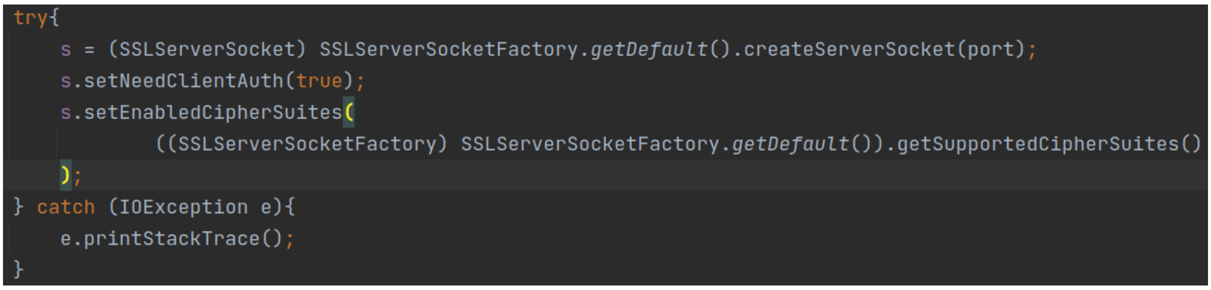
Picture 6: Node class's thread-pool, (Node.java file, chordProtocol package, line 41)

4. Java Secure Socket Extension

In this project, we use the Java Secure Socket Extension (JSSE) when communicating between peers, because it provides a higher level of security compared to regular TCP. It is used in all the protocols described in the Protocols section, due to the necessity of exchanging messages between peers in all of them.

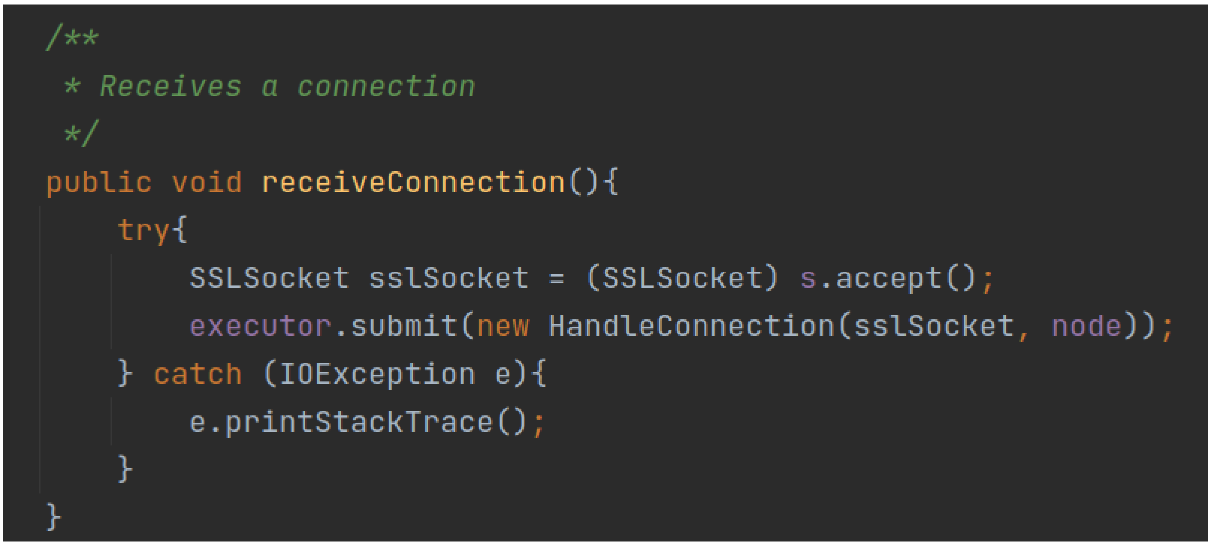
In this project, each peer acts as both a server and a client, in the sense that it can both receive incoming connections and establish new connections itself.

For the first case, receiving incoming connections, the previously described *MessageReceiver* class creates a *SSLServerSocket* at the beginning of the program's execution, responsible for detecting incoming connections and creating a new *SSLSocket* for each connection it receives.



```
try{
    s = (SSLServerSocket) SSLServerSocketFactory.getDefault().createServerSocket(port);
    s.setNeedClientAuth(true);
    s.setEnabledCipherSuites(
            ((SSLServerSocketFactory) SSLServerSocketFactory.getDefault()).getSupportedCipherSuites()
    );
} catch (IOException e){
    e.printStackTrace();
}
```

Picture 7: creation of the SSLServerSocket on the MessageReceiver class constructor (MessageReceiver.java file, messages package, lines 28-36)

