CSC8113 Group Project Team 3

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

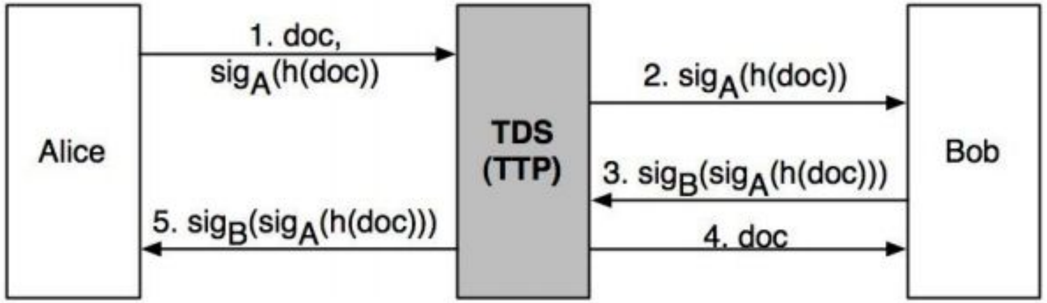
Shiwei,Mo S.Mo2@newcastle.ac.uk

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

Project Scope

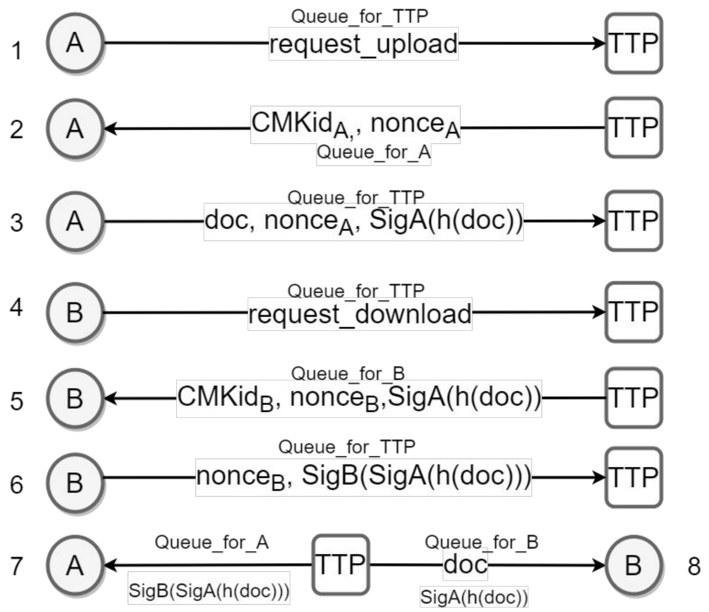
- Design and develop a Trusted Third Party (TTP) fair exchange service (system),
- Using Amazon Web Services (AWS),
- Based on the protocol steps of the depicted protocol below:

- Add a document to the store (Step 1)
- Request an identified object(Step 2 - returning the signature)
- Get a document (Steps 3 and 4)
- Get a receipt associated with the document (Step 5)
- Guarantee Fairness



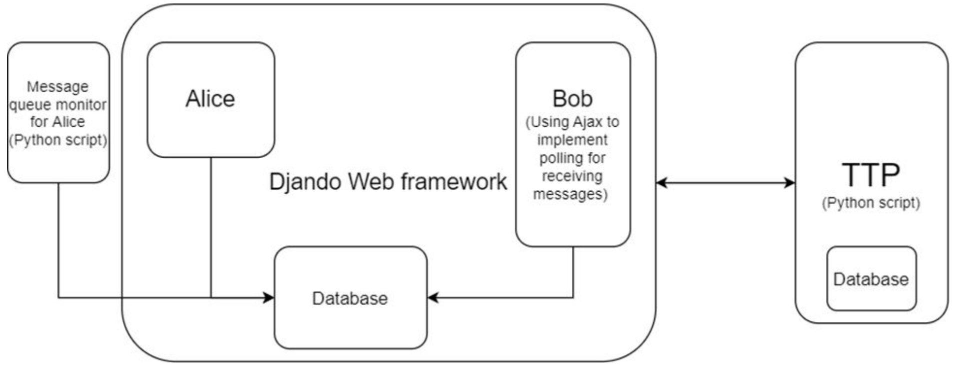
Project's Protocol Implementation

Protocol is split into 8 different steps. Last two steps are performed transactionally to ensure fairness. Each step will be explained thoroughly after the illustration of the design approach.



1 A Queue_for_TTP request_upload TTP
2 A CMKid_A,, nonce_A Queue_for_A TTP
3 A Queue_for_TTP doc, nonce_A, SigA(h(doc)) TTP
4 B Queue_for_TTP request_download TTP
5 B Queue_for_B CMKid_B, nonce_B,SigA(h(doc)) TTP
6 B Queue_for_TTP nonce_B, SigB(SigA(h(doc))) TTP
7 A Queue_for_A SigB(SigA(h(doc))) TTP Queue_for_B doc SigA(h(doc)) B 8

System Structure



Message queue monitor for Alice (Python script)
Alice
Djando Web framework
Bob (Using Ajax to implement polling for receiving messages)
Database
TTP (Python script)
Database

Design: Technologies Used

Amazon Web Services:

- Amazon Simple Queue Service (SQS) for message transmission between Client and TTP(or TDS) .
- Amazon Key Management Service (KMS) for sign and verification of messages on both sides.
- Amazon Simple Storage Service (S3) for document manipulation on both sides.
- Amazon Relational Database Service (RDS) for database manipulation on TTP side.
- Amazon Elastic Cloud Service (EC2) to run the system online.

