Distributed Backup Service for Internet
FEUP-SDIS 2020/21 – Project 2

Class 7, group 21

Breno Accioly de Barros Pimentel	(up201800170@edu.fe.up.pt)
Diogo Miguel Ferreira Rodrigues	(up201806429@edu.fe.up.pt)
João António Cardoso Vieira e Basto de Sousa	(up201806613@edu.fe.up.pt)
Rafael Soares Ribeiro	(up201806330@edu.fe.up.pt)

2nd of June, 2021

Contents

Chapter 1

Overview

This report addresses the **second project** of course SDIS (Distributed Systems) at the Faculty of Engineering of the University of Porto (FEUP).

This project aims at designing and implementing a peer-to-peer backup system which guarantees:

- A dynamic network of peers
- Decent performance on storing/restoring/deleting files
- Fault-tolerance
- Safety

We designed an account-based service which resembles many cloud storage services from the end-user's point of view. The system can be joined by starting the PeerDriver executable:

```
java PeerDriver <ID> <RemoteAccessPoint> [<IP>:<Port>]
```

where `<IP>:<Port>` is the socket address of a gateway peer that can be used so the local peer can join the system. This argument is optional because, if the peer wants to create a new system (whether no system existed before or the peer wants to create a new, parallel system), it needs no gateway peer.

The user can interact with the peer object by calling the TestApp class, which connects to the peer running under a certain remote access point:

```
java TestApp <RemoteAccessPoint> <Username> <Password> <Operation>
```

The operation is optional (running the command does nothing, except if the account with that username did not exist, in which case it only creates the account and exits), and can be one of the following:

`BACKUP <Origin> <Destination> <ReplicationDeg>` Backup local file `<Origin>` with remote path `<Destination>` and a certain replication degree
`RESTORE <Origin> <Destination>` Restore remote file `<Origin>` and store it locally to path `<Destination>`
`DELETE <Origin>` Delete remote file `<Origin>`
`DELETEACCOUNT` Delete account
`LEAVE` Causes peer to leave the system in an orderly fashion; the peer process is ended as well

This means a local file can be remotely stored under any name the user wishes, allowing for great flexibility.

Features

A basic, functioning implementation yields 12/20. To reach the 20/20 mark, we implemented the following features:

- (+2) We use thread-based concurrency as much as possible.
- (+2) We use SSLEngine in our final project version.
- (+4) Our project is scalable at the design level (chord) and implementation level (thread pools and asynchronous I/O)
- (+2) We implemented a fault-tolerant solution, with replication degrees, decentralized data storage and decentralized data indexing (user metadata files).

Chapter 2

Protocols

We have devised a system with a significant amount of different protocols; this was so we could reuse as much code as possible, and had the additional advantage that each protocol is quite easy to implement and easy to reason on, as each protocol aims at performing a very concrete (and often simple) task, and it is easy to understand which protocols will use which protocols. It has the inconvenience that the diagram depicting relations between protocols is quite dense.

The peer is allowed to directly use the following protocols:

- Authenticate
- BackupFile
- DeleteFile
- Join
- RestoreFile
- Leave

All communications are made using TCP connections.