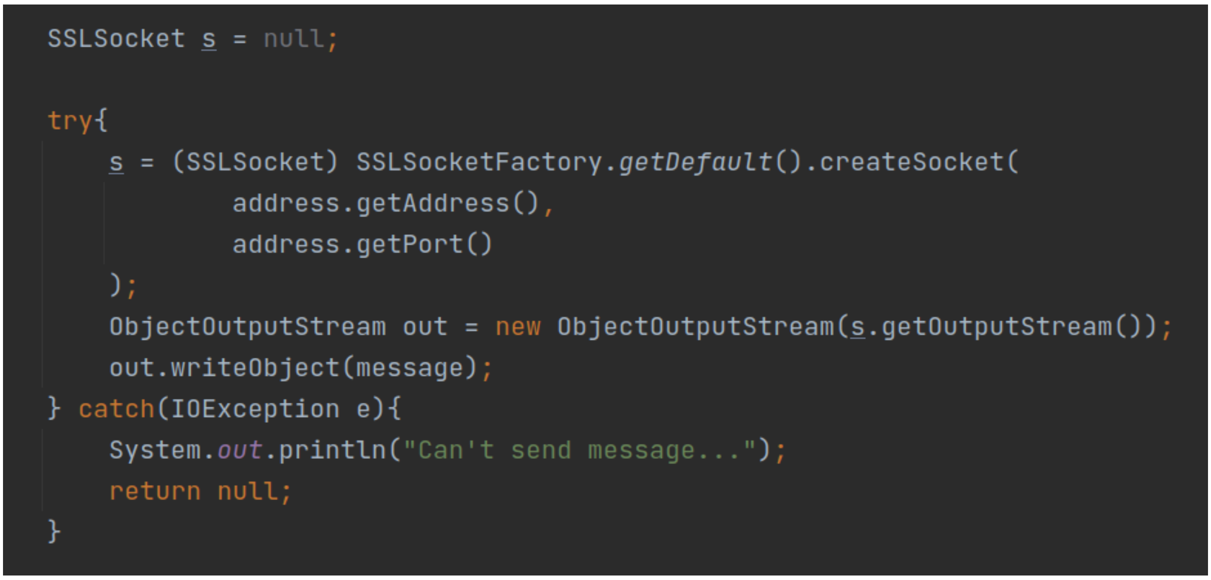


```java
/**
 * Receives a connection
 */
public void receiveConnection(){
    try{
        SSLSocket sslSocket = (SSLSocket) s.accept();
        executor.submit(new HandleConnection(sslSocket, node));
    } catch (IOException e){
        e.printStackTrace();
    }
}
```

Picture 8: creation of a SSLSocket for an incoming connection (MessageReceiver.java file, messages package, lines 44-51)

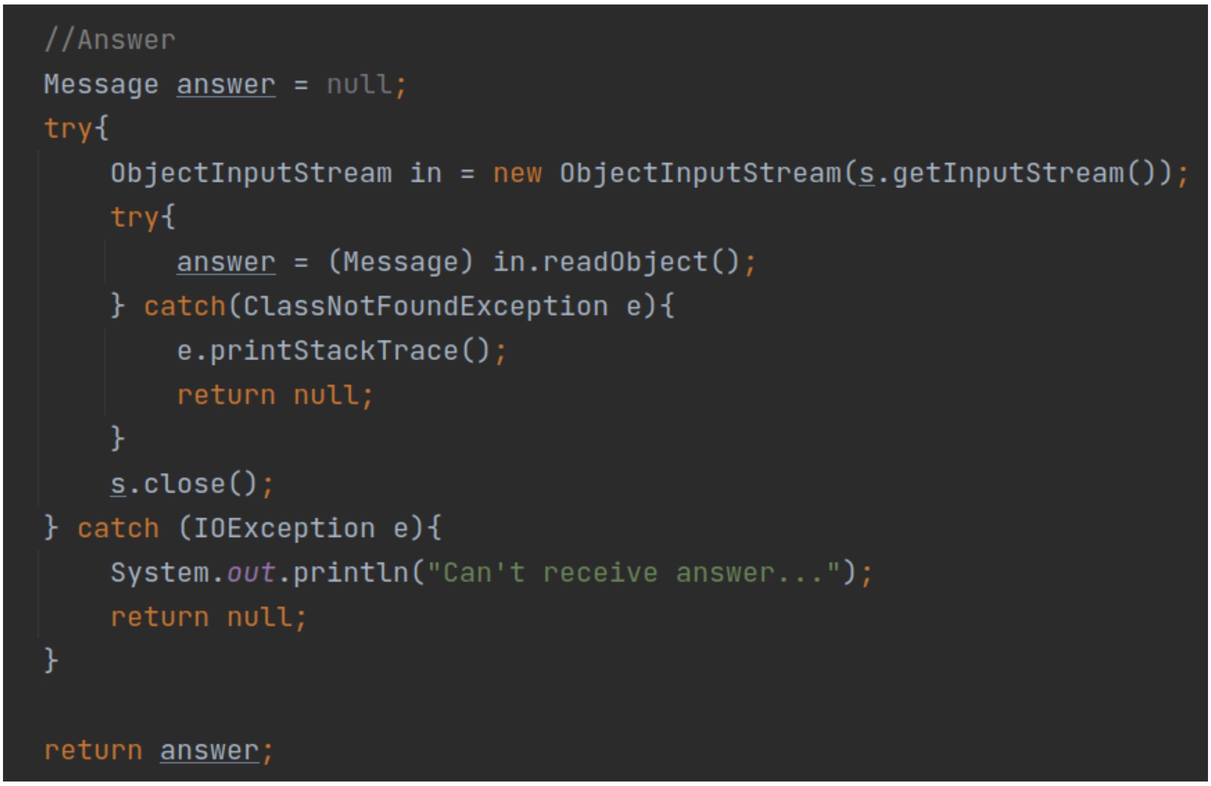
The *SSLServerSocket* requires client authentication and uses all the supported cypher-suites.

