U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems

Second Project
2019/20

Class 1 - Group 22

João Paulo Monteiro Leite -- up201705312@fe.up.pt
Márcia Isabel Reis Teixeira -- up201706065@fe.up.pt
Maria Helena Ferreira -- up201704508@fe.up.pt
Sofia de Araújo Lajes -- up201704066@fe.up.pt

Introduction

This report aims to describe how we designed and implemented a peer-to-peer distributed backup service for the Internet using only Java SE. The idea is to use the free disk space of the computers on the Internet for backing up files on one's own computer. The service must support the backup, restore, and deletion of files, keeping in mind the concurrency in the protocols' development. In addition, the participants in the service must retain total control over their own storage, and therefore they may delete copies of files that they have previously stored.

Overview

Our project presents a solution using TCP and includes SSLSockets, scalability using the Chord algorithm, and Java Nio for file access. Finally, it also deals with fault-tolerance, assuring the records of the peer unavailable for a period are according to the reality.

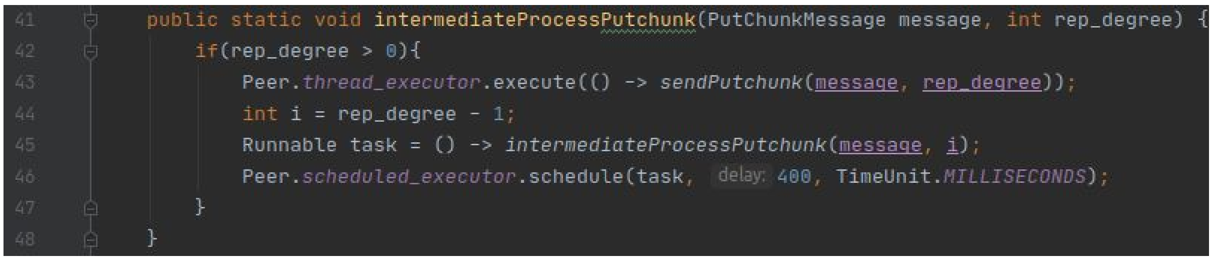
Protocols

All protocols resort to RMI to use the Test Application - which remains the same as in project one - this module contacts the peer to request a protocol's execution; all connections use TCP and only SSLSockets are used, however, these protocols are independent to the way the connection between peers is defined, and therefore, it shall be mentioned in JSSE topic. The approach used in the protocols is described as follows.

Backup

By calling "java [class path.]TestApp <peer access point> BACKUP <file's path> <replication degree>", the peer with access point will first divide the requested file into chunks and then execute BACKUP protocol (which is described in class BackupProtocol in project.protocols module).

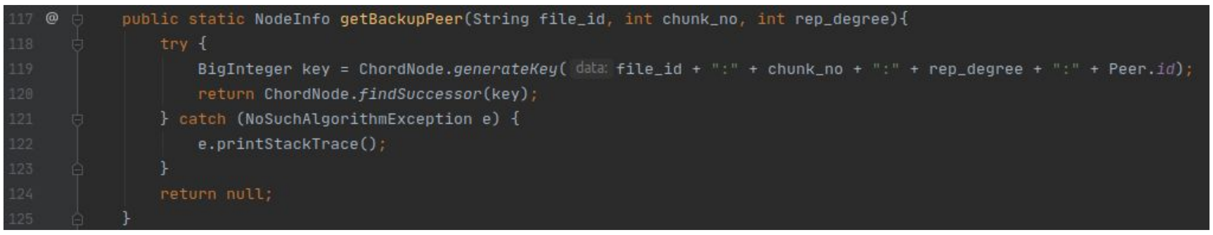
Each chunk must be stored in a number of distinct peers equal to the replication degree desired for the file, so the operation to send each chunk has a cycle that sends a PUTCHUNK for each $i \in [0, replication\ degree[$, i being a natural number, which can be observed in *intermediateProcessPutchunk* method.



```java
public static void intermediateProcessPutchunk(PutChunkMessage message, int rep_degree) {
    if(rep_degree > 0){
        Peer.thread_executor.execute(() -> sendPutchunk(message, rep_degree));
        int i = rep_degree - 1;
        Runnable task = () -> intermediateProcessPutchunk(message, i);
        Peer.scheduled_executor.schedule(task, delay: 400, TimeUnit.MILLISECONDS);
    }
}
```

Figure 1: *intermediateProcessPutchunk* method (BackupProtocol class in project.protocols module)

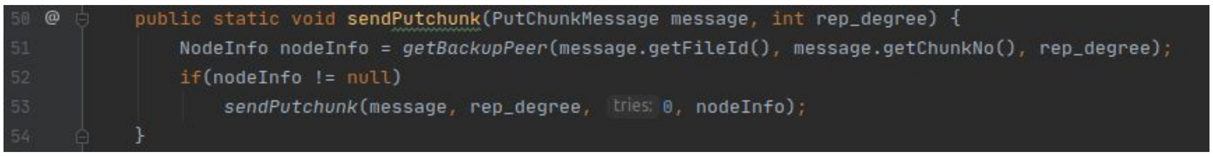
Since we communicate through TCP, we must decide which peers are to be requested to save the chunk. Since the Chord algorithm was implemented, peers within the same network have unique keys, and every peer can reach any peer by knowing its key. Therefore, for each PUTCHUNK message, the key in which we want to save it in is generated by hashing (with sha256 algorithm) a string composed by the following elements: file id, chunk number, i (value mentioned above) and peer's id (sent via argument of the class' constructor), concatenated with a ":" between each two. After the key is decided, it is necessary to find the successor - through Chord - that best fits (the peer with key closest to the required by equal or excess, let's call this peer X). This process is described in method getBackupPeer and the hash is generated in method *generateKey* of ChordNode class (in project.peer module).



```java
public static NodeInfo getBackupPeer(String file_id, int chunk_no, int rep_degree){
    try {
        BigInteger key = ChordNode.generateKey( data: file_id + ":" + chunk_no + ":" + rep_degree + ":" + Peer.id);
        return ChordNode.findSuccessor(key);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    return null;
}
```

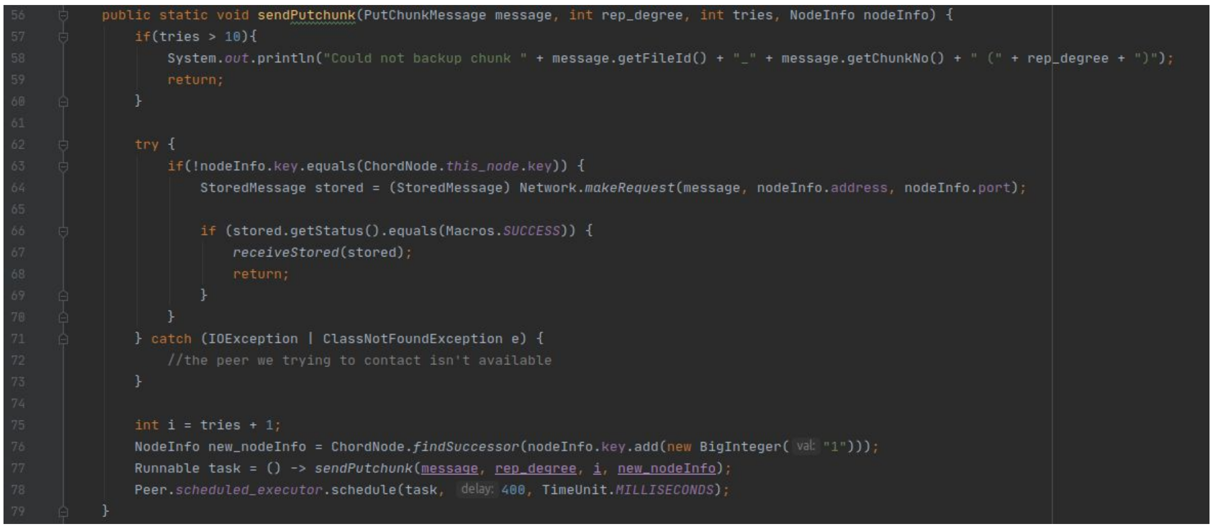
Figure 2: *getBackupPeer* method (BackupProtocol class in project.protocols module)

