

A file-sharing platform in private cloud

Alexandros, Rantos-Charisopoulos

A.Rantos-Charisopoulos2@newcastle.ac.uk

Christos, Grigoriadis C.Grigoriadis@newcastle.ac.uk

Yuchen, Guo Y.Guo32@newcastle.ac.uk

Zhiniu, Wu Z.Wu26@newcastle.ac.uk

Shiwei, Mo S.Mo2@newcastle.ac.uk

Ziqi, Cui Z.Cui7@newcastle.ac.uk

March 28, 2020

Abstract

File sharing is one of the most used services throughout the internet. It has opened a whole new world for people to communicate, exchange information freely, share their progress instantly and much more. However, such a success is always seen as an easy target for exploiters and unethical individual. To tackle such behaviours we present a fair exchange file sharing platform that was implemented using various *Amazon Web Services* () [1] and it is based on Coffey and Saidha fair exchange protocol [2].

1 Introduction

With the rapid development of technology, people are able to exchange files instantaneously via the internet. File sharing platforms, however, must ensure the participating entities that the *downloader* gets the requested file with the corresponding evidence from the *uploader* who must receive the evidence of receipt for every entity that downloads a file of his. The fairness of such platforms are crucial and essential to preserve a trusted environment for everyone to use.

We present a file-sharing system that is based on the *inline* fair exchange protocol of Coffey and Saidha and guarantees fairness for all participating parties. To achieve this we also use some ready solutions provided from AWS that will be analysed later on.

2 Project Requirements

Based on the protocol, as well as the given project specification, our project's basic requirements are as follows:

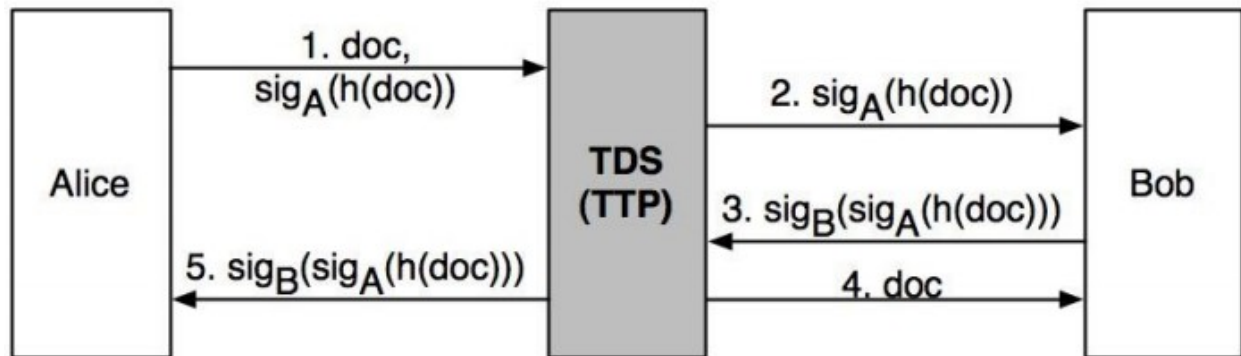


Figure 1: Project specification

The client should be able to:

- (i) Upload a document to the store.
- (ii) Request an identified document, receiving the corresponding signature.
- (iii) Provide a receipt and in return;
- (iv) Download the aforementioned document.
- (v) Get a receipt associated with the downloaded document.
- (vi) Throughout this protocol, fairness should be guaranteed.

After the implementation of the basic system, we extended its functionality by adding:

- (i) Secure communication between participants.
- (ii) Authorisation of actions performed by service clients.
- (iii) Abort exchange functionality.

3 Basic Concepts, Terms & Notation

In this section, the necessary definitions will be given for the comprehension of this report. Furthermore, the notation which will be used later, will also be described.

3.1 Properties

We will be referring to the following properties in our report:

Fairness. Each exchange must give equal rights to both entities and not favour one over another.

Strong fairness. At the end of exchange, both parties must have acquired the corresponding evidence and information of the communication.

Non-reputability. The main property which every protocol must possess. Every party that takes part in an exchange protocol must have a guarantee that either both have proof of the successful exchange or none has. In this way, fairness is also established. In order that to be accomplished, a protocol must provide not only *non-repudiation of origin* but also *non-repudiation of receipt*.

Non-repudiation of origin (NRO). The undisputed evidence of receiver that was involved in an exchange in which the sender acquired the receiver's response.

Non-repudiation of receipt (NRR). The undisputed evidence of the communication initiator that the recipient has participated in the exchange and received his message.

Timeliness. The continuous availability and reachability of any information from each participating party during the protocol execution.

ACID. Stands for Atomicity, Consistency, Isolation, Durability and it is database property which guarantees validity, correctness and consistency for any *transaction*. Transaction is a set of jobs that need to be completed coherently and independently. All sub-tasks should complete or none.

3.2 Digital Signatures

It is crucial for a non-repudiation service to undeniably guarantee the identity of every party that takes part in it. For that reason, each entity is obliged to digitally sign crucial data that transmits. The secret key of each signed entity is used to form its signature and the corresponding public key is used to derive the true sender of a particular message. So the secret key is directly connected to the signature and if the former is compromised, the latter is too.

3.3 Trusted Third Party

This implementation is relying on an external principal in order to satisfy a fair exchange which is dependable and independent. More specifically, we use an *inline* TTP which means that every message exchange is monitored by the TTP. In that way, strong fairness is guaranteed in the cost of performance since TTP's involvement and message processing hinders their fast transition.

3.4 Notation

We use the following notation in this report.

M	: message/file
$H()$: an one-way, collision-resistant Hash Function
A	: sender/uploader
B	: receiver/downloader
$A \rightarrow B : M$: A sends M to B
$Sig_A(M)$: M is digitally signed by A
NRO	: Non-repudiation of origin
NRR	: Non-repudiation of receipt

4 Coffey and Saidha protocol [2]

This is the protocol that our project is based on and it should be noted that in this section the original protocol is presented as is, without any modifications or alterations needed for this project specification. This protocol falls under the inline TTP category and was first introduced in 1996. The basic idea is using TTP as a non-repudiation server (NRS) which handles every message transmission between two parties like shown in the diagram below:

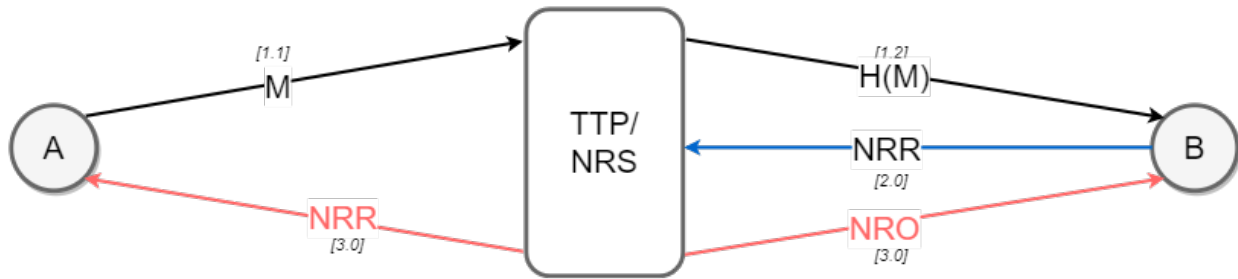


Figure 2: High-Level and simplified depiction of Coffey and Saidha Protocol [4]

For the sake of clarity, the time-stamping authority (TSA) is not shown in Figure 1. This protocol makes use of this authority in order to justify the validity and reliability of messages' time-stamps. This is done not only to minimize any delays from system dependencies such as a large distributed system but also because different CPU clocks cannot be truly synchronized hence providing a correct timestamp is impossible. However, a new responsibility is generated for both of the requesting entities because they must provide a certificate that will be valid until the communication with TSA is completed. For simplicity, we have not implemented TSA.

The initiation of this protocol starts by A collecting the timestamp from TSA who responds by adding the timestamp (t_A) to the message that is signed by A. For TSA

to eliminate any malicious attacks (e.g. eavesdropping) the response is encrypted with the public key of A so only the private key of A can decrypt the message. This technique is used throughout the whole protocol.

1. $A \rightarrow TSA : M_A$, where $M_A = Sig_A(L, A, B, M, EOO)$
2. $TSA \rightarrow A : eV_A(Sig_A(M_A, TSA, t_A))$

If A verifies that the timestamp is indeed valid, he can continue the protocol by opening a communication channel with the selected NRS who, in turn, sends back a *nonce* (number only used once). Signatures are used again to avoid malicious attacks. This nonce is integrated into the A's message which is sent back to NRS. This way NRS is not borne the computational cost of incorporating the nonce into A's message.