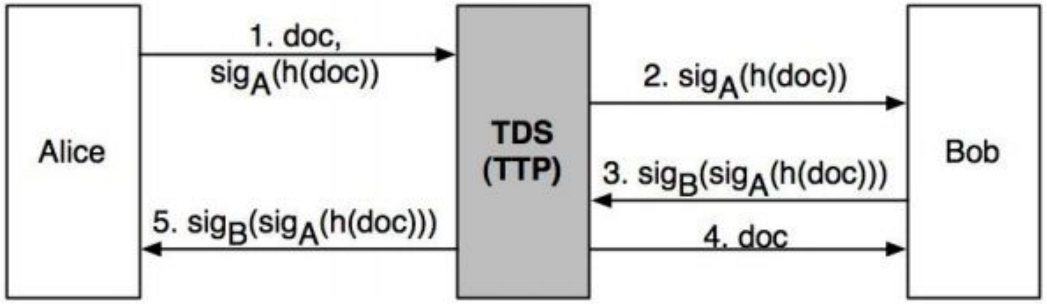
Programming Languages and Framework:

- Python is our main programming language
- Django Web framework is used in our Client side (Sender and Receiver)
- HTML, CSS and javascript for Front-end design and implementation of Client
- Postgresql & mySQL for databases

Design: Message Transmission 1/3

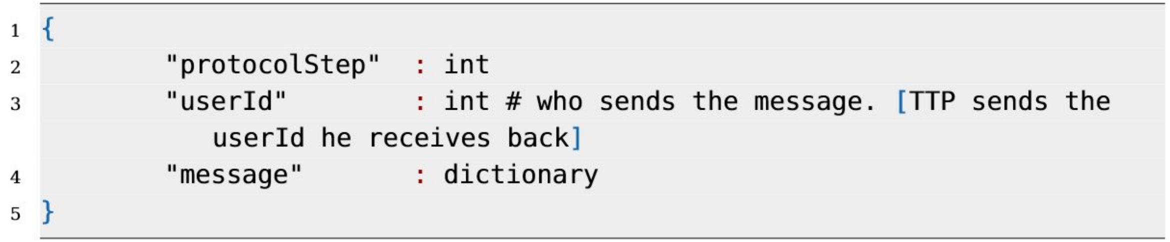
- Decided to use AWS SQS (Simple Queue Services for each part of the design (sender, received and TTP).
- We split the project as Client Side (Alice and Bob) and Server Side (TTP), note that a Client can upload or download a document.
- 3 Queues are implemented for sender, receiver and TTP respectively.
- Each entity listens to its own message queue, and each message is sent to the corresponding queue rather than broadcasted to all queues.
- Each entity is using polling to receive new messages.
- Queue structure is FIFO (First-In-First-Out) and messages are securely transmitted through SSE encryption, thus messages cannot be tampered and if intercepted no information can be gained.

Referencing the depicted graph on the right, an Arrow (which is a message transmission) from Alice or Bob to TTP is sent to TTP Queue. Arrows from TTP to Alice, are messages sent to A Queue (or sender queue). Lastly, Arrows from TTP to Bob are messages sent to B Queue (or receiver queue).



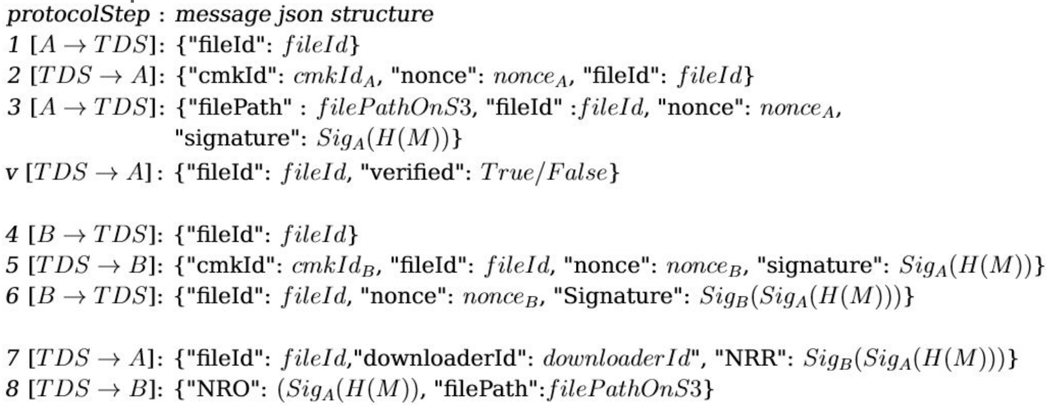
Design: Message Transmission 2/3

For Queue's messages we use a generic *json* structure for every transmission throughout the protocol:



```
{
        "protocolStep"  : int
        "userId"        : int # who sends the message. [TTP sends the
           userId he receives back]
        "message"       : dictionary
}
```

Message dictionary content varies, depending on the protocol step. More specifically, the content of the dictionary per protocol step is:



protocolStep : *message json structure*

1 $[A \to TDS]$: {"fileId": $fileId$}

2 $[TDS \to A]$: {"cmkId": $cmkId_A$, "nonce": $nonce_A$, "fileId": $fileId$}

3 $[A \to TDS]$: {"filePath" : $filePathOnS3$, "fileId" :$fileId$, "nonce": $nonce_A$, "signature": $Sig_A(H(M))$}

v $[TDS \to A]$: {"fileId": $fileId$, "verified": $True/False$}

4 $[B \to TDS]$: {"fileId": $fileId$}

5 $[TDS \to B]$: {"cmkId": $cmkId_B$, "fileId": $fileId$, "nonce": $nonce_B$, "signature": $Sig_A(H(M))$}