When sending messages, the implementation is different. For this purpose, the class *MessageSender*, responsible for sending messages, has a method *sendWithAnswer* that receives the message to be sent and the address where the message will be sent to. This method starts by creating a *SSLSocket* and establishes a connection with the received address. It then proceeds to write the message to the socket's *OutputStream* and then waits for a response, listening on the socket's *InputStream*. If the sending or reading of messages or even the connection establishment fail, the method prints an error message and aborts its execution.



```java
SSLSocket s = null;

try{
    s = (SSLSocket) SSLSocketFactory.getDefault().createSocket(
            address.getAddress(),
            address.getPort()
    );
    ObjectOutputStream out = new ObjectOutputStream(s.getOutputStream());
    out.writeObject(message);
} catch(IOException e){
    System.out.println("Can't send message...");
    return null;
}
```

Picture 9: creation of a SSLSocket and sending a message through it in the sendWithAnswer method (MessageSender.java file, messages package, lines 27-47)



```java
//Answer
Message answer = null;
try{
    ObjectInputStream in = new ObjectInputStream(s.getInputStream());
    try{
        answer = (Message) in.readObject();
    } catch(ClassNotFoundException e){
        e.printStackTrace();
        return null;
    }
    s.close();
} catch (IOException e){
    System.out.println("Can't receive answer...");
    return null;
}

return answer;
```

Picture 10: receiving a response through the InputStream of the socket on the sendWithAnswer method (MessageSender.java file, messages package, lines 49-65)

5. Scalability

To improve the scalability of the system, we implemented some provisions. At the design level, the Chord protocol was implemented and at the implementation level, we used thread-pools and Java NIO (for Asynchronous I/O) to help us make the system more scalable.

5.1 Scalability at the Design Level

As previously stated, we implemented the Chord protocol, which is, as described in its original paper, a “scalable peer-to-peer lookup service for Internet applications”.

5.1.1 The Chord Structure

The Chord protocol is based on consistent hashing, which has the purpose of assigning each data item a unique identifier through the hashing of its data, or part of it. In our implementation, each node is assigned an id in the network, which is generated by hashing the node’s address and port. When deciding the size of the network, we opted for having a maximum of 32 nodes, mainly for simplicity of the development of this project, but this value could be increased if we needed a bigger network without having to change the structure of our Chord implementation. With a maximum of 32 nodes in the network, each node has a finger table that contains 5 entries. Each node also knows its predecessor. This way, the network is treated like a ring, where each node knows which node comes behind him in the circle and 5 of the nodes that come after him in the circle. In order to ensure these values are correct, the chord protocol requires three routines to be run periodically. These routines, stabilize, check predecessor and fix fingers will be explained later in this section.

5.1.2 The Finger Table

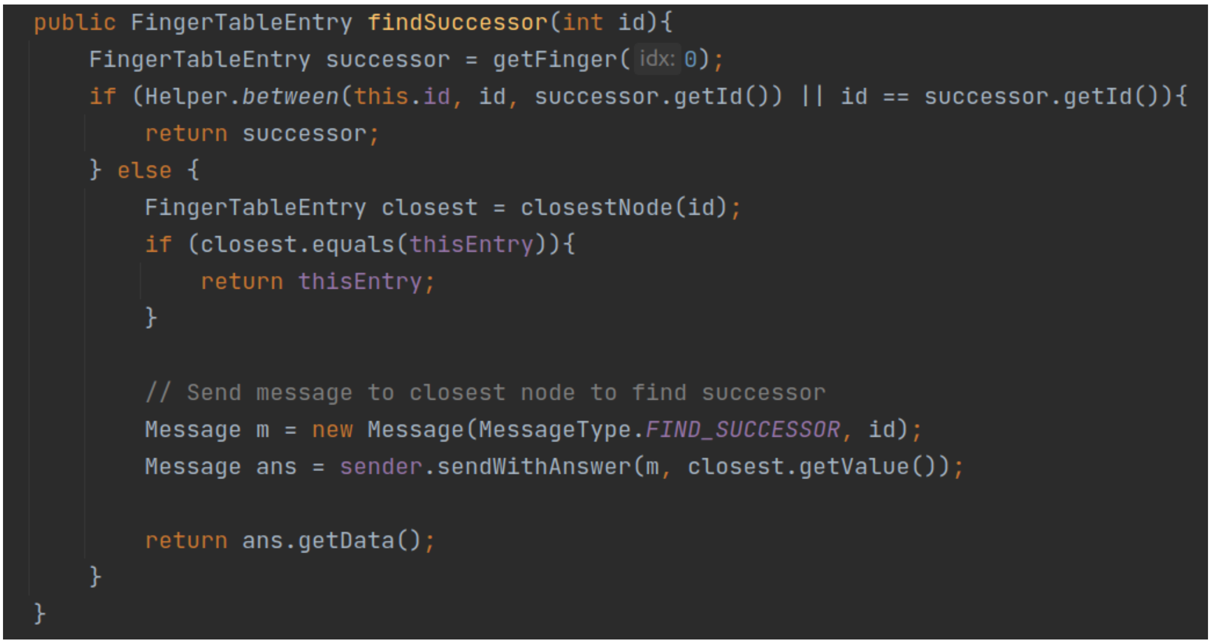
The purpose of storing a finger table on each node is to improve the search of a node on the ring, making this search run in logarithmic time, in relation to the number of nodes in the network. In chord, all arithmetic operations are performed in modulo 2^m, where 2^m is the number of nodes in the network. In our case, with a maximum of 32 nodes, m is 5. Each node n then stores m entries in its finger table, where each entry represents the successor of the node with id ($n + 2^i$), where i is the position of the entry in the finger table (in our case, 0 <= i <= 4). The purpose of the finger table is to, in some way, “jump” other nodes when performing a lookup. For example if a node with id 3 wants to search for node with id 20, instead of propagating the search to its direct successor, it might propagate it to node with id 19 ($3 + 2^4$), which would be the last entry in the node’s finger table.

