VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
INFORMATION TECHNOLOGIES STUDY PROGRAM

Semester Project

Privacy Oriented P2P Communication

Student:
Julius Valma

Supervisor:
Agnė Brilingaitė, Doc., Dr.

Vilnius
2023

Contents

Abstract

There is no truly *Peer to Peer* messenger on the market. Even the most privacy oriented messenger "Signal" still feeds every message through internal service [5], to hide sender's IP address and other purposes. So objectively, most private method to communicate is still offline conversation. Objective of this project is to solve this issue and make the most private and safe messenger.

The main idea of Mesa messenger is to create an active network of peers and routes between them to forward the messages in the background of the devices of the peers. This will allow to fully hide the IP address of sender, because when the message will reach the receiver it will have already traveled 4 to 5 peers.

Santrauka

Privati P2P Komunikacija

Rinkoje nėra nė vieno iš tiesų *Peer to Peer* žinučių programos. Net labiausiai į privatumą orientuota žinučių programa "Signal" vis dar perduoda kiekvieną žinutę per vidinį serverį, kad būtų paslėptas siuntėjo IP adresas. Taigi, objektyviai žiūrint, privačiausias bendravimo būdas vis dar yra tiesioginis pokalbis. Šio projekto tikslas - išspręsti šią problemą ir sukurti kuo privatesnę ir saugesnę žinučių programą.

Pagrindinė "Mesa messenger" idėja - sukurti aktyvų "Peers" tinklą ir maršrutus tarp jų, kad pranešimai būtų perduodami "Peers" įrenginių fone. Tai leis visiškai paslėpti siuntėjo IP adresą, nes kai žinutė pasieks gavėją, ji jau bus apkeliavusi 4-5 "Peers".

Introduction

Nowadays most people use messengers to communicate regarding private and work matters. While most of the users are regular people, for example, parents asking their children about how the day has been, there is a large group of society that uses the messengers as a safe space to share their concerns, ask for help in communities, and share personal information. Communication in this case becomes vital, and privacy of the conversations sometimes means safety of the people. In today's digital age, privacy-oriented messengers have become exceedingly important. First, messengers are an important mean in opposing governments (for example Iran and Russia). Second of all, users share private information day by day. It may be health state, information about sexual orientation or any other personal data that can be potentially dangerous for the sender if ever breached.

Privacy-oriented messengers give the users a mean of safe communication. In an era, when data itself became a commodity, many standard messaging applications (like *Facebook messenger, Viber*) are collecting user sent data - messages' content, time, location, and they potentially can use the data for ads personalization and targeting. On the other hand, corrupt governments ofter require these applications to provide full conversations on demand. For example, in 2016, the Russian government passed a law that required all foreign and domestic software companies to share any collected user data to the state government on a request. This led to Google moving some of their data centers to Russia [1].

Oppressive governments use control over messengers as a powerful tool to use against oppositions, for example the Iranian state has unlimited power over 3rd party foreign applications and just outright blocks them if there is even a slight possibility of any private messaging [1]. The Iranian government also deliberately slows down the Internet connection in the whole country to make the communication between the people even harder.

An important player in the before mentioned situations is *Telegram*. Created by Russian entrepreneur *Pavel Durov* after being forced by Russian state to disclose private users data from the *VKontakte* platform, which was created as a *Facebook* clone in 2006. Russian regime used private information as a tool for oppressing opposition [1]

After the pressure of state, *Durov* invested his own money into creating a truly privacy oriented application *Telegram*, and it entered the scene with a promise of freedom, privacy, and resistance; virtues that are engraved in the platform's design [1]. It was an important step in applications security, as many of the current solutions are not bone proof. According to Brilingaitė, Bukauskas and Kutka [4] an average messaging application is quite a dangerous one. About 10% of all requests and permissions made by the applications are potentially dangerous. Furthermore applications often store unencrypted data in local databases and even make calls to unsecured HTTP routes.

In contrast, *Telegram* is sharing first places with *Signal* in most secure mainstream messaging applications according to Botha, Van 't Wout, & Leenen [3]. The end to end encryption, decentralised mesh of 3rd party servers and open position of the authors of the application about not sharing user created data makes this messenger unreachable for governments.

In the year 2020 when the protests in Belarus began against presidential election fraud, *Telegram* played a major role in safe communication between the protesters. They could gather in certain places, or warn each other of coming war police. It even led to Belarusian state government to ban the Telegram together with other internet communication applications as covered by Wijermars and Lokot [9].

Although there are safe applications like Telegram and Signal, there are still major problems with them. They use the internal servers to pass on the messages between the users, and the users

can only trust that these vendors do not have the key to decrypt messages.

One possible solution to this problem is a decentralized approach, similar to a blockchain. Peer-to-peer (P2P) communication is not a new technology and was used from the very beginning of the Internet. However, nowadays, corporations tend to steer clear or P2P solutions for one simple reason---money.

Keeping everything centralised allow the vendors to collect the users' data. In the last 15 years all big tech players were focused on a single goal---collect the data. Data hoarding was and still is seen as a major way of earning money. The data lets vendors do better advertisement, sell it to interested parties and in return gather huge amount of resources from investment rounds. This data trend let to total centralisation of Web which at the end robbed users of privacy.

Peer-to-Peer application architecture could in theory solve main problem of centralised messaging applications---users have to trust the vendors. The trust is not something that engineers have a possibility to make, therefore a technical solution should be developed. To avoid this this paper suggests to use decentralised mesh of peers to handle all data transferring and forwarding to ensure secure and reliable communication.

Therefore, the goal of this project and paper is to provide a proof of concept for secure, P2P messenger application and present the design of the software which ensures the security capabilities. The prototype called Mesa is implemented using *WebRTC* protocol in the fronted with the backend acting as a manager of events.

The technical background of P2P communication is covered in Section 1. The privacy problems and technical solutions are covered in Section 2. Database model and mesa specific models are overeviewed in Section 3. The *WebRTC* protocol implementation in the *Mesa* application is covered in the Section 4. Communication protocol and how the requests are secured between the fronted and backend is in Section 6. Use cases are presented in Section 8. Functional requirements of the system are in Section 8. Non-functional requirements of the system are in Section 9. The system architecture overeview and in depth analysis of used technologies is in Section 10.

1 Background

1.1 WebRTC Protocol

In large thanks to *Blockchain*, there is a new P2P technology trend [6]. This led to an interest in *WebRTC* and similar non-server centric technologies. Also, privacy became one of important topics in the new software technology world. The European Union released new *GDPR* requirements and made the data privacy regulations more precise and factual [8]. These new trends make the motivation of this study to develop a concept of a truly privacy oriented messenger.

The main problem with even the privacy oriented messenger apps like *Telegram* or *Signal* is that the messages still travel through the apps servers [5]. *Signal* assures that it does not store messages on their internal servers, but it is only true as long as users believe in a multi-million dollar corporation. Generally speaking most of the messenger applications have the following process of sending the messages between the users as presented in Figure 1.