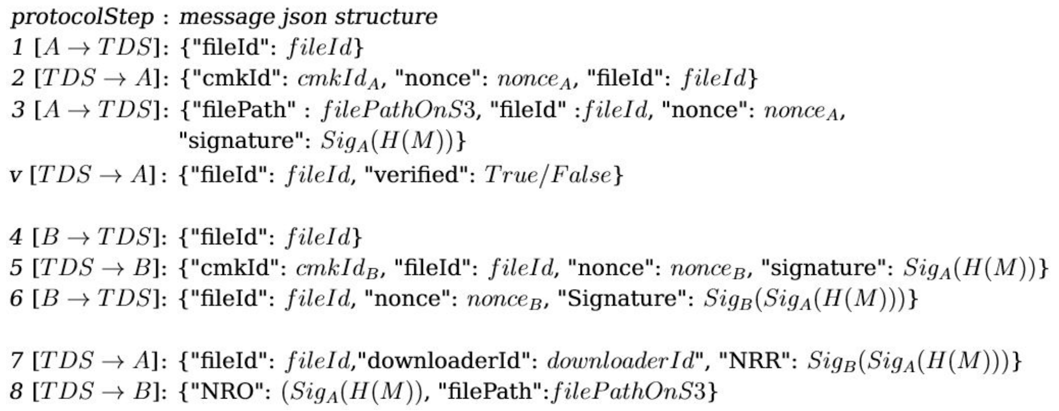
6 $[B \to TDS]$: {"fileId": $fileId$, "nonce": $nonce_B$, "Signature": $Sig_B(Sig_A(H(M)))$}

7 $[TDS \to A]$: {"fileId": $fileId$,"downloaderId": $downloaderId$", "NRR": $Sig_B(Sig_A(H(M)))$}

8 $[TDS \to B]$: {"NRO": $(Sig_A(H(M))$, "filePath":$filePathOnS3$}

Design: Message Transmission 3/3

Clarification of *message* dictionary variables:



protocolStep : *message json structure*

1 $[A \to TDS]$: {"fileId": $fileId$}

2 $[TDS \to A]$: {"cmkId": $cmkId_A$, "nonce": $nonce_A$, "fileId": $fileId$}

3 $[A \to TDS]$: {"filePath" : $filePathOnS3$, "fileId" :$fileId$, "nonce": $nonce_A$, "signature": $Sig_A(H(M))$}

v $[TDS \to A]$: {"fileId": $fileId$, "verified": $True/False$}

4 $[B \to TDS]$: {"fileId": $fileId$}

5 $[TDS \to B]$: {"cmkId": $cmkId_B$, "fileId": $fileId$, "nonce": $nonce_B$, "signature": $Sig_A(H(M))$}

6 $[B \to TDS]$: {"fileId": $fileId$, "nonce": $nonce_B$, "Signature": $Sig_B(Sig_A(H(M)))$}

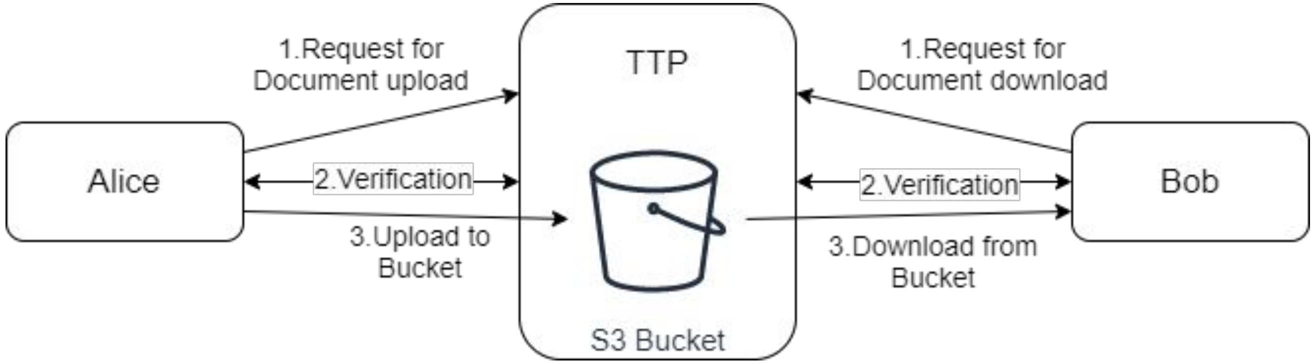
7 $[TDS \to A]$: {"fileId": $fileId$,"downloaderId": $downloaderId$", "NRR": $Sig_B(Sig_A(H(M)))$}

8 $[TDS \to B]$: {"NRO": $(Sig_A(H(M))$, "filePath":$filePathOnS3$}

- FileId -> Unique identifier of the file to be uploaded
- CmkId$_X$ -> Key ID assigned to user X
- FilePath -> File path of the file in S3 Bucket
- Nonce -> Random number generated for security purposes
- Signature$_X$(M)-> Signature generated by user X, by signing message M
- NRO -> Non-Repudiation evidence of Origin
- NRR -> Non-Repudiation evidence of Receipt
- DownloaderId -> ID of user that requested the file

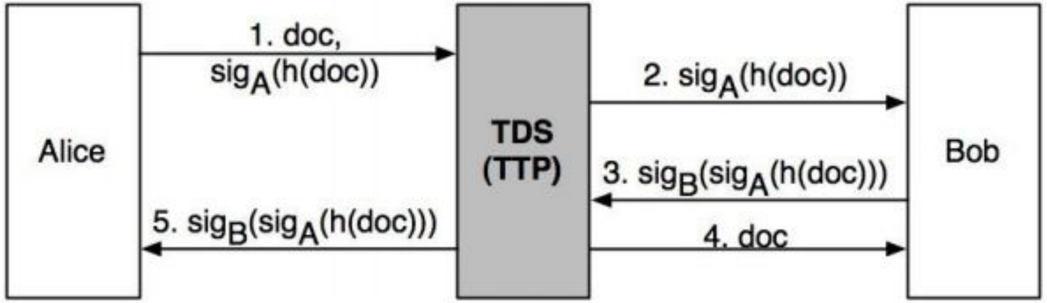
Design: Document Manipulation

- Used AWS S3 (Simple Storage Service) for document upload and download
- Users (via Django framework and Front-end) can upload a document to TTP’s S3 storage and after successful verification, the document is considered “verified”.
- Users can also download a “verified” document, only after a successful verification scenario
- Verifications and Signatures will be discussed thoroughly in the next slide
- S3 gives us the privilege to upload files in general (such as images, .txt files) and not only documents.