5.1.3 Lookup

When a node *n1* wants to lookup the successor of another node *n2*, it first checks if *n2*’s id is between his own id and the id of its immediate successor. If it is, then *n2*’s successor will be *n1*’s current successor. If it is not, then it finds the entry of its finger table which is the closest preceding one to *n2* and propagates the search to that node. The following example illustrates this routine, when a chord network has 5 nodes, with ids 2, 7, 10, 20 and 31, and node 2 wants to lookup node 20:

1. Node 2 checks if 20 is between 2 and 7 (his successor) in the chord ring, which it is not;
2. Node 2 then proceeds to find the closest preceding entry to node 20 on his finger table, which is node 10 and sends a message to node 10 asking for node 20’s successor;
3. Node 10 checks if 20 is between 10 and 20 (his successor) in the chord ring, which it is, and returns its successor, which is node 20.

The following picture shows our implementation of this routine. For simplicity purposes, the error checking and handling were removed from the screenshot.



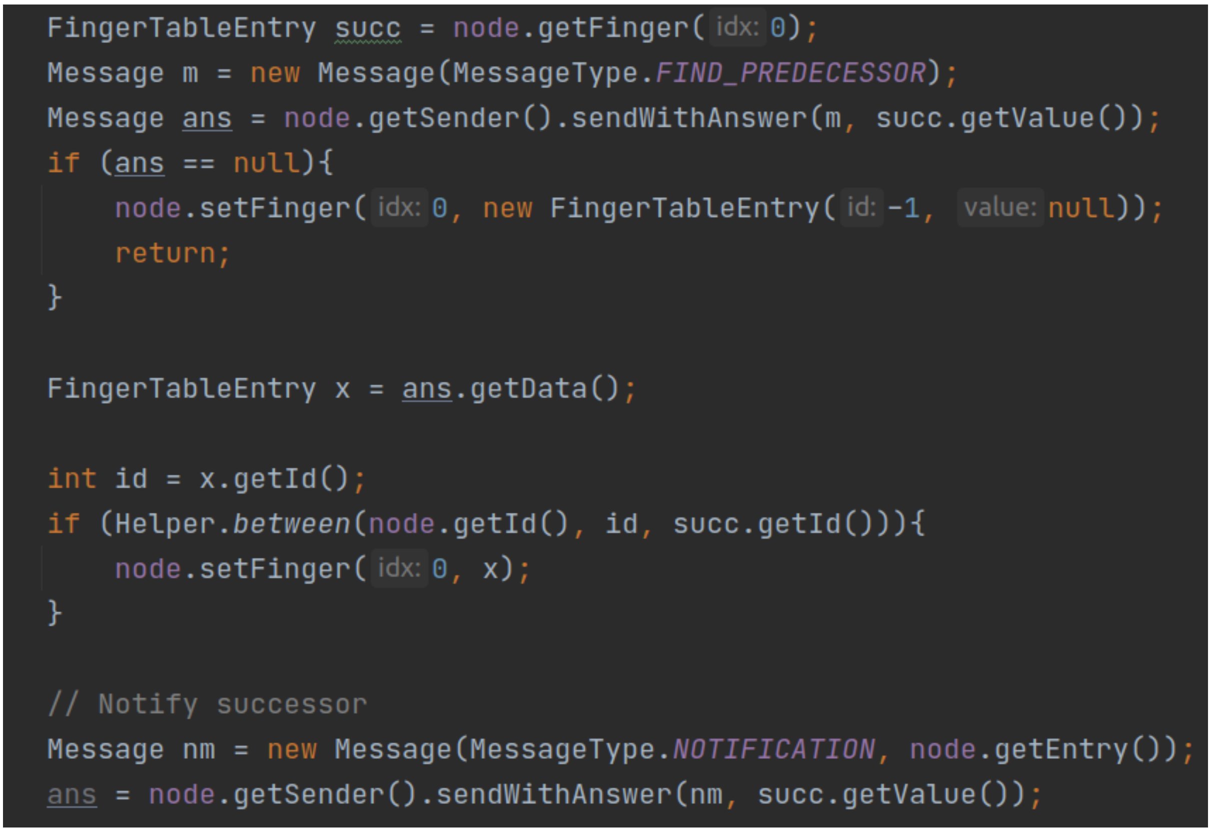
```
public FingerTableEntry findSuccessor(int id){
    FingerTableEntry successor = getFinger( idx: 0);
    if (Helper.between(this.id, id, successor.getId()) || id == successor.getId()){
        return successor;
    } else {
        FingerTableEntry closest = closestNode(id);
        if (closest.equals(thisEntry)){
            return thisEntry;
        }

        // Send message to closest node to find successor
        Message m = new Message(MessageType.FIND_SUCCESSOR, id);
        Message ans = sender.sendWithAnswer(m, closest.getValue());

        return ans.getData();
    }
}
```

Picture 11: findSuccessor method of class Node in the chordProtocol package (Node.java file, chordProtocol package, lines 215-243)

5.1.4 Stabilize Routine

This is one of the three routines that are run periodically to ensure that all the information is correct in every node of the network. This routine asks the node's current successor for its predecessor. If there has not been a new node joining the network, then the predecessor of the node's successor should be itself. If this is not the case, then we know that the current node's successor should be the node received as a response. Furthermore, this routine also notifies the node's current successor that the current node might be its predecessor. This is especially important when our successor leaves the network and, after finding our new successor, it still has not updated its own predecessor.

The following picture shows our implementation of this routine. Once again, for simplicity purposes, the error checking and handling code was removed from the screenshot.



