TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Ioane Sharvadze
166798IVSM

FRAMEWORK FOR PEER-TO-PEER DATA SHARING OVER WEB BROWSERS

Master's Thesis

Supervisor: Vishwajeet Pattanaik
M.Tech.

Supervisor: Dirk Draheim
PhD

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Ioane Sharvadze
166798IVSM

RAAMPROGRAMM PEER-TO-PEER ANDMETE JAGAMISEKS VEEBIBRAUSERITE KAUDU

Magistritöö

Juhendaja: Vishwajeet Pattanaik
M.Tech.

Juhendaja: Dirk Draheim
PhD

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Ioane Sharvadze

06.05.2018

Abstract

The Web was originally designed to be a decentralised environment where everybody could share a common information space to communicate and share information. However, over the last decade the Web has become increasingly centralized. Web applications such as Facebook, Twitter, Google and others store user data in centralized data stores or data silos; each controlling their silos with their own authentications and access control mechanisms. This has led to serious concerns about data ownership and misuse of personal data. Furthermore, not only such centralized applications prevent users from switching between similar applications, but they also disallow data exchange between different platforms.

While there are approaches to solve these problems, none of them provide simple and extendable solution. To this end, this thesis presents an application independent, browser-based framework for sharing data between applications over peer-to-peer networks. The framework empowers the end users with complete data ownership, by allowing them to save sharable data locally and share it directly with friends.

The thesis presents the functional requirements, implementation details, security aspects and limitations of the framework. It also discusses the challenges that were encountered while designing the framework, especially why it is difficult to create a server-less application for the Web.

The final part of this thesis is a demonstration of the framework through a chat application, designed as a Google Chrome browser extension. This is achieved by testing the different aspects of the framework while exchanging data over networks in different countries.

Keywords: Decentralization, data ownership, human-computer interaction, peer-to-peer, social web, security, web apps, WebRTC

Thesis is written in English and is 54 pages long, including 7 chapters and 26 figures.

List of abbreviations and terms

WebRTC	Web Real-Time Communication
TUT	Tallinn University of Technology
P2P	Peer-to-peer
IP Address	Internet Protocol address
NAT	Network address translator
RDF	Resource Description Framework
URL	Uniform Resource Loader
STUN	Session Travel Utilities for NAT
TURN	Traversal Using Relays around NAT
JSON	JavaScript Object Notation
HTTP	Hypertext transfer protocol
SHA-1	Secure Hash Algorithm 1

Table of contents

List of figures

1 Introduction

Over the last two decades the World Wide Web has become an integral part of our lives. Thanks to the increasing number of online encyclopedias, news outlets, wikis, media sharing websites and social media platforms; the web has transformed from a mere medium for broadcast to a dynamic social environment. Internet-based applications today, allow users to create and share unprecedented quantities of knowledge and information [1]. However, this increase in number of social Web applications has also given rise to numerous disconcerting issues such as filter bubble, fake news and privacy concerns.

These web applications and service often come in different forms. Most of these services are centralized systems and store user resources in closed data silos. Due to this nature of web applications, users typically end up creating dedicated local accounts and are bound to particular services and resources [2] [3] [4]. Furthermore, users trust web applications and service providers such Facebook, Google Plus, Twitter and others to store and manage their personal data with the intention to receive personalized services. However, as recent incidents indicate such centralized data could also be utilized to harvest user data, manipulate user mindset [1] and to spread fake news and propaganda. In order to achieve true data ownership and flexibility, it is necessary to design a distributed, application-independent data sharing mechanism for the Web.

1.1 Goal of the thesis

This thesis focuses on enabling peer-to-peer data sharing between web applications. The goal is to reduce dependency on centralized servers (i.e., data silos) and to empower end-users with true data ownership. Ultimately, we would like to 're-decentralize' the Web by eliminating the need for servers and allows users to communicate with each other without any middleware.