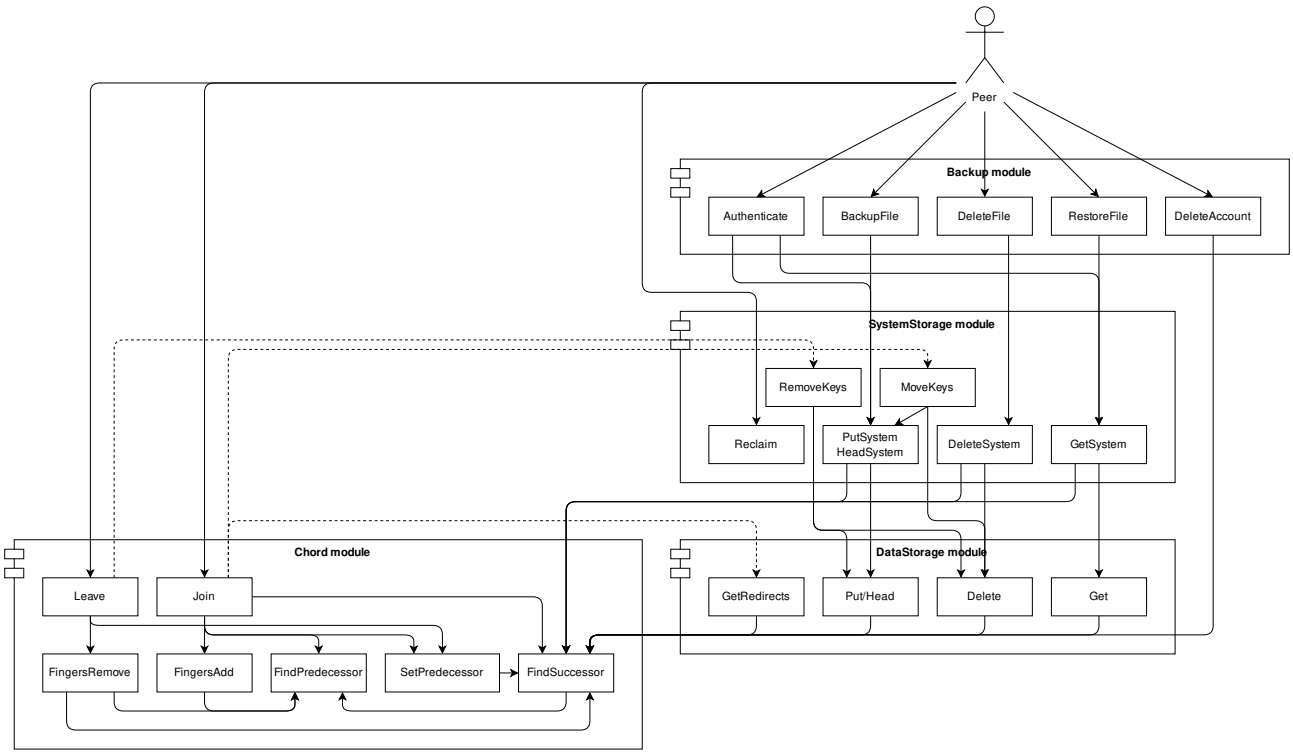

Figure 2.1: Protocols diagram

Definitions

Each chunk/datapiece has a key, which is a binary unsigned number with m bits. This means there are 2^m different keys, from 0 to $2^m - 1$ inclusive. Let $M = 2^m$ be the modulus we are operating with.

The key space can be imagined as a circle of perimeter M, where 0 is in the intersection between the circle and the yy-positive axis, and keys increase in clockwise direction. Given a key k, we can thus say $k \equiv k + i \cdot M, \forall i \in \mathbb{Z})$.

The *distance* between nodes a and b is defined as the number of increments to a that we need to arrive to b in the modulus-M space. That is, $distance(a, b) = (b - a + M) \mod M$. This distance can be interpreted as the distance from a to b in clockwise direction.

The *successor* of a key k ($successor(k)$) is the next node after or at k; technically, $s = successor(k)$ is the node that minimizes $distance(k, s)$.

The *predecessor* of a key k ($predecessor(k)$) is the previous node before or at k; technically, $p = precedessor(k)$ is the node that minimizes $distance(p, k)$.

A node r is responsible for all keys for which their successor is r.

Chord module

This module provides very basic interfaces to use the chord protocol. It allows one to:

- Ask for the socket address and key of the successor of a key
- Ask for the predecessor of a node for which we know the socket address
- Set the predecessor of the current node to a new value
- Inform all nodes that need to know that a new node was added to system and they should update their fingers tables
- Inform all nodes that need to know that a node was removed from the system and they should update their fingers tables

Our implementation closely follows the one provided in (Stoica et al., 2001).

Consider that $x \in [a, b]$ means that $distance(a, x) \leq distance(a, b)$.

Simple messages

`SUCCESSOR` message

Request	Response
`SUCCESSOR`	`<NodeKey> <IP>:<Port>`

Upon receiving this message, the node responds with its locally-stored successor, which is its 0-th finger.

`PREDECESSOR` message

Request	Response
`PREDECESSOR`	`<NodeKey> <IP>:<Port>`

Upon receiving this message, the node responds with its locally-stored predecessor.

FindPredecessor protocol

- **Arguments:** node we are asking to
- **Returns:** predecessor of said node

This protocol allows to find the predecessor of a certain key k.

We know a node n' (with successor s') is the predecessor of k iff $k \in (n', s']$. Thus, until we do not reach that condition, we must update n' for it to get ever closer to meeting that condition. We can do that by successively updating n' with the approximation n' gives for the predecessor of k, through the `CPFINGER` message.

`CPFINGER` message

Request	Response
`CPFINGER <Key>`	`<NodeKey> <IP>:<Port>`

Instructs n' to search, using its fingers table, for the largest finger f that precedes k.

FindSuccessor protocol

- **Arguments:** key
- **Returns:** key and socket address of the successor of that key

The FindSuccessor protocol allows to find the successor of a certain key k. It essentially performs the heavy-lifting using the FindPredecessor protocol, and then sends that node a `SUCCESSOR` message.

SetPredecessor protocol

- **Arguments:** node key and socket
- **Returns:** -

This protocol can be called to set the predecessor of the successor s of a node r to a specific node. It consists of sending a `SETPREDECESSOR` message to the successor s of node r.

`SETPREDECESSOR` message

Request	Response
`SETPREDECESSOR <NodeKey> <IP>:<Port>`	None

Instructs a node s to set its predecessor to the values on the message. This message requires no response, only that the receiving end closes the connection.

FingersAdd protocol

- **Arguments:** -
- **Returns:** -

The joining node r must deduce which nodes need to update their fingers tables. To do that, it will find each predecessor of $r - 2^i$ for $0 \leq i < m$, and notify each of those nodes to the possibility that it might have to change its i-th finger given that r is a possible candidate for that finger, using `FINGERADD` messages.

`FINGERADD` message

Request	Response
`FINGERADD <NodeKey> <IP>:<Port> <FingerIdx>`	None

Instructs a node r receiving this message that it should check if the node f referenced in the message is its new i-th finger. This message requires no answer, only that the socket is closed once the request is fulfilled.

Upon receiving this message, r first checks if the message is instructing it to check if it is itself its i-th finger; because this protocol is only called on joining, and in this case the sending node has already built its fingers table correctly it just ignores the message.

We otherwise arrive at the non-trivial stage, where r checks if f is its new i-finger by testing if $distance(r + 2^i, f) < distance(r + 2^i, k)$; if it's false, just ignore; if it's true:

1. Update $r.finger[i]$ and all indices before i if they also comply to that condition
2. Forward the `FINGERADD` message to its predecessor p without changing it.

