SDIS REPORT

Distributed Backup Service for the Internet

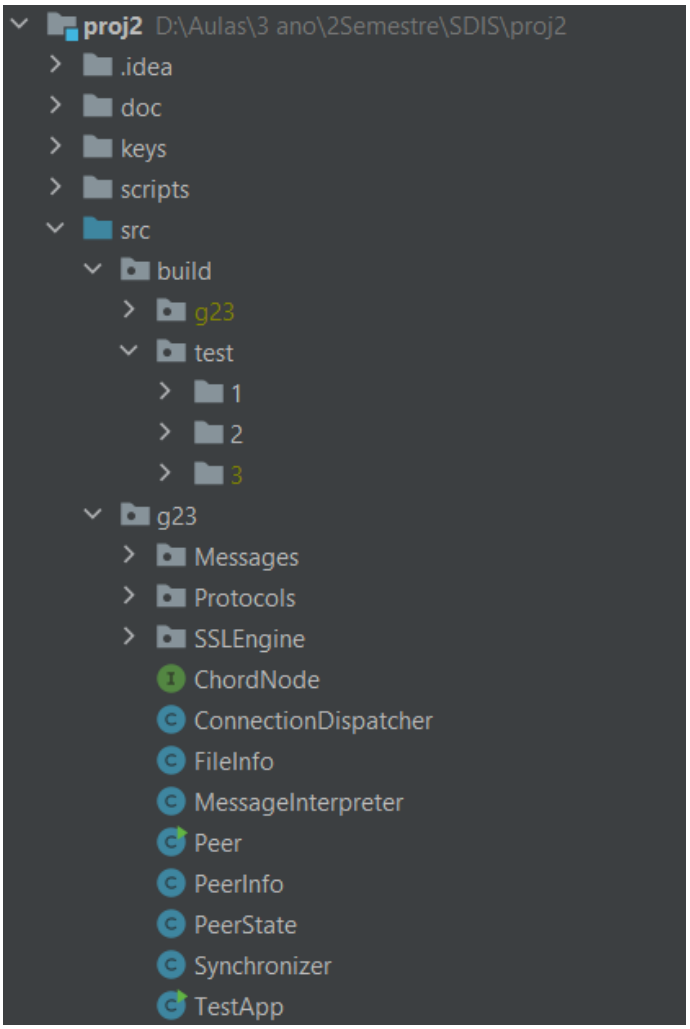
Ricardo Fontão - 201806317
Henrique Ribeiro - 201806529
Diogo Rosário - 201806582
Davide Castro - 201806512

Overview

The proposed project was to build a Distributed Backup Service for the Internet. We decided to keep some features from the service developed for the first project of the curricular unit. We kept the decentralized design but it now uses **Chord** to keep the network consistent. We also kept all the four protocols that were implemented in the first project: **Backup**, **Restore**, **Delete** and **Reclaim**.

Our program implements concurrency of protocols by using **Thread Pools**. It also makes use of the **SSLEngine** class to ensure secure communication of each protocol between peers. For scalability, as said before, we make use of the **Chord** protocol and also of **Java NIO**. Finally, we also implemented several measures of **Fault-tolerance**.

The file tree should be the same as follows:



Other instructions to run the program can be found in the readme.md

Protocols

In this project we implemented the same protocols that were implemented in the first project, which are: backup, restore, delete, reclaim and state. All messages are sent using TCP (SSLEngine), and the messages are sent as byte arrays as they are serializable, letting us take advantage of this in order to swap messages very easily. The messages contain various information such as the sender's ID, the replication degree, the file size as well as many other things.

```
public class Message implements Serializable {
      private MessageType type;
      private long senderId;
      private long fileId;
      private int replicationDegree;
      private int currentReplicationsDegree;
      private byte[] body;
      private String address;
      private int port;
      private long fileSize;
      private boolean isSeen = false;
}
```

