# Distributed Backup Service for the Internet

FEUP - SDIS 2021

Second Project - Group t6g21

Daniel Garcia Silva (up201806524)

João Gonçalves (up201806162)

Rafael Cristino (up201806680)

Pedro Queirós (up201806329)

## Contents

# 1 Overview

This project's main objective was to improve the first project's peer-to-peer distributed backup service, allowing it to work over the internet.

The project implements the four previously described protocols (backup, restore, delete and reclaim), together with the delete enhancement (so that when a peer that has backed up a file is offline and joins his replica is deleted).

We devised a completely multithreaded design with the use of Java NIO and thread-pools, using asynchronous writes to files and non-blocking I/O for sending and receiving messages.

For secure communication, we use JSSE's SSLEngine, which allows us to have more control over the asynchronicity of the communication and to encrypt/decrypt the sent/received messages.

A Chord network implementation is used to allow our peers to find other peers, locate where a file will be or is stored. The use of Chord also ensures our project's scalability and fault-tolerance.

Each peer also has non-volatile state that is written regularly to the disk so that, in the advent of a failure, it can recover on start-up.

# 2 Implemented Protocols

The protocols we implemented are the backup, restore, delete and reclaim protocols. Firstly, we'll give a short explanation about the way the Remote Method Invocation interface works, then we will go through each protocol in detail and finally we'll take a look at how the peers implement Chord among themselves.

## 2.1 RMI Interface (client)

To interface with the peer, that is, to tell him what to do, it features an RMI interface that can be used with a client application. We provide an implementation of such an application, in the form of the TestApp.

### 2.1.1 How to run the peer / TestApp

The peer can be ran with the included scripts (that are located in the folder scripts):

- `./compile.sh` - Compiles the peer and the TestApp. Must be executed in the source directory.
- `./cleanup.sh` - Deletes the peer's file system (their non-volatile state plus the files they've backed up and restored). Must be ran in the build directory.
- `./peer.sh <peer_access_point> <peer_server_port> [<other_peer_ip> <other_peer_port>]` - Initiates the peer. The last two arguments are optional. If they are not present, the peer will create a new chord ring. Must be ran in the build directory.
  - `peer_access_point` - the identifier to which the peer will be bound in the RMI service.
  - `peer_server_port` - the port of the server that the peer will open (to communicate with the other peers).

  - `other_peer_ip` - the ip of a peer that is already in the chord ring.
  - `other_peer_server_port` - the port of the server of the peer that is already in the chord ring.

- `./test.sh <peer_access_point> BACKUP|RESTORE|DELETE|RECLAIM|STATE|FINGERS [<operand1> [<operand2]]` - Run the test app with a command for the peer.
  - `peer_access_point` - the identifier to which the peer will is bound in the RMI service.
  - `operand1` e `operand2` - arguments for the peer command. The commands possible are:
    * `BACKUP <file_to_backup> <replication degree>` - initiates the backup protocol.
    * `DELETE <name_of_file_to_delete>` - initiates the reclaim protocol.
    * `RESTORE <name_of_the_file_to_restore>` - initiates the restore protocol.
    * `RECLAIM <amount_of_space_desired>` - initiates the reclaim protocol.
    * `STATE` - shows the peer's state (files it has backed up, ...).
    * `FINGERS` - shows the peer's Chord finger table.

## 2.2 Protocols (peer to peer)

In this section we will describe in detail the protocols that were implemented. Here, when there is need to refer to Chord, we are specifying the Chord messages that are exchanged between the peers, we only reference the Chord functionality that is used. Chord is explained in more detail in the next section.

#### General messages

Firstly, lets introduce the general messages that are used by many of the protocols: `PROCESSEDYES`, `PROCESSEDNO` and `REDIRECT`.