The proposed framework was originally intended to enable P2P communication for a real-world application called WebWeaver. The WebWeaver application is currently being developed as Google Chrome browser extension that allows end-users to weave and share

web page elements, independent of content ownership on the Web. However as mentioned earlier, due to the recent increase of interest in re-decentralization of the web, it is imperative that such a P2P framework must be designed as a generic system. This would empower both web developers and end-users by providing API based server-free communication between applications and preventing misuse of user data, respectively.
This thesis presents the client library and web services, that allow applications to store data on user devices and share them over a P2P network.

1.2 Literature Review

This chapter describes some of the recent scientific contributions associated with data ownership and privacy concerns. These platforms allow users to create and share data via decentralized networks. Section 1.2.1 discusses the 'Musubi' a mobile social application platform that enables secure, real-time data sharing with the phone. Sections 1.2.2 – 1.2.4 discuss decentralized social Web applications, CIMBA, Solid and Dokieli. Analyzing these platforms and understanding their approaches, contributions and limitations would help to identify the gaps; and thus, state a clear problem definition.

1.2.1 **Musubi**

Musubi is a mobile social application platform that enables users to share data in real-time feeds. The platform ensures data safety and privacy by supporting end-to-end public key encryption. Musubi allows users to interact with their friends directly through their address books. Additionally, all user data is stored on the phone. Therefore, provides end users with a complete data ownership.
Framework is dedicated for mobile app communications, and because establishing direct P2P connection over mobile networks (3G) is impossible, data transfer is completely dependent on a centralized service Trusted Group Communication Protocol [5].

While Musubi's goals seem similar to ours, there are some subtle differences. For instance, unlike our system Musubi only supports group sharing and does not support public data sharing. Also, Musubi always requires a server for data transfer, where as our system does not require a server to store or relay data, in case both sender and receiver are online.

1.2.2 **CIMBA**

CIMBA or Client-Integrated Micro-Blogging architecture is a decentralized social Web application that increases data ownership by allowing users to choose where their data is stored. The architecture uses WebID [3] and WebID-TLS to identify users at the Web scale and to authenticate requests.

1.2.3 **Solid**

Solid is a decentralized platform for social Web applications, that allows users to manage their data independent of applications. Unlike conventional web applications, users are required to store their data in a personal online data stores (i.e. pods). Applications are allowed to access user data, based on the permissions provided by the user. The users are identified using WebID [3] and have a full control over how their data is accessed and can switch between applications and pods at any time. The platform uses RDF based resources to exchange data between applications and pods [2].

Although Solid supports data ownership, but it still stores data on non-user devices. It might be hard for users to set up servers or find free hosting service. It would be better if users could start using applications without any configuration and store their own data on own personal computers. Also, linked-data does not provide solution for real time peer to peer data sharing; for this reason, it would be challenging to develop social real-time applications such as chats on such a platform.

1.2.4 **Dokieli**

Dokieli is a decentralized browser-based authoring and annotation platform. The platform supports social interactions and allows users to retain the ownership of their data. Documents created in dokieli are independent and interoperable as they follow the standards and best practices of HTML, RDF and Linked Data [4].

Like CIMBA and Solid, dokieli also enhances user's data ownership experience. However, each of these platforms are bound by their technologies; which may discourage

developers from adopting these recommendations. Lastly, as mentioned earlier none of these platforms allow storage on personal computers and end-to-end communication.

1.3 Research Questions

Given the lack of a generic, application-independent data sharing platform for P2P networks over the Web, we propose the following main research questions:

1. How to establish P2P network between web browser?

The aim is of this research question is understand how P2P networks could be established over web browsers. And, how could Socket.io and WebRTC [6] could be used to establish communication in such a scenario? This is also important because, it would help solve NAT traversal [7] problem using JavaScript.

2. How to create a server-less P2P network?

This question is important to establish if P2P networks can be initiated in truly server-less conditions. If it's impossible, what other alternate approaches could be used to reduce dependency on server?

3. How to forward messages in case receiver if offline?

Since, the proposed framework is to be designed for P2P networks, it is important to understand how the system would behave it either the receiver is offline. By brief analysis it would mean that one would not be able to establish P2P connection in such scenario. However, would it be possible to device a fallback solution that could still deliver messages? If so, how the system would function?

