U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Second Assignment Report

SDIS 2020/2021 - MIEIC

Distributed Backup Service for the Internet

Ana Inês Oliveira de Barros	up201806593
Eduardo da Costa Correia	up201806433
João de Jesus Costa	up201806560
João Lucas Silva Martins	up201806436

Introduction

The main goal of this report is to describe the design and implementation of the developed peer-to-peer distributed backup service for the Internet developed for the Distributed Systems course unit.

In our case, we opted to develop on top of what was already done in the first project, so a good foundation was already available for us to focus on the new main features.

Overview

Just like in the first project, we support all the specified operations for the backup service:

- Backup of a file, with a desired replication degree;
- Restore of a backed up file;
- Delete a file's backup;
- Limit local storage used by the application;
- Retrieve service state information.

Summary of features:

- **SSLEngine**: To provide secure communication through TCP connections between peers.
- **Chord:** To guarantee the service's scalability with fault tolerance mechanisms, like:
    - Successor list;
    - Fix fingers;
    - Notify;
    - Stabilize.
- **Thread Pools:** To handle and send multiple messages at once.
- **Java NIO:** To execute non-blocking I/O procedures and handle incoming connections.

Protocols

RMI

RMI provides the mechanism by which the server and the client communicate and pass information back and forth. In our case, both the *Peer* class and the *ChordNode* class act as RMI servers programs which need to create the initial remote objects and export them to the RMI runtime, making them available to receive incoming remote invocations from the client application.

TestApp

Interface implemented by the *Peer* class:

```java
public interface TestInterface extends Remote {
    String join() throws RemoteException;

    String backup(String filePath, Integer replicationDegree) throws RemoteException;

    String restore(String filePath) throws RemoteException;

    String delete(String filePath) throws RemoteException;

    String reclaim(int maxCapacity) throws RemoteException;

    String state() throws RemoteException;
```

[TestInterface.java]

The following setup procedure creates and exports one remote object and registers it with the RMI registry.

```java
public static void main(String[] args) {
    // parse cmdline args
    Peer prog = null;

    try {
        prog = new Peer(args);
    } catch (IOException e) {
        System.err.println("Couldn't initialize the program.");
        e.printStackTrace();
        usage();
    }

    assert prog != null;
```

[Peer.java]

```java
// setup the access point
TestInterface stub;
try {
    stub = (TestInterface) UnicastRemoteObject.exportObject(prog, 0);
    prog.registry = LocateRegistry.getRegistry();
    prog.registry.bind(("peer" + prog.id), stub);
} catch (Exception e) {
    System.err.println("Failed setting up the access point for use by the testing app.");
    System.exit(1);
    // e.printStackTrace();
}
try {
    prog.initCoordNode();
} catch (IOException e) {
    System.err.println("Couldn't create node socket");
```

[Peer.java]

The *Peer* class also provides an implementation for each remote method in the remote interface *TestInterface* shown above. Here is an example of an implementation of one of these methods (*delete*):

```java
public String delete(String filePath) throws RemoteException {
    String fileId;
    try {
        fileId = DigestFile.getHash(filePath);
    } catch (IOException e) {
        throw new RemoteException("Deletion of " + filePath + " failed.");
    }
    return "File " + filePath + " deletion: " + this.deleteFromId(fileId);
}
```

[Peer.java]

For client testing, the *TestApp* class implements the client role by allowing us to invoke the sub protocols provided by the service to back up, restore and delete files, as well as to reclaim the storage space being used by the service and to inspect the internal state of a peer.

In the code snippet below, the client looks up for the remote object, using the same name used by the *Peer* class to bind its remote object. Also, the client uses the *LocateRegistry.getRegistry* API to create a remote reference to the registry on the server's host.

```java
public void join(ChordInterface nprime) throws RemoteException {
    this.predecessor = null;
    this.setSuccessor(nprime.findSuccessor(this.getId()));

    // init finger table
    for (int i = 0; i < m; ++i) {
        int fingerStartId = this.getFingerStartId(i);
        this.fingerTable[i] = nprime.findSuccessor(fingerStartId);
    }
}
```

[chord/ChordNode.java]