3. $A \rightarrow NRS : Request$
4. $NRS \rightarrow A : nonce_A$
5. $A \rightarrow NRS : eV_{NRS}(NRO)$, where $NRO = Sig_A(L, A, B, M, EOO, nonce_A)$

It is now time for NRS to request B's EOR by sending him a new nonce together with the hashed value of EOO that received from A. B have to interact with TSA, similarly as A before, in order to get a valid timestamp (t_B) on the EOR and respond back to NRS with the complete NRR.

6. $NRS \rightarrow B : eV_B(M_B)$, where $M_B = Sig_{NRS}(H(EOO), nonce_B)$
7. $B \rightarrow TSA : eV_{TSA}(M_B)$
8. $TSA \rightarrow B : e_B(NRR)$, where $NRR = Sig_{TSA}(M_B, TSA, t_B)$
9. $B \rightarrow NRS : eV_{NRS}(NRR, nonce_B)$

Since both parties were honest and followed the steps of the exchange successfully, NRS can now give A the Non-repudiation of Receipt and B the Non-repudiation of Origin, both of which remain in the NRS storage for future reference, if needed.

10. $NRS \rightarrow A : eV_A(NRR)$
11. $NRS \rightarrow B : eV_B(NRO)$

Even though the performance bottleneck of the communication channels between the entities and authorities, the above protocol provides a complete solution for a non-repudiation service which means that is also capable of solving any disputes. In such case, a trusted mediator is called to check the collected evidence of the sender, receiver and lastly of the NRS. A successful exchange has taken place if and only if the originator or NRS are able to submit the NRR, or the recipient submits the NRO. In any other instance, the protocol has not proceeded into actual data exchange, therefore no parties have obtained anything from this communication.

5 System design

5.1 Amazon Web Services(AWS) Selection

We briefly describe which services we decided on selecting for our system. At the end of this report, we evaluate them and compare them with other options. We used the following Amazon Web Services:

1. **Simple Queue Service (SQS)**: for communication purposes between entities.
2. **Relation Database service (RDS)**: for persistent data storage.
3. **Key Management Service (KMS)**: for signing and verifying documents.
4. **Simple Storage Service (S3)**: for documents storage.
5. **Elastic Compute Cloud (EC2)**: for an online website.

It should be noted that S3 is managed by the TTP. For that reason, the TTP has *FULL_CONTROL* policy over the buckets and bucket's objects and the Client has only *read/write* access but denied the *DeleteObject* and *DeleteBucket* actions. Same logic applied to the KMS. For the SQS, each party has each own read or write permissions as explained in the following subsection about communication.

5.2 Communication Design

As requested the time-stamping authority is not taken into consideration in this implementation.

In the beginning, A requests permission from the *TDS* (Trusted Document Storage; we also refer to it as TTP) to upload a file. TDS generates a *CMK* (Customer Master Key) which sends to A together with a *nonce* (number only used once).

1. $A \rightarrow TDS : RequestForUpload$
2. $TDS \rightarrow A : CMK_{id_A}, nonce_A$

A answers back to TDS with the NRO while he uploads the file on S3.

3. $A \rightarrow TDS : Sig_A(H(M), file_path, nonce_A)$

Then TDS verifies A's signature & $nonce_A$ and downloads the file from S3.

In order B to download a file, B must send a request to download to TDS who will respond with a nonce like before but will also send the hash of the requested file. B proceeds with signing the message forming the NRR and sends it back to TDS.

4. $B \rightarrow TDS : RequestToDownload$
5. $TDS \rightarrow B : CMK_{id_B}, nonce_B, Sig_A(H(M))$
6. $B \rightarrow TDS : nonce_B, Sig_B(Sig_A(H(M)))$

Once again, TDS verifies the response from B and if valid, TDS sends the NRR and NRO to A and B, correspondingly.

7. $TDS \rightarrow A : NRR$
8. $TDS \rightarrow B : NRO, filePath$

Both parties get the evidence of the exchange and B also downloads the file. Furthermore, TDS firstly updates the values on the database and then sends the messages. This way if any party does not receive the corresponding evidence due to eg. a failure, TDS can look on the database to see if the exchange was successful or not and proceeds by sending the evidence to the requester.

5.3 Database Design

We decided to use the same technology on both ends; MySQL, a relational database. The reasons behind this selection will be explained later on. In this section, only the structure and table schemas of our databases will be described.

5.3.1 Client - Side

Here, the database schema consists of 4 tables. Firstly, we have the *users* table in which the necessary information is stored in order the specific user to be authenticated when logging into the website. The rest of tables are associated with the file exchanges. In *accounts_uploadfile*, we store which files are being uploaded or have successfully uploaded. The *status* field indicates the progress stage of each upload. Explicitly:

$status = 1$: Upload is progressing
 $status = 2$: Successful upload
 $status = 3$: Upload has been aborted

The same logic is used for the *accounts_downloadfile* table in which the requests for download are being stored. Lastly, there is the *account_eceipt* table in which for every successful download a new row is inserted to store the *NRR* for the uploader. The table schema is the following:

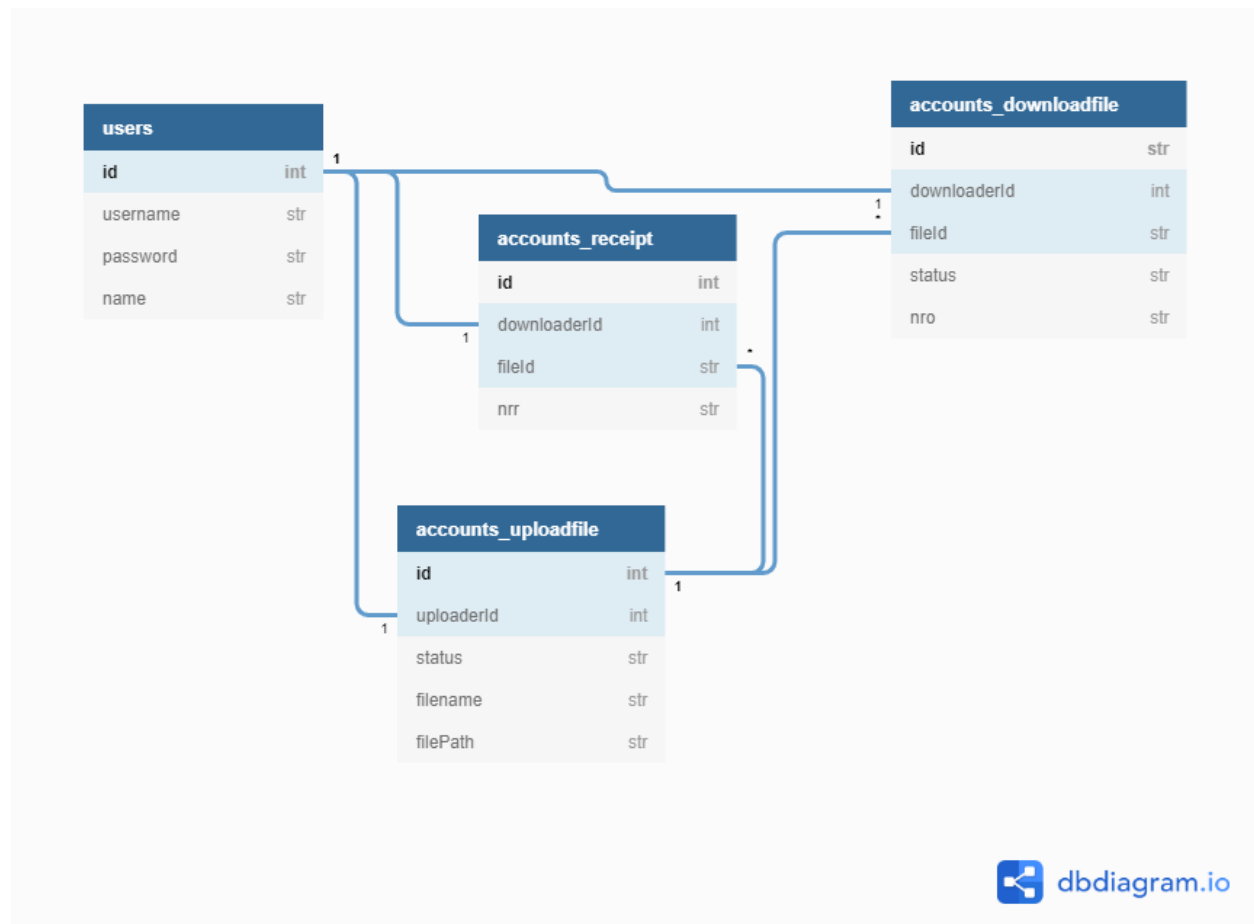


Figure 3: Client database diagram [5]

5.3.2 Server/TTP - Side

From TTP's perspective, the implementation is as follows. For the economy of the report it would be better; first to discuss database interface implementation and then continue with the actual script.

