FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	SFTP Proxy for AWS S3
Student:	Bc. Matej Matula
Supervisor:	Ing. Tomáš Janeček
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

SFTP is the most widely used open-source standardized protocol to exchange files, it is based on the SSH protocol and communicates in streaming fashion.
AWS is one of the most prominent cloud providers and its Simple Storage Service (S3) is object storage (file storage) exposed via REST API.

The goal of the thesis is to design and implement a Java service (using existing open-source libraries) and which will expose an SFTP endpoint, backed by AWS S3.
It should be possible to connect to the service using any SFTP client and upload, download and list files and directories stored in AWS S3.
The service should be configurable to map different AWS S3 paths to different users. Users will be authenticated by the public key.
The service should be published under MIT license on GitHub or a similar platform.

One of the main challenges will be to come up with an efficient way to bridge the streaming SFTP protocol with request/response based AWS S3 REST API.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 8, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Sftp proxy for AWS

Bc. Matej Matula

Department of Software Engineering
Supervisor: Ing. Tomáš Janeček

July 30, 2020

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on July 30, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Matula, Matej. *Sftp proxy for AWS*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Pokrok v technológii, najmä v oblasti cloud computingu, tlačí čoraz viac používateľov k využívaniu svojich služieb. Používatelia používajú svoje úložné servery už mnoho rokov. Prechod na úložisko typu cloud nemusí nutne znamenať, že používatelia musia prestať používať svojich obľúbených klientov SFTP alebo prepísať časť kódovej základne, ktorá s ňou komunikuje. Táto práca sa zaoberá vytvorením služby okolo úložiska S3 Amazon Web Service, ktoré funguje ako server SFTP. Týmto spôsobom môžu klienti/procesy pokračovať v prístupe k súborom uloženým v S3 pomocou existujúceho protokolu SFTP s existujúcimi metódami autentifikácie SFTP, takže prechod do cloudu nie je zrejmý.

Klíčová slova AWS, amazon-web-services, S3, sftp

Abstract

The progress in technology, especially in cloud computing pushes more and more users into using their services. Users have been using their storage servers for many years. Moving to cloud-based storage doesn't necessarily mean users have to stop using their favorite SFTP clients or rewrite part of the codebase

that handles communication with it. This thesis deals with creating a service around Amazon Web Service's S3 storage that acts like SFTP server. This way, client/processes can continue to access the files stored in S3 using the existing SFTP protocol with existing SFTP authentication methods making the transition to a cloud seemless.

Keywords AWS, amazon-web-services, S3, sftp

Contents

List of Figures

List of Tables

Introduction

As the technology progresses, so raises the access to the information and the reason to store the information. With this, the storage requirement for that information has also increased. The amount of data stored can be so big, users need to use multiple storage servers. For this reason, many companies moved to cloud services, with unlimited online storage capacity. Cloud storage provides a web user interface and API allowing the users to manipulate or communicate with cloud or manipulate with their stored data. Users usually communicate with their storage servers using SSH File Transfer Protocol (SFTP), thus moving to a different storage service can lead to broken or dysfunctional production code, scripts, or API. Some cloud services provide SFTP endpoint, but it is not free. Users, paying for cloud services, need to pay extra for SFTP endpoints. Teams using cloud services need to manage access to particular storage, create and grant access to members of the team individually using on the level of their cloud service. In some use cases, it is preferred to map the stored data into a custom directory structure for better clarity or more explicit meaning of the content of data. Using official SFTP endpoints, users are obliged to stick with the directory structure of cloud service and their way of granting access to specific storage. This thesis was created to fulfill the need of custom mapping and creating users with specific permissions on the level of applications. In section "Objective" I talk what is the aim of the thesis and describe what solution could be used as a replacement. In section "Concepts" I explain and describe terms that I use in this thesis, so the reader will be familiar with them. In section "Technologies" I describe technologies that were used to create and/or test this application. In the section "Analysis and design" I explain terms used in analysis and identify requirements for the creation of this application and describe them using use cases and user scenarios accompanied by activity diagrams for each user scenario. In section "Implementation" I describe the process of creating the application, explaining the structure, dependencies, and relationships between each package while also describing the thought process behind the work. In the section "Perfor-

mance" I demonstrate the times it takes to upload and download files using the application and compare it to the default way of uploading and downloading files to AWS. In section "Testing" I explain the way testing was done and explain the idea behind the test. In "Conclusion" I write my thoughts of this work and future possibilities that can be achieved with this work and possible adjustments or expansion of this work.

Chapter 1

Objective

The objective of this thesis is to create a proxy for AWS that acts as an SFTP server and store the underlying files to AWS S3 behind the scenes. Additionally, the application should support custom mapping between AWS S3 bucket and the SFTP file tree with configurable permissions at a user account level.

1.1 State of art

Currently, there are not many solutions that could be used. Big corporations tend to use official endpoints that come with fees. The fees vary from cloud to cloud. Various cloud services charge various fees with various ranges of price. Amazon web services currently provide one service for integrating S3 storage with SFTP servers.

1.1.1 Transfer family

"The Amazon Web Services provide official and fully managed support for file transfers directly into and out of Amazon S3. It supports Secure File Transfer Protocol (SFTP), File Transfer Protocol over SSL (FTPS), and File Transfer Protocol (FTP), you select the protocols, identity provider, and endpoint configuration to enable transfers over the chosen protocols".[1]

This service is not free of charge. You pay for every hour the server is running and for every gigabyte you(or your team)upload or download. The pricing table for SFTP protocol is shown in the table below:

Name of the service	**price**
Time SFTP is enabled on your endpoint	$0.30
SFTP data uploads	$0.04 per gigabyte (GB) transferred
SFTP data downloads	$0.04 per gigabyte (GB) transferred

Table 1.1: transfer family cost table

Using the example listed on the official site, the cost of SFTP server running for one month with 1gb trasnfered:

Cost of endpoint	$0.30 * 24 hours * 30 days = $216
Cost of data transfer	$0.04 * 1 GB * 30 days = $1.20
total	$216 + $1.20 = $217.20.

Table 1.2: transfer family cost table

Chapter 2

Concepts

In this section I describe and explain terms used in this thesis.

2.1 Network protocol

The easiest and the most straightforward way how to explain what network protocol stands for is as predefined rules that describe how to format, send, and receive data. Network protocols allow communication between two sides by following these predefined rules. “In a sense, protocols are to communication what algorithms are to computation. An algorithm allows one to specify or understand a computation without knowing the details of a particular CPU instruction set. Similarly, a communication protocol allows one to specify or understand data communication without depending on detailed knowledge of a particular vendor’s network hardware.”[2]

2.2 SSH

”Secure Shell (SSH) is a cryptographic network protocol for operating network services securely over an unsecured network.”[3] Networks services can be secured with SSH. SSH operates on the client-server model, which means SSH client connects to the SSH server over a channel. In order for connection to be established, the remote machine must run software operating the SSH server(usually called SSH daemon), which listens on a specific network port, authenticate incoming requests, and handles the communication. SSH server Applications using SSH provides a command-line interface, log in to server and remote command execution on the server.SSH server provides numerous ways of authentication including the most basic way - using a password, and the most common way - public key authentication.

2.3 Public-key cryptography

Public-key cryptography, or asymmetric cryptography, is a cryptographic system that uses public keys, which may be disseminated widely, and private keys, only known to user. The generation of keys depends on cryptographic algorithms based on mathematical problems to produce one-way functions. For maximum and effective security, the private key must be kept private; the public key can be openly distributed without compromising security.[4] The public key of the receiver is used by the sender to encrypt the message sent over the channel and the private key of the receiver is used by the receiver to decrypt the message encrypted by his public key.

2.4 SSH public key authentication

In order to authenticate using the public key, the user needs to generate a key pair - private and public key. The public key must be copied into a file within the user's home directory, to `~/.ssh/authorized\_keys`.

When users want to authenticate to the SSH server, the server checks its `authorized\keys` file for the public key, generates a random message, and encrypts it with the user's public key. Client, upon receiving the encrypted message from the server, decrypts the message using his private key, adds session ID which was previously negotiated and generates MD5 hash from this value, and sends it back into the server. The server knowing the session ID and sent message compares MD5 hashes and decides if the client is in possession of the private key or not, and thus authenticating him or not.

2.5 SFTP

"A file transfer utility provides a means of exchanging files between two computer systems sharing a network. This involves copying a file from a file system on one computer running some operating system to the file system on another computer running another operating system."[5] SSH File Transfer Protocol (or Secure File Transfer Protocol) is a network protocol that provides file access, file transfer, and file management over any reliable data stream. This protocol has currently 6 versions and requires authentication to the server it is connected to, usually runs over SSH session. SSH provides a secure channel between client and server (or remote computer). We shoudln't confuse SFTP with FTP run over SSH. It is new protocol designed from the ground up by the IETF SECSH working group. Simple File Transfer Protocol is often confused with SFTP.[6] Unlike FTP (File transfer protocol) SFTP transfers are over the control channel, thus there is no need to open another data channel in order to complete the file transfer. In this time, SFTP is pretty much considered as a standard for secure data transfer.

2.6 Cloud computing

Cloud computing is the on-demand delivery of IT with pay-as-you-go pricing. User does not buy, own, and maintain physical data centers and servers, instead user can access technology services, for example computing power, storage, and databases, on an as-needed basis from a cloud provider like Amazon Web Services (AWS).[7] Cloud computing enables organizations or other users to obtain a flexible, secure IT infrastructure with , in same way that national electric grids enable homes to plug into energy source.[8] In other words, it means accessing, storing, and manipulating data and programs over the internet on the remote machine instead of on the local computer. According to researches supporting cloud services in applications accounts for more than a third of all IT spending worldwide.

2.7 REST

The REST Web is the subset of the HTTP based World Wide Web, in which it provides uniform interface semantics, create, retrieve, update and delete, and manipulate resources only by the exchange of representations and messages.[9] The REST communication is stateless, meaning each communication does dont depend on state of conversation or application. Each exchanged message is standalone message.

2.8 API

Application Programming Interface(API) is software intermediary, that allows two applications to talk to each other. For example client programs use API to communicate with web services. In other words, API exposes a set of data and functions for better communication.

2.9 High-availability

Systems that provide nearly full-time availability are known as high availability systems. These are the systems that provide their functionality even after a failure occurs in the system, either hardware or software. The most common way of providing High availability is to have duplicated system components. If one becomes unable to provide its functionality another one can be used, there is no single point of failure.[10]

2.10 Eventual Consistency

Eventual consistency(or optimistic replication) is a consistency model used in distributed computing to achieve high availability that informally guarantees

that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.[11]

The idea behind eventual consistency is quite simple it consists of 3 steps:

- Replica data across all participans(every node of server).
- On each participant perform updates locally.
- Propagate local updates to others participants.

Nodes must be configured to handle conflicted updates consistently so the behavior will be deterministic. Eventually, consistent systems may return any value before all values across every participant are propagated and system converges.

2.11 Amazon web service

Amazon web service(AWS) is easy to use computing platform offered by Amazon. It offers flexible and scalable cloud computing services. AWS uses the "Pay-as-you-go" model, meaning user's fees are calculated based on multiple factors. The user can select and configure hardware, OS, software, networking features required availability, redundancy, security, and service options. It is a combination of infrastructure as a service, platform as a service and software as a service offering.

2.12 AWS S3

"Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance."[12]. It is used to store, retrieve, or manipulate with any amount of data remotely. Any file can be uploaded or downloaded from AWS storage.

The concept of S3 revolves around Buckets and Objects. The bucket is a container for objects. There can be multiple buckets for the user account. Everything that is stored inside S3 is an object. This object has meta-data attached to it (name of the object, size, and date). Every object needs to be placed inside the bucket, where it is identified by a unique identifier (key). Objects also include their Access Control list that indicates whether object can be shared across the internet or not. As of today, AWS S3 offers various features:

- Storing objects up to 5 TB in size.
- Read and write entire object.
- Every object has a unique developer assign key.

- Collect objects into bucket.
- Every object has a unique URL.
- Full control of access rights.
- Eventual consistency data model.
- Supports versioning.

2.12.1 AWS S3 - Bucket

A bucket is a container for every objects stored in Amazon S3. Bucket cannot contain another bucket. Every object stored on AWS S3 must be contained in a bucket.[13]

Buckets serve several purposes:

- They organize the Amazon S3 namespace at the highest level.
- They identify the account responsible for storage and data transfer charges.
- They play a role in access control.
- They serve as the unit of aggregation for usage reporting.

Buckets can be configured to be created in specific region.

2.12.2 AWS S3 - Objects

Objects are fundamental for Amazon S3. They containts object data and meta data.[13] The data portion is opaque to Amazon S3. The metadata is a set of name-value pairs that serves as the way to describe the object. These include some default metadata, such as the date last modified, and standard HTTP metadata, such as Content-Type. Custom meta-data can be declared too.[13]

Object are uniquely identified inside a bucket they belong to.

2.12.3 AWS S3 - Keys

A key is a unique identifier for an object stored inside the bucket. Every object can have only one key. The objects inside AWS S3 are uniquely identified by bucket, key, and version ID of said object.

2.12.4 AWS S3 - Regions

Regions represents the geographical AWS Region where buckets will be stored. Regions can be chosen to reduce latency or cost. Objects stored in a Region never leave the Region unless you explicitly transfer them to another Region. For example, objects stored in the Europe (Ireland) Region never leave it.

2.12.5 Amazon S3 data consistency model

Amazon S3 provides read-after-write consistency for PUTS of new objects in S3 bucket.However if a HEAD or GET request to a key name before the object is created is made, and object is created after that request a subsequent GET might not return the object due to eventual consistency.

Amazon S3 offers eventual consistency for overwrite PUTS and DELETES in all Regions.

Updates to a single key are atomic. If PUT is used on existing key, a subsequent read might return the old data or the updated data. However it will never return corrupted, or partial data.

Amazon S3 achieves high availability by replicating data across multiple servers within AWS data centers. If a PUT request is successful,data is safely stored. This change must be first replicated across Amazon S3, before the replicaiton is completed following behavior can be observed.

- A process writes a new object to Amazon S3 and immediately lists keys within its bucket. Until the change is fully propagated, the object might not appear in the list.
- A process replaces an existing object and immediately tries to read it. Until the change is fully propagated, Amazon S3 might return the previous data.
- A process deletes an existing object and immediately tries to read it. Until the deletion is fully propagated, Amazon S3 might return the deleted data.
- A process deletes an existing object and immediately lists keys within its bucket. Until the deletion is fully propagated, Amazon S3 might list the deleted object. [13]

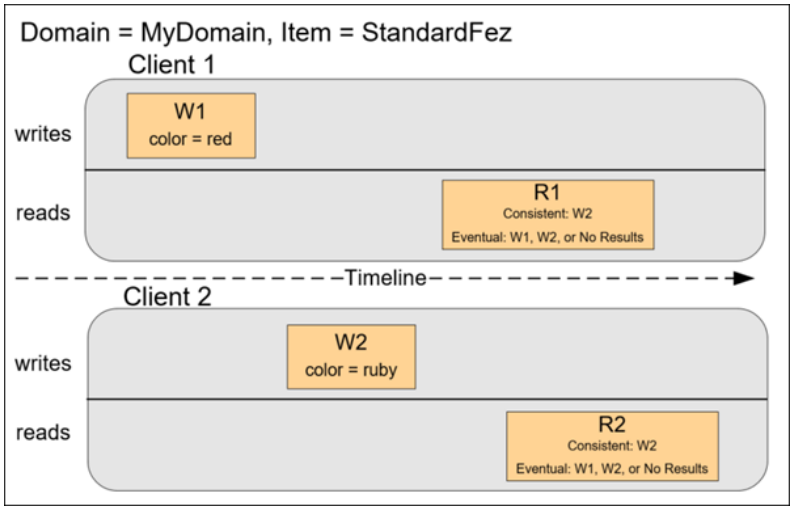

Figure 2.1: data consistency model
Source: https://docs.aws.amazon.com/

2.12.6 Proxy

A proxy server is a server application or appliance that acts as an intermediary for requests from clients seeking resources from servers that provide those resources.[14] In other words, the proxy server is masking the origin of the request to the resource server and masks the origin of the requested data to the client. Client, instead of communicating directly with the resource server communicates with the proxy server that handles retrieving requested resources from the resource server and sending it to the client.

Chapter 3

Technologies

3.1 Java

Java is a general-purpose programming language. Java is class-based and object-oriented. It is intended to let developers write once, run anywhere (WORA),[15] meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.[16]

Java was selected for business reasons, as this application should be part of bigger systems written in java.

3.2 GitLab

"GitLab is a web-based DevOps lifecycle tool that provides a Git-repository manager providing wiki, issue-tracking and continuous integration/continuous deployment pipeline."[17]

3.2.0.1 SSHD MINA

Apache SSHD is a 100% pure java library to support the SSH protocols at client and server side. This library is based on Apache MINA, a scalable and high performance asynchronous IO library.

SSHD's purpose isn't to replace the SSH client or SSH server from Unix operating systems, but to provides support for Java based applications requiring usage of SSH.[18]

3.2.1 JSCH

JSch is a pure Java implementation of SSH2. JSch allows user to connect to an sshd server. It can be used as client side of the sftp.

3.2.2 Maven

Maven is tool that is used for automation of build in mainly for Java projects. Maven resolves around concept of a Maven Build Lifecycle. A lifecycle is composed of one or more sequential phases in a fixed order.[19] Maven mainly solves two problems of building the software project. How the software is build, and its dependencies. order (imposed by the Maven designers).

Chapter 4

Analysis and design

Analysis and design is a crucial step of software development, that assures we are aware of criteria required by the development and we meet expectations of quality of developed software. It also helps to transform specifications into implementation - human-readable specifications into code.

4.1 Requirements

The software requirements are a description of software functionalities and capabilities. They describe how the act, appear, or perform.