If, however, X is not able to save the chunk (there is no storage space available, it has previously stored the chunk or is the same peer that requested the backup) then he returns a STORED message with status "FAILED", for which we proceed to add 1 unit to X's key and, once again, find the node to which the new key corresponds to (it will compulsorily be X's immediate successor); if he is not able to either, we ask it's successor, and so consecutively, until we reach a maximum number of tries and, if so, fails, or the chunk is successfully stored.



```java
public static void sendPutchunk(PutChunkMessage message, int rep_degree) {
    NodeInfo nodeInfo = getBackupPeer(message.getFileId(), message.getChunkNo(), rep_degree);
    if(nodeInfo != null)
        sendPutchunk(message, rep_degree,  tries: 0, nodeInfo);
}
```

Figure 3: this *sendPutchunk* method is used to start the process; after getting the peer (node) to request, it calls the function's overload (BackupProtocol class in project.protocols module)

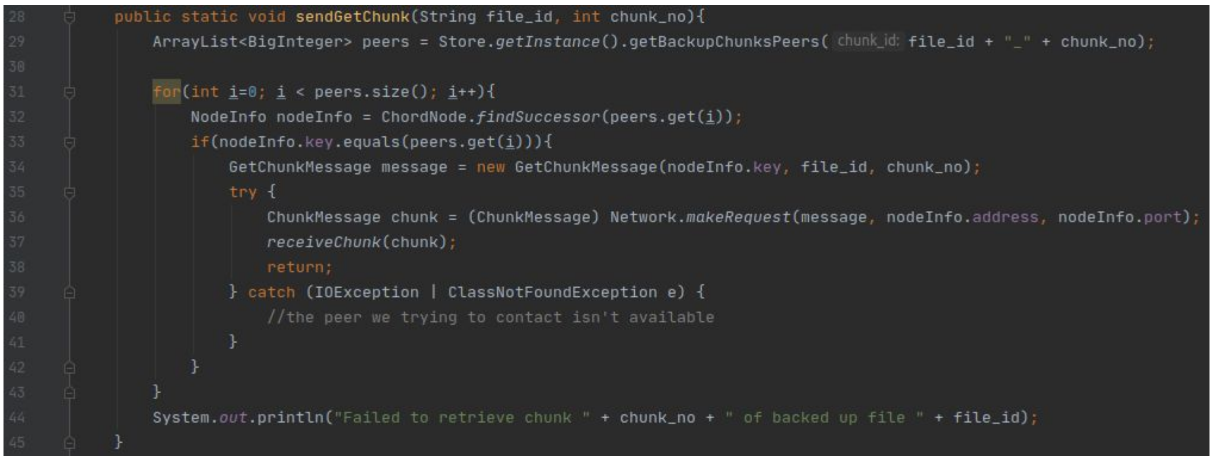


```java
public static void sendPutchunk(PutChunkMessage message, int rep_degree, int tries, NodeInfo nodeInfo) {
    if(tries > 10){
        System.out.println("Could not backup chunk " + message.getFileId() + "_" + message.getChunkNo() + " (" + rep_degree + ")");
        return;
    }

    try {
        if(!nodeInfo.key.equals(ChordNode.this_node.key)) {
            StoredMessage stored = (StoredMessage) Network.makeRequest(message, nodeInfo.address, nodeInfo.port);

            if (stored.getStatus().equals(Macros.SUCCESS)) {
                receiveStored(stored);
                return;
            }
        }
    } catch (IOException | ClassNotFoundException e) {
        //the peer we trying to contact isn't available
    }

    int i = tries + 1;
    NodeInfo new_nodeInfo = ChordNode.findSuccessor(nodeInfo.key.add(new BigInteger( val: "1")));
    Runnable task = () -> sendPutchunk(message, rep_degree, i, new_nodeInfo);
    Peer.scheduled_executor.schedule(task,  delay: 400, TimeUnit.MILLISECONDS);
}
```

Figure 4: this *sendPutchunk* method tests if the maximum number of tries has been achieved, if not, tries to contact, the peer sent in parameter nodeInfo, if it is successful, the process ends there, if not, tries to contact the nodeInfo's immediate successor (BackupProtocol class in project.protocols module)

Restore

When the restore protocol is requested to a peer through the TestApp, this will create a process for each chunk to be retrieved (running sendGetChunk method of RestoreProtocol class). First, a list of all the peers that have stored that chunk (these registers are obtained during the backup protocol) is retrieved through singleton object Store (in project.store module), then we try to contact each peer, one by one, until one of them successfully returns the requested chunk.

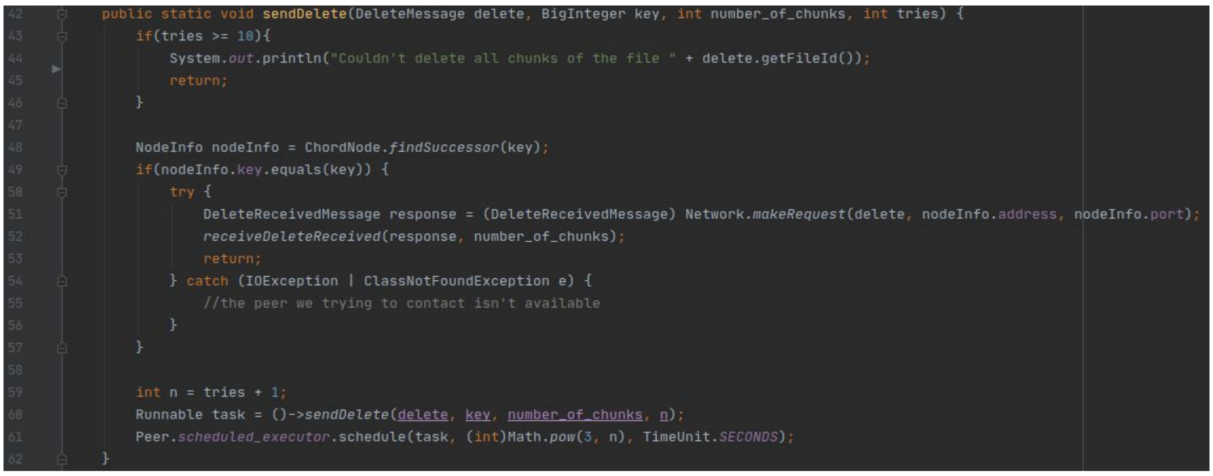


```java
public static void sendGetChunk(String file_id, int chunk_no){
    ArrayList<BigInteger> peers = Store.getInstance().getBackupChunksPeers( chunk_id: file_id + "_" + chunk_no);

    for(int i=0; i < peers.size(); i++){
        NodeInfo nodeInfo = ChordNode.findSuccessor(peers.get(i));
        if(nodeInfo.key.equals(peers.get(i))){
            GetChunkMessage message = new GetChunkMessage(nodeInfo.key, file_id, chunk_no);
            try {
                ChunkMessage chunk = (ChunkMessage) Network.makeRequest(message, nodeInfo.address, nodeInfo.port);
                receiveChunk(chunk);
                return;
            } catch (IOException | ClassNotFoundException e) {
                //the peer we trying to contact isn't available
            }
        }
    }
    System.out.println("Failed to retrieve chunk " + chunk_no + " of backed up file " + file_id);
}
```

Figure 5: *sendGetChunk* method (RestoreProtocol class in project.protocols module)

Delete

As stated before, the Backup protocol registers the key of all peers that stored each chunk. Therefore, the Delete protocol simply sends a direct message to the registered peers in order to delete all chunks of the specified file. If a peer is not responding to the connection attempt, after 10 tries, the initiator prints an informative error and deletes it from the record anyways.