Likewise on the TTP side, we use MySQL. There are 5 distinct tables. Firstly, we use a table(*ongoingUploads*) to monitor the upload progress for each file. When an upload is flagged as successful we delete the corresponding row from the *ongoingUploads* table and insert it to another table called *completeUploads* in which all the uploaded files are stored with the corresponding *NRO* . Similarly, we use a table(*ongoingDownloads*) to store all the downloads requests that are still in progress. A row is been added to that table each time someone requests to download a file. The same row is deleted from it when the exchange is completed and it is added to the *completedExchanges* table. We acknowledge the fact of overcomplication of such design but our goal was to minimize the lookup of successful and in-progress uploads and downloads, hence the two tables for each. The table schema is as shown:

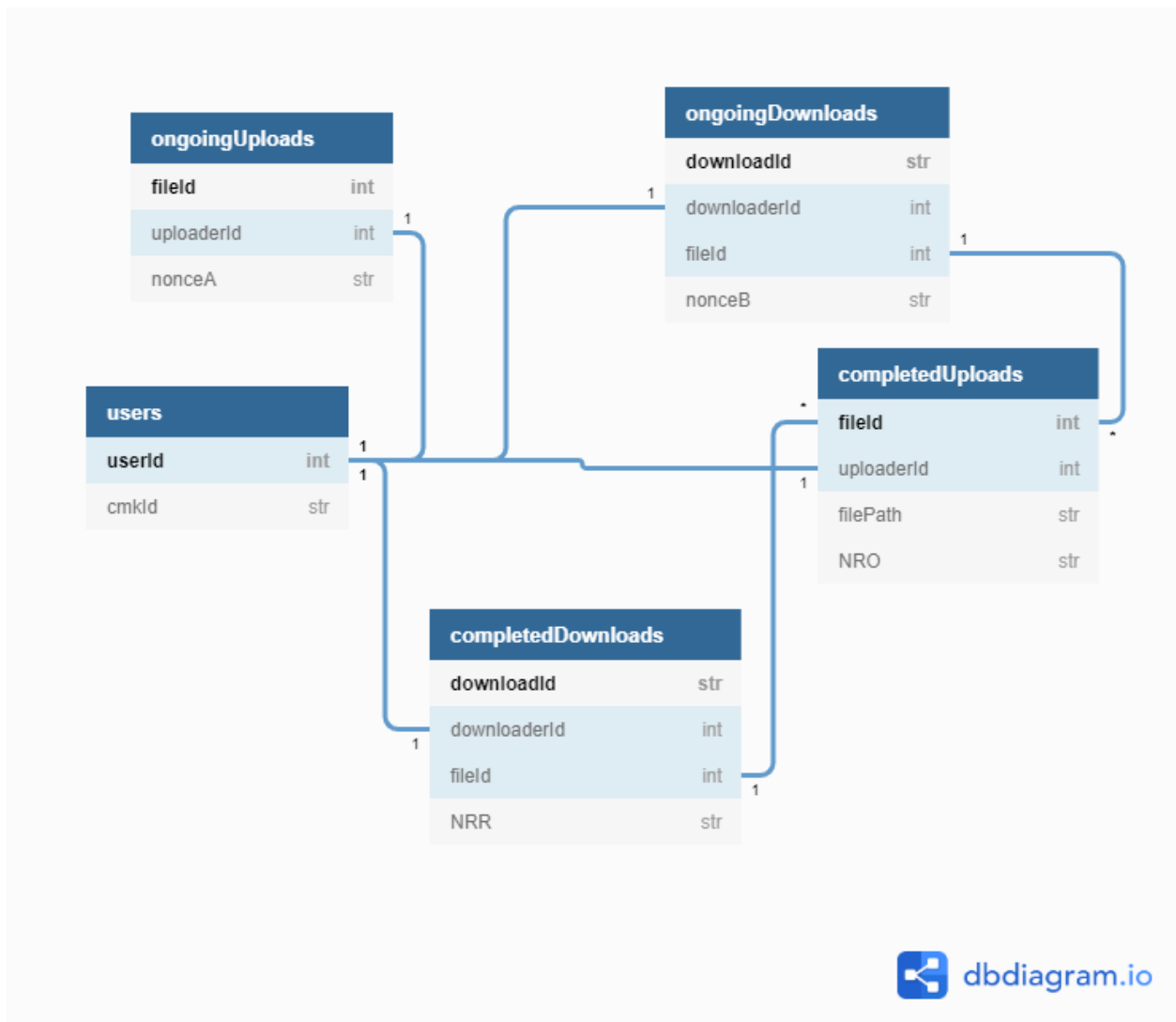


Figure 4: TTP/TDS database diagram

5.4 Communication with TTP

Amazon Simple Queue Service (SQS) has been used as the communication channel between client and TTP. The primary advantage is that messages will not be lost when TTP is inactive/unreachable. In particular, we use 3 SQS. One for TTP, one for uploaders and one for downloaders.

We use this generic *json* structure for every transmission throughout the protocol:

```

1 {
2     "protocolStep" : str
3     "userId"       : int # who sends the message. [TTP sends the
                        userId he receives back]
4     "message"      : dictionary
5 }
  
```

Assume A wants to upload a file and B wants to download it. In this scenario, the message for each *protocolStep* will have a unique form that is described below:

protocolStep : message json structure

- 1 $[A \rightarrow TDS]$: {"fileId": *fileId*}
- 2 $[TDS \rightarrow A]$: {"cmkId": *cmkId_A*, "nonce": *nonce_A*, "fileId": *fileId*}
- 3 $[A \rightarrow TDS]$: {"filePath" : *filePathOnS3*, "fileId" :*fileId*, "nonce": *nonce_A*,
"signature": *Sig_A(H(M))*}
- 4 $[TDS \rightarrow A]$: {"fileId": *fileId*, "verified": *True/False*}
- 5 $[B \rightarrow TDS]$: {"fileId": *fileId*}
- 6 $[TDS \rightarrow B]$: {"cmkId": *cmkId_B*, "fileId": *fileId*, "nonce": *nonce_B*, "signature": *Sig_A(H(M))*}
- 7 $[B \rightarrow TDS]$: {"fileId": *fileId*, "nonce": *nonce_B*, "Signature": *Sig_B(Sig_A(H(M)))*}
- 8 $[TDS \rightarrow A]$: {"fileId": *fileId*, "downloaderId": *downloaderId*, "NRR": *Sig_B(Sig_A(H(M)))*}
- 9 $[TDS \rightarrow B]$: {"NRO": (*Sig_A(H(M))*), "filePath":*filePathOnS3*}

5.5 Abort extension design

An uploader is able to request an abort by sending the following message to TTP:

$a [A \rightarrow TDS]$: {"userId": *userId*, "fileId": *fileId*}

According to these 2 scenarios, TTP responds correspondingly:

1. A aborts before the file is successfully uploaded. TTP deletes any rows from *ongoingUploads* table.

$a [TDS \rightarrow A]$: {"userId": *userId*, "fileId": *fileId*, "abort":1 }

2. A has successfully uploaded a file. TTP deletes any ongoing downloads for this file. Also, these two new sub scenarios are generated:

- (i) The requested file has not been acquired by anyone so it can be safely deleted from the storage. TTP deletes the corresponding row from the *completedUploads* table. The client deletes the file from the database, as well.

$a [TDS \rightarrow A]$: {"userId": *userId*, "fileId": *fileId*, "abort":2 }

- (ii) The requested file has been acquired by at least one user. In this case, TTP flags that file (*abort* = 1) and practically ignores any current and future download requests. However, the file is not deleted from the storage and it is kept for future reference for the completed exchange(s). The client hides the file from the available download files.

$a [TDS \rightarrow A]$: {"userId": *userId*, "fileId": *fileId*, "abort":3 }

5.6 Authorisation of actions extension design

Authorisation of actions was implemented through Django framework, by introducing user authentication in our Client Side. Only authorised users can perform any of the protocol steps.

5.7 Secure communication extension design

Our communication is achieved exclusively by Amazon's queues. Therefore, creating queues with SSE encryption was the way to go. Thus, we avoid scenarios of message tampering/modification and in case of communication relay by an adversary, a message would be encrypted, therefore adversary would not get any valuable information.

