Project 2 - Distributed Backup Service for the Internet

Distributed Systems

Report

T3G23

Carlos Eduardo da Nova Duarte (up201708804) *up201708804@fe.up.pt*

Paulo Roberto Dias Mourato (up201705616) *up201705616@fe.up.pt*

Simão Pereira de Oliveira (up201603173) *up201603173@fe.up.pt*

Tiago Candeias Verdade (up201704003) *up201704003@fe.up.pt*

29/05/2020

Overview

For this project, we have implemented a distributed backup service for the internet that supports **backup, restore, delete, reclaim**, and **state** operations.

Our implementation was designed to be totally distributed by using **Chord** to locate the file's replicas. **Thread-pools** are being used to achieve concurrency, as well as **Java NIO** for asynchronous I/O.

To ensure secure communication between peers and clients, **JSSE** features were used, mainly the **SSLEngine** interface.

The usage of thread-pools and Java NIO together with the Chord protocol helped us achieve **much higher scalability**. More information is available in the *Scalability* section.

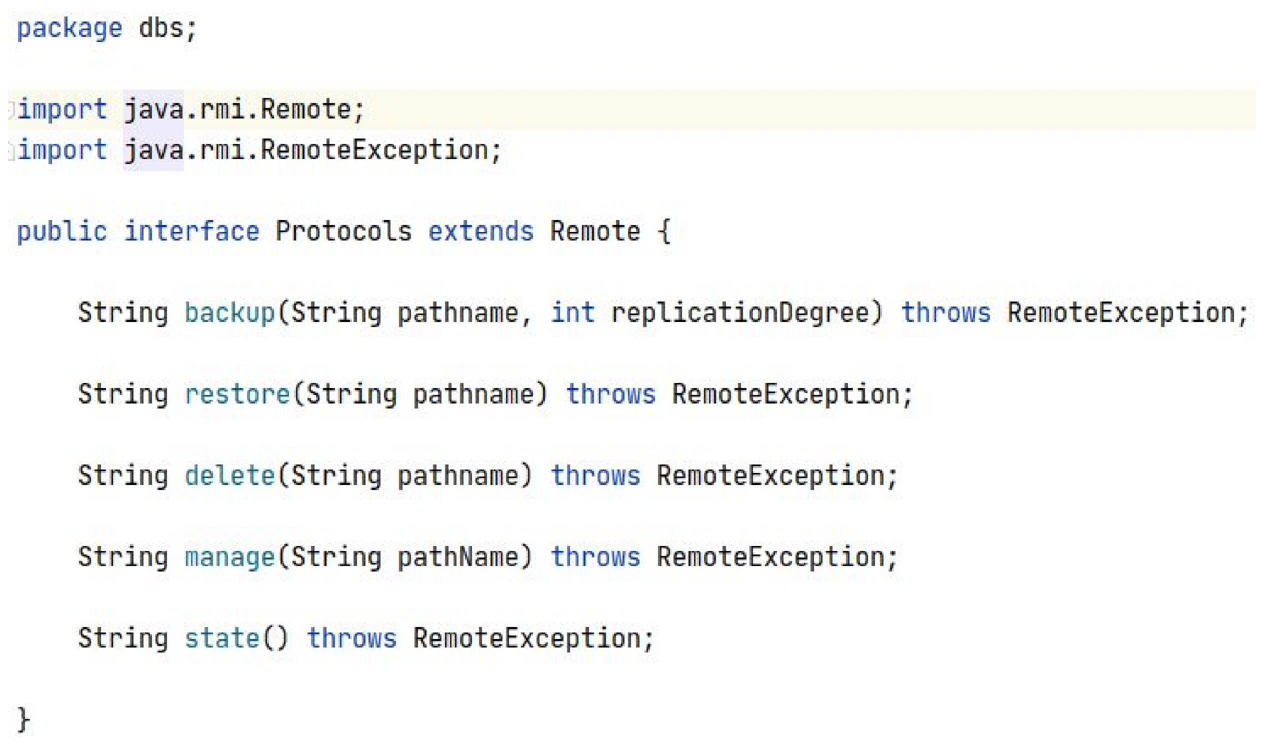
Also, because we've implemented a Chord protocol overlay, we were able to improve the system's **fault-tolerant capabilities**. We have also applied some tweaks of our own that we will describe in the *Fault-tolerance* section.

Protocols

As already stated in the *Overview* section, we have implemented backup, restore, delete, manage, and state protocols. In the following sections we will describe the implementation of these protocols. Because we've used RMI and TCP with JSSE, we will also detail the format of each protocol's messages along the sections.

RMI interface

RMI was used so that the TestApp could interact with other peers. The interface has the following methods:



```java
package dbs;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Protocols extends Remote {

    String backup(String pathname, int replicationDegree) throws RemoteException;

    String restore(String pathname) throws RemoteException;

    String delete(String pathname) throws RemoteException;

    String manage(String pathName) throws RemoteException;

    String state() throws RemoteException;

}
```

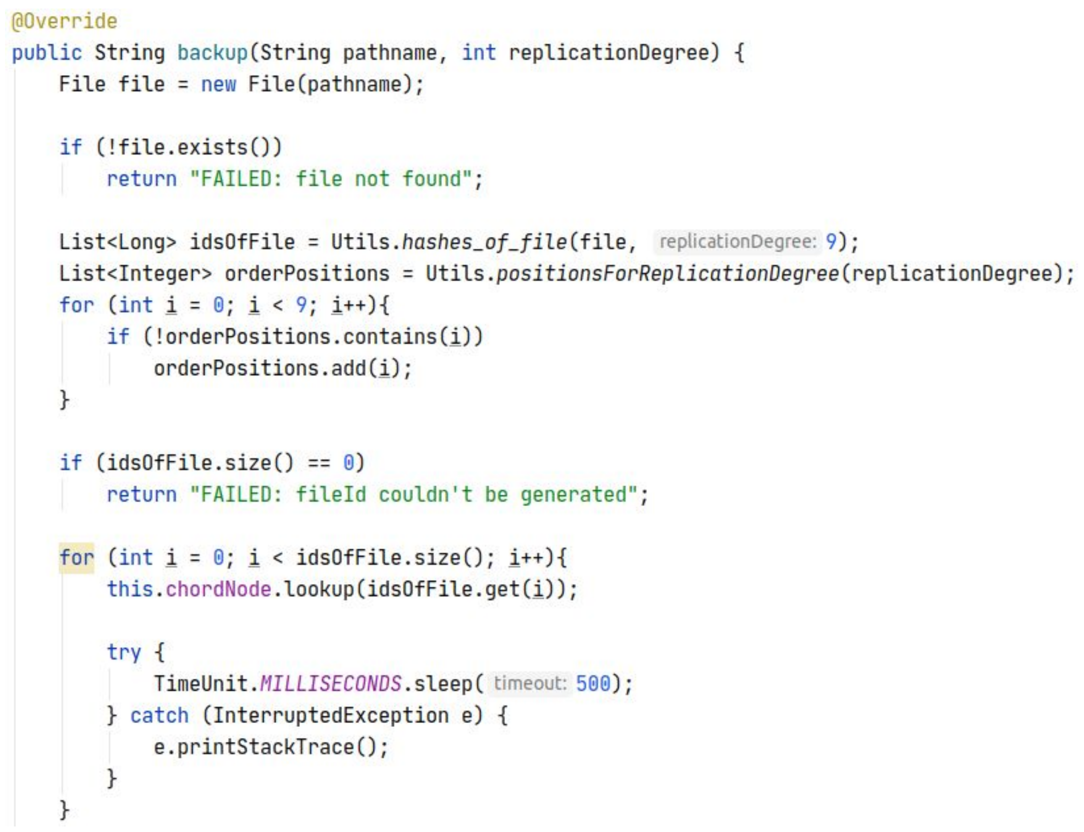
Every method received the name of the file as a parameter (**pathName**). The backup method also needs the replication degree (**replicationDegree**) that indicates the number of replicas to be created and distributed in the system.

Backup

We've decided to only backup whole files instead of splitting them into chunks across the peer network.

NOTE: We have considered the maximum replication degree value to be 9.

