U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Project 2 - Distributed Backup Service for the Internet

Distributed Systems, MIEIC
02/06/2021

André Filipe Meireles do Nascimento - up201806461
Gustavo Sena Mendes - up201806078
Luís Filipe Sousa Teixeira Recharte - up201806743
Rodrigo Campos Reis - up201806534

Turma 4 Grupo 22

Contents

Overview

In this project we developed a Peer-to-Peer distributed backup service for the Internet. The idea is to use the free disk space of the computers on the Internet for backing up files in one's own computer. The service supports the backup, restore and deletion of files. 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, thus a space reclaim operation is also supported.

Regarding the service design we opted to replicate full files instead of chunks as it seemed more reasonable in this project, and a totally distributed design using Chord, to locate a file’s copies.

When it comes to security we use JSSE for secure communication. Initially we started to develop the project with SSLEngine to achieve higher concurrency, but due to some difficulties encountered when we started implementing the protocols we switched to the more simple SSLSockets/SSLServerSockets interface.

To address scalability issues, Chord was used at a design level as stated above, and at an implementation level Java NIO for Files operations and Thread-pools to submit the protocol’s requests and to submit the tasks associated with the messages exchanged with the SSLSockets.

Fault tolerance was guaranteed with the Chord's fault-tolerant features and will be described in detail in its section.

The operations supported by the backup service are the follow:

- Backup - backs up a given file with the desired replication degree;
- Restore - restores a file that was previously backed up;
- Delete - deletes the backed up copies of the given file;
- Reclaim - sets the new storage max capacity (specified in KBytes), initiating a backup sub-protocol if needed after deleting file copies;
- State - shows the client information about the current peer: chord predecessor, routing table entries, successor list, storage capacity, backed up files and stored files copies.
- Shutdown - safely shutdowns the peer, sending its file copies to its successor

Project Structure

Our project root is divided into 6 folders:

- build/: Created after compiling the code;
- doc/: Contains the report and the readme file;
- files/: Contains files used in testing;
- scripts/: Contains bash scripts to help compile, test and execute the program in different operating systems;
- PeerStorage/: Created after starting at least one peer, will contain each peer folder (for example peer1) and within that folder the file copies that peer is backing up and the peer storage serializable file;
- src/: Contains the source code developed during this project.

Regarding the src/ folder, we organized it in packages, each of which are responsible for fulcral parts of the project:

- chord: Responsible for the participants of the P2P network;
- messages: Contains message classes used in peer communication;
    - chord: Contains the messages used in chord communication;
    - protocol: Contains the messages sent during the protocol execution;
- peer: Contains the main methods for starting protocols actions;
- sslsocket: Responsible for the P2P communication;
- storage: Contains classes responsible for information representation of files and node storage;
- tasks: Classes responsible for processing the different messages received by the peer;
    - chord: Classes responsible for processing the messages related to chord communication;
    - protocol: Classes responsible for processing the messages related to the different protocols;
- test: Responsible for sending the protocols to be executed to each peer using RMI (TestApp);
- utils: Contains useful methods and constants used by other packages.

Protocols

Our distributed backup service uses two different APIs: the client interface, that uses Java's RMI protocol, and the server interface, using a peer-to-peer protocol implemented using TCP with SSLSockets.

Client Interface Protocol

The communication between our Client and the Peer is made using Java's RMI API, through the TestApp, which provides a simple way of implementing RPC, taking care of service serialization and of the implementation of both client and server stubs.

Peer-to-Peer Protocol

To exchange messages between peers we implemented a simple TCP protocol with SSLSockets interface, that will be detailed in the JSSE section. Since we are using SSLSockets we took advantage of the ObjectInputStream and ObjectOutputStream that interface provides and we send Message objects directly to the sockets, this way we do not need to encode the messages and we have access to the Message object on the receiver side. Every Message contains the sender node reference, in other words, the sender GUID and InetSocketAddress, and each specific Message objects (classes that extend this, under the src.messages.chord and src.messages.protocol packages) contain the needed fields.

Chord Message Protocol

Join & Guid Messages

When a node wants to join the system, it sends a *JoinMessage* to one of the nodes connected to the network, ideally the boot peer, in order to receive its successor and its id. This information is sent in a *GuidMessage* as a response to the joining request.