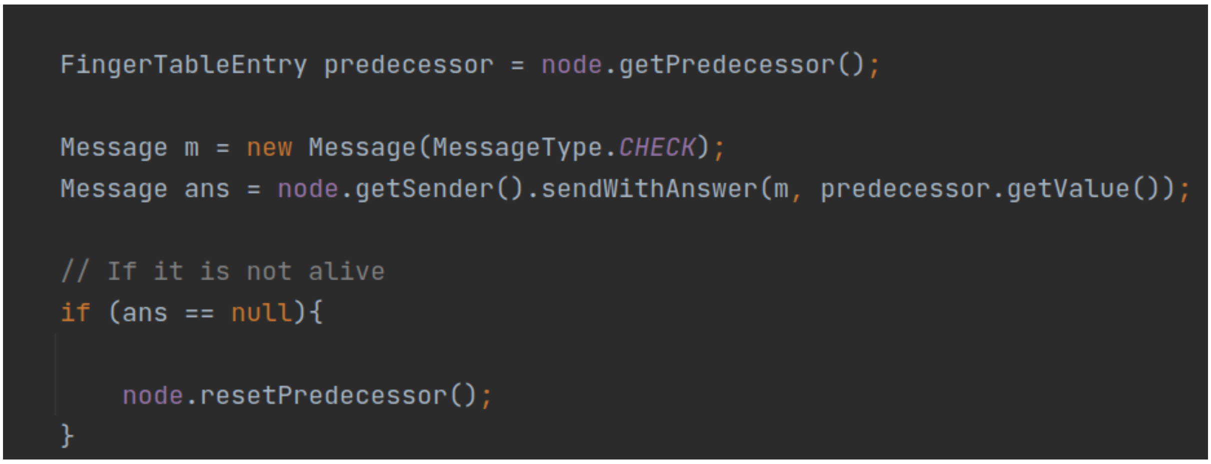
```
FingerTableEntry succ = node.getFinger( idx: 0);
Message m = new Message(MessageType.FIND_PREDECESSOR);
Message ans = node.getSender().sendWithAnswer(m, succ.getValue());
if (ans == null){
    node.setFinger( idx: 0, new FingerTableEntry( id: -1,  value: null));
    return;
}

FingerTableEntry x = ans.getData();

int id = x.getId();
if (Helper.between(node.getId(), id, succ.getId())){
    node.setFinger( idx: 0, x);
}

// Notify successor
Message nm = new Message(MessageType.NOTIFICATION, node.getEntry());
ans = node.getSender().sendWithAnswer(nm, succ.getValue());
```

Picture 12: run method of class Stabilization in the chordProtocol package (Stabilization.java file, chordProtocol package, lines 33-72)

5.1.5 Check Predecessor Routine

The *Check Predecessor* routine is responsible for checking if the current node’s predecessor is alive or not. The predecessor may not be alive due to the node crashing or leaving the network, in which case the current node must reset its predecessor and wait for a notification from another node that thinks it might be its predecessor.

The following picture shows our implementation of this routine.



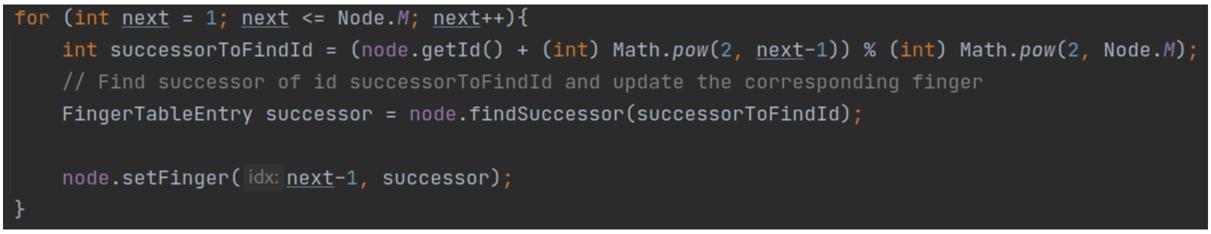
```
FingerTableEntry predecessor = node.getPredecessor();

Message m = new Message(MessageType.CHECK);
Message ans = node.getSender().sendWithAnswer(m, predecessor.getValue());

// If it is not alive
if (ans == null){

    node.resetPredecessor();
}
```

Picture 13: run method of the CheckPredecessorFailure class in the chordProtocol package (CheckPredecessorFailure.java file, chordProtocol package, lines 30-68)

5.1.6 Fix Fingers Routine

The *Fix Fingers* routine is responsible for ensuring the correctness of all finger table entries on the current node. In order to do so, it performs a lookup of the successor of each key in the finger table and updates it based on the result it receives. For example, for a node with id 2 and 5 entries on its finger table, this routine would perform a lookup for keys 3, 4, 6, 10 and 18. This routine is particularly important in the event of nodes joining the network, because it updates the current node's information regarding the newly joined nodes.