Figure 1 represents a generic flow of data that is true for most of the generic messenger applications. The steps are as follows:

Step 1: Sender compiles a text and sends the message to the receiver. Messages are always traveling to the internal server first. The protocol used is HTTPS which ensures the security

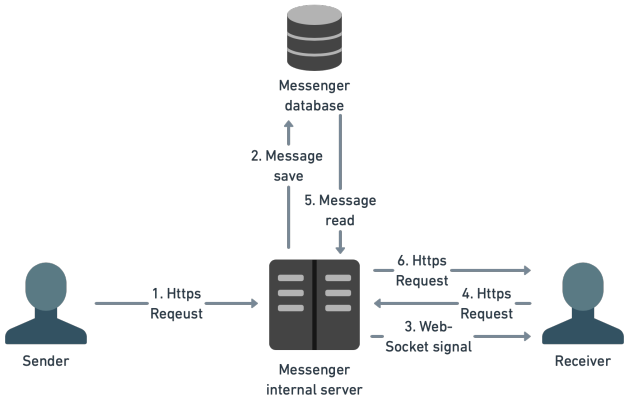

Figure 1. Generic messenger data flow

of the request to the server.

Step 2: Server receives the messages and processes it. It starts with authentication and checking if that message could be validly sent to the receiver. In most applications the data is then checked for malicious content and validated. After these steps the message is then saved on the database and the next step is initiated.

Step 3: Server initiates a signal to the receiver about a new message. This is done through *WebSockets* protocol on an always connected private user channel. The signal contains information that the receiver should re-fetch the messages.

Step 4: Receiver initiates the re-fetch of messages though *HTTPS* request.

Step 5: Server receives the request and fetches the newest messages from the database. After this, it responds to the request with a list of new messages.

Step 6: Receiver gets the newest messages and the whole process of message sending is over.

Having reviewed the whole data flow it becomes clear that saving user generated data on the server could be the main problem of all messenger applications. Knowing that it even seems more safe to use old *SMS* protocol as the messages are never saved on any 3rd party servers.

To counter this, *Signal* and *WhatsApp* started using end-to-end encryption. However this method has flaws as well.

Signal and *WhatsApp* claim that the user created content is always end-to-end encrypted. The logic of end-to-end encryption is presented in Figure 2.

In the figure, the locks represent encrypted and unencrypted data. From the start the sender compiles a message with the key to it and sends the encrypted message to the server. After that the server saves the encrypted message in the database. Now the receiver can fetch the encrypted message from the server and have the key to it - decrypt the message and show it to the user.

In this scenario, only sender and receiver can read the message, and the strategy secures the message transaction from man-in-the-middle attacks. However, a major drawback of *E2E* encryption is that it only works as long as users believe the service provider.

The core problem in all of the before mentioned solutions is data traveling through centralized internal servers. To solve this problem the communication had to be decentralized. To solve the

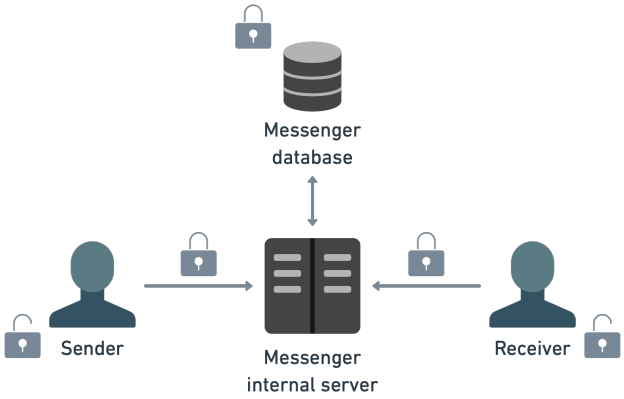

Figure 2. Generic end-to-end

problem *Blockchain* could be used, however the blockchain actually ensures that sent messages would occur on every one of the peers. Therefore, the *Blockchain* was rejected in this study. To make the direct connection between the peer *WebRTC* protocol was selected.

WebRTC is an open source communication protocol released by *Google* in 2011 [2]. It ensures media, such as video, audio or raw bytes, communication between the browsers of the users. It works just as any network connection, the data travels through open ports in the network. The important part is that two clients can talk directly with each other without any internal servers.

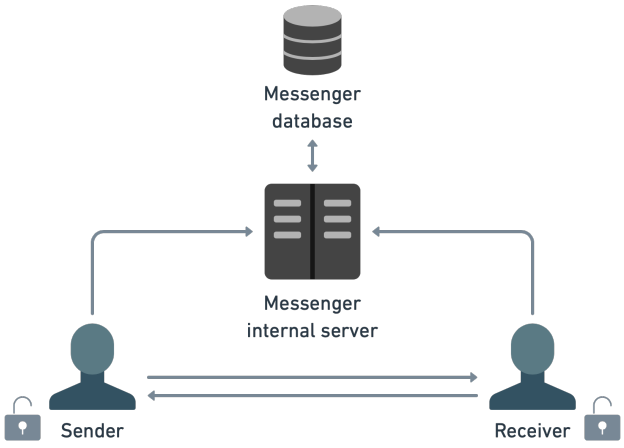

Figure 3. P2P Protocol

With *WebRTC* the connection could now look as presented in Figure 3. The server is responsible for authentication, user search and other non message related functions, and the users can communicate with each other directly.

In P2P, problem of hiding the users' identities is not solved. This was the main issue for *Signal* which does not use P2P only because it seemed impossible to hide sender IP address with *WebRTC* protocol, because to make any *WebRTC* request the receiver has to know the sender public IP address and vice versa. To overcome this, *Signal* passes the messages though the internal server and therefore uses the server as an improvised *VPN* solution.

Having a whole network of peers and forwarding the message between them may solve the issue. In Figure 4, all of the peers can communicate with other peers. The important part is all of these communications could be made in the background of the browser or the application. The

main idea behind prototype Mesa is not only sending messages straight from sender to receiver, but finding a longer path to forward the message between 3 to 5 peers before it reaches the sender. This ensures that the receiver never gets the sender identity and the forwarders do not know that they are the forwarders.