1.4 Research Methodology

This section defines the research methodology of the thesis. This research follows the design science research approach. That is, to resolve the identified research questions an IT artifact is created and evaluated. Based on the design science research methodology the thesis is composed of following activities:

a. Design as an Artifact. The thesis presents a framework for P2P data sharing via web browsers, as the artifact.

b. Problem Relevance. The impact of ever increasing number of centralized systems on the Web cannot be understated. Especially, with the recent scandals about organizations misusing user data from social media platform [8]; data ownership and privacy concerns have become a key challenge for users and developers. In order to protect the end-user interests and to establish trust in the Web, it is necessary to design data agnostic platform that do not understand user data.

c. Demonstration. The thesis demonstrates the proposed solution as a chat application (designed as a Google Chrome browser extension). The features of the artifact are discussed as functional requirements in the later section.

d. Evaluation. The chat application artifact is evaluated using a real-world scenario. The application is evaluated by the author, by sending messages between two devices located in two different networks (in the current case, different countries.) The functional requirements of the artifact are evaluated using three test scenarios: testing P2P network, testing offline sending and testing security.

e. Research Contribution. The proposed artifact is a generic framework that enable P2P data sharing among applications via web browsers. The system also supports users' data ownership by allowing them to store shared information on their personal computer.

f. Communication of research. The solution in the thesis is available on the web and can be utilized by end-users, developers and researchers.

1.5 Functional Requirements

As stated above, this thesis was inspired from particular application, called WebWeaver platform. To model adequate solution, it's necessary to exact requirements that WebWeaver platform [9] has stated for data management and then generalize it for other applications use.

In future developer community will add additional requirements and help to generalize framework's usage with different scenarios and requirements.

Below, it will be discussed what WebWeaver application does. It is an annotation tool that enables users to leave a comment on a web page. Users can mark comment as public, private or share it with a closed group of people. Thus, WebWeaver is a social application, that needs ability to share and control data visibility.

Having restrictions on data access means that framework should be built with security and privacy in mind. Also, developers need an easy way to integrate this functionality, so framework should be simple enough to hide data management complexity from application developers.

Below are requirements that arise from particular use cases of the WebWeaver platform, but later I will show that this framework also works the same way for other applications.

1.5.1 **Private Share**

Private share is a concept where data can be viewed/modified by only single user, information does not leave user's computer and is accessible by author only.

Imagine if Michael added a private annotation on a web page on his computer, in this case, only Michael should be able to see own annotation, data should only be stored on a local device and be accessible all the time for Michael only.

1.5.2 **Public Share**

Public share is where data can be accessed by any user of the application, it means that data can be searched and downloaded and saved by any person in network.

Specific scenario is where Michael added a public annotation on a web page, that means that everybody accessing this website and using same application should be able to see this annotation. This means that framework should be able to public data and make it available with a particular key (key in this scenario is a webpage URL).

1.5.3 **Private Between friends**

Private share, means where data can be viewed only by closed set of people. Framework should handle delivery of the information to the peers from an author's device.

This is the scenario when Michael added annotation and wants to share with his friends. In this case, only friends should be able to see the annotation.

1.5.4 **Offline Peer**

Specific requirement for framework is to handle situations where peer who should receive information is not online. This should happen quite often since, not all users are online at the same time.

1.5.5 **Peer-to-peer**

Framework should provide peer-to-peer data transfer capabilities if sender and receivers are online. Image the case where Michael is sending annotation to Bob, who is also online, framework should transfer information with a peer-to-peer network. Because of that, data sharing will be instant and more secure, since the data will not be stored anywhere, except user devices.

1.5.6 **Saving the data**

The framework should allow saving annotations in local storage, so that it is always accessible even if data is gone from the network. This is the case where Michael sees Jim's annotation on Monday, but he can also see it on Tuesday, even if Jim is not in the network anymore.

1.5.7 **Security, Integrity**

While network is distributed, framework should validate data authenticity. That means that if message is created by Alice, but Bob maliciously tampered a message after Alice sent it, Michael will understand that data is invalid, because framework detected changes. Also, private annotation should not be accessible by other users. Servers in between should not be able to read and understand the data, and the data must be encrypted.