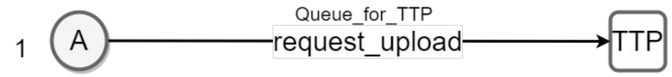


Design: Sign and Verify

- Use of AWS KMS (Key Management Service) in both Client and TTP side.
- TTP is generating a key-pair (using elliptic curve cryptography scheme ECC_NIST_P256 to further reduce key size) for every user in our system, once they request to either upload a file or request to download one.
- The key ID is sent over the Queues to let Alice or Bob sign (either Sig_A or Sig_B)
- TTP will receive from its Queue signatures signed by users and can verify them using their key ID.
- To guarantee correct signature, for the appropriate message, TTP is re-creating the hash of the uploaded file in Step 1 of the graph below and performs verification. The same approach is used for verification in Step 3 of graph, by inputting as Sig_B's message, the appropriate NRO.



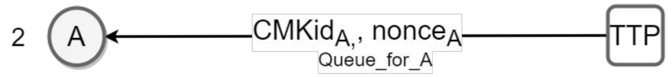
Step by step protocol explanation: Step 1 and 2



1 [$A \rightarrow TDS$]: {"**fileId**": $fileId$}

Note that the protocol step and user Id is sent in <u>each</u> message transmission!

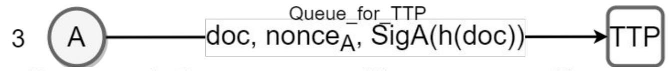
- A is requesting to upload a file with a file ID.



2 [$TDS \rightarrow A$]: {"**cmkId**": $cmkId_A$, "**nonce**": $nonce_A$, "**fileId**": $fileId$}

- TTP (after receiving protocol step 1), responds with protocol step 2, generating key ID (or sends an already existing key for the current user), a nonce and sending back the file ID

Step by step protocol explanation: Step 3 and v



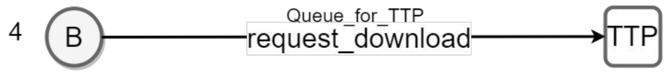
$$3\ [A \rightarrow TDS]\text{: \{"filePath" : } filePathOnS3\text{, "fileId" :}fileId\text{, "nonce": } nonce_A\text{,}$$
$$\text{"signature": } Sig_A(H(M))\text{\}}$$

- After A received protocol step 2, A uploads the file on TTP’s S3 Bucket and sends the file path on S3, the file ID, the nonce that was previously generated and the Sig_A of the doc (or file).

$$\text{v } [TDS \rightarrow A]\text{: \{"fileId": } fileId\text{, "verified": } True/False\text{\}}$$

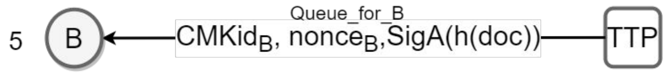
- TTP (after receiving protocol step 3), responds with protocol step v, sending back the file ID and a boolean called verified, that is True, if and only if, nonce and signature are verified. Upon verification the file is successfully uploaded.

Step by step protocol explanation: Step 4 and 5



4 $[B \rightarrow TDS]$: {"fileId": $fileId$}

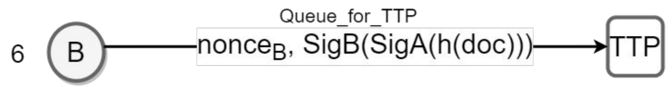
- B sends to TTP protocol step 4, with the file ID of the file that B wants to download.



5 $[TDS \rightarrow B]$: {"cmkId": $cmkId_B$, "fileId": $fileId$, "nonce": $nonce_B$, "signature": $Sig_A(H(M))$}

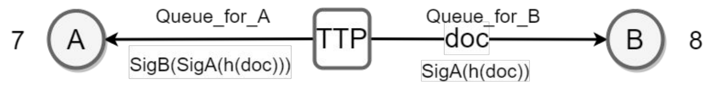
- TTP (after receiving protocol step 4), responds with protocol step 5, generating key ID (or sends an already existing key for the current user), a nonce and sending back the file ID as well as the signature generated from the uploader (or entity A) of the specified file.

Step by step protocol explanation: Step 6, 7 and 8



6 [$B \rightarrow TDS$]: {"fileId": $fileId$, "nonce": $nonce_B$, "Signature": $Sig_B(Sig_A(H(M)))$}

- After B receives from TTP protocol step 5, B returns the previously generated nonce and a Sig$_B$, signing the NRO of the specified document.



7 [$TDS \rightarrow A$]: {"fileId": $fileId$,"downloaderId": $downloaderId$", "NRR": $Sig_B(Sig_A(H(M)))$}
8 [$TDS \rightarrow B$]: {"NRO": $(Sig_A(H(M))$, "filePath":$filePathOnS3$}