4.2 Requirement analysis

In software engineering, requirements analysis has goal to determine the needs to meet the criteria of the new product or project, taking account of the possibly conflicting requirements , analyzing, documenting, validating software requirements.[20]

The results of the requirement analysis should be the answer to the question "what software must do "

A side product of requirement analysis are use cases which are usually identified and created during requirement analysis and help to describe functional requirements in use case scenario

4.3 Functional requirement

Functional requirements explain what has to be done by identifying the necessary task, action, or activity that must be accomplished. Functional requirements analysis will be used as the top-level functions for functional analysis.[20]

Following functional requirements were identified during requirement analysis:

- Adding layer of communication with AWS S3 - application must be able to communicate with AWS S3 with given credentials for authorization with user still using SFTP protocol
- Defining user accounts on application level - application must be able to define users account, given credentials and connection to AWS S3
- Deciding if user account acts as typical SFTP server or communicates with AWS S3 - application must allow user to choose, if given account is account communicating with AWS S3 proxy with given credentials, or is account that communicates with remote machine, thus acting as basic SFTP server
- Adding mapping of bucket to custom folder and adding permission - application must be able to specify custom mapping of AWS S3 bucket to selected directory structure, thus allowing more explicit meaning of stored data and add permission to those mappings

4.4 Non-functional requirements

Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system, rather than specific behaviors.

These non-functional requirements have been identified during analysis:

- Secure authentication - application must use a secure way of authentication - public key authentication instead of less secure password authentication
- User cannot notice that it is not communicating with SFTP server - the communication with AWS S3 must be hidden from the user
- The private key of the server must be loaded from the classpath

4.5 Use cases

Use cases are actions that describe the interaction of a defined role within the application. As mentioned earlier, use cases were identified during requirement analysis.

Actor(or a role) "specifies a role played by a user or any other system that interacts with the subject."[21] We can distinguish two different user roles in this application. They divide the functionality of the application based on the selected user role.

- Owner/Admin
- User
- The owner/Admin is the person running the application. Owner/Admin's purpose is to configure the application and to create users and adding permissions to them, thus allowing other user roles to fully use the functionality of the application.
- The user is a target user of the application, that uses the functionality of this application to communicate with a remote SFTP server or AWS S3 storage.

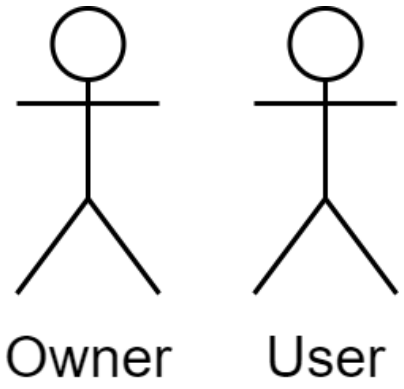

Figure 4.1: Diagram of actors

According to identified actors, we can assign use cases to each actor

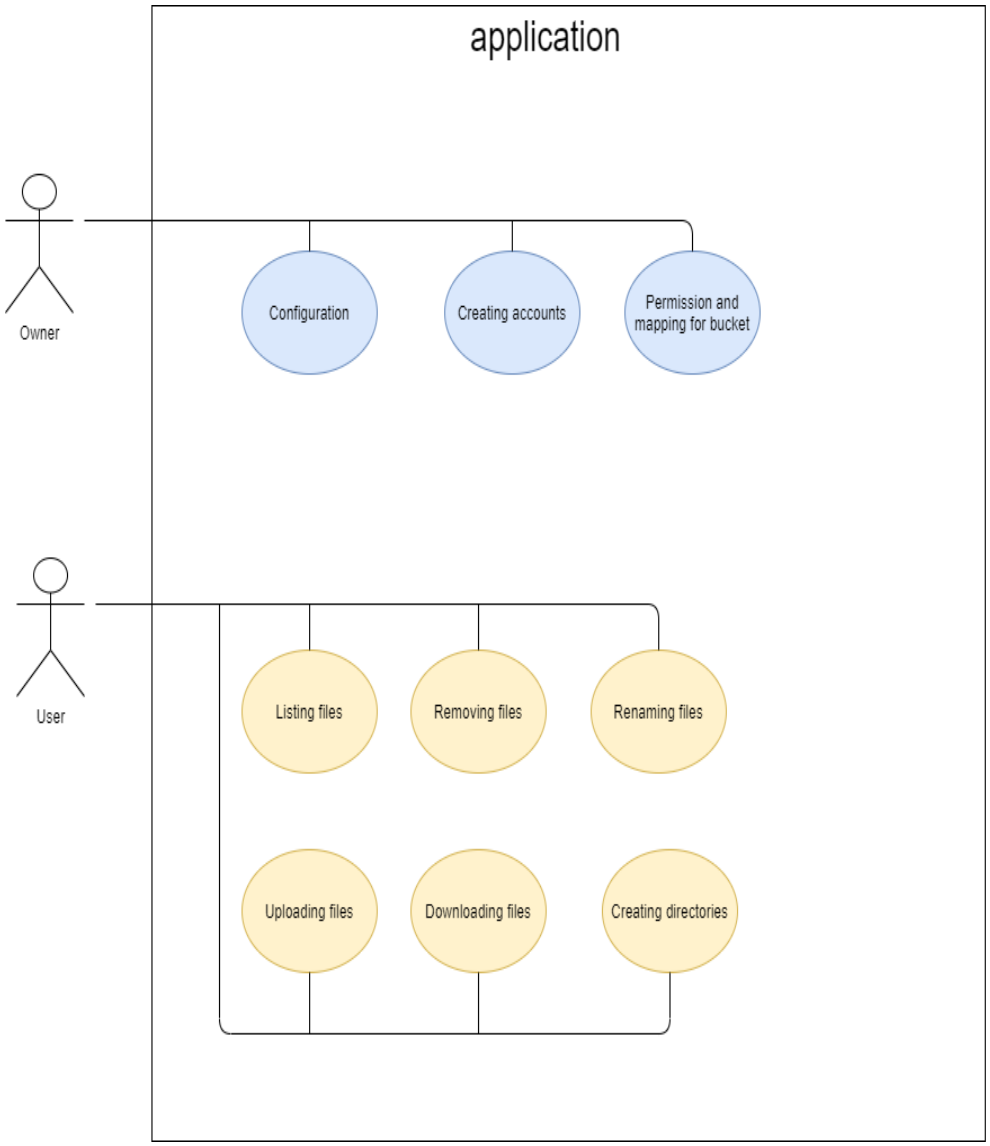

Figure 4.2: Use cases

4.5.1 Owner

4.5.1.1 Configuration

The owner configures the whole application using in-application config. The owner sets the running port of the application and buffer sizes for downloading and uploading files to AWS S3 storage. The owner can configure if the upload and download will be paraller or not, and number of threads that will be used.

4.5.1.2 Creating accounts

The owner creates user accounts, he adds user name and a public key which serves for public-key authorization. Depending on what type of account of what account owner wishes to create, either account that communicates with AWS or account that acts as typical SFTP accounts, the owner creates an instance of the selected user account.

4.5.1.3 Permission and mapping for bucket

While creating an account, the owner sets mapping for the bucket. He sets what bucket from passed AWS S3 instance should be mapped on directory structure he chooses, adding permissions access to this bucket. Let's say we have following buckets in AWS :

bucket1
bucket2

The Owner can choose the directory structure where this buckets will be mapped to. For example he can do following mapping:

bucket1 —> /dir1/dir2/bucket1
bucket2 —> /dir1/dir2/bucket1/bucket2
In the SFTP client, all files in bucket1 will have prefix
/dir1/dir2/bucket1
added to their path
and all files in bucket2 will have prefix
/dir1/dir2/bucket1/bucket2
added to their path.

4.5.2 User

4.5.2.1 Login

User logs in into the application using his credentials.

4.5.3 Listing files

User can list files in directory.

4.5.4 Removing files

User can remove file.

4.5.5 Renaming files

User can rename file.

4.5.6 Moving files

User can move file.

4.5.7 Uploading files

User can upload file.

4.5.8 Downloading files

User can download file.

4.5.9 Creating directories

User can create directory.

4.5.10 Deleting directories

User can delete directory.

4.6 User scenarios

User scenarios are stories that help to show or demonstrate how can selected actor achieve the goal in the application. It describes the interaction of user and system in greater detail than use cases, helping target audience to better understand the concept of use case and understand the functionality.

4.6.1 Owner

4.6.1.1 Configuration

The owner adds dependency of SFTP proxy for AWS to his project, creates an instance of Configuration, and sets required parameters of this configuration. Basic configuration attributes such as port, private key path, and list of users are required in each and every configuration. Each specific implementation of the Configuration interface may require additional parameters in order for it to work as expected. After creating this configuration, the owner passes it to the server which uses its parameters to set up the application.

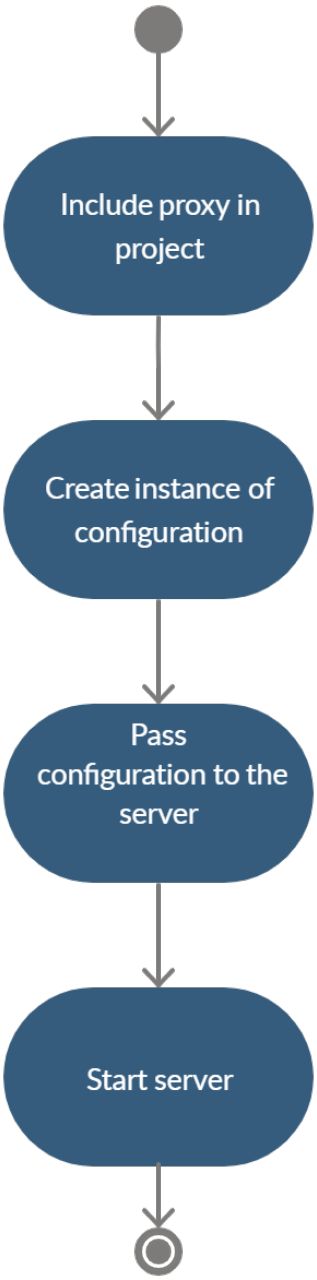

Figure 4.3: Activity diagram of creating configuration

4.6.1.2 Creating accounts

The owner creates an account by creating an instance of one of the implementations of the UserConfiguration interface. According to what the user account owner wishes to create he selects appropriate implementation. Every user account needs to have a user name, the public key in string format, and file system that decides what file system will operate on user's requests (and thus deciding if user account shall be typical SFTP account or account that communicates with AWS S3 storage). Each specific implementation of the UserConfiguration interface may require additional attributes in order for it to work. After creating a wanted user account, the owner passes it to the configuration.

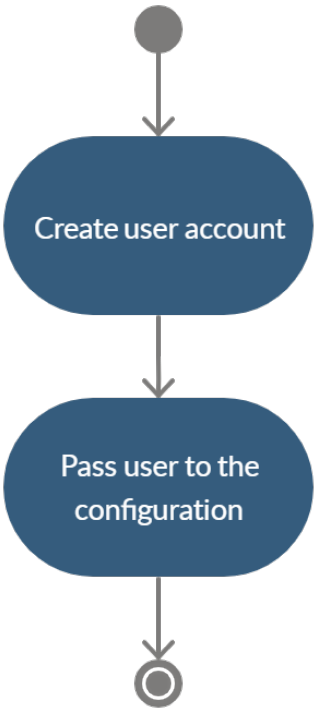

Figure 4.4: Activity diagram of creating user account

4.6.1.3 Permission and mapping for bucket

In a specific account that communicates with AWS S3 storage, the owner needs to specify the mapping of buckets to his created and selected custom directory tree. In every mapping, the owner specifies what bucket to be mapped, custom

directory tree where it should be mapped, and permissions of the user whom the mapping belongs to this bucket.

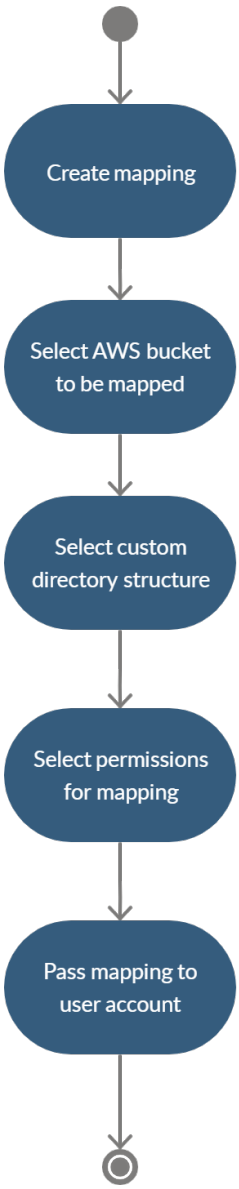

Figure 4.5: Activity diagram of creating user mapping of buckets and permissions

4.6.2 User

4.6.2.1 Login

The user decides what client he wants to use for SFTP connections. The user inputs his username and his private key for authorization. If users credentials matches the account stored in servers configuration user is connected to the SFTP server, if not user's credentials are refused and to log into the server he needs to input correct credentials.

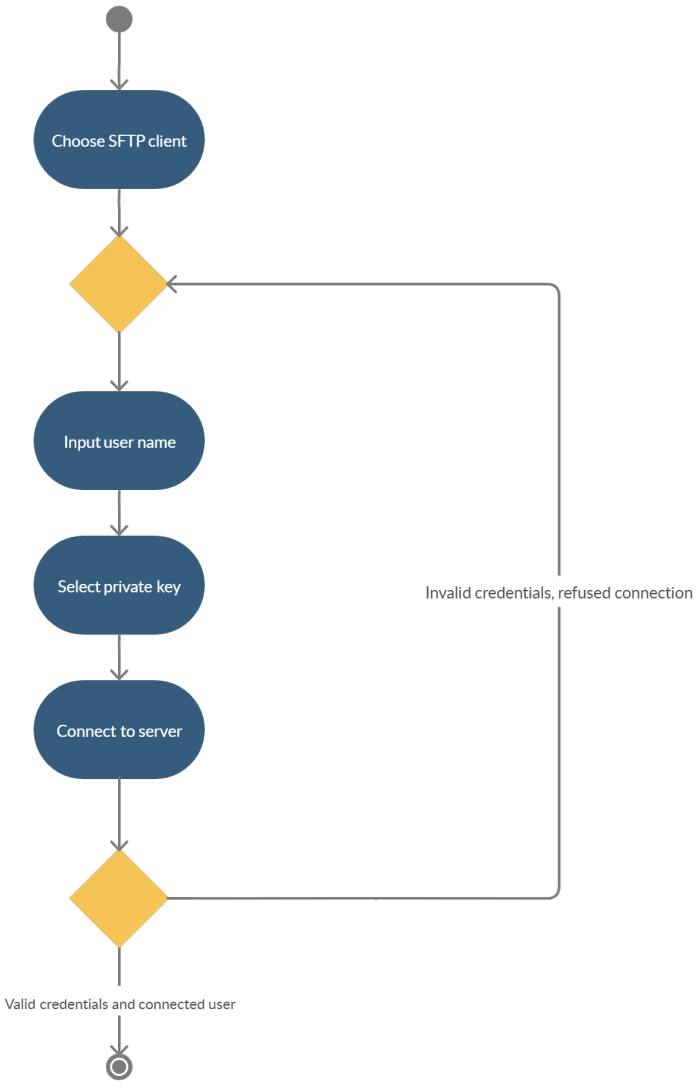

Figure 4.6: Activity diagram of connecting to server

4.6.2.2 Listing files

User logs into the server. Users can change freely directory using the client of his choice. User sends command to list the files and the files are returned to the users.

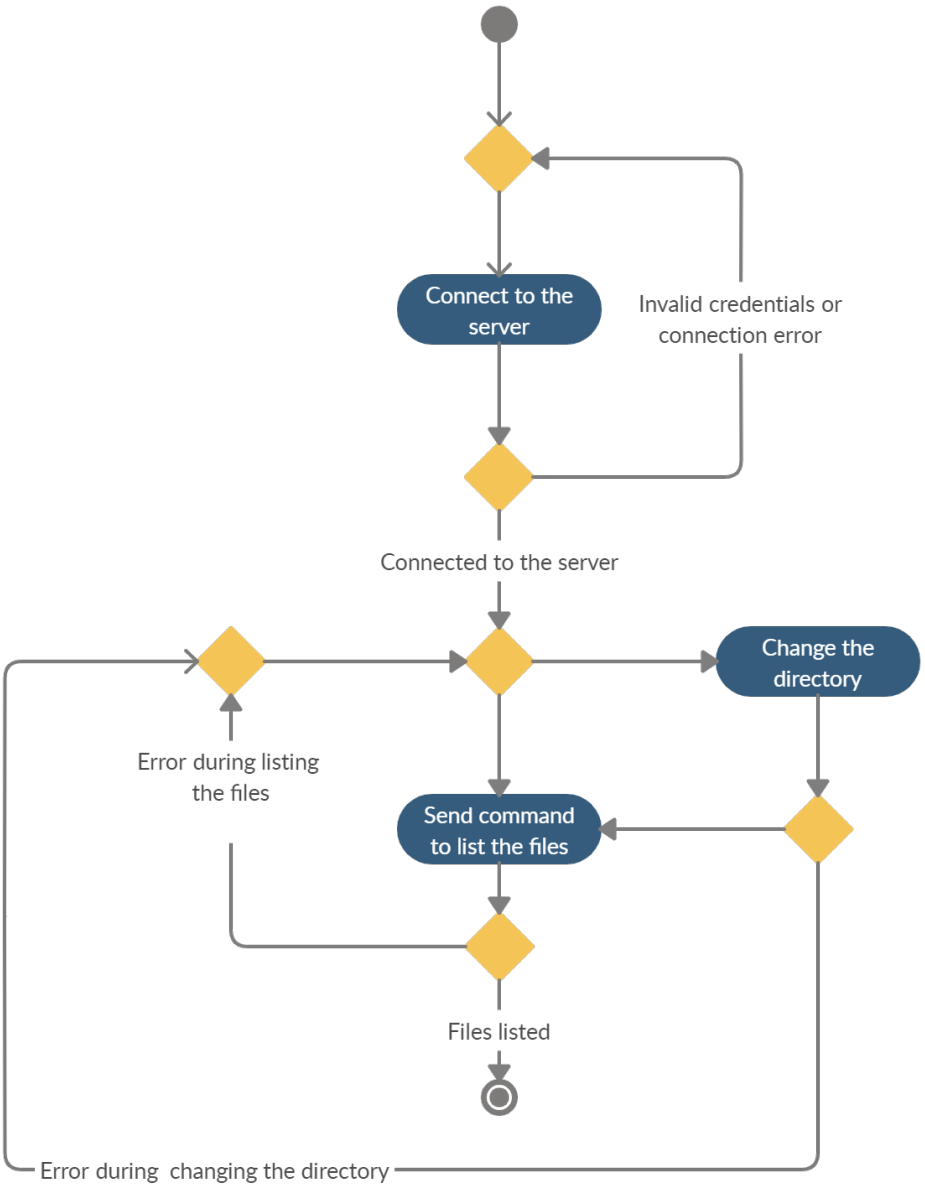

Figure 4.7: Activity diagram of listing files

4.6.2.3 Removing files

Users logs into the server. Changes directory to a directory that contains file he wants to remove and remove specific file by sending command using the client of his choice. Users can delete files by specifying the whole path of the file, thus he can omit changing directory.