Lookup & LookupReply Messages

In order to find a successor, our protocol uses *LookupMessages,* sent recursively through the network nodes with a given key, where each node waits for a response. When a message reaches its destination, the target node responds with a *LookupReplyMessage* containing its information (id, address and port).

GetPredecessor & Predecessor Messages

These messages are used when a node wants to ask its successor for the successor's predecessor. The node sends a *GetPredecessorMessage* and waits for a *PredecessorMessage* response containing the node information (id, address and port).

Notify Message

A node sends a *NotifyMessage* in order to tell its successor that it thinks that it might be its predecessor.

Check Message

A node sends a *CheckMessage* and waits for a response in order to see if its predecessor is still alive.

Successors & SuccessorsReply Messages

These messages are used periodically for a node to be able to keep its next successors updated. The node sends a SuccessorsMessage to its successor and waits for a SuccessorsReply response that contains the successors list of the node successor.

AlertPredecessor & AlertSuccessor Messages

These messages are used when a node wants to leave the ring. The node sends an *AlertPredecessorMessage* to its predecessor containing its successors list, afterwards, an *AlertSuccessorMessage* is sent to the successor containing the StorageFiles that the node is storing and its predecessor.

CopyKeys & CopyKeysReply Messages

These messages are used after a node joins the ring. The node sends a CopyKeysMessage to its successor signaling that he's online, and afterwards receives a *CopyKeysReplyMessage* as reply, containing the StorageFiles that should be stored in that node.

Backup Protocol

When a backup is started the initiator peer verifies if the indicated file was already backed up by the current peer, and if so the backup is canceled. Proceeding, the peer generates *4 * replication_degree* keys, which are random, distinct, and between 0 and the maximum number of nodes supported by the Chord ring, and will represent the file keys. The peer then iterates through these keys to find distinct peers that should be responsible for them, until the number of peers meets the desired replication degree or all keys have been iterated (meaning the replication degree will be below desired). Finally, a unique file id is generated, the file information and data are read, and afterwards a *BackupMessage* for each key is generated (with a *StorageFile* (structure that holds that file information such as the file id, original path, size, replication degree and the owner) and with the byte[] that contains the file data), and the respective *backupFile* runnable is submitted to the PeerScheduledThreadPoolExecutor, doing parallel backups.

In the submitted backupFile function, the backup initiator peer sends the respective *BackupMessage* to the respective target peer which will respond accordingly. When the target peer receives this message it will start a *BackupTask* to process it: if it already has a backup of this file or does not have enough storage space to store the file copy it will reply with an *ErrorMessage* with “HAVE” or “FULL” body respectively; otherwise it will store the file copy under its directory (./PeerStorage/peer{id}), writing the file data from the BackupMessage, saving a StorageFile object in its storedFiles Map, and finally replying with an OkayMessage. The initiator peer processes the target peer reply, adding the respective file key to its sent file storing keys in case it was an OkayMessage response, so it knows which keys are effectively being stored (the real replication degree).

Relevant code:

- src.peer.Peer, backup
- src.peer.Peer, backupFile
- src.storage.StorageFile
- src.storage.NodeStorage
- src.message.protocol.BackupMessage
- src.tasks.protocol.BackupTask

Restore Protocol

When a restore is started the initiator peer verifies if he has previously done a backup for that file, and if that's not the case the restore will be aborted. Proceeding, this peer will get the stored keys associated with that sent file (will get the respective StoredFile from its sentFiles map (src.storage.NodeStorage line 14)) and iterate through those keys, finding the current key's successor node which should be storing the wanted file, and start the restore process for that node (src.peer.Peer restoreFile function, line 328). If this occurs with success the restore will terminate and the file will be restored to its original path with a "restored_" prefix on the file name, otherwise the initiator peer will try to repeat this process on the other keys until the restore is completed successfully, or no more keys are available which means the restore failed.

In the *restoreFile* function the initiator peer will send a *GetFileMessage*, containing the id of the desired file to restore. The target peer after receiving this message will start a *GetFileTask* to process it: if this peer does not have the file it will reply with an *ErrorMessage* with "NOFILE" in the body; otherwise it will read the file copy data and reply with a *FileMessage* containing that data. In case the initiator peer receives a *FileMessage* it will write the data received to the respective restored file path, completing the restore process.