6 System Implementation

Since we had a clear plan, we tried to follow the design principle of the separation of concerns(*SoC*). In that way, we tried to split the project into as many subtasks as possible so each member could progress independently from the rest. That allowed us for easier unit-testing and overall faster progress. The main points of our implementation are outlined with the corresponding code snippets on the following subsections. Comments were removed from the code snippets for the shake of space economy.

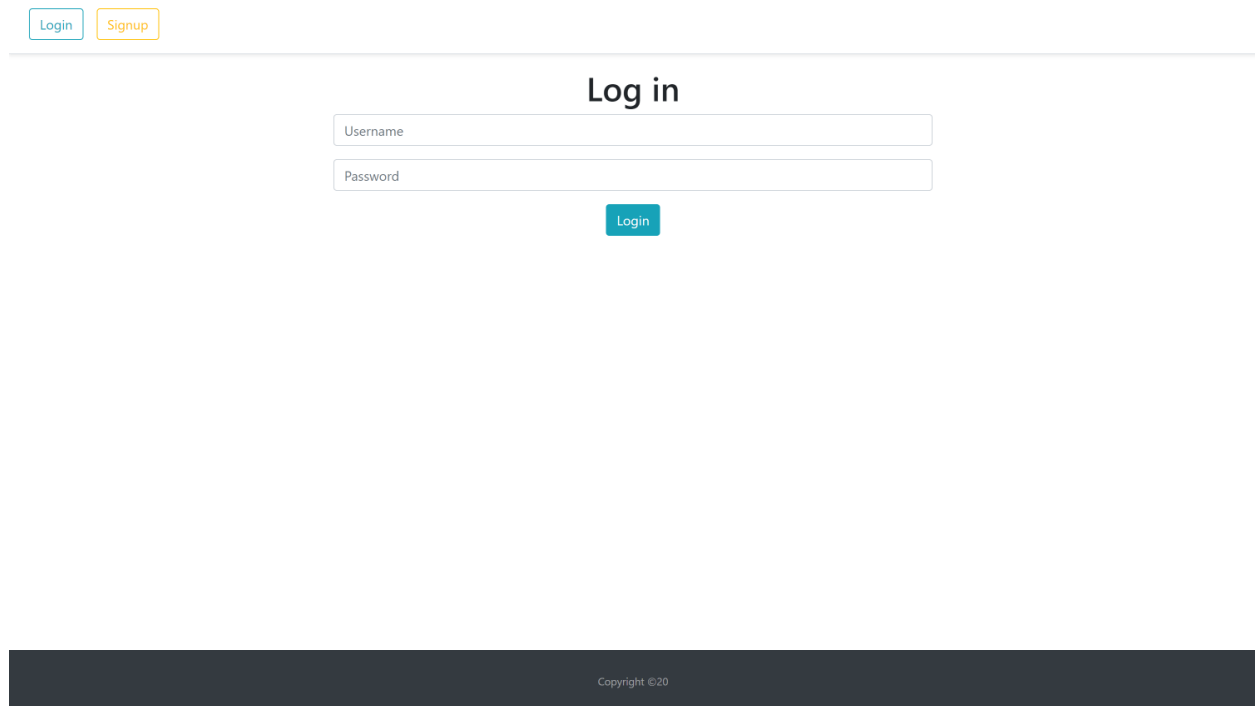
6.1 Client Side

The client system is a full-stack solution. We use a set of technologies such as HTML, JavaScript, and CSS for the front-end part and the Django framework combined with MySQL database on RDS for the backend. Furthermore, there is a separate python script that does the polling for the uploader's message queue. It connects to the same database as Django in order to make the necessary changes per protocol step.

We create and design the table schema including the table relationships through model.py file in Django framework as follows:

```
1 class UploadFile(models.Model):
2     uploaderID = models.ForeignKey(User, verbose_name=' userID ', on_delete=models
        .CASCADE, blank=True, null=True)
3     status = models.IntegerField(verbose_name='statusCode')
4     fileName = models.CharField(max_length=200, verbose_name='fileName')
5     filePath = models.CharField(max_length=500, verbose_name='filePath', blank=
        True, null=True)
```

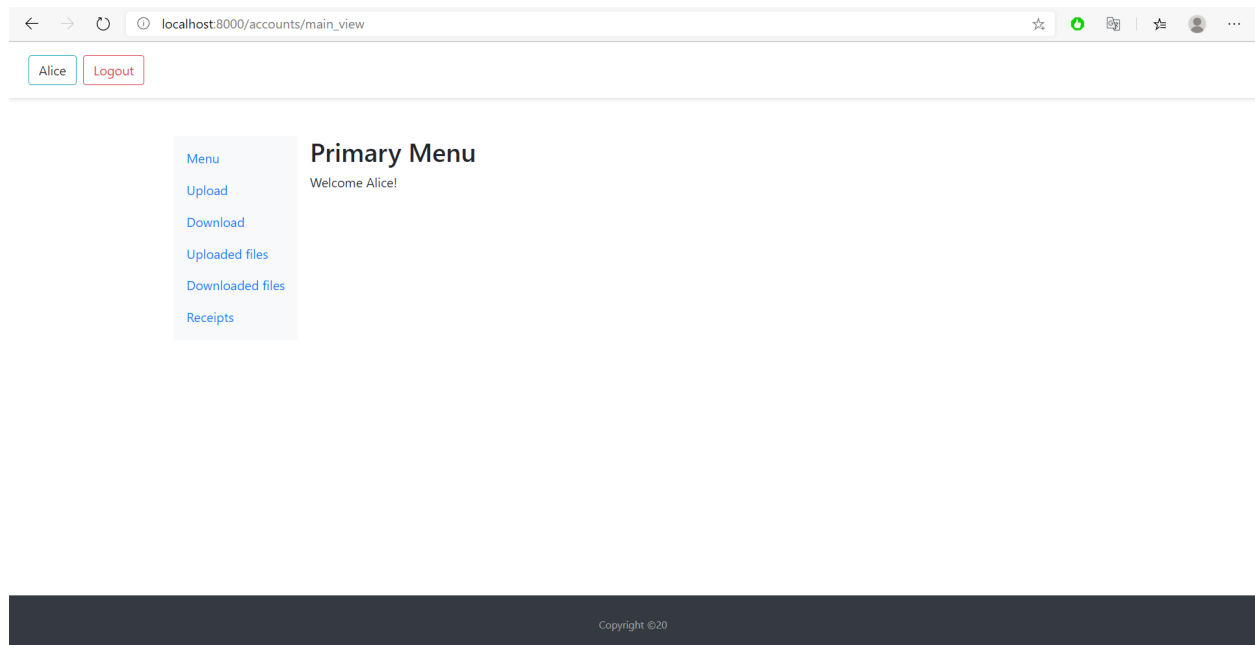
Firstly, users need to register with username and password via the registration page. After successful registration, their data is stored in the *USER* table in the database. Users who already have an account can log in through the login page. While logging in, we query the credentials in the *USER* table in the database to determine whether the account password is correct.



The log in page features a header with 'Login' and 'Signup' buttons. The main content area is titled 'Log in' and contains two input fields: 'Username' and 'Password'. Below these fields is a 'Login' button. The footer consists of a dark grey bar with the text 'Copyright ©20'.

Figure 5: Log-in page

After the user logs in successfully, he is redirected to the main menu. The menu contains a vertical navigation menu with the following options: Upload, Download, Uploaded files, Downloaded files, and Receipts.



The main menu is displayed in a web browser window with the address bar showing 'localhost:8000/accounts/main_view'. The header includes 'Alice' and 'Logout' buttons. The main content area is titled 'Primary Menu' and includes a 'Welcome Alice!' message. A vertical navigation menu on the left lists the following options: Menu, Upload, Download, Uploaded files, Downloaded files, and Receipts. The footer is a dark grey bar with the text 'Copyright ©20'.

Figure 6: Main menu

On the Upload page, shown below, the user can select a file to upload. By clicking the

upload button, the client will send an upload request to TTP and store a record in the database as well. The actual code snippet is attached after the Upload figure.