One can also check if $p \neq f$ before performing step 2, as it already knows the message will be ignored by p due to the initial check; however, this check avoids an additional TCP connection, thus we do both checks.

A node receiving a `FINGERADD` message currently only closes the incoming socket after having performed all processing, including forwarding the `FINGERADD` message and waiting for that request to be over; this is opposite to the most desirable option, where a node receiving a `FINGERADD` message would immediately close the incoming socket so as not to exhaust the number of TCP connections.

We decided to go with the first option, as it is better for testing, since the FingersAdd protocol will only end once all nodes that had to update their fingers tables are done; otherwise, it is relatively common for tests to begin running before the network stabilizes, and test results are unreliable unless the tests are performed after a sleep period (of about 100ms).

FingersRemove protocol

- **Arguments:** -
- **Returns:** -

This protocol is meant to notify the network that a node r is about to leave, and as such other nodes must update their fingers tables to remove any evidence that r ever existed.

The operation of this protocol is the same as FingersAdd, except it sends `FINGERREMOVE` messages.

`FINGERREMOVE` message

Request	Response
`FINGERREMOVE <OldKey> <OldIP>:<OldPort> <NewKey> <NewIP>:<NewPort> <FingerIdx>`	None

Instructs a node r that, if it has a finger corresponding to the old node f_{old} reported in the `FINGERREMOVE` message, that finger should be replaced by the new node f_{new}, as f_{old} is being removed from the system.

A `FINGERREMOVE` message is processed in a very similar way to a `FINGERADD` message, starting with a simple check if f_{old} is equal to the current node.

In the non-trivial stage, all that changes is that, instead of the condition being related to distances, it merely checks if fingers with a certain index i or less have the same value as f_{old}, and if so they are replaced by f_{new}.

If at least one finger was changed, similarly to `FINGERADD`, the same `FINGERREMOVE` message is forwarded to the current node's predecessor, except if said predecessor is equal to f_{old}, in which case this step is ignored.

Join protocol

- **Arguments:** gateway node
- **Returns:** -

The Join protocol allows a fresh node to join the system.

On join, the joining node r must perform three steps:

1. Initialize its fingers table and predecessor
2. Update the predecessors and fingers tables of other nodes
3. Transfer objects from its successor to itself

The joining node needs to know the socket address of at least one node that is already in the chord; we will call it the gateway node g, as it is the joining node's gateway into the chord while the joining node knows nothing about that.

Initialize fingers table and predecessor

Get predecessor

r gets its successor by asking g for its 0-th finger. Then it sends a `PREDECESSOR` message to its newly-found successor. Because the successor of r was not yet told that r joined the network, it will return what it thinks is its predecessor, when actually it is the predecessor of r.

Build fingers table

To build its fingers table, r asks to g what is the successor of $k = r + 2^i$, for all values of $0 \leq i < m$, using several `FINDSUCCESSOR` messages directed at g.

The joining node can asynchronously make all `FINDSUCCESSOR` requests to g, but that yields $O(m \log N)$ time complexity, while (Stoica et al., 2001) states that, if requests are made sequentially, some trivial tests can be made to skip asking about some fingers. We decided to keep everything asynchronous and ignore this improvement.

Update other nodes

The joining node r needs to notify other nodes in the system to update the information they have about the system. Node r must namely notify its successor that r is its new predecessor, using protocol SetPredecessor. It must also notify some of the other nodes in the system to update their fingers tables with its own key and socket address, by running one instance of the FingersAdd protocol.

Move keys

To move the keys (and perform any other operations the upper layer deems necessary), the Join protocol accepts a runnable, which the upper layer must provide, specifying how to move keys between nodes.

The upper layers are likely to use an instance of the GetRedirects protocol, and another of the MoveKeys protocol.

Leave protocol

The Leave protocol allows a node to leave the system, and is quite similar to the Join protocol.

Let s be the successor of the leaving node r.

On leave, the leaving node r must perform two steps:

1. Update the predecessors and fingers tables of other nodes
2. Transfer objects from itself to its successor

Update predecessors and fingers tables of other nodes

Node r can achieve this by using first the SetPredecessor protocol, with its predecessor's key and socket address.

To update other nodes' fingers tables, r uses the FingersRemove protocol.

Move keys again

The upper layer must provide a runnable to execute whatever actions are required to move the keys from the leaving node to the rest of the system.

The upper layer is likely to use an instance of the RemoveKeys protocol.

Data storage module

This is a very simple data storage protocol, which is tasked with storing, retrieving and deleting datapieces in a specific node, taking into account storage limits of each node. Does not guarantee that a datapiece is accessible if the node storing it is not shutdown in an orderly fashion.

Put protocol

- **Arguments:** a node key, datafile key, data
- **Returns:** boolean indicating success

When the Put protocol is called without a node key, it defaults to the own node's key (because this node key corresponds to the original caller of the Put protocol; if the request wraps around the whole system and comes back to the node that initially called the Put protocol, the node must signal the error); when the Put protocol is called, said node has to save the datapiece using the specified key, using its successors if necessary.

