Distributed Backup Service for the Internet

SDIS - Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática e Computação (Maio de 2020)

Relatório Final

Daniel Brandão	up201705812
Henrique Santos	up201706898
Martim Silva	up201705205
Pedro Moás	up201705208

Overview

This project is a peer-to-peer distributed backup service for the internet. The main operations are the same as the first project: Backup, Restore, Delete and Reclaim. The project uses Thread Pools, nonblocking I/O with SSL Engine, Java NIO (asynchronous file channels, selectors, thread pools, socket channels and server socket channels), Chord and Chord's fault-tolerant features, so as to maximize Security, Concurrency, Scalability and Fault-tolerance.

Protocols

The communication protocol used in the project was TLS (TCP), using SSL Engine, so as to provide secure and nonblocking communication, except for the communication between a peer and the client testing application (TestApp) which used RMI, and contains the following operations:

```
public interface TestAppRemoteInterface extends Remote {
   void backupFile(String fileName, int repDeg) throws Exception;
   void restoreFile(String fileName) throws Exception;
   void deleteFile(String fileName)  throws Exception;
   void reclaimDiskSpace(int space)  throws Exception;
   PeerState getInternalState() throws RemoteException;
}
```

from TestAppRemoteInterface.java, lines 6-12

Besides the subprotocol-specific messages, the following messages are also defined:

- OK - Successful operation
- NOK - Generic failure response
- NO_RES - No resource, i.e the requested resource does not exist.
- IO_EX - An IO Exception happened
- DUPE - Duplicate, i.e the peer already contains that resource

Backup

For this subprotocol, the sent message has the following format:

`PUTCHUNK <fileId> <chunkNo> <owner> <replicaNo> <isRedirect> CRLF CRLF <body>`

By using the lookup operation from Chord, the peer will send the PUTCHUNK message to the node responsible for it, as well as its immediate successors, until the replication degree is met. The function that backs up a chunk on a given node is shown below.

```
public String backupOnNode(InetSocketAddress address, int replicaNo,
boolean isRedirect) {
   ...
   try (MySslEngineClient client = new
MySslEngineClient(TCPGroups.DB(address))) {
       Log("Successful connection to peer " + address);
       String response = client.writeRead(new PutChunkMessage(this.chunk,
this.ownerAddress, replicaNo, isRedirect).toString(), Message.MAX_SIZE);
       ...
       return response;
   }
   catch (IOException e) {
       Log("Couldn't store chunk " + chunk + " in peer " + address);
       Log("(Exception: " + e.getMessage() + ")");
   }
   return RawMessage.NOT_OK;
}
```

from PeerTCP.ChunkBackup.java, lines 76-91

When a peer receives a PUTCHUNK message and has space for the chunk, it will save it, responding with OK. Other possible responses are: DUPE, if it is already storing that chunk (but still considered a success message), NOK and IO_EX.

```
@Override
public Message processPutChunkMessage(PutChunkMessage message) {
      ...
   try {
       chunk.save(this.backupFolderPath);
   } catch (IOException e) {
       e.printStackTrace();
```

```
        Log("IO Exception: Could not save chunk: " + chunk);
        Log("Error: " + e.getMessage());
        return new RawMessage(RawMessage.IO_EXCEPTION);
    }
    this.storedLog.addChunk(chunk.getChunkID(), chunk.size(), message.owner,
message.replicaNo);

    return new RawMessage(RawMessage.OK);
}
```

from PeerTCP.MessageWorker.java, lines 113-152

Restore

For this subprotocol, the sent message has the following format:

`GETCHUNK <fileId> <chunkNo> CRLF CRLF`

In this case, the peer simply iterates through its list of storers for that chunk, sending the message to each of them, until one responds with CHUNK, which has the following format:

`CHUNK <fileId> <chunkNo> CRLF CRLF <body>`

```
public Chunk restoreChunk() throws ChunkException {
    List<InetSocketAddress> storers = this.peerState.getBackupLog().getChunkStorers(chunkID);

    for (InetSocketAddress node : storers) {
        Log("Trying to restore chunk from peer " + node);
        try (MySslEngineClient client = new MySslEngineClient(TCPGroups.DR(node))) {
            Log("Successful connection to peer " + node);
            String response = client.writeRead(new GetChunkMessage(this.chunkID).toString(),
Message.MAX_SIZE);
            Log("Tried to restore chunk from peer " + node + " and got response: " +
response.substring(0, Math.min(100, response.length())));

            if (response.equals(RawMessage.NO_RESOURCE))
                Log("Couldn't restore chunk from peer " + node + ": Claims he has no such
chunk");
            else {
                ChunkMessage chunkMessage = (ChunkMessage)
MessageParser.parseMessage(response);
                Log("Successfully retrieved chunk: " + chunkMessage.getChunk());
                return chunkMessage.getChunk();
            }
        }
```

```
        catch (IOException e) {
            Log("Failed to retrieve chunk from peer " + node);
        }
    }
    throw new ChunkException("Failed to retrieve chunk. Aborting...");
}
```

from PeerTCP.ChunkRestore.java, lines 28-51

When a peer receives a chunk message, it immediately sends the chunk back. However, if it does not have it in storage, it responds with NO_RES.

Delete

For this subprotocol, the sent message has the following format:

```
DELETE <fileId> CRLF CRLF
```

The delete subprotocol is rather straightforward, a peer first locates the fileID of the pathname it wishes to delete, then marks it for deletion, which means that it can no longer be restored. After retrieving all the peers that store at least one chunk of that file, it sends a DELETE message to each of them.

```
public void deleteFile(InetSocketAddress storer) {
    try (MySslEngineClient client = new
MySslEngineClient(TCPGroups.CC(storer))){
        String response = client.writeRead(new
DeleteFileMessage(fileID).toString(), Message.MAX_SIZE);
        Log("Tried to delete file " + this.fileID + " at " + storer + " and
got response " + response);
        if (response.equals(RawMessage.OK) ||
response.equals(RawMessage.NO_RESOURCE))
            peerState.getBackupLog().removeStorer(fileID, storer);
        else Log("Error deleting file " + this.fileID + ": Peer " + storer +
" responded with " + response);
    }
    catch (IOException e) {
        Log("Error deleting file from storer " + storer + " :" +
e.getMessage());
    }
}
```

From PeerTCP.FileDeletion.java, lines 24-35

When a peer receives a DELETE message, it simply deletes all the chunks belonging to that file, responding with OK.

```
@Override
public Message processDeleteFileMessage(DeleteFileMessage message) {
   if (!this.storedLog.hasFile(message.fileID))
       return new RawMessage(RawMessage.NO_RESOURCE);
   this.storedLog.removeFileRecords(message.fileID);

   File folder = new File(backupFolderPath, message.fileID);
   if (folder.exists()) {
       for (String chunkNo : folder.list()) {
           new File(folder.getPath(), chunkNo).delete();
       }
       folder.delete();
       return new RawMessage(RawMessage.OK);
   }
   return new RawMessage(RawMessage.NO_RESOURCE);
}
```

From PeerTCP.MessageWorker.java, lines 87-102

In order to make sure that peers don't store a chunk for a file that was previously deleted, the initiator peer will only "forget" about the file when all its recorded storers confirm with OK to the DELETE message. To complement that, each peer periodically runs the updateDeletions task, which notifies the peers who store a chunk of a file that's marked for deletion.

```
private void updateDeletions() {
   Log("Updating file deletions");
   for (WaitDeletionEntry deletionEntry :
this.peerState.getBackupLog().filesWaitingForDeletion()) {
       new FileDeletion(deletionEntry.getFileID(), this.localNode,
this.peerState).deleteFile(deletionEntry.getAddress());
   }
}
```

From Peer.java, lines 299-304

Reclaim

For the reclaim protocol, the peer finds the chunks it needs to free, starting by the largest ones. Using asynchronous file channels, calls freeChunk when the chunk is loaded, deleting it from the filesystem right after.