The client then invokes the lookup method on the registry to look up the remote object by name in the server host's registry. This returns a reference to a registry at the named host and the default registry port (usually *1099*). Finally, the client creates a new *TestInterface* object and invokes the desired method of the remote object.

```
TestInterface stub = null;
try {
    Registry registry;
    if (rmiinfoSplit.length > 1)
        registry = LocateRegistry.getRegistry("localhost", Integer.parseInt(rmiinfoSplit[1]));
    else
        registry = LocateRegistry.getRegistry();
    stub = (TestInterface) registry.lookup(rminame);
} catch (RemoteException | NotBoundException e) {
    System.err.println("Couldn't find/get the desired remote object.");
    e.printStackTrace();
    System.exit(1);
}
assert stub != null;

String reply = null;
String oper = args[1];
try {
    switch (oper.toUpperCase()) {
        case "BACKUP":
            if (args.length != 4) usage();
            String filePath = args[2];
            int replicationDegree = Integer.parseInt(args[3]);
            System.out.println("BACKUP " + filePath + " " + replicationDegree);

            reply = stub.backup(filePath, replicationDegree);
            break;
```

[TestApp.java]

Chord

Similarly, RMI provides the mechanism by which a *Peer* joins the Chord ring and by which each *ChordNode* instance communicates with their successors, predecessor, and *fingers*.

Each *ChordNode* instance exposes a few methods to be used by other nodes on the chord ring. Interface that exposes these methods:

```java
public interface ChordInterface extends Remote {
    int getId() throws RemoteException;
    ChordInterface getPredecessor() throws RemoteException;
    ChordInterface getSuccessor() throws RemoteException;

    ChordInterface[] getSuccessors() throws RemoteException;

    ChordInterface findSuccessor(int id) throws RemoteException;
    ChordInterface findPredecessor(int id) throws RemoteException;
    ChordInterface closestPrecedingNode(int id) throws RemoteException;


    void notify(ChordInterface n) throws RemoteException;

    InetAddress getAddress() throws RemoteException;
    int getPort() throws RemoteException;
    Map<Pair<String, Integer>, Integer> getStoredChunksIds() throws RemoteException;
}
```

[chord/ChordInterface.java]

The following setup procedure creates and exports one remote object and registers it with the RMI registry.

```java
ChordInterface stub;
try {
    stub = (ChordInterface) UnicastRemoteObject.exportObject(this, 0);
    registry.bind(this.id.toString(), stub);
    System.out.println("Registered node with id: " + this.id);
    System.out.println(this);
} catch (Exception e) {
    System.err.println("Failed setting up the access point for use by chord node.");
    e.printStackTrace();
    System.exit(1);
}
```

[chord/ChordNode.java]

The *ChordNode* class provides an implementation for each remote method in the remote interface *ChordInterface* shown above. Here is an example of an implementation of one of these methods (*getSuccessor*):

```java
public ChordInterface getSuccessor() throws RemoteException {
    boolean goneBad = false;

    for (int i = 0; i < this.succList.length; ++i) {
        ChordInterface succ = this.succList[i];
        if (succ == null) break;

        try {
            succ.getId();
            if (goneBad) {
                this.reconcile(succ);

                // My successor died, call backup protocol on the chunks i think he was storing
                this.backupSuccessorChunks();
            }
            return succ;
        } catch (RemoteException ignored) {
            goneBad = true;
            this.succList[i] = null; // node is dead => bye bye
        }
    }

    return null;

    // return this.fingerTable[0];
}
```

chord/ChordNode.java

Each node has a join method that automatically joins an existing chord ring, by selecting a random node already there. To do this, the node queries the rmiregistry for the list of objects there and joins the first one available. If no node is found, it is assumed that the node is the only one in the chord ring.

When the lookup is successful, the node initializes its first successor (in the successor list) and its finger table. Both of these are initialized with the assistance of the existing node, by querying who it thinks are successors of the IDs specified (id of the current node and IDs of the fingers).

```java
    ChordInterface node;
    try {
        System.out.println(peerId);
        node = (ChordInterface) this.registry.lookup(peerId);
    } catch (NotBoundException e) {
        return "An attempted lookup to a node in the network failed";
    }
    this.chordNode.join(node);
    // Resume pending tasks
    this.handlePendingTasks();
    return "Join success";
}
```

Peer.java

Messages and routing

The header of all messages exchanged between Peers follows the following template. In some messages, some fields can be omitted. To signal this, they'll be replaced by "..." in each message.

```
MESSAGE <file_id> <source_address> <source_port>
<destination_address> <destination_port> <destination_id> <path>*
```

- **file_id:** ID of the file. Obtained from the name, content, and other metadata of the file using SHA-256 hashing algorithm.
- **source_address:** IPv4 address of the peer which sent the message originally.
- **source_port:** Listening port of peer which sent the message originally.
- **destination_address:** IPv4 address of destination peer.
- **destination_port:** Listening port of destination peer.
- **destination_id:** ID of destination chord node.
- **path:** Collection of chord node IDs through which the message went.