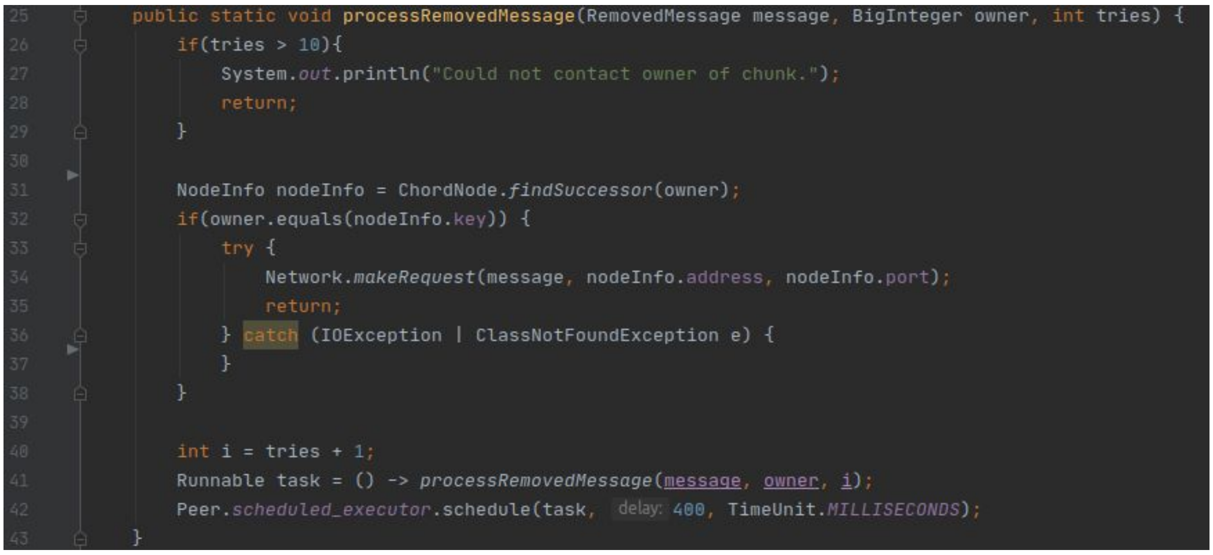


```java
public static void sendDelete(DeleteMessage delete, BigInteger key, int number_of_chunks, int tries) {
    if(tries >= 10){
        System.out.println("Couldn't delete all chunks of the file " + delete.getFileId());
        return;
    }

    NodeInfo nodeInfo = ChordNode.findSuccessor(key);
    if(nodeInfo.key.equals(key)) {
        try {
            DeleteReceivedMessage response = (DeleteReceivedMessage) Network.makeRequest(delete, nodeInfo.address, nodeInfo.port);
            receiveDeleteReceived(response, number_of_chunks);
            return;
        } catch (IOException | ClassNotFoundException e) {
            //the peer we trying to contact isn't available
        }
    }

    int n = tries + 1;
    Runnable task = ()->sendDelete(delete, key, number_of_chunks, n);
    Peer.scheduled_executor.schedule(task, (int)Math.pow(3, n), TimeUnit.SECONDS);
}
```

Figure 6: *sendDelete* method (DeleteProtocol class in project.protocols module)

Reclaim

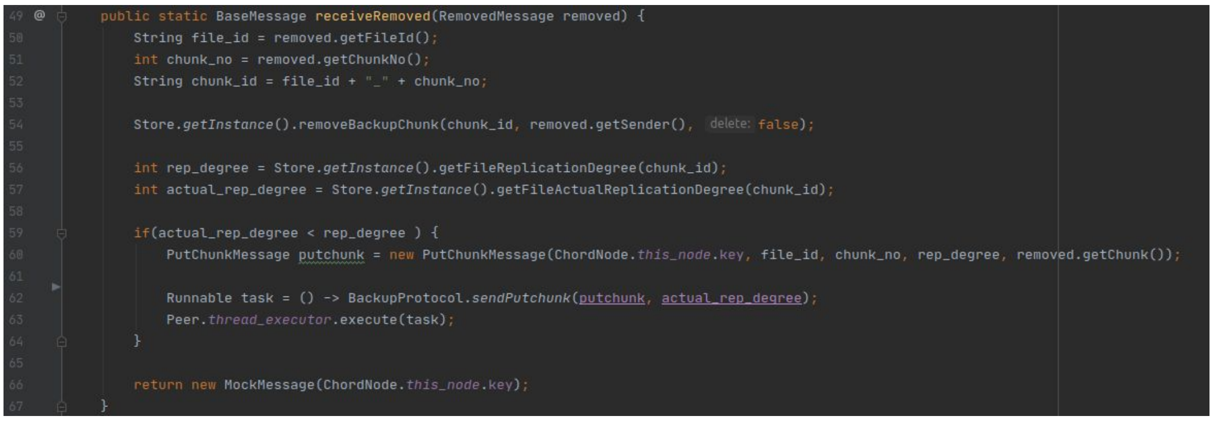
If a peer changes the space available to store other peer's chunks using the Reclaim protocol, two scenarios can happen: in the first scenario, the memory used to save copies is less or equal than the one total capacity allowed, and no other implications are needed; in a second scenario, this condition is not met and, furthermore, the peer selects chunks to eliminate in order to fulfill this requirement - the criteria chosen is deleting first the chunks with highest replication degree - and for each one sends a REMOVED message to the owner peer (who requested the backup of the chunk).



```java
public static void processRemovedMessage(RemovedMessage message, BigInteger owner, int tries) {
    if(tries > 10){
        System.out.println("Could not contact owner of chunk.");
        return;
    }

    NodeInfo nodeInfo = ChordNode.findSuccessor(owner);
    if(owner.equals(nodeInfo.key)) {
        try {
            Network.makeRequest(message, nodeInfo.address, nodeInfo.port);
            return;
        } catch (IOException | ClassNotFoundException e) {
        }
    }

    int i = tries + 1;
    Runnable task = () -> processRemovedMessage(message, owner, i);
    Peer.scheduled_executor.schedule(task,  delay: 400, TimeUnit.MILLISECONDS);
}
```

Figure 7: *processRemovedMessage* method (RestoreProtocol class in project.protocols module)

This last peer is responsible for his own files and for realizing their replication degree, therefore, after receiving the REMOVED, it must try to backup the chunk. The owner peer does not save its own chunks, therefore, just to simplify, REMOVED message also returns, in its body, the content of the chunk; this way, this peer only has to create a PUTCHUNK message using the returned body content and call the sendPutchunk method (from BackupProtocol class).



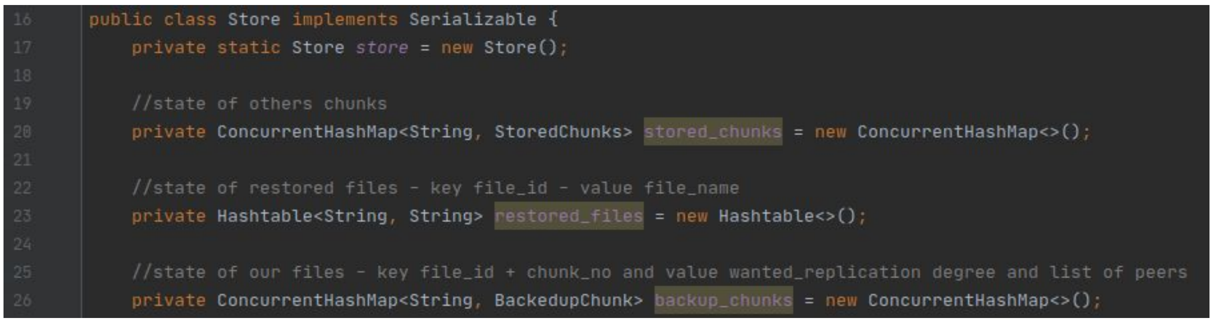
```java
public static BaseMessage receiveRemoved(RemovedMessage removed) {
    String file_id = removed.getFileId();
    int chunk_no = removed.getChunkNo();
    String chunk_id = file_id + "_" + chunk_no;

    Store.getInstance().removeBackupChunk(chunk_id, removed.getSender(),  delete: false);

    int rep_degree = Store.getInstance().getFileReplicationDegree(chunk_id);
    int actual_rep_degree = Store.getInstance().getFileActualReplicationDegree(chunk_id);

    if(actual_rep_degree < rep_degree ) {
        PutChunkMessage putchunk = new PutChunkMessage(ChordNode.this_node.key, file_id, chunk_no, rep_degree, removed.getChunk());

        Runnable task = () -> BackupProtocol.sendPutchunk(putchunk, actual_rep_degree);
        Peer.thread_executor.execute(task);
    }

    return new MockMessage(ChordNode.this_node.key);
}
```