- `PROCESSEDYES <sender_id>` - this message generally means that the request was processed with success and that it says yes to what it asked. For example, in the backup protocol, this is used to say that the peer processed the request and needs to receive the file data.
- `PROCESSEDNO <sender_id>` - this message generally means that the request was processed with success and that it says no to what it asked. For example, in the backup protocol, this is used to say that the peer processed the request but doesn't need to receive the file data.
- `REDIRECT <sender_id> <node_address> <node_port> <node_id>` - this message is used to redirect the request to the peer referenced by the triple (node_address, node_port, node_id).
- `NODE <sender_id> <node_address> <node_port> <node_id>` - this message brings a peer's reference (node_address, node_port, node_id).
- `GETSUCCESSOR <sender_id>` - this message asks a peer to reply with is successor by means of a `NODE` message.
- `GETPREDECESSOR <sender_id>` - this message asks a peer to reply with is predecessor by means of a `NODE` message.

- `DATA <sender_id> <file_key>` - this message brings the data of the file with the specified key in its body.

The TestApp is implemented in the `TestApp` class, and the RMI interface specified in the `configuration.ClientInterface` interface.

### 2.2.1 Backup

The backup protocol has the goal of backing up a file among the peers. Instead of splitting the file in chunks, the whole file is backed up to each peer.

To do this, we need to locate the peer that will store the file. To do so, we use the Chord's interface to `lookup` the file's key, which was generated based on its metadata. This will return us the peer that should hold the file.

The backup protocol uses the `PUTFILE` message:

- `PUTFILE <sender_id> <file_key> <byte_amount>` - specifies the sender, references the file and includes its size in bytes.

Once the peer knows where to send the file to, he sends a `PUTFILE` message. A peer that receives this message replies with `PROCESSEDYES` if he can back up the file and can receive the data, with `PROCESSEDNO` if he has already backed up the file (doesn't need the data) and with `REDIRECT` if he can't back up the file due to storage restrictions. In this last case, the redirect message brings the peer's successor, to whom the peer who initiated the backup will try to send the file, by resending the putfile message.

When a peer redirects a `PUTFILE` request he also stores the file's id (which we call a *file pointer*), because if he received the request then when the peer wants to restore the file he will probably ask him for it. This way, by storing all the files that he redirected, he knows to redirect also when a `GETFILE` request comes in. This works in chain as there can be multiple peer's who can't store a given file in a row.

A potential problem with using these file pointers is that a peer stores the pointer without knowing whether the file was really backed up or not. For example, if all the peers to the right of the peer that stored the pointer can't back up the file, they all will have dangling pointers, and a restore may result in extra steps because of that. To do that, after backing up the file, the peer sends a `REMOVEPOINTER` message to all the peers who had stored a pointer after the last successful backup.

Once the peer who started the backup received the `PROCESSEDYES` message he sends the file's `DATA`. To implement the replication degree, we then send the file (by means of a `PUTFILE` again) to the successor of the node that stored one of its replicas, and so forth, until either the real replication degree is equal to the desired or we went all around the Chord ring arriving back on the original node.

The backup protocol with Chord raises a question. What happens if a peer is down and a file whose key maps to that peer's id is backed up? If the peer receives a request for that once he comes back on, he won't know that it was backed up so he won't be able to send it back. This question is resolved in section 2.2.5.

This protocol is implemented in the `actions.Backup` and `server.BackupHandler` classes.

### 2.2.2 Restore

The restore protocol has the goal of allowing a peer that backed up a file to recover it. To do so, as in the backup protocol, we `lookup` the file's key to find the peer who has the responsibility of storing it (according to Chord - original peer) and then ask him for it.

The restore protocol uses the `GETFILE` message:

- `GETFILE <sender_id> <file_key> <node_address> <node_port> <node_id>` - specifies the sender, references the file and also references the node to send it to.

The peer who initiates the restore sends the `GETFILE`. A peer that receives it replies with `PROCESSEDYES` if he has the file and will send it, with `PROCESSEDNO` if he doesn't have the file (restore failed) and `REDIRECT` if he has a file pointer for that file. After a `PROCESSEDYES`, the peer will then send the requested file's `DATA` to its owner. After a `REDIRECT` message, the peer who started the restore will do it again to the original peer's successor and so on, until he either receives a `PROCESSEDYES` and the restore is successful or a `PROCESSEDNO` and the restore is failed.