1.5.8 **Technical**

Technical requirements for the framework is that, it should work with Chrome browser web application and Chrome Browser Extension. Web application and extension can have different technical requirements for framework in order to work. Framework should be able to run in Background page, so that application can receive connections and share/receive data even if chrome extension is closed.

1.6 **Limitations**

Initial research revealed that it is impossible to establish Peer-to-peer communication between users without third party server [7]. The reason is that in real environment, most of the devices in Web are hidden behind Network Address translators (i.e. NAT). That means that not all the users in the Web have unique IP address. NAT's give user a local address, that is only unique in local network, but not unique in Wide Web, that is why NAT, needs to translate local IP into unique public IP and Port configuration for communication with outside systems.

That means that if user is connected to a network and with NAT, many different users within network will receive same public IP, but with different Ports configuration. After user is disconnected from a network, it might receive different IP and Port configuration, for that reason NATs don't allow incoming connection requests to the user.

To overcome this problem, peers should start requesting connections to each other simultaneously. In that case NATs will most likely (64% times for TCP connections) enable Peer-to-Peer connection. This technique is called Hole Punching [7].

Unfortunately, that means that in some cases it is not even possible to send data to peer without a middleware server. That case should be considered and as a fallback mechanism, relay service be allowed. That means that if Michael wants to send a data to Dwight, both peers will need to connect to the public server (that obviously is not behind the NAT) and first Michael will send data to server and server will redirect the data to Dwight.

2 Proposed Framework

In this section, solution is proposed, while in next section, implementation will be discussed based on this proposal.

2.1 Why Server?

Let's revise limitation. Initial research revealed that, it is impossible to implement server-less application, that will connect users and make it possible to transfer data. There are several reasons:

2.1.1 P2P connection establishment

To make a peer to peer connection it's necessary share peer IP addresses with one peer to another. When both peers know each other's IP addresses, they will need to start connecting with each other simultaneously, because most of the time they are behind NAT, and establishing connection when users are behind NAT, required NAT hole punching technique [7]. Symmetric NAT traversal technique, requires peers to start sending data to peer public IP & Port, for that reason it is required to have a server, that shares public IP addresses between users and helps them to establish P2P connection.
This topic has been long researched, that's why Google standardized peer-to-peer connection establishment process and created WebRTC (web real time communications) protocol [6]. WebRTC is a set of APIs that is implemented by most of the contemporary browsers. It has been first implemented in Google Chrome browser and thus it fulfils browser supporting requirement. WebRTC is just an API, for making complete solution, some signaling implementation should be used to establish WebRTC connection. For the sake of implementation, *Socket.io* [10] is proposed, because it is simple, well documented and widely popular library.

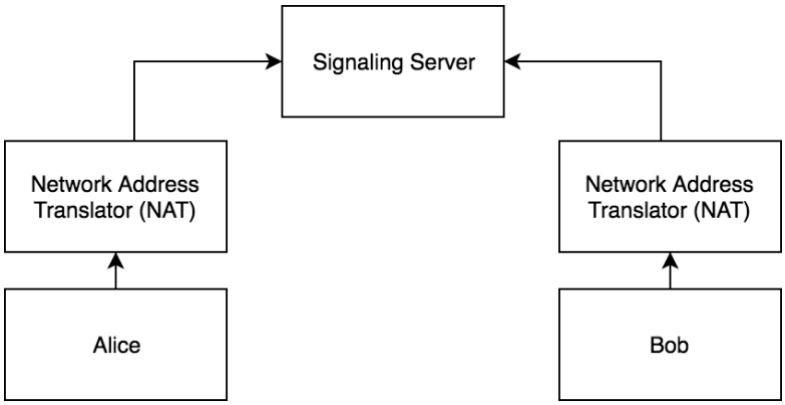

Figure 1. Signalling behind NAT

With WebRTC approach, before establishing connection, Alice and Bob will first connect to signaling server. Whenever Alice wants to connect with Bob, she will send connection request to Bob. Signaling server will deliver connection request to Bob, since Bob is connected to signaling server.