Message.java: line 5

Backup

For the backup protocol, instead of storing chunks, like the first project, we decided to store complete files since we are using TCP which is more reliable.
The first peer that stores a file is the peer whose ID is closest to the hashed ID of the file we want to backup. In order to support fault tolerance here, when the replication degree of a file is higher than one, this peer is responsible for sending the message to its successors until the replication degree is met. When a peer receives a PUTFILE message it connects to the initiator peer sending a IWANT message asking the initiator peer to send the file.
As a stress test, we backed up a file with 4GB and it worked with no problems, finishing in around 1 minute.

```java
long fileSize = 0;
Path filePath;

if (path == null) { //peer has a backup of the file
    if (!this.peer.getStoredFiles().containsKey(hash)){
        return;
    }
    filePath = Path.of( first: "backup/" + hash);
    try {
        fileSize = Files.size(filePath);
    } catch (IOException e) {
        e.printStackTrace();
    }

}

if (hash == -1) //OWNER
    hash = Peer.getFileId(path, peer.getId());

String[] msgArgs = {
        String.valueOf(this.peer.getId()),
        String.valueOf(hash),
        String.valueOf(replicationDegree),
        String.valueOf(currentReplicationDegree),
        String.valueOf(fileSize),
        String.valueOf(this.peer.getAddress().getAddress().getHostAddress()),
        String.valueOf(this.peer.getAddress().getPort())
};

Message msgToSend = new Message(MessageType.PUTFILE, msgArgs, body: null);
(new BackupMessageSender(this.peer, msgToSend)).run();
```

Backup.java: line 57 (creates and sends the PUTFILE message to the first peer)

```java
//Backup to successor until replication degree is reached
for (int i = 0; i < this.peer.getSuccessors().size(); i++) {
    try {
        PeerInfo successor = this.peer.getSuccessors().get(i);

        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        System.out.println("Sending BACKUP (propagation) to " + successor.getAddress().getPort());
        oos.writeObject(this.message);
        oos.flush();
        byte[] msg = bos.toByteArray();
        SSLClient toSendMsg = new SSLClient(successor.getAddress());
        toSendMsg.write(msg, msg.length);
        bos.close();

        //Managed to propagate to a successor.
        break;
    } catch (IOException e) {
        if (i == 0)
            this.peer.getFingerTable().set(0, this.peer.getPeerInfo());
        if (i == this.peer.getSuccessors().size() - 1)
            System.out.println("Couldn't find an active successor, stopping BACKUP propagation");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Backup.java: line 105 (code that propagates the PUTFILE message)

```java
if (this.message.getCurrentReplicationDegree() > 0) {
    (new Backup(this.peer, this.message)).run();
}
```

ReceiveFile.java: line 114 (code that checks if the replication degree has been met, and if not, requests the propagation of the message)

```java
SSLClient fromServer = new SSLClient(new InetSocketAddress(message.getAddress(), message.getPort()));


String[] msgArgs = {
        String.valueOf(this.peer.getId()),
        String.valueOf(message.getFileId())
};
Message fileRequest = new Message(MessageType.IWANT, msgArgs,  body: null);


ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(fileRequest);
oos.flush();
byte[] msg = bos.toByteArray();
fromServer.write(msg, msg.length);
bos.close();
```

ReceiveFile.java: line 57 (code sends the IWANT message to the initiator peer)

Restore

On the restore protocol, since we know where the file is stored (the peer whose ID is closest to the hashed ID of the file) we ask him directly. When a peer receives this message he connects again to the initiator peer, sending him the file. In case the peer is unreachable or does not have the file, the peer propagates the message to its peer making him the one in charge of sending the requested file.