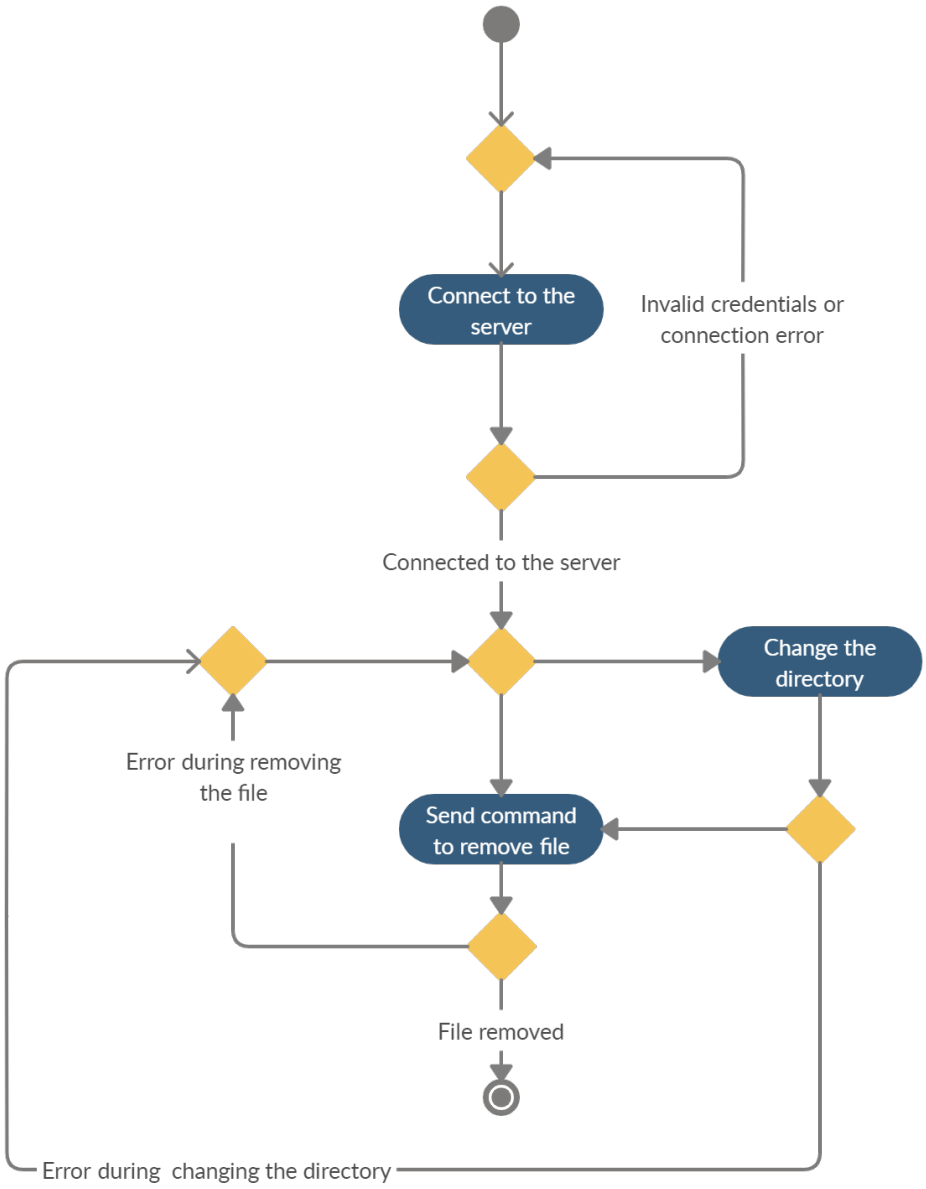

Figure 4.8: Activity diagram of removing files

4.6.2.4 Renaming files

User logs into the server. Changes directory to a directory that contains file he wants to remove and rename the specific file by sending command using the client of his choice. User provide the name of the file he wants to rename and the new name of said file. The name of the new file must be unique to the directory it is stored in, there cannot exist file or directory with user-specified new name of the file.

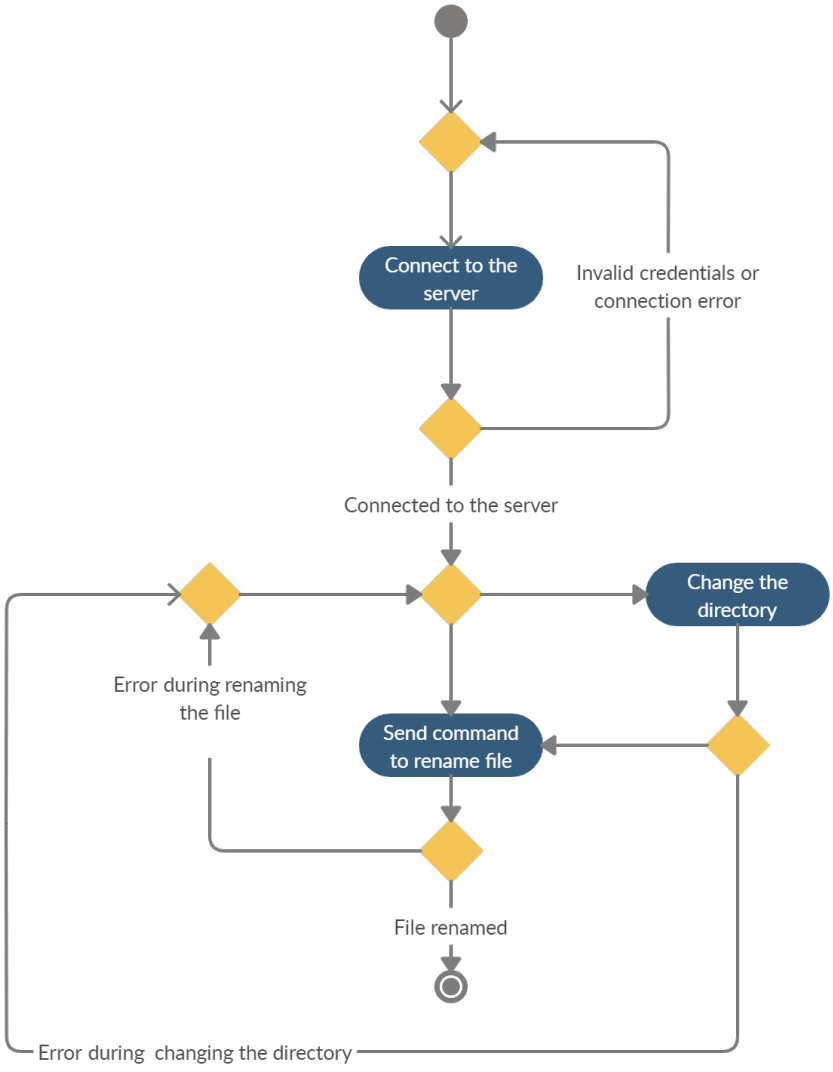

Figure 4.9: Activity diagram of renaming files

4.6.2.5 Uploading files

User logs into the server. User changes directory to directory where he wants to put a new file and upload the file by sending command using the client of his choice. The user specifies the file he wants to upload and uploads files. Users can change directory and select a current working directory as a place where to upload file or he can specify the absolute path where to upload the file.

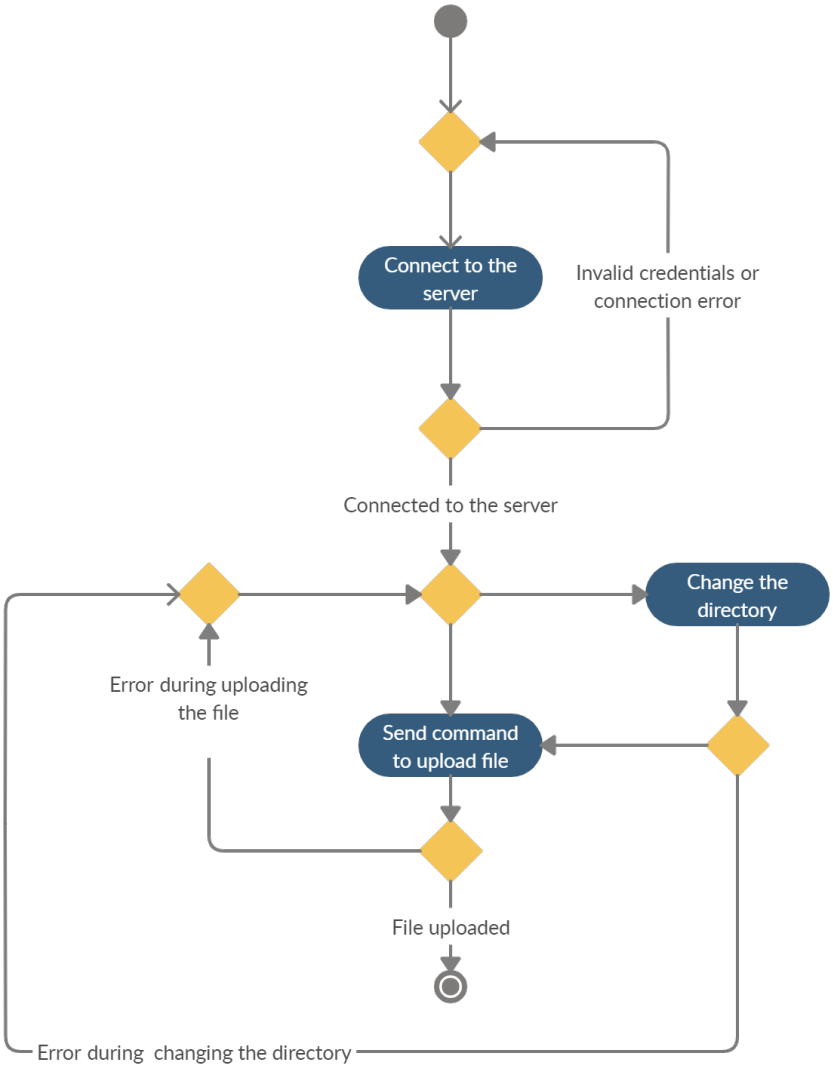

Figure 4.10: Activity diagram of uploading files

4.6.2.6 Downloading files

User logs into the server. User changes directory to directory that contains file he wants to download and downloads the file by sending command using the client of his choice. The user specifies the file he wants to download and download path. Users can change directory and select a current working directory as a place where to upload file or he can specify the absolute path where to upload the file.

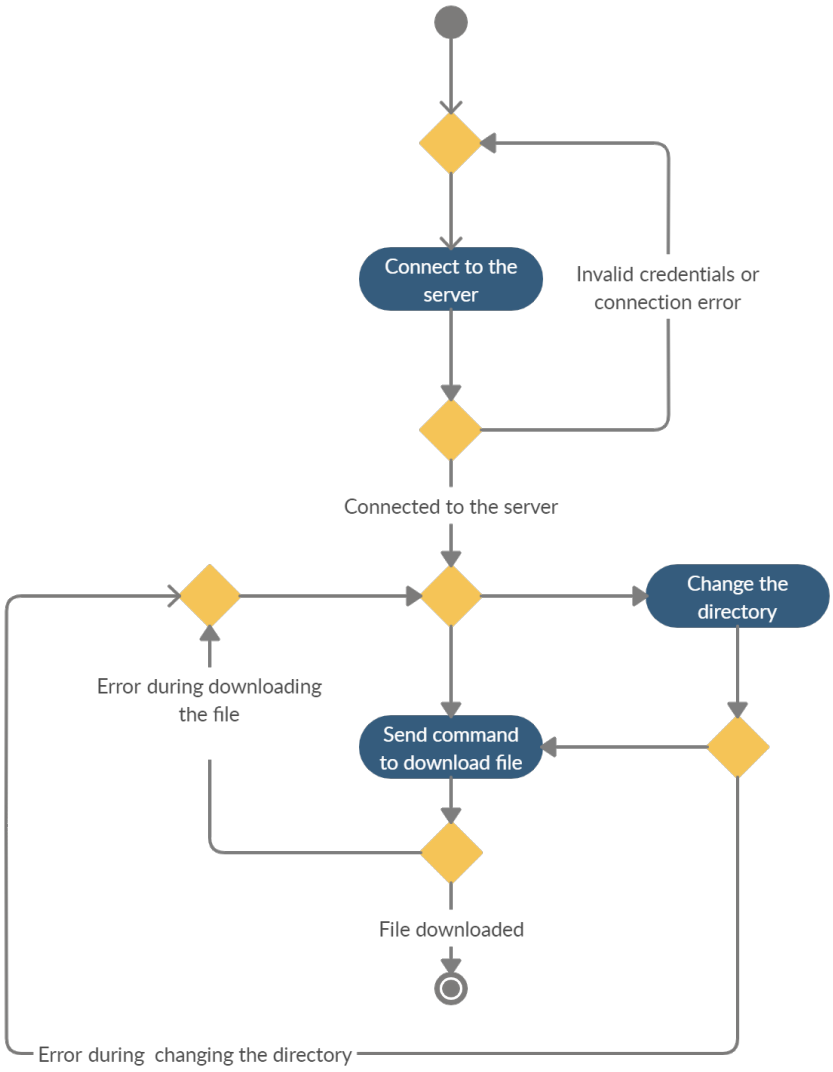

Figure 4.11: Activity diagram of downloading files

4.6.2.7 Creating directories

User logs into the server. The user changes the current working directory to directory where he wants to create a directory and creates directory be sending command using the client of his choice providing the name of the directory. The name of the directory to be created must be unique, meaning there must not exist any file/directory with that name in the directory where a new directory should be created.

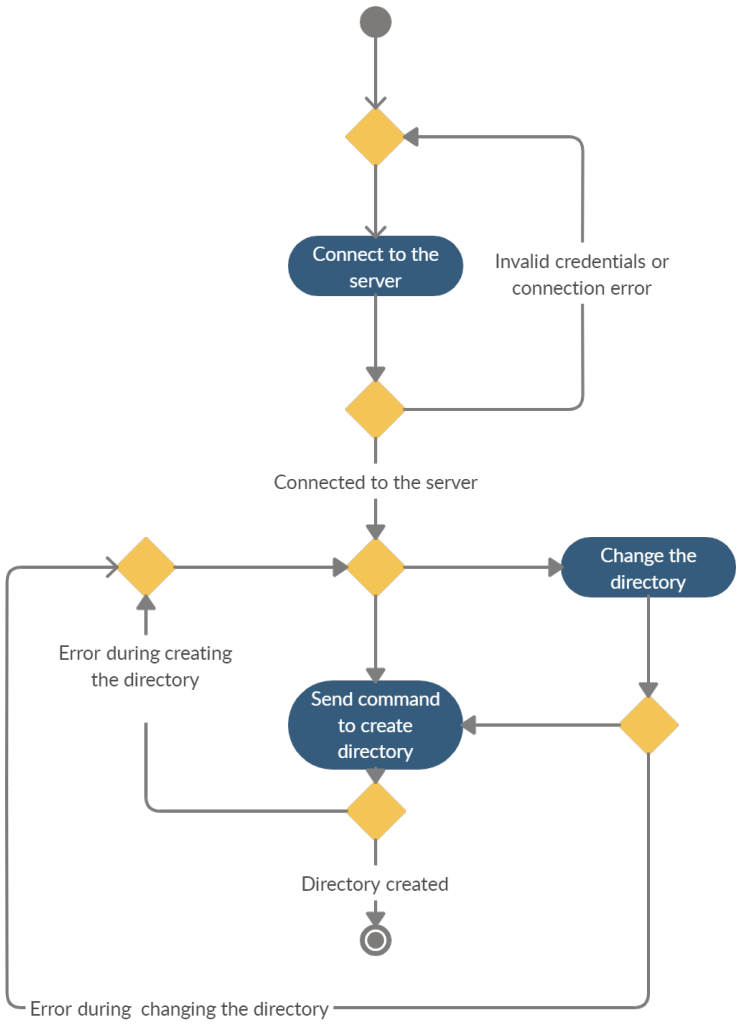

Figure 4.12: Activity diagram of create directories

4.6.2.8 Deleting directories

User logs into the server. User changes directory to directory which contains directory he wants to delete and deletes directory be sending command using the client of his choice. In order for a directory to be deleted, it must not contain any files or directories. Otherwise, error informing the user that the directory is not empty is returned.