Relevant code:

- src.peer.Peer, restore
- src.peer.Peer, restoreFile
- src.storage.StorageFile
- src.storage.NodeStorage
- src.message.protocol.GetFileMessage
- src.message.protocol.FileMessage
- src.tasks.protocol.GetFileTask

Delete Protocol

When a delete is started the initiator peer verifies if he has previously done a backup for that file, and if that's not the case the delete will be aborted. Proceeding, this peer will get the stored keys associated with that sent file, looping through them all getting each key's respective successor, in other words, the peer that is storing the file with that key. Afterwards a *deleteFile* runnable is submitted to the PeerScheduledThreadPoolExecutor, doing parallel deletes for each of the keys.

In the *deleteFile* function the initiator peer will send a *DeleteMessage*, containing the id of the desired file to delete. The target peer after receiving this message will start a *DeleteTask* to process it: if this peer does not have the file or was unable to delete it, it will reply with an *ErrorMessage* with "NOFILE" or "NOTDELETED" in the body, respectively; otherwise it will have successfully deleted the file copy and reply with an *OkayMessage* containing the associated file key in the body. When the initiator peer receives the *OkayMessage* it will remove the incoming file key from the sent file's storing keys. When all deleted for that file finish, the initiator peer will check if those file's storing keys Set is empty, if so that means that all peers successfully deleted the file and that he can remove that StorageFile instance from his sentFiles Map.

Relevant code:

- src.peer.Peer, delete
- src.peer.Peer, deleteFile
- src.storage.StorageFile
- src.storage.NodeStorage
- src.message.protocol.DeleteMessage
- src.tasks.protocol.DeleteTask

Reclaim Protocol

In the reclaim protocol the Client specifies the new storage capacity in bytes. If the currently occupied storage space by copies of files is greater than the new desired capacity the peer will start the deletion of those copies until the occupied space is lower or equal to the new capacity. After deleting a file's copy the peer will send a *RemovedMessage*, containing the file id and key, to the owner of that file's copy, without waiting for a reply. After receiving the *RemovedMessage*, the file's owner peer will start a *RemoveTask*, first checking if he has that file in his sentFiles Map, ignoring the message if that's not that case, otherwise it will remove the received file key from the file's stored keys and start a sub backup protocol for that file if need (if the replication degree drops below desired).

Relevant code:

- src.peer.Peer, reclaim
- src.storage.StorageFile
- src.storage.NodeStorage
- src.message.protocol.RemovedMessage
- src.tasks.protocol.RemovedTask

Concurrency Design

Our application was developed with a multi-threaded design in order to have a fast and efficient implementation.

In order to handle concurrent service requests we use an ScheduledExecutorService that allows us to execute tasks in an scheduled way for an efficient split of the application workload. We implemented this interface in the Peer class, that has a scheduler for the protocol functions (for example, submitting a backupFile to the scheduler: src.peer.Peer), in the ChordNode class, that as a scheduler to execute the stabilization tasks periodically (src.chord.ChordNode, startPeriodicStabilize), and in the SSLSocketPeer class that has a scheduler to submit the received Message's task (src.sslsocket.SSLSocketPeer, run).

While using various threads to run the subprotocols, the same objects are accessed in different threads so they are stored using a ConcurrentHashMap in src.storage.NodeStorage, instead of an HashMap. This data structure is a thread-safe structure and is optimized for multi-threading. The space occupied in a node is stored in a variable on the node storage and is incremented or decremented using synchronized functions so it remains consistent throughout the node's "life".