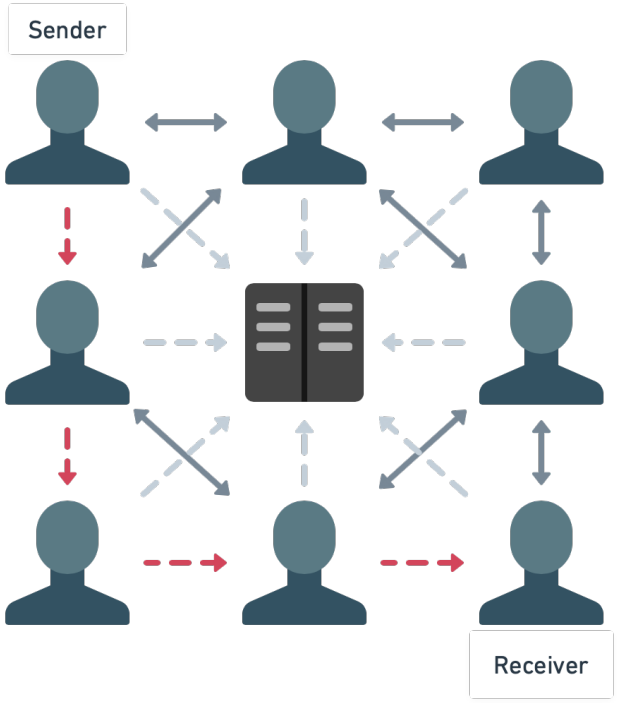

Figure 4. Mesa P2P Flow

The red dashed arrows in Figure 4 show the path which the message from sender to receiver will take. As shown in Figure 4, the connection to the server is still maintained by each of the peers, however data never actually reaches the server, therefore, protecting the users from data leaks.

The *WebRTC* protocol is pretty safe from man-in-the-middle attacks as well as a handshake between the peers is made before each transaction of data and the data itself is automatically end-to-end encrypted, that is ensured by the *WebRTC* protocol. However, to go the extra mile, *Mesa* will end-to-end encrypt the message again, to ensure that only sender and receiver have the key to

1.2 Privacy problems with other messengers: *Signal* Overview

To solve the privacy problem the message needs to somehow omit any servers that are not under user control. This is needed to ensure that message content is not shared with any 3rd party and is not forwarded through the server and that could in theory save the message author, IP address and time of the message. Biggest player in the privacy oriented messenger solutions is *Signal*.

Signal, although really technically advanced and open source, has a couple of positives and negatives:

Positives:

- P2P calls and video calls ar implemented.
- No user email needed for registration.

- All of the data is stored locally on the user device.
- Data is E2E encrypted.

Negatives:

- Each message is forwarded through a *VPN* like service to anonymize the sender.
- There is no possibility to choose to save the messages in the cloud.
- Hard to transfer account to another device.

Summing up, *Signal* is a suitable solution to ensure privacy, however, it is not a guarantee that messages are never saved on a server. It is logical to anonymize the user in the provided way, however, it still lacks the flexibility to go 100% serverless.

The *Signal*, as well as many other voice and video chats, uses *WebRTC* for video and audio streaming to other users. The Signal E2E encrypts all of the user data and sends it to the corresponding peer.

1.3 Generic P2P Application Implementation

To establish a connection between the peers any *P2P* application uses a flow similar to presented in Figure 5.

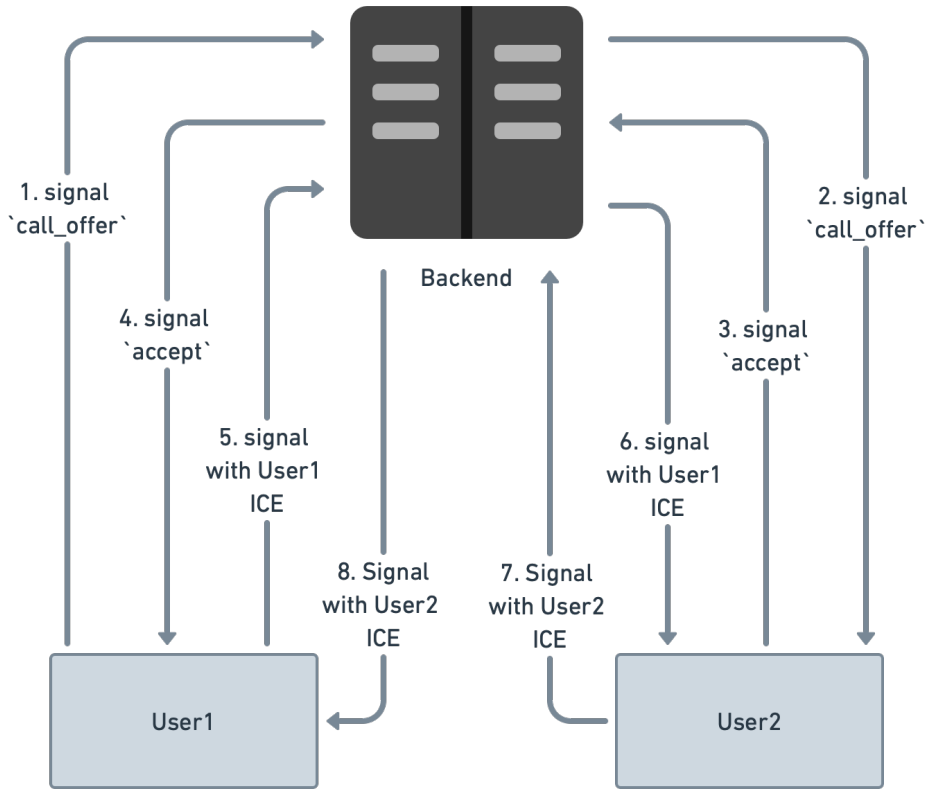

Figure 5. Generic P2P Implementation

All of the signals (displayed on the diagram) sent to and from the server are sent via *WebSockets*. This is the usual way of real-time communication in messaging applications.

The most important part of the connection process are steps 5 through 7 - when users interchange their **ICE**. *ICE* stands for *Interactive Connectivity Establishment*. It is basically a set of private and public IP addresses through which it is possible to establish a connection with a peer. It is generated most of the time though calling STUN servers and getting information about the user's internet location. The ICE candidate is then sent to another client and it is the key to the P2P communication. As expected, this is quite sensitive information, as the user's public IP address could deanonymize the user.

The problem with deanonymization is resolved in one of the Signal's patches. The connection process is actually terminated if the caller is not in the contact book of the receiver. But it was possible to get the exact user's DNS server just by calling their number on Signal.

The contact book principle is logical and resolves most of the problems, unless users want to chat with somebody they cannot completely trust. This problem is resolved with *Mesa* messenger prototype.