Upon starting this protocol, a node r should perform the following actions:

1. Find its successor s
2. If r has stored the datapiece locally, return true
3. If r has space for that datapiece
    1. Store it locally
    2. If it points to its successor, send a `DELETE` message to s
    3. Return successfully
4. If s is the node that originally asked to put this datapiece, then return false
5. If r does not yet point to s, make it point to s
6. Send a `PUT` message to s
7. Wait for the response, and return according to the message's response

`PUT` message

Request	Response
`PUT <NodeKey> <UUID><LF><Body>`	`<RetCode>`

Upon receiving a `PUT` message, the node should start the Put protocol locally using the node key it got from the `PUT` message (not its own key), and answer the `PUT` message according to what the Put

protocol returns: `1` for successful, `0` otherwise.

Delete protocol

- **Arguments:** key
- **Returns:** -

When the Delete protocol is called for a certain node, said node assumes it has that datapiece, and tries to delete it.

Upon starting this protocol, a node r performs the following actions:

1. Find its successor s
2. If r has not stored that datapiece and does not point to s, return true.
3. If r has the datapiece stored locally
    1. Delete it locally
    2. Return true
4. Send a `DELETE` message to s
5. Wait for the response, and return with the appropriate value according to the response

`DELETE` message

Request	Response
`DELETE <UUID>`	`<RetCode>`

Upon receiving a `DELETE` message with a certain key, a node starts the Delete protocol for itself in order to delete the datapiece with the mentioned key. It responds with the boolean value returned by the Delete protocol it started.

Get protocol

- **Arguments:** key
- **Returns:** data

The Get protocol allows a node to get a certain datapiece by its key, assuming it has the datapiece or that one of its successors has it.

Upon calling the Get protocol locally, node r does the following:

1. Find its successor s
2. If r locally has the datapiece, return the datapiece
3. If r points to its successor
    1. Send a `GET` message to s
    2. Returns according to the response to the `GET` message:
        1. If it fails, it returns `null`
        2. It it succeeds, it returns an array of bytes with the datapiece contents
4. It otherwise fails, and returns `null`

`GET` message

Request	Response
`GET <UUID>`	`<RetCode><Body>`

Upon receiving a `GET` message, a node starts the Get protocol locally, and responds to the message according to whatever the Get protocol returns: if the protocol fails (by returning `null`), the response has a `<RetCode>` of 0 and the body is empty; if the protocol succeeds, the response has a `<RetCode>` of 1 and the body contains the contents of the datapiece.

Head protocol

- **Arguments:** key
- **Returns:** true if found, false otherwise

The Head protocol allows a node to check if a certain datapiece exists; it is basically equivalent to the Get protocol but does not return any data; instead, it returns only a boolean informing if the datapiece is stored or not. If the Get protocol was to answer with a non-null byte array then this protocol answers with true; if the Get protocol was to answer with a null byte array then this protocol answers with false. The advantage is that it does not require expensive data transfers.

`HEAD` message

Request	Response
`HEAD <UUID>`	`<RetCode>`

Equivalent to `GET` message, but returns only the success status.

GetRedirects protocol

- **Arguments:** -
- **Returns:** List of UUIDs to be redirected

This protocol allows a joining node to know which datapieces its predecessor is redirecting to its successor; before r joined, p's successor was s, but now it is r, so r must learn which datapieces it should redirect to s.

When executing this protocol, node r does the following:

1. Find its predecessor p
2. Send a `GETREDIRECTS` message to p
3. Wait for the response, and store the list of UUIDs in an array
4. Save those UUIDs as redirections

`GETREDIRECTS` message

Request	Response
`GETREDIRECTS`	`<RedirectUUID1><LF><RedirectUUID1><LF>...`

s responds to the message with the list of UUIDs of all datapieces that it is redirecting to its successor, so that r can equally point to its successor.

System storage module

Supports the same operations as the data storage module (add, retrieve and delete datapieces) except when called the PutSystem, GetSystem and DeleteSystem protocols do not require to know where the datapieces should be stored according to their keys, and these protocols can be called from any node (while data storage protocols could only be called from the node it was supposed to apply).

It additionally supports protocols to:

- Join the system in an orderly fashion
- Change maximum allowed storage space for the current node
- Leave the system in an orderly fashion

The `PUTSYSTEM`, `DELETESYSTEM`, `GETSYSTEM` and `HEADSYSTEM` messages have the same formats as their corresponding Data Storage messages `PUT`, `DELETE`, `GET` and `HEAD`.

PutSystem protocol

- **Arguments:** UUID, data
- **Returns:** -

Finds the successor s of the key of the given UUID, sends a `PUTSYSTEM` message to s and returns the response to that message.

DeleteSystem protocol

- **Arguments:** UUID
- **Returns:** -

Finds the successor s of the key of the given UUID, sends a `DELETESYSTEM` message to s and returns the response to that message.

GetSystem protocol

- **Arguments:** UUID
- **Returns:** data

Finds the successor s of the key of the given UUID, sends a `GETSYSTEM` message to s and returns the response to that message.

HeadSystem protocol

- **Arguments:** UUID
- **Returns:** data

Finds the successor s of the key of the given UUID, sends a `HEADSYSTEM` message to s and returns the response to that message.

MoveKeys protocol

- **Arguments:** -
- **Returns:** -

Requests the successor s of node r to re-store the keys that no longer belong to s and now belong to r.