The first thing that's done is the generation of every possible ID for a file. Multiple IDs are allowed for the same file because we need to support backing up files to several peers in order to make the system more fault-tolerant. The usage of replication degrees aids this process. Then, for every valid ID that can be assigned to a file's replica, we lookup in the Chord ring who is its successor. While doing this for all 9 possible IDs might look like a waste of time and resources at first, this is crucial to ensure that each replica will stay at a different peer (otherwise there is the possibility of a peer holding multiple files in case the number of peers in the network is very small).



```
@Override
public String backup(String pathname, int replicationDegree) {
    File file = new File(pathname);

    if (!file.exists())
        return "FAILED: file not found";

    List<Long> idsOfFile = Utils.hashes_of_file(file,  replicationDegree: 9);
    List<Integer> orderPositions = Utils.positionsForReplicationDegree(replicationDegree);
    for (int i = 0; i < 9; i++){
        if (!orderPositions.contains(i))
            orderPositions.add(i);
    }

    if (idsOfFile.size() == 0)
        return "FAILED: fileId couldn't be generated";

    for (int i = 0; i < idsOfFile.size(); i++){
        this.chordNode.lookup(idsOfFile.get(i));

        try {
            TimeUnit.MILLISECONDS.sleep( timeout: 500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

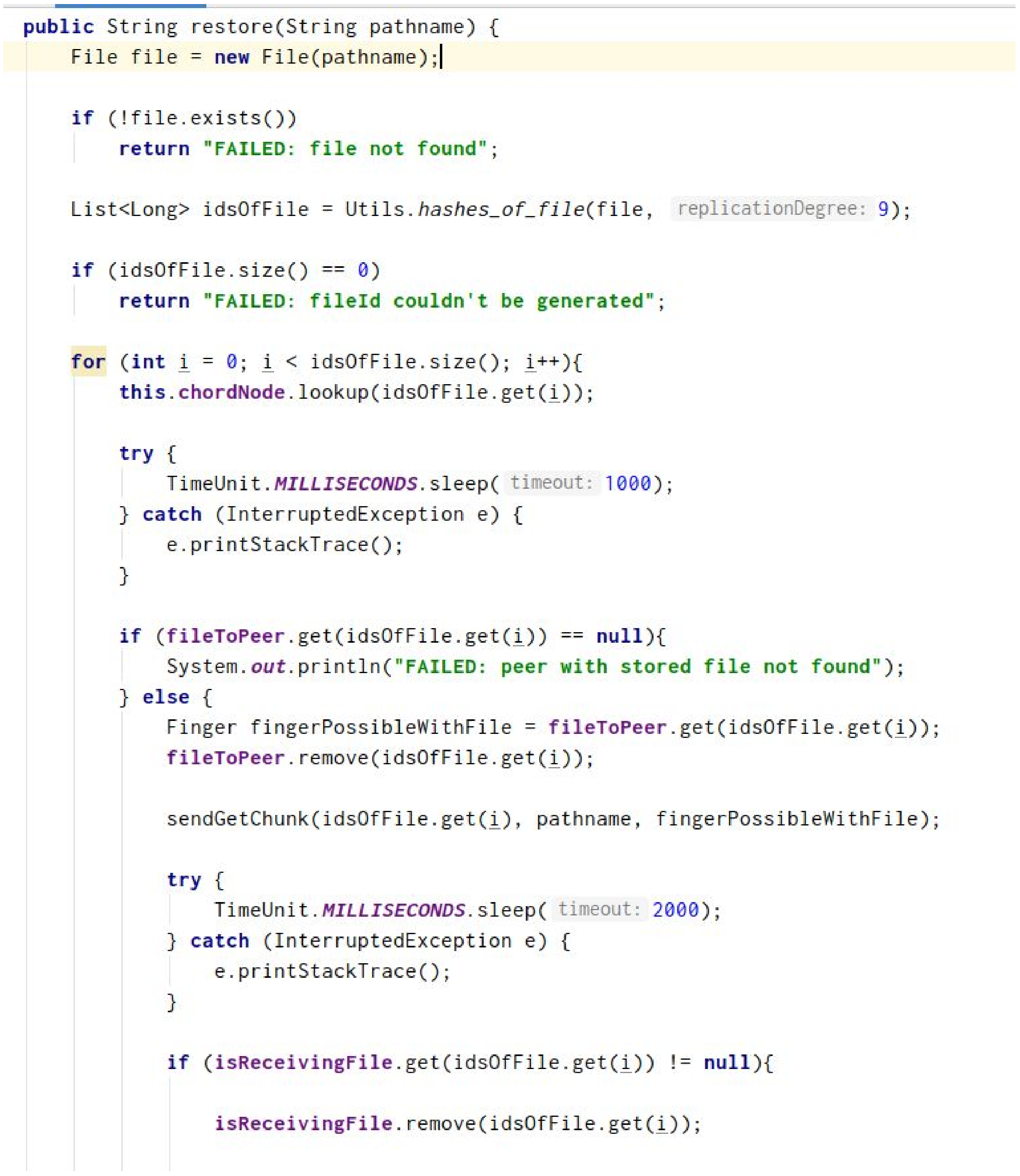
After obtaining the list of possible ids we send the PUTCHUNK message to the peers. The format of the backup TCP messages is the following:
PUT_CHUNK <host> <port> <sending_peer_id> <chunk_number> <replica_id>

As stated above there is a maximum replication degree of 9. To guarantee that each peer only receives a single copy of the file, the initiator peer checks if it has previously sent a copy to that peer, in case it hasn't the backup proceeds as expected.

Although the files are stored as one, on the peers storing the replica, they are received as chunks since there is a 16KB message limit when using SSL over TCP. Initially, we were sending the file as one, but the header was only sent in the first message, to overcome this we preemptively split the file into chunks to guarantee that each message has a header.

Restore

In the restore protocol, we receive the file name and generate all the possible ids it could have in the network. This step allows us to run the restore protocol on peers that have no information on the file and who might have it stored.
With the ids of possible holders of the file we asked one by one for the file and after obtaining the file the loop is stopped and the protocol ends.



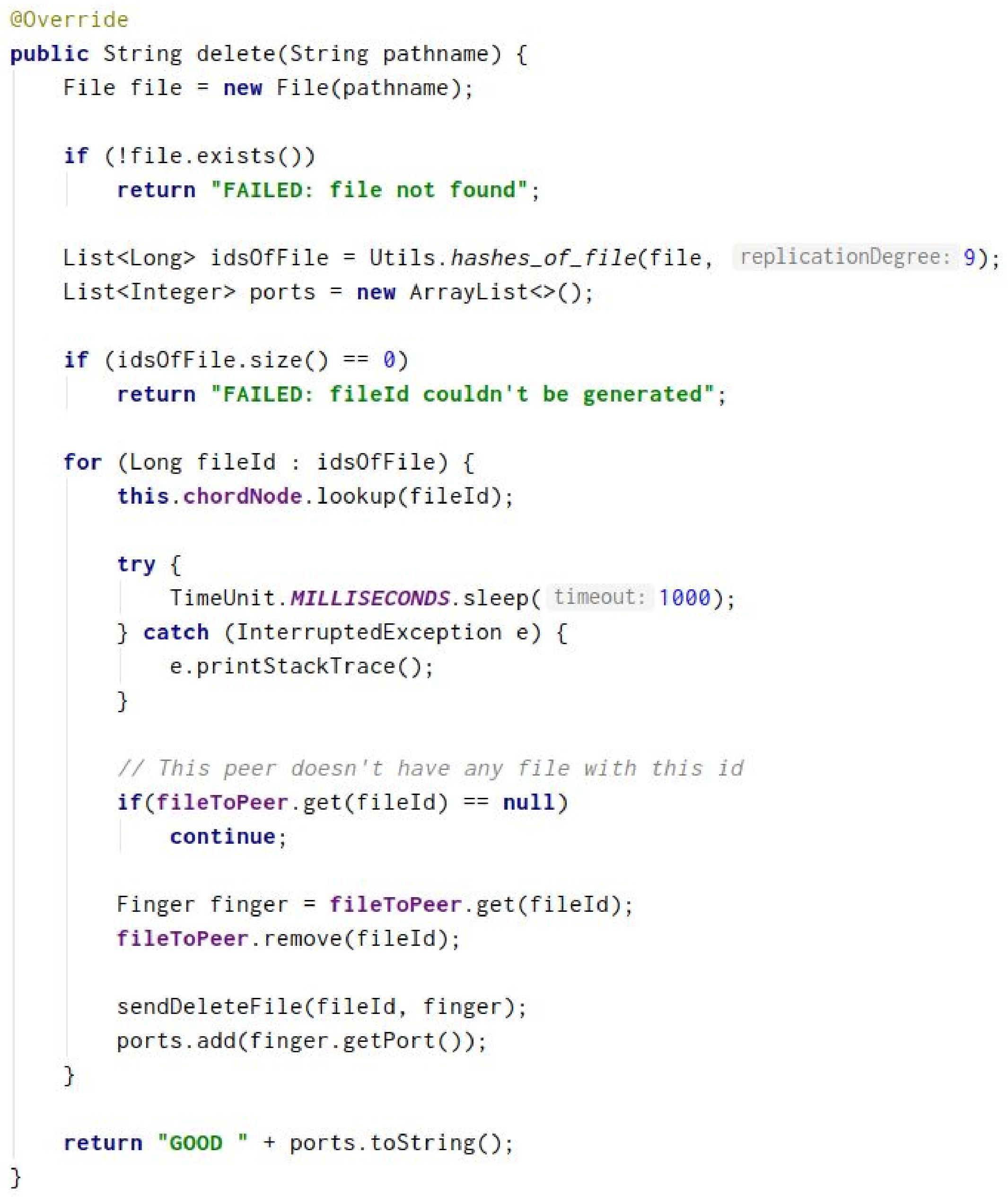
```java
public String restore(String pathname) {
    File file = new File(pathname);

    if (!file.exists())
        return "FAILED: file not found";