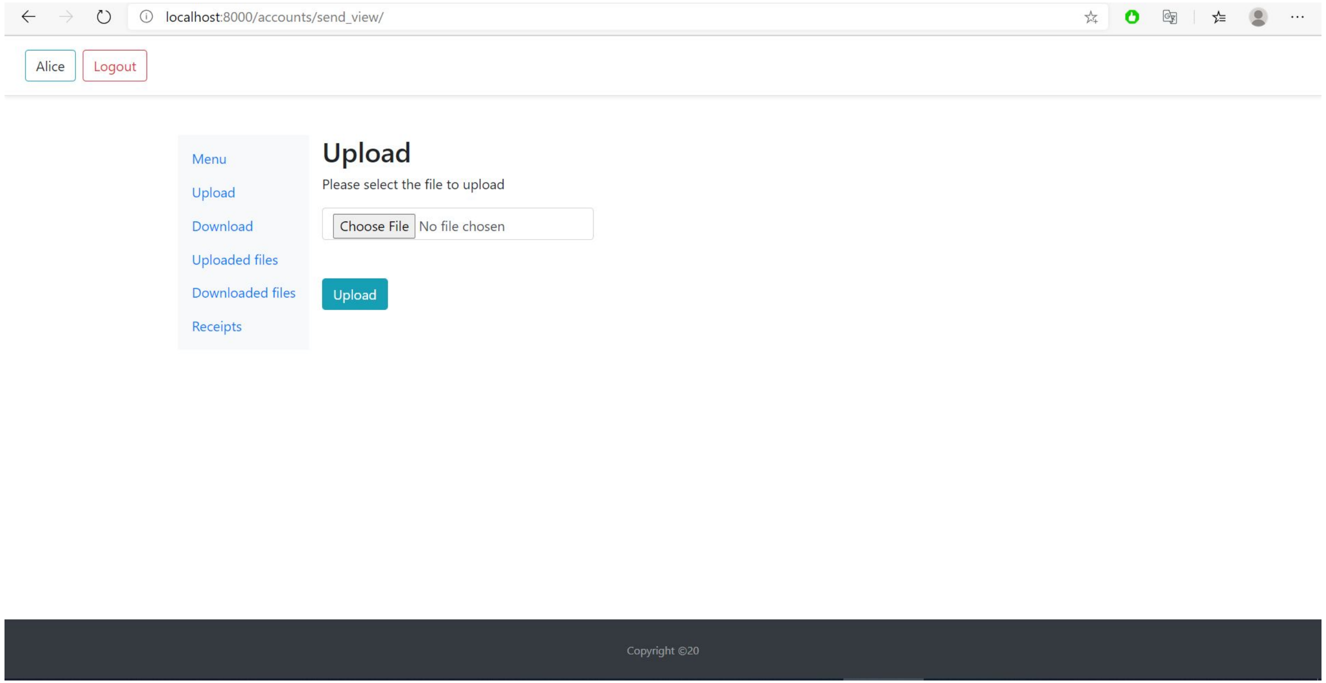
- TTP (after receiving protocol step 6), responds with protocol step 7 and 8.
- Step 7 sends the file ID of the specified document, the user ID of the downloader (or entity B) and the NRR to the uploader (or entity A).
- Step 8 sends the NRO of the specified document and the actual file path on S3 to download.

Implemented Protocol extensions:

- Security extension
    - SSE encryption for AWS SQS
    - Restricted access of AWS services only to specific security groups & users

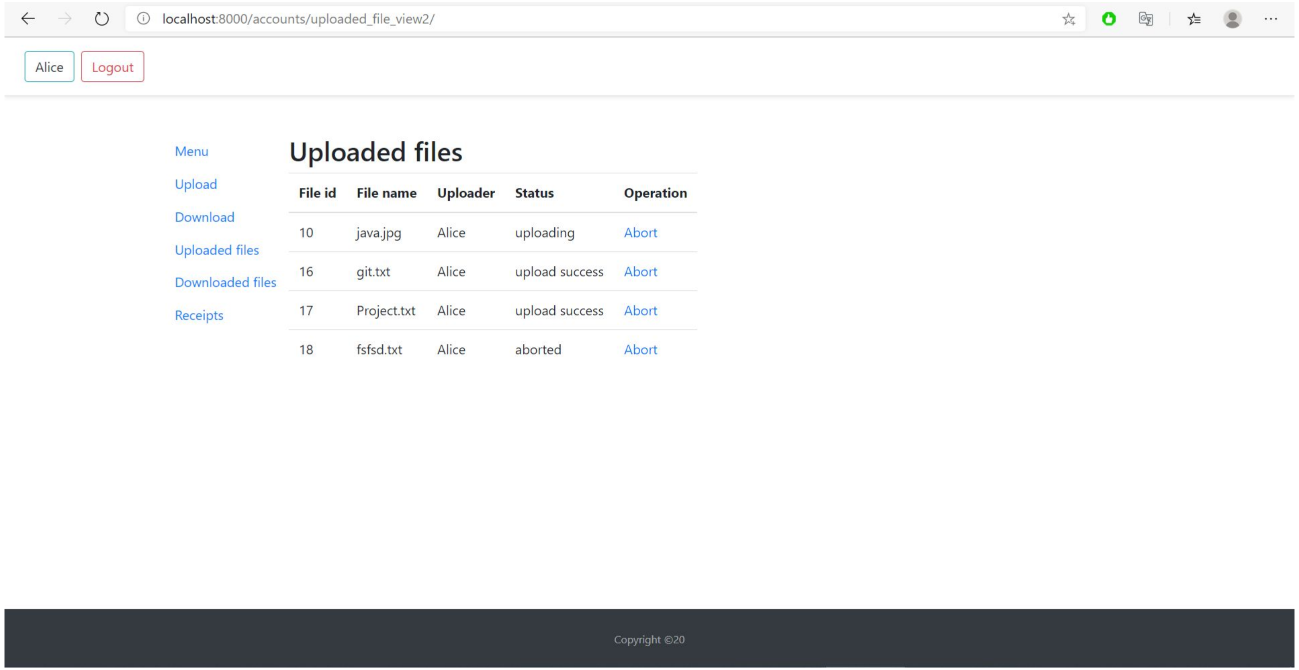
- Abort protocol implementation
    - A can request an abort
    - Abort is successful only when B has not obtained NRO & doc and A has not obtained NRR
    - Fairness guaranteed