After IP addresses are shared and metadata is transferred, they can connect with each other and start sending the data.

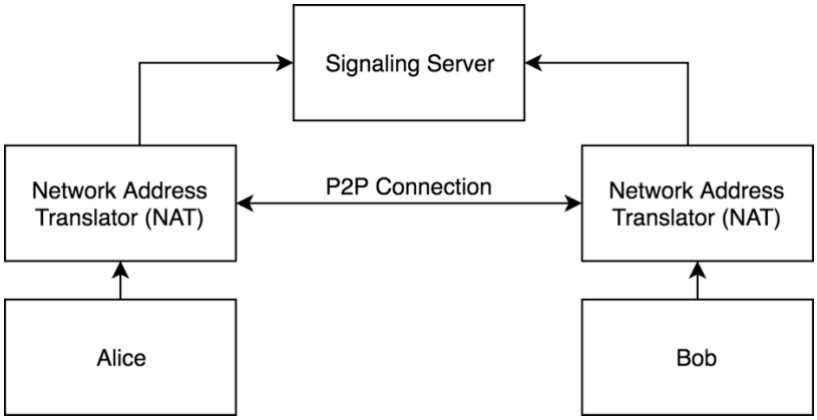

Figure 2. Established P2P Connection

2.1.2 **Public data holding**

In order to share public data, it needs to be stored on a public server, because users might not know each other and thus it would be impossible to establish P2P connection. Also asking all users for public data would be extremely inefficient.

For that reason, when Alice wants to publish data, it should be stored on a centralized server and be indexed with a known key, so that other users who might be interested with information, will request server by same key.

2.1.3 **Sharing data when peer is Offline**

Another reason for using server-side application is that users might not be online when data is being shared. Imagine scenario when Alice shares something with Bob, but Bob is not online. In this case when Bob comes online, Alice might not be online. Without server it would not be possible to shared data unless both of them are online. If we have centralized server this scenario would easily be fixed. Alice would leave a message to Bob, when Bob gets online he would check centralized server for messages addressed to Bob and download message.

2.2 **Server Architecture**

2.2.1 **Goals**

Before starting discussing server, it should be built with several goals in mind:

1. Server is not application specific.
2. Server's APIs can be implemented differently.
3. Server should not be trusted.

These goals arise from requirements. Because empowering users with freedom and data ownership, it's necessary that server should have least knowledge of application specifics. If servers, have a knowledge and understanding of the application data, then privacy and coupling problems will arise. So, to avoid these problems, ultimately, it's possible to hide application information at all. That means that server just hosts any kind of application and does not have knowledge about hosted data and its usage.

Because applications are not coupled on server implementation, it will be possible to use different servers with different implementation that support same APIs. That will give server maintainers freedom to choose how much data they allow to hold, whom they allow to communicate or if they add additional security checks.

By nature of this solution, if server does not know about data and cannot check data integrity, users are in charge of validating sender of the data and integrity. In typical client server applications, users trust server one they have a secure connection with it. That means that if server sends a message to Alice from Bob, without checking Alice will believe that information is correct. But in this case, server should not be trusted, because it does not guarantee security, because server does not understand the information and

cannot validate it, so because of that clients can themselves check the correctness of the information.

To sum up server is very generic, that can serve any type of application, without any registration needed. Server is not trusted, and it should not know anything about the data that is shared. Also, application should be ready to work in case server fails to verify sender and data integrity.

Similar idea of application was mentioned in Musubi [5] that enables developers to use common servers to share data between users. It also made sure that data is encrypted, so that users could verify sender and data integrity. But unlike our solution it was designed for mobile phones and lacked peer to peer data sharing support.

2.2.2 **Solution**

To accomplish requirements of Peer to Peer data sharing, a service should be created, that can help peers to establish connection, that service is called "Live-Rooms". Second Service that helps to communicate with offline users is called "Message Box". First service is for establishing peer to peer connections, second is just temporary store of information, like a real-life post office. Here are scenarios in order to better understand proposed architecture.