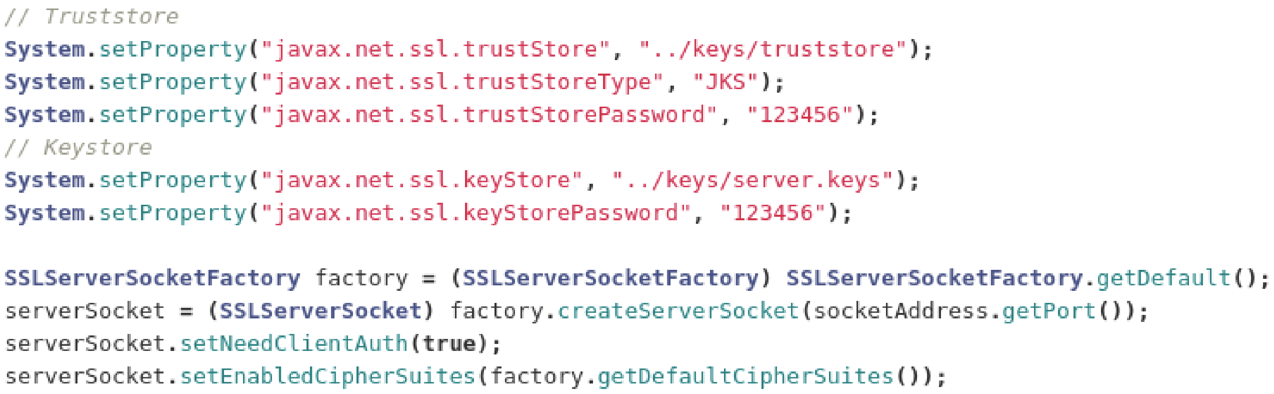
Regarding I/O operations with files we use Java NIO's functions to read, write and delete files (java.nio.file.Files package)

JSSE

As mentioned earlier, this project makes use of JSSE for secure communication between the peers. SSLEngine was used initially when implementing Chord and did not cause major issues that we were unable to fix, but when we started to implement the protocols we faced too many errors (such as ‘Tag mismatch’ errors) that would not allow us to continue implementing them consistently. This interface was the better choice, since it allows higher concurrency, but due to the time constraints and the need to implement the protocols correctly, we switched to the more simple SSLSockets/SSLServerSockets interface. This simpler interface proved much more easier to implement but limits the level of concurrency achievable.

Our SSLSocket implementation can be seen in the src.sslsocket.SSLSocketPeer class where both the server functionalities (the run method, that runs a loop to accept new connections to the server socket, and stop method to stop the loop and close the server socket) as well as the client side functionalities (the rest of the methods, that allow to create a new socket to a given address and send/receive messages from that connection) were implemented, making use of the SSLServerSocket, SSLServerSocketFactory, SSLSocket and SSLSocket Factory classes, all from javax.net.ssl package.

Regarding authentication the server requires user authentication, and the cypher suites files are located in the *keys* directory.



```
// Truststore
System.setProperty("javax.net.ssl.trustStore", "../keys/truststore");
System.setProperty("javax.net.ssl.trustStoreType", "JKS");
System.setProperty("javax.net.ssl.trustStorePassword", "123456");
// Keystore
System.setProperty("javax.net.ssl.keyStore", "../keys/server.keys");
System.setProperty("javax.net.ssl.keyStorePassword", "123456");

SSLServerSocketFactory factory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
serverSocket = (SSLServerSocket) factory.createServerSocket(socketAddress.getPort());
serverSocket.setNeedClientAuth(true);
serverSocket.setEnabledCipherSuites(factory.getDefaultCipherSuites());
```

Fig.1 - Server cipher suites.

Scalability

Since we wanted our Application to be a Distributed Scalable System, we implemented the Chord Protocol. Chord uses a variant of consistent hashing to assign keys to each peer and specifies how keys are assigned to nodes, and how a node can discover the value for a given key by first locating the node responsible for that key. A node will store the values for all the keys for which it is responsible.

This protocol is scalable since communication cost and the state maintained by each node scale logarithmically with the number of nodes. A Chord node requires information about O(log N) other nodes for efficient routing. Chord implements a faster search method by requiring each node to keep a *routing table* containing up to m nodes, where m is the number of bits in the key/node identifiers.

Chord Protocol implementation

Join

Whenever a new node wants to join the system a *JoinMessage* is sent to one of the nodes in the network so it can receive its successor and its id. After that, the new joining node sends a *CopyKeysMessage* to its successor in order to obtain the files that should be with him, then it receives the *CopyKeysReplyMessage* with the StorageFiles, and starts a restore protocol for each file.

Keys

In order to get a node's identifier we first concatenate the string representation of the node's IP address and Port number, calculate the SHA-1 hash of this string and apply the modulos of the maximum number of nodes in the network (2^m). We use SHA-1 as a base hash function, because it's expected to have good distributional properties.

Node

Each node is aware of its successor node on the ring. Since we implemented the Fault-tolerance features of Chord, each node contains information about its r nearest successors (more detail in the Fault-tolerance section). Besides that, every node knows who its predecessor is, and contains a routing table. All nodes joined to the ring run 3 tasks periodically in order to maintain the consistency of the protocol, *Stabilize*, *FixFingers* and *CheckPredecessor*.