`MOVEKEYS` message

Request	Response
`MOVEKEYS <SenderId> <IP>:<Port>`	None

s does the following for each datapiece it is storing with key k such that $distance(k, r) < distance(k, s)$:

1. Load the datapiece from secondary memory.
2. Start the Delete protocol for that datapiece.
3. Start the PutSystem protocol for that datapiece.

Reclaim protocol

- **Arguments:** New storage space
- **Returns:** -

When a node calls the Reclaim protocol, it says it wants to change the memory size it allows the program to use.

If the new storage space is larger than the current storage space being used, the storage space is only increased and nothing more happens.

If the new storage space is smaller than the current storage space being used, then some datapieces will have to be relocated.

To do that, the node r does the following:

1. Change the maximum allowed storage space to match the new storage space
2. While the current storage space being used is larger than the maximum allowed storage space
    1. Pick one random datapiece r is storing locally; assume the datapiece has key k
    2. Load the datapiece contents to memory
    3. Run the DeleteSystem protocol for key k
    4. Run the PutSystem protocol for key k and the corresponding datapiece data

If any of the following operations fails, the protocol fails but it is absolutely guaranteed that the node that called Reclaim will meet its new maximum allowed storage space, even if a datapiece is deleted permanently.

Main module

This module allows the main operations a peer needs to perform:

- Authenticate with a username and password
- Backup a file with a certain replication degree
- Delete a file (including all replicas of each chunk)
- Restore a file

Each user has a username and a password, and can use the system as a distributed remote storage area, where its files are only accessible to itself, and a user can log-in from any peer to perform any operation (backup/restore/delete a file or delete its account). A username may not contain characters forbidden in Windows or Linux file names (`<`, `>`, `:`, `"`, `/`, `\`, `|`, `?`, `*`).

User information is stored in a special *user metadata file*. Each file belongs to one user (including its own user metadata file).

Each file has a replication degree D and a user-defined path p; this defines the location of the file inside the user storage area.

A file has an ID f calculated from the user u that owns it (its *owner*) and p ($f = concatenate(u, \texttt{'/'}, p)$). The ID of the n-th chunk of a file with ID f is $c = concatenate(f, \texttt{'-'}, n)$. Each replication of a chunk with ID c is called a *replica*, and each replica is numbered with a replication index d ($0 \leq d < D$); a replica has ID $r = concatenate(c, \texttt{'-'}, d)$. Thus, the format for a file name is either:

- `<User>/<Path>-<ChunkNo>-<ReplicationIdx>` for a regular file.
- `<User>-<ChunkNo>-<ReplicationIdx>` for a user metadata file.

A replica in the main module always corresponds to a datapiece in lower-level protocols, and the key k corresponding to a certain replica ID r is $k = h(r)$ where h is a hashing function. In our case, we decided to use as hashing function the SHA-256 algorithm (which outputs a 256 bit hash) and use only its first 64 bit to fit it into a Java `long`. Bear in mind that this hash is only needed to find the destination using the chord protocol, not to store the files; these are stored locally using the path

specified by the user, so it is guaranteed that, even if two files have the same hash, they will not overwrite each other.

Authentication protocol

- **Arguments:** username, password
- **Returns:** -

For each user u, the system stores a file with path u with all the metadata about that user:

- Username
- Password (only the password hash is stored)
- A table with the complete list of all files backed-up by that user (the ID of that table is the file path), having for each file its:
  - Path
  - Number of chunks
  - Replication degree

This allows all system data to be fully distributed. This file is stored, loaded, edited and deleted using the BackupFile, DeleteFile and RestoreFile protocols with a replication degree of 10; this means the user metadata file also benefits from redundancy and does not need to use lower-level protocols, including having to calculate the key from its ID. Ideally, this information would be stored in a distributed database, but to limit complexity we decided to use the same file backup system to store user metadata.

Upon starting the Authentication protocol, the peer sends an `AUTHENTICATE` message to any peer (as of now, its successor, because it can always be found in constant time).

`AUTHENTICATE` message

Request	Response
`AUTHENTICATE <Username> <Password>`	`<Metadata>`

Upon receiving this message, a peer starts its `AUTHENTICATE` message processor:

1. Calls the RestoreFile protocol for the user's metadata file
2. If the RestoreFile protocol fails
   1. A new user metadata file is created for that user
   2. The new user metadata file is stored to the system using an instance of BackupFile protocol
   3. Recall the message processor
3. If the RestoreFile protocol succeeds
   1. If the user metadata file is broken, respond with `1` (`BROKEN`)
   2. If the stored hashed password is different from the hash of the password in the `AUTHENTICATE` message, respond with `2` (`UNAUTHORIZED`)
   3. If the stored hashed password is equal to the hash of the password in the `AUTHENTICATE` message, respond in format `0<Body>` (`SUCCESS`) where `<Body>` contains the user metadata file's contents

BackupFile protocol

- **Arguments:** file path, data, replication degree
- **Returns:** -

If a file with the same user and path is already being stored in the system, undefined behavior happens; that's why `Peer#backup` first checks if said file already exists, and if so it returns an error.

The peer first notifies the node that is storing the user metadata file to add said filename to the user metadata, using the message

`ENLISTFILE <Username> <FilePath> <FileID> <NumChunks> <ReplicationDeg>`

to tell said node to add it to the list of files the user backed-up. This step is optional and can be opted-out; for instance, when backing-up a user metadata file you want to backup that file, but there's not really anywhere to enlist that file. And if there was such a system metadata file containing a list of all user metadata files, then who would enlist the system metadata file? This would suggest the existance of an infinite chain of responsibility, which does not make sense.