Figure 8: *receiveRemoved* method (RestoreProtocol class in project.protocols module)

Important notes

It is relevant to mention two singleton serializable objects: Store and FilesListing; the first has three most important structures: *stored_chunks*, *backup_chunks,* and *restored_files*; being the first two *ConcurrentHashMap* since the purpose of the structures might cause extreme conflict while many threads try to access them. The first represents the chunks that were asked to be stored by other peers, each entry has the file id of the chunks as key and an object *StoredChunks* that hoards the key of the peer who owns the file, an array with the number of each chunk stored and the desired replication degree. The second structure (*backup_chunks*) is used to register the peers who have stored each chunk of our backed up files. The key is the chunk's id: file id + "_" + chunk number and the value is an object *BackedupChunk* that saves the file id of the chunk, the chunk's number, the desired replication degree and a list of keys (*BigInteger*) of all the peers that have stored the chunk. Finally, the *restored_files* keep a register of the files that have been asked for restore, by identifying the file's id and its name.

In regard to FilesListing class, this has only one ConcurrentHashMap: files, this accumulates the files that were backed up, having the file's name as the key and an object FileInfo as the value, this holds the file's name, its id, its path and the total number of chunks it is divided in.

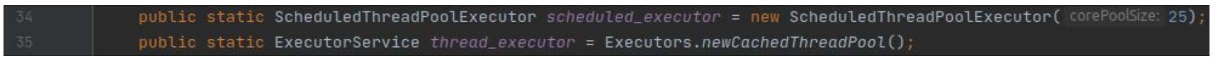


```java
public class Store implements Serializable {
    private static Store store = new Store();

    //state of others chunks
    private ConcurrentHashMap<String, StoredChunks> stored_chunks = new ConcurrentHashMap<>();

    //state of restored files - key file_id - value file_name
    private Hashtable<String, String> restored_files = new Hashtable<>();

    //state of our files - key file_id + chunk_no and value wanted_replication degree and list of peers
    private ConcurrentHashMap<String, BackedupChunk> backup_chunks = new ConcurrentHashMap<>();
```

Concurrency design

Concurrency is supported using two types of thread pools: ScheduledThreadPoolExecutor - necessary to maintain scheduled actions and delays in processes, not interfering with the correct functioning of other processes - and CachedThreadPoolExecutor - handles most percentage of created threads due to its ability to escalate and maintain a large number of active threads.

These two executors and declared and initialized in class Peer.



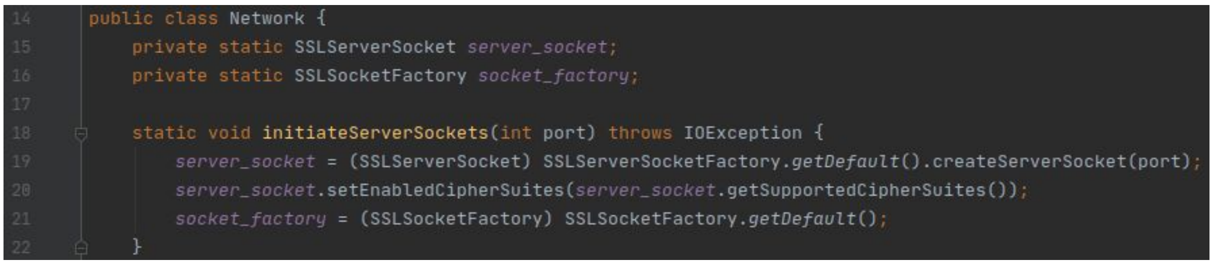
```java
    public static ScheduledThreadPoolExecutor scheduled_executor = new ScheduledThreadPoolExecutor( corePoolSize: 25);
    public static ExecutorService thread_executor = Executors.newCachedThreadPool();
```

Figure 10: Declaration and initialization of thread pool executors (in class Peer of module project.peer)

We also made sure to use structures meant to deal with thread concurrency, such as ConcurrentHashMap and CopyOnWriteArrayList, as well as some classes from java.nio to deal with file access concurrency (such as ByteBuffer, AsynchronousFileChannel, CompletionHandler, etc).

JSSE

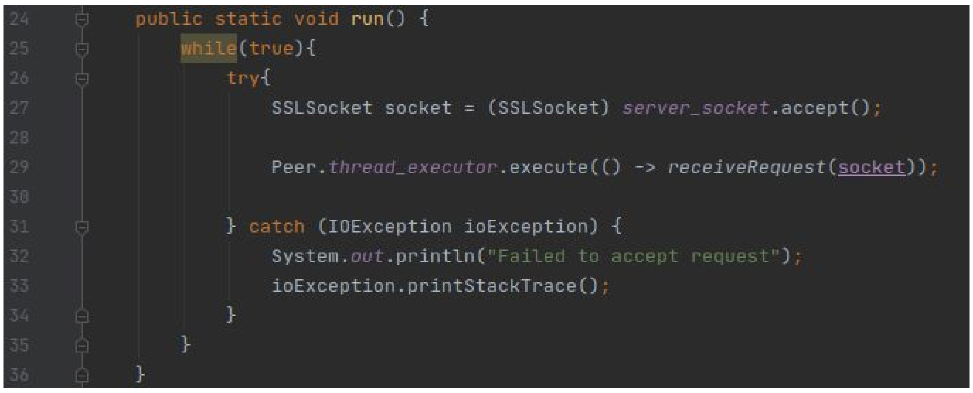
Declaration and initialization of JSSE utilities SSLServerSocket and SSLSocketFactory can be found in module Network in method *initiateServerSockets*.



```java
public class Network {
    private static SSLServerSocket server_socket;
    private static SSLSocketFactory socket_factory;

    static void initiateServerSockets(int port) throws IOException {
        server_socket = (SSLServerSocket) SSLServerSocketFactory.getDefault().createServerSocket(port);
        server_socket.setEnabledCipherSuites(server_socket.getSupportedCipherSuites());
        socket_factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
    }
```

Figure 11: Declaration and initialization of server socket and a socket factory (in class Network of module project.peer)

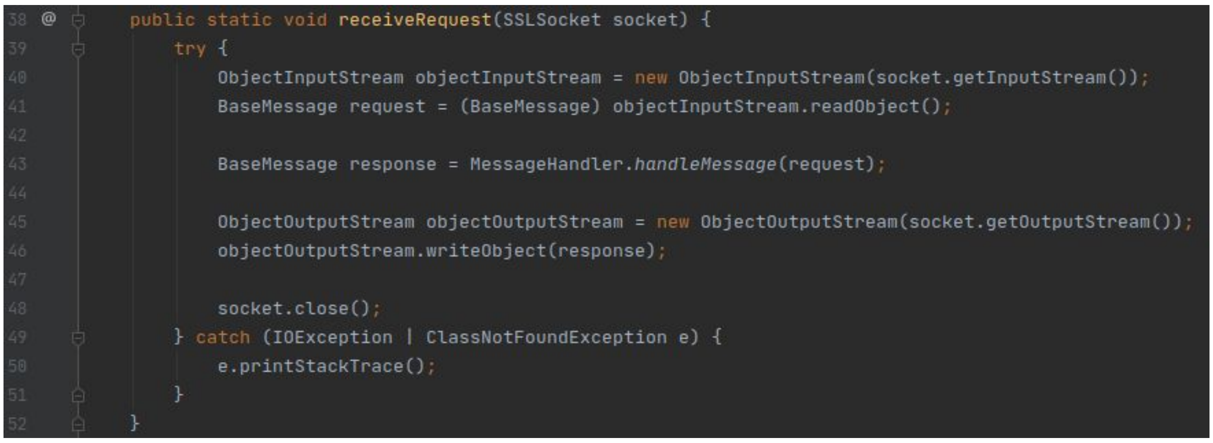
There is an infinite loop in *run* method, this is used to receive requests in the server_socket and, once accepted, execute the request's processing in another thread.



```java
    public static void run() {
        while(true){
            try{
                SSLSocket socket = (SSLSocket) server_socket.accept();

                Peer.thread_executor.execute(() -> receiveRequest(socket));

            } catch (IOException ioException) {
                System.out.println("Failed to accept request");
                ioException.printStackTrace();
            }
        }
    }
```