    List<Long> idsOfFile = Utils.hashes_of_file(file,  replicationDegree: 9);

    if (idsOfFile.size() == 0)
        return "FAILED: fileId couldn't be generated";

    for (int i = 0; i < idsOfFile.size(); i++){
        this.chordNode.lookup(idsOfFile.get(i));

        try {
            TimeUnit.MILLISECONDS.sleep( timeout: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        if (fileToPeer.get(idsOfFile.get(i)) == null){
            System.out.println("FAILED: peer with stored file not found");
        } else {
            Finger fingerPossibleWithFile = fileToPeer.get(idsOfFile.get(i));
            fileToPeer.remove(idsOfFile.get(i));

            sendGetChunk(idsOfFile.get(i), pathname, fingerPossibleWithFile);

            try {
                TimeUnit.MILLISECONDS.sleep( timeout: 2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            if (isReceivingFile.get(idsOfFile.get(i)) != null){

                isReceivingFile.remove(idsOfFile.get(i));
```

The format of the restore TCP message is the following:
GET_CHUNK <host> <port> <sending_peer_id> <file_id> <file_name>
The answer to this request is as follow:
CHUNK <host> <port> <sending_peer_id> <chunk_number> <file_id> <file_name>

As explained in the backup protocol, when sending the file it is split into various CHUNK messages.

Delete

The delete protocol can be called from any peer since there are scenarios where we don't know the file replication degree we generate every possible id for the file, which is the same as considering the file was backed up with a replication degree of 9.
For the valid peers we send the delete message and remove any information we might have locally.

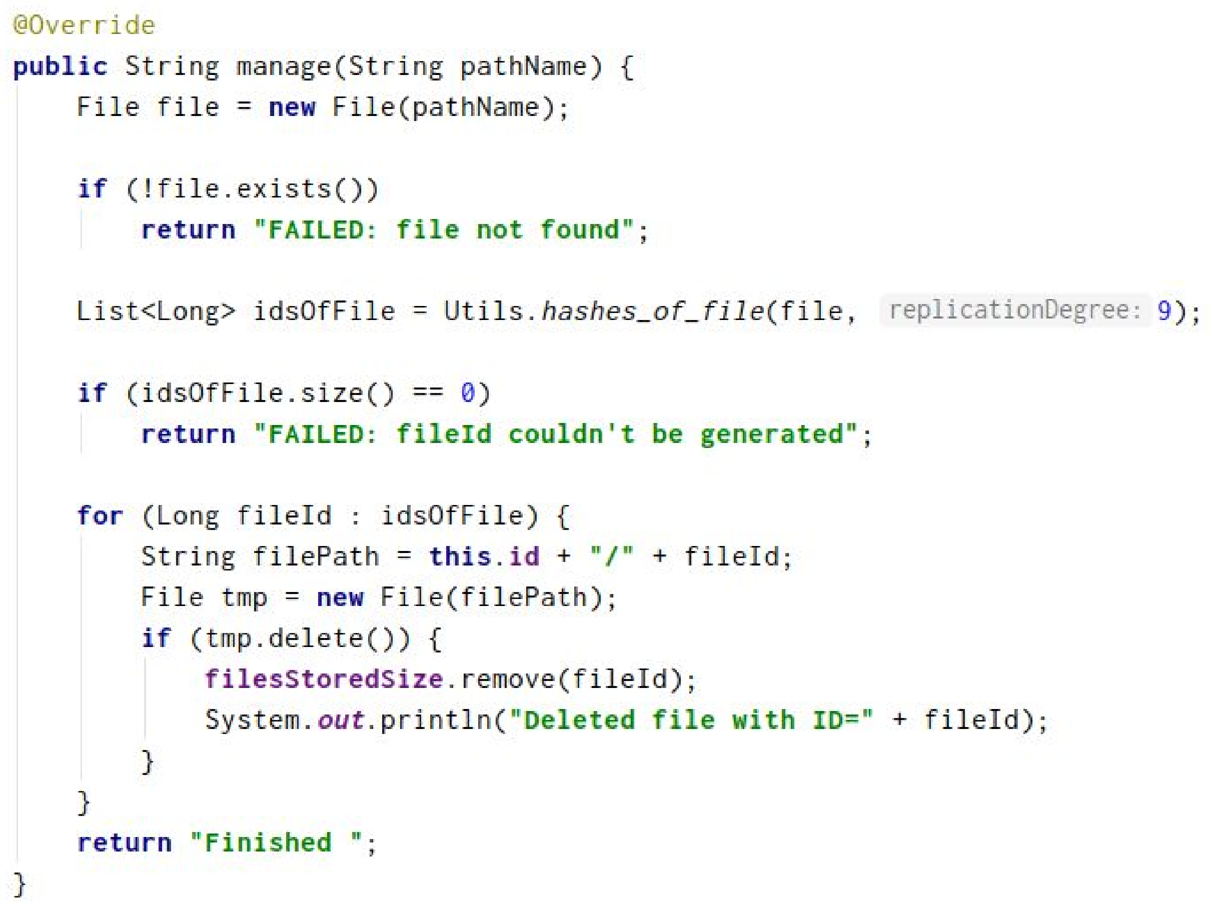


```java
@Override
public String delete(String pathname) {
    File file = new File(pathname);

    if (!file.exists())
        return "FAILED: file not found";

    List<Long> idsOfFile = Utils.hashes_of_file(file, replicationDegree: 9);
    List<Integer> ports = new ArrayList<>();

    if (idsOfFile.size() == 0)
        return "FAILED: fileId couldn't be generated";

    for (Long fileId : idsOfFile) {
        this.chordNode.lookup(fileId);

        try {
            TimeUnit.MILLISECONDS.sleep( timeout: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // This peer doesn't have any file with this id
        if(fileToPeer.get(fileId) == null)
            continue;

        Finger finger = fileToPeer.get(fileId);
        fileToPeer.remove(fileId);

        sendDeleteFile(fileId, finger);
        ports.add(finger.getPort());
    }

    return "GOOD " + ports.toString();
}
```

The format of the delete message is the following:
DELETE <host> <port> <sending_peer_id> <file_id>

Manage

This protocol is similar to the delete one except it only deletes a file locally and does not propagate that change throughout the network and the other peers.
Since we don't know under which id a file is stored in a given peer we try to delete a file with every possible id for the file.

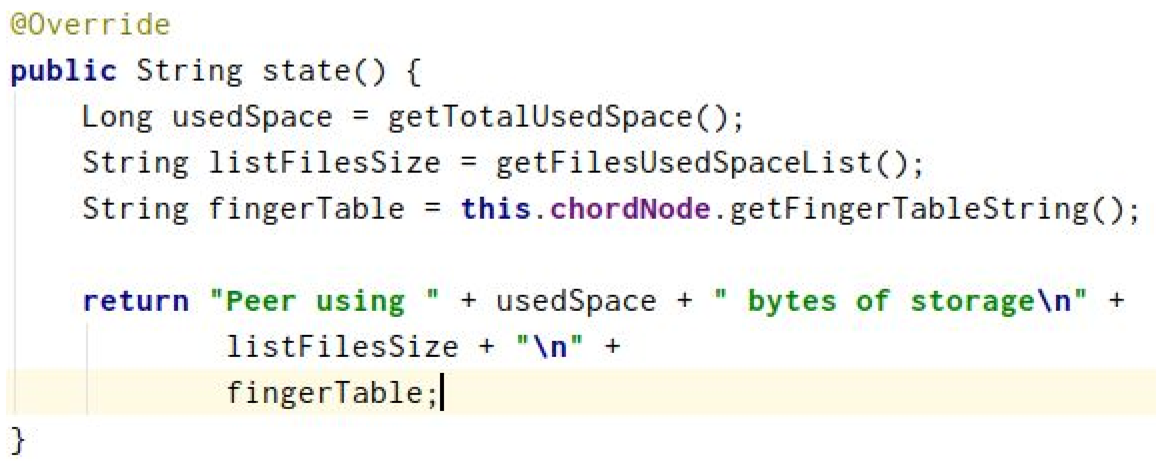


```java
@Override
public String manage(String pathName) {
    File file = new File(pathName);

    if (!file.exists())
        return "FAILED: file not found";