Then the BackupFile protocol divides the file into several chunks, each with at most 64 kB(64 000 B$), calculates each chunk replica's UUID using the chunk ID and the replication index $d \in [0, D)$, and runs the PutSystem protocol for each replica.

DeleteFile protocol

- **Arguments:** file path, number of chunks, replication degree
- **Returns:** -

If a file with said path is not stored in the system, the protocol fails; that's why `Peer#delete` first checks if said file does not exist by consulting that user's metadata file, and if it does not exist it returns an error.

To delete a file, the DeleteFile protocol goes through each chunk, and each replica, and calls the DeleteSystem protocol for every replica of every chunk of that file.

Then, the peer informs the node storing the user metadata file to remove said filename from the user metadata, using the message

`DELISTFILE <Username> <FilePath>`

to tell said node to remove it from the list of files the user backed-up. Similarly to the `ENLISTFILE` message, this step can be opted-out.

RestoreFile protocol

- **Arguments:** file path, number of chunks, replication degree
- **Returns:** file contents

This protocol cannot work without the replication degree or number of chunks, that's why `Peer#restore` first consults the user metadata file to find that data. If the user does not have that file, the function fails.

The peer consults the user metadata file, and finds how many chunks the file is divided into and the replication degree D; then:

1. For each chunk c:
    1. Initialize a list of not-found replicas L
    2. For each replication index d ($0 \leq d < D$)
        1. If the GetSystem protocol succeeds
            1. Store it
            2. Continue
        2. Store the replica ID $r = concatenate(c, \texttt{'-'}, d)$ in list L
    3. If a replica was not found:
        1. Exit with error
    4. For each replica not found:
        1. Run the PutSystem protocol for that replica, using another replica's contents

In the end, we just need to assemble the chunks.

RestoreUserFile protocol

- **Arguments:** username
- **Returns:** file contents

The RestoreFile protocol requires the number of chunks, but for a user metadata file we do not know that a priori as we have not stored that information anywhere (and if we did, it would have the same problem as described in BackupFile related to hierarchy). Thus, a special method must be applied to restore the user metadata file.

For that, we do a similar processing for each chunk as in RestoreUser, except we wait for each chunk to be finished, and when the peer fails to find any replica of the c-th chunk of the file it assumes the file has no more chunks, and as such returns whatever it has. There is only one exception: if not a single chunk is found the RestoreUserFile protocol fails.

DeleteAccount protocol

- **Arguments:** username, password
- **Returns:** -

The DeleteAccount protocol allows a peer to delete its account, including all files tracked by that account. The account to be deleted is assumed to be the one the user logged-in.

When calling the DeleteAccount protocol, the peer sends a `DELETEACCOUNT` to the peer storing replica 0 of chunk 0 of that user's metadata file.

`DELETEACCOUNT` message

Request	Response
`DELETEACCOUNT <Username> <Password>`	`<Success>`

This message tells the receiver to delete the account. The receiver first gets the user metadata file, then deletes each of the files owned by that user, and finally deletes the user metadata file from the system. It returns 1 if successful, or 0 otherwise.

Chapter 3

Concurrency design

Initial design: futures

As for parallel processing, we initially implemented a solution based on `FixedThreadPool`s and `CompletableFuture`s, which is a simple and comprehensible framework. As usual, futures allow for two distinct types of programming:

- `then`-based programming, where callback functions can be specified, and several callbacks and futures can be chained to do a certain sequence of actions. Several futures can also be chained if the program is required to perform a sequence of asynchronous tasks which depend on each other.
- The synchronous-looking way, which consists of replacing a few return types with futures, and when that future is effectively needed the program can call `get()` in Java (or `await` in JavaScript).

The second option has one major disadvantage: if the futures are being run with a `FixedThreadPool`, when a future calls `get()` on another future the CPU deduces the calling thread is blocked and switches context to do something else, but the resources required to maintain a thread are still being used. This means that, although the process might seem to be using many threads, a large portion of those might be blocked. Thus thread pool usually needs the number of threads it has to be at least one order of magnitude above the maximum parallelism level of the machine (if the CPU can run 8 threads, a good value for the number of threads in the pool is 20-100).

The first option is usually the best, because chaining promises allows, for instance, for the executor to realise that, in a situation where we have `aFuture.then(doSomething)`, the same thread that ran `aFuture` can also run `doSomething` without introducing much overhead and avoiding `get()` calls (which reduces the number of futures that are blocked waiting for other futures).

We tried to frequently return futures so we could chain them as much as possible, but still it was not enough, as some protocols needed to create several instances of lower-level protocols (each instance corresponding to a future) and then wait for all of them to complete. Aside from that, the implementation became rather messy.

Currrent solution: fork-join pools

When processing incoming messages with `Peer.ServerSocketHandler`, we decided to stick with a `FixedThreadPool` solution with 20 threads. However, for everything else (including protocols on the initiator side and message processing on the receiving side) we opted for a simpler solution: `ForkJoinPool` and the associated classes.