```
long fileId = file.getHash();
PeerInfo succID = this.peer.findSuccessor(fileId);

String[] msgArgs = {
        String.valueOf(this.peer.getId()),
        String.valueOf(fileId),
        String.valueOf(this.peer.getAddress().getAddress().getHostAddress()),
        String.valueOf(this.peer.getAddress().getPort())
};
Message msgToSend = new Message(MessageType.GETFILE, msgArgs,  body: null);
```

Restore.java: line 37 (code that prepares the GETFILE message)

```
try {
    Files.deleteIfExists(Path.of( first: "restore/" + fileToBeRestored.getPath()));
    WritableByteChannel toNewFile = Channels.newChannel(Files.newOutputStream(Path.of( first: "restore/" + fileToBeRestored.getPath())));

    byte[] buffer = new byte[50000];

    int bytesRead = 0;
    while (true) {
        try {
            bytesRead = sslServer.read(buffer);
        } catch (SSLFinishedReadingException e) {
            break;
        }
        ByteBuffer b_buffer;
        b_buffer = ByteBuffer.wrap(buffer,  offset: 0, bytesRead);
        toNewFile.write(b_buffer);
    }

    toNewFile.close();

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

ReceiveRestoreFile.java: line 33(code that receives the file to restore)

Delete

Similarly to the restore protocol, the delete protocol sends a message to the first peer that stored a file. If the file replication degree was more than one, this peer propagates the message to its successor until the number of peers deleting the file is equal to the replication degree.

```
try {

    synchronized (this) {
        peer.removeSpace(Files.size(Paths.get( first: "backup/" + key)));
    }


    System.out.println("Deleted FILE " + key);
    try {
        Boolean deleted = Files.deleteIfExists(Path.of( first: "backup/" + key));
    } catch (Exception e) {
        e.printStackTrace();
    }

    this.peer.getStoredFiles().remove(key);
    this.message.decrementCurrentReplication();
    System.out.println("CURRENT REPLICATION: " + this.message.getCurrentReplicationDegree());
```

DeleteFile.java: line 44 (code that deletes a specific file)

```
for(int i=0; i<this.peer.getSuccessors().size();i++) {
    try {
        PeerInfo successor = peer.getSuccessors().get(i);
        SSLClient sslClient = new SSLClient(successor.getAddress());

        System.out.println("Sending DELETE (propagation) to " + successor.getAddress().getPort());

        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(this.message);
        oos.flush();
        byte[] msg = bos.toByteArray();
        sslClient.write(msg, msg.length);
        bos.close();

        sslClient.shutdown();

        break;
    } catch (Exception e) {
        if(i == 0)
            this.peer.getFingerTable().set(0,this.peer.getPeerInfo());
        if(i == this.peer.getSuccessors().size() - 1)
        {
            System.out.println("Couldn't connect with any successor to send DELETE.");
            return;
        }
    }
}
```

Delete.java: line 98 (code that propagates the DELETE message)

Reclaim

Just like the first project, the reclaim limits the space a peer can use to backup files. When a peer has to reclaim an amount of space that makes it impossible to store a file, he needs to delete the file. In order to maintain the replication degree, before deleting the file, the peer sends a message to its successor that can be propagated. This message will stop when it reaches a peer with the file stored or the original peer. Once this happens, the peer that received the message will send a backup message instead of a deleted so that a new peer can request the file through a IWANT message.

```
if (this.peer.getFiles().containsKey(message.getFileId())) { //peer is owner of the file
    System.out.println("RECEIVED REMOVED " + message.getFileId() + ". I AM OWNER");

    String filePath = this.peer.getFiles().get(message.getFileId()).getPath();
    (new Backup(this.peer, filePath, message.getReplicationDegree(), message.getCurrentReplicationDegree())).run();

} else if (this.peer.getStoredFiles().containsKey(message.getFileId())) { //peer has a backup of this file

    System.out.println("RECEIVED REMOVED " + message.getFileId() + ". I HAVE A BACKUP");
    (new Backup(this.peer, message.getFileId(), message.getReplicationDegree(), message.getCurrentReplicationDegree())).run();

} else { // send message to successor

    System.out.println("RECEIVED REMOVED " + message.getFileId() + ". FORWARDING...");
    (new RemovedMessagePropSender(message, this.peer)).run();
}
```