Each Chord node acts as a **router**, so it forwards messages that are not meant for it through the ring to the closest preceding node it is aware of (successor list/finger). For each node the message passes through, the node's ID is added to the path field of the message.

```java
public void send(Message message) {
    if (this.messageIsForUs(message)) {
        System.out.println("\tNot sending message (its for me): " + message + "\n");
        messageHandler.handleMessage(message);
    } else { // message isn't for us
        System.out.println("Sending (ReHopping): " + message + "\n");
        this.sendToNode(message); // resend it through the chord ring
    }
}
```

chord/ChordController.java

If a message is meant for the current node (it isn't aware of any preceding node closest to the destination node), then the node will proceed to handle the corresponding message. Otherwise, the current node will send the message to the closest preceding node of the message's destination node (*destination_id* field of the message). In case the current node is the closest one, the message will be sent to the node's successor. This explanation refers to the code below:

```java
private void sendToNode(Message message) {
    ChordInterface nextHopDest = null;
    try {
        nextHopDest = closestPrecedingNode(message.getDestId());
    } catch (RemoteException e) {
        System.err.println("Could not find successor for message " + message + ". Message not sent.");
        e.printStackTrace();
    }
    assert nextHopDest != null;
    try {
        if (nextHopDest == this)
            nextHopDest = getSuccessor();
        message.setDest(nextHopDest);
        message.addToPath(nextHopDest.getId());
    } catch (RemoteException e) {
        System.err.println("Could connect to chosen next hop dest : TODO Max tries with timeout " + message);
        e.printStackTrace();
    }

    this.sock.send(message);
}
```

chord/ChordController.java

Backup

The backup service generates an identifier for each file it backs up. It then splits it into chunks and then backs up each chunk independently. To back up a chunk, the initiator-peer sends a PUTCHUNK containing the contents of that chunk.

The PUTCHUNK message is sent to the node responsible for the chunk. The node responsible for the chunk is the node whose ID is the closest to the chunk's ID. PUTCHUNK messages to store replications of this chunk are sent to the next N successors of the responsible node. PUTCHUNK Message Handler.

Header of a PUTCHUNK message:

```
PUTCHUNK [...] <replication_degree> <sequence_number> <chunk_no>
```

- **replication_degree:** Desired replication degree of the chunk.
- **sequence_number:** Remaining stores to achieve the desired replication degree.
- **chunk_no:** Number of the chunk being stored.

A peer that stores the chunk upon receiving the PUTCHUNK message, replies with a STORED message to its predecessor. This acknowledgment message is used to update a node's perceived chunks of its successor. The uses and detail implementations of this data structure are specified further in the report. STORED Message Handler

Header of STORED message:

```
STORED [...] <chunk_no> <chunk_id>
```

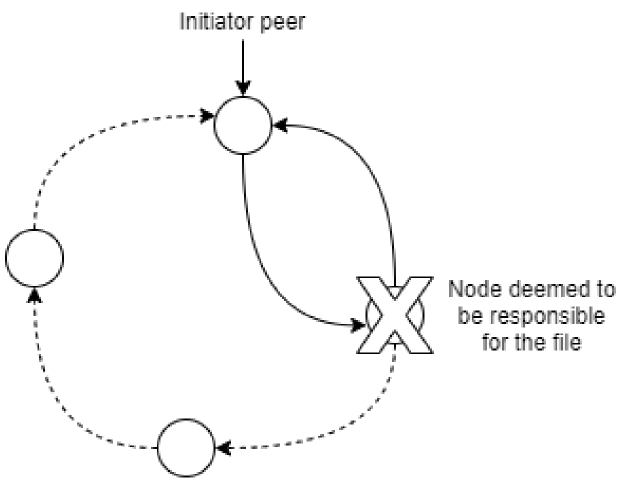
- **chunk_no:** Number of the stored chunk.
- **chunk_id:** ID of the stored chunk.