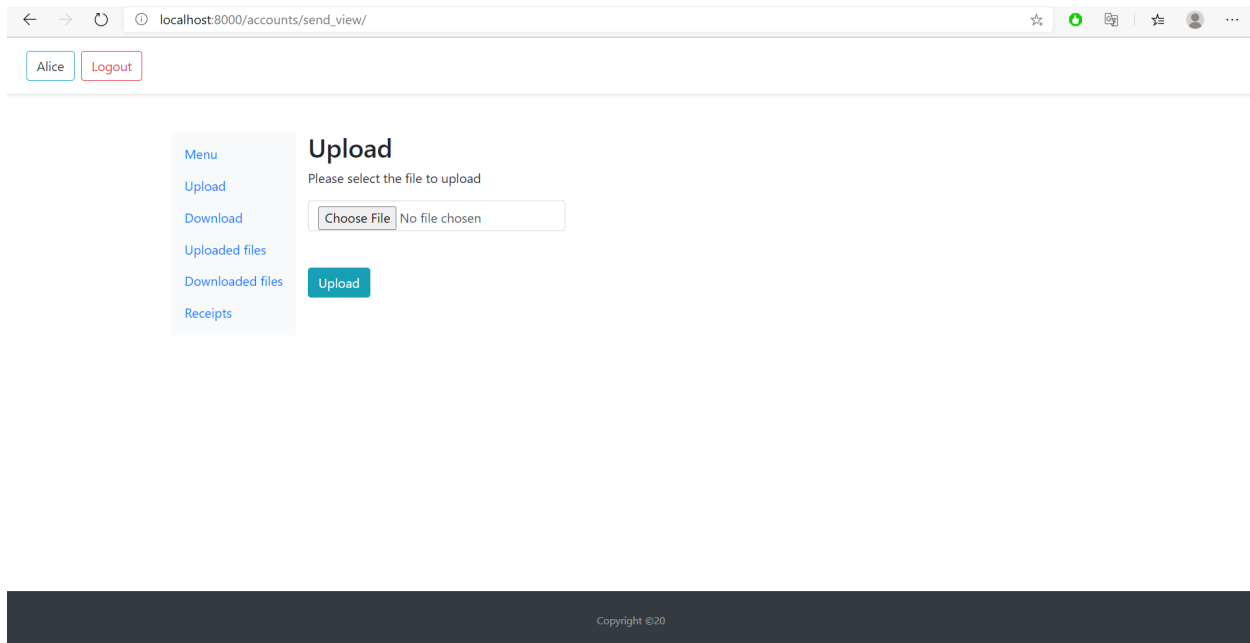


Figure 7: Upload page

```

1 def send(request):
2     path = os.path.abspath(os.path.dirname(os.getcwd())) + '/client/uploads/' +
        files.name
3     path = path.replace('\\', '/')
4     user = User.objects.filter(id=userId)
5     uploadFile = UploadFile(userID=user[0], state=1, filename=files.name, filePath
        =path)
6     uploadFile.save()
7     sqs = boto3.resource('sqs')
8     queue = sqs.get_queue_by_name(QueueName='Queue_for_TTP_fifo')
9     response = queue.send_message(MessageBody=msg, MessageGroupId="1",
        MessageDeduplicationId=dt.datetime.now().
        strftime('%j%H%M%S') + "_" + str(userID))

```

Then the monitor for A script runs. There is an infinite loop in this script so it will be constantly polling for messages (as an extension polling rate can be dynamic, based on demand, e.g. number of active users etc.). We use a method to retrieve the first message from the queue and delete it after receiving:

```

1 def receive_first_msg():
2     sqs = boto3.resource('sqs')
3     queue = sqs.get_queue_by_name(QueueName='Queue_for_A_fifo')
4     for message in queue.receive_messages(MessageAttributeNames=['Author']):
5         dic = json.loads(message.body)
6         if dic is not None:
7             message.delete()

```

```

8         return dic
9     else:
10        return None

```

Then the script will perform the corresponding operations according to TTP's response. If the received TTP's message includes,

```

1 protocolStep: "v"
2 verified: "True"

```

the script will update the file status in database to 1, which indicates that the specific file is uploaded successfully.

Furthermore, we also use *KMS* to sign the file's digest:

```

1 def sign_by_KMS(keyId, hash_code):
2     kms_client = boto3.client('kms')
3     sign_response = kms_client.sign(
4         KeyId=keyId,
5         Message=hash_code,
6         MessageType='DIGEST',
7         SigningAlgorithm='ECDSA_SHA_256'
8     )
9     sig = sign_response["Signature"]
10    return sig

```

However, the signature generated by KMS is a type of *bytes* object which is not *JSON* serializable. For this reason, we convert it to hex format so we can send it via the SQS. For signature verification, we convert it back to bytes again. The corresponding code snippet for the conversions:

```

1 sig_hex = sig.hex() # convert to hex
2 sig_bytes = bytes.fromhex(sig_hex) # convert to bytes

```

It should be noted that we used the *ECC* (Elliptic-curve cryptography) over Rivest–Shamir–Adleman (*RSA*) encryption algorithm. This is done because we needed only to sign and verify documents and not encrypt them. ECC cannot be directly used for encryption but it is applicable for digital signatures. Lastly, its advantage over RSA is that it generates a smaller size key with the same cryptographic strength.

When the user enters the Download page, all downloadable files in the *accounts_uploadfile* table will be queried and displayed.

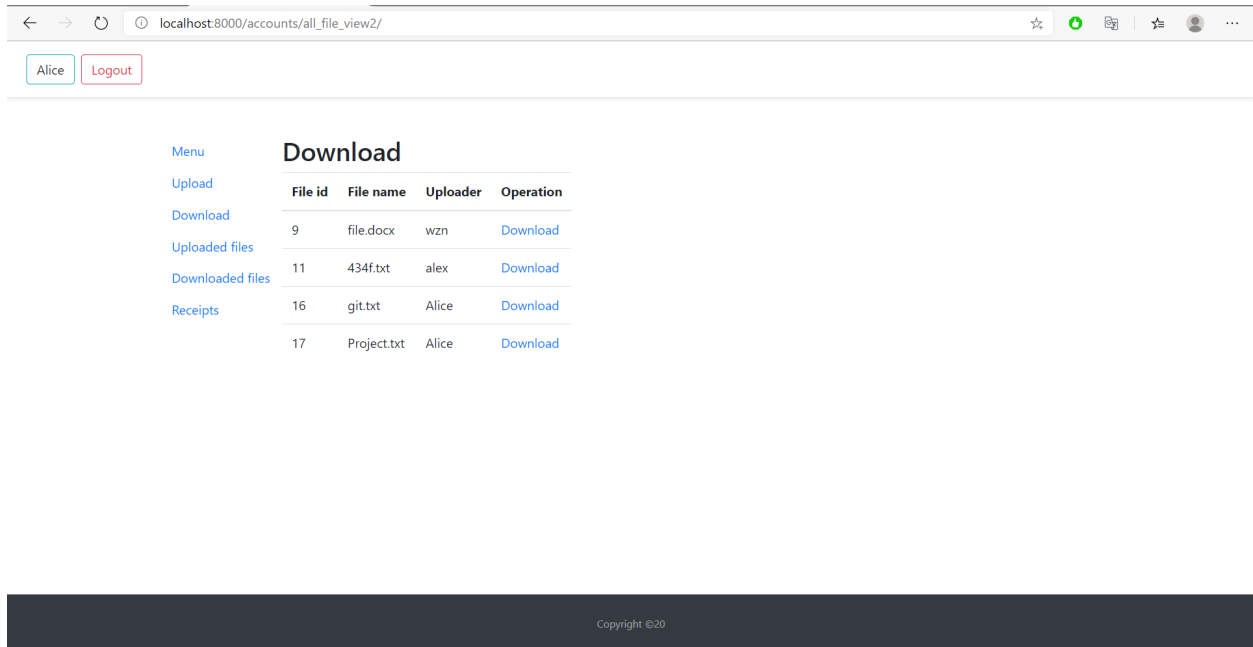


Figure 8: Download page

Once the user clicks the download button, a request will be sent to TTP and then the client will use Ajax to continuously invoke a method to obtain first message from SQS until the final step is acquired:

```

1 function clickMe() {
2     $.ajax({
3         url: "{% url 'accounts:download' %}",
4         type: "POST",
5         dataType: 'json',
6         data: {"fileId": fileId, "userId": userId},
7         timeout: 5000,
8         success: function (data) {
9             if (data.msg==2){
10                 $("#abc").append("<br>successful");
11                 clearInterval(myinVar)
12             }
13         }
14     })
15 }
```