ReceiveRemoved.java: line 29 (code that handles IDELETED messages)

```
for (int i = 0; i < this.peer.getSuccessors().size(); i++) {
    try {
        SSLClient fromServer = new SSLClient(this.peer.getSuccessors().get(i).getAddress());

        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(message);
        oos.flush();
        byte[] msg = bos.toByteArray();
        fromServer.write(msg, msg.length);
        bos.close();

        fromServer.shutdown();

        break;
    } catch(Exception e){
        if(i == 0)
            this.peer.getFingerTable().set(0,this.peer.getPeerInfo());
        if(i == this.peer.getSuccessors().size() - 1)
        {
            System.out.println("Couldn't connect with any successor to send REMOVED.");
            return;
        }
    }
}
```

RemovedMessagePropSender.java: line 26 (code that propagates IDELETED messages)

```
System.out.println(peer.getCurrentSpace());
System.out.println("RECLAIMING TO " + space);
peer.getStoredFiles().entrySet().stream().sorted(Map.Entry.comparingByValue())
        .takeWhile(m -> peer.getCurrentSpace() > space)
        .forEach(this::removeFile);
peer.setMaxSpace(space);
```

Reclaim.java: line 25 (code that checks if we need to delete any backed up file)

State

The state protocol gives the current state of the peer.

```
public PeerState state() throws RemoteException {
    return new PeerState(maxSpace, currentSpace, files, storedFiles, ongoing: null);
}
```

Peer.java: line 185 (code that returns the current peer state)

Concurrency Design

We implemented our application with support for multiple protocols running at the same time on the same peer as well as multiple instances of the same protocol at the same time ensuring consistency across all operations.

In order to avoid busy waiting a scheduled thread pool was used instead of sleeps in these functions making the CPU resources available while waiting to complete the task as we can schedule a thread to be run after a specific amount of time without wasting any resources in the meantime.

Firstly, we have two threads that are responsible for the protocols:

```
this.listenerThread = Executors.newSingleThreadExecutor();
this.protocolPool = Executors.newScheduledThreadPool( corePoolSize: 16);
```

Peer.java: line 35

The first thread is responsible for listening to all messages that reach the peer and call the correct function to deal with them, and the second one is responsible for executing the protocols.

Upon receiving a message, the first thread checks the type of the message and chooses an operation to run accordingly. To handle multiple messages to be processed and different protocols and operations to be

executed simultaneously, we used Java's ScheduledExecutorService interface to create thread pools in the peer. The operations are then dispatched as a Runnable to the pool that it's related to, using the execute() method. This implementation with multiple pools also avoids starvation, allowing the resources to be split more correctly to different tasks.

We also have another six thread, that are executed periodically, responsible for keeping the chord working as intended:

```
this.stabilizer = Executors.newSingleThreadScheduledExecutor();
this.successorFinder = Executors.newSingleThreadScheduledExecutor();
this.fingerFixer = Executors.newSingleThreadScheduledExecutor();
this.predecessorChecker = Executors.newSingleThreadScheduledExecutor();
this.checkIfPeersAreAlive = Executors.newSingleThreadScheduledExecutor();
this.stateSaver = Executors.newSingleThreadScheduledExecutor();
```

Peer.java: line 35

The first thread is responsible for keeping the chord stabilized, which means, making sure that the successor and predecessor of every peer is correct.

The second thread is used to fill out the list of successors we are using to improve our fault tolerance features.

The third makes sure all fingers of our finger table are correct.

The fourth thread checks if our predecessor is still alive, and if it isn't, our predecessor is set to null.

The fifth thread is used in a fault tolerance protocol. This thread tries to connect with the peers that have our files backed up through chord methods, and if we can't reach them we take appropriate actions.

Finally, the last thread is used to save our current state so that when we turn the peer back on we know our state when we were disconnected.

On top of that, when dealing with some delicate information such as the current used space, we used synchronized methods in order to try to make our code as resistant to race conditions as possible.

```
synchronized (this) {
    peer.removeSpace(Files.size(Paths.get( first: "backup/" + key)));
}
```

DeleteFile.java: line 47 (updates the current used space of the peer)

Lastly, the peer implements the RMI interface and as such, multiple protocols and be executed at the same time without interfering with each other.

To attest our application fulfills the requirements for concurrency we tested various protocols at the same time.