In order to keep the file replicated, the peer will redirect the chunk to another one, by sending a PUTCHUNK message and adding a redirect entry to its state.

```
private void freeChunk(Chunk chunk) {
   Node firstNode = this.localNode.getSuccessor();
   Node nextNode = firstNode;

   for (int attempts = 0; attempts < 10 && nextNode != null; attempts++) {
       boolean success = redirectToNode(chunk, nextNode);
       if (success)
           return;
       nextNode = LocalNode.cycleNode(nextNode, firstNode);
   }
   Log("Error in reclaim: Couldn't keep replication degree of chunk " +
chunk.getChunkID());
   redirectToNode(chunk, null);
   this.peerState.getStoredLog().deleteStoredChunk(chunk.getChunkID(),
this.backupFolderPath);
}
```

From Peer.java, lines 208-221

Periodically, a peer will check its redirect entries and notify the original owner, by sending a REDIRECT message:

REDIRECT <fileId> <chunkNo> <oldAddress> <newAddress> CRLF CRLF

```
private void notifyRedirects() {
   Log("Notifying of redirects");
   for (Map.Entry<ChunkID, RedirectEntry> mapEntry :
this.peerState.getRedirects().entrySet()) {
       ChunkID chunkID = mapEntry.getKey();
       InetSocketAddress ownerAddress = mapEntry.getValue().getOwner();
       InetSocketAddress newAddress = mapEntry.getValue().getAddress();
       Log("Notifying redirect of chunk " + chunkID + " to " + newAddress);

       try (MySslEngineClient client = new
```

```
MySslEngineClient(TCPGroups.CC(ownerAddress))) {
            String response = client.writeRead(new RedirectMessage(chunkID,
this.localNode.getAddress(), newAddress).toString(), Message.MAX_SIZE);
            if (response.equals(RawMessage.OK))
                this.peerState.clearRedirect(chunkID);
        } catch (IOException e) {
            Log("Failed to send redirect to " + ownerAddress);
        }
    }
}
```

From Peer.java, lines 281-297

Concurrency design

To ensure a concurrent implementation of our project, we opted for the use of Thread Pools (provided by the ***ExecutorService*** and ***ScheduledExecutorService*** classes from the java.util.concurrent package) whose threads are assigned tasks to run.

```
private static final ScheduledExecutorService threadPool =
Executors.newScheduledThreadPool(15);
    ...
threadPool.execute(new MessageWorker(message, localNode, peerState,
connection));
```

from PeerTCP.TCPServerThread.java, lines 22 and 56

Furthermore, to prevent cycles from blocking during their execution, we used scheduled tasks with a fixed delay, so that other tasks may run in the downtime of these loops.

```
updaterRedirects.scheduleAtFixedRate(this::notifyRedirects, 5000, 10000,
TimeUnit.MILLISECONDS);
```

from Peer.java, line 96

In order to ensure data cohesion during concurrent accesses, we used concurrent-safe data structures such as the ***ConcurrentHashMap*** and ***AtomicReferenceArray*** classes.

```
private final ConcurrentHashMap<ChunkID, StoredChunkEntry> chunkStoredMap =
new ConcurrentHashMap<>();
    ...
```

```
chunkStoredMap.put(chunkID, new StoredChunkEntry(chunkSize, owner,
replicaNo));
```

from StateLogging.StoredChunkLog.java, lines 21 and 32

```
private final AtomicReferenceArray<Node> fingerTable;
   ...
fingerTable.set(i, this);
```

from Chord.LocalNode.java, lines 31 and 217

Finally, so as to prevent blocking in I/O operations and allow concurrent access to files, the ***AsynchronousFileChannel*** class from java.nio was used in all file-related accesses, such as in the backup, restore and reclaim protocols. These were also delegated to the aforementioned thread pools.

```
AsynchronousFileChannel fileChannel =
AsynchronousFileChannel.open(Paths.get(this.peerFolderPath, filePath),
Collections.singleton(StandardOpenOption.WRITE), Chunk.chunkThreadPool);
   ...
fileChannel.write(buffer, chunkNo * 64000, buffer, new
CompletionHandler<Integer, ByteBuffer>() {
   ...
}
```