Restore

The node that starts a restore request of a file will need to recover all the backed up chunks of that file. To recover a chunk, the initiator node sends a GETCHUNK message to the node responsible for that chunk. In case this node hasn't stored the chunk (due to limiting its backup storage space), it will redirect the GETCHUNK message to its successor. The message will be propagated across the chord ring until it reaches a node that is storing the desired chunk. (for example, after all of its space being reclaimed)

In our first implementation of the protocol, the initiator peer was responsible for stopping the propagation of the GETCHUNK. We quickly found that this approach was faulty when handling some cases, such as the following:

1. The initiator node initially sends the message to the node it deems responsible for the file, which corresponds to its direct predecessor.
2. It forwards the message to its successor (back to the initator), in case it hasn't stored the requested file.
3. The initiator node would wrongly deduce the message had looped through the wrong ring.



Our solution to this problem is to pass the responsibility of stopping the propagation of the message to the responsible peer. This is achieved by using a flag, has_looped, which indicates if the message has passed through the responsible node twice. In this case, it will resend the GETCHUNK message directly to the initiator peer. Upon receiveng this GETCHUNK message, the initiator peer will be informed that no node in the network is storing the file, by checking if the message was initially sent by him and that flag has_looped set to true. If this is the case, it will know that the backup process has failed because at least one chunk wasn't found in the network.

To prevent this, we added two extra fields, *looped* and *responsible*.

GETCHUNK Message Handler

Header of a GETCHUNK message:

```
GETCHUNK [...] <chunk_no> <looped> <responsible>
```

- **chunk_no:** Number of the chunk being retrieved.
- **looped:** True when the message has looped through the chord ring. False, otherwise.
- **responsible:** ID of the node responsible for the chunk

Upon receiving a GETCHUNK message, a node that has a copy of the specified chunk shall send it in the body of a CHUNK message directly to the node who requested the chunk. CHUNK Message Handler

Header of a CHUNK message:

```
CHUNK [...] <chunk_no>
```

- **chunk_no:** Number of the chunk whose body is being sent.

Delete

To delete all file chunks existing in the network, the initiator peer sends a DELETE message to its successor. This message passes through the entire network until it reaches the initiator peer. If a peer has chunks referring to the file specified in the message, they are deleted. DELETE Message Handler

Header of a DELETE message:

```
DELETE [...]
```

Reclaim

When reclaiming disk storage, the node may need to delete a copy of a chunk it has backed up. Upon deleting a backed up chunk, it sends a REMOVED message to the node responsible for the chunk. If the peer that receives this message isn't storing the chunk, the message will be resent through the ring, similarly to the previous messages. When a peer that is either the initiator or is one of the nodes that are storing the chunk file receives the message, it reinitiates the backup protocol. This is achieved by sending a PUTCHUNK with the desired replication degree and respective chunk to the responsible peer of that chunk. REMOVED Message Handler

Header of a REMOVED message:

```
REMOVED [...] <chunk_no> <chunk_id>
```

- **chunk_no:** Number of the removed chunk.
- **chunk_id:** ID of the removed chunk.

State

This operation allows us to observe the service state. In response to such a request, the node sends to the client the following information:

- For each file whose backup it has initiated:
  - The file path name
  - The backup service ID of the file
  - The desired replication degree
  - For each chunk of the file:
    - Its ID

        - Its perceived replication degree
- For each chunk it stores:
    - Its ID
    - Its size (in KBytes)
    - The desired replication degree
    - Its perceived replication degree
- For each chunk that the peer’s successor is storing:
    - Its fileId
    - Its number
    - Its ID
- The peer's storage capacity, i.e. the maximum amount of disk space that can be used to store chunks, and the amount of storage (both in KBytes) used to back up the chunks.