This protocol is implemented in the `actions.Restore` and `server.ServerRouter` classes.

### 2.2.3 Delete

The delete protocol allows a peer to delete from the network a file that it previously requested a backup for. As previously, the peer who starts a delete will `lookup` the file's key to find the peer who has the responsibility of storing it, and then send him the delete request.

The delete protocol uses the `DELETE` message:

- `DELETE <sender_id> <file_key>` - specifies the sender and references the file to be deleted.

Once the `DELETE` message is sent to the peer, it has two possible responses: a `PROCESSEDYES` in the case where the peer had the file or a pointer to it and he deleted it and a `PROCESSEDNO` in the case where the peer didn't have neither the nor the pointer.

The peer only stops sending `DELETE` messages once he receives a `PROCESSEDNO` message, which means that there are no more replicas in the next successors. A `PROCESSEDYES` means that there are more replicas, in the case where it comes from a peer who had a pointer, or that there may be more replicas, in the case where it comes from a peer who had the file. Finally, the peer who started the backup removes the file from its non-volatile state.

This protocol is implemented in the `actions.Delete` and `server.ServerRouter` classes.

### 2.2.4 Reclaim

The reclaim protocol allows a peer to limit the space available for backed up files in its storage.

When this protocol executes, if the already occupied space surpasses the desired limit, the peer determines the best file to eliminate (the ones whose deletion maximizes the space occupied while not surpassing the defined limit). Then, the peer deletes those files and starts a backup protocol for each of them, to "guarantee" that they maintain their replication degree.

### 2.2.5 Check

The check protocol allows a peer to, after connecting to a chord ring, gather pointers for the files that should belong to him (and that he doesn't have).

The check protocol uses the `CHECK` and the `ADDPOINTER` messages:

- `CHECK <sender_id> <node_address> <node_port> <node_id>` - specifies the sender and references the node to send the messages to.
- `ADDPOINTER <sender_id> <file_key>` - specifies the sender and references the file that the pointer refers to.

Upon entering the chord ring, the peer will send a `CHECK` message to his successor (because if a file that was supposed to be his was backed up it will have gone to his successor).

Upon the reception of a check message, a peer checks the files that he has saved (after receiving them from other peers) and the file pointers that he has. Then, that peer will send an `ADDPOINTER` message to the one who requested the check for each of the files that should be his (files that are located before the peer who initiated the check).

This protocol is implemented in the `actions.CheckFiles` and `server.ServerRouter` classes.

## 2.3 Chord

The core function of Chord is to lookup a key. The result of that lookup is the node to which the key maps (the closes node to the key that comes after it).

Each node of the chord ring is represented by its id, address and port. This is done through the `chord.ChordNode` class:

```
public class ChordNode {
    private final InetSocketAddress address;
    private final int id;

    public ChordNode (InetSocketAddress a, int id) {
        this.address = a;
        this.id = id;
    }

    public InetSocketAddress getInetSocketAddress() {
        return address;
    }

    public InetAddress getInetAddress() {
        return address.getAddress();
    }

    public int getPort() {
        return address.getPort();
    }

    public int getId() {
        return id;
    }

    @Override
    public String toString() {
```

```
        return "NODE[id=" + id + "]@" + address.getAddress().getHostAddress() +
            ":" + address.getPort();
    }
}
```

### 2.3.1 Messages

Our implementation of the Chord algorithm is based on the following messages:

- `LOOKUP <sender_id> <key>` - specifies the sender and the key to lookup.
- `LOOKUPRESPONSE <sender_id> <key> <node_address> <node_port> <node_id>` - specifies the sender id and the key that was lookup, bringing the resulting node referenced by (node_address, node_port and node_id).
- `NOTIFY <sender_id> <node_address> <node_port> <node_id>` - specifies the sender id and the node that is notifying (the peer who receives this message is being notified that the referenced node thinks he is his predecessor).