This set of classes allows the user to run *tasks*, which are small (but not too minuscule) pieces of code that produce some effect (`RecursiveAction`) or result (`RecursiveTask`). This framework is specially designed with recursive tasks in mind, since a task can internally create other, smaller tasks. A task can either be run with `invoke()`, which causes its code to run in the current thread as if it was a

regular `Supplier`, or it can also run in parallel if the main task calls `fork()` for the task to start running in parallel, and later `join()` to wait for their completion (and if subtasks have end-products they can be retrieved using `get`; this function can also be called even if the task was not forked before, as it will automatically fork); a collection of tasks can be collectively forked by calling `invokeAll` from inside the main task, and then each task can be joined individually.

A `ForkJoinPool` keeps, for each task, a queue of subtasks it summoned, and each task is assigned a thread. If a task blocks waiting for a subtask, the thread a starts running the subtask; if another thread b does not have anything to do and notices a is struggling to keep up with a large number of queued tasks, b *steals* work from a and starts running some of those subtasks itself, so that when a joins the subtasks, some of them have already been processed by b and thus do not require a to block nor process them. This is called *work stealing* .

There are two main advantages with this approach:

- `ForkJoinPool` takes care of creating new threads and deleting threads that have not been active for a while (usually 1 minute), which means we no longer have to estimate how many threads are enough.
  - `ForkJoinPool` also notices when a task blocks waiting for a subtask to be completed, and that thread starts processing that task, instead of the future approach where the calling thread blocks and another thread is allocated to the subtask/subfuture.
  - Additionally, the class `ForkJoinPool.ManagedBlocker` can be implemented to tell the pool that that specific task blocks waiting for something, and as such the pool must create a new thread to replace the thread that will block. This is useful when writing to files; even if we are using NIO classes for reading and writing, we need to convert the `CompletableFuture`s it uses into tasks, which we do by calling `get()` on those futures, thus making them blocking. This has two advantages: the reading is still non-busy (because the blocking task is blocked and the CPU knows that) and we can integrate the NIO classes into our `ForkJoinPool` framework.
- Code can be run synchronously (`invoke`) or in parallel (`fork`/`invokeAll` and `join`/`get`) at the programmer's discretion.

This gives us more control over the whole process, as well as keeping the code tidier. We additionally observed a 15% decrease in execution time of automated tests, which is explained by less thread blocking and better thread management (as threading is mostly managed by the `ForkJoinPool`).

The framework already creates a process-level static `ForkJoinPool`, which means we do not need to worry with creating a pool of our own (although we could).

Chapter 4

JSSE

We implemented two versions, one using the SSLSocket and another with the SSLEngine. Both versions are slower than the non-encrypted version of the project, as expected.

SSLSocket

Coming from a `Socket`-based implementation, using SSLSocket is straightforward: we passed the certificate JVM parameters via the system properties and started the handshake before a connection, closing it when it's done.

SSLEngine

For the SSLEngine implementation, we created two classes, `SecureSocketChannel` (extending `SocketChannel`) and `SecureServerSocketChannel` (extending `ServerSocketChannel`), that handles the negotiation process, exchanging the protocol parameters, writing/reading from a socket channel and correctly closing the sockets.

In a `Socket`-based implementation, because each side of the TCP connection was only used to send information once, the end of a message could simply be signalled by closing that end of the TCP connection. But with TLS it is more arduous to make it work correctly, so we used a special flag byte (`FLAG`, `0x7E`) to signal the end of a message. This flag would be placed in the app data buffer after the user-provided information. This flag is then decoded at the receiving end and used by the `doRead` function to exit the `doRead` cycle.

Because there is a risk (even if minute) that the message body contains a flag byte, we used a byte stuffing mechanism to unambiguate that situation; in this strategy, an escape byte (`ESC`, `0x7D`) is defined and all data bytes b equal to `FLAG` or `ESC` are encoded into two bytes, the first being `ESC` and the second being $b' = b \oplus$ `0x20`; thus, during the unstuffing process, if `ESC` is found, the next character is decoded with $b = b' \oplus$ `0x20`, as the relation $x = (x \oplus y) \oplus y$ is trivial.

Issues with SSLEngine

Although `SSLEngine` is frequently referred as being *flexible*, it is so flexible that it is nearly impossible to use. The documentation relative to `SSLEngine` is very scarce, examples are short, incomplete and rather useless. Besides, the workings of the `SSLEngine` are impenetrable.

The fact that using `SSLEngine` is a requirement to reach the maximum project grade is rather concerning, as it does not present much of an improvement over `SSLSocket` in this project. Also, it requires a very large amount of time to study, implement and debug when compared to the relatively small payoff of 1/20 in the project grade, given it is a 4-people project implemented over about 7 weeks during a semester with four more curricular units. It is thus hard to understand why this is a requirement to

reach maximum grade, unless the goal is to limit the number of people that reach that goal, or simply deter the students from paying any interest to this project.

We nevertheless managed to implement this feature.

Chapter 5

Scalability

Design level

We fully implemented the Chord protocol (`Chord` module), with keys having 62 bit, which we chose as it fits in a 64 bit signed integer (`long`), and does not use the corner cases where numbers are close to the limits the types can represent.

Implementation level

As we described in-depth in section 3, we are using Java NIO to read/write files, a thread pool to process incoming messages and a `ForkJoinPool` to manage protocols on the initiator and receiver sides.

Chapter 6

Fault-tolerance

We implemented provisions for two types of peer shutdown.

Gracefully leaving

A peer is said to *gracefully leave* the system if it is allowed to perform some actions before leaving the system. This consists of informing some nodes of the system that it is about to leave (and as such that they must update their predecessor/fingers), as well as transfer its keys and files to its successor. This is the expected behavior of a peer that is being shutdown normally for maintenance, if the network is being reorganized or if there is a power outage and the peer has a few minutes before backup batteries run out of energy.