2 Privacy Problems and Solutions

2.1 The Mesa Messenger Anonymity System

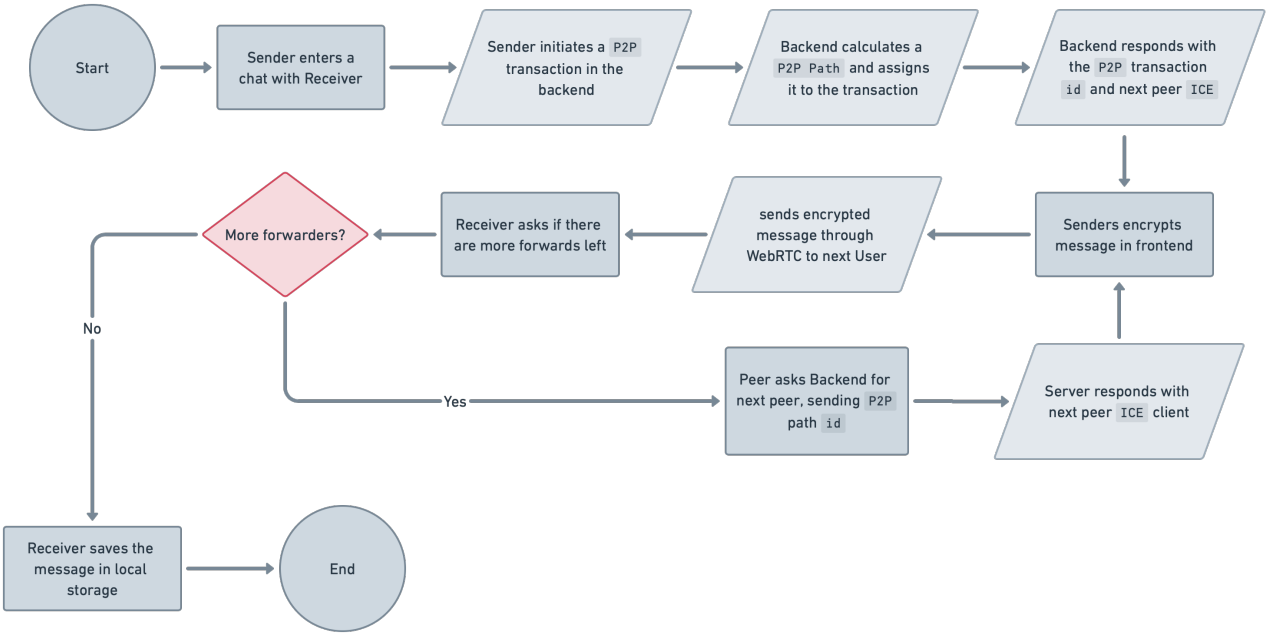

Figure 6. Mesa High-level System Flowchart

Figure 6 provides a high level overview of the *Mesa* messenger data flow. The flow starts with the sender entering *P2P* chat with the receiver. When the message is being sent by the sender the following steps are happening:

- Sender presses send button in the *P2P* chat.
- A *P2P* transaction is initiated in the backend.
- A *P2P* path is compiled and saved in the backend, the path finding is reviewed in more detail in section .....
- Sever responds with *P2P* transaction *ID* and *ICE* client of the next peer that should receive the message.
- Sender sends the message via *WebRTC* protocol to next peer which is the first forwarder.

Next item list will describe the forwarding process from the perspective of any of the 3 forwarders. The actions are the same for all 3 peers. Also none of the peers have any knowledge on their role in the *P2P* path. The backend only sends the *ICE* client of the previous and next peer, however no intel on the peer roles.

- Forwarder receives a *WebSocket* command *ready_to_receive* which makes the forward ready for receiving of the *P2P* message. The command contains the ICE client of sender.
- Forwarder initiates a *WebRTC* handshake with the previous peer.
- Forwarder receives the message and the *P2P* path *ID*.

- Forwarder sends request to the backend asking the next peer *ICE* client.
- Forwarder waits for the *WebRTC* handshake with the next peer.
- Forwarder sends the message to the next peer together with *P2P* pathd id.

When the message reaches the receiver the flow is as follows:

- The receiver receives a *WebSocket* command *ready_to_receive* which makes the forward ready for receiving of the *P2P* message. The command contains the ICE client of sender.
- The receiver initiates a *WebRTC* handshake with the previous peer.
- The receives the message and the *P2P* path *ID*.
- The receiver asks the backend for next peer *ICE*.
- The server recognizes that the call is made by the receiver.
- The server responds with the command to save the message in the given chat.
- Server marks the transaction as completed.
- The sender is notified that the message reached the receiver.

Message forwarding between the peers is the main idea of *Mesa* messenger. The forwarding is implemented to completely hide the identity of sender and receiver. This process is made in the background of client apps, without any mark of this in the frontend. The encrypted messages are not retained neither in the forwarders nor in the backend. The decrypted messages are only available for sender and receiver. To forward the messages safely and retain the privacy of sender and receiver, the pathfinding will be done asynchronously in the backend service and the resulting path will only be retained in the backend in in-memory database 'Redis'.

2.1.1 Understanding the Advantages of Mesa's Method

To understand the advantages of this method let's break this down:

- **Sender** - Knows the user entity of the receiver, and ICE of a random forwarder. Also has a private key that has the ability to decrypt the contents of the message.
- **Random Forwarders** - Only know the ICE of random senders and random forwarders. Also there is no additional information if the next peer is a forwarder or receiver.
- **Receiver** - Knows the user entity of sender and ICE of random forwarder. Also has a private key that has the ability to decrypt the contents of the message.

2.1.2 Perfect Message Rate (PMR)

Perfect Message Rate (PMR) - percentage of messages that receiver gets from sender.

To keep the PMR up to 100%, the receiver will notify the sender when the messages are received. A timeout of 10 seconds is set when sending out the message and if the signal from the receiver never comes - a new path is generated in the backend, the previous path is removed and the sending of the message is retried.

3 Database

Mesa uses *PostgreSQL* relational database management system to store data: user profile data, authentication data, information about active peers of the P2P system, which routes are available and lately used. Also, there is possibility to store conversations and files on the server, although it is more of a feature than core functionality, as main function of Mesa is to store messages on end user device storage system.

The reason for using relational database is that the stored data is quite specific. There are a lot of data joining therefore the associations are important.