We represent a node with a class named ChordNode.

Routing table

The i^{th}entry table of the node n will contain the information about the node with id corresponding to $successor((n + 2^{i-1})\ mod\ 2^m)$. The first entry of the table will always be the node's successor.

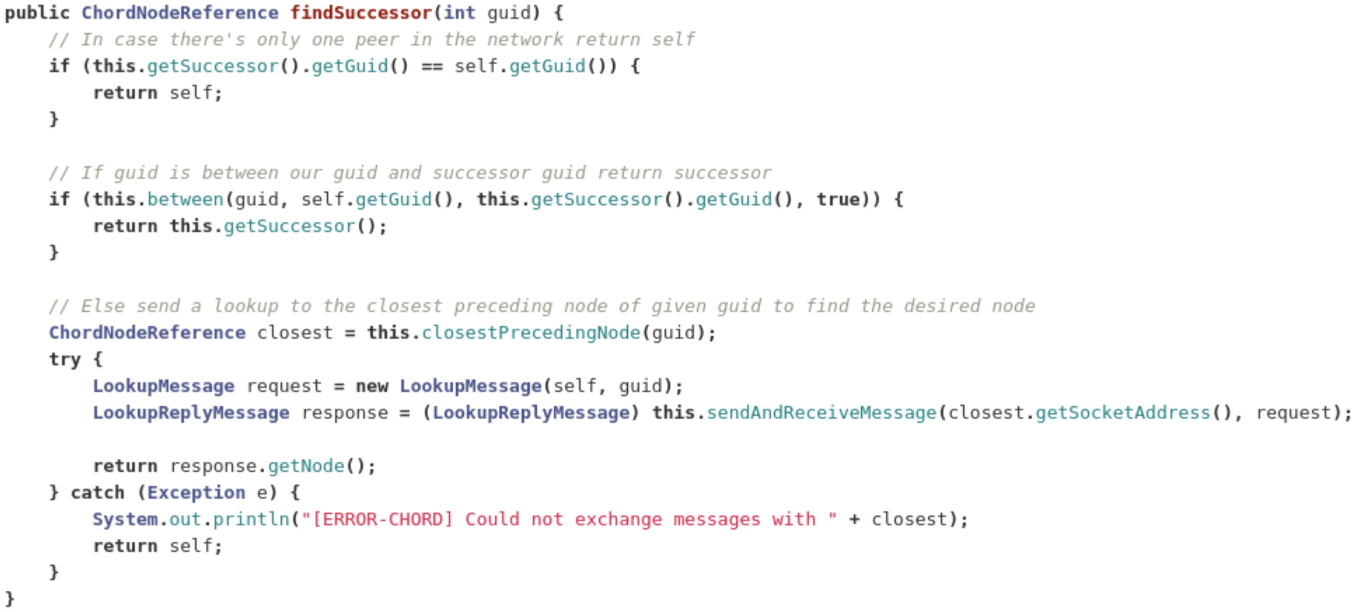
The entries of the table are updated periodically running the *FixFingers* method that calls *findSuccessor* method for each key generated by $(n + 2^{i-1})\ mod\ 2^m$.

Our routing table is represented by an array with *ChordNodeReference*s containing the id, address and port of each node.