from Peer.java, lines 142 and 151

JSSE

We used SSL Engine for every communication on our project except, of course, for the *TestApp* class, which uses RMI. By using the SSL Engine class, we enabled secure communications between peers. Despite the SSLSockets class providing much of the same security functionality as SSL Engine, all of the inbound and outbound data is automatically transported using the underlying Socket class, which, by design, uses a blocking model, limiting the level of concurrency achievable. Because concurrency was a big priority in our project, it needed to use SSL Engine to solve this problem, since it is independent of the transport mechanism, meaning it could use nonblocking I/O channels.

The SSL Engine was configured to use Transport Layer Security (TLS) protocol, the server requires client authorization and default SSL Engine cipher-suites (TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384).

```
engine.setUseClientMode(false);
engine.setNeedClientAuth(true);

ServerSocketChannel serverSocket = ServerSocketChannel.open();
serverSocket.configureBlocking(false);
serverSocket.bind(new InetSocketAddress(hostname, port));
```

from SslEngine.MySslEngineServer.java, lines 21-26

```
KeyStore ksKeys = KeyStore.getInstance("JKS");
ksKeys.load(new FileInputStream(keystoreFilepath),
passphrasekeys.toCharArray());
KeyStore ksTrust = KeyStore.getInstance("JKS");
ksTrust.load(new FileInputStream(truststoreFilepath),
passphrasekeys.toCharArray());


// KeyManagers decide which key material to use
KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
kmf.init(ksKeys, passphrasetrust.toCharArray());

// TrustManagers decide whether to allow connections
TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
tmf.init(ksTrust);

// Get an instance of SSLContext for TLS protocols
sslContext = SSLContext.getInstance("TLS");
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), new
SecureRandom());
```

from SslEngine.MySslEnginePeer.java, lines 39-55

The transport layer is using Socket Channels, and the server is also using Server Channels. The server uses the Selector class to examine I/O events on multiple channels, and, with the event keys it creates, holds information about who is making the request and what type of the request is. If it is an acceptable event key, it will configure the client and do the handshake protocol, and if it is a readable event key, the server will use a thread from the thread pool to read, analyze, and sometimes respond depending on the ServerListener implementation.

```
public void selectKeys() throws IOException {
   selector.select();
   Set<SelectionKey> readyKeys = selector.selectedKeys();
   Iterator<SelectionKey> iterator = readyKeys.iterator();
   while (iterator.hasNext()) {
       SelectionKey key = iterator.next();
       iterator.remove();

       if (key.isValid() && key.isAcceptable()) {
           SocketChannel client = ((ServerSocketChannel)
key.channel()).accept();
           client.configureBlocking(false);
           SSLEngine engine = sslContext.createSSLEngine();
           engine.setUseClientMode(false);
           doHandshake(engine, client);
           client.register(selector, SelectionKey.OP_READ, engine);
       } else if (key.isValid() && key.isReadable()) {
           executor.execute(() -> serverListener.onReadReceived(new
MySslServerConnection(key, MySslEngineServer.this)));
       }
   }
}
```

from SslEngine.MySslEngineServer.java, lines 44-64

Scalability

➔ *'This can be at the design level, e.g. using Chord, or at the implementation level, e.g. using thread-pools and asynchronous I/O, i.e. Java NIO. (If you use Chord and Java NIO with thread-pools, your ceiling will raise by 4 points)'*

Besides having used Java NIO and thread pools, as described before, we also decided to use the Chord protocol to ensure scalability. By keeping information about only a few nodes (around log(N), if N represents the amount of nodes in the network), and with the automatic finger table adjustments (when a node joins or leaves the ring), the problem of scalability is solved.