    List<Long> idsOfFile = Utils.hashes_of_file(file,  replicationDegree: 9);

    if (idsOfFile.size() == 0)
        return "FAILED: fileId couldn't be generated";

    for (Long fileId : idsOfFile) {
        String filePath = this.id + "/" + fileId;
        File tmp = new File(filePath);
        if (tmp.delete()) {
            filesStoredSize.remove(fileId);
            System.out.println("Deleted file with ID=" + fileId);
        }
    }
    return "Finished ";
}
```

As this is a local protocol there are no messages being exchanged.

State

This protocol serves more as a debugging interface than one with actual functionalities. This protocol displays the total used space and the size of each stored file. It also displays information relative to the chord algorithm, it presents the list of successors, the id of the predecessor, and the finger table entries.

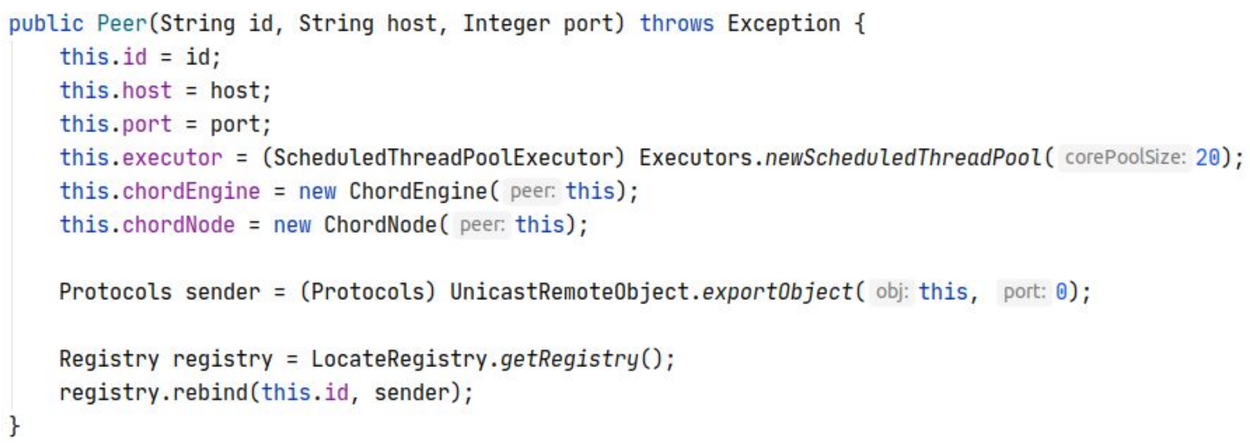


```
@Override
public String state() {
    Long usedSpace = getTotalUsedSpace();
    String listFilesSize = getFilesUsedSpaceList();
    String fingerTable = this.chordNode.getFingerTableString();

    return "Peer using " + usedSpace + " bytes of storage\n" +
            listFilesSize + "\n" +
            fingerTable;
}
```

As with the manage protocol there aren’t any messages being swapped as this is a standalone protocol.

Concurrency design

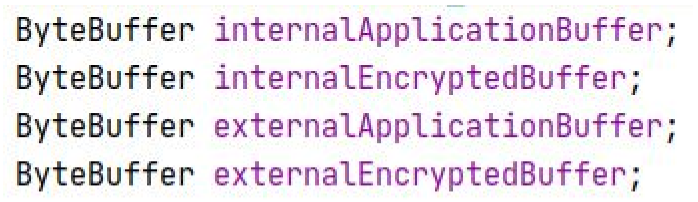
One of our main goals with this project was to achieve the maximum amount of concurrency possible. To do so, we’ve used scheduled thread-pools. We have opted for the scheduled “variant” of thread-pools because it allows us to schedule updates to the peer’s finger table once every 10 seconds. These are created in the Peer’s constructor.



```
public Peer(String id, String host, Integer port) throws Exception {
    this.id = id;
    this.host = host;
    this.port = port;
    this.executor = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool( corePoolSize: 20);
    this.chordEngine = new ChordEngine( peer: this);
    this.chordNode = new ChordNode( peer: this);

    Protocols sender = (Protocols) UnicastRemoteObject.exportObject( obj: this,  port: 0);

    Registry registry = LocateRegistry.getRegistry();
    registry.rebind(this.id, sender);
}
```

We are also making use of the Java NIO library by using several structures from it, such as **ByteBuffers, Selectors, SocketChannels,** and **ServerSocketChannels**.

Every peer has a *NetworkManager* object that has 4 ByteBuffers used to store data going in and out of the peer, both encrypted and unencrypted:



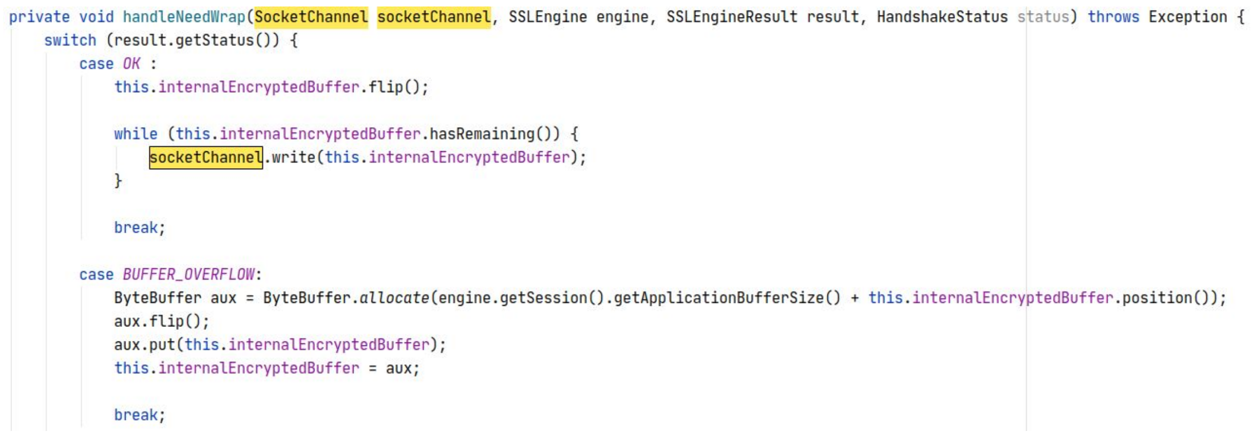
```java
ByteBuffer internalApplicationBuffer;
ByteBuffer internalEncryptedBuffer;
ByteBuffer externalApplicationBuffer;
ByteBuffer externalEncryptedBuffer;
```

internalApplicationBuffer stores the peer's unencrypted data to be sent to other peers.
internalEncryptedBuffer stores the peer's encrypted data to be sent to other peers.
externalApplicationBuffer stores unencrypted data received from other peers.
externalEncryptedBuffer stores encrypted data received from other peers.

Using Java NIO has its downsides, such as making us handle a lot more "could go wrong" aspects of the code. For instance, it was our job to handle buffer overflows (this was handled in sync with the SSLEngine implementation), as well as allocating the needed space for the buffers. Cleaning and flipping the buffers were also aspects that we had to handle ourselves.

However, despite these difficulties, it allowed us to achieve a higher level of concurrency.

Socket channels are the medium that transports data in and out of the previously explicated ByteBuffers. These read from and write to TCP sockets and also encoding and decoding the buffered data properly. We have used both SocketChannels and ServerSocketChannels in the *NetworkManager* class whenever there is communication between peers (when handshaking and transferring data chunks, for example).

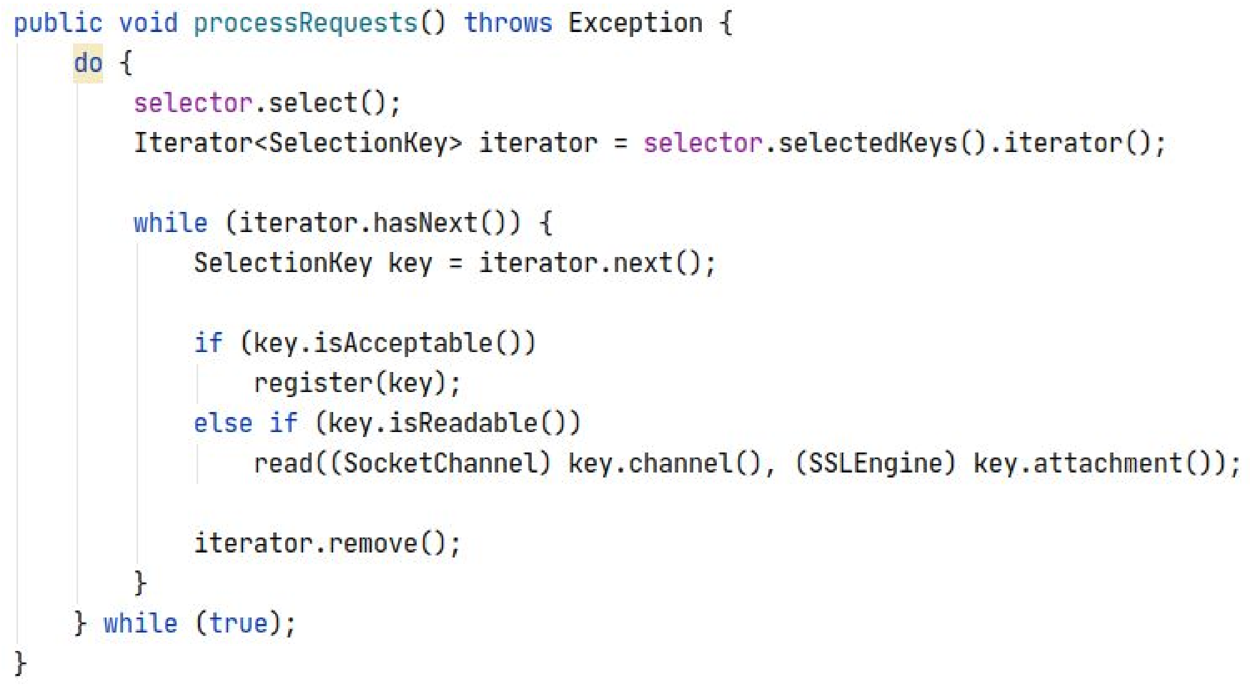