Find Successor method

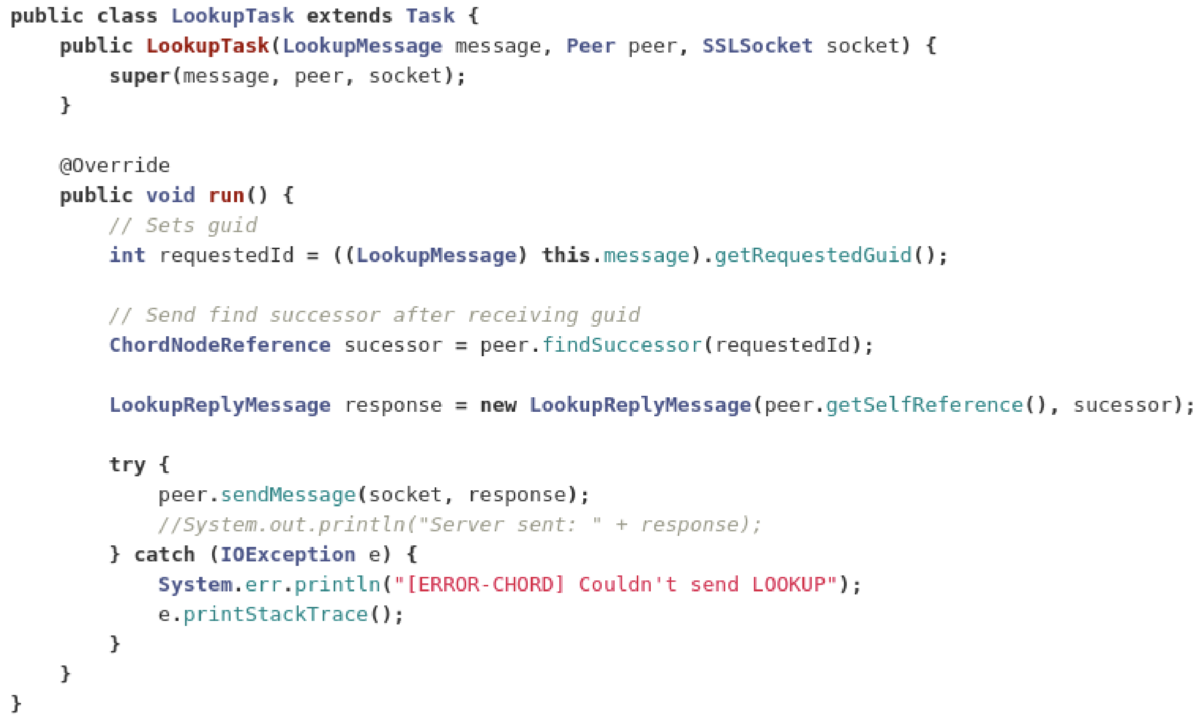
This is the most important algorithm of the protocol. First, we check if the successor’s id is equal to our id, this happens when there’s only one node in the network. Then we check if the key is equal to the id of the current node. If it is, then the current node is the successor.

Every time a node wants to lookup a key *k* and passes the steps below, it will pass the query to the closest successor of *k* found in the routing table, with a recursive strategy, sending a *LookupMessage* and waiting for a response. When receiving a *LookupMessage,* the node runs the task associated with that message, which calls *findSuccessor* with the key received in the message and waits again for a response. When the *LookupMessage* reaches the target, the *LookupReplyMessage* is sent as a response to the waiting nodes, until it reaches the initial node who called the *findSuccessor* method.



```java
public ChordNodeReference findSuccessor(int guid) {
    // In case there's only one peer in the network return self
    if (this.getSuccessor().getGuid() == self.getGuid()) {
        return self;
    }

    // If guid is between our guid and successor guid return successor
    if (this.between(guid, self.getGuid(), this.getSuccessor().getGuid(), true)) {
        return this.getSuccessor();
    }

    // Else send a lookup to the closest preceding node of given guid to find the desired node
    ChordNodeReference closest = this.closestPrecedingNode(guid);
    try {
        LookupMessage request = new LookupMessage(self, guid);
        LookupReplyMessage response = (LookupReplyMessage) this.sendAndReceiveMessage(closest.getSocketAddress(), request);

        return response.getNode();
    } catch (Exception e) {
        System.out.println("[ERROR-CHORD] Could not exchange messages with " + closest);
        return self;
    }
}
```

Fig.2 - implementation of find successor method sending a LookupMessage and waiting for response.



```java
public class LookupTask extends Task {
    public LookupTask(LookupMessage message, Peer peer, SSLSocket socket) {
        super(message, peer, socket);
    }

    @Override
    public void run() {
        // Sets guid
        int requestedId = ((LookupMessage) this.message).getRequestedGuid();

        // Send find successor after receiving guid
        ChordNodeReference sucessor = peer.findSuccessor(requestedId);

        LookupReplyMessage response = new LookupReplyMessage(peer.getSelfReference(), sucessor);

        try {
            peer.sendMessage(socket, response);
            //System.out.println("Server sent: " + response);
        } catch (IOException e) {
            System.err.println("[ERROR-CHORD] Couldn't send LOOKUP");
            e.printStackTrace();
        }
    }
}
```

Fig. 3 - Implementation of *LookupTask* when a node receives a *LookupMessage*.