### 2.3.2 Implementation

Because chord is fully distributed, it needs to keep the ring's consistency through it's lifetime. This is done through the setup of special functions that are recurrently called with a given time interval between each execution.

- `updateFingers` - Keeps the finger table of the node up to date, by executing regular lookups of each of the table's entries' keys and replacing the corresponding node if applicable.
- `stabilize` - Keeps the successor and predecessors up to date. Works by obtaining the node's successor's predecessor and seeing if it is equal to himself. If not, it means that his successor has changed and it changes it to the predecessor of his previous successor. Finally, he notifies that node that he is his predecessor.
- `updateSuccessorsSuccessors` - Keeps the successors list up to date (more on that on the fault-tolerance section).
- `checkPredecessor` - Checks if the predecessor is still alive.

Implementation of updateFingers (line 204, `chord.Chord`):

```
public void updateFingers() throws Exception {
    Logger.debug(self, "Update fingers");

    int originalFingerToFix = nextFingerToFix;
    nextFingerToFix = nextFingerToFix + 1;
    if (nextFingerToFix > this.m - 1) nextFingerToFix = 0;

    ChordNode fingerValue = lookup(getFingerTableIndexId(nextFingerToFix)).get();
    if (fingerValue == null) {
        Logger.error("Lookup returned null in updateFingers!");
        nextFingerToFix = originalFingerToFix;
        return;
    }
```

```
    try
    {
        fingerTable.get(nextFingerToFix);  // if already exists
        fingerTable.set(nextFingerToFix, fingerValue);
    }
    catch (IndexOutOfBoundsException e)
    {
        try
        {
            if (fingerTable.size() == 0) fingerTable.add(fingerValue);
            else
            {
                fingerTable.get(nextFingerToFix - 1); // if it's filled up to this point
                fingerTable.add(fingerValue);
            }
        }
        catch (IndexOutOfBoundsException e1)
        {
            nextFingerToFix--; // To try to fix this one again
            throw new Exception("updateFingers: finger table was not valid.");
        }
    }
    Logger.debug(self, "Update fingers ended");
}
```

Implementation of stabilize (line 246, `chord.Chord`):

```
public void stabilize() throws Exception {
    Logger.debug(self, "Stabilize");

    if (successor == null || successor.getId() == getId()) return;

    Logger.debug(self, "Sending GETPREDECESSOR to " + successor);
    Message reply = SSLClient.sendQueued(successor, MessageFactory.getGetPredecessorMessage(se

    try {
        ChordNode predecessorOfSuccessor = reply.getNode();  // if it has no predecessor it wi
        Logger.debug(self, "Got PREDECESSOR = " + predecessorOfSuccessor);

        if (isBetween(predecessorOfSuccessor, self, successor, false)) {
            successor = predecessorOfSuccessor;
        }
    } catch (Exception e) {
        Logger.debug(self, "Didn't have predecessor yet!");
    }

    notifyPredecessor(successor);
    Logger.debug(self, "Stabilize ended");
}
```

Implementation of checkPredecessor (line 307, `chord.Chord`):

```
public void checkPredecessor() {
```

```
    if (predecessor == null) return;
    Logger.debug(self, "Checking predecessor...");

    if (!SSLPeer.isAlive(predecessor.getInetSocketAddress()))
    {
        predecessor = null;
        Logger.debug(self, "Predecessor was not alive!");
    }
    else Logger.debug(self, "Predecessor was alive!");
}
```

**A node enters the ring**
When a node enters the ring, he must lookup his **successor**.

# 3 Concurrency Design

Our concurrent design is based on each peer having the function of both a client and a server. The client is used to connect to other peers and the server to receive connections from other peers.

## 3.1 Server

Once a peer turns on he starts his server thread, which is based on the `SSLEngine` for encryption and TCP sockets for transferring data with the clients.