If Alice wants to send data to Bob, first she should check service called "Live-Rooms" and If Bob is online, Alice and Bob will start sending signaling metadata to each other in order to establish peer to peer connection.

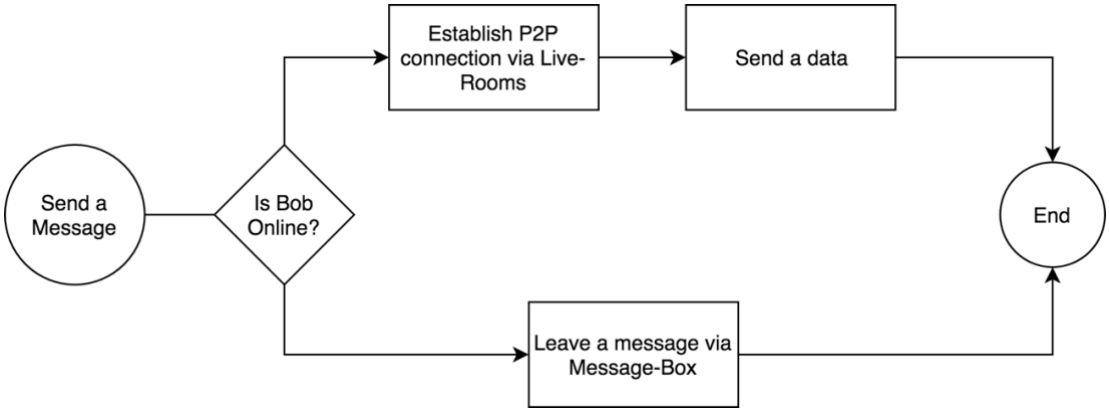

Figure 3. Message Sending Process

If Bob is not online, then Alice can leave a message in centralized storage i.e. "Message Box". Bob will synchronize "Message Box" once he comes online.

These two services are not connected together, thus application can choose different service providers, or support only one of them.

Services use unique key in order to distinguish between users.

2.3 Message Box

2.3.1 Functionality & Technology

The first step of implementing server is a message box. As explained above it's a simple database where it holds messages.
Service will have these required functionalities.

1. List Message Id's for User.
2. Download Message by message Id.
3. Save shared Message.
4. List Public Messages by key.
5. Download Public messages by key.
6. Save public Message with key.

As listed, "Message-Box" has two sets of functionalities. First three functions are for message relay, that means to send a message when another peer is not online.
Last three functions are for hosting/querying public data, that should be available for everybody. Public data should be associated with a key, that is non-unique string. How key is selected it's a decision of the application. In case of WebWeaver, the key could be a URL of the webpage. In that case if user wants to annotate webpage "*www.google.com*" User will set a key a URL, so that every other user, who visits website, could query "Message-Box" with web page URL as a key to retrieve any public data.
To model server, most commonly used client-server architecture - REST [11] is selected. Because it is most commonly used and accepted approach to model server that has a data and wants to expose to a public. Also, technologically it is possible to implement client for browser, mobile and other servers.

2.3.2 Extended functionalities

These functions are all that is required. Some servers might extend functionality and add authentication, that will force users to prove that they are indeed who they claim to be. Some servers might choose different strategies in order to evict the data.
Some might enable only chosen public keys to leave information. This might be chosen for companies that want to enable only small group of people to use their server.

Public servers will most likely implement time invalidation mechanism, where message is deleted after some time, or when there is not enough memory to save new messages. This would make sense, because this service is meant to hold data temporarily, to just synchronize messages.
Some servers might set limits for users who have too many notifications can particular user leave.
Any extension should be possible unless they provide basic minimal functionalities.