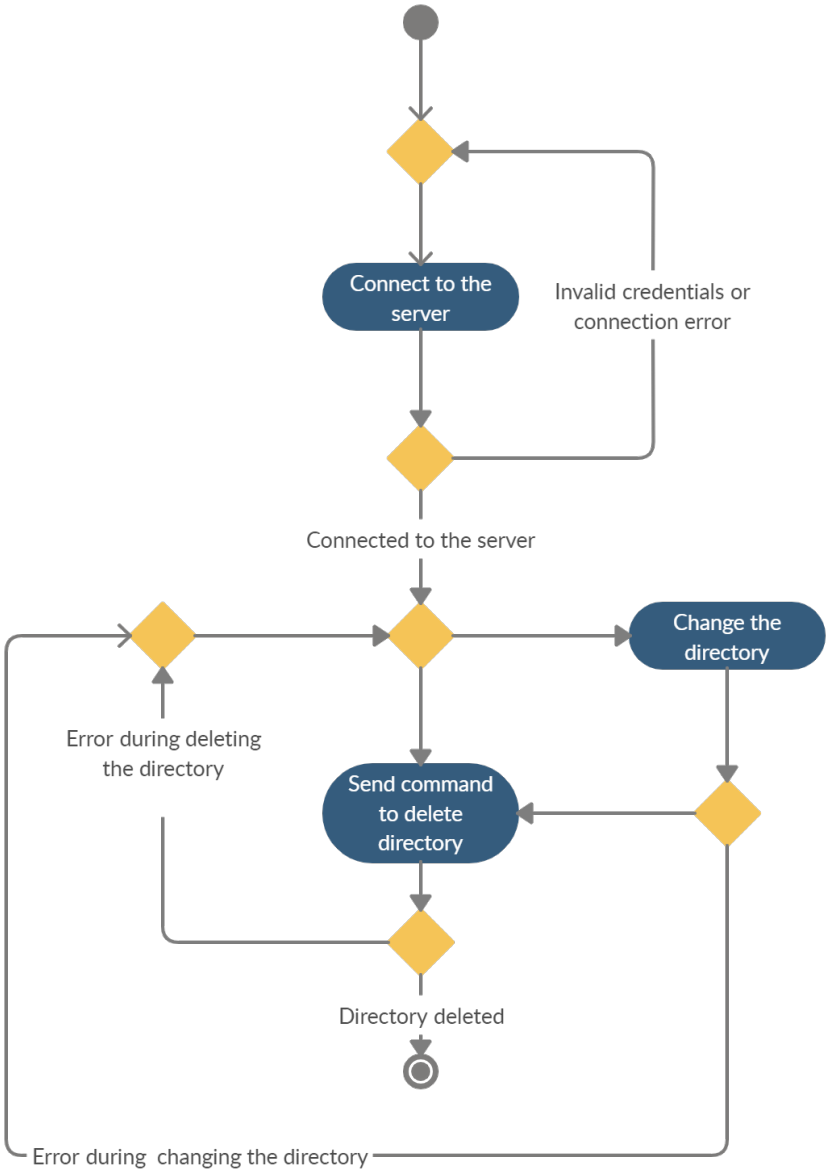

Figure 4.13: Activity diagram of deleting directories

4.6.2.9 Moving files

User logs into the server. Changes directory to a directory that contains file he wants to move and moves the specific file by sending command using the client of his choice. User provide the name of the file he wants to move and the new destination of the file. The new destination of the file must be unique, there cannot exist file or directory with user-specified new destination of the file.

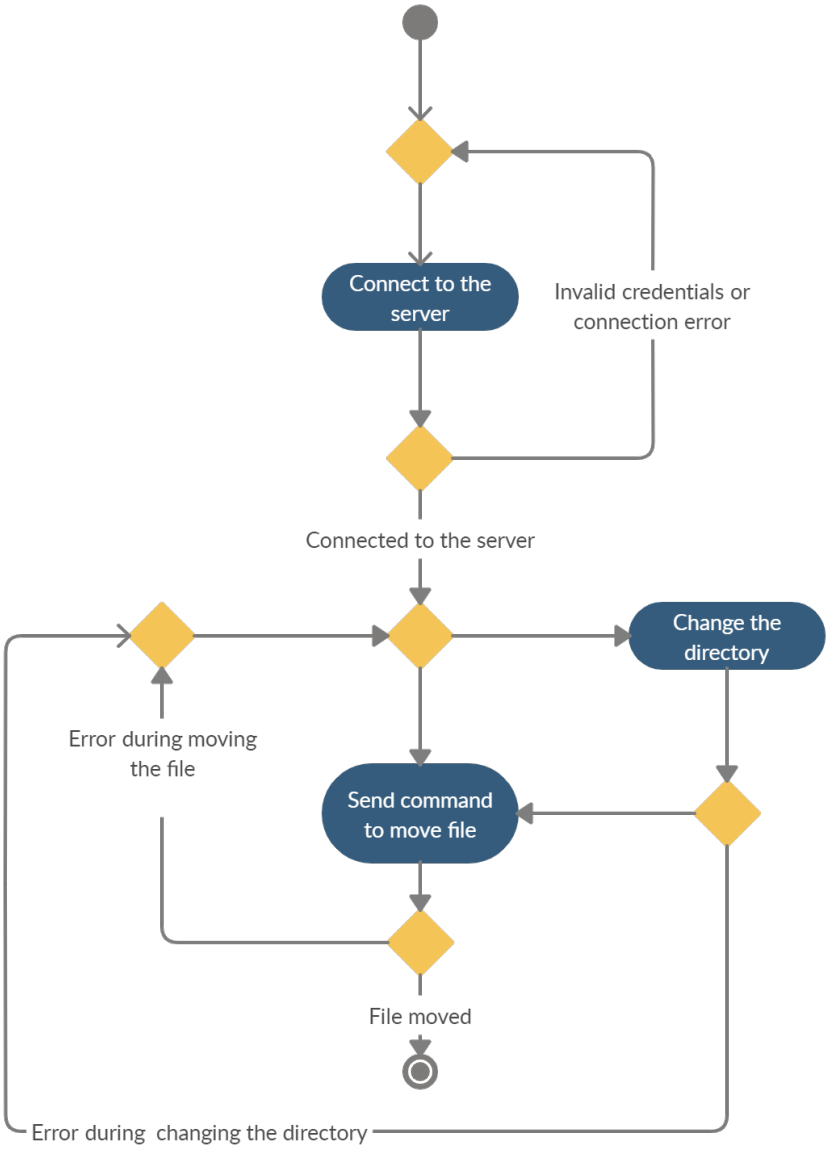

Figure 4.14: Activity diagram of moving files

Chapter 5

Realisation

In this section, I describe the process of creating an application and explain particular dependencies and relationships between packages.

5.1 Configuration

The first step of realization was to create a configuration that would hold all information needed for the server to run in one place. As mentioned earlier, every configuration needs to specify the port on which the application will listen on. The port must be available. If the user knows what port will be free he can directly pass port to configuration. There is another possibility to find a free port. To find a free port user can open a new server socket(passing port number 0 will result in finding available port to listen on), get its port number, and close the socket. Example of this code could look like this:

```
private int generateRandomPort() {
   ServerSocket s = null;
   try {
      s = new ServerSocket(0);
      return s.getLocalPort();
   } catch (Exception e) {
      throw new RuntimeException(e);
   } finally {
      assert s != null;
      try {
         s.close();
      } catch (IOException e) {
         e.printStackTrace();
      }}}
```

Another required attribute of every configuration is the path to the private key that ensures the identity of the server will not change after restarting the application. The private key needs to be retrieved from the classpath. The last required parameter that every configuration needs to have is a list of users. Configuration for S3 needs additional parameters. This configuration needs to have S3ProxyFactory, which provides AmazonS3 client, the client provided by Amazon Web Services that serves as tool for direct communication with S3 storage. The next attributes required by configuration for the proxy server are download batch size and upload batch size. These attributes serves as a way to define the size of a buffer the data will be loaded to and then downloaded from, and the size of buffer the data will be stored to and then uploaded from. S3 storage has minimum upload size when uploading files in parts(the last part can be smaller), this minimum size is 5MB of data, thus the minimum value of upload batch size is this value, throwing an error if user inputs a smaller value. The bigger this value is the bigger buffer is which leads to bigger memory consumption by uploading files but ultimately leads to fewer requests to S3 thus leading to lower cost when uploading a file. Download batch size has no limits, but cannot be 0 or less. The buffer for downloading is not a direct requirement of AWS S3 storage but serves as an abstraction for fewer requests to the S3 storage, which are paid and resulting in a smaller cost of downloading the requested file. Instead of retrieving data for every SFTP request, we preload the bigger batch of data. Last two arguments are indicator, if the download and upload should be done in paraller way and how many threads should be used for this.

5.2 MINA

For this project, SSHD Apache MINA was selected as the SFTP library. Mina supports both client and server-side. The project includes MINA as maven dependency and extends its functionality for the desired purpose. Mina offers custom implementations of PublickeyAuthenticator that offers users to implement their own way of authorization using the public key. Mina is initialized by creating an instance of a server. We set the port, which the server will listen on(as mentioned, port must be free), add the path to the private key that will ensure the identity of the server won't change after restarting the application. SshServer takes any implementation of KeyPairProvider for the private key. For this project, BouncyCastleGeneratorHostKeyProvider was chosen with .pem key format. SshServer requires factories of subsystem, which will be created after the user logs in and will handle communication with SFTP. In this project, custom factory was created, which creates an instance of the custom subsystem that extends the functionality of the SFTP subsystem provided by MINA(which is described later in this section) Last important argument for this project is PublicKeyAuhtneticator that handles

the way users are authenticated. The server is started and stopped using its start() and stop() method. The start method is non-blocking which means an infinite loop must follow to assure application won't stop. The stop method must assure all resources are freed.

In the project this code part of code is used to initialize server:

```
sshd = SshServer.setUpDefaultServer();
sshd.setPort(configuration.getPort());
sshd.setKeyPairProvider(new BouncyCastleGeneratorHostKeyProvider(
Paths.get(ClassLoader.getSystemResource(configuration.getPrivateKeyPath()).toURI())));

CustomSftpSubSystemFactory factory = new
    CustomSftpSubSystemFactory.Builder()
.withS3ProxyFactory(configuration.getS3ProxyFactory())
.withDownloadBatchSize(configuration.getDownloadBatchSize())
.withUploadBatchSize(configuration.getUploadBatchSize()).build();

sshd.setSubsystemFactories(Collections.singletonList(factory));
sshd.setPublickeyAuthenticator(new
    UserKeySetPublickeyAuthenticator(configuration.getUserConfigurations()));
sshd.start();
```

In project, we take mina and connect custom parts to it. Mina calls them in its internal implementation.

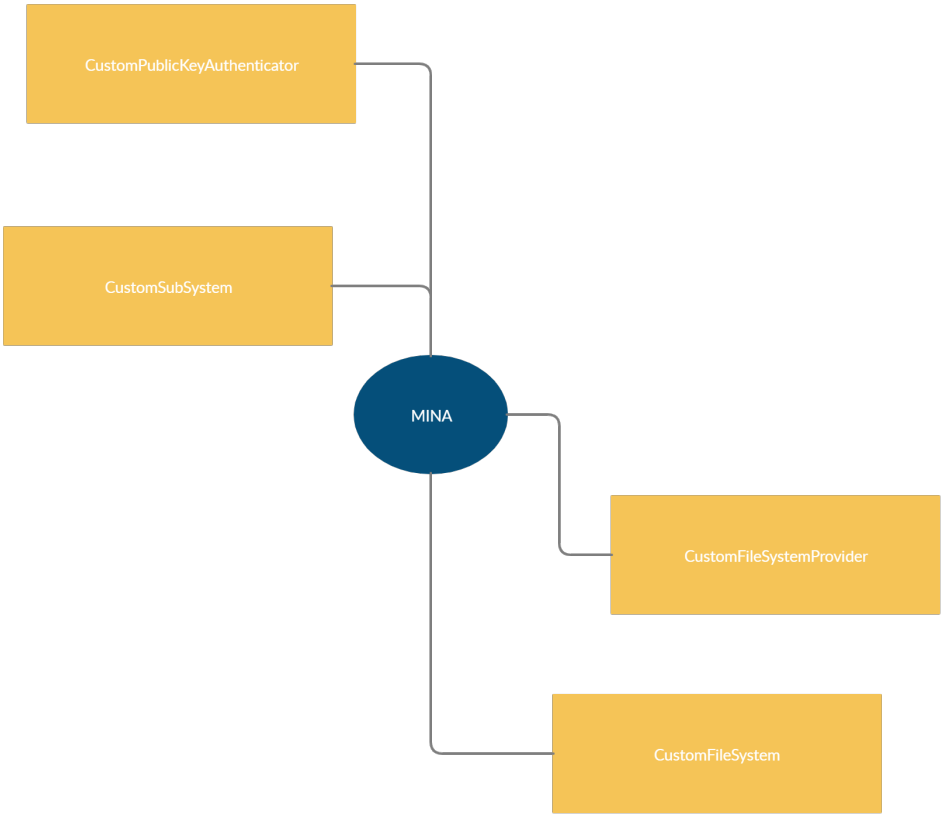

Figure 5.1: MINA

5.3 Authentication

As mentioned earlier, public key authorization is used for authenticating users. Custom made PublickeyAuthenticator takes a list of users as a constructor parameter. In the constructor, the user list is mapped into a map, with the name of the user as key and user as value. During authenticating, the authenticate method is called internally, which takes the user name and public key of user that is trying to authenticate as an argument. The method first checks if the map contains any value with an arrived user name, if not the user does not exist and returns false. If the user with arrived user name exists, the method compares the public key that has arrived and the public key stored on the server belonging to the user name. If they match the user, the user has provided correct credentials and object representing this user is stored into the session of connection and true is returned, meaning the user is authorized, otherwise false is returned and the user needs to connect with correct credentials.

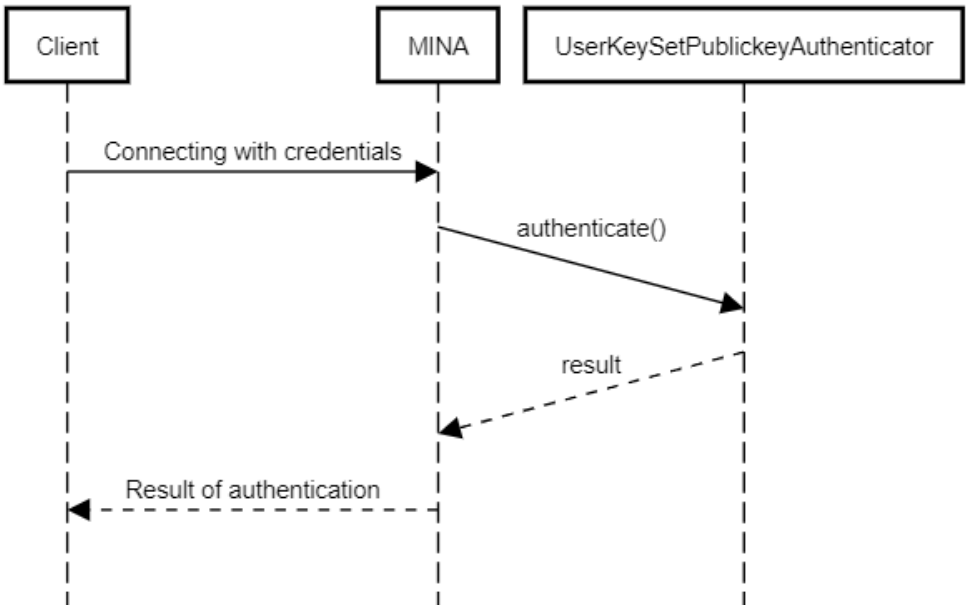

Figure 5.2: Authentication on server

5.4 S3 proxy

In this part, proxy around communication with AWS is discussed. The proxy resides in its own package. All classes in this package are related to this communication.

S3 proxy provides communication with Amazon web Services. S3Proxy is created by a proxy factory which is a necessary parameter of S3Configuration. Users can implement the S3ProxyFactory interface and create his own implementation deciding the way S3Proxy will be created. S3ProxyFactory needs an instance of AmazonS3 for remote communication with AWS which is supplied to it as an argument to its constructor. AmazonS3 is SDK provided by Amazon. S3 storage provides REST API for users, AmazonS3 is a wrapper that encapsulates REST API calls and handles mapping to classes provided by this SDK. Internally, S3Proxy makes calls to AWS using AmazonS3, mapping results to classes defined in its package, and returning them. During mapping to its own classes, S3Proxy takes only required parameters from returned objects.

It is up to the user to provide the way Supplier<AmazonS3> for S3ProxyFactory is created or retrieved, the user can create it the way that suits his use cases.

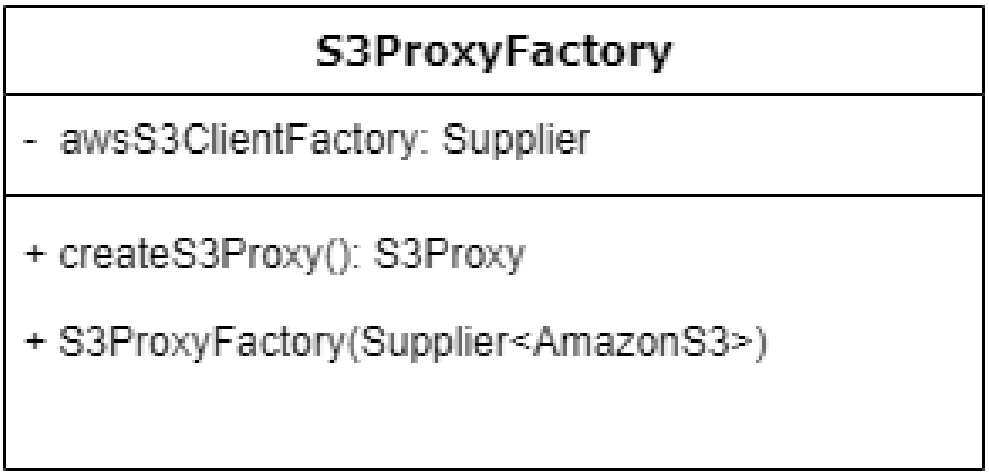

Figure 5.3: factory class

This Supplier will be used for every account created and passed to configuration, meaning they will all share the same credentials. In order to retrieve credentials, the user will need to register on the AWS site and configure his account first.

Every call using AmazonS3 object may throws two exceptions:

- AmazonServiceException - indicating that request send to amazon was correct, however service was not able to process it for some reason
- SdkClientException - indicating the service could not be contacted or client couldnt parse response from the service. It is base exception for all client exceptions, however it is good practice to catch AmazonServiceException and then SdkClientException

The S3Proxy defines its exception called S3ProxyException. AWS exceptions are caught and then mapped into S3ProxyException which is thrown from all methods that communicate with S3. By this concept, implementation is hidden from outside of the package. S3Proxy is interface thus allowing us to create end-to-end testing.

5.4.1 Listing files

The purpose of listing files is to retrieve info about files inside the directory. However S3 does not know the concept of directories, everything is just an object. The user interface of S3 may show directories, but it is just an abstraction around it.

AmazonS3 provides a method for listing the files called listObjects. This method takes ListObjectsRequest as an argument. In order to list the files, we need to pass the bucket name, prefix, and delimiter to this request. The bucket represents the name of the bucket, we want to list the object from. The prefix indicates prefix of the file we want to list directories from, for example, if we had directory structure such as

/bucket/dir1/dir2/file1
/bucket/dir1/dir2/file2
/bucket/dir1/dir2/dir3/file3
/bucket/dir1/dir2/dir3/file4

and we wanted to list files contained in directory dir2, we would pass prefix such as

/bucket/dir1/dir2

This alone would list all objects that contain a said prefix, including

/bucket/dir1/dir2/dir3/file3
/bucket/dir1/dir2/dir3/file4

which is unwanted behavior in the typical non-recursive directory listing. For this we pass delimiter argument into the request, passing "/" will result in the typical directory listing, not recursive.