The following picture shows our implementation of this routine.



```
for (int next = 1; next <= Node.M; next++){
    int successorToFindId = (node.getId() + (int) Math.pow(2, next-1)) % (int) Math.pow(2, Node.M);
    // Find successor of id successorToFindId and update the corresponding finger
    FingerTableEntry successor = node.findSuccessor(successorToFindId);

    node.setFinger( idx: next-1, successor);
}
```

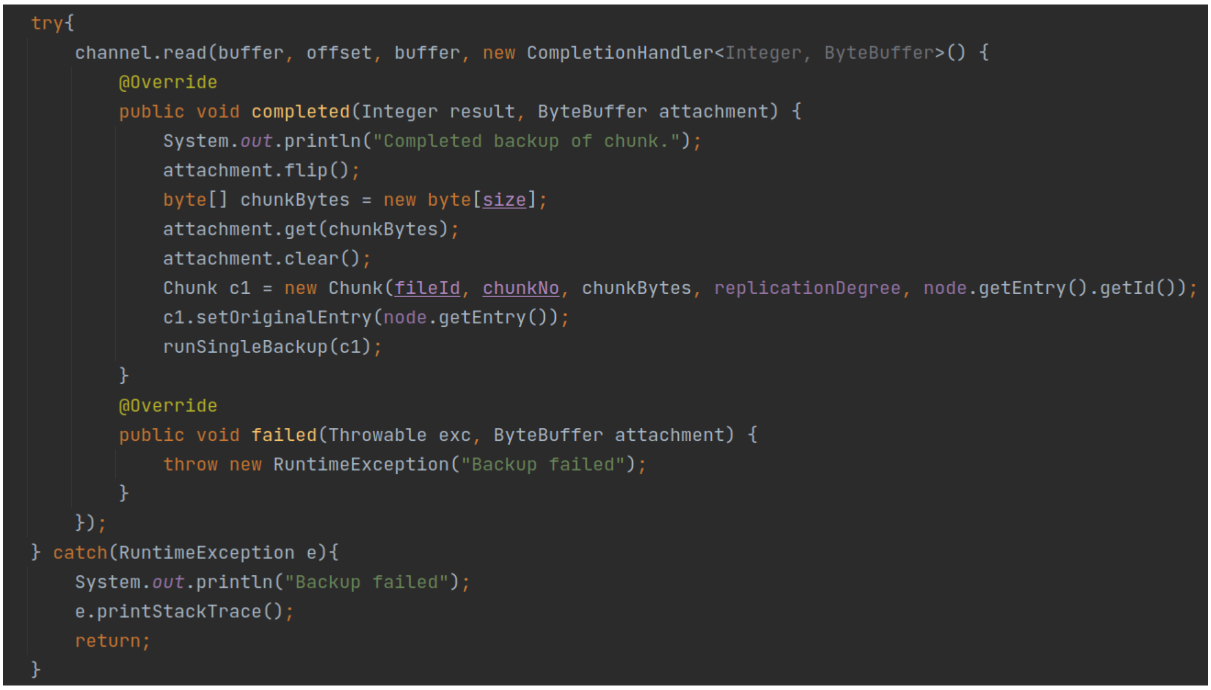
Picture 14: run method of the FixFingers class in the chordProtocol package (FixFingers.java file, chordProtocol package, lines 26-32)

5.2 Scalability at the Implementation Level

At the implementation level, our implementation relies on the use of thread-pools and Java NIO to improve the scalability of the developed system. The use of thread-pools in this project is thoroughly described in the Concurrency Design section, so this section will focus more on the use of Java NIO.

With Java NIO, we can perform asynchronous I/O operations. This means that we can continue executing our code while we are reading or writing to a file. In our implementation, Java NIO is used mostly in the backup and restore protocols. However, the reclaim protocol can also trigger a backup of the deleted chunks if their replication degree drops below the desired one, so Java NIO also plays an important role in this protocol.

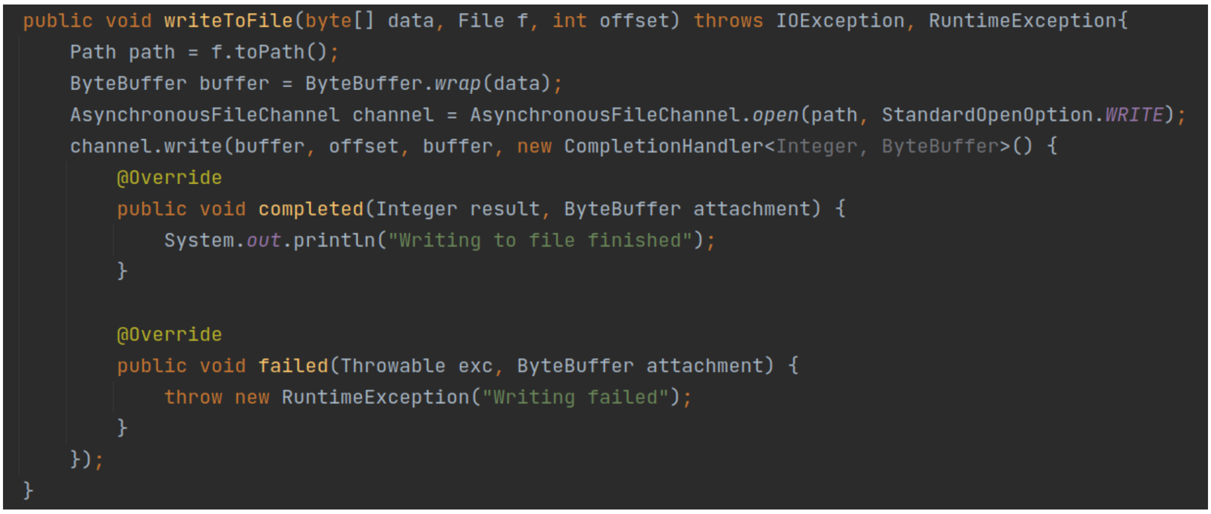
One use of Java NIO is when reading chunks from a file in order to perform their backup. We can keep reading chunks from a file without having to wait for the backup of the chunk to finish to move on to reading the next one and, when the reading operation completes its execution, we can call a handler function that will then perform the backup of the read chunk. The following picture demonstrates the procedure of reading a chunk from a file and performing its backup when the reading operation is finished.



```
try{
    channel.read(buffer, offset, buffer, new CompletionHandler<Integer, ByteBuffer>() {
        @Override
        public void completed(Integer result, ByteBuffer attachment) {
            System.out.println("Completed backup of chunk.");
            attachment.flip();
            byte[] chunkBytes = new byte[size];
            attachment.get(chunkBytes);
            attachment.clear();
            Chunk c1 = new Chunk(fileId, chunkNo, chunkBytes, replicationDegree, node.getEntry().getId());
            c1.setOriginalEntry(node.getEntry());
            runSingleBackup(c1);
        }
        @Override
        public void failed(Throwable exc, ByteBuffer attachment) {
            throw new RuntimeException("Backup failed");
        }
    });
} catch(RuntimeException e){
    System.out.println("Backup failed");
    e.printStackTrace();
    return;
}
```