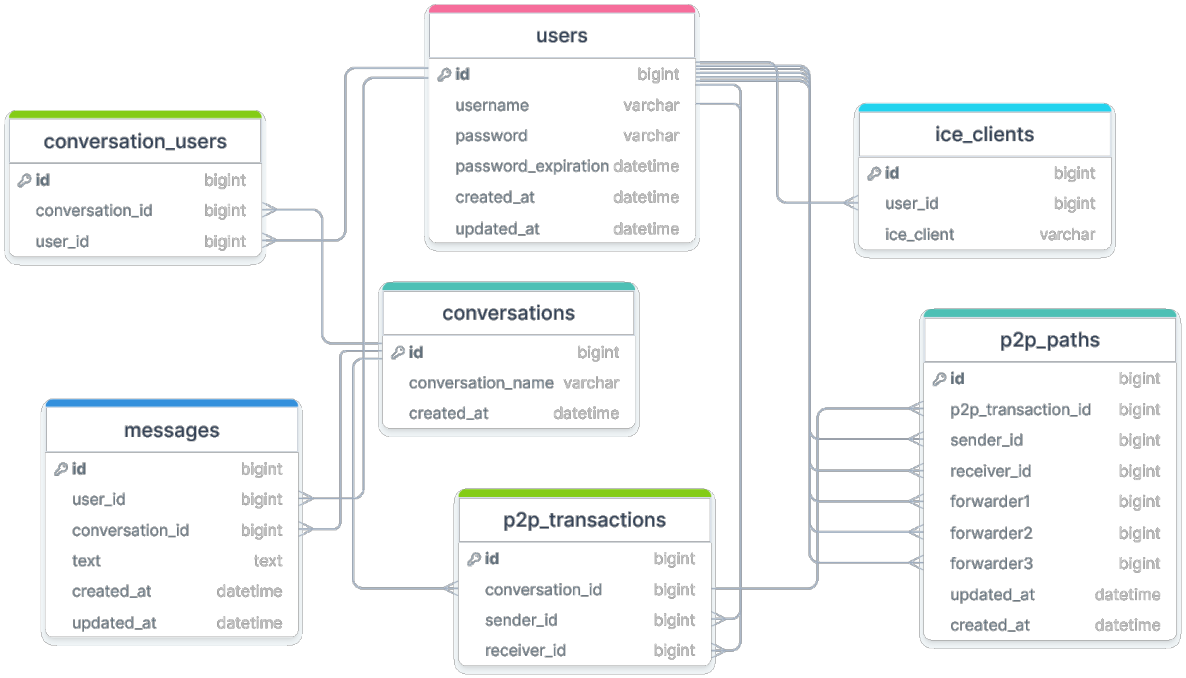

Figure 7. Relational model of the Mesa

The figure 7 represents the relational database of the*Mesa* backend. Most important table in this database is *users*. As every conversation in Mesa is held between users or user groups. The *conversation* is an object that can represent conversation between 2 or more users. *conversation_users* holds a list of users that are in a conversation. *files* is just a table that holds file references that are sent to any of the conversations. Also the filename is added, as files will be stored on separate database with random filenames for security.

The more interesting side of the database is on the right hand side. The *p2p_transactions* table stores *p2p messages* transaction objects. The transaction object mainly is the same idea as a transaction in the relational database. It is used to ensure that *p2p message* always reaches the receiver as it has the *completed* tag, which when toggled to false means the transaction is incomplete and right after the completion it is turned to true. This object is also used to implement the retry system. This is needed because P2P connections are not always as reliable as simple REST API, the peers could have ICE clients change and in this case the transaction will not be completed.

The *p2p_paths* table stores the compiled paths that are created for *p2p_transactions*. The list stores the user_id's of sender and receiver and also all forwarders that should receive the message. Also, when giving client the destination of the message, this table is joined with *ice_clients* table which in itself holds the iceś of the users. This value is updated every minute by every connected client.

The *ice_clients* table is crucial to ensure that the server always stores the freshest ICE clients of all connected users. The table entries are renewed every minute and the *updated_at* field is also refreshed. The *updated_at* field is needed for *p2p_path* calculation. The *ice_client* column is a varchar, which is hashed in the frontend of the application.

Converstions table holds connections between the users. There is a conversation author and a creation date. User can have many conversation users. The *conversation_users* table on the other hand holds the information about what users are in the conversation. It is not limited to only holding dialog objects, because it is possible to add unlimited amount of *conversation_users* to any given conversation. However, for now there is a constraint in the backend code limiting that functionality.

Messages table is for storing the messages sent by users to different conversations. The message content is stored in the *text* column and is stored as a simple string for now. There is no constraint to the length of the message in the backend, however the fronted limits the messages to 150 characters. *User_id* represents author of the message and the *conversation_id* represent the conversation to which the message was sent. Also the message editing is enabled in the backend and the frontend, therefore the *updated_at* field represent the editing date.

4 WebRTC

WebRTC (Web Real-Time Communication) is an open-source technology that enables end users to interchange data without forwarding it to a backend service of any kind. It is similar to LAN gaming back in the day, when to play multiplayer games with your friend you just had to be connected to the same network.

One of the most important benefits of this technology for *Mesa* messenger is decentralization and therefore security of the communications. The decentralization enables users to communicate between the frontends with great speed and reliability.

4.1 Connecting Two Peers

The whole process consists of two parts: Signaling and exchanging necessary information. The signaling part is quite straightforward and is implemented using backend service with WebSockets:

1. Client1 sends an offer to client2
2. Client2 accepts the offer and sends a signal to client1
3. Client1 gathers needed resources and sends it to client2
4. Client2 receives the resources and sends its own resources back to client1

The resources mentioned above are as follows:

- **Session Description Protocol (SDP)** - it contains information about client's supported media protocols and network information
- **Interactive Connectivity Establishment (ICE)** - it is an interface which contains information about possible communication channels between peers. Mainly it has the public IP address of the peer and possible network paths (IP and Ports pairs) which would make the direct connection possible. This information can be gathered with the help of STUN servers which work just like API - returning all needed information after calling them. Stun servers are mostly public and hosted by big tech companies like Google or Mozilla.
- **Datagram Transport Layer Security (DTLS) fingerprint** - it is a self-signed certificate which is used to make sure that the other peer is still the same.

4.2 The Security of WebRTC

As was already mentioned WebRTC uses DTLS fingerprints. What was not yet mentioned is that this protocol also ensures end-to-end encryption of any data sent through WebRTC. This ensures that even if somebody would get a hold of data sent from one peer to another - it would be encrypted and impossible to read [7].

Another important security measure is a certificate exchange between the peers. In the first step of the communication peers exchange the self-signed TLS certificates. These certificates are then saved on the client side and checked each time any new data is received from another peer. This prevents the MiTM (Man in the Middle) attacks [7].

5 Mesa P2P Implementation

5.1 Keeping the peers information

The Mesa client-side application is constantly connected to a signaling server via *WebSockets*. To ensure that every user has a fast and reliable way to chat with other peers, every 2 minutes the client sends an *ICE* client to the backend where the *ICE* is stored in the ICE clients table.

Variable	Description
ice_client	Information needed for a connection
user_id	User id of sender of this ICE client
updated_at	Timestamp when this ICE client was sent

Table 1. Description of Variables

This action ensures that, at all times, the server has a large number of clients to choose from to forward the messages between. Another important notice is that the freshness of an ICE candidate (time elapsed from the last update) is used as a weight property for the path-finding algorithm.

5.2 Path-Finding

```
FUNCTION generatePath(sender_id, receiver_id)
  path <- [sender_id]
  candidates <- getLast100IceCandidates()

  WHILE LENGTH(path) < 4
    sampled_user_id <- SAMPLE(candidates)
    APPEND(path, sampled_user_id)
  END WHILE

  APPEND(path, receiver_id)
  RETURN path
END FUNCTION
```

Figure 8. Mesa Path-Finding Algorithm

This bit of path-finding algorithm represents how simple the process is. The input is only the *sender* and *receiver* id, which is added to the front and back of the path.

Then the algorithm randomly selects peers from the *last 100 ICE clients* from connected users. For the time being the application is using *only 3 forward clients* but that can definitely change in the future.

What is not yet implemented is data redundancy. For now if any client disconnects or their *ICE candidate* becomes not valid, the path-finding will have to start from the beginning. In the future thought, forwarders redundancy is planned and minimum 2 possible paths will be executed at the same time to minimize the possibility of forwarding retry.