6.2 TTP Side

6.2.1 Database - MySQL on RDS

For the communication with the database we created our own interface for better testing and development purposes. Abstraction was our goal that is why we separated the database functionality from the rest *TTP's* main script. This class, upon instantia-

tion, connects to the database on RDS and upon deletion it rollbacks any changes and disconnects.

```
1 class MySQL_DB(object):
2     def __init__(self):
3         self.conn = mysql.connector.connect(host="OUR_ENDPOINT_ON_AWS", port
4             =3306,database="THE_DATABASE_NAME",user="admin",password="
5             OURPASSWORD")
6         self.cur = self.conn.cursor()
7     def __del__(self):
8         self.cur.rollback()
9         self.cur.close()
10        self.conn.close()
```

Another method was created for the automatic reset of the database; the deletion of all the tables and their recreation:

```
1 def __create_tables__(self):
2     tables = (
3         """CREATE TABLE users (
4             userId INTEGER PRIMARY KEY,
5             cmkId VARCHAR(255) NOT NULL) """,
6         """CREATE TABLE ongoingUploads (
7             fileId INTEGER PRIMARY KEY,
8             uploaderId INTEGER NOT NULL,
9             nonce VARCHAR(255) NOT NULL,
10             FOREIGN KEY(uploaderId)
11                 REFERENCES users(userId) """,
12         """CREATE TABLE completedUploads (
13             fileId INTEGER PRIMARY KEY,
14             uploaderId INTEGER NOT NULL,
15             filepath VARCHAR(255) NOT NULL,
16             nro VARCHAR(255) NOT NULL,
17             abort INTEGER NOT NULL,
18             FOREIGN KEY(uploaderId)
19                 REFERENCES users(userId)
20             ON UPDATE CASCADE ON DELETE CASCADE) """,
21         """CREATE TABLE ongoingDownloads (
22             downloadId VARCHAR(255) PRIMARY KEY,
23             downloaderId INTEGER NOT NULL,
24             fileId INTEGER NOT NULL,
25             nonce VARCHAR(255) NOT NULL,
26             FOREIGN KEY(downloaderId)
27                 REFERENCES users(userId) """,
28         """CREATE TABLE completedDownloads (
29             downloadId VARCHAR(255) PRIMARY KEY,
30             downloaderId INTEGER NOT NULL,
31             fileId INTEGER NOT NULL,
32             nrr VARCHAR(255) NOT NULL,
33             FOREIGN KEY(downloaderId)
34                 REFERENCES users(userId)
35             ON UPDATE CASCADE ON DELETE CASCADE,
36             FOREIGN KEY(fileId)
```



```

37                                     REFERENCES completedUploads(fileId)
38                                     ON UPDATE CASCADE ON DELETE CASCADE) """)
39     for table in tables:
40         self.cur.execute(table)
41         self.conn.commit()
42
43     def __reset__(self):
44         self.cur.execute("SELECT table_name FROM information_schema.tables WHERE
45                             table_schema = %s", ("mysqldata",))
46         rows = self.cur.fetchall()
47         print(rows)
48         for row in rows:
49             print("drop table " + row[0])
50             self.cur.execute("DROP TABLE " + row[0] + " cascade")
51 self.conn.commit()

```

The main functionality of the interface can be seen on the following code segments. The first segment serves the insertion purpose. For an easier interaction with this function, there is a dictionary that holds all the SQL insert queries.

```

1  INSERT_DIC = {
2  "insertUsers"      : "INSERT INTO users(userId,cmkId) VALUES(%s,%s)",
3  "insertOngUp"      : "INSERT INTO ongoingUploads( fileId ,uploaderId,nonce) VALUES(%s,%s,%s
4  )",
5  "insertCompUp"     : "INSERT INTO completedUploads( fileId ,uploaderId ,filepath ,nro,abort)
6  VALUES(%s,%s,%s,%s,%s)",
7  "insertOngDown"    : "INSERT INTO ongoingDownloads(downloadId,downloaderId, fileId ,nonce)
8  VALUES(%s,%s,%s,%s)",
9  "insertCompDown": "INSERT INTO completedDownloads(downloadId,downloaderId, fileId ,nrr)
10 VALUES(%s,%s,%s,%s)"
11 }
12
13 def insertInto(self, query, params):
14     try:
15         response = self.cur.execute(INSERT_DIC[query], params)
16         self.conn.commit()
17         return response
18     except mysql.connector.Error as error:
19         print(error)
20         return -1

```

Here are the row "transfer" from *ongoingUploads* to *completedUploads*, when the upload is verified and valid. The same is done for the downloads which also returns the *filepath* in order for TTP to send it to the downloader.

```

1  def completeUpload(self, fileId ,filepath ,nro):
2      try:
3          self.cur.execute("SELECT u.uploaderId FROM ongoingUploads as u WHERE
4                              fileId = %s", (fileId ,))
5          record = self.cur.fetchall()
6          if (len(record) == 0):
7              return
8          self.cur.execute(INSERT_DIC["insertCompUp"], (fileId ,record
9              [0][0],filepath ,nro,0))

```

```

8         self.cur.execute("DELETE FROM ongoingUploads as u WHERE fileId
                           = %s", (fileId,))
9         self.conn.commit()
10    except mysql.connector.Error as error:
11        print(error)
12        return -1
13    def completeDownload(self, downloaderId, fileId, nrr):
14        try:
15            downloadId = str(downloaderId) + "_" + str(fileId)
16            self.cur.execute(INSERT_DIC["insertCompDown"], (downloadId,
17                                                              downloaderId, fileId, nrr))
18            self.cur.execute("DELETE FROM ongoingDownloads as u WHERE downloadId =
                              %s", (downloadId,))
19            self.cur.execute("SELECT u.filepath FROM completedUploads as u WHERE
                              fileId = %s", (fileId,))
20            record = self.cur.fetchall()
21            self.conn.commit()
22            return record[0][0]
23        except mysql.connector.Error as error:
24            print(error)
25            return -1

```

Lastly, the abort code logic that was explained in the previous section:

```

1    def abort(self, fileId):
2        try:
3            if (self.fileExists(fileId)):
4                self.cur.execute("DELETE FROM ongoingDownloads as u WHERE
                                  fileId = %s", (fileId,))
5                if (self.isFileDownloaded(fileId)):
6                    self.cur.execute("UPDATE completedUploads SET abort=1 WHERE
                                      fileId = %s", (fileId,))
7                    self.conn.commit()
8                    return 3
9            else:
10               self.cur.execute("SELECT u.filepath FROM completedUploads as u
                                 WHERE fileId = %s", (fileId,))
11               record = self.cur.fetchall()
12               self.cur.execute("DELETE FROM completedUploads as u WHERE
                                 fileId = %s", (fileId,))
13               self.conn.commit()
14               return record[0][0]
15            else:
16               resp = self.cur.execute("DELETE FROM ongoingUploads as u WHERE
                                       fileId = %s", (fileId,))
17               self.conn.commit()
18               return 1
19        except mysql.connector.Error as error:
20            print(error)
21            return -1

```

6.2.2 TTP structure

Server script implements the behaviour of TTP. To begin with, once we instantiate the script, we perform constant polling of the TTP queue. The rate of polling is derived from calling the "traffic" method. Currently, this method returns a static rate, due to the short-time interval of our project, but as an extension, we decided that we could change it to dynamic, based on the number of active users on the Django framework.

Once the queue receives a message (note that messages follow a unified format) it checks for the protocol step. If the protocol step is 1, it calls *perform_step2* method. If the protocol step is 3, it calls *perform_document_upload* method. If the protocol step is 4, it calls *perform_step5* method. If the protocol step is 6, it calls *perform_step7and8* method. Finally, if the protocol step received is "a", it calls *perform_abort* method. All of the aforementioned methods parse as arguments the message (as a python dictionary). "Perform" methods will be thoroughly explained, after discussing upon the helper methods.

Helper methods are *key_for_user*, *signature_verification*, *raw_signature_verification*, *create_file_digest*, *download_doc_return_hash*, *delete_doc* and *traffic*. The latter method was already explained before.