Picture 15: reading a chunk from a file using Java NIO on the Backup class, backupFile method (Backup.java file, subProtocols package, lines 102-127)

This behaviour is also very useful when writing to a file. For example, during the restore protocol, when we receive a chunk, we can start writing it to the restored file and continue receiving chunks while the writing operation is still executing. The following picture shows our implementation of this operation.



```java
public void writeToFile(byte[] data, File f, int offset) throws IOException, RuntimeException{
    Path path = f.toPath();
    ByteBuffer buffer = ByteBuffer.wrap(data);
    AsynchronousFileChannel channel = AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);
    channel.write(buffer, offset, buffer, new CompletionHandler<Integer, ByteBuffer>() {
        @Override
        public void completed(Integer result, ByteBuffer attachment) {
            System.out.println("Writing to file finished");
        }

        @Override
        public void failed(Throwable exc, ByteBuffer attachment) {
            throw new RuntimeException("Writing failed");
        }
    });
}
```

Picture 16: writeToFile method of class ChunkFileSystemManager (ChunkFileSystemManager.java file, filesystem package, lines 467-482)

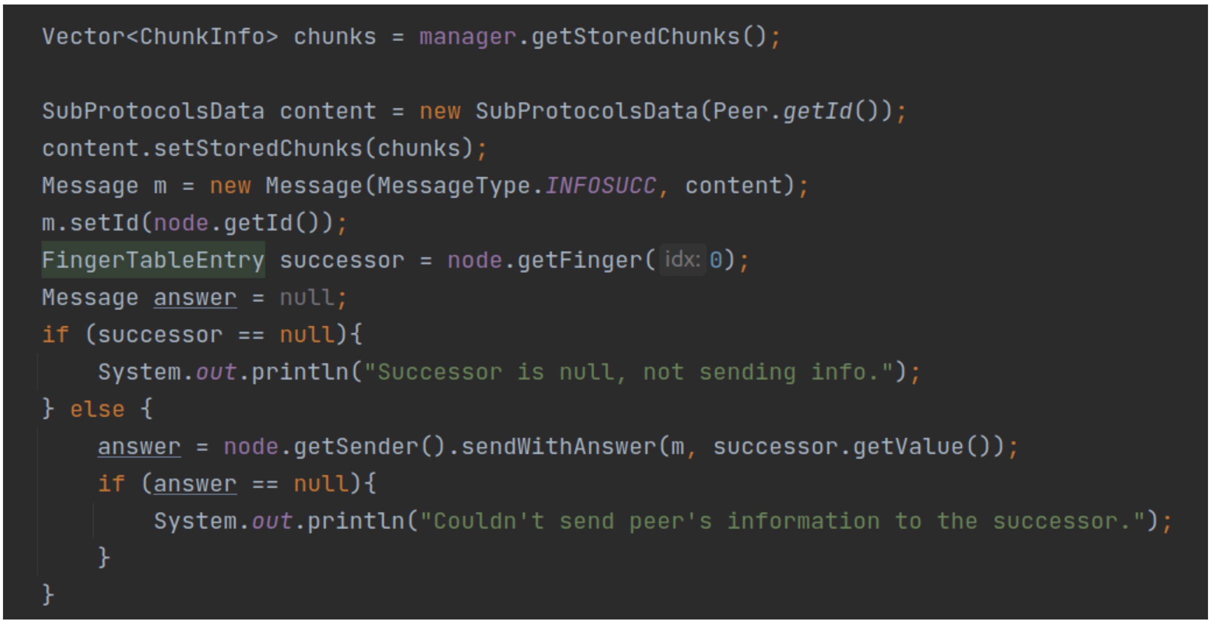
The use of Java NIO for I/O operations significantly decreases the overhead of their concurrent execution, making the system more scalable.

6. Fault-Tolerance

Fault-tolerance was another critical part of this project, as a distributed backup system needs to avoid single points of failure that would lead to the incorrect execution of some of its operations.