```java
private void handleNeedWrap(SocketChannel socketChannel, SSLEngine engine, SSLEngineResult result, HandshakeStatus status) throws Exception {
    switch (result.getStatus()) {
        case OK :
            this.internalEncryptedBuffer.flip();

            while (this.internalEncryptedBuffer.hasRemaining()) {
                socketChannel.write(this.internalEncryptedBuffer);
            }

            break;

        case BUFFER_OVERFLOW:
            ByteBuffer aux = ByteBuffer.allocate(engine.getSession().getApplicationBufferSize() + this.internalEncryptedBuffer.position());
            aux.flip();
            aux.put(this.internalEncryptedBuffer);
            this.internalEncryptedBuffer = aux;

            break;
```

Example usage of SocketChannel when wrapping data

We have also used *Selectors* on the *server-side* of each Peer (each one has a Server and Client object that gives them the ability to act as both whenever needed. We thought this separation made the code more legible and well-structured). These allow a single thread to examine I/O events on multiple channels. This way, whenever a channel is ready for writing or

reading, the selector will act accordingly. In order for this to happen, we register the created channels in the selector.
When a new request comes in, the server iterates through the SelectionKeys (one assigned to each registered channel) and reads the data when it’s available to be read.



```java
public void processRequests() throws Exception {
    do {
        selector.select();
        Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();

        while (iterator.hasNext()) {
            SelectionKey key = iterator.next();

            if (key.isAcceptable())
                register(key);
            else if (key.isReadable())
                read((SocketChannel) key.channel(), (SSLEngine) key.attachment());

            iterator.remove();
        }
    } while (true);
}
```

Processing of new requests by the *Server* component of the peer and handling of SelectionKeys

JSSE

We have used the SSLEngine class which enables secure communications using protocols such as SSL and TLS.

We are using JSSE for all communications, in all protocols. In order to make use of the integrity, authentication, and privacy protection features provided by this interface, we have used the following cipher suites:

- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384;
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256;
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384;
- TLS_RSA_WITH_AES_256_GCM_SHA384;
- TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384;
- TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384;
- TLS_DHE_RSA_WITH_AES_256_GCM_SHA384;

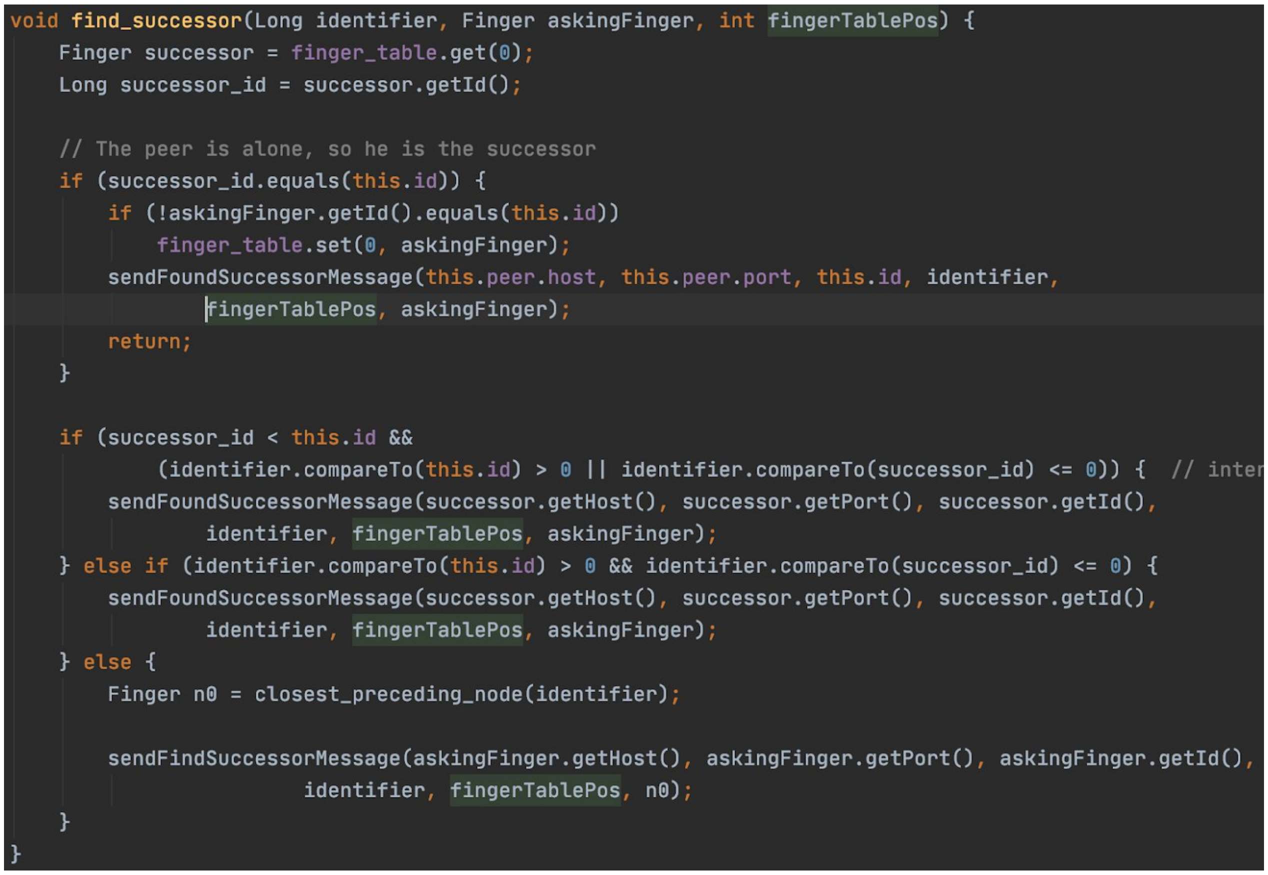
- TLS_DHE_DSS_WITH_AES_256_GCM_SHA384;
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256;
- TLS_RSA_WITH_AES_128_GCM_SHA256;
- TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256;
- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256;
- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256;
- TLS_DHE_DSS_WITH_AES_128_GCM_SHA256;
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384;
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384;
- TLS_RSA_WITH_AES_256_CBC_SHA256;
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384;
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384;
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA256;
- TLS_DHE_DSS_WITH_AES_256_CBC_SHA256;
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA;
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA;
- TLS_RSA_WITH_AES_256_CBC_SHA;
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA;
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA;
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA;
- TLS_DHE_DSS_WITH_AES_256_CBC_SHA;
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256;
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256;
- TLS_RSA_WITH_AES_128_CBC_SHA256;
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256;
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256;
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256;
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA256;
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA;
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA;
- TLS_RSA_WITH_AES_128_CBC_SHA;
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA;
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA;
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA;
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA;
- TLS_EMPTY_RENEGOTIATION_INFO_SCSV.

Scalability

We tried to achieve a very high degree of scalability in order to ensure that our service runs with minimal effort in small and big scale systems.

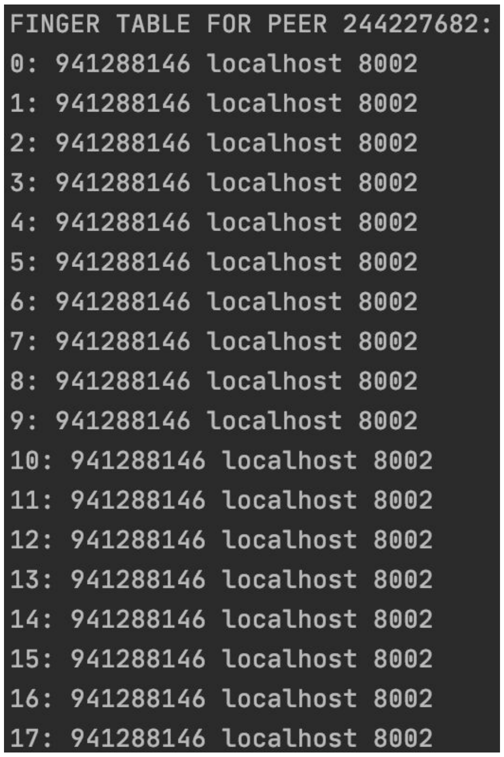
To accomplish this goal we are using the peer-to-peer **Chord** protocol to distribute the files, **Thread-pools** to achieve concurrency, as well as **Java NIO** for asynchronous I/O.

By using the **Chord** protocol we ensure that we have an algorithm to determine in which peer one file is stored or will be stored. In our implementation, each peer corresponds to a node in the chord. In order for a peer to communicate with other peers, it has a *finger table* that stores information on which host and port some of its successors are running. Due to the nature of the Chord protocol its scalability is ensured by its logarithmic file and peer search time.

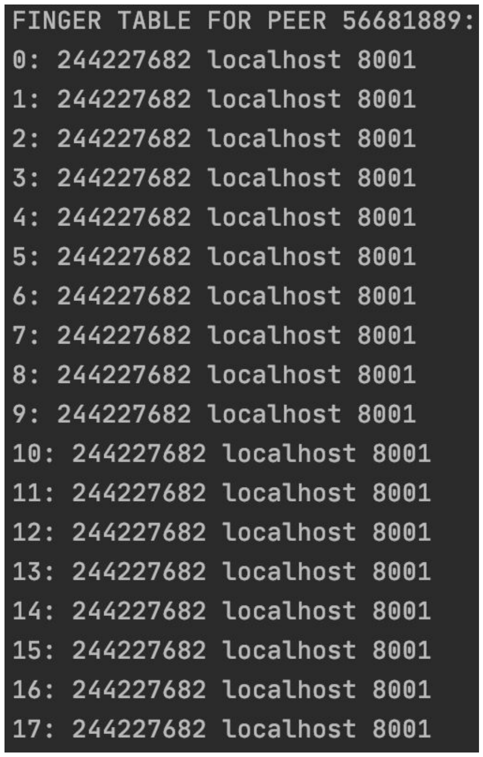