Chord is scalable in a dynamic system. It uses a modified version of consistent hashing and does not require that all nodes be aware of all other nodes in the system, and thus can cut down the amount of data a node is required to store, and how many

messages must be sent to locate data. Chord is, therefore, scalable, because the cost of look-up grows as the logarithm of the number of nodes.

The Chord protocol needs to deal with nodes joining on the system and with nodes failing or leaving voluntarily. This is done through the steps described below. To ensure the lookups are executed correctly, the Chord protocol must ensure that all the information a node store is up to date. This is done on the Stabilization protocol.

Stabilization

We have to implement 3 concurrent and periodic tasks. This is done through the **stabilization protocol,** that each node runs periodically in the background, which updates Chord's finger tables and successor pointers.

The first periodic task has the objective to learn about newly joined nodes. Each time a node runs *stabilize*, it asks its successor for the successor's predecessor, and decides whether it should be the node's successor. It also notifies the new successor of his predecessor (the node itself).

The second task (running *fixFingers*) updates Chord's finger tables and successor pointers, if the finger tables entries are not correct. This is how new nodes initialize their finger table and how new existing nodes add new nodes to their finger tables.

The third, and final, task (running *checkPredecessor*) clears a node's predecessor if it has failed. This allows that the node can set a new predecessor.

```
/**
* Called periodically, verifies node immediate successor, and tells the
successor about this node */
public void stabilize() {
   if (!this.joined) return;

   Node notify = this.getSuccessor().getPredecessor();
   if (notify.getID().isBetween(this.getID(), this.getSuccessor().getID()))
{
       this.setSuccessor(notify);
   }
   if (!notify.getID().equals(this.id))
       this.getSuccessor().setPredecessor(this);
}
/**
* Called periodically, refreshes finger table entries
```

```
*/
private void fixFingers() {
   if (!this.joined) return;
   System.out.println("UPDATING node " + this);

   for(int i = 1; i < fingerTable.length(); i++) {
       ChordID toLookup = this.getID().add2Pow(i);
       Node successor = this.getSuccessor();
       if (toLookup.isBetween(this.getID(), successor.getID()))
           fingerTable.set(i, successor);
       else
           fingerTable.set(i, successor.lookup(toLookup));
   }
}
/**
* Called periodically, checks if whether predecessor has failed
*/
public void checkPredecessor() {
   if (!this.joined) return;

   if (!predecessor.ping()) {
       setPredecessor(this.otherPredecessor);
   } else this.otherPredecessor = predecessor.getPredecessor();
}
```

from SslEngine.MySslEnginePeer.java, lines 122-161

Node Joining

```
/**
* Join a Chord Ring containing node other
* @param other
*/
public void join(Node other) {
   if (this.joined)
       throw new RuntimeException("Node " + this.getID() + " is already in
a ring.");

   this.joined = true;

   if (other != null) {
       Node successor = other.lookup(getID().add2Pow(0));
```

```
        this.setSuccessor(successor.setPredecessor(this));
    }


Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(this::stab
ilize, 4000, 4000, TimeUnit.MILLISECONDS);

Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(this::fixF
ingers, 4000, 4000, TimeUnit.MILLISECONDS);

Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(this::chec
kPredecessor, 4000, 4000, TimeUnit.MILLISECONDS);

Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(this::chec
kSuccessor, 4000, 4000, TimeUnit.MILLISECONDS);
    Executors.newSingleThreadScheduledExecutor().execute(new
LocalNodeServer(this));
}
```

from Chord.LocalNode.java, lines 101-117

When a node joins a Chord Ring, it has to set its successor, and through the **stabilization protocol**, it can construct the information about its finger table, predecessor and successor.

Lookup

The DHT provides a single operation: lookup(key) returns the address of the node responsible for the key (in our case, returns a Node object that contains an address). Because each node maintains the routing information for about O(log|N|) other nodes, this operation can be resolved via O(log|N|) messages to other nodes (the process of exchanging messages with other does is explained in the next topic).