Figure 12: *run* method (in class Network of module project.peer)

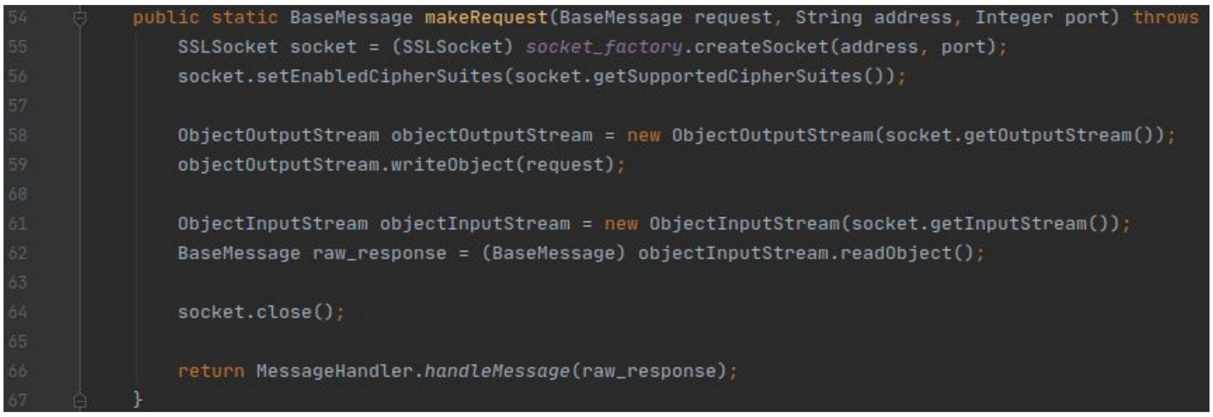
This request is processed in *receiveRequest* method, ObjectInputStream is used to read the request's information, this is then handled by *handleMessage* method (of MessageHandler in module project.message), which returns a response message - based on the request message type - to be delivered to the initiator peer using ObjectOutputStream.



```java
@   public static void receiveRequest(SSLSocket socket) {
        try {
            ObjectInputStream objectInputStream = new ObjectInputStream(socket.getInputStream());
            BaseMessage request = (BaseMessage) objectInputStream.readObject();

            BaseMessage response = MessageHandler.handleMessage(request);

            ObjectOutputStream objectOutputStream = new ObjectOutputStream(socket.getOutputStream());
            objectOutputStream.writeObject(response);

            socket.close();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
```

Figure 13: *receiveRequest* method (in class Network of module project.peer)

The *makeRequest* method is used to communicate with a designated peer through specific messages that elaborate on each request. This is done opening an SSLSocket (through the socket factory) nominating the destination peer's address and port, and proceeding to write, to that socket, the message, using ObjectOutputStream - meanwhile, the receiving peer must receive and deal with the request and return a response - and, finally, receive the response through ObjectInputStream and deal with it using *handleMessage* (from MessageHandler class).

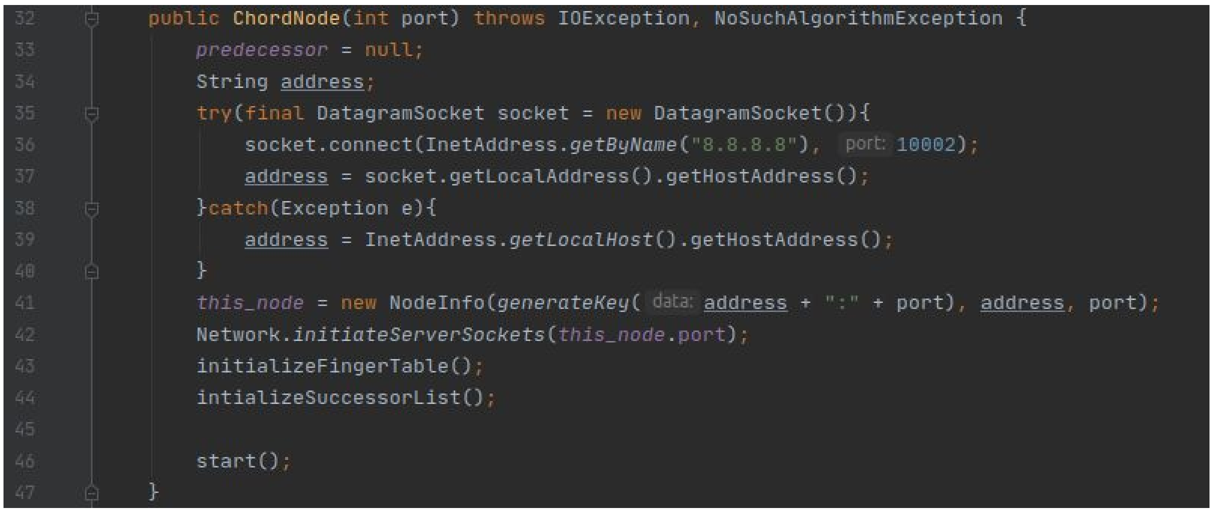


```java
    public static BaseMessage makeRequest(BaseMessage request, String address, Integer port) throws
        SSLSocket socket = (SSLSocket) socket_factory.createSocket(address, port);
        socket.setEnabledCipherSuites(socket.getSupportedCipherSuites());

        ObjectOutputStream objectOutputStream = new ObjectOutputStream(socket.getOutputStream());
        objectOutputStream.writeObject(request);

        ObjectInputStream objectInputStream = new ObjectInputStream(socket.getInputStream());
        BaseMessage raw_response = (BaseMessage) objectInputStream.readObject();

        socket.close();

        return MessageHandler.handleMessage(raw_response);
    }
```

Figure 14: *makeRequest* method (in class Network of module project.peer)

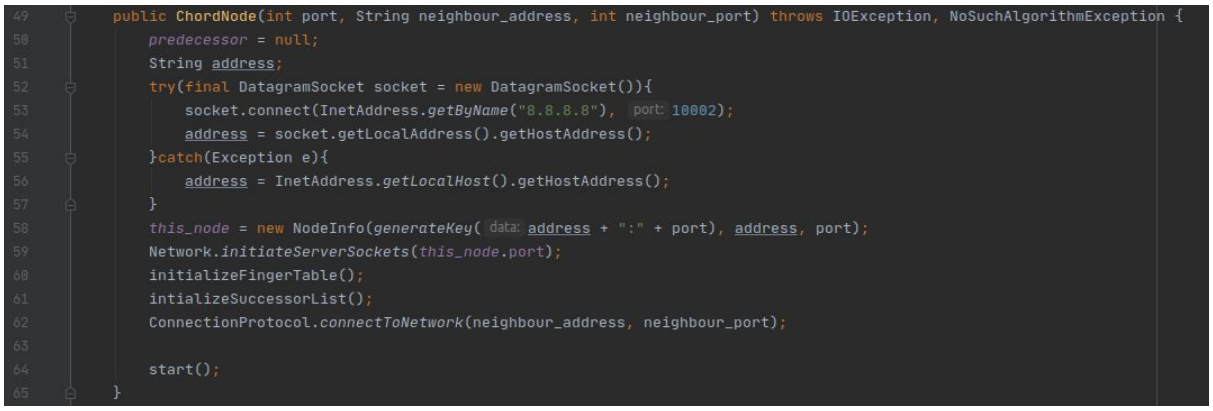
Scalability

In order to establish scalability at the design level, we implemented the Chord algorithm. This is described in ChordNode class in project.peer module. There are two constructors, one to initiate a new network (the peer who called it being the first in it) and another for when a new peer wants to join an already existing network.



```java
    public ChordNode(int port) throws IOException, NoSuchAlgorithmException {
        predecessor = null;
        String address;
        try(final DatagramSocket socket = new DatagramSocket()){
            socket.connect(InetAddress.getByName("8.8.8.8"),  port: 10002);
            address = socket.getLocalAddress().getHostAddress();
        }catch(Exception e){
            address = InetAddress.getLocalHost().getHostAddress();
        }
        this_node = new NodeInfo(generateKey( data: address + ":" + port), address, port);
        Network.initiateServerSockets(this_node.port);
        initializeFingerTable();
        intializeSuccessorList();

        start();
    }
```