```java
void find_successor(Long identifier, Finger askingFinger, int fingerTablePos) {
    Finger successor = finger_table.get(0);
    Long successor_id = successor.getId();

    // The peer is alone, so he is the successor
    if (successor_id.equals(this.id)) {
        if (!askingFinger.getId().equals(this.id))
            finger_table.set(0, askingFinger);
        sendFoundSuccessorMessage(this.peer.host, this.peer.port, this.id, identifier,
                fingerTablePos, askingFinger);
        return;
    }

    if (successor_id < this.id &&
            (identifier.compareTo(this.id) > 0 || identifier.compareTo(successor_id) <= 0)) {  // inter
        sendFoundSuccessorMessage(successor.getHost(), successor.getPort(), successor.getId(),
                identifier, fingerTablePos, askingFinger);
    } else if (identifier.compareTo(this.id) > 0 && identifier.compareTo(successor_id) <= 0) {
        sendFoundSuccessorMessage(successor.getHost(), successor.getPort(), successor.getId(),
                identifier, fingerTablePos, askingFinger);
    } else {
        Finger n0 = closest_preceding_node(identifier);

        sendFindSuccessorMessage(askingFinger.getHost(), askingFinger.getPort(), askingFinger.getId(),
                        identifier, fingerTablePos, n0);
    }
}
```

Function to find the successor of a peer or file



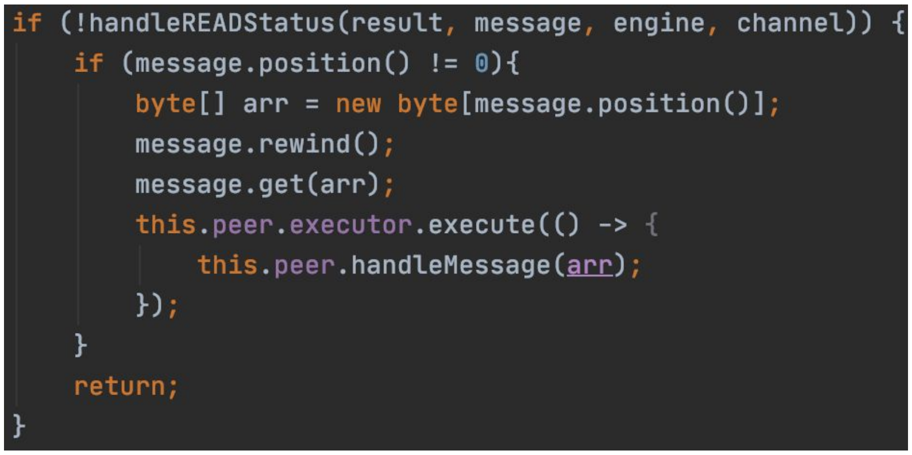
```
FINGER TABLE FOR PEER 244227682:
0: 941288146 localhost 8002
1: 941288146 localhost 8002
2: 941288146 localhost 8002
3: 941288146 localhost 8002
4: 941288146 localhost 8002
5: 941288146 localhost 8002
6: 941288146 localhost 8002
7: 941288146 localhost 8002
8: 941288146 localhost 8002
9: 941288146 localhost 8002
10: 941288146 localhost 8002
11: 941288146 localhost 8002
12: 941288146 localhost 8002
13: 941288146 localhost 8002
14: 941288146 localhost 8002
15: 941288146 localhost 8002
16: 941288146 localhost 8002
17: 941288146 localhost 8002
```



```
FINGER TABLE FOR PEER 56681889:
0: 244227682 localhost 8001
1: 244227682 localhost 8001
2: 244227682 localhost 8001
3: 244227682 localhost 8001
4: 244227682 localhost 8001
5: 244227682 localhost 8001
6: 244227682 localhost 8001
7: 244227682 localhost 8001
8: 244227682 localhost 8001
9: 244227682 localhost 8001
10: 244227682 localhost 8001
11: 244227682 localhost 8001
12: 244227682 localhost 8001
13: 244227682 localhost 8001
14: 244227682 localhost 8001
15: 244227682 localhost 8001
16: 244227682 localhost 8001
17: 244227682 localhost 8001
```

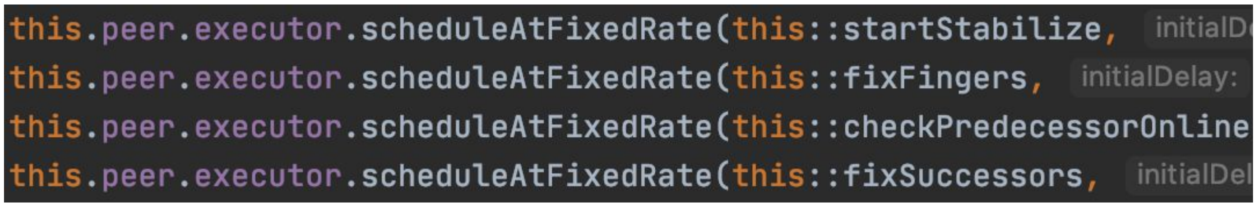
Example of two finger tables of different peers.

By using **Thread-pools** we ensure that we can be executing multiple tasks at the same time in the same peer. With this, it is possible to receive and send messages to other peers, process multiple received messages, and run the chord stabilization functions all at the same time. However, since it is possible to be running multiple actions at the same time we had to take some precautions in terms of synchronization to ensure the integrity of all data, such as using *synchronized* blocks and *ConcurrentHashMaps*.



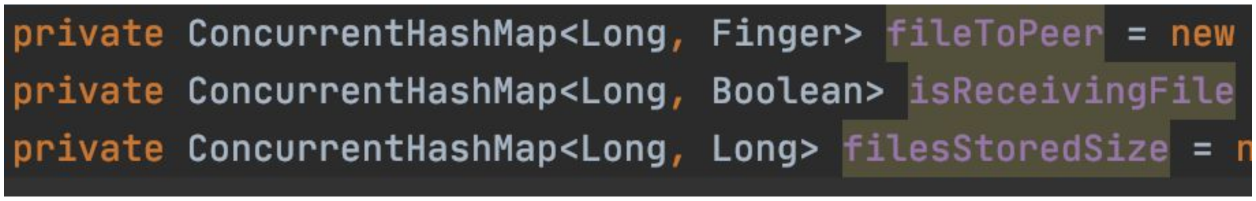
```java
if (!handleREADStatus(result, message, engine, channel)) {
    if (message.position() != 0){
        byte[] arr = new byte[message.position()];
        message.rewind();
        message.get(arr);
        this.peer.executor.execute(() -> {
            this.peer.handleMessage(arr);
        });
    }
    return;
}
```

Creation of a new thread to process an incoming message