6 Frontend - Backend Communication

Mesa fronted - backend communication consist of the 2 main channels - *session WebSocket* and *REST API*. This is portrayed in the figure 9. The *session WebSocket* is connected on each login to the application and is constantly live until the user logs out or turns off the application. On the other hand, the *REST API* is only called on demand and works only one way - client to server.

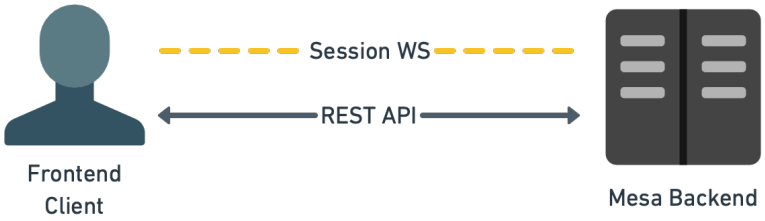

Figure 9. Fronted - Backend Communication Diagram

The *session WebSocket*, that is presented as a yellow dashed line in the figure 9, is used for different services and background use cases in the *Mesa* application. The use cases of this channel are as follows:

- User notifications - the notifications are of new messages or users trying to connect to P2P chat. As botch sender and receiver need to be connected for the message transaction to get completed, it is crucial to get the receiver of the message to know that he should connect. Most of those processes are done in the background, without any visual representation. However, when the messages are getting delivered, the notifications from the *WebSocket* channel may be shown in the UI.
- Refresh events - these events are crucial for interactive UI of the application. When using non *P2P* chat the receiver client of the message is notified in the background of the application, that the messages from this conversation should be re-fetched from the backend. This is done through the *WebSockets* because it does not require constant *ping's* to the backend to know when to refresh.
- Background *P2P* messages forwarding - these commands are constantly sent to all active peers. This is needed to ensure that *P2P* messages are forwarded through the forwarders. These commands include: *request to receive message*, *receiving of the ICE client of sender*.

On the other hand, the *REST API* that is presented as a dark grey solid line in the figure 9, is used for more user facing data flows. Those cases are as follows:

- User registration / Authentication - the interface for those endpoints are used on the login and registration. Data is sent non encrypted, however on the backend, the service encrypt password before storing it in the database.
- Account management - User has ability to remove the account whenever he wants. However, it is not allowed to create an account with the same username, because of the database unique constraints.

- Messages functionality in non *P2P* chat - Messages are sent with the simple HTTP requests and are then stored in the database. Also, every non *P2P* message triggers a *WebSocket* command to send a refresh signal to the other user from the conversation.

- User search - There is a user search functionality, which allows to find users easily. It is done through a simple SQL query in the backend, however the data is strictly sanitized.

It is important to notice that both *WebSockets* and *REST API* channels are divided into 2 kinds of routes. First kind is public, like the registration and login and the other one is protected, like user search, messages creation and fetching, all of the background *WebSockets* commands. This is done to prevent data theft and user impersonation. Although all of the protected routes are authorized with the help of cookie based token, there is another layer of security involved.

6.1 Security of User Generated Data

There is a couple of levels of security involved in crucially important *Mesa* endpoints. The user access to private endpoints is granted with *JWT* tokens. The tokens are generated in the backend with the help of *Devise* package for *Ruby on Rails*. The *Devise* package provides abstracted methods and classes for user authentication and authorization. It implements the password checks, password encryption (with the help of *OpenBSD bcrypt() password hashing algorithm* and username management. It also provides an interactor that can be used as a module in any class of the application, which contains current user object.

The *Devise* library also implements *JWT denylist* which is a way to manage the sessions. Every token issued in the backend has an expiration date of an hour but sometimes it might be useful to expire that token earlier, for example in case of logout or closing the app. In that case the token is added to a *denylist* which is a table in the database that stores *JWT* tokens which should not be accepted even if their expiration date is still in the future.

There is another layer of data protection in the *Mesa* application. Couple of validator classes were developed to ensure that the data always stays in the hands of owner. These classes can also be used as modules in any class of the application and the validators are as follows:

- *Conversation User Validator* - Checks if the current user, which is given by the *Devise* library, is in any kind related to a given conversation. It not only checks the owner of the conversation but also all users in *conversation_users* table. If the current user is not in a given conversation a *403* response will be given to the frontend. This ensure that by *URL* or *request body* manipulation, malicious user will not be able to access other users conversations.

- *P2P Transaction Validator* - Checks if the current user is a *P2P Path owner*. This check is crucial, because it restrict an unauthorized user of doing any actions with the *P2P Transaction* which could result in Man in the Middle attacks.

- *P2P Path Peer* - Checks if a user is in a *P2P path*. This ensures that no user steals the path and gets the next peer ICE for malicious reasons. This also ensures that Man in the Middle attack is not possible.

7 Use Cases

The diagram below displays the use cases of the Mesa messenger. There is just one actor, which is the user of the system. There will be no different roles in this system. Main things that user will do is send, receive, edit messages, find users to chat with, hide the identity by forwarding the messages in the background between different peers.