The delimiter causes keys that contain the same string between the prefix and the first occurrence of the delimiter to be grouped into a single result and stored in the CommonPrefixes collection. This collection is not returned elsewhere in the response. Each rolled-up result counts as only one return against the MaxKeys value[aws docs listing].

After receiving a response from amazon storage, the response is mapped into S3Proxy's own class holding the information. The response contains metadata about requests and responses, but also a list of common prefixes that we can interpret as directories and list of summaries of objects inside our specified directory.

Since bucket can hold unlimited numbers of objects, amazon splits the response using pagination for a large list of objects. Due to this fact we need to check if the list is truncated or not, and send another request depending on it. This is ideally done in loop until non truncated list is returned from amazon.

5.4.2 Other communication with AWS

Other communications with S3 storage are simple calls to amazon using their AmazonS3 object. All exceptions in these calls are caught and mapped into S3ProxyException which was mentioned earlier. Every return value (if there is) is mapped into S3Proxy's own class thus hiding the implementation and removing the need for another dependency when using the S3Proxy package.

Some method may require more arguments, for more clarity, instead of passing raw arguments, Requests are implemented. The specific requests hold all data needed for the method which ensure more clarity and less chance for failure.

5.5 First approach - overriding methods that handle command procesing

The main class that provides processing SFTP commands is SftpSubsystem. This class contains core functionality for SFTP processing. The first version of the application tried to override its functionality by extending this class and override methods invoked during the processing of particular commands. This approach works quite well, however, it, in a certain way violates "Don't repeat yourself" principle. This principle deals with reducing the repetition of software patterns and redundancy.

Each method invoked to process the SFTP command receives buffer as a parameter. This buffer represents data sent by the client to the server including all metadata, parameter, and data itself needed to execute the command. Needles to say, this buffer needs to be parsed. Overring these methods require to implement once again the parsing of this buffer, extracting all information needed. Not only it could lead to faulty parsing of the buffer, it directly violates the "Don't repeat yourself" principle.

Another violation comes with handling the handles. Every time SFTP wants to read the content of the directory, upload, or download file, it needs to create a handle first. The reason for this simple. None of these processes are finished in one request. To read and list the data of directory, SFTP client sends openDir requests to the server. The server creates a handle for the desired directory, assigning unique hash to it and storing it. This handle then contains a list of files in the directory. As a response, the server sends a generated handle hash to the client. SFTP then sends readDir command to the server with received handle as a parameter. The server then parses the handle parameter and extracts DirectoryHandle which it reads the content of directory from. This command is repetitively sent by the client until EOF is returned, indicating no more files from the directory are left to be read.

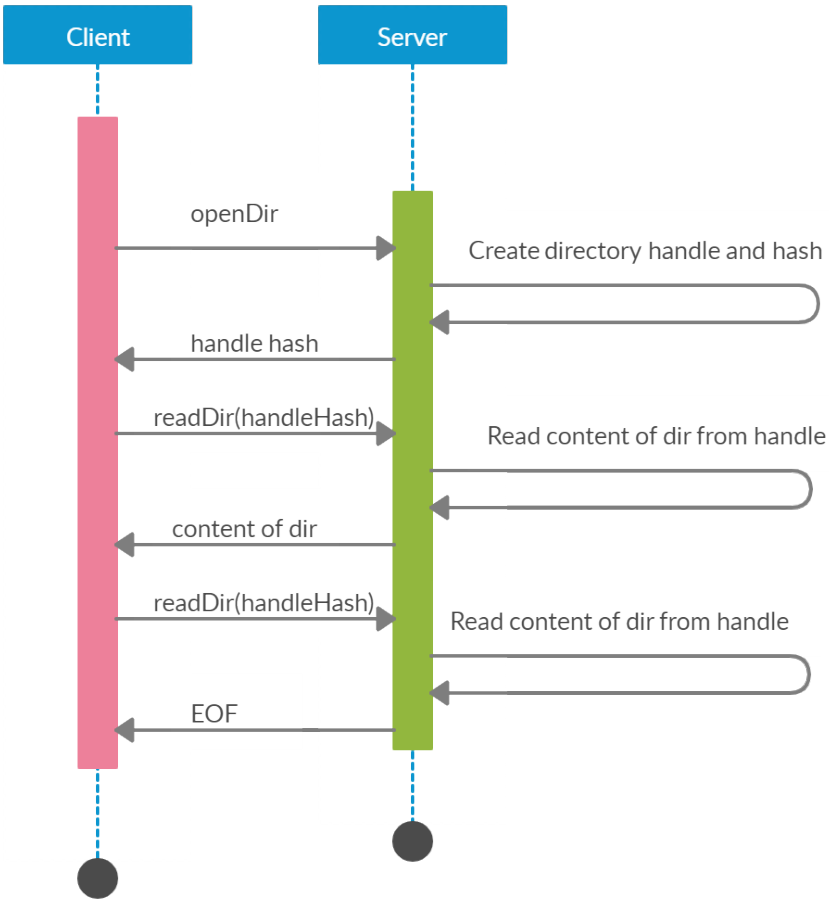

Figure 5.4: Opening and reading directory

Reading and downloading files are handled correspondingly. Initially, the client sends an open request that opens the file and generates a handle hash which is sent back to the client. Each subsequent write or read command then works with this handle.

Due to this, implementing an application with this approach would require to create custom handles, each for directory and file further violating the "Do not repeat yourself" principle. Apache MINA does not provide the easy(or pretty) way how to override the functionality of it's defined handles. Implementing whole new handles is the best way to achieve the wanted result. However, this also adds extra work in the overriding of the SftpSubSystem.

Another reason this approach was discarded was due to the way MINA is implemented. To write clean and readable code and follow Separation of Concern principle we should separate SFTP handling and abstraction around

AWS, meaning every method handling processing of commands should have the following structure:

- parse buffer for required information
- call service that handles communication with AWS
- send response to the client with appropriate parameters

But in many cases, the service would need to import dependencies from MINA itself, which is the quite unwanted situation and would complicate development, if we wanted to switch MINA for any other library, or if MINA would publish a new version that would have newly found crucial bugs or vulnerabilities fixed, but changed the structure of the project or used Java 9 modules. This can be avoided by once again violating "Do not repeat yourself" principle and implementing required dependencies from MINA once again in the service.

5.6 Second approach - implementing FileSystem and Java NIO

The second and last approach was through implementing custom FileSystem, FileSystemProvider, and Path interfaces provided by standard JDK. SftpSubSystem takes SftpFileSystemAccessor as a constructor parameter. This file accessor is used to open files and directories. However, this is not used when renaming or deleting directories. In order to delete or rename the directory, MINA uses Java NIO File technology. Fundamentally file accessor uses the same way of handling files, making implementing custom FileSystem, FileSystemProvider, and Path the optimal way how to handle all operations. In this case, SftpSubSystem is also extended, but no method for command handling is overwritten. The purpose of this extended class is to keep original functionality, but add the way for the user to pass arguments from configuration to it, and to set filesystem.

5.7 S3Path

The Path interface represents the system dependant file path. This custom S3Path represents the AWS path. It presents functions to retrieve file names from the path, its parents, subpath, and or sibling. This path has a direct dependency on the FileSystem that created it. Every time command is being processed, the path to the file is represented as S3Path, fundamentally the provider from FileSystem attached to S3Path is extracted, and particular methods are invoked to achieve the goal.

5.8 S3FileSystem

Custom implementation of FileSystem is a root of functionality. When the connection for the user is established, this custom file system is set for the user. The most important function of this file system is creating S3Paths and holding a reference to custom S3FileSystemProvider.

5.9 FileSystemprovider

Custom implementation of FileSystem provides the business logic for abstraction around AWS. It handles communication with the S3Proxy package and logic that is required for some operations.

5.9.1 Determining AWS path from local Path

Thanks to custom mapping, the path of the file that is received by the server isn't in the exact format, meaning we cant use this path to make AWS call. We must first determine what mapping to use and then use this mapping to construct the AWS path from the local path. During initialization map of prefixes for every mapping is created, if we have the mapping

/dir1/dir2/dir3

the map of preffixes contains

/dir
/dir/dir2
/dir/dir2/dir3

values indicating if its full path or not, in this case only /dir/dir2/dir3 is the full path.

The rules for determining the best candidate for mapping are simple, the mapping that contains the longest prefix is the most suitable. However, the path must contain a full prefix to be considered as adequate mapping. In case we have the following mapping

/bucket1 -> /dir1/dir2/bucket1
/bucket2 -> /dir1/dir3/bucket1/bucket2

And the path for listing for listing is

/dir1/dir/bucket1

the first mapping is considered as AWS mapping, for second mapping does not contain full prefix thus making it local mapping instead.

5.9.2 Opening and reading directories

While the client sends openDir request server creates a handle for the directory(opening it) that contains the content of the required directory. Opening the directory is handled by FileSystemProvider. During opening the directory, the stream is created. This stream contains Collection of DirEntries - that

represents entry(files) in the directory. In order for S3FileSystemProvider to retrieve entries, it must determine if it should list directory from the custom structure, or make a call to S3Proxy and receive data from AWS. The provider first iterates over all mappings, checking if the path for listing matching any local path, if yes it adds next directory as the entry. For example, if our mapping looks like

/bucket1 -> /dir1/dir2/bucket1

/bucket2 -> /dir1/dir3/bucket1/bucket2

And the path for listing for listing is

/dir1

the provider would return dir2 and 3 entries. In the next step, the provider retrieves matching mapping for AWS and makes a call to the bucket belonging to this mapping.

In the full example, if our mapping looks like

/bucket1 -> /a/b/bucket1

/bucket2 -> /a/b/bucket1/c

and the directory path to be listed was

/a/b/bucket1

the provider will first check all mappings and see if any this path represent any local path in mapping, in this case, it represents a local path in the second mapping

/bucket2 -> /a/b/bucket1/c

it takes the mapping, and returns next dir in the tree, in this case

c

then it determines the mapping to list AWS files. In this case, both mappings contain the requested URL, however as mentioned earlier, the most suitable mapping is the mapping with the longest full prefix. In this case, it is

/a/b/bucket1

for it is the full path. This approach as consistent with how mounting works in the unix world. i.e we can think of the bucket as being mounted into the virtual filesystem tree exposed by SFTP. When the provider determines the mapping, it constructs the AWS path from the local path and calls S3Proxy to list files. After receiving data from S3Proxy, provider extract directories and summaries of objects and map it into the SftpFile. This process repeats as long as the batch of files is truncated, issuing the call to S3Proxy to load the next batch of files until all files are loaded. When all files are loaded they are stored inside DirectoryStream waiting to be read.

This process is done during creating the handle for a directory as mentioned in previous chapters. The handle holds the reference to this directory stream and retrieves the iterator to a collection stored inside. During reading MINA uses this iterator to indicate if there are any entries left to read or not.

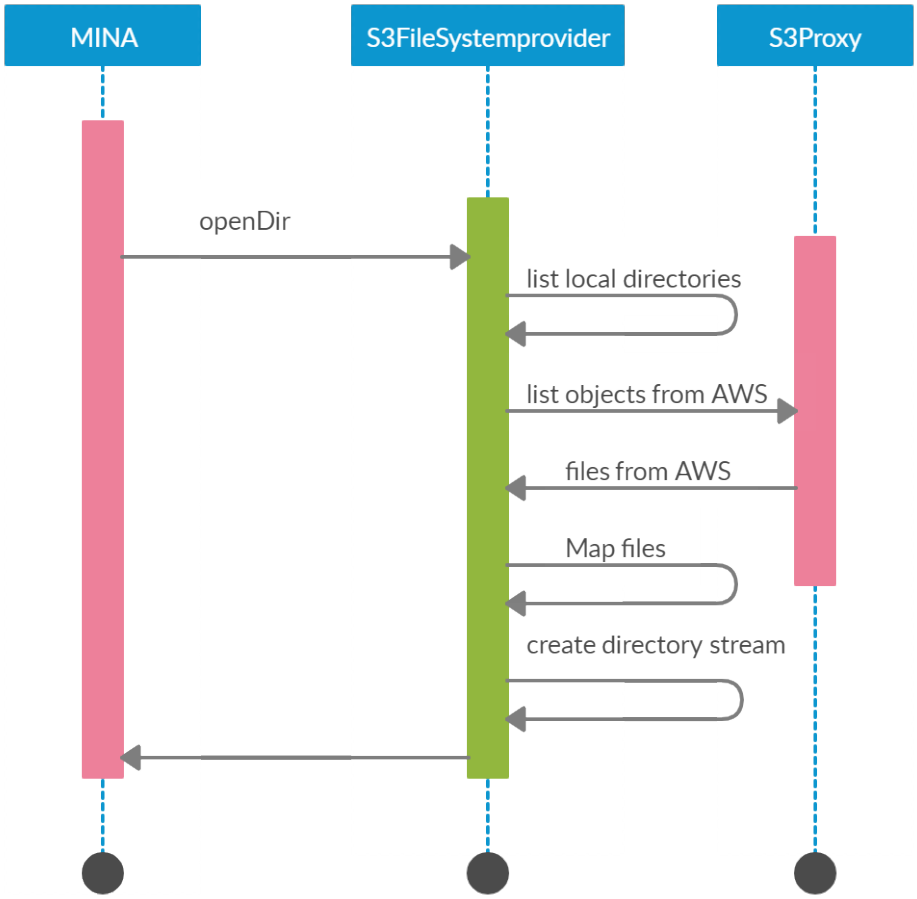

Figure 5.5: Opening dir

5.9.3 Retrieving information about files

Information about files can be requested using doStat, doLstat or doFstat command. These two, doStat and doLstat differ only in one thing, doLstat follows symbolic links. The last command, doFstat is the same as doStat, but handle hash is required instead of a path to the file. Pieces of information about the file can be also requested internally, for example during the listing of files in the directory, we must know its size, name, modified date, and other attributes.

We must somehow find out these pieces of information. When we want to retrieve attributes of a local directory, no call to AWS is needed. The only thing we need to know about the local file is that it is always the directory, and we consider its size 0. To retrieve information about files from AWS we must make a call to AWS using S3Proxy. The incoming path does not distinguish

between files and directories, making it a little bit harder to determine if we want information about file or directory inside AWS. Directories are an abstraction in AWS, but we can simulate directory inside AWS by creating a file with "/" suffix, that acts as a directory. So whenever we want to retrieve the file from AWS, we must check if the file with fileName exists and if not if the file with fileName + "/" exists. We must assure there will never be case when both of these files exists during uploading files.

Knowing this, we can retrieve files by first trying to retrieve files with a given fileName, and if it does not exist we need to try to retrieve files with fileName and "/" suffix, if none exists, NoSuchFileException is thrown indicating file does not exist.

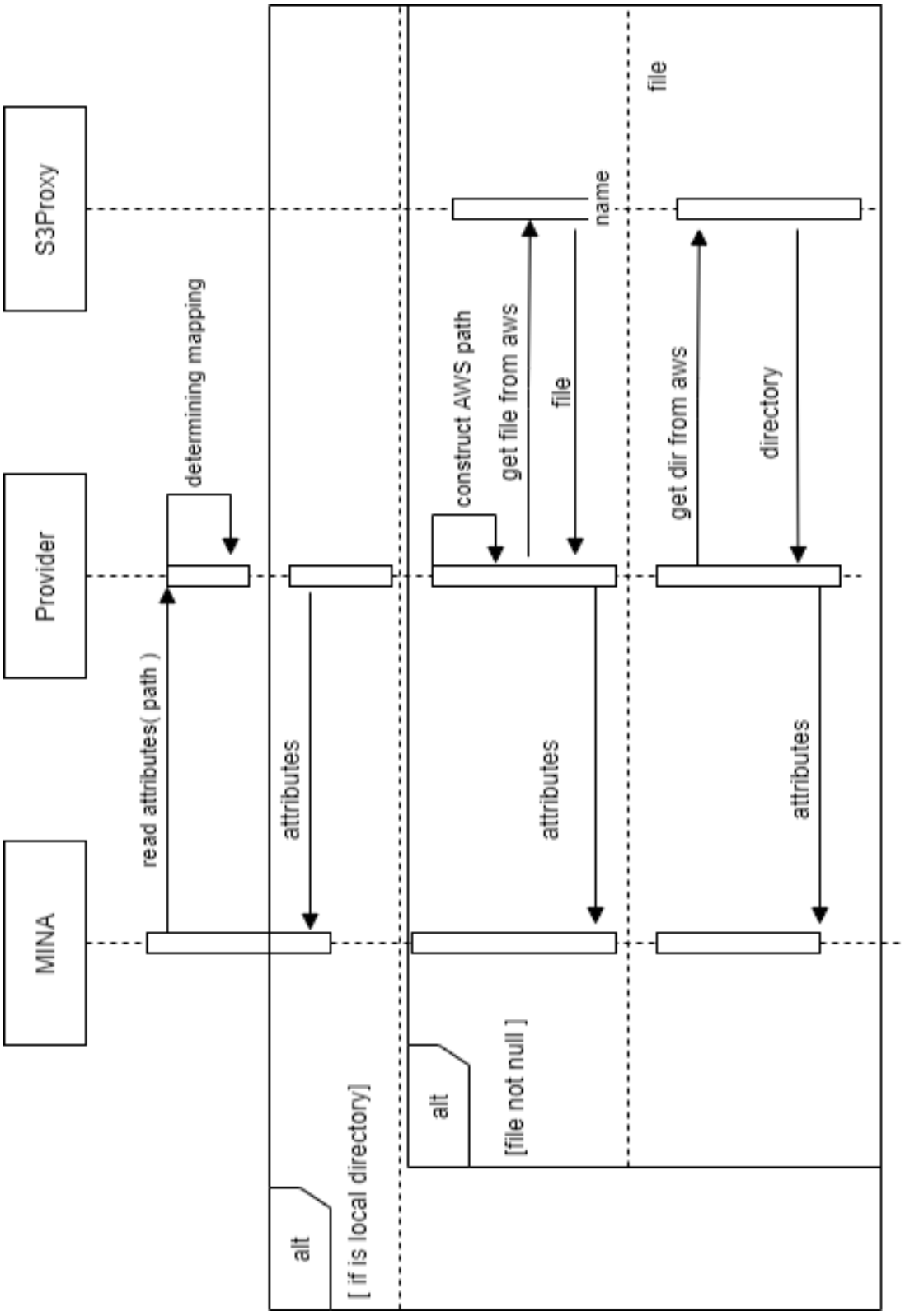

Figure 5.6: Diagram of happy flow of retreiving file attributes

5.9.4 Opening files

To open files, Java NIO newChannel method is invoked. This creates a channel that acts as a gateway to the file's properties. We distinguish two types of opening. Opening for reading and opening for the downloading. During the first one, we must first check if the file with the same name with "/" suffix, as we try to upload, exists. This ensures no file and directory inside AWS with the same name will exist and prevents trouble and unwanted behavior.