JSSE

Our project is based on JSSE features, besides RMI communication. To ensure secure communication between the nodes in the network we decided to use the **SLLEngine** interface supplied by Java in order to send messages using **TLS** and **SSL** security protocols. All application protocols in the project make use of JSSE.

The TLS protocol version used is 1.2 and the server requests client authentication using the keys and truststore files located in the /keys directory in the root of the project. Furthermore, we keep the default cipher-suites available in Java 11.

The key and truststore files are opened inside the **SSLEngineOrchestrator** and the password is passed when building the **SSLClient** and **SSLServer** classes. These 2 classes both extend the abstract class SSLEngineOrchestrator which is our main class for handling the SSLEngine related operations (handshake, wrap, unwrap) being SSLClient and SSLServer representations of client and server interfaces, as the SSLClient connects to a SSLServer's socket channel that is created on connection accept by the server node's ServerSocketChannel.

```
public SSLEngineOrchestrator(SocketChannel socketChannel, SSLContext sslContext, InetSocketAddress address, boolean isClient) {
    this.socketChannel = socketChannel;
    this.sslContext = sslContext;
    this.sslEngine = sslContext.createSSLEngine(address.getHostName(), address.getPort());
    this.sslEngine.setUseClientMode(isClient);
    this.sslEngine.setNeedClientAuth(true);

    try {
        if(isClient)
            socketChannel.connect(address);

        this.socketChannel.configureBlocking(true);
    } catch (IOException e) {
        e.printStackTrace();
    }

    SSLSession sslSession = this.sslEngine.getSession();
    int netBufSize = sslSession.getPacketBufferSize();
    int appBufSize = sslSession.getApplicationBufferSize();

    netInBuf = ByteBuffer.allocate(netBufSize);
    netOutBuf = ByteBuffer.allocate(netBufSize);
    appInBuf = ByteBuffer.allocate(appBufSize);
    appOutBuf = ByteBuffer.allocate(appBufSize);

    taskExecutor = Executors.newSingleThreadExecutor();
    doHandshake();
}
```

SSLEngineOrchestrator.java: line 31 (abstract class **SSLEngineOrchestrator** constructor)

```java
public static SSLContext createContext(boolean client, String pass) throws Exception {
    KeyStore keystore = KeyStore.getInstance("JKS");
    KeyStore truststore = KeyStore.getInstance("JKS");

    char[] passphrase = pass.toCharArray();

    keystore.load(new FileInputStream(client ? clientKeysFile : serverKeysFile), passphrase);
    truststore.load(new FileInputStream(truststoreFile), passphrase);

    KeyManagerFactory keyFactory = KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
    keyFactory.init(keystore, passphrase);

    TrustManagerFactory trustFactory = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
    trustFactory.init(truststore);

    SSLContext sslContext = SSLContext.getInstance("TLS");
    sslContext.init(keyFactory.getKeyManagers(), trustFactory.getTrustManagers(), random: null);
    return sslContext;
}
```

SSLEngineOrchestrator.java: line 218 (creates the SSL context with the correct keys and truststore files)

We use this interface across the protocols by creating an instance of SSLClient class when a peer needs to send a message to another peer, using their address. All peers have a ConnectionDispatcher thread running at all times that is always looking for connections, creating a SSLServer with the SocketChannel returned from serverSocketChannel.accept() and dispatching it to a MessageInterpreter runnable in another thread, so it can read and process the message.

```java
public ConnectionDispatcher(Peer peer) {
    this.peer = peer;
    try {
        this.sslContext = SSLEngineOrchestrator.createContext( client: false, pass: "123456");
        this.channel = ServerSocketChannel.open();
        this.channel.bind(this.peer.getAddress());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

In order to communicate, both SSLServer and SSLClient handshake, which then allows us to read and write messages back and forward using the methods in SSLEngineOrchestrator (read and write). We then close the connection by calling the shutdown method.

We implemented this interface in our project using SocketChannels with the intent of using Java NIO Selectors, but due to time constraints and problems with the interface's complexity using non-blocking channels, we were unable to implement selectors, so we ended up using SocketChannels without selectors and with blocking mode enabled.

Therefore, even though not all features we planned were implemented in time, we feel like JSSE features were widely explored, and we can guarantee security and confidentiality while two peers are communicating.

Scalability

At the design level we implemented the **Chord** protocol. Regarding the implementation of maintaining the network integrity, we used the paper referenced in the Lectures' slides. The exchange of messages between each peer for everything related to Chord is done via RMI. This implies that all peers must be connected to the same RMI server. Each one of them registers with their Chord ID. The first peer to be turned on, it sets its own successor to himself and every entry on its finger list to null. The finger table size is determined by the *m* which is the number of bits of the ID of each peer. In our case we are applying the MD5 hash function to the string: "<peer_ip_adress>:<peer_port>" and taking the LeastSignificantBits(64 bits) multiplied by -1(the LeastSignificantBytes of a UUID are always negative) and storing it in a variable of type long. The process for file IDs is similar but the hashed string contains the filename, last modified date of the file and the owner ID.