Example of a peer’s state output:

```
CMD: state
Files I initiated the backup of:
Chunks I am storing:
        File ID: d06f4be60f15642ad3b89887fcf829688acb7fe8674c87088482f7f885cb22de
        Chunk no: 1 Id: 7
                Size: 30000
                Desired replication degree: 2
Chunks my succ is storing:
        FileId: d06f4be60f15642ad3b89887fcf829688acb7fe8674c87088482f7f885cb22de ChunkNo: 0 ChunkId: 6
5
        FileId: d06f4be60f15642ad3b89887fcf829688acb7fe8674c87088482f7f885cb22de ChunkNo: 2 ChunkId: 9
6
        FileId: d06f4be60f15642ad3b89887fcf829688acb7fe8674c87088482f7f885cb22de ChunkNo: 1 ChunkId: 7
Storing 30KB of a maximum of infinite KB.
```

Concurrency design

Multi-threading and Thread pools

The *SockThread* class implements the **Runnable** interface and contains a **fixed thread pool** from the java.util.concurrent package. This type of pool always has a specified number of threads running. Whenever a task terminates, a new task from the thread pool waiting queue is picked up from the newly free thread.

Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads (waiting queue). Whenever a message is received or needs to be sent, a message handler task is added to the corresponding thread pool, for execution:

```java
public void send(Message message) {
    this.sendThreadPool.execute(() -> this.sendInner(message));
}
```

sender/SockThread.java

```java
private void readOuter(SelectionKey key, SocketChannel socketChannel, SSLEngineData d) {
    try {
        boolean isClosed = this.read(socketChannel, d);

        // create message instance from the received bytes
        if (isClosed) {
            key.cancel();
            d.thread.shutdown();

            if (d.content.size() > 0) {
                ByteArrayInputStream bis = new ByteArrayInputStream(d.content.toByteArray());
                d.content.reset();

                ObjectInput in = new ObjectInputStream(bis);
                Message msg = (Message) in.readObject();
                bis.close();

                // handle message
                this.receiveThreadPool.submit(() -> this.observer.handle(msg));
            }
        }
    } catch (IOException | ClassNotFoundException ignored) {
        key.cancel();
        d.thread.shutdown();
        // System.err.println("Lost message.");
    }
}
```

sender/SockThread.java

Appropriate data structures and atomic values

With the use of threads, there is a need for using appropriate data structures and atomic values (in critical regions). We used a ConcurrentHashMap from the java.util.concurrent package which supports full concurrency of retrievals and adjustable expected concurrency for updates. We also used AtomicBooleans from the java.util.concurrent.atomic package. This package contains a small toolkit of classes that support lock-free thread-safe programming on single variables. The code snippets below illustrate some examples where these classes are used.

```
private final AtomicBoolean running = new AtomicBoolean(false);
private final ExecutorService sendThreadPool = Executors.newFixedThreadPool(MAX_CONNS);
```

src/sender/SockThread.java

```
InetAddress address;
Integer port;
private final ConcurrentMap<Pair<String, Integer>, CompletableFuture<byte[]>> receivedChunks;

public MessageHandler(SockThread sock, ChordController chordController) {
    this.controller = chordController;
    this.receivedChunks = new ConcurrentHashMap<>();
    this.address = sock.getAddress();
    this.port = sock.getPort();
}
```

src/sender/MessageHandler.java

Synchronized methods and statements

Synchronized methods and statements enable a simple strategy for preventing thread interference and memory consistency errors. We used these in our project to achieve better concurrency:

```
synchronized (State.st) {
    DigestFile.deleteFile(message.getFileId());
    State.st.removeSuccChunk(message.getFileId());
}
```

sender/MessageHandler.java

Java NIO

The Java NIO Selector is a component which can examine one or more Java NIO Channel instances, and determine which channels are ready for operations like reading and/or writing. This way, a single thread can monitor multiple channels, and thus multiple network connections.

In our project, Java NIO is used for the writing and reading of messages, and accepting of connections. Each peer acts both as a server and as a client. A selector is used to accept new connections and read from existing sockets. All socket channel reading and writing operations are asynchronous.

```
Iterator<SelectionKey> selectedKeys = this.selector.selectedKeys().iterator();
while (selectedKeys.hasNext()) {
    SelectionKey key = selectedKeys.next();
    selectedKeys.remove();
    if (!key.isValid())
        continue;

    try {
        if (key.isAcceptable()) {
            this.acceptCon(key);
        } else if (key.isReadable()) {
            SocketChannel socketChannel = (SocketChannel) key.channel();
            SSLEngineData d = (SSLEngineData) key.attachment();

            try {
                d.thread.submit(() -> this.readOuter(key, socketChannel, d));
            } catch (RejectedExecutionException ignored) {
                key.cancel();
            }
        }
    } catch (CancelledKeyException ignored) {
    }
}
```

sender/SockThread.java

Upon accepting a new connection, we attribute the new socket channel a new thread (up to a limit). Every time this socket channel is notified about pending read operations, it uses this thread (single thread thread-pool). This is important, because it allows us to read concurrently and guarantees that the messages of each SSLEngine are processed sequentially.