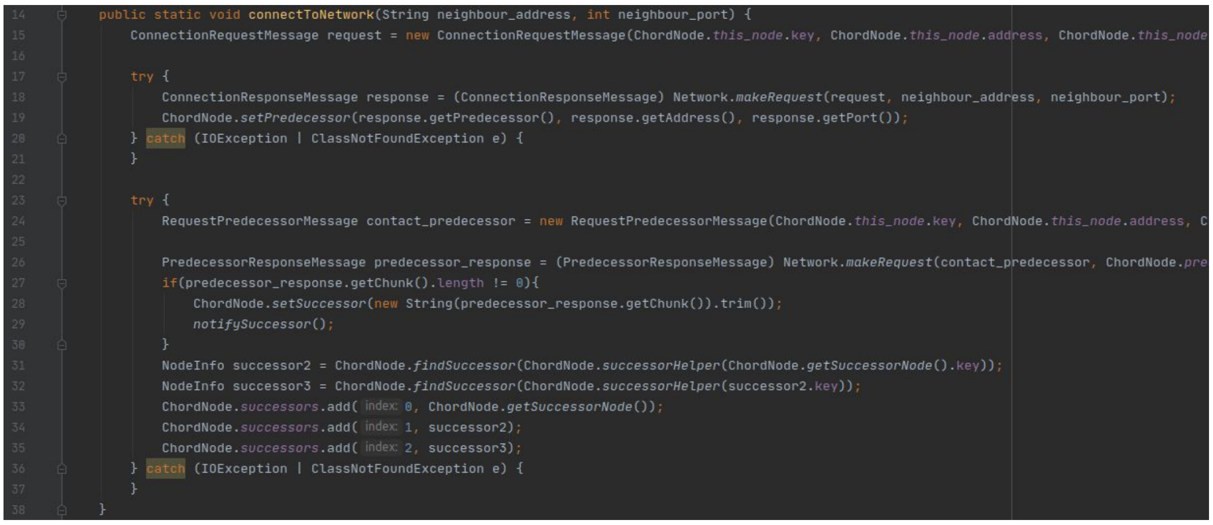
Figure 15: ChordNode constructor to initiate a new network   (in class ChordNode of module project.peer)



```java
    public ChordNode(int port, String neighbour_address, int neighbour_port) throws IOException, NoSuchAlgorithmException {
        predecessor = null;
        String address;
        try(final DatagramSocket socket = new DatagramSocket()){
            socket.connect(InetAddress.getByName("8.8.8.8"),  port: 10002);
            address = socket.getLocalAddress().getHostAddress();
        }catch(Exception e){
            address = InetAddress.getLocalHost().getHostAddress();
        }
        this_node = new NodeInfo(generateKey( data: address + ":" + port), address, port);
        Network.initiateServerSockets(this_node.port);
        initializeFingerTable();
        intializeSuccessorList();
        ConnectionProtocol.connectToNetwork(neighbour_address, neighbour_port);

        start();
    }
```

Figure 16: ChordNode constructor to join an existing network given the address and port of a peer of known to be part of it   (in class ChordNode of module project.peer)

The biggest difference between these two is the call to *connectToNetwork* method of ConnectionProtocol (class in project.protocols module) who will contact a peer by its address, request to join the network using a CONNECTION_REQUEST message and receive a CONNECTION_RESPONSE message which will respond with the key, address and port of this new peer's predecessor, then, contact the predecessor and obtain its successor (which will now be the new peer's successor) and contact him. This is a change we made in the original algorithm, who first requested the successor and then obtained its predecessor. We first thought it would be better, but we came to realize the original method might have been better and easier to implement, however, since no problems were resulting from it and time was running short, we did not get to change it.

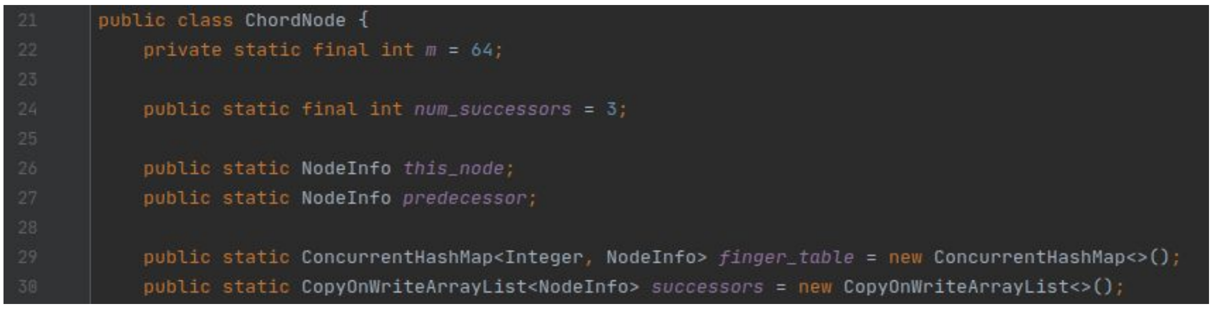


```java
public static void connectToNetwork(String neighbour_address, int neighbour_port) {
    ConnectionRequestMessage request = new ConnectionRequestMessage(ChordNode.this_node.key, ChordNode.this_node.address, ChordNode.this_node

    try {
        ConnectionResponseMessage response = (ConnectionResponseMessage) Network.makeRequest(request, neighbour_address, neighbour_port);
        ChordNode.setPredecessor(response.getPredecessor(), response.getAddress(), response.getPort());
    } catch (IOException | ClassNotFoundException e) {
    }

    try {
        RequestPredecessorMessage contact_predecessor = new RequestPredecessorMessage(ChordNode.this_node.key, ChordNode.this_node.address, C

        PredecessorResponseMessage predecessor_response = (PredecessorResponseMessage) Network.makeRequest(contact_predecessor, ChordNode.pre
        if(predecessor_response.getChunk().length != 0){
            ChordNode.setSuccessor(new String(predecessor_response.getChunk()).trim());
            notifySuccessor();
        }
        NodeInfo successor2 = ChordNode.findSuccessor(ChordNode.successorHelper(ChordNode.getSuccessorNode().key));
        NodeInfo successor3 = ChordNode.findSuccessor(ChordNode.successorHelper(successor2.key));
        ChordNode.successors.add( index: 0, ChordNode.getSuccessorNode());
        ChordNode.successors.add( index: 1, successor2);
        ChordNode.successors.add( index: 2, successor3);
    } catch (IOException | ClassNotFoundException e) {
    }
}
```

Figure 17: *connectToNetwork* method (in class ConnectionProtocol of module project.protocols)

In both constructors, the predecessor is initiated with null (as the algorithm dictates) and the address of the peer is obtained (is the address of the machine), we save the information about the current chord node in *this_node* using an object NodeInfo that contains the peer's key, address and port, initialize the server sockets in Network class and the *finger_table* and *successors* list. The key is generated by hashing with sha256 algorithm the string: address + ":" + port; it is always a BigInteger and can only go up to 2^{m-1}, *m* being a constant representing the number of bits of the key (according to the Chord algorithm), we initiate it as *m*=64.