```java
public static long getFileId(String path, long peerID) {
    MessageDigest digest = null;
    try {
        digest = MessageDigest.getInstance("MD5");
        byte[] hash = digest.digest((path + Files.getLastModifiedTime(Path.of(path)) + peerID).getBytes());

        UUID id = UUID.nameUUIDFromBytes(hash);
        long idLong = id.getLeastSignificantBits() * -1;
        idLong = idLong % (long) Math.pow(2, Peer.m);

        return idLong;

    } catch (Exception e) {
        e.printStackTrace();
        return -1;
    }

}
```

Peer.java:line 497

```java
private static long calculateID(InetSocketAddress address) {

    MessageDigest digest = null;
    try {
        digest = MessageDigest.getInstance("MD5");
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }

    byte[] hash = digest.digest(address.toString().getBytes());
    UUID id = UUID.nameUUIDFromBytes(hash);
    long idInt = id.getLeastSignificantBits() * -1; // UUID LSB always returns negative numbers
    idInt = Math.round(idInt % Math.pow(2, Peer.m));

    System.out.println("My id is: " + idInt);
    return idInt;
}
```

Peer.java:line 209

Therefore we are working with an m = 64. This means we can have up to 2 ^ 64 peers in the network making this a very scalable solution.

When a new peer is introduced it tries to join the network by contacting a peer already in it and asking which is the successor for its own id so the correct successor value can be set properly. For example, if peer 1, 5 and 10 are in the network and peer 3 wants to join, its successor must be set to 5. If this process ends without problems the node is in the network and the stabilization functions, which will be discussed later, will stabilize it over time.

```java
public boolean join(PeerInfo peer) {
    predecessor = null;

    Registry registry;
    ChordNode stub;
    try {
        registry = LocateRegistry.getRegistry();
        stub = (ChordNode) registry.lookup(String.valueOf(peer.getId()));
        PeerInfo response = stub.findSuccessor(this.getId());

        fingerTable.set(0, response);

    } catch (RemoteException | NotBoundException e) {
        System.err.println("FAILED TO JOIN CHORD. EXITING GRACEFULLY...");
        return false;
    }

    return true;
}
```

Peer.java:line 346

To make a lookup of a successor of a specific ID we use the **findSuccessor** method.

```java
public PeerInfo findSuccessor(long id) throws RemoteException {

    if (chordIdInBetween(id, this.info, this.fingerTable.get(0))) {
        return fingerTable.get(0);
    } else {
        PeerInfo closestPeer = this.closestPrecedingPeer(id);
        if (closestPeer.getId() == this.info.getId()) {
            return this.info;
        }

        try {
            Registry registry = LocateRegistry.getRegistry();
            ChordNode stub = (ChordNode) registry.lookup(String.valueOf(closestPeer.getId()));

            return stub.findSuccessor(id);
        } catch (Exception e) {
            for (int i = 0; i < this.fingerTable.size(); i++) {
                PeerInfo pi = this.fingerTable.get(i);
                if (pi.getId() == closestPeer.getId())
                    this.fingerTable.set(i, null);

            }
        }
    }
    return null;
}
```

Java.peer:line 226

First the method checks if the requested ID is between “us” and our successor. If it is then the successor info will be returned, else we continue. Then we check our finger table from the end to the beginning (closestPrecedingPeer method) and if we find a peer that is closer to the requested ID but still “smaller”, we call the findSuccessor method in that peer and we return that value. This means that if needed we can traverse the entire network to find the correct successor of a specific ID. This method is very useful when we want to know which peer needs to receive our Backup messages or when a peer wants to join the network.