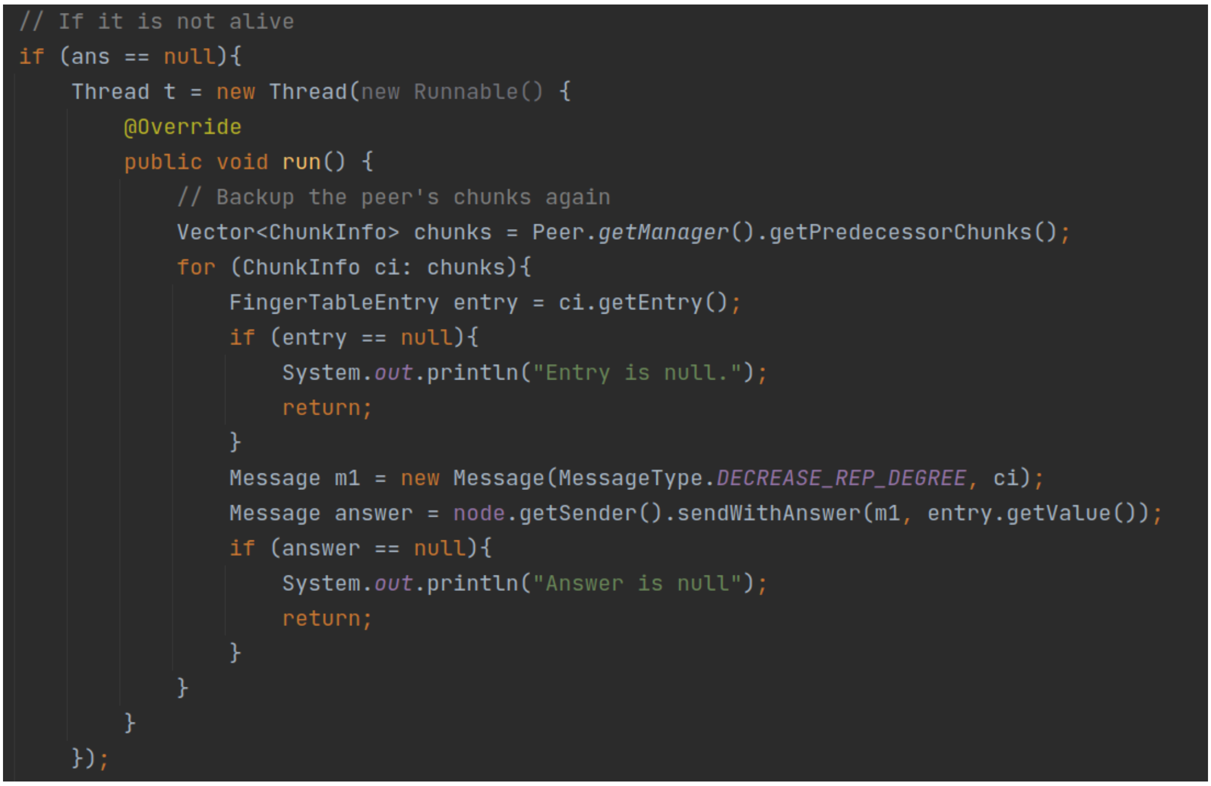
The first fault-tolerance measure implemented was the use of a replication degree when backing up a file. This replication degree determines how many peers should store each chunk of a certain file. By setting a high replication degree, we can ensure that the file is stored in multiple peers and, if one fails, the chunks are still available to be restored by the other peers that previously stored it.

There is also the problem of node failure. In chord, this is handled by the three periodic routines, that update the values on each node’s finger tables considering the failure of the node. However, this does not solve the fact that the chunks a node stores are lost when it fails. In order to prevent this from happening, we implemented a new periodic routine, called *Inform*, which is responsible for telling each node’s successor and predecessor which chunks the current node has stored. With this information, when a node fails, his successor can then proceed to warn the peer who originally backed up the lost chunks, by sending a DECREASE_REP_DEGREE message to it, so that it can decrease the chunk’s current replication degree and start a new backup for it if the new replication degree value drops below the desired one. The following pictures show the routine behaviour and the detection of a node’s failure by its successor.



```
Vector<ChunkInfo> chunks = manager.getStoredChunks();

SubProtocolsData content = new SubProtocolsData(Peer.getId());
content.setStoredChunks(chunks);
Message m = new Message(MessageType.INFOSUCC, content);
m.setId(node.getId());
FingerTableEntry successor = node.getFinger( idx: 0);
Message answer = null;
if (successor == null){
    System.out.println("Successor is null, not sending info.");
} else {
    answer = node.getSender().sendWithAnswer(m, successor.getValue());
    if (answer == null){
        System.out.println("Couldn't send peer's information to the successor.");
    }
}
```

Picture 17: sending information about the peer’s chunks to its successor on method run of class Inform (Inform.java file, subProtocols package, lines 37-52)



```
// If it is not alive
if (ans == null){
    Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            // Backup the peer's chunks again
            Vector<ChunkInfo> chunks = Peer.getManager().getPredecessorChunks();
            for (ChunkInfo ci: chunks){
                FingerTableEntry entry = ci.getEntry();
                if (entry == null){
                    System.out.println("Entry is null.");
                    return;
                }
                Message m1 = new Message(MessageType.DECREASE_REP_DEGREE, ci);
                Message answer = node.getSender().sendWithAnswer(m1, entry.getValue());
                if (answer == null){
                    System.out.println("Answer is null");
                    return;
                }
            }
        }
    });
```

Picture 18: sending a DECREASE_REP_DEGREE message to the chunk’s original peer when the predecessor fails, on method run of class CheckPredecessorFailure (CheckPredecessorFailure.java file, chordProtocol package, lines 38-61)

With this measure implemented, both a node and its successor would have to fail simultaneously for the chunks to be lost.

7. Conclusion

This project helped our group understand some important concepts about Distributed Systems and provided us some experience in implementing these types of systems.

Even though it was not an easy project to develop, it was definitely very rewarding in terms of the knowledge and experience gained by our group's members.