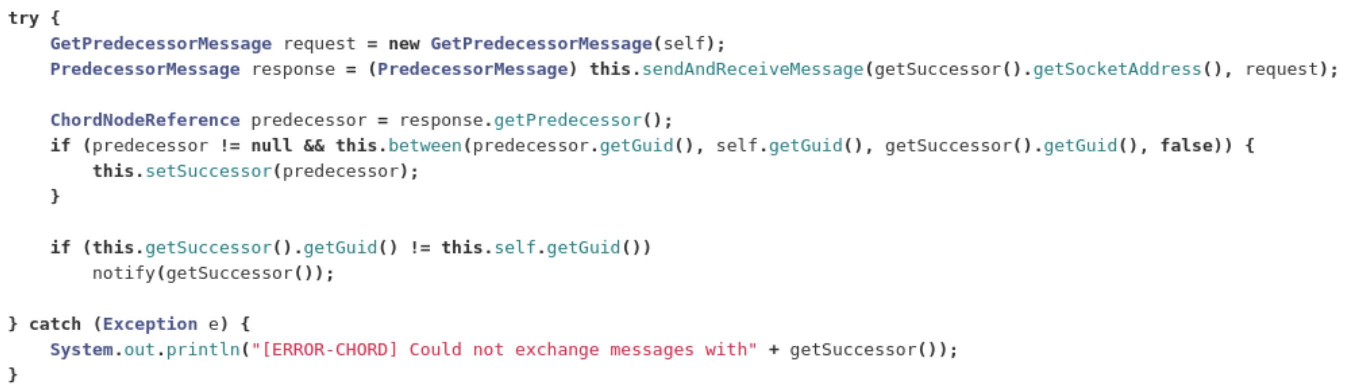
Stabilize method

To avoid making wrong lookups, each node's successor must be always up to date. This method is useful every time a node joins or leaves the ring, therefore, this method is called periodically.

To maintain the list of *r* successors up to date, we use this method (further explanation in Fault-Tolerance).

Each time a node runs *stabilize,* it sends a *PredecessorMessage* to its successor node asking for the successor's predecessor, and decides if the predecessor received in *PredecessorReplyMessage* should be our successor instead. This would be the case if a new node joined the system.

In addition, *stabilize* notifies its successor, giving its successor the chance to change its predecessor to be our node in case it is null.

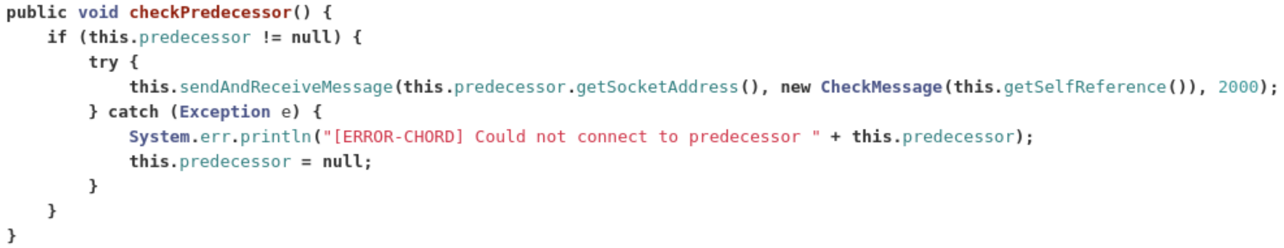


```java
try {
    GetPredecessorMessage request = new GetPredecessorMessage(self);
    PredecessorMessage response = (PredecessorMessage) this.sendAndReceiveMessage(getSuccessor().getSocketAddress(), request);

    ChordNodeReference predecessor = response.getPredecessor();
    if (predecessor != null && this.between(predecessor.getGuid(), self.getGuid(), getSuccessor().getGuid(), false)) {
        this.setSuccessor(predecessor);
    }

    if (this.getSuccessor().getGuid() != this.self.getGuid())
        notify(getSuccessor());

} catch (Exception e) {
    System.out.println("[ERROR-CHORD] Could not exchange messages with" + getSuccessor());
}
```

Fig. 4 - Stabilizing implementation on *stabilize* method.