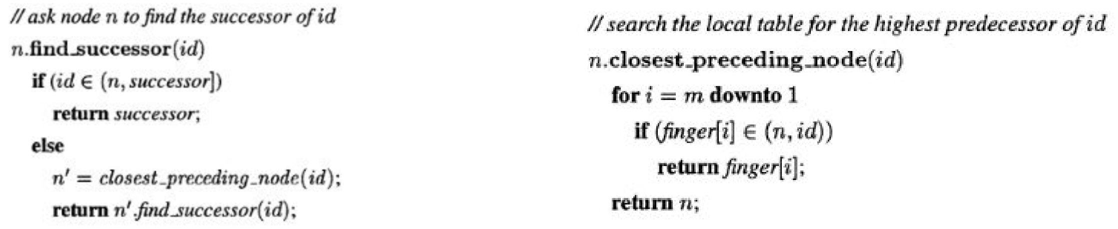
finger_table is a structure that, according to the algorithm, holds the *m* fingers of the node, which are the nodes whose key is equal or higher than: $key + 2^{i-1}$, where i is the entry number in the table (also the hashmap's key) and $i \in [1, m]$. *successors* is a list that hoards 3 immediate successors (the utility of the last will be explained in the Fault-tolerance module of the report).



```java
public class ChordNode {
    private static final int m = 64;

    public static final int num_successors = 3;

    public static NodeInfo this_node;
    public static NodeInfo predecessor;

    public static ConcurrentHashMap<Integer, NodeInfo> finger_table = new ConcurrentHashMap<>();
    public static CopyOnWriteArrayList<NodeInfo> successors = new CopyOnWriteArrayList<>();
```

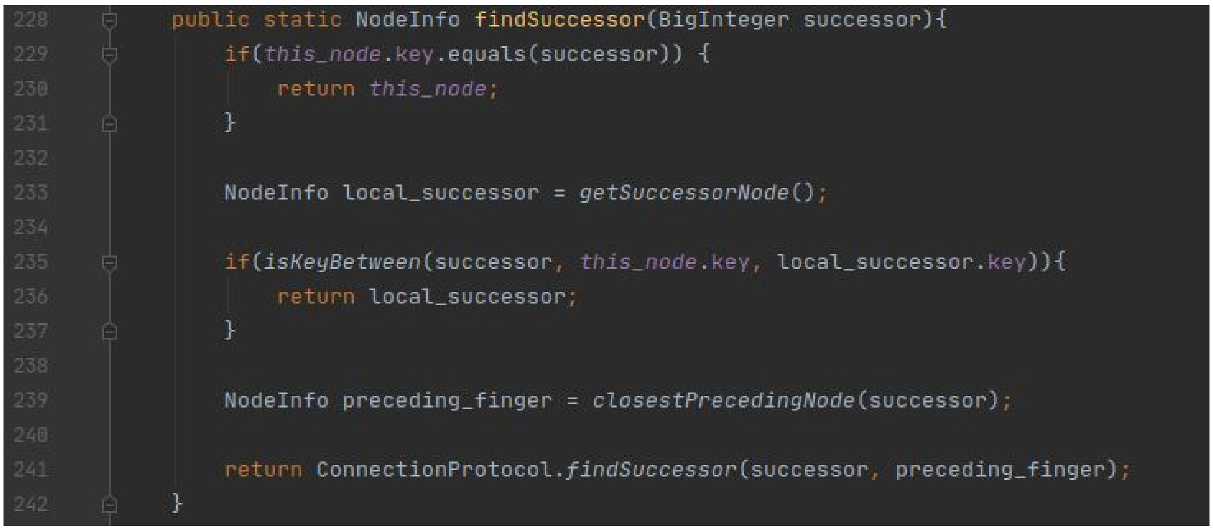
Figure 18: important structures and variables in ChordNode class (of module project.peer)

findSuccessor is another very important method of this class and it is built according to the algorithm's pseudocode:



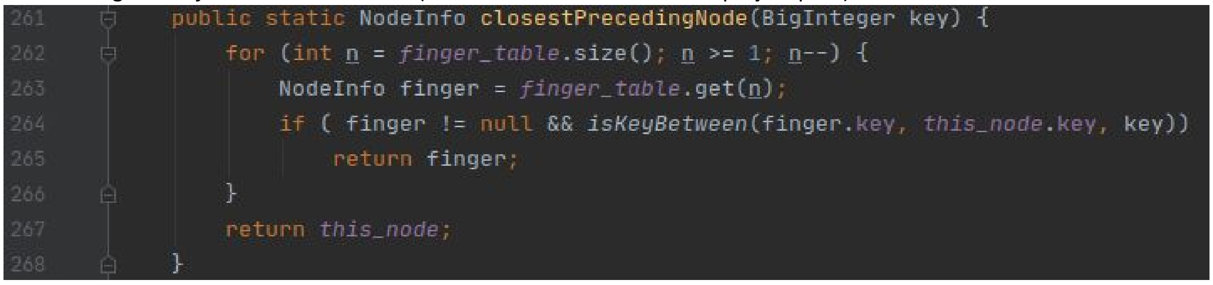
```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    n′ = closest_preceding_node(id);
    return n′.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i = m downto 1
    if (finger[i] ∈ (n, id))
      return finger[i];
  return n;
```

Figure 19: Pseudocode taken from: Chord: A Scalable Peer-to-Peer Lookup Protocol For Internet Applications; Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan, Member, IEEE



```java
public static NodeInfo findSuccessor(BigInteger successor){
    if(this_node.key.equals(successor)) {
        return this_node;
    }

    NodeInfo local_successor = getSuccessorNode();

    if(isKeyBetween(successor, this_node.key, local_successor.key)){
        return local_successor;
    }

    NodeInfo preceding_finger = closestPrecedingNode(successor);

    return ConnectionProtocol.findSuccessor(successor, preceding_finger);
}
```

Figure 20: *findSuccessor* method (in ChordNode class of module project.peer)



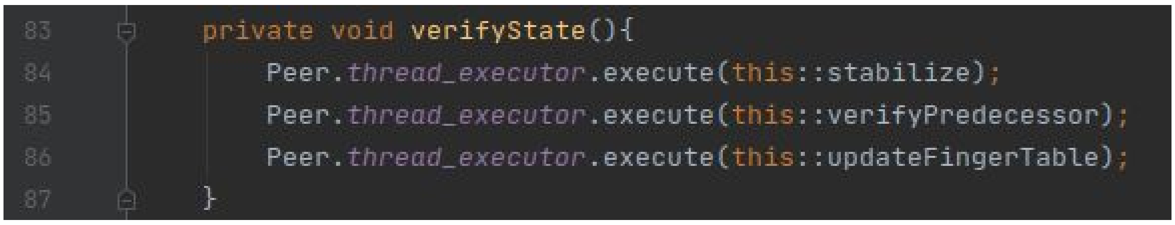
```java
public static NodeInfo closestPrecedingNode(BigInteger key) {
    for (int n = finger_table.size(); n >= 1; n--) {
        NodeInfo finger = finger_table.get(n);
        if ( finger != null && isKeyBetween(finger.key, this_node.key, key))
            return finger;
    }
    return this_node;
}
```

Figure 20: *closestPrecedingNode* method (in ChordNode class of module project.peer)

Fault-tolerance

Any system can have a fault, especially when a peer goes down; in order to ensure the stability of the program, it is necessary to deal with what happens to the peers in the network and to what happens to the chunks that were stored in that peer.

The first is solved by chord itself, it is necessary to update the state periodically, in accordance to the original Chord: first, verify immediate successor, he returns his predecessor and the peer decides if the this is his new successor or not, if so, notifies the new successor; update *finger_table* and *successors* list and consult his predecessor (if he is down, predecessor is set to null). The process is first call in the end of the constructor, in start method (line 69: Peer.*scheduled_executor*.scheduleAtFixedRate(this::verifyState, 3, 10, TimeUnit.*SECONDS*);).