During downloading, we must find out the size of the file, for we need to know the exact data we want to download, otherwise, an error could occur from the AWS. The type of opening is distinguished by an attribute that is passed by SFTP, thanks to this we can find the mapping that represents the path and determine if we have access for demanded operation.

5.9.5 Downloading file

During the downloading of the file, previously discussed S3FileChannel is used. Since we are dealing with AWS files, we need to retrieve parts of objects from S3 storage using S3Proxy. This is where downloadBatchSize from configuration comes in play. We can call S3Proxy to get part of an object from AWS on every read call from SFTP(every call to AWS is paid), or we can approach this the better way, and preload batch of data, store it in the buffer and read data from it instead of calling AWS every time, thus saving calls and saving money. On top of this, accessing AWS S3 API for every SFTP read message would be extremely slow - hence some kind of buffer is necessary. The fact that buffer size is configurable allows the user to choose the appropriate trade-off between high memory consumption of the application(larger buffers) and more frequest S3 API calls with higher cost(smaller buffers).

The initial read loads the amount of data as defined by downloadBatchSize and store it in the buffer. Next reads take data from this buffer, if the size of the data we want to read is bigger than elements left in the buffer, we simply load the next batch of data from AWS to fill the buffer. This process is repeated until no data in AWS is left and all data from the buffer are read, making the reading process finished.

During the call to the AWS, we must calculate the start and end of the inclusive byte range to download. If we want to retrieve more data than object offers, an error response could be returned from AWS, meaning we must compare the total number of data read + the size of the data we want to fetch with the total file size which was preloaded during the open command and adjust the requested size of data in the last call to AWS.

If the parallel flag is set in configuration, loading data from AWS is done in parallel way, each thread from number of threads defined in configuration is used to load batch of data. Each thread knows the range of data to load, and know the position in buffer where to store it.

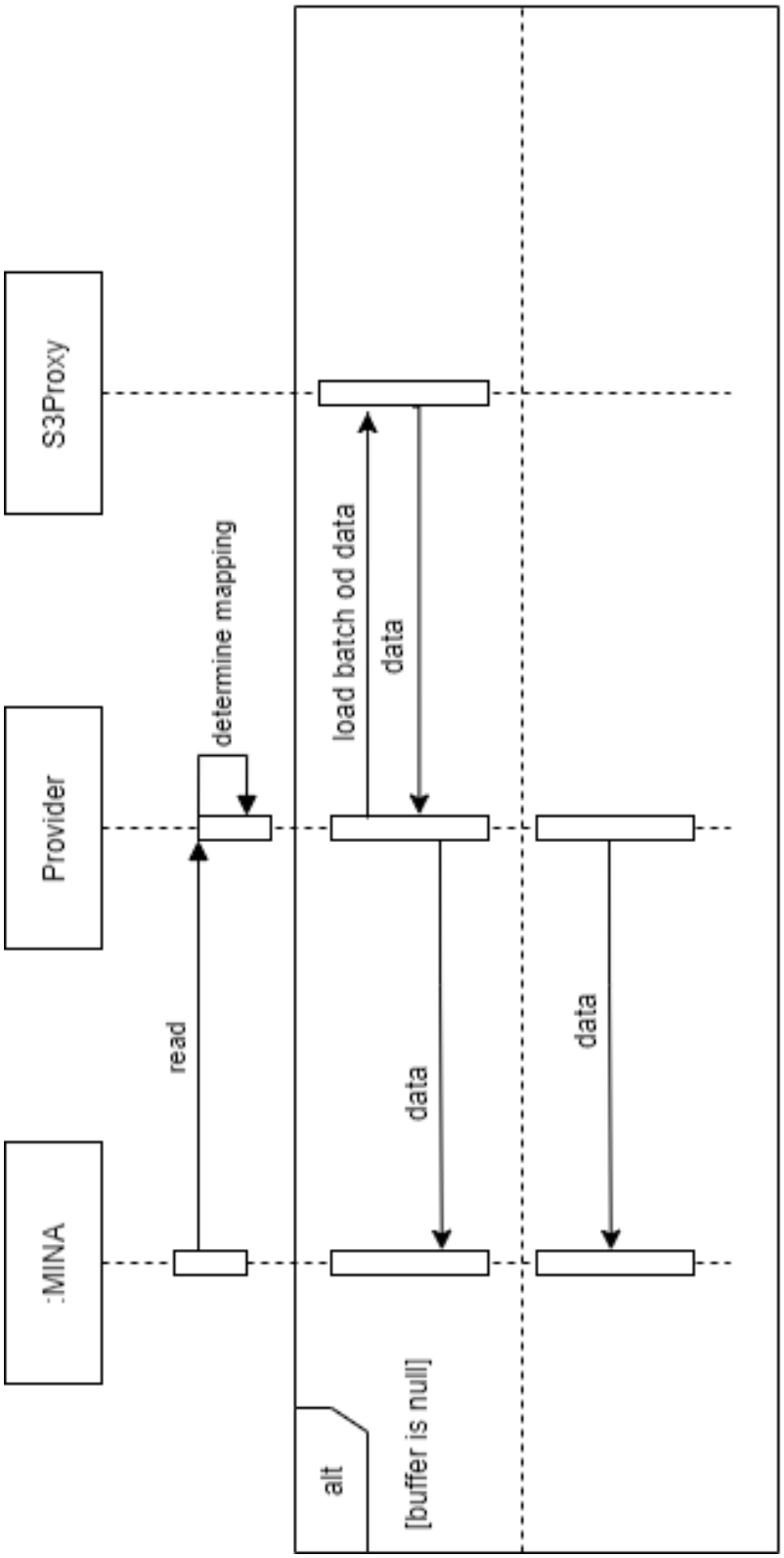

Figure 5.7: Diagram of happy flow of downloading file

5.9.6 uploading files

While uploading files, we must take into consideration that the file may be big and thus may require multiple SFTP write calls. AWS handles uploading the big file using MultipartUplad. Before we start uploading files, we issue the start of the multi-part upload. This returns multipart upload id, that we store inside the map. We use this upload id to tell AWS which upload the part of the file belongs to. This is where the uploadBatchSize parameter from configuration comes in play. AWS allows 5 MB as the minimum size of the part that is being uploaded, with the exception of the last part that can be smaller. The parameter indicates how big uploaded parts will be, the bigger buffer saves AWS calls(thus money) but demands bigger memory space. It is up to the user to choose his preference.

When we upload files, we check if we have multipart upload id, if not we issue a call to AWS and retrieve one. This multipart upload is then stored inside S3FileSystemProvider. Then we start storing data into the buffer, when buffer proceeds uploadBatchSize parameter, we send the part to the AWS. This process repeats as many times as the buffer gets filled. When every data from the file is sent from the client to the server, the close command is sent to the server. During this, we flush everything that is the inside buffer(if any data are still not sent to AWS) and inform AWS to finish the multipart upload. After this, the file is uploaded.

When parallel flag is set in configuration, each upload is handled in thread meaning we can upload the data and at the same time recieve another data from SFTP and starts to fill another buffer. When the close is issued, we wait for all threads to finnish uploading and then we finnish upload same way we finnish it using non parallel version.

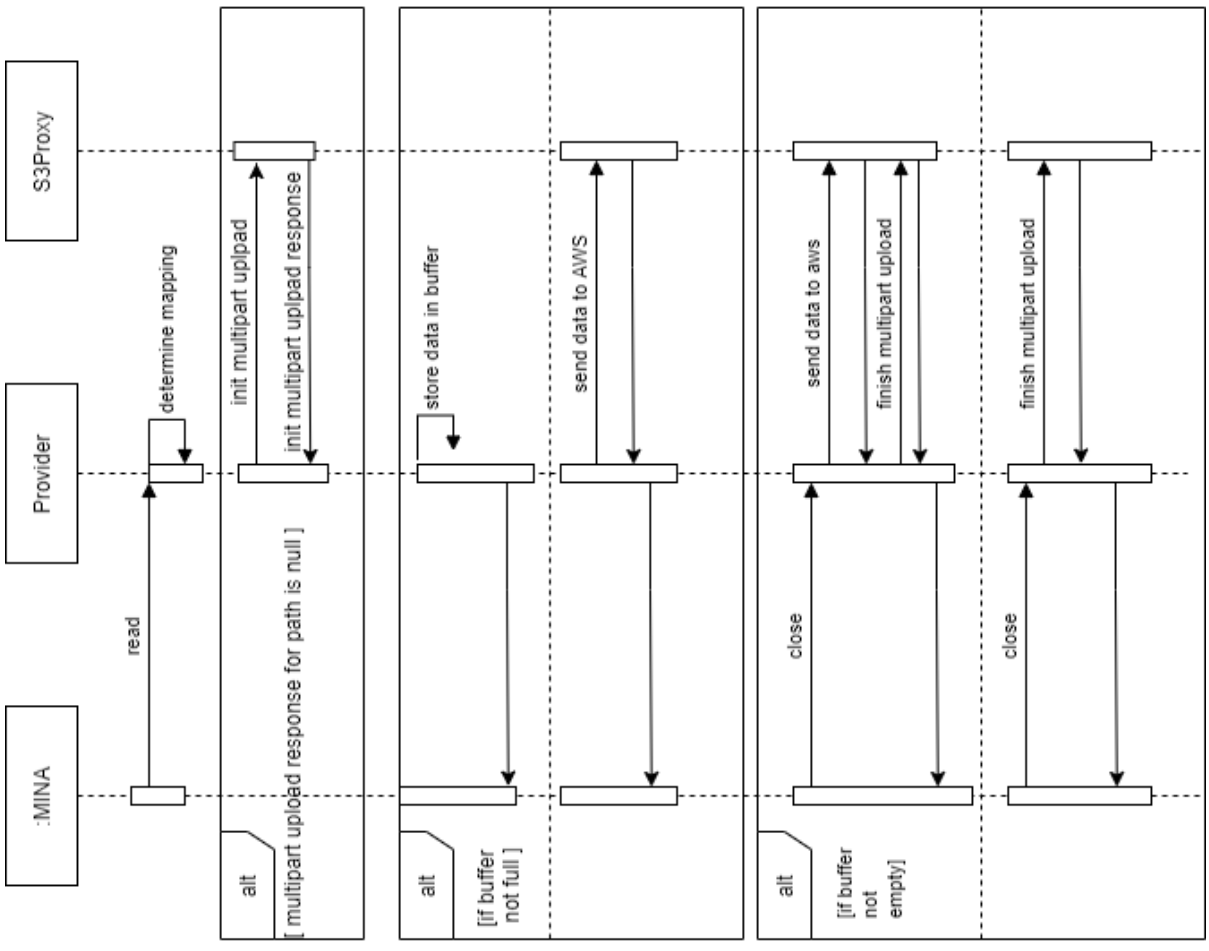

Figure 5.8: Diagram of happy flow of uploading files

5.9.7 Creating directories

When we create file in AWS with name that represents directories, these directories are shown in AWS console. For example if we create file */dir1/dir2/file.txt* both, dir1 and dir 2 are shown inside the AWS console. But they are not directories, as soon as we delete file */dir1/dir2/file.txt*
both directories are deleted too since they serve no other real purpose. We want to have directories that persists, that can contain zero files and be empty. In order to achieve this, we can upload empty file, with empty metadata. Doing this we create directory that persists even if all files inside are deleted. When MINA recieves mkdir command, it delegates the command to S3FileSystemProvider that creates directory - a file with content length 0.

5.9.8 Deleting

There are two cases when considering deleting the file. It's either a regular file or directory. The first thing we need to do is to check what kind of file we are deleting. We retrieve objects from AWS as discussed earlier and determine if we are dealing with the directory or regular file. If the file to be deleted is a regular file, we simply construct the AWS path and use S3Proxy to delete the file. As per standard, the directory cannot be deleted if not empty. If we are dealing with directory we must first check if the directory is empty. We achieve this by trying to list files located in this directory using S3Proxy. If the directory isn't empty, we throw an exception indicating it is not empty, otherwise, we delete the directory. In order to delete the directory that is not empty, the client must send recursive delete - delete all items inside and then directory itself.

5.9.9 Renaming and moving files

According to the SFTP standard, SFTP does not know support command, but rename suits this functionality. Similar to SFTP, AWS does not know the concept of renaming or moving files. What we can do, however, is to copy the file with a different name, and delete the original file. Renaming or moving regular file is simple, we just rename it and do checks if file like that already exists. When renaming or moving directories we must first consider the structure of files. As mentioned earlier, AWS does not know the concept of directories, meaning when we have a structure similar to this:
/bucket1/dir1/file1
and we rename
dir1
to
dir2
path */bucket1/dir1/file1*
still exists, because dir1 does not represent the directory, it is part of the full

name of the file. This means, when moving or renaming directory, it is not enough to just rename or move the directory itself, we must do the same action for every file that is ”contained” in this directory, in order to move the whole directory. When MINA receives rename or move command and delegates it to S3FileSystemProvider, we first check if its regular file or directory. If its regular file we simply copy objects with the new name and delete the old one. If it is the directory, we first list every file contained inside. Then we proceed to determine the new name of each file and copy the file. When we are copying files with the big path (e.g path containing directories) we first create each directory, and then copy the final object - ensuring we create persistent directories and not ”AWS” directories that cease to exists after files inside are deleted.

5.9.10 Clearing reasources

When we prematurely stop downloading file, or upload wont finish(it will not tell AWS to finish multipart-upload) we need to clear resources, either clear buffer or abort the multipart upload. In case of canceling uploading or downloading file, close() command is sent to server from client. This command is sent in both cases, prematurely canceling or finnishing. For this type of closing FileChannel provides implCloseChannel() method, where we can clear resources in case of downloading, and finnish multipart upload in case of uploading file. When finnishing multipart-upload, we need to first check if buffer is empty. If not we send data left in buffer as another part of multipart-upload and then finnish it. However in case connection closes or client disconnected, filesystem's close() method is used. In this methos all unfinished multipart uploads are aborted. The same can also be acomplished by configuring AWS to automatically abort multipart uploads after some period of time.

Chapter 6

Performance

6.1 Goal

The goal of this section is to demonstrate the difference in times it takes to download and upload the file. The extended functionality of MINA is compared to uploading/downloading files to AWS using just java code, without SFTP. Multiple sizes of the buffers are tested and demonstrated how they affect the speed of download/upload. The functionality is also compared to the Amazon Web Services Command Line Interface(AWS CLI) - to default command line way how to upload or download the files.

6.2 The way of measuring

The larger file will be selected with a set size. The size of the file should be bigger, for we better observe how the download and upload differ and act. A smaller file may result in a non-sufficient observation. Every way of uploading the data was done multiple times and the resulted time was averaged.

We will divide file to

- 1 whole file
- 2 equally large halves the file
- 4 equally large parts the file
- 20 equally large parts the file
- 200 equally large parts of the file.

6.3 Enviroment

Amazon Elastic Compute Cloud (EC2) is a part of Amazon.com's cloud-computing platform, Amazon Web Services (AWS), that allows users to rent virtual computers on which to run their own computer applications. The instance that was used for measuring was available on a free tier.

name of parameter	**value**
vCPUs	1
RAM (GiB)	1
Memory (GiB)	8

Table 6.1: Table of EC2 specifications

AWS EC2 was selected as a way to make maximum usage of fast access to S3 and reduce latency as much as possible.

6.4 Measuring

During the measuring of the times, the file size of 2gb was selected. The performance test divided the test into 4 categories:

- uploading 2gb as the whole file
- uploading 2 equally large files with the total size of 2gb
- uploading 4 equally large files with the total size of 2gb
- uploading 20 equally large files with the total size of 2gb
- uploading 200 equally large files with a total size of 2gb.

This method is more close to the real-world usage of transferring files. Even if we eventually upload the same size of data, the more files we upload the more side work we need to do besides uploading. In the SFTP, for example, we need to first initiate the open request, write data and in the end, we need to send the close request. This is the reason why times may differ, even if we are uploading the same size of data.