Key_for_user method: This method takes as a parameter a user ID. It firstly connects to the KMS client. (It should be noted at this point, that whenever we connect to any Amazon web service, we handle exceptions using try-except statements.) Then it checks if the user ID already has a key pair, if not, it creates a new one and stores it to *Users* table. *Key_for_user* method returns a *cmk_id*:

```
1 cmk_id = self.db.getKeyId(userId)
2 if (cmk_id == None):
3     response_ck = kms_client.create_key(
4         Description=userId,
5         KeyUsage='SIGN_VERIFY',
6         CustomerMasterKeySpec='ECC_NIST_P256', # only sign-verify. Elliptic curve
           cryptography
7         Origin='AWS_KMS',
8         BypassPolicyLockoutSafetyCheck=False, #default
9         Tags=[{
10             'TagKey':  userId,
11             'TagValue': ''}]
12     )
13 cmk_id = response_ck['KeyMetadata']['KeyId']
14 self.db.insertInto("insertUsers", (userId, cmk_id))
15 kms_client.create_alias(
16     AliasName='alias/' +userId,
17     TargetKeyId=cmk_id
18 )
19 return cmk_id
```

Signature_verification method: This method takes as parameters a user ID, a message and a signature. Firstly, it connects to KMS client. Then, using our database,

we derive the key ID of the user ID that was parsed in parameters. After that, we perform verification using message and signature and return a boolean depicting if the verification is successful or not.

```

1 cmk_id = self.db.getKeyId(userId)
2 if cmk_id is not None:
3     response_verify = kms_client.verify(
4         KeyId=cmk_id,
5         Message=message,
6         MessageType='DIGEST',
7         Signature=signature,
8         SigningAlgorithm='ECDSA_SHA_256',
9     )
10    return response_verify[ 'SignatureValid' ]
11 else:
12    return False

```

Raw_signature_verification method: This method performs exactly the same operation as the aforementioned method, with the only difference that the message is not of DIGEST type. This implementation was done because we do not only need to verify hash digests but signatures parsed as messages as well (i.e $Sig_A(Sig_B)$).

Create_file_digest method: This method returns a hash digest of a file. We use it to re-create a hash of the document uploaded to the store.

```

1 def create_file_digest(self, file_path):
2     BUF_SIZE = 2**16
3     with open(file_path, "rb") as f:
4         file_hash = hashlib.sha256()
5         # use buffer for faster digest calculation
6         buffer = f.read(BUF_SIZE)
7         while buffer:
8             file_hash.update(buffer)
9             buffer = f.read(BUF_SIZE)
10    return file_hash.digest()

```

Download_doc_return_hash method: This method connects with S3 and downloads the specified document, parsed in parameters, it then calls the *create_file_digest* and returns the hash of the document created.

Delete_doc method: That is the last helper method. It takes as a parameter the file path of a document, accesses S3, and deletes the document from the bucket calling AWS's *delete_object* method.

Moving on to our perform methods, we start with *perform_step2*. It should be noted that perform methods return a boolean. If the boolean is true, then the message that they acted upon is deleted, else not. This method takes place after receiving a message with protocol step = 1. The first step is to get the key ID of the user that sent protocol step 1, generate a nonce and send back to the user that sent the aforementioned message; the key ID (cmkId), the nonce, the file ID by dumping them

into a json message. After this step, it inserts to insert ongoing uploads table the appropriate data. Decoupling dictionary to local variables:

```
1 def perform_step2(self, dic):
2     print('Starting step 2..')
3     curr_userId = dic["userId"]
4     dic2 = json.loads(dic["message"])
5     fileId = dic2["fileId"]
```

Getting key ID:

```
1 cmkId = self.key_for_user(curr_userId)
```

Generating nonce:

```
1 nonce = os.urandom(16)
```

Creating an appropriate dictionary:

```
1 step2msgdic = {
2     "cmkId": cmkId,
3     "nonce": nonce,
4     "fileId": fileId
5 }
6 step2dic = {
7     "protocolStep": 2,
8     "userId": curr_userId,
9     "message": step2msgdic
10 }
11 json_msg = json.dumps(step2dic)
```

Sending a message to A's queue:

```
1 response_a = self.queue_for_a.send_message(MessageBody=json_msg, MessageGroupId="1",
2 MessageDeduplicationId=dt.datetime.now().strftime('%j%H%M%S'))
3 print(response_a.get('MessageId'))
4 print(response_a.get('MD5OfMessageBody'))
5 print('Finished step 2..')
6 return True
```

Perform_document_upload method takes place after receiving a message with protocol step = 3. Message inputs, except protocol step and user ID are; file path, file ID, nonce and signature. Firstly we encode the signature we received to bytes.

```
1 signature = dic2["signature"]
2 signature_byte = bytes.fromhex(signature)
```

Then, we get the upload nonce stored to our database and we re create the hash of the specified document using the file path on S3:

```
1 OngUpNonce = self.db.getUploadNonce(fileId)
2 hDoc = self.download_doc_return_hash(filePath)
```

And we perform verification using the current user ID we received in the message, the hash doc that we created as the content that was signed and the signature in bytes as the signature created. If the verification is correct and the nonce we received is the same as the one stored in the *ongoingUploads* table, we then update our database:

```

1 verification_bool = self.signature_verification(curr_userId,hDoc,signature_byte)
2 if (hDoc == None or verification_bool == None or OngUpNonce == None):
3     return False
4 if (verification_bool==True and OngUpNonce==nonce):
5     self.db.completeUpload( fileId , signature , filePath)
6     booleanForCurrentFile = True
7 else:
8     booleanForCurrentFile = False
9     print("Did not pass verification. Doc Discarded")

```

We send back to A a protocol step "v" with a boolean "verified" determining the state of the file.

Perform_step5 method: This method is called after the downloader (or B) requests a document to download. The document is requested using the file ID. Firstly, we check our database, if the file with the specified file ID exists:

```

1 fileExists = self.db.fileExists(fileId)

```

If the file exists, we generate a key ID for the current user (if it already exists we simply sent it), we generate a nonce and we get the signature of the uploader of this specified file using our database interface. We update our ongoing downloads table and send the message to B's queue (or downloader's queue) as follows:

```

1 if fileExists == True:
2     cmkId = self.key_for_user(curr_userId)
3     if cmkId == None:
4         return False
5     nonce = os.urandom(16)
6     signature = self.db.getNro(fileId)
7     downloadId = curr_userId + "_" + fileId
8     step5msgdic = {
9         "cmkId": cmkId,
10        "nonce": nonce,
11        "signature": signature
12    }
13    step5dic = {
14        "protocolStep": 5,
15        "userId": curr_userId,
16        "message": step5msgdic
17    }
18    self.db.insertInto("insertOngDown", (downloadId, curr_userId, fileId, nonce))
19    json_msg = json.dumps(step5dic)
20    response_b = self.queue_for_b.send_message(MessageBody=json_msg,
21        MessageGroupId="1",
22        MessageDeduplicationId=dt.datetime.now().strftime('%j%H%M%S'))
23    return True
24 else:
25     print('File does not exist.')
26     return True

```

If the file does not exist, we return true and delete the message.

Perform_step7and8 method: Our last the method is called *perform_step7and8* and performs the 2 last steps transactionally. This method is called upon receiving a message with protocol step = 6. Besides the protocol step and user ID we receive: the file ID, the nonce and the signature. The signature is actually $Sig_B(Sig_A(M))$ for the specified document(M) that has the aforementioned file ID. Nonce should be the same that was generated when executing step 5 for the current transmission. We first check if the file exists, then we get the previously stored download nonce from our database and the NRO (or Sig_A) of the document with the specified file ID:

```
1 OngDownNonce = self.db.getDownloadNonce(fileId , curr_userId)
2 NRO = self.db.getNro(fileId)
```

At this point we perform verification, using *raw_signature_verification* method for the current user, with the signature that we received. The message signed should be the NRO that we got from our database. If the verification is correct, we send to A (the uploader) the NRR, which is the signature we received and the NRO plus the document (as file path on S3) to B (the downloader). We are sending those messages transactionally as a step to guarantee fairness.

Perform_abort method: The last method is called *perform_abort* and it is called after receiving protocol step = 'a'. We simply receive a file ID and perform the abort method in our database. In return, we send to the uploader back a protocol step = 'a' and an abort integer that will decide the state of the file. In successful abort, we also call the delete document method. Database call:

```
1 abort = self.db.abort(fileId)
2 if (isinstance(abort, str)):
3     self.delete_doc(abort)
4     abort = 2
```

7 System evaluation

7.1 Amazon Web Services Evaluation

We are confident enough that we did a great selection among the services provided by AWS for our system. The objective was to design a fast, scalable and extensible system. We evaluate our choices for AWS on the subsections that follow.