This case is quite trivial to solve. The leaving peer u must do what we just mentioned, consisting of:

1. Informing the successor of u that its precedessor is no longer u, but u's predecessor.
2. Informing some nodes that they should change their fingers tables if they have u as one of their fingers.
3. For each file it is storing, delete it locally and re-store it in the system (since all other peers' tables have already been updated, the system no longer knows that u exists, and will store the datapieces somewhere else, ignoring u in the process)

Finally, u cleans up any remains of its activity from disk and exits.

Abnormal leaving

A peer is said to *abnormally leave* the system if it does not have time to perform some or all actions it needs for the leaving to be considered graceful. This is the expected behavior of a peer that finds an exception, or is suddenly shutdown before leaving or even if it was in the middle of leaving. This is a problem because the information in that peer was not transferred to the rest of the system nor is there a guarantee as for when the peer will rejoin the system, or if it will rejoin at all.

To solve this problem, we first assume that, once a peer leaves the system, the data it was storing is definitely lost. This allows a simpler analysis of the problem. As such, if a peer is rejoining the system, it must first delete any remnants of previous runs of that peer in the current system. Also, if a peer that the system thought existed fails to respond to a request, the system assumes the peer has unexpectedly shutdown and assumes all data it had stored as permanently lost.

This means the `SystemStorage` module does not guarantee data is not lost and might fail to retrieve some datapieces that were successfully stored before. This problem is solved by the upper layer (`Main`), which replicates the same chunk several times, and as such can retrieve another replica of the same chunk, and latter re-store the replica it failed to find.

To check if another peer is operating, the current peer simply opens a socket connecting to that peer; this will fail if the peer does not accept the connection, as we are speaking of TCP connections which require a handshake. Ideally, this socket will then be supplied to the protocol that needs it, but if the only goal is to check the peer exists and get its information (without needing a socket), a `HELLO` message will be sent, with the only goal of telling the receiving end to respond with a `1` and ignore (otherwise the receiving end could get confused as to the type of the message, as it would otherwise be empty).

At the chord level

How to achieve chord correctness

Assuring successor correctness

No mather what, each node must aggressively maintain an ordered list of N successors, in case a few of those successors fail without notice. The ClosestPrecedingFinger algorithm tries to return the largest finger that precedes key k, but if it fails to find that finger it must go on to lower-index fingers until it reaches finger 0; if even so the finger corresponds to a node that does not exist, the algorithm must go on to explore the ordered list of N successors and pick the first valid successor, so it can keep making progress. Thus, if we assure the successors list is somewhat correct we can also assume that FindPredecessor and FindSuccessor are working correctly.

To assure this, each node keeps a list L of at most $N = 10$ successors. If the number V of nodes in the chord is less than N the list only has V elements, and if $V > N$ the list has N successors. A node n periodically corrects L by finding, in order, the first element of L that is valid; then it considers that node to be the successor and the first element of the list s_0, and asks that node about its successor s_1 using a `SUCCESSOR` message; then n asks s_1 about its successor s_2, and so on, until it knows the information about s_{N-1}.

When joining the system, after the joining peer n performs all usual steps, it will send `NTFYSUCCESSOR` (*notify successor*) messages to its at most N predecessors, telling them that n joined the system and that they might want to consider the possibility of adding n to their successors lists. When leaving the system in an orderly fashion, the leaving peer sends `UNTFYSUCCESSOR` (*unnotify successor*) messages to its at most N predecessors, telling them to remove n from their successors lists if n is in those lists; currently this message triggers a full re-computation of the successors list.

Assuring fingers correctness

Since we already assured FindSuccessor to be working, we can also recalculate our fingers when needed.

Assuring predecessor correctness

Predecessor correctness can be trivially assured by calling the FindPredecessor protocol for the current node.

Corrections

We implemented mid-operation corrections to allow corruptions to be fixed while performing a certain action. For instance, say a peer needs to connect to a finger, but that finger is offline; when the peer opens the connection with another peer it notices the other peer did not accept the connection. Instead of aborting the protocol and having to wait for periodic consistency checks, the peer launches an instance of the FindSuccessor protocol to fix that finger, and only then does it return that finger.

Periodic consistency checks: FixChord protocol

We run the chord consistency checks every 10 s. This protocol:

1. Recalculates all N successors.

2. Gets all fingers, which will trigger a finger fix for each finger that is offline.
3. Update predecessor.

At the file level

Corrections

We have already described on the section about the Main module how we perform corrections of file inconsistencies (i.e., if we fail to find a replica of a certain chunk, we re-store it).

Periodic consistency checks: FixMain protocol

We run the data consistency checks every 1 min. This protocol consists of:

1. Identifying all the users for which at least one replica of a chunk of their metadata files is stored locally.
2. For each user, get its user metadata file using an instance of the `RestoreFile` protocol
3. For each user, for each file the user has stored, call the FixFile protocol.

The FixFile protocol works similar to the RestoreFile protocol, but instead uses mostly HeadSystem messages to check the replicas exist; if not, it retrieves one of the reachable replicas and uses that to restore all the missing replicas of that chunk.

Bibliography

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, *31*(4), 149–160. https://doi.org/10.1145/964723.383071