```
this.peer.executor.scheduleAtFixedRate(this::startStabilize,  initialDe
this.peer.executor.scheduleAtFixedRate(this::fixFingers,  initialDelay:
this.peer.executor.scheduleAtFixedRate(this::checkPredecessorOnline
this.peer.executor.scheduleAtFixedRate(this::fixSuccessors,  initialDel
```

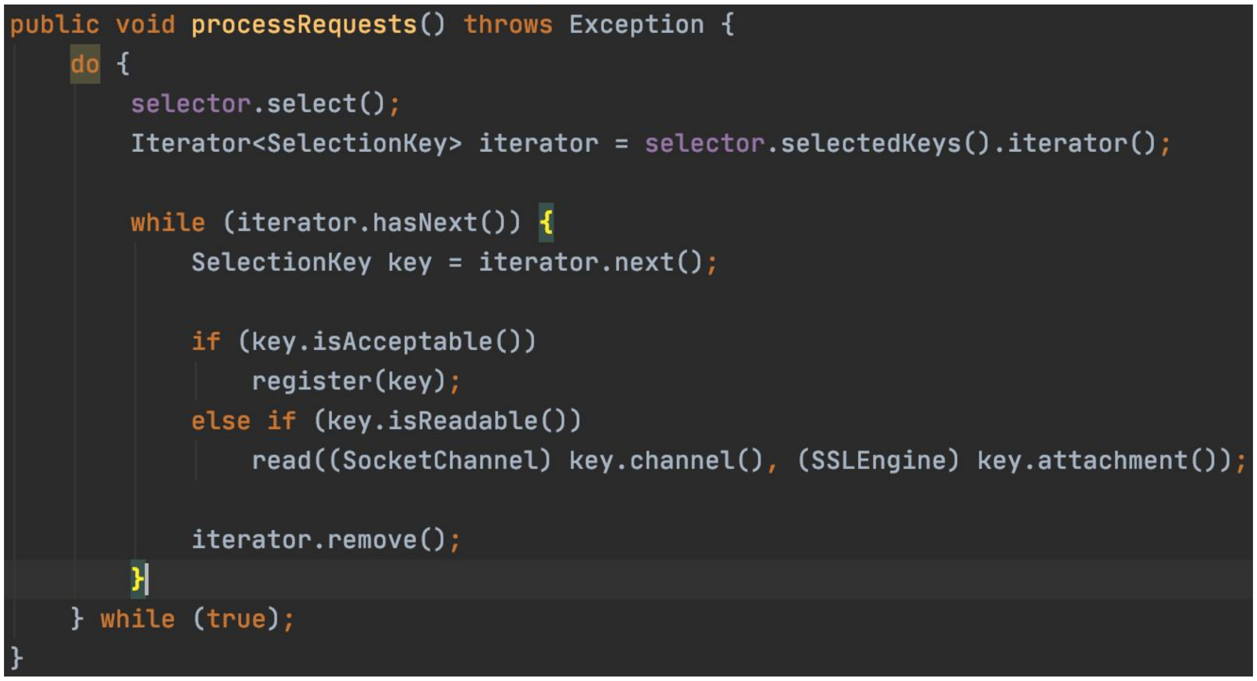
Use of the Thread-Pool to run multiple functions at fixed rates



```
private ConcurrentHashMap<Long, Finger> fileToPeer = new
private ConcurrentHashMap<Long, Boolean> isReceivingFile
private ConcurrentHashMap<Long, Long> filesStoredSize = n
```

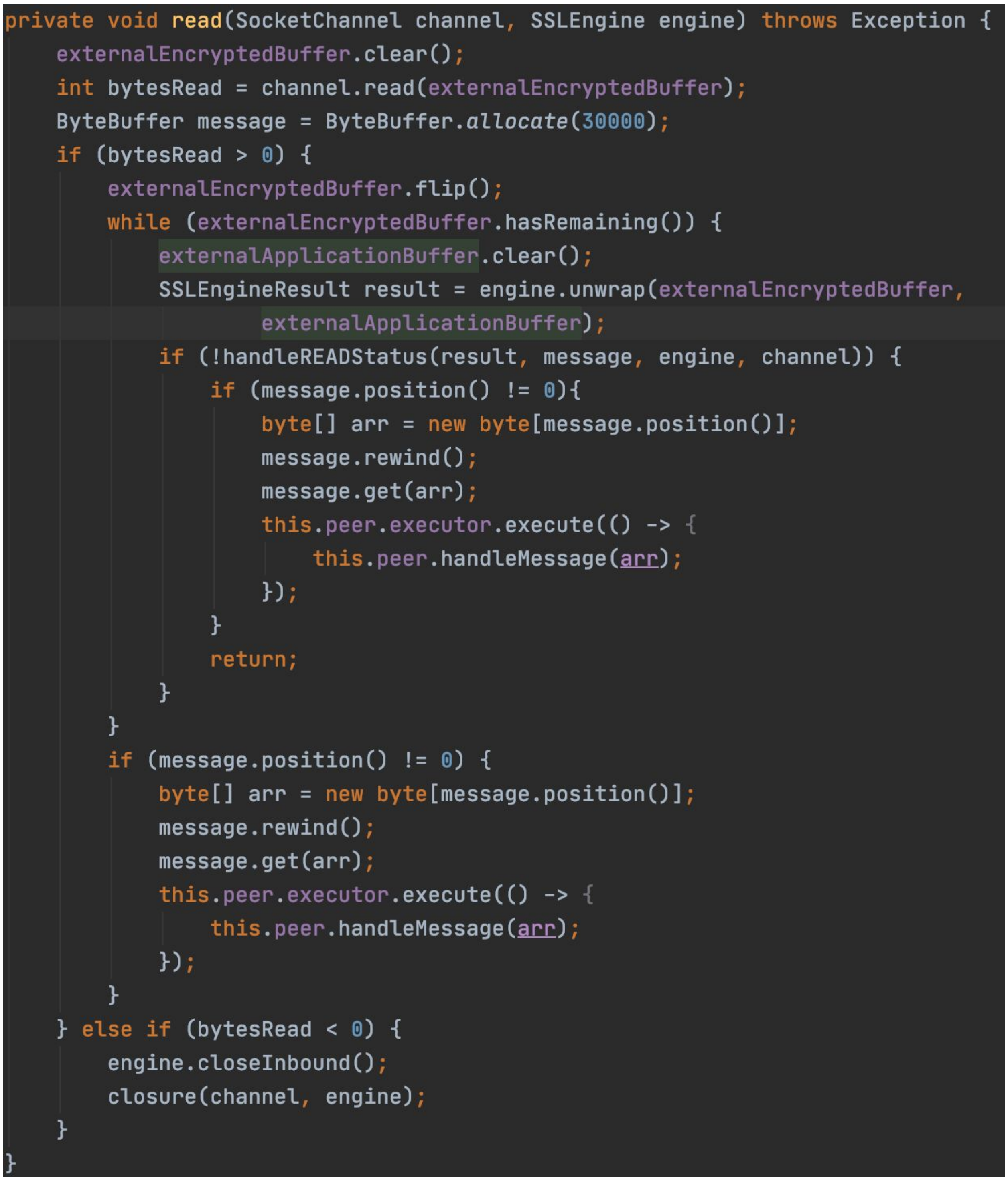
Use of ConcurrentHashMaps to ensure the integrity of data and peer states

By using **Java NIO** we are allowed to manage multiple channels using very few threads. Which, in the case where there are a lot of multiple open connections can be proved more efficient that the blocking IO counterpart.



```
public void processRequests() throws Exception {
    do {
        selector.select();
        Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();

        while (iterator.hasNext()) {
            SelectionKey key = iterator.next();

            if (key.isAcceptable())
                register(key);
            else if (key.isReadable())
                read((SocketChannel) key.channel(), (SSLEngine) key.attachment());

            iterator.remove();
        }
    } while (true);
}
```

Use of a selector for channels management

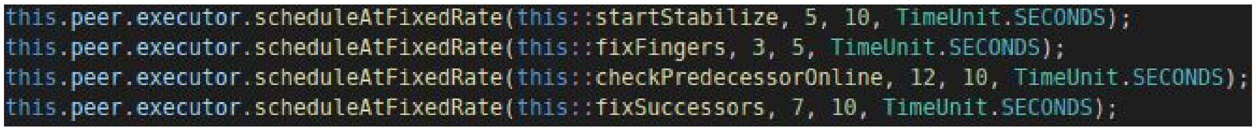