JSSE

Peers send and receive messages through TCP socket channels. The data flow through these channels is encrypted using Java's SSLEngine. This means that SSLEngine is used for all message communication (all protocols).

Our implementation requires client authentication, JKS, and SunX509. TLS version 1.3 is used, and our code conforms to the half-close policy (new requirement after TLS version 1.2). This means that all connections end with the independent closure of both the outbound and inbound ends, thus preventing truncation attacks.

Both reading and writing is done with non-blocking sockets and Java NIO's features (discussed further in the scalability section of the report). This is the main reason why SSLEngine is used.

```java
hs = res.getHandshakeStatus();

// check status
switch (res.getStatus()) {
    case OK:
        // do nothing ?
        break;
    case BUFFER_OVERFLOW:
        // the client maximum fragment size config does not work?
        peerAppData = this.handleOverflow(engine, peerAppData);
        break;
    case BUFFER_UNDERFLOW:
        // bad packet, or the client maximum fragment size config does not work?
        // we can increase the size
        peerNetData = this.handleUnderflow(engine, peerNetData, peerAppData);
        // Obtain more inbound network data for src,
        // then retry the operation.
        break;
    case CLOSED:
        engine.closeOutbound();
        return 0;
    default:
        throw new IllegalStateException("Unexpected value: " + res.getStatus());
}
```

sender/SockThread.java

After the server detects that the connection is being closed by the peer (through the unwrapping of a CLOSE message), it proceeds with an orderly shutdown. If the connection is shut down by the peer without following the orderly shutdown procedure, the server still attempts to close the connection in an orderly fashion. The received message is processed after the connection is closed, even in the case of *end of stream* errors or other non-orderly shutdowns.

The client can time out while waiting for the server to acknowledge the orderly shutdown. In these cases, the client closes its socket channel and leaves.

Note: We are using the SSLEngine configuration that allows for bigger packets.

Scalability

In order to achieve higher scalability in our project, we decided to implement scalability both at design level and at implementation level. At design level, we implement a **Chord** architecture that provides routing of the messages. At implementation level, we used thread-pools to simultaneously handle several processes and Java NIO to achieve asynchronous I/O operations. All of these subjects have already been described in previous sections in this report.

Furthermore, to prevent hashing collision of both file and Chord node's IDs, we used the SHA-256 algorithm.

Additionally, a node may join the ring directly, by specifying the IP address and port of an existing node in the ring. Otherwise, it is also possible for a node to automatically join any of the nodes in the chord ring by looking at the objects that are bound to the RMI registry. When joining the chord ring, the new node queries the existing node for its successor and fingers, thus joining the ring.

Fault-tolerance

Since we implemented the Chord protocol, we took advantage of its fault-tolerant features to achieve a robust and reliable implementation.

We are using m = 128, so we have 7 fingers in our finger table. Assuming each node has a 50% chance to fail (quite high!), we achieve the following chances of all nodes failing:

- 3 successors: ≃ 12%
- 4 successors: ≃ 6%
- 5 successors: ≃ 3%
- 6 successors: ≃ 1%
- **7 successors: ≃ 0%**

With this in mind, we decided to go with the **7 successors list size**, since it is virtually impossible for all the successors of a node to fail at once.

We use the **SHA-256 hashing function** to generate IDs. This helps us achieve some tolerance to denial-of-service (DoS) attacks.