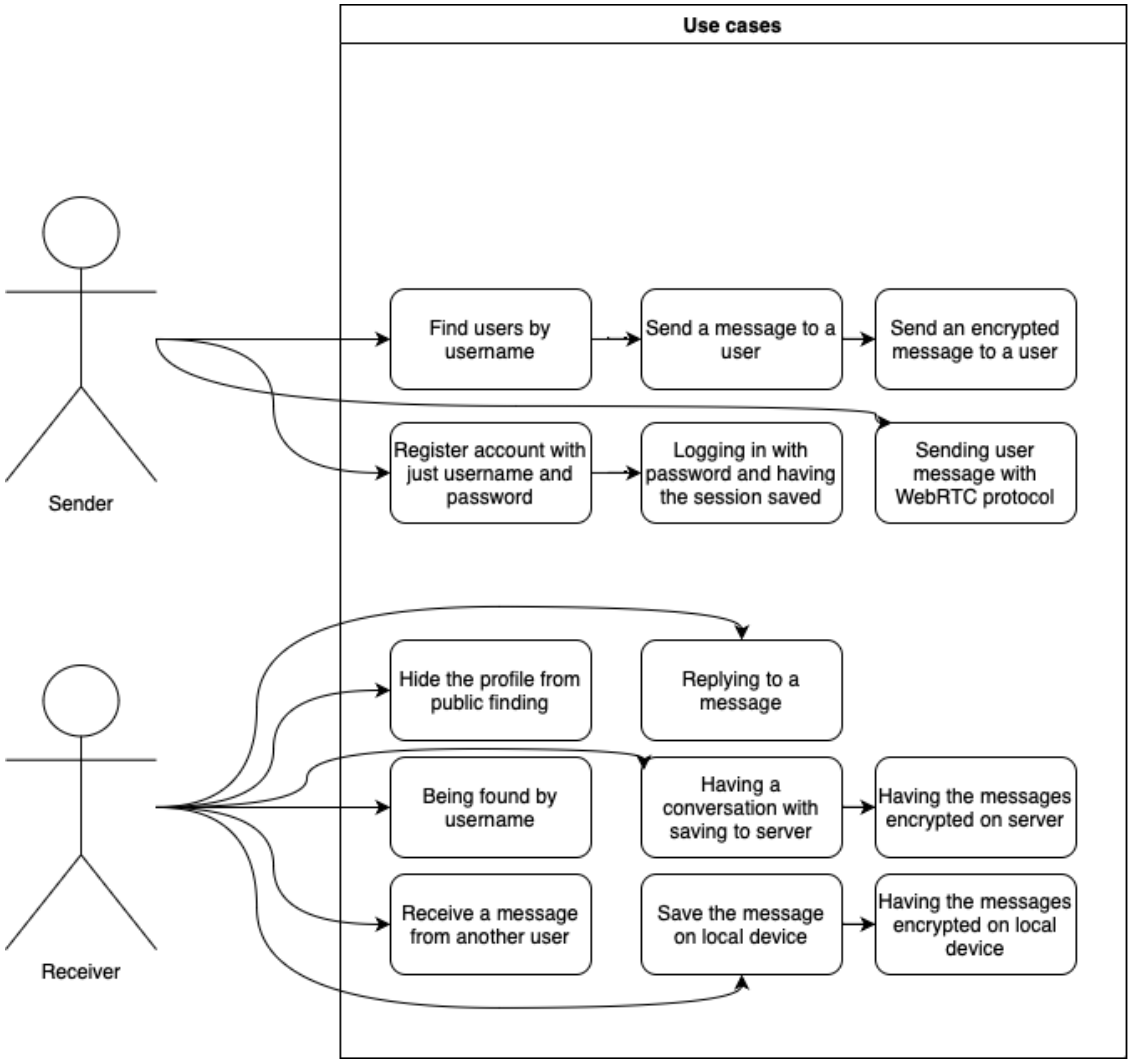

Figure 10. Mesa messenger use cases

The most important in any messenger is to send and receive messages to and from fellow users. What makes the Mesa different is possibility to send the messages via WebRTC protocol. This will allow users to send and receive messages not through our internal service, but directly between the peers.

Also, it is really important that the messages will be end-to-end encrypted. This will make the privacy of the messenger even better. Moreover, the encrypted messages must be saved on the local device and bu user choice not synchronised with the server.

8 Functional Requirements

The functional requirements are a way for the user to communicate with the developers. Therefore, the team has decided on using user story template to represent functional requirements.

- As a User, I expect to be able to run messenger on all popular platforms (MacOS, Windows, Linux, IOS, Android).
- As a User, I want to create an account and have a safe way of authentication.
- As a User, I want to have a possibility to store the messages only on my local storage device.
- As a User, I want to have a possibility to send the messages directly to another user (without any internal services).
- As a User, I want to have a possibility to send and store messages on the internal server (so that messages could be transfered to another device).
- As a User, I expect all of sent messages to be end-to-end encrypted.
- As a User, I expect to be able to send and receive files.
- As a User, I expect to have a possibility to completely hide my identity (including IP address).
- As a User, I expect to have a possibility to transfer my local saved messages to another device.

9 Non-functional Requirements

SECURITY All of the requests must be secure, therefore encrypted. All messages must be end-to-end encrypted and must only be readable with private key of the user. Also, all inputs must be validated to ensure data safety.

COMPATIBILITY System must be cross compatible and easily run on different platforms. Platforms include: MacOS, Windows, Linus, IOS, Android, etc.. It should be easy to switch and a method of transferring messages from one platform to another must be ensured.

USABILITY Although Mesa will be advanced on the backend and ensure maximum privacy and seccurity, the users must be able to easily use the application and there must be no learning curve when coming from other messengers. Therefore, the interface must be simple, easy to use, intuitive.

SCALABILITY Mesa must be ready to a very large scale. The most important thing about any messenger is how many people you can reach out to with it. For this purpose, from the very beginning database caching and sharding must be implemented as well as advanced deployment with kubernetes.

IMMUTABILITY Database of message will be decentralised, meaning all members of any conversation will have a copy of the data on their local device storage. Immutability is key for privacy and reliability of the service.

10 System Architecture

10.1 Architectural Goals and Constraints

These are some of the architectural goals and constraints that need to be met for the project to be considered a success:

- Mesa must have a way of storing encrypted messages locally on device storage. When choosing this storage option, it must be ensured that messages are not saved on internal Mesa servers.
- The sent messages must be end to end encrypted with a possibility of decrypting them with private key of each user.
- Mesa must be cross-platform (*MacOS, Windows, Linux*) and preferably on the mobile as well (*Android, IOS*).
- Mesa must ensure seamless P2P communication without additional user input.
- Mesa must be able to create a network between multiple Peers (>5). Therefore find the paths between the peers and make the paths the most efficient.
- Mesa messenger application must be able to send the requests in the background to forward messages between the users.

10.2 UML Demployment Diagram

The UML deployment diagram models the physical deployment of software components. In the diagram, hardware components, such as web servers, are presented as nodes, with the software components that run inside the hardware components presented as artifacts. The Mesa backend service is deployed on Open Nebula servers and communication with peers is established via HTTP requests and *WebSockets*.

The peers on the other hand, are communicating with each other via *WebRTC* protocol, which is fully Peer-to-Peer and does not go through any internal servers of the Mesa. The backend service is only needed to capture active connections of the peers and forward the nodes of connection to other peers to make the connection possible. This is one of the main challenges with *WebRTC* protocol - there is no look up table, therefore the Peer needs actively tell their connection to the other Peer.

First the peers find their own active connection to which they can receive data. After the peers send their possible connections to Mesa server, where the connections are stored in nodes table. After all of this, Mesa server with the help of path finding algorithm find a path between the peers and sends the path back to the peers. After receiving path the peers start sending the messages to all of the peers in the network.