```
private PeerInfo closestPrecedingPeer(long id) {
    for (int i = fingerTable.size() - 1; i >= 0; i--) {
        if (fingerTable.get(i) != null && chordIdInBetween(fingerTable.get(i).getId(), this.info.getId(), id)) {
            return fingerTable.get(i);
        }
    }
    return this.info;
}
```

Peer.java:line 253

To keep the network stable we have threads that periodically run the stabilization functions. The first one is the **stabilize** method. In this method we ask our successor what their predecessor is. If that predecessor is between us and our successor then it is our new successor, else if our successor is ourselves then that predecessor is our new successor. In the end we notify that peer (sendNofitication and notify methods) that we think he is our successor and it will set the predecessor reference accordingly. The other stabilization method is the fix_fingers method. Each call it fixes the next finger by getting the successor of id -> id + 2 ** (i - 1) % 2^m where i is the index of the finger being fixed.

```java
public void stabilize() {
    PeerInfo ni = null;
    for (int i = 0; i < this.successors.size(); i++) {
        try {
            Registry registry = LocateRegistry.getRegistry();
            ChordNode stub = (ChordNode) registry.lookup(String.valueOf(successors.get(i).getId()));
            ni = stub.getPredecessor();
            break;
        } catch (Exception e) {
            if (i == 0)
                this.fingerTable.set(0, this.info);
            if (i == this.successors.size() - 1) {
                System.out.println("Couldn't find any active successor, exiting stabilization.");
                return;
            }
            System.out.println("Couldn't find successor to stabilize, trying the next on the successor list.");
        }
    }


    if (ni != null && chordIdInBetween(ni.getId(), this.info, this.fingerTable.get(0))) {
        fingerTable.set(0, ni);
    }

    if (ni != null && (this.fingerTable.get(0).getId() == this.getId()))
        fingerTable.set(0, ni);

    this.sendNotification(this.fingerTable.get(0));
}
```

Peer.java:line 283

```java
private void fix_fingers() {

    next += 1;
    if (next > m) {
        next = 1;
    }
    try {
        fingerTable.set(next - 1, findSuccessor( id: (this.getId() + (long) Math.pow(2, next - 1)) % (long) Math.pow(2, m)));
    } catch (RemoteException e) {
        this.fingerTable.set(next - 1, null);
    }

}
```

Peer.java:line 389

```java
private void sendNotification(PeerInfo node) {
    if (node.getId() == this.getId())
        return;
    Registry registry;
    ChordNode stub;
    try {
        registry = LocateRegistry.getRegistry();
        stub = (ChordNode) registry.lookup(String.valueOf(node.getId()));
        boolean response = stub.notify(this.info);

    } catch (RemoteException | NotBoundException e) {
    }

}

public boolean notify(PeerInfo node) throws RemoteException {
    if (this.predecessor == null || chordIdInBetween(node.getId(), this.predecessor, this.info)) {
        this.predecessor = node;
        return true;
    }
    return false;
}
```

Peer.java:line 366

Finally the **check_predecessor** method periodically checks if our predecessor is still alive keeping the predecessor field updated.

```java
public boolean check_predecessor() {
    if (this.predecessor != null) {
        Registry registry;
        ChordNode stub;

        try {
            registry = LocateRegistry.getRegistry();
            stub = (ChordNode) registry.lookup(String.valueOf(this.predecessor.getId()));
            return stub.isAlive();

        } catch (RemoteException | NotBoundException e) {
            this.predecessor = null;
            return false;
        }
    }
    return false;
}
```

Peer.java:line 407

Fault-tolerance

In order to guarantee that our program can work even in some unexpected situations (such as, a peer stops working or crashes) we took some measures to ensure nothing goes wrong.

One feature we added was a list that not only contains our direct successor, but also the successor of the successor and its successor.