We call the *fixFingers* method periodically to refresh the finger table entries:

```java
public void fixFingers() {
  if (nextFingerToFix >= m)
      nextFingerToFix = 0;

  int succId = this.getFingerStartId(nextFingerToFix);
  try {
      // System.out.println("I am " + this.id + " and I'm updating finger " + nextFingerToFix + ", id: " + succId);
      fingerTable[nextFingerToFix] = this.findSuccessor(succId);
      // System.out.println("They tell me it's: " + fingerTable[nextFingerToFix].getId());
  } catch (RemoteException e) {
      fingerTable[nextFingerToFix] = this;
  }
  ++nextFingerToFix;
}
```

chord/ChordNode.java

We use the *notify* method to notify a node's successor of its existence, so the successor can change its predecessor to this node.

```java
public void notify(ChordInterface nprime) throws RemoteException {
    int nprimeId = nprime.getId();

    if (this.predecessor == null) {
        this.predecessor = nprime;
    } else {
        try {
            int predecessorId = this.predecessor.getId();
            if (ChordNode.inBetween(nprimeId, predecessorId, this.id))
                this.predecessor = nprime;
        } catch (RemoteException ignored) {
            // if our predecessor died, we accept the new one
            this.predecessor = nprime;
        }
    }
}
```

chord/ChordNode.java

To stabilize the network and guarantee that it is consistent, the *stabilize* method is called periodically. The corresponding node asks the successor about its predecessor, verifies if its immediate successor is accordingly and tells the successor about it.

```java
public void stabilize() throws RemoteException {
    // TODO if statement is needed?
    //if (this.predecessor != null) {
    //}

    ChordInterface succ = this.getSuccessor();
    if (succ == null) return; // very bad

    // update predecessor
    ChordInterface me$ = succ.getPredecessor();
    if (me$ != null) { // if pred knows about a succ
        try {
            if (ChordNode.inBetween(me$.getId(), this.id, succ.getId())) {
                this.setSuccessor(me$);
                succ = me$;
            }
        } catch (RemoteException ignored) {
            // if we fail getting the me$ id, it is dead and we don't want it as a successor
        }
    }
    succ.notify(this); // notify successor about us

    this.reconcile(succ);
}
```

[chord/ChordNode.java]

The *checkPredecessor* method periodically checks if a predecessor is alive.

```java
public void checkPredecessor() {
    if (predecessor != null) {
        try {
            this.predecessor.getId();
        } catch (RemoteException e) {
            this.predecessor = null;
        }
    }
}
```

[chord/ChordNode.java]

When a peer disconnects the network (due to crashing or simply quitting), its predecessor will send a REMOVED message as described in the RECLAIM sub protocol.

This will ensure that the replication degree will be maintained in the network, as the backup protocol will be reinitiated again if a node that is either storing the file or is its initiator exists. To achieve this, when a node sets its new direct successor, it stores a record of all the new successors' chunk IDs. Furthermore, when a node stores a new file or deletes an old one, it sends its predecessor either a STORED or a REMOVED message, respectively.

This approach greatly improves fault tolerance since the only way that a stored chunk can be lost in the network is if each pair of predecessors and successors crash simultaneously, which is highly unlikely.

Conclusion

To conclude, we think that this project allowed us to better understand the adequate protocols and algorithms used to develop secure, fault tolerant and scalable systems on the internet, by applying the knowledge acquired in the courses classes. It also allowed us to refine some concepts applied in the first project and implement our own solution.

That being said, we designed this project to be as robust as we could make it and handle all kinds of possible scenarios, leading us to believe we implemented it successfully and achieved the goals of this course unit.

Despite this, we believe that the protocol could have been implemented in a slightly different way to achieve better results and performance.

Firstly, we could use a network overlay like Pastry instead of Chord, because it works better on file/chunk backup systems that support replication.

Secondly, we could send only the headers of the messages and make it so the peers that want the bodies of them, ask for it directly from the initiator peer. Although doing this would make the protocol more complex, it would reduce the size of the messages that are sent in each hop, reducing the stress on the network.

Thirdly, we could make use of an input buffer to save incoming data. This would allow us to send arbitrarily sized messages and, consequently, divide the files into bigger chunks (currently the files are divided into chunks of 30 KB). These changes would require some minor design changes relating to the way the data is read, but the benefits would be noticeable. Also, we would need to divide the messages that we’re sending.

Bibliography

1. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. 2001. *Chord: A Scalable Peer-to-peer Lookup Service for Internet*.
2. Oracle. “Java Secure Socket Extension (JSSE) Reference Guide”. https://docs.oracle.com/en/java/javase/15/security/java-secure-socket-extension-jsse-reference-guide.html
3. alkarn. “Server and Client implementation with SSLEngine”. https://github.com/alkarn/sslengine.example
4. Maarten van Steen, Andrew S. Tanenbaum. 2018. *Distributed Systems*.
5. Olliotte Rusty Harold. 2013. *Java Network Programming*.