User interface (1/5): Upload a file



A registered user can request to upload a file through this page. The *upload* button fires the protocol and sends a message to TTP Queue in order to get a verification for the particular upload

User interface (2/5): Uploaded files


localhost:8000/accounts/uploaded_file_view2/

Uploaded files

File id	File name	Uploader	Status	Operation
10	java.jpg	Alice	uploading	Abort
16	git.txt	Alice	upload success	Abort
17	Project.txt	Alice	upload success	Abort
18	fsfsd.txt	Alice	aborted	Abort

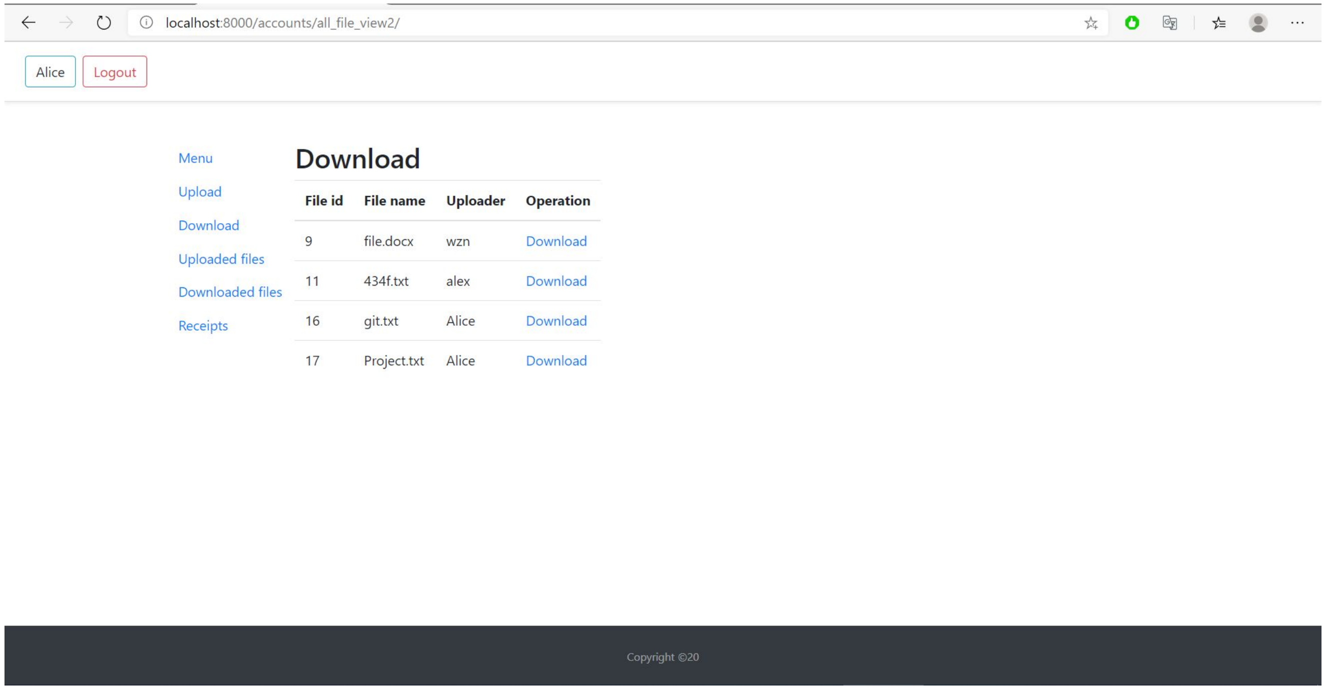
Copyright ©20

The status of each requested uploaded file is shown here for the currently logged user.

Uploading means that TTP has not verified the file yet, while ***uploaded*** means that the file is uploaded and the NRO is verified and saved on TTP.

There is an option to request an abort for any file. If successful, the Status will change to ***aborted***.

User interface (3/5): Download a file


localhost:8000/accounts/all_file_view2/

Alice Logout

Menu
Upload
Download
Uploaded files
Downloaded files
Receipts

Download

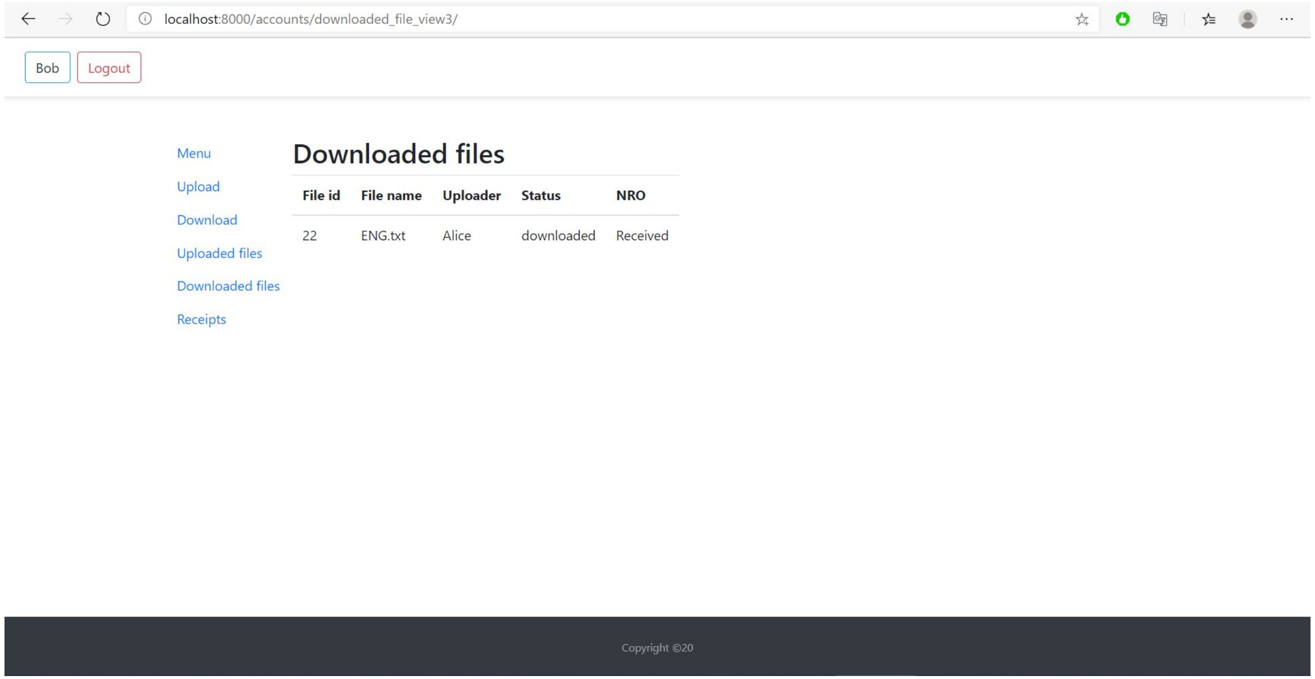
File id	File name	Uploader	Operation
9	file.docx	wzn	Download
11	434f.txt	alex	Download
16	git.txt	Alice	Download
17	Project.txt	Alice	Download