```
private ArrayList<PeerInfo> successors;
```

Peer.java: line 47

This way, when our successor fails we know which peer is most likely to become our successor and we can send the messages to this peer instead, letting us continue with the execution of the program.

```
for (int i = 0; i < this.peer.getSuccessors().size(); i++) {
    try {
        SSLClient fromServer = new SSLClient(this.peer.getSuccessors().get(i).getAddress());

        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(message);
        oos.flush();
        byte[] msg = bos.toByteArray();
        fromServer.write(msg, msg.length);
        bos.close();

        fromServer.shutdown();

        break;
    } catch(Exception e){
        if(i == 0)
            this.peer.getFingerTable().set(0,this.peer.getPeerInfo());
        if(i == this.peer.getSuccessors().size() - 1)
        {
            System.out.println("Couldn't connect with any successor to send REMOVED.");
            return;
        }
    }
}
```

RemovedMessagePropSender.java: line 25 (example of iterateration the successors list in case the successor fails)

On top of that, we also keep a record of who has each of the initiator peer's files backed up. This way we can try to reach them to make sure they are alive and if they are not we can send a new backup message in order to maintain the desired replication degree. This record is updated when the peer receives a IWANT message or a IDELETED message.

```
this.peer.getFilesStoredInPeers().get(this.message.getFileId()).add(this.message.getSenderId());
```

SendFile.java: line 59 (adds new peer to the record of who has the file backed up)

```
this.peer.getFilesStoredInPeers().get(msg.getFileId()).remove(msg.getSenderId());
```

MessageInterpreter.java: line 70 (removes peer from the record of who has the file backed up)

```
public void peersAreAlive() {
    ArrayList<Map.Entry<Long, Long>> stuffToDelete = new ArrayList<>();
    try {
        for (Map.Entry<Long, HashSet<Long>> entry : this.filesStoredinPeers.entrySet()) {
            for (Long l : entry.getValue()) {

                if (this.findSuccessor(l).getId() != l) {
                    stuffToDelete.add(new AbstractMap.SimpleEntry<>(entry.getKey(), l));

                    this.protocolPool.schedule(new Backup( peer: this,
                                    this.files.get(entry.getKey()).getPath(), replicationDegree: 1, currentReplicationDegree: 1),
                            delay: 5, TimeUnit.SECONDS);
                }
            }
        }
    } catch (Exception e) {
        System.out.println("Peer died.");
    }

    for (Map.Entry<Long, Long> toDelete : stuffToDelete) {
        this.filesStoredinPeers.get(toDelete.getKey()).remove(toDelete.getValue());
    }
}
```

Peer.java: line 126 (periodic function that checks if all peers that have backed up files are alive and sends a backup if not)

This function does not send the backup request immediately, but waits 5 seconds before sending it. This schedule lets the peers fix their finger table before sending the message in order to guarantee that the file is stored in the correct place.

References

- Esmond Pitt. Fundamental networking in Java. Springer Science & Business Media, 2005.
- Java Secure Socket Extension (JSSE) Reference Guide

https://docs.oracle.com/en/java/javase/16/security/java-secure-socket-extension-jsse-reference-guide.html

- IBM, Non-blocking I/O with SSLEngine

https://www.ibm.com/docs/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/ssleng.html

- Java 11 SSLEngine documentation

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/javax/net/ssl/SSLEngine.html

- Chord peer-to-peer wikipedia page

https://en.wikipedia.org/wiki/Chord_(peer-to-peer)

- Stoica et al., "Chord: A scalable peer-to-peer lookup protocol for Internet applications", IEEE/ACM Transactions on Networks, (11)1:17-32, Feb 2003

https://dl.acm.org/doi/pdf/10.1109/TNET.2002.808407?download=true