```java
private void read(SocketChannel channel, SSLEngine engine) throws Exception {
    externalEncryptedBuffer.clear();
    int bytesRead = channel.read(externalEncryptedBuffer);
    ByteBuffer message = ByteBuffer.allocate(30000);
    if (bytesRead > 0) {
        externalEncryptedBuffer.flip();
        while (externalEncryptedBuffer.hasRemaining()) {
            externalApplicationBuffer.clear();
            SSLEngineResult result = engine.unwrap(externalEncryptedBuffer,
                    externalApplicationBuffer);
            if (!handleREADStatus(result, message, engine, channel)) {
                if (message.position() != 0){
                    byte[] arr = new byte[message.position()];
                    message.rewind();
                    message.get(arr);
                    this.peer.executor.execute(() -> {
                        this.peer.handleMessage(arr);
                    });
                }
                return;
            }
        }
        if (message.position() != 0) {
            byte[] arr = new byte[message.position()];
            message.rewind();
            message.get(arr);
            this.peer.executor.execute(() -> {
                this.peer.handleMessage(arr);
            });
        }
    } else if (bytesRead < 0) {
        engine.closeInbound();
        closure(channel, engine);
    }
}
```

Function to read from a channel

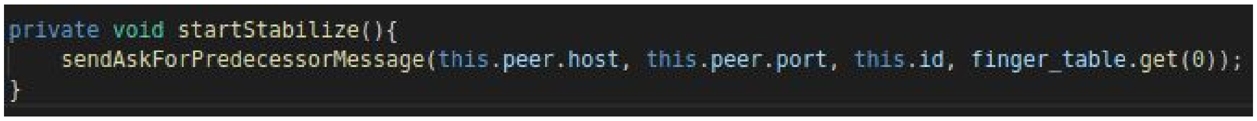
Fault-tolerance

In our project, as we chose a decentralized design, more specifically Chord, its fault-tolerant features help keeping the chord ring maintain its stabilization after node joins and departures. We also keep a list of successors (not only one) so that if one fails the node can still contact another successor, maintaining the chord ring fully operational.



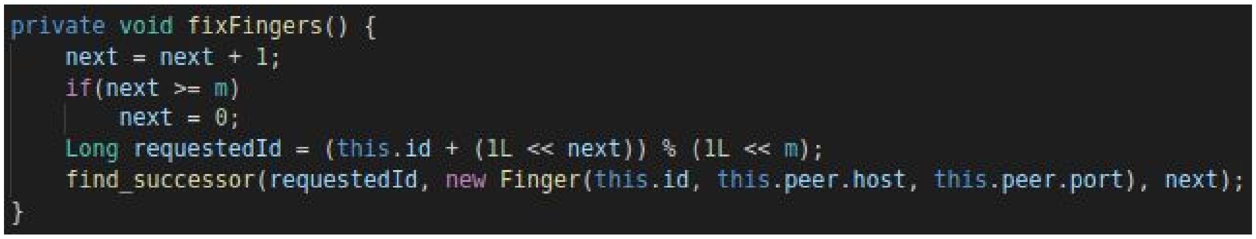
```java
this.peer.executor.scheduleAtFixedRate(this::startStabilize, 5, 10, TimeUnit.SECONDS);
this.peer.executor.scheduleAtFixedRate(this::fixFingers, 3, 5, TimeUnit.SECONDS);
this.peer.executor.scheduleAtFixedRate(this::checkPredecessorOnline, 12, 10, TimeUnit.SECONDS);
this.peer.executor.scheduleAtFixedRate(this::fixSuccessors, 7, 10, TimeUnit.SECONDS);
```

Each of these functions are continuously called according to the defined time intervals, promoting constant stabilization.



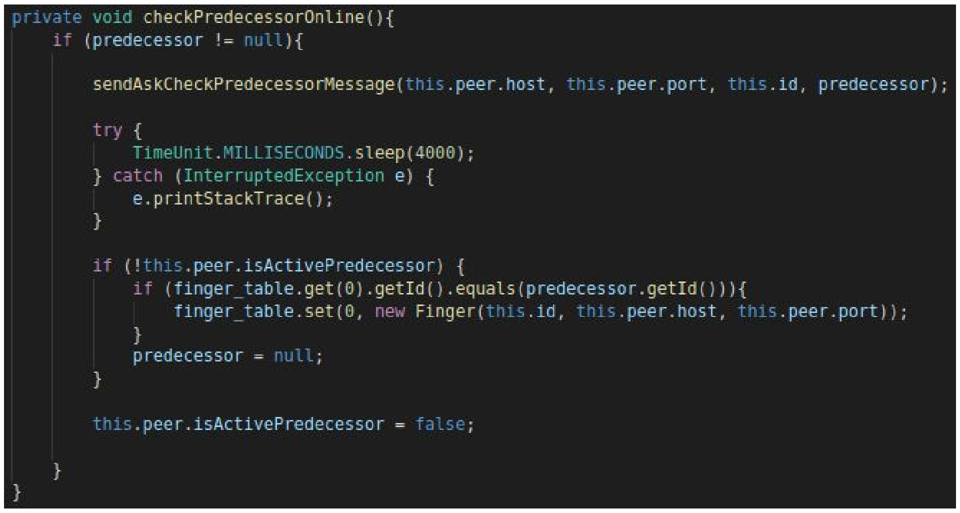
```java
private void startStabilize(){
    sendAskForPredecessorMessage(this.peer.host, this.peer.port, this.id, finger_table.get(0));
}
```

Function startStabilize() checks if the successor's predecessor node is the current node, if not we set the current node’s successor to that. This ensures integrity of the chord when a new peer joins.



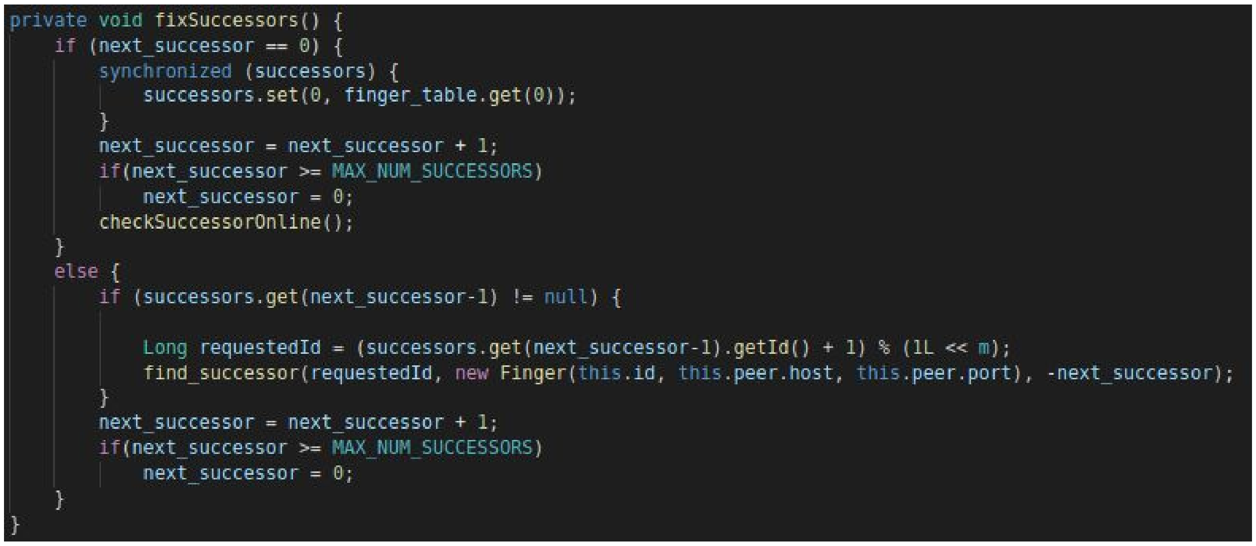
```java
private void fixFingers() {
    next = next + 1;
    if(next >= m)
        next = 0;
    Long requestedId = (this.id + (1L << next)) % (1L << m);
    find_successor(requestedId, new Finger(this.id, this.peer.host, this.peer.port), next);
}
```

The function fixFingers() is run periodically and it updates the finger table one position at a time in order for the node to know all changes made to the chord between runs.



```
private void checkPredecessorOnline(){
    if (predecessor != null){

        sendAskCheckPredecessorMessage(this.peer.host, this.peer.port, this.id, predecessor);

        try {
            TimeUnit.MILLISECONDS.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        if (!this.peer.isActivePredecessor) {
            if (finger_table.get(0).getId().equals(predecessor.getId())){
                finger_table.set(0, new Finger(this.id, this.peer.host, this.peer.port));
            }
            predecessor = null;
        }

        this.peer.isActivePredecessor = false;

    }
}
```

Function checkPredecessorOnline() checks if the predecessor of the current node is online, if not its predecessor's ids are turned into usable ids for that node. Also sets the predecessor to null, making it available to accept another node as a predecessor.



```
private void fixSuccessors() {
    if (next_successor == 0) {
        synchronized (successors) {
            successors.set(0, finger_table.get(0));
        }
        next_successor = next_successor + 1;
        if(next_successor >= MAX_NUM_SUCCESSORS)
            next_successor = 0;
        checkSuccessorOnline();
    }
    else {
        if (successors.get(next_successor-1) != null) {

            Long requestedId = (successors.get(next_successor-1).getId() + 1) % (1L << m);
            find_successor(requestedId, new Finger(this.id, this.peer.host, this.peer.port), -next_successor);
        }
        next_successor = next_successor + 1;
        if(next_successor >= MAX_NUM_SUCCESSORS)
            next_successor = 0;
    }
}
```

Function fixSuccessors() checks if the list of successors is correct. In the case of the first direct successor going offline it sets its successor as the next in the list. This function ensures that even if a peer disconnects from the chord, the remaining ones will be working as expected.