7.1.1 Simple Queue Service (SQS)

A key factor for our project is the communication part between Clients and the Trusted Third-Party. In our first meeting, we decided to put a strong effort into exploring Amazon Web Services and in what those services have to offer. Our first approach was to communicate via a REST API that would handle all HTTP/HTTPS requests. That could be achieved by using specific frameworks such as Django and Flash or create an API endpoint through AWS and send or receive requests with python. However, once

we found out the capabilities of SQS and the simplicity in implementing them in our source code, we settled on using it. Every member of the team knew how queues, as a structure, operate, thus we could manage to send and retrieve a message in just a few lines of code, that we derived by studying the Amazon documentation and performing practical examples/test cases. Whereas in a REST API environment it would have been a much more difficult and time-consuming implementation scenario. For both Client and TTP we used long polling as means of message receival. Compared to building a REST API system, Amazon's queues proved to be far simpler in terms of coding and more solid in terms of security, due to SSE encryption. Finally, SQS is not free, therefore there is the drawback of bills, which can be relatively high if the demand for the service is high. Contrary, if the demand is low, costs are insignificant. Last but not least, SQS provides message timestamps which could be used to implement TSA. However, we did not activate that option since it was out of project scope and we had no time to experiment enough with that feature.

7.1.2 Simple Storage Service (S3)

Another fundamental part of our project was the actual document storage. Our main concern was where to store documents and how we would transmit them across entities. Then, the security of those files was up next for consideration. We could easily implement a local TTP save of documents, but we crossed it out as it is not considered secure at all. Saving documents in a database would be a very rough scenario, as in some databases there is no such implementation and in some others there is a per-document limitation which could be extending with certain frameworks. Such a decision was instantly dismissed as we wanted to keep imported API's to a minimum. The solution to the problem came with Amazon's S3. With such service, we could securely upload and download any file, no matter its size and no matter its type (we could store any type of virtual data). Furthermore, S3's Bucket and Object access policy came in handy since we restricted access outside of the Client and the TPP. S3's drawback is cost again, only in a high traffic/high demand scenario. Other file storage and transfer options were Amazon Elastic Block Store or Amazon Elastic File System. All of the file storage and transfer services would fit perfectly for our scenario, however, S3 offers an automated backup and restoration, as well as relatively better (and robust) performance than the rest of the services.

7.1.3 Key Management Service (KMS)

Fairness is guaranteed through the delivery of evidence via queues. Evidence in our scenario is considered the non-repudiation of origin (NRO); plus the specified document delivery; and the non-repudiation of receipt (NRR). In order to create such evidence for the given protocol, we had to implement a public-key cryptography scenario. More specifically, entity X would sign a message with its private key and another entity (TTP in our protocol) would verify X's signature using X's public (known) key. Imple-

mentation could be achieved either through stock python libraries (such as pycrypto) or through the use of AWS. Pycrypto and rest stock libraries proved inefficient as we would have to manually distribute private keys, which is something counter-intuitive in respect to security measures; plus, key creation and safe key storage would be a rough procedure to deal with, in such a short time-span. Thus, we decided on using AWS. The most popular, well-known and most used AWS libraries for public-key cryptography and signature verification were KMS and PKI. As AWS PKI is mainly focusing on managing certificate authorities, we decided that such service does not fit appropriately in our project development. On the other hand, AWS KMS fits perfectly in our project as it is designed to help us, the developers, to easily create, control and safely store keys that we need to use for our cryptographic operations. Therefore, with KMS we got rid of manual key distribution and storage, every key couple was saved in KMS's storage with an alias for each user of our system.

7.1.4 Relational Database Service (RDS)

Amazon through RDS provides a ready cloud database solution. The setup is easy and simple and all the configurations can be made via a User Interface. Specifically, we opted for the following properties:

1. **Build-in Encryption (AES-256):** Amazon handles automatically the data encryption, the user authentication and lastly the decryption ensuring a secure environment for both client and TTP. It should be noted that the impact on performance was minimal so there was no disadvantage in using it besides not being able to use Read Replicas for faster data read.
2. **Deletion protection:** Database cannot be deleted from a user. This option was activated after the completion of our tests and it was much needed for the TTP since the database is crucial and should not be deleted.
3. **Storage Autoscaling:** In case of increased demand, Amazon recommends specific settings and allows us to scale the database on the fly without any downtime.
4. **Backup.** Database snapshots are taken automatically to reduce the amount of lost information in case of a failure.
5. **Network isolation:** We restricted the access only to certain IP's so no one else could connect to the database.

7.1.5 Elastic Compute Cloud (EC2)

To simulate a real website, we uploaded the Django project on a EC2 server. Likewise with the RDS EC2 provides autoscaling and greater network isolation since you need a specific private key(.pem file) to connect to the server. There were other options,

such as AWS Elastic Beanstalk; but we kept using EC2 due to its easier management, auto-scaling and relatively quicker deployment in comparison to AWS EB.

Concluding, each Amazon Web Service, offers a safe, easy to learn and easy to use pre-built environment for every aforementioned implementation scenarios. Thus, AWS is letting our team focus more on the actual design, the effectiveness and efficiency of our project instead of investing time and energy in implementing mini-tasks to build sub-functionalities for our protocol. However, there is always a trade-off. As mentioned before, AWS is not free and in high traffic applications, it can be relatively costly.

7.2 Choice Reflection

Throughout the development process we kept evaluating our choices and researching new approaches when we faced with a problem. Firstly, we had understood the project wrong but on our third group meeting in which our monitor was present, he helped us acknowledge our wrong approach and explained us the correct specification of the project. We started implementing the original protocol itself, so A was sending the file to a declared recipient B. Luckily, our design was such that we could easily change it without any problems.

Furthermore, as far as the database is concerned, we tried to use a NoSQL database since it is to use and the performance would be great since we had no complicated queries. We tried to use DynamoDB because of the auto-scaling capabilities and the key-value type. However, such approach was, later on, proved insufficient since we needed valid information and the much needed *ACID* property for our database. The next choice was PostgreSQL. Unfortunately, after implementing everything we realised that PostgreSQL could not store *NULL* bytes thus causing problems with the storage of signatures, nonces and later on with their verification. Finally, we ended up using the MySQL on RDS for both sides. At first, we had a MySQL docker on EC2 but we decided for our ease, and for future scalability, to have both databases on RDS.

Finding the optimal polling rate was a tough decision to make and we had no time to come up with a dynamic solution. On the client-side, the polling is made in two ways. For the uploaders queue, it is done through a python script that runs and checks the *SQS* for any new messages (like the TTP script). On the other hand, the other downloaders' queue is being polled upon the interaction with the frontend through ajax. This way we reduce the amount of overall requests but we, also, acknowledge the fact that it is not as fast and responsive as the constant polling via an infinite loop. Another option was using *Celery* library to asynchronously process both queues but a member of ours failed to accomplish the integration with Django. On the bright side, the expenses of *SQS* were restricted due to less overall requests because we opted for "ajax-polling" design.

7.3 Outcome

We manage to build a fair file-sharing platform that relies greatly on AWS. The advantage of that choice was the easy deploy of the vital services for our system, which significantly reduced the amount of coding. However, the cost of such implementation is high and can grow even higher if there is greater demand for the service. For this reason, the solution we suggest is not optimal for in-house development but it is optimal for scenarios where developer's time is more expensive than the overall AWS cost.

8 Conclusion

In this paper, we implemented a Trusted Document Store platform in which users can exchange files securely. For achieving fairness we based our communication model on a non-repudiation inline protocol of Coffey and Saidha. Furthermore, we took advantage of the ready solution provided by AWS to ensure that the security is unbreachable rendering it sufficient against any replay attacks, tampering or eavesdropping. However, first and foremost our implementation provides strong fairness, hence there is no possible way one of the two participating parties get evidence for a given exchange and the other not; either both of them will have acquired the corresponding evidence or none.

Moreover, this piece of software can be further enhanced by performing a more sophisticated message polling. That could be done by either implementing a machine learning algorithm for better utilization of the message queue or by simply manipulating the queue via AWS Lambda Endpoints. Another possible addition could be the implementation of another external platform for timestamps, the TSA or activating the message timestamp option on SQS. However, these approaches need to be researched first.

References

- [1] "Amazon Web Services (AWS) - Cloud Computing Services", Amazon Web Services, Inc., 2020. [Online]. Available: <https://aws.amazon.com/>. [Accessed: 26- Mar-2020].
- [2] T. Coffey and P. Saidha, "Non-repudiation with mandatory proof of receipt", *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 1, pp. 6–17, 1996
- [3] P. D. Ezhilchelvan, "Algorithms in E-commerce: Fair Exchange Protocols," in *Distributed Algorithms*, Newcastle University, ch. 3 2019
- [4] "Flowchart Maker & Online Diagram Software", Draw.io, 2020. [Online]. Available: <https://www.draw.io/>. [Accessed: 26-Mar-2020].

- [5] "Database Relationship Diagrams Design Tool", Dbdiagram.io, 2020. [Online]. Available: <https://dbdiagram.io/> [Accessed 26-Mar-2020].