Check Predecessor method

Finally, the last method running periodically is the *checkPredecessor* method that is used to clear its predecessor node in case it has failed, this allows it to accept a new predecessor in notify. This method sends a *CheckPredecessorMessage* to its predecessor, if it does not receive an *OkMessage* response within 2 minutes, it assumes that the predecessor is down and puts it to null.



```java
public void checkPredecessor() {
    if (this.predecessor != null) {
        try {
            this.sendAndReceiveMessage(this.predecessor.getSocketAddress(), new CheckMessage(this.getSelfReference()), 2000);
        } catch (Exception e) {
            System.err.println("[ERROR-CHORD] Could not connect to predecessor " + this.predecessor);
            this.predecessor = null;
        }
    }
}
```

Fig. 5 - Check predecessor implementation method.

Leave

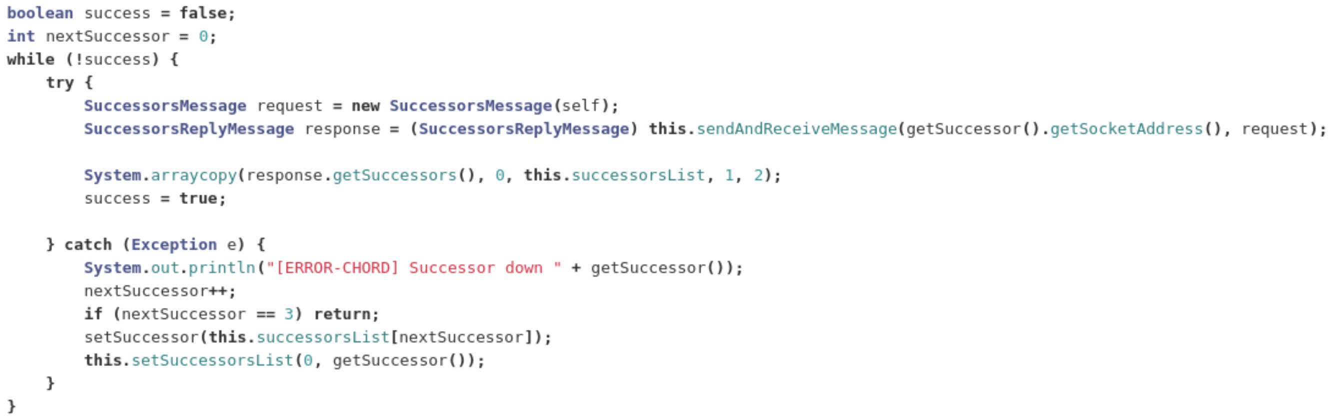
Whenever a new node wants to leave the system an *AlertPredecessorMessage* is sent to node’s predecessor with the successors list, so that the predecessor can update its successors list, and its own successor. Besides that, an *AlertSuccessorMessage* is sent to node’s successor with its predecessor so it can be updated too, also, a sub backup protocol is made in the *AlertSuccessorTask* in order to send its files to its successor. This way, the peer safely leaves the system. When this does not happen, the chord nodes stabilize with our implementation described in Fault-Tolerance.

To make a node leave the network we implemented a shutdown method to be called by the *TestApp*.

Fault-Tolerance

The correctness of the Chord protocol relies on the fact that each node knows its successor. However this invariant can be compromised if nodes fail. To increase robustness, each node in our protocol maintains a successors list of size r, containing the node’s first r successors. If a node’s immediate successor does not respond, the node can substitute its successor to the next entry in its successors list.

This is done in the stabilize method where each node sends a *SuccessorsMessage* to its successor and then waits for a response that contains the successors list of its successor. The successors list is built with the node successor in the first position, and then copying its successor's successor list, removing its last entry, and prepending it to its own list.



```
boolean success = false;
int nextSuccessor = 0;
while (!success) {
    try {
        SuccessorsMessage request = new SuccessorsMessage(self);
        SuccessorsReplyMessage response = (SuccessorsReplyMessage) this.sendAndReceiveMessage(getSuccessor().getSocketAddress(), request);

        System.arraycopy(response.getSuccessors(), 0, this.successorsList, 1, 2);
        success = true;

    } catch (Exception e) {
        System.out.println("[ERROR-CHORD] Successor down " + getSuccessor());
        nextSuccessor++;
        if (nextSuccessor == 3) return;
        setSuccessor(this.successorsList[nextSuccessor]);
        this.setSuccessorsList(0, getSuccessor());
    }
}
```

Fig. 3 - Fault tolerance loop implementation using successors list.