Copyright ©20

From this page a logged in user can request a specific file to download.

The download operation sends a request message into TTP Queue for the corresponding file.

User interface (4/5): Downloaded files


localhost:8000/accounts/downloaded_file_view3/

Downloaded files

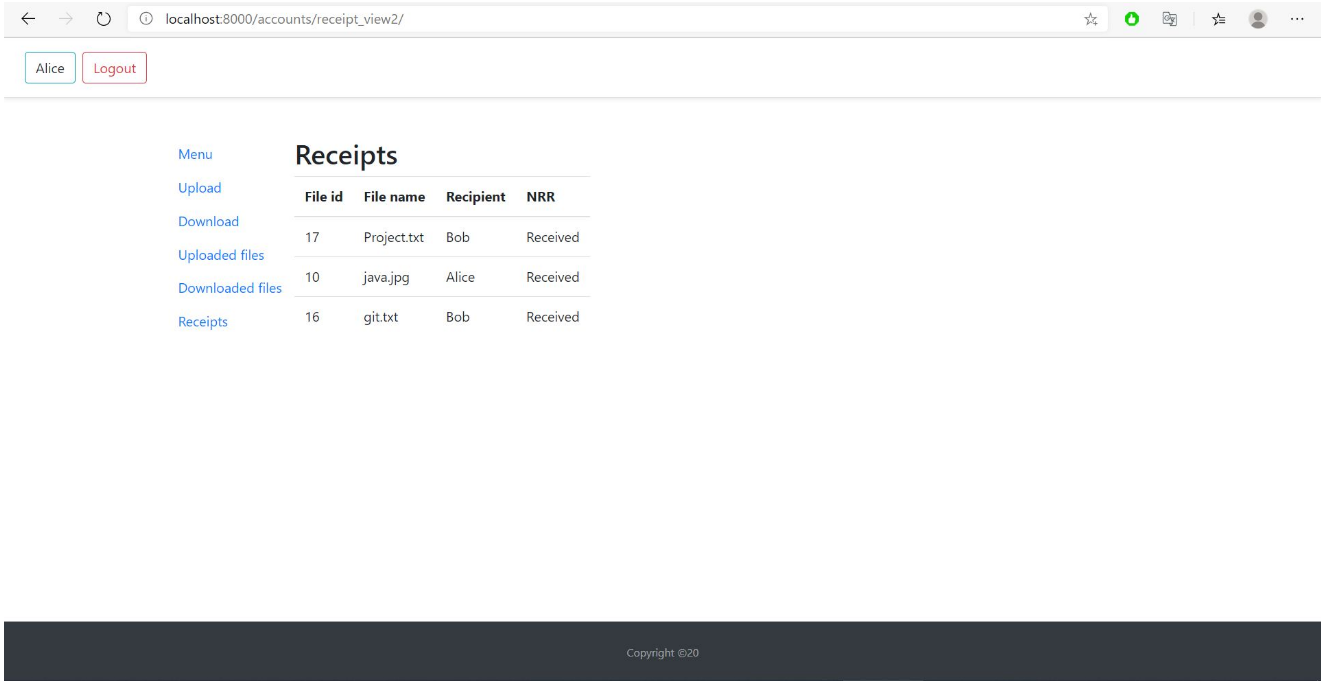
File id	File name	Uploader	Status	NRO
22	ENG.txt	Alice	downloaded	Received

Copyright ©20

Shows all the downloaded files from each registered user.

Downloaded stands for a verified download request and the user has received the NRO and the file.