Upon receiving a key, the first thing the lookup does is to check if the key is between the node and its successor. If so, then the successor is returned immediately. Otherwise, another lookup is necessary (forwardLookup).

```
public Node lookup(ChordID key) {
  if (this.getID().equals(key)) return this;

  Node successor = getSuccessor();
  if (key.isBetween(this.getID(), successor.getID()) ||
```

```
key.equals(successor.getID()))
        return successor;

    return forwardLookup(key);
}
```

from Chord.LocalNode.java, lines 79-87

After determining that another lookup is necessary, a node shall go through its finger table (starting at the end, leading to quicker results), and find the node that is between the initial node and the node successor of the key. This will forward the lookup request to another node and wait for its response. If no node is able to fulfill the requirements, then the successor is not found, and an exception is thrown.

```
private Node forwardLookup(ChordID key) {
    for(int i = fingerTable.length() - 1; i >= 0; i--)
        if (fingerTable.get(i).getID().isBetween(this.getID(), key))
            return fingerTable.get(i).lookup(key);

    throw new RuntimeException("No successor found...");
}
```

from Chord.LocalNode.java, lines 89-95

Communication between nodes

The nodes communicate via exchanging messages. We have 4 types of messages, which all extend *ChordMessage* . These are called *SetPredecessorMessage*, *GetSuccessorMessage*, *GetPredecessorMessage* and *LookupMessage*, which represent simple tasks that must occur in order for Chord to fully operate.

```
public abstract class ChordMessage {
    public static final int MAX_SIZE = 1024;

    public final InetSocketAddress sender;

    public ChordMessage(InetSocketAddress sender){
        this.sender = sender;
    }

    public abstract Node process(Node processor);
```

```
}
```

from Chord.Message.ChordMessage.java, lines 7-17

We structured our program so that each physical *Node* is represented as a *LocalNode* object, and every reference it contains to any other node is represented as a *RemoteNode* object (predecessor, finger table elements, etc.).

That way, every time a *Node* wants to communicate with another, it uses the *RemoteNode* abstraction, which contains every method that a *Node* should have (*getSuccessor*(), *lookup*(), etc.), but they are simply wrappers for the *sendMessage* method, which performs the actual communication with an object of the class *LocalNodeServer*.

As shown below, it can be seen as an application of the RMI concept.