6.4.1 Upload

During the uploading, the following times were measured:

number of files	**5mb**	**50mb**	**100mb**	**200mb**	**parallel**	CLI
1	115.624s	94.113s	92.243s	91.832s	89.9.447s	34.733s
2	117.912s	94.547s	92.833s	92.021s	90.302s	35.421s
4	118.952s	94.898s	93.140s	92.328s	90.311s	35.723s
20	123.221s	100.509s	95.186s	94.033s	93.704s	36.128s
200	167.454s	155.491s	166.275s	169.201	146.609s	36.848s

Table 6.2: Table of upload times on EC2 using multiple files

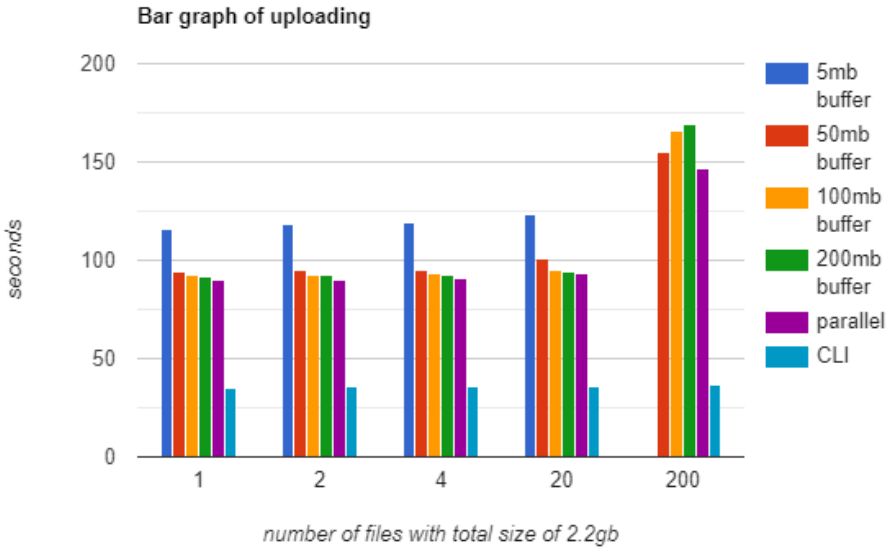

Figure 6.1: Bar graph upload times on EC2 using multiple files

In the following graph, we can observe the approximate speed of transferring data during the upload:

number of files	**5mb**	**50mb**	**100mb**	**200mb**	**parallel**	CLI
1	17.3Mb/s	21.3Mb/s	21.7Mb/s	21.8Mb/s	22.2Mb/s	57.6Mb/s
2	16.9Mb/s	21.2Mb/s	21.5Mb/s	21.7Mb/s	22.1Mb/s	56.4Mb/s
4	16.8Mb/s	21.1Mb/s	21.8Mb/s	21.7Mb/s	22.1Mb/s	55.9Mb/s
20	16.2Mb/s	19.9Mb/s	21.1Mb/s	21.7Mb/s	21.3Mb/s	55.3Mb/s
200	11.9Mb/s	12.9Mb/s	12.1Mb/s	11.8Mb/s	13.6Mb/s	54.3Mb/s

Table 6.3: Table of the transfering speed of the upload

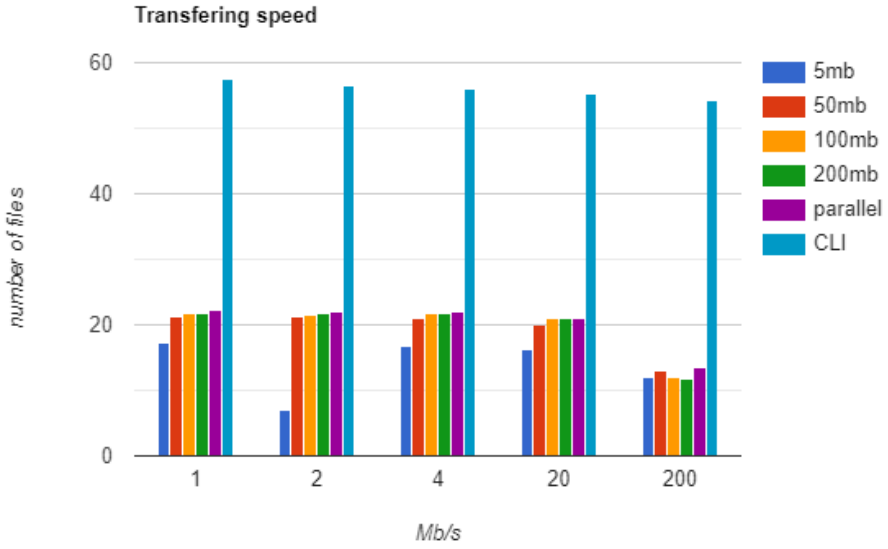

Figure 6.2: Bar graph of the transfering speed of the upload

During the parallel approach, 20threads and 20mb buffer were used.

6.4.2 Download

During the downloading, the following times were measured:

number of files	5mb	50mb	100mb	200mb	parallel	CLI
1	111.825s	88.912s	87.328s	86.504s	88.008s	32.806s
2	111.535s	94.422s	92.822s	92.031s	88.113s	33.012s
4	113.729s	94.837s	93.990s	93.007s	92.122s	34.268s
20	115.009s	96.881s	95.698s	95.064s	94.994s	34.322s
200	172.381s	150.011s	151.601s	151.209s	152.988s	35.625s

Table 6.4: Table of download times on EC2 using multiple files

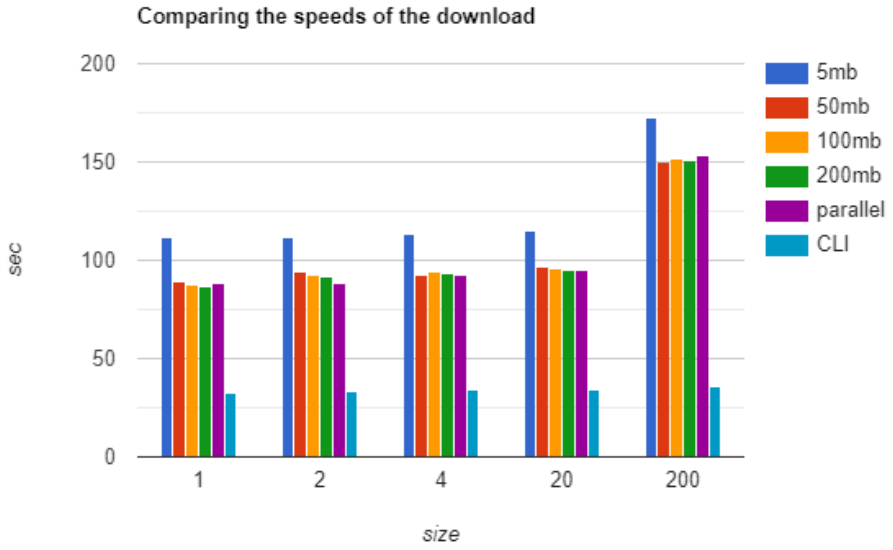

Figure 6.3: Bar graph of download times on EC2 using multiple files

During the parallel approach, 20threads and 200mb buffer were used.

number of files	5mb	50mb	100mb	200mb	parallel	CLI
1	17.9Mb/s	22.5Mb/s	22.9Mb/s	23.1Mb/s	22.7Mb/s	60.9Mb/s
2	17.9Mb/s	21.2Mb/s	21.5Mb/s	21.7Mb/s	22.7Mb/s	60.6Mb/s
4	17.6Mb/s	21.1Mb/s	21.3Mb/s	21.5Mb/s	21.7Mb/s	58.4Mb/s
20	17.3Mb/s	20.6Mb/s	20.9Mb/s	21Mb/s	21.0Mb/s	58.3Mb/s
200	11.6Mb/s	13.3Mb/s	13.2Mb/s	13.2Mb/s	13Mb/s	56.1Mb/s

Table 6.5: Table of the transfering speed of the download

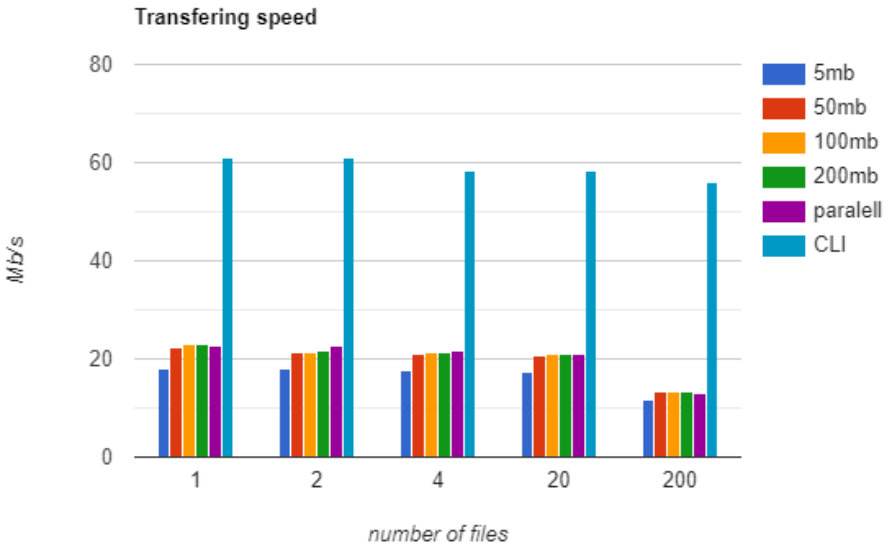

Figure 6.4: Bar graph of the transfering speed of the download

6.4.3 First observation

We can see the CLI approach is much faster than the implemented approach using SFTP. There are multiple reasons why this call is faster:

- AWS CLI is using native calls(boto python) which are going to be a bit faster
- when using AWS CLI we have established the client connection in java we must first establish this connection(s)

CLI also uses different technology. CLI uses TransferManager where in java code, we use AmazonS3 client. For example for uploading the file, the transfer manager requires the whole file to be passed to it, and use threads to upload the file. It also knows the size of the whole file beforehand, as it requires the whole file to work, thus it can support random access to the data and optimize the upload. It, however also uses multi-part upload as used in java code. The requirement of the whole file prevents the java code to use it, as SFTP sends a chunk of files to the server.

6.4.4 More accurate way

Comparing the speed of CLI with an implemented solution is not sufficient. We are basically comparing two different ways of uploading data. Better mea-

surement of the performance is to take java code that uploads data to AWS and run it without SFTP overheat. The uploading to SFTP does not consist only of the opening file, writing the file, and closing file. SFTP can invoke more requests during the uploading of the file(for example stat request). If we measure only the time of manipulation with S3FileChannel(open S3FileChannel, write data, close S3FileChannel), we can measure the time it takes to process and upload incoming data without adding overheat of SFTP. This way we can compare measured times and find out the difference in performance.

Following times were measured for uploading of files using this approach:

number of files	**java**	**proxy upload**	**sftp**	**sftp overheat**
1	64.893s	65.104s	117.967	52.763s
2	65.020s	65.297s	121.516s	56.419s
4	68.171	68.512	123615	55.103s
20	69.263s	69.848	124385	55.137s
200	107.153s	109.914s	172.554s	65.64

Table 6.6: Table of upload times on EC2 with overheat

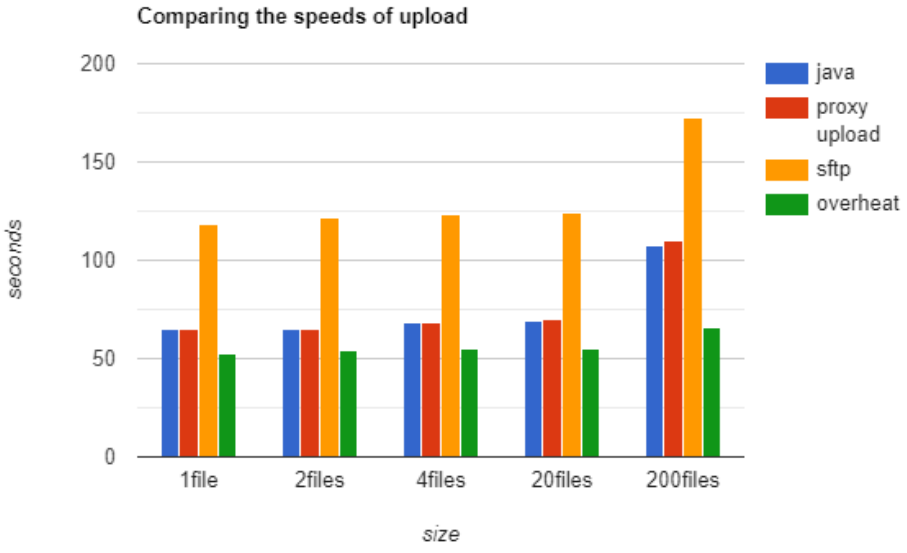

Figure 6.5: Bar graph of upload times on EC2 with overheat

As we can observe, both approaches averaged at similar times. The reason why SFTP adds such a large overheat can be caused by the implementation of the library itself. We must also take into consideration, that SFTP calls

AWS every time it needs to check if file exists, read its attributes, etc.(these calls are internally used multiple times) that adds to this overheat.

During the download the following times were measured:

number of files	java	proxy download	sftp	sftp overheat
1	32.469s	33.828s	117.967	52.763s
2	33.486s	33.880s	121.516s	54.150s
4	33.747s	33.948s	123.615s	55.103s
20	34.863s	35.248s	124.385s	55.137s
200	43.291s	45.610s	172.554s	65.64

Table 6.7: Table of download times on EC2 with overheat

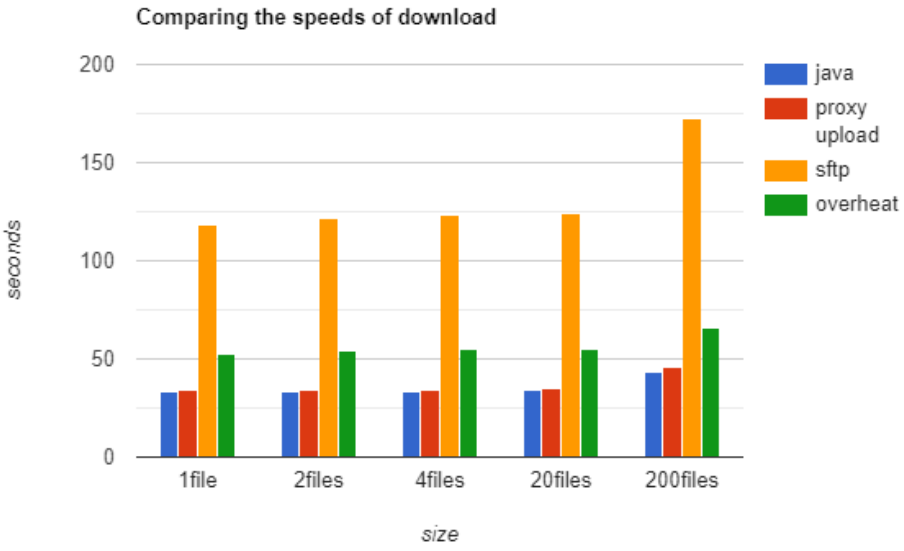

Figure 6.6: Bar graph of download times on EC2 with overheat

Same as with upload, the average of resulted times does not differs in big way.

6.4.5 Comparing the overheat of SFTP

In both cases, download, and upload we measured SFTP overheat. We can validate this overheat by adjusting SFTP not to communicate with AWS. We measure the time it takes to upload the file, however, in S3FileChannel, we do nothing. It is preferred to measure this trough uploading the file. The client sends data to the server as long as all of the data weren't send to the server.

When we download files, the client issues read requests as long as EOF is not returned from the server. For this reason, we can better simulate this using uploading.

In the following table, we compare SFTP overheat we calculated during upload of the files in the previous section and newly measured overheat without AWS calls

number of files	calculated overheat	measured overheat
1	53.763	55.507s
2	54.150	55.040s
4	55.103s	55.317s
20	55.137s	55.248
200	65.64s	66.569s

Table 6.8: Table of compared overheats

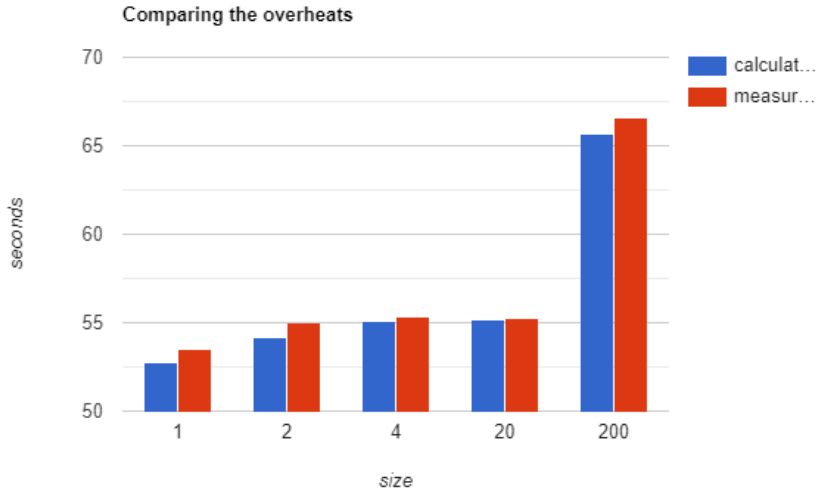

Figure 6.7: Bar graph of compared overheats

As seen in the graph, the times do not differ in a big way.

As the next step, we can measure the time it takes MINA to upload the file using its default functionality and measure the time it takes to open the file, write to file, and close the file.

That way we can measure overheat in its default implementation and compare it.

number of files	mina	open/write/close	sftp overheat
1	58.227s	4.498s	53.729s
2	58.347	4.547s	53.800s
4	59.949	6.993s	52.956s
20	62.868s	7.219s	55.649s
200	73.973s	8.990s	64.983s

Table 6.9: Table of overheats using MINA's default functionality

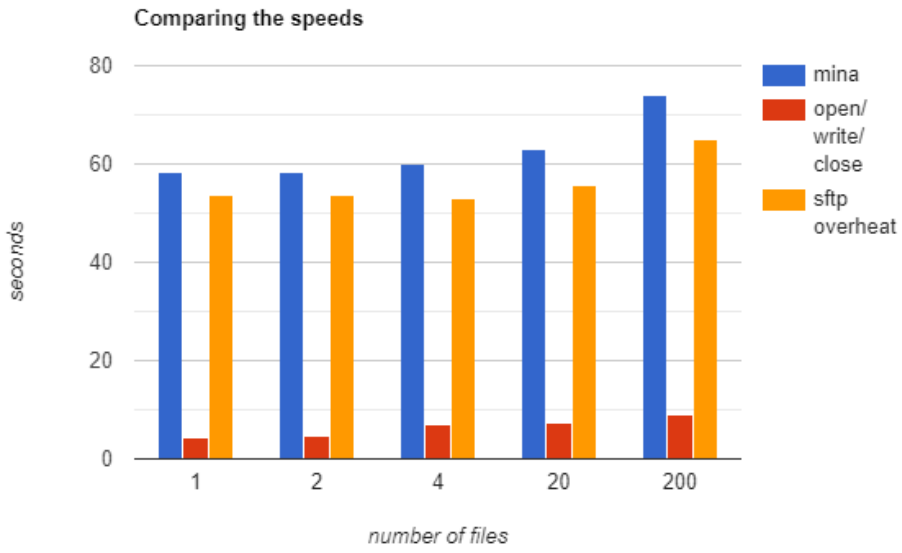

Figure 6.8: Bar graph of overheats using MINA's default functionality

As seen from the table and graph, using MINA's default implementation also results in quite a big SFTP overheat when uploading or downloading files.