The server is based on a loop that uses the `java.nio.channels.Selector` class to manage multiple `SocketChannel` instances in a single thread. Upon reception of a message, the server submits a new thread to the executor that will handle the message and act according to the protocols described above.

This loop is present in line 70 of the `sslengine.SSLServer` class:

```
(...)
while (this.available){
    this.selector.select();
    Iterator<SelectionKey> selectedKeys = selector.selectedKeys().iterator();

    while (selectedKeys.hasNext()) {
        SelectionKey key = selectedKeys.next();
        selectedKeys.remove();
        try {
            if (!key.isValid())
                continue;

            if (key.isAcceptable())
                this.accept(key);

            else if (key.isReadable())
            {
                // read data and handle it
(...)
```

## 3.2 Client

A client action thread is started by RMI every time a remote method invocation is executed. This thread is responsible for an instance of one of the described protocols, and there may be more than one of these threads running at a time.

These client threads start asynchronous connections with the other peers' servers to make the requests needed to fulfill their protocols.

Example of sending a message to the a server (line 16 of `actions.CheckFiles`):

```
try {
    Logger.debug(Logger.DebugType.CHECK, "Checking files.");
        SSLClient.sendQueued(configuration.getChord().getSuccessor(),
            MessageFactory.getCheckMessage(configuration.getPeerId(),
            configuration.getChord().getSelf()), false);
} catch(Exception e) {
    Logger.error("checking files", e, true);
}
```

## 3.3 Uilities used for concurrency

### 3.3.1 ScheduledThreadPoolExecutor

To support this concurrency design, we use the `java.util.concurrent.ScheduledThreadPoolExecutor` class to introduce any required delays (in substitution of `Thread.sleep()`), to execute the message processing tasks, to start new connections, etc.

The use of this class eliminates the overhead of instantiating a new thread every time we need to run an asynchronous task, because the threads are instantiated once at the start of the program and reused throughout its execution. It is also better than using `Thread.sleep()` because with sleep the thread wouldn't be able to be reused while the timeout passed and very useful to schedul recurring functions. By using the thread pool the thread is liberated and when the timeout finishes the task is executed in an unoccupied thread (if available, if not it waits in the queue).

Example of the use of thread scheduling to implement the Chord time-recurring functions (line 62 of `chord.Chord`):

```
(...)
    configuration.getThreadScheduler().scheduleWithFixedDelay(() -> {
        try {
            updateFingers();
        } catch(Exception e) {
            Logger.error(e, true);
            Logger.debug(self, "Got exception in update fingers! " + e.getMessage());
        }
    }, configuration.getRandomDelay(1000, 100), 250, TimeUnit.MILLISECONDS);
```

### 3.3.2 AsynchronousFileChannel

To further eliminate blocking calls, we used the `java.nio.channels.AsynchronousFileChannel` when writing to the file system (to store chunks, files or the peer's non-volatile state) so that those

calls wouldn't block the rest of the execution of the program.

This was useful because in this case, whenever we write, we don't need to wait for it to finish, we can just go on with the normal execution of the program. However, for reads, we would need the result of the read almost immediately after the reading call (if it was asynchronous we would have to wait in the same way), so we opted to keep using synchronous reads.

### 3.3.3 ConcurrentHashMap and ConcurrentLinkedQueue

In the places that needed the use of maps/queues and also their concurrent access and modification we used the class `java.util.concurrent.ConcurrentHashMap<K,V>` and `java.util.concurrent.ConcurrentLinkedQueue<E>`.

Example of this is the class `state.PeerState`, which is shared among all threads and supports the Peer's non-volatile state.

# 4 JSSE

For assuring a secure connection between peers, we used the `SSLEngine` class in all communication. The use of SSL/TLS gives integrity protection, authentication and confidentiality. The cipher suite used it negotiated through the handshake, which is done at the start and end of a connection.