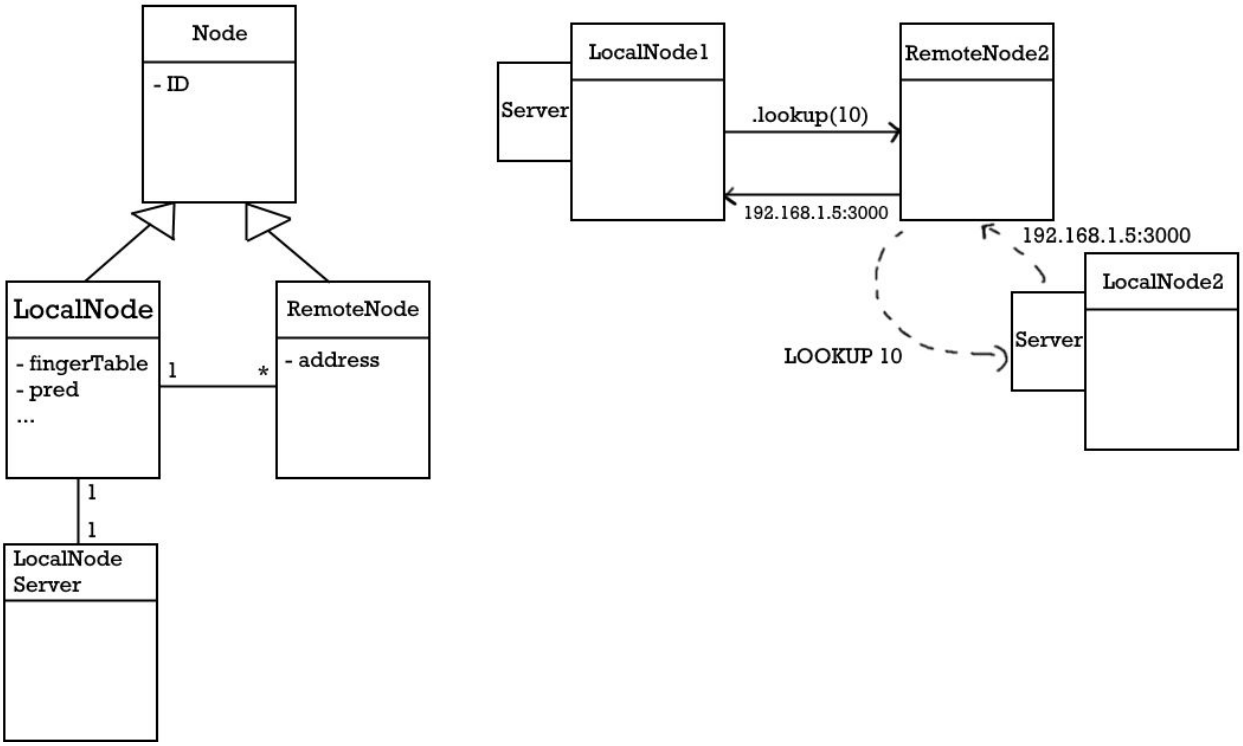

Fig. 1: Diagram representation of the Chord class design, as well as a communication example.

The sendMessage method implementation can be seen below:

```
private Node sendMessage(ChordMessage message) {
```

```
    try (MySslEngineClient client = new MySslEngineClient(address)) {
        client.write(message.toString());
        client.shutdownOutput();
        String response = client.read(ChordMessage.MAX_SIZE);
        return new RemoteNode(ChordMessageParser.parseAddress(response),
this.ownerAddress);
    } catch (IOException e) {
        System.err.println("Error sending message on remote node " +
this.getID() + ": " + e.getMessage());
        return this;
    }
}
```

from Chord.RemoteNode.java, lines 68-78

When the *LocalNodeServer* gets a connection, the SSLEngine interface calls the *onReadReceived* method, which parses and processes the received message.

```
@Override
public void onReadReceived(MySslServerConnection connection) {
    try {
        String message = connection.read(ChordMessage.MAX_SIZE);
        if (message == null)
            return;
        //Log("Received message: "  + message);
        ChordMessage chordMessage =
ChordMessageParser.parseMessage(message);
        if (chordMessage instanceof NullChordMessage)
            throw new IOException("Invalid message: " + chordMessage);
        Node toReturn = chordMessage.process(node);
        connection.write(toReturn.getAddress().toString());
        connection.shutdownOutput();
    } catch (IOException e) {
        connection.close();
    }
}
```

from Chord.LocalNodeServer.java, lines 26-42

Java NIO

To enhance scalability of the service, several classes from the java.nio package were used. For instance, the ***AsynchronousFileChannel*** class was employed in most I/O

file operations to minimize I/O block downtime (see the *Concurrency* section for an example).

The implemented thread pools, from the **ExecutorService** and ***ScheduledExecutorService*** *classes*, also guarantee that there is a fixed number of threads running at all times, preventing exponential growth of the total thread count. In addition to that, since thread pools keep a fixed number of threads initialized, the overhead of new thread initialization is completely avoided.

By using the ***SocketChannel*** and ***ServerSocketChannel*** classes from this package, instead of direct reads from the sockets, we prevent blocking in I/O from connections and allow this wait time to be used for other operations.

Finally, java.nio ***Selectors*** were employed in order to monitor several channels without the need to employ a different thread for each of these.