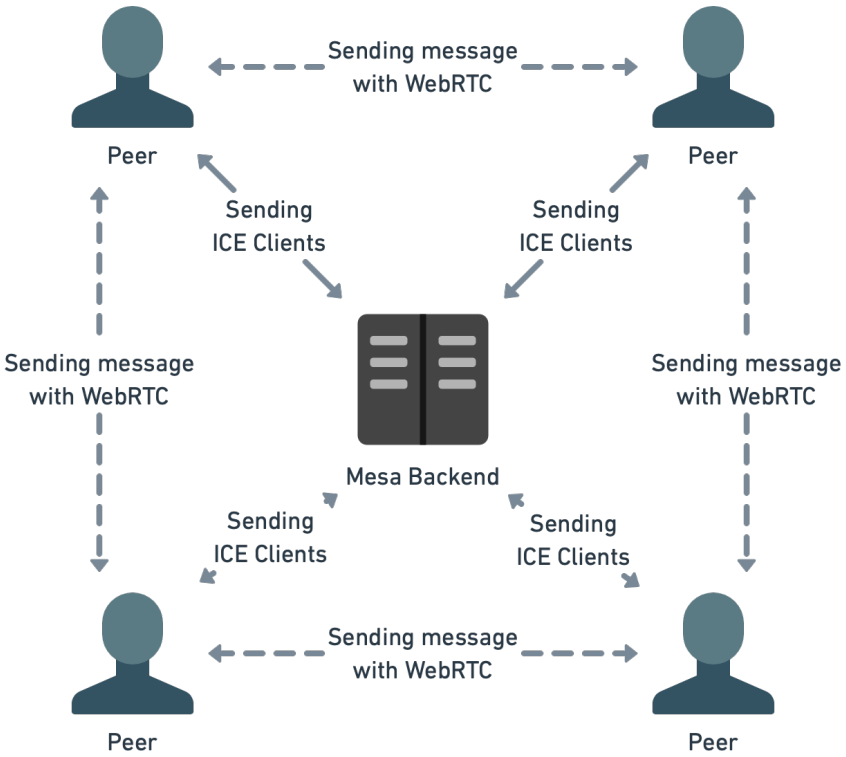

Figure 11. Peers and server representation

Ubuntu Web Server(BACKEND)

- Backend service is written with *Ruby* with *Ruby on Rails* framework. It is chosen for good *WebSockets* capabilities and Redis compatibility
- Redis will be used as a database caching service to make the speeds of reading even faster.
- *PostrgreSQL* will be used as a database, because it is lightweight and open source.

Electroj.js Client (FRONTEND)

- Frontend of the Mesa will be written in Electron.js, which will ensure cross compatibility.
- Electron.js also enables frontend to communicate through *WebRTC* protocol.

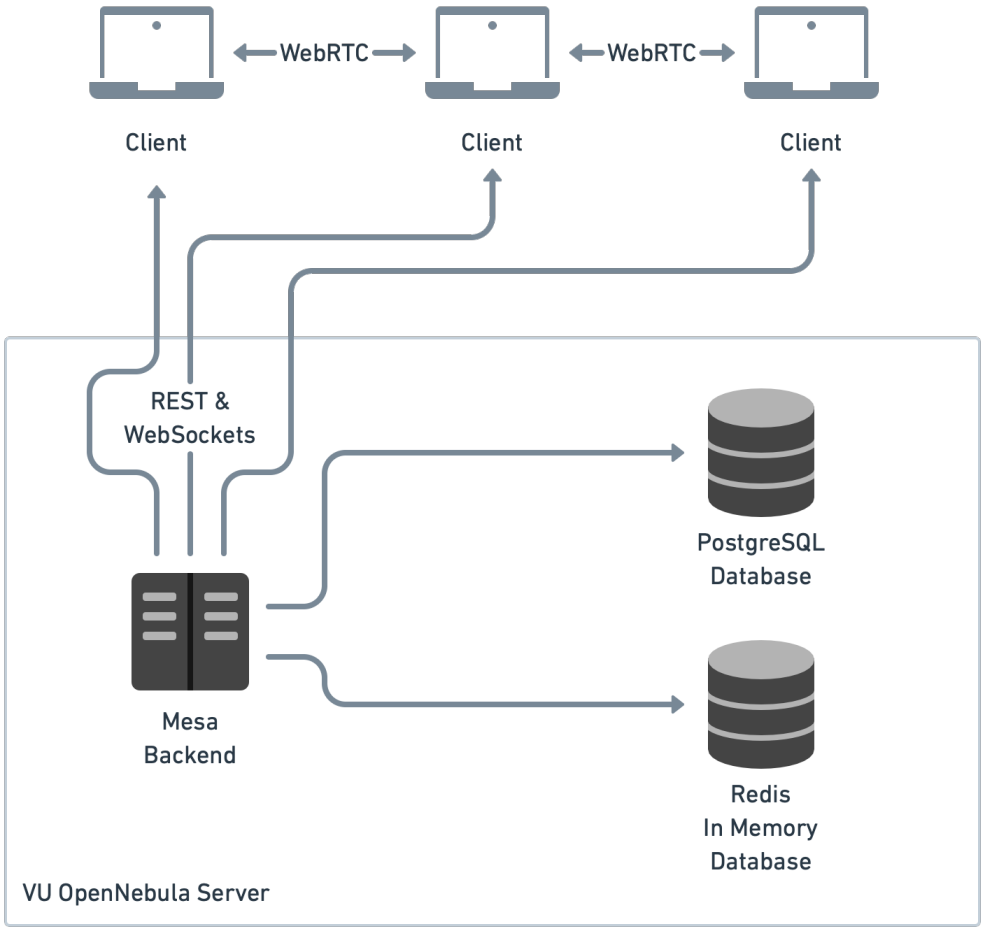

Figure 12. UML Deployment Diagram

As shown in figure 12 the clients communicate with backend via *Rest* and *WebSockets*. The *WebSockets* are needed to ensure real time connection of all the peer to the backend service. This is crucial when the clients have 2 roles: just client and a peer that can forward the messages. The clients send their *Ice client* data to the server every minute, to ensure that the peers information is up to date and best quality of communication can be ensured.

The *Redis* is a key value in-memory database that is used for caching the most often fetchable data. For example the clients *Ice* candidates are stored there to ensure the fastest message forwarding possible.

References

[1] A. Akbari and R. Gabdulhakov. Platform surveillance and resistance in iran and russia: The case of telegram. *Surveillance Society*, 17(1/2), 2019.

[2] Harald Alvestrand. Google release of webrtc source code. https://lists.w3.org/Archives/Public/public-webrtc/2011May/, 2011. Accessed: yyyy-mm-dd.

[3] J. Botha, C. Van 't Wout, and L. Leenen. A comparison of chat applications in terms of security and privacy. In *18th European Conference on Cyber Warfare and Security (ECCWS)*, July 2019.

[4] Bukauskas L. Kutka E. Brilingaitė, A. Detection of premeditated security vulnerabilities in mobile applications. In *18th European Conference on Cyber Warfare and Security (ECCWS)*, July 2019.

[5] Brian X. Chen. Worried about the privacy of your messages? download signal. *The New York Times*, Dec 2016.

[6] Y. Chen and C. Bellavitis. Blockchain disruption and decentralized finance: The rise of decentralized business models. *Journal of Business Venturing Insights*, 13:e00151, 2020.

[7] B. Feher, L. Sidi, A. Shabtai, and R. Puzis. The security of webrtc. Ben-Gurion University of the Negev. Available at: mailto:feherb@bgu.ac.il,liorsid@bgu.ac.il,shabtaia@bgu.ac.il, puzis@bgu.ac.il.

[8] P.M. Schwartz. Global data privacy: The eu way. *New York University Law Review*, 94(4):771, 2019.

[9] M. Wijermars and T. Lokot. Is telegram a "harbinger of freedom"? the performance, practices, and perception of platforms as political actors in authoritarian states. *Post-Soviet Affairs*, 38(1-2):125--145, 2022.