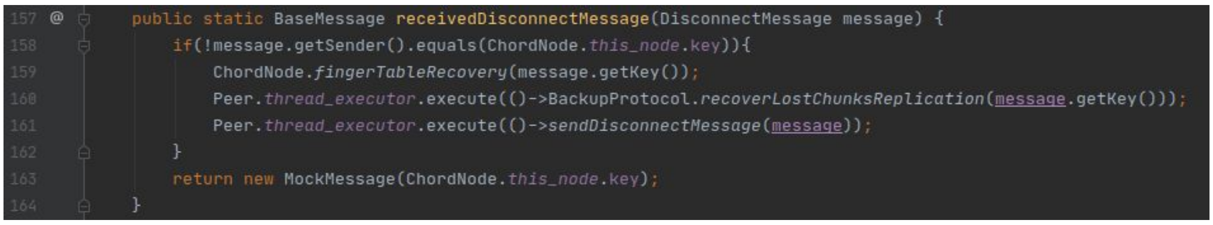


```java
private void verifyState(){
    Peer.thread_executor.execute(this::stabilize);
    Peer.thread_executor.execute(this::verifyPredecessor);
    Peer.thread_executor.execute(this::updateFingerTable);
}
```

Figure 20: *verifyState* method is periodically called with the goal of stabilizing and updating the state of the peer in the network as mentioned above (in ChordNode class of module project.peer)

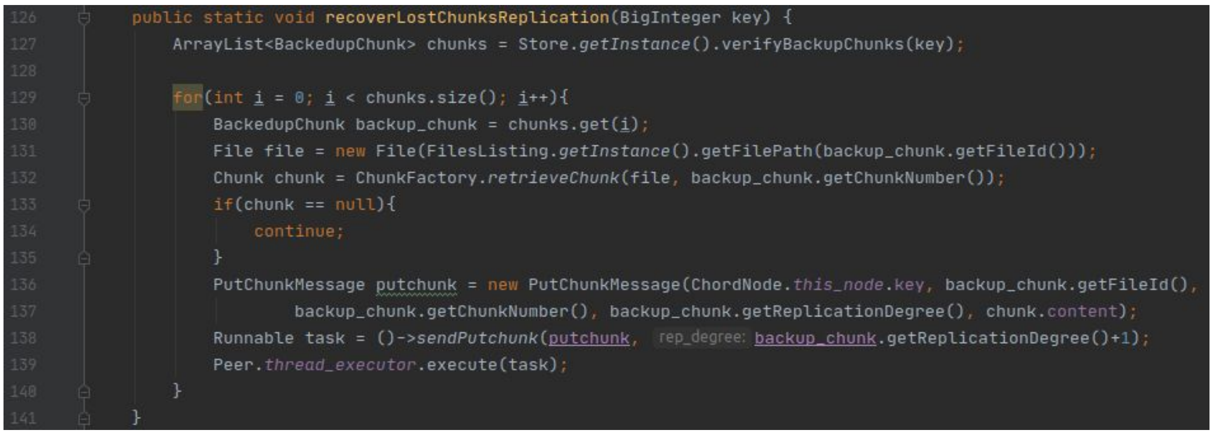
When a peer notices his successor is down, he recovers using the next immediate successor in the list (*successors*), then updates the *finger_table* and, finally sends a DISCONNECT message to his successor warning him about the peer that just went down; that node then sends it to his successor and so forward until it goes all around the ring and returns to the peer who initiated this chain reaction.

The second need mentioned in the first paragraph of this section is resolved when a peer receives this message, he will check his storage and assume the worst: that the peer who went down will not come back up; therefore, all chunks of own files that were stored on that peer will have to be replicated and kept in other peers as to ensure the replication degree.



```java
@   public static BaseMessage receivedDisconnectMessage(DisconnectMessage message) {
        if(!message.getSender().equals(ChordNode.this_node.key)){
            ChordNode.fingerTableRecovery(message.getKey());
            Peer.thread_executor.execute(()->BackupProtocol.recoverLostChunksReplication(message.getKey()));
            Peer.thread_executor.execute(()->sendDisconnectMessage(message));
        }
        return new MockMessage(ChordNode.this_node.key);
    }
```

Figure 21: receivedDisconnectMessage method (in ConnectionProtocol class in project.protocols module)

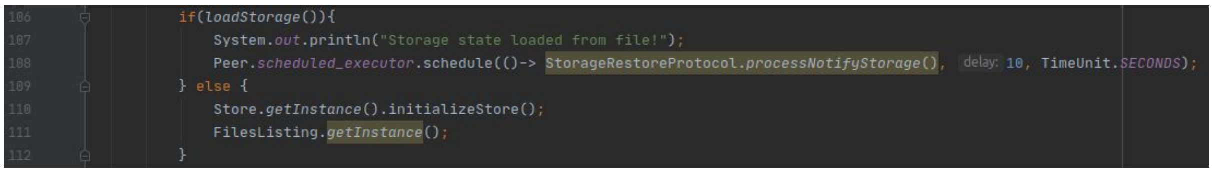


```java
    public static void recoverLostChunksReplication(BigInteger key) {
        ArrayList<BackedupChunk> chunks = Store.getInstance().verifyBackupChunks(key);

        for(int i = 0; i < chunks.size(); i++){
            BackedupChunk backup_chunk = chunks.get(i);
            File file = new File(FilesListing.getInstance().getFilePath(backup_chunk.getFileId()));
            Chunk chunk = ChunkFactory.retrieveChunk(file, backup_chunk.getChunkNumber());
            if(chunk == null){
                continue;
            }
            PutChunkMessage putchunk = new PutChunkMessage(ChordNode.this_node.key, backup_chunk.getFileId(),
                    backup_chunk.getChunkNumber(), backup_chunk.getReplicationDegree(), chunk.content);
            Runnable task = ()->sendPutchunk(putchunk,  rep_degree: backup_chunk.getReplicationDegree()+1);
            Peer.thread_executor.execute(task);
        }
    }
```

Figure 21: *recoverLostChunksReplication* method (in BackupProtocol class in project.protocols module)

Furthermore, if the peer does come back up to the network, it is necessary to make sure the file records are according to reality. To satisfy this, we created a new protocol, StorageRestoreProtocol.

The Store and FilesListing are serializable, therefore, when a peer connects to the network and has a folder with its id previously created, it assumes it as its, and retrieves the information it holds restoring the last state of the store (with all previously backed up and stored chunks as well as restored files). The protocol is called after loading the storage.



```java
        if(loadStorage()){
            System.out.println("Storage state loaded from file!");
            Peer.scheduled_executor.schedule(()-> StorageRestoreProtocol.processNotifyStorage(),  delay: 10, TimeUnit.SECONDS);
        } else {
            Store.getInstance().initializeStore();
            FilesListing.getInstance();
        }
```

Figure 21: protocol call after the storage is loaded in the main method of Peer class (in project.peer module)

It starts by sending a NOTIFY_STORAGE message for each set of chunks of the same file stored (each entry of *stored_chunks*) to the owner of that file, it then receives, as a reply, the chunks of that file that the owner wants him to keep in order for him to fulfill the replication degree.

Then it does the same but for his own backed up chunks, i.e., contacts the peers who stored the chunks of the files he requested for backup so as to guarantee they were not deleted while the peer was not available. Furthermore, there might be two scenarios: the replication degree is completed (no need for further action), the replication degree is not met (proceeds to backup the specific chunks by retrieving, from the original file, the exact chunks; if the file no longer exists, the registries regarding it are erased).

processNotifyStorage sends a message (using *sendNotifyStorage*) for each situation described before and *receiveNotifyStorageResponse* deals with the response, upon receiving it. The peer who receives the NOTIFY_STORAGE message deals with the request in *receiveNotifyStorage* and responds with a NOTIFY_STORAGE_RESPONSE message. All this process is declared in the same module, project.protocols.RestoreStorageProtocol.