Lastly, we can compare the measured overheats from MINA's default functionality, and from the application itself.

number of files	mina	proxy
1	53.729s	55.507
2	53.800s	55.040
4	52.956s	55.317
20	55.649s	55.248
200	64.983s	66.659

Table 6.10: Table of the comparasion of the overheats

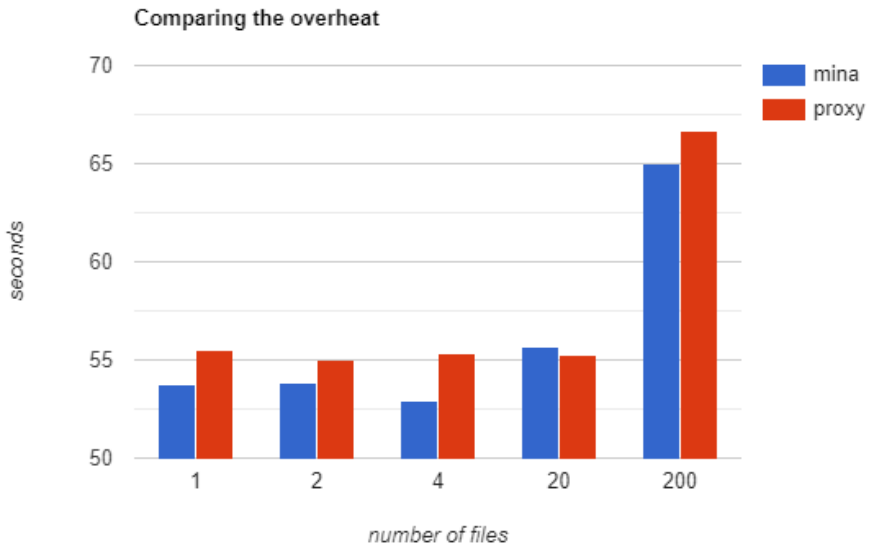

Figure 6.9: Bar graph of the comparasion of the overheats

We can observe that the overheats in both cases are close. From these performance tests we can assume the implemented proxy's speed is comparable with java approach, however keeps the overheat of the MINA and SFTP.

Chapter 7

Testing

Program testing is a basically destructive process, where we try to find the errors in the program. Successful test case helps us furthers progress in this direction by causing the program to fail.[22]

Eventually, we want to establish the level of functionality that ensures the program will follow our desired rules and behaves as expected in a deterministic way. It is hard and we can even say impossible to create test cases for every functionality of our program. The bigger the program the more impossible task we would try to achieve by covering every edge case of functionality with tests.

By testing, we can determine if we

- correctly understood all functional and non-functional requirements.
- tested software's architecture is adequate.
- identified all requirements for the software.
- overlooked flaws in the design.

The tests were created prior to implementation and expected to fail. As the implementation was done the tests were expected to pass. The test defined the expected behavior of the unit or service they were testing. The tests were testing the functionality of all classes and services. For the testing purpose, JUnit, Mockito, and JSch were used. Mockito was used for mocking classes and for implementing matchers. JSch was used as an SFTP client for tests. Before every test, the requirements were configured. If the test needed FileSystem for its functionality, the FileSystem was created. If the test needed to communicate with the actual SFTP server, the server was configured and created. If it needed mocked functionality, the class was mocked. This was all done in method with @Before annotation, thus invoking before every test. Example of the setup requirements for running test:

```
@Before
public void setUp() throws Exception {
   int port = generateRandomPort();
   sftpServer = createSftpServer(port, new
       S3ProxyFactoryImpl(this::createAmazonS3Client));
   sftpServer.start();

   sftp = new JSch();
   sftp.addIdentity(Paths.get(ClassLoader.getSystemResource(name).toURI()).toString());

   Session session = sftp.getSession("test", "localhost", port);
   session.setConfig("StrictHostKeyChecking", "no");
   session.setConfig("PreferredAuthentications", "publickey");
   session.connect();

   sftpChannel = (ChannelSftp) session.openChannel("sftp");
   sftpChannel.connect();
}
```

In this configuration we do not want to check if the server we are connecting to is known host, that is the reason we set StrictHostKeyChecking property to no.

After every test, we must ensure the connection is closed.

```
   @After
   public void cleanUp() throws IOException {
      sftpChannel.disconnect();
      sftpServer.stop();
   }
```

7.1 Unit testing

Unit testing is the method of testing where the units in the program are tested individually. Developers usually write unit tests to document requirements and detect if functionality matches with wanted results. Unit tests can also reveal flaws in design.[23]

7.1.1 Testing filesystem

In tests handling testing functionality of implemented filesystem, multiple tests were created. The first and easiest components that were tested were S3PathTest and S3FileSystemTest, testing as the name suggest S3Path and S3FileSystem. Both tests are unit tests, they do not require any mocking. However, S3PathTests needs the S3FileSystem in order for it to work as expected.

Each of these tests is testing all methods that are provided by the interface these classes implement.

Listing 7.1: Example of unit test

```
@Test
public void testGetPath() throws Exception {
   S3Path path = fileSystem.getPath("/");
   Assert.assertTrue(path.getRoot().toString().equals("/"));
   Assert.assertTrue(path.getNameCount() == 0);

   S3Path path2 = fileSystem.getPath("a");
   Assert.assertTrue(path2.getRoot() == null);
   Assert.assertTrue(path2.getNameCount() == 1);

   S3Path path3 = fileSystem.getPath("/a");
   Assert.assertTrue(path3.getRoot().toString().equals("/"));
   Assert.assertTrue(path3.getNameCount() == 1);
}
```

Next, S3FileChannelTest was implemented. This test tests the functionality of reading and opening files. This test needs S3Proxy in order to be able to write or read data. S3Proxy is mocked using mockito, to return wanted values and act as the real implementation of S3Proxy. In each test method we also check the exact number of times, each S3Proxy method is called, to ensure we do not call proxy methods more time than needed.

Some methods of S3Proxy require specific request as parameters. In order to mock this method, each specific request had his own matcher implemented.

Listing 7.2: Example of mocked test

```
@Test
public void testRead() throws Exception {
   String contentPart1 = "testu";
   String contentPart2 = "01234";

   List<S3ObjectSummaryWrapper> s3ObjectSummaries = Collections
   .singletonList(createS3ObjectSummaryWrapper("testiik", "file1",
       10, new Date()));

   Mockito.when(proxy.listObjects("file1", "testiik"))
   .thenReturn(createS3ObjectListing(EMPTY_DIRECTORY_LIST,
       s3ObjectSummaries));

   Mockito.when(proxy.getObject("testiik", "file1", 0,
       5)).thenReturn(createS3ObjectWrapper(contentPart1));
   Mockito.when(proxy.getObject("testiik", "file1", 5,
       5)).thenReturn(createS3ObjectWrapper(contentPart2));

   S3Path path = fileSystem.getPath("/a/b/testiik/file1");
   List<S3toSftpMapping> mapping =
       account.getAwsOpenMapping(path.toString());
   S3FileChannel channel = new S3FileChannel(path,
       Collections.singleton(StandardOpenOption.READ),
       mapping.get(0));

   byte[] data = new byte[5];
   ByteBuffer byteBuffer = ByteBuffer.wrap(data, 0, 5);

   channel.read(byteBuffer);

   byte[] data2 = new byte[5];
   ByteBuffer byteBuffer2 = ByteBuffer.wrap(data2, 0, 5);

   channel.read(byteBuffer2);

   Assert.assertTrue(new String(data).equals(contentPart1));
   Assert.assertTrue(new String(data2).equals(contentPart2));
}
```

Lastly, S3FileSystemProviderTest is testing the functionality of all other SFTP functions. This test also uses the advantage of mocking and mocks S3Proxy. Since here we do not call S3Proxy directly, we check the number of calls to the specific S3Proxy methods we need.

Listing 7.3: Example of mocked test

```
@Test
public void testRename() throws Exception {
   Mockito.when(proxy.listObjects("file1","testiik"))
.  .thenReturn(createS3ObjectListing(EMPTY_DIRECTORY_LIST,
    Arrays.asList(createS3ObjectSummaryWrapper("testiik","file1",4000,null,"tag"))));

   Mockito.when(proxy.listObjects("file2","testiik"))
   .thenReturn(createS3ObjectListing(EMPTY_DIRECTORY_LIST,EMPTY_OBJECTSUMMARY_LIST));

   Mockito.when(proxy.listObjects("file2/","testiik"))
   .thenReturn(createS3ObjectListing(EMPTY_DIRECTORY_LIST,EMPTY_OBJECTSUMMARY_LIST));

   S3Path path = fileSystem.getPath("/a/b/testiik/file1");
   S3Path path2 = fileSystem.getPath("/a/b/testiik/file2");
   provider.move(path, path2);

   Mockito.verify(proxy, Mockito.times(1)).listObjects("file1",
       "testiik");
   Mockito.verify(proxy, Mockito.times(1)).listObjects("file2",
       "testiik");
   Mockito.verify(proxy, Mockito.times(1)).listObjects("file2/",
       "testiik");
   Mockito.verify(proxy, Mockito.times(1)).copyObject("testiik",
       "file1", "testiik", "file2");
   Mockito.verify(proxy, Mockito.times(1)).deleteObject("testiik",
       "file1");

   Mockito.verifyNoMoreInteractions(proxy);
}
```

7.2 Mapping Test

In S3ToSftpMappingTest, the mapping and its functionality are tested. We test if the mapping can return next local dir if we are considering local path, if we can determine the correct mapping and if we can construct the correct AWS path from the local path.

7.3 Util Test

In SftpUtilTest, the methods that resolve around parsing and retrieving bucket and file name from received SFTP path are tested.

7.4 Integration Test

The next step in the testing process is the integration of units. Program build from units that work probably and has been tested should function when tested. But it may not be true. This is because of the differences in the types of errors that one discovers at various levels of testing. When the testing unit, the developer's focus is mainly on algorithmic aspects, testing if the unit performs the required function. The goal of integration testing is to test the interaction between particular units.[24]

In other words, integration tests determine if all previously developed and tested units of software work correctly together as part of the system. It does not matter what approach is used for creating software, every time the development comes to the point where it needs to test the communication between each unit and the functionality as a whole unit composed of smaller units integration tests are used.

Listing 7.4: Example of integration test

```
@Test
public void testListing() throws SftpException {
   Vector firstEntries = sftpChannel.ls("/");
   sftpChannel.cd("/");

   List<LsEntryMatcher> matchers1 =
       Arrays.asList(createLsEntryMatcher(".", 0, "drw-------"),
   createLsEntryMatcher("a", 0, "drw-------"));

   List<ChannelSftp.LsEntry> entries1 = new ArrayList<>(firstEntries);
   Assert.assertThat(entries1,
       Matchers.containsInAnyOrder((Collection) matchers1));

   Vector secondEntries = sftpChannel.ls("a");
   sftpChannel.cd("a");

   List<LsEntryMatcher> matchers2 =
       Arrays.asList(createLsEntryMatcher(".", 0, "drw-------"),
   createLsEntryMatcher("..", 0, "drw-------"),
       createLsEntryMatcher("b", 0, "drw-------"));

   List<ChannelSftp.LsEntry> entries2 = new
       ArrayList<>(secondEntries);
```

```
  Assert.assertThat(entries2,
      Matchers.containsInAnyOrder((Collection) matchers2));

  Vector thirdEntries = sftpChannel.ls("b");
  sftpChannel.cd("b");

  List<LsEntryMatcher> matchers3 =
      Arrays.asList(createLsEntryMatcher(".", 0, "drw-------"),
  createLsEntryMatcher("..", 0, "drw-------"),
      createLsEntryMatcher("testiik", 0, "drw-------"));

  List<ChannelSftp.LsEntry> entries3 = new ArrayList<>(thirdEntries);
  Assert.assertThat(entries3,
      Matchers.containsInAnyOrder((Collection) matchers3));

  try {
     sftpChannel.cd("wrong");
     Assert.fail("should fail");
  } catch (Exception e) {
     Assert.assertThat(e.getMessage(), Matchers.equalTo("No such file
         or directory"));
  }

  try {
     sftpChannel.ls("wrong");
     Assert.fail("should fail");
  } catch (Exception e) {
     Assert.assertThat(e.getMessage(), Matchers.equalTo("No such file
         or directory"));
  }

  Vector d = sftpChannel.ls("testiik");
  List<ChannelSftp.LsEntry> resultList = new ArrayList<>(d);
  Assert.assertThat(resultList,
      Matchers.hasItem(Matchers.hasProperty("filename",
      Matchers.is("c"))));
}
```

Conclusion

I have analyzed the applications that could be used as a replacement for this thesis. I have analyzed AWS S3 storage, the SFTP protocol, and multiple ways to extend the functionality of APACHE MINA. Methods were discussed and the best method was used to implement the objective of the thesis. After implementation, the application was properly tested and measured, assuring correct functionality and proper performance. The usage of direct communication with AWS using the command line interface is faster, which is expected, however, the created application offers comfortable abstraction over SFTP protocol, cost-free, and with an adequate speed of download and upload. Using more powerful instances on EC2 will result in better performance that keeps the difference in speed between CLI and this thesis the same, however, it could make it negligible. After further performance testing, I discovered the download and upload speed using a proxy is approximately the same as using just java code with the same functionality, however, SFTP and MINA add overheats that affect the performance. In further works, other SFTP library could be selected and tested performance using this library.

This thesis is a good choice for integration in applications, where SFTP protocol was previously used, and the server was replaced with AWS S3 storage, which provides an advantage over standard CLI.

In the future, the application could be extended to offer support for more cloud storage or added database for extended functionality. More types of accounts could be added, each representing different cloud storage or way of communicating with said storage.

I think the objective of the thesis was fulfilled. As the product of this thesis will be published as open-source, I believe many developers will find it useful.

Bibliography

[1] https://www.aws.amazon.com. AWS Transfer Family[online]. 2020 [cit. 2020-07-27]. Avaible at: https://aws.amazon.com/aws-transfer-family/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc

[2] Comer,DOUGLAS E. Internetworking with TCP/IP Vol. I: Principles, Protocols, and Architecture. New Jersey: Prentice-Hall International, 1991. ISBN 978-0132169875.

[3] YLONEN, T. a C. LONVICK. The Secure Shell (SSH) Protocol Architecture[online]. 2006 [cit. 2020-07-27]. Avaible: https://tools.ietf.org/html/rfc4251

[4] STALLINGS, William. Cryptography and Network Security. : Principles and Practice. Prentice Hall, 1999. ISBN 9780138690175.

[5] BROWN, L. Secure file transfer over TCP/IP.[online]. 1992 [cit. 2020-06-20]. Avaible at: https://ieeexplore.ieee.org/document/271896.

[6] Barrett, Daniel; Silverman, Richard E., SSH, The Secure Shell: The Definitive Guide, Cambridge: O'Reilly, 2001. ISBN 0-596-00011-1.

[7] https://www.aws.amazon.com. What is Cloud Computing?[online]. 2020 [cit. 2020-07-27]. Avaible at: https://aws.amazon.com/what-is-cloud-computing

[8] Varia, Jinesh, and Sajee Mathew. Overview of amazon web services [online] 2014 [cit. 2020-07-27]. Avaible at: http://cabibbo.dia.uniroma3.it/asw-2014-2015/altrui/AWS_Overview.pdf

[9] w3.org. Web Services Architecture[online]. 2004 [cit. 2020-07-27]. Avaible at: https://www.w3.org/TR/ws-arch/

[10] Gadir, O.M., Subbanna, K., Vayyala, A.R., Shanmugam, H., Bodas, A.P., Tripathy, T.K., Indurkar, R.S. and Rao, K.H. High-availability cluster virtual server system[online]. 2005 [cit. 2020-07-27]. Avaible at: https://patentimages.storage.googleapis.com /5a/80/46/dbc7b6753bfdd5/US6944785.pdf

[11] Vogels, W. Eventually consistent[online]. 2009 [cit. 2020-07-27]. Avaible at: https://dl.acm.org/doi/10.1145/1435417.1435432

[12] https://aws.amazon.com/s3/. Amazon s3 [online]. 2020 [cit. 2020-07-27]. Avaible at: https://aws.amazon.com/s3/

[13] https://docs.aws.amazon.com[online]. 2020 [cit. 2020-07-27]. Avaible at: https://docs.aws.amazon.com/AmazonS3/ latest/dev/Introduction.html

[14] World-Wide Web Proxies[online]. 1994 [cit. 2020-07-27]. Avaible at: http://courses.cs.vt.edu/ cs4244/spring.09/documents/Proxies.pdf

[15] www.computerweekly.com. Write once, run anywhere?[online]. 2002 [cit. 2020-07-27]. Avaible at: https://www.computerweekly.com/feature/Write-once-run-anywhere

[16] Oracle Corporation. "1.2 Design Goals of the Java™ Programming Language"[online]. 1999 [cit. 2020-07-27]. Avaible at: https://www.oracle.com/java/technologies/introduction-to-Java.html

[17] Https://gitlab.com. About Us [online]. 2020 [cit. 2020-07-27]. Avaible on: https://about.gitlab.com/features/gitlab-ci-cd/

[18] Https://mina.apache.org. "About" [online]. 2020 [cit. 2020-07-27]. Dostupné z: https://mina.apache.org/sshd-project/

[19] Mcintosh, Shane, Bram Adams, and Ahmed E. Hassan. The evolution of Java build systems[online]. 2011 [cit. 2020-07-27]. Avaible at: https://link.springer.com/article/10.1007/s10664-011-9169-5

[20] Kotonya, Gerald; Sommerville, Ian. Requirements Engineering: Processes and Techniques. Chichester, UK: John Wiley and Sons. 1998. ISBN 9780471972082.

[21] OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2[online]. 2009 [cit. 2020-07-27]. Avaible at: https://www.omg.org/spec/UML/2.2/Superstructure/PDF

[22] Myers, Glenford J., Corey Sandler, Tom Badgett. The art of software testing. John Wiley & Sons, 2011.

[23] N. Tillmann ; W. Schulte, Unit tests reloaded: parameterized unit testing with symbolic execution[online]. 2006 [cit. 2020-07-27]. Avaible at:https://ieeexplore.ieee.org/abstract/document/1657937

[24] M.E. Delamaro ; J.C. Maidonado ; A.P. Mathur. Interface Mutation: an approach for integration testing[online]. 2001 [cit. 2020-07-27]. Avaible at: https://ieeexplore.ieee.org/abstract/document/910859

Appendix A

Acronyms

CI Continous Integration

AWS Amazon Web Services

CLI Command Line Interface

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format