Firstly, a client needs to connect (handshake) with the server. Then, both the server and the client use the `unwrap` and `wrap` `SSLEngine` to, respectively, decrypt and encrypt the data that is sent from side to side. Finally, to close the connection, both part handshake for the last time.

Example of the use of the `SSLEngine` `unwrap` and `wrap` functions (lines 113 and 63 of `sslengine.SSLPeer`):

```
(...)
while (peerNetData.hasRemaining()) {
    peerAppData.clear();
    SSLEngineResult result = engine.unwrap(peerNetData, peerAppData);

    switch (result.getStatus()) {
        case OK:
            peerAppData.flip();
            bytesRead = result.bytesProduced();
            break;

        case CLOSED:

(...)

SSLEngineResult result = engine.wrap(appData, netData);

switch (result.getStatus()) {
    case OK:
        netData.flip();

        while (netData.hasRemaining())
```

```
            socket.write(netData);

        Logger.debug(DebugType.SSL, "Sent to the client " + new String(message));
(...)
```

# 5 Scalability

In order to be able to communicate with other peers, the previous project's peers used multicast channels. However, this project uses direct communication from one peer to another, so each peer has to know the destination peer's required information. If every peer was to keep every other peer's information, the service would not be scalable, due to memory usage. In addition to this, every peer would have to update their records every time another peer was added to the service.

In order to prevent this, a Chord network was implemented. In this network, each peer is a node, with a unique id. We used Java's primitive *int* to store this id, which was generated with the SHA-256 algorithm, so our service may have up to $2^{32} = 4294967296$ peers. Despite this, each peer keeps a hash table (finger table) of only 32 entries, with information about other nodes, needing at most 32 messages to perform a lookup (find another node).

Furthermore, to lookup a key we need only `log n` hops on average, which is made possible due to the existence of the finger table, which improves the performance a lot when compared to a simple distributed hash table.

The chord implementation can be found in the `chord.Chord` class.

# 6 Fault Tolerance

The *stabilize* and *updateFingers* methods, which are run periodically, not only serve as maintenance for when new nodes are inserted into the network, but also for when nodes are removed from it. *Stabilize* checks and updates information about successors and predecessors, while *updateFingers* updates the peer's finger table accordingly.

However, our Chord nodes also maintain a list of successive successors, in order to accelerate the process of stabilization in case of unexpected shutdown. Upon successor failure, this list provides the peer with a new successor immediately, even if the finger table was filled with the failed successor's information (which happens in case of very few nodes in the network).

This list is updated in the following way (line 424 of `chord.Chord`):

```
public void updateSuccessorsSuccessors() {
    if (successor == null) return;
    List<ChordNode> newSuccessorsSuccessors = new ArrayList<>();
    for (int i = -1; i < 4; i++)
    {
        ChordNode node = i == -1 ? successor : newSuccessorsSuccessors.get(i);
        try {
            Message reply = SSLClient.sendQueued(node, MessageFactory
                    .getGetSuccessorMessage(getId()), true).get();
            if (reply == null || reply.getNode().getId() == getId()) break;
            newSuccessorsSuccessors.add(reply.getNode());
        }
```

```
            catch(Exception e)
            {
                Logger.error("updating successor's successors
                    (couldn't get successor of " + node +  ")", e, false);
                break;
            }
        }
        synchronized(this) {
            successorsSuccessors = newSuccessorsSuccessors;
        }
    }
```

In the advent that a node leaves the network, all of its entries are removed from the finger tables. Line 446 of `chord.Chord`:

```
    public void peerIsDown(ChordNode node) {
        Logger.debug(self, "In peerIsDown (" + node + ")");
        if (successor != null && successor.getId() == node.getId())
        {
            synchronized(this) {
                if (successorsSuccessors.size() != 0)
                {
                    successor = successorsSuccessors.get(0);
                    successorsSuccessors.remove(successorsSuccessors.get(0));
                }
                else successor = self;
            }
        }
        removeFromFingerTable(node);

        if (predecessor != null && predecessor.getId() == node.getId()) predecessor = null;
    }
```