Fault-tolerance

As it was described in the assignment description:

➔ *'The goal is to avoid single-points of failure. E.g. if you choose a centralized server you can replicate it and use, e.g., Paxos or just plain primary-backup. If you choose a decentralized design, you can implement Chord's fault-tolerant features.'*

Chord Implementation

In the Chord's fault tolerance protocol, rather than just storing a single successor, a node keeps a list of successors of size r. Choosing r = 2log(N) suffices to maintain lookup correctness. Because the value we chose for N is 8, then 2log(N) = 1.8, therefore, r = 2. We will also save 2 predecessors to speed the process up.

If a successor fails, a node can replace it with the next one from its list, then update its list of successors. To check if a successor's node fails, we have a method on the Remote Node called ping that tries to connect with the successor. The same thing is done regarding the predecessor.

```
@Override
public boolean ping() {
   if (ownerID.equals(this.id))
       return true;
   try(MySslEngineClient ignored = new MySslEngineClient(address)) {} catch
```

```
(IOException e) {
        return false;
    }
    return true;
}
```

from Chord.RemoteNode.java, lines 58-66

Since r = 2, we just need to store another successor and other predecessor. If a successor is not alive, then we have to replace it with the other successor. After replacing a successor, we also have to replace the other successor we are keeping. The *checkSuccessor* method is called periodically. The same thing applies to the predecessor and the other predecessor we store. The method *checkPredecessor* is called periodically also. Therefore, the ping method is not also responsible to check if a successor is still alive, but is also responsible to check if the predecessor is still alive.

```
/**
* Called periodically, checks if whether predecessor has failed
*/
public void checkPredecessor() {
    if (!this.joined) return;

    if (!predecessor.ping()) {
        setPredecessor(this.otherPredecessor);
    } else this.otherPredecessor = predecessor.getPredecessor();
}

/**
* Chord's fault tolerance
* @return
*/
public void checkSuccessor() {
    if (!this.joined) return;
    if(!getSuccessor().ping()) {
        setSuccessor(this.otherSuccessor);
    }
    else {
        this.otherSuccessor = getSuccessor().getSuccessor();
    }
}
```

from Chord.LocalNode.java, lines 152-174

Maintaining replicas

In order to avoid single points of failure, a peer can backup a file with a desired replication degree, so that a certain number of errors can occur, without impacting the global service. As described before, the peer responsible for a chunk stores the first replica, and the immediate successors will store the following replicas, always skipping the original owner. That way, if, for some reason, a chunk can't be recovered from the peer responsible for it, the request can simply be made to its successor.

Chord consistency

When the Chord ring changes, either by having a new node joining or by another one leaving , the peers responsible for each chunk might change, so, in order to keep the service consistent, each peer not only logs the chunks it is storing, but also keeps their replica number. That way, a peer can easily find out, by using the DHT's lookup operation, whether or not it should be storing a certain chunk, and redirect it to the peer that should actually store it.

That consistency is ensured by a scheduled task (using thread pools), which calls the method updateOwners:

```
private void updateOwners() {
   Log("Updating owners redirects");

   for (Map.Entry<ChunkID, StoredChunkEntry> entry :
this.peerState.getStoredLog().getMap().entrySet()) {
       ChunkID chunkID = entry.getKey();
       StoredChunkEntry chunkEntry = entry.getValue();

       Node node = this.responsibleFor(chunkID, chunkEntry.getReplicaNo(),
chunkEntry.getOwner());
       if (node.getID().equals(this.localNode.getID())) {
           Log("I am the correct owner of the chunk " + chunkID + " rep " +
chunkEntry.getReplicaNo());
       }
       else {
           Log("I am not the correct owner of the chunk " + chunkID + " rep
" + chunkEntry.getReplicaNo() + ", " + node.getID() + " is.");
           this.peerState.getStoredLog().getChunk(chunkID,
this.backupFolderPath, Chunk.chunkThreadPool, (chunk) -> {
               System.out.println("Redirecting chunk " + chunk);
```

```
                redirectToNode(chunk, node);
            });
        }
    }
}
```

from Peer.java, lines 245-264

References

Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications

SDIS theory and practical content

Consistent Hashing - Georgia Tech - Network Congestion

P2P Systems-Failures in Chord