User interface (5/5): Receipts


localhost:8000/accounts/receipt_view2/

Receipts

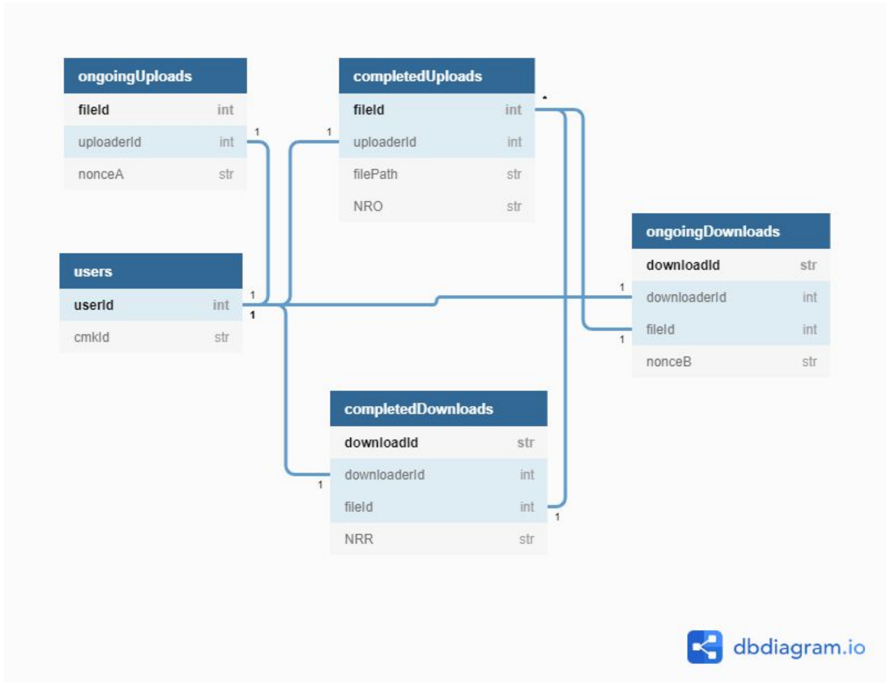
File id	File name	Recipient	NRR
17	Project.txt	Bob	Received
10	java.jpg	Alice	Received
16	git.txt	Bob	Received

Copyright ©20

Shows every verified download that was performed by each user.

Each recipient has received an NRR and each uploader of each file has received the corresponding NRO.

Database Schema: TTP/TDS



ongoingUploads	
fileId	int
uploaderId	int
nonceA	str

completedUploads	
fileId	int
uploaderId	int
filePath	str
NRO	str

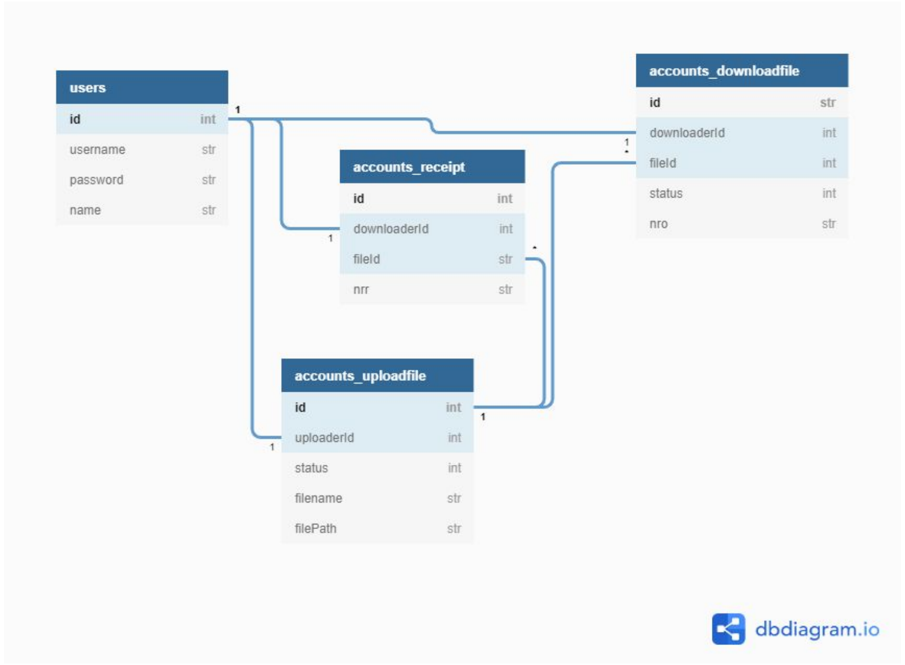
users	
userId	int
cmkId	str

ongoingDownloads	
downloadId	str
downloaderId	int
fileId	int
nonceB	str

completedDownloads	
downloadId	str
downloaderId	int
fileId	int
NRR	str

- Users table to store each users cmkId.
- Split uploads and downloads into two tables. The requests that are pending (***ongoing***) and the verified ones (***completed***).
- Each pending requests when verified, the corresponding row from ongoing table gets transferred to the completed one.
- Use of Foreign Keys & relationships to establish data integrity.
    - Download only files from CompletedUploads table
    - Uploader/Downloader must exist as a user in system
    - Cascade update/deletion
- For easier record retrieval;
  downloadId = downloaderId + “_” + fileId.
- Simple & Scalable database design

Database Schema: Client



users	
id	int
username	str
password	str
name	str

accounts_receipt	
id	int
downloaderId	int
fileId	str
nrr	str

accounts_downloadfile	
id	str
downloaderId	int
fileId	int
status	int
nro	str

accounts_uploadfile	
id	int
uploaderId	int
status	int
filename	str
filePath	str

- *Accounts_receipt* depicts each receipt given.
- *Accounts_downloadfile* shows all the downloaded files and *Accounts_uploadfile* all the uploaded files
- *Accounts_uploadfile status*:
  - 0 == fail upload
  - 1 == uploaded
  - 2 == uploading
  - 3 == abort
- *Accounts_downloadfile status*:
  - 1 == downloaded
  - 2 == downloading
- Use of Foreign Keys & relationships to establish data integrity.

Conclusion: Pros and Cons

Pros

- Amazon Simple Queue Service (SQS) simple, user-friendly interface and SSE encryption
- Amazon Key Management Service (KMS) simplistic and secure, sign and verification methods and key storage.
- Amazon Simple Storage Service (S3) guarantees scalability and accepts any file type rather than document files only.
- Amazon Relational Database Service (RDS) secure, auto-scalable, persistent storage.
- Django Framework pre-built authentication system & security middleware.

Cons

- Expensive system design for high-demand/high-traffic applications; multiple AWS pile up the cost.
    - Relatively costly SQS Service (50 cents per million requests in FIFO queues) in comparison to a free REST API, handling HTTP requests.
- SQS requires polling; system might be idle when message needs to be processed.