2.4 **Live-Rooms**

As Explained before, "Live-Rooms" is a place where data can be sent via P2P network. Live-Room would handle all cases (except public data) if all the users were online all the time. But because in real live people are not always on, we need to extend service with Message Box.
The idea of Live-Room is to enable fast (real-time) data sharing between users with P2P traffic. Some applications might prefer to only user Live-Room, because no third-party server will ever hold the data. But still data encryption is required, as for by nature of peer to peer connection establishment [7].
As for Live-Room, it will be place where online users gather and wait for incoming connections. Each user will connect to its own "Room" that will be associated with user id. Whenever anybody wants to connect with user, they will connect in this room and ask for connection with specified user id.
After asking, user will decide whereas they want to accept connection or not. Some applications might choose to block connections from unknown users, while some might allow. Connection blocking will be decision for application client.
After users are connected, they can send data, verify delivery and close connections.

2.5 **Client**

In order to easily use framework with applications, library client must be written, so that all the complexity of the data handling is hidden from application developers (framework users). The library should be compiled in single JavaScript file so that it can be added in html file as a single script file. In this way, developers can use the functionality of a framework with a simple API.
Client Library should not expose inner implementation to the user, but provide an abstraction to manage (save, delete, receive, query) public, private and shared data.

2.5.1 **Configurations**

Library user should be configurable in order to satisfy different kinds of application needs. Above, it is mentioned that there can be different kinds of servers that extend basic functionality and provide more security and robustness, that is why it's needed to have a way to change default configuration from the client library.
For the initial version of client library following parameters can be configured:

- "Live-Rooms" URL
- "Message-Box" URL
- "Message-Box" synchronization interval
- Local Database Name
- "Live-Rooms" WebRTC configuration that contains: STUN [6] server URLs.

All the parameters can be left as optional, because everything can be left as default.
In case of "Live-Rooms" and "Message Box" URLs, default public service URLs will be used. These default services will be maintained on Heroku Cloud Application platform [12], thus they will have strict limitations on a memory and performance.
Local Database name will be generated with default name, that will be generated based on user Id. This feature is important since, database should be changed upon user Id change. Imagine the scenario when user logs in with different user Id, in this case different database should be used, to avoid reading other users data. In case of automatic database name, application will use corresponding user's database.
"Live-Rooms" WebRTC configuration is needed to change default STUN [6] server URLs, that help to establish peer to peer connection. This might be needed to be changed in case application developer wants to use its own STUN server to improve performance and security. Otherwise application will use default STUN server maintained by Google.

2.5.2 **API/Usage**

This section describes Client Library API and explains it's behavior.
In order to start using library in a web application, developer has to import single JavaScript library script file.
Developer will be able to use those methods:

- sync()

- fetchPublicDataByKey()
- publish(key, callback)
- getByKey(key, callback)
- getByAuthor(key, callback)
- saveData(data, sharedWith, callback)
- listenDataChanges(callback)

Sync method provides a way to force synchronization between “Message Box” and application. This is useful if application needs to get fresh data and cannot wait for scheduled update interval. In Applications, this might be a case when user forces synchronization by clicking refresh button.

Fetch public data by key is useful when library user wants to read public data. The client library will try to fetch the “Message Box” for any available public data and return it to a user. Note that public data is not saved on device unless developer calls save data without any sharing option. This is for scenario when user is not sure if public data is needed. Imagine entering public channel, where you don’t wish to save data, but just view current information.

Publish is a method to make data publicly available, it needs a key in order to make data identifiable by other users. The data will be hosted by a “Message-Box” service.

Get by key will be used to retrieve data with application defined key that is a String id. Note that key is not required to be unique, so user will receive list of results in the call back, or null if nothing was found.

Get by author works the same way as get by key, but the results are queried by author id, that is a public key of the message author.

Listen Data Changes is a convenience method in order to keep application state in sync with storage. If somebody sends a message via “Live-Rooms” or “Message-Box” retrieves new messages from the cloud, users will get new data objects in callback. This should be a good place to query local database again and display fresh results if needed.

2.5.3 **Background Synchronization**

Library is designed to work both with simple Web applications and Chrome Extensions. The difference is that Chrome Extensions can store unlimited data and have the ability to synchronize data even if user has not opened application.

Chrome extensions have a notion of Background Pages, [13] that empower application developers to run scripts while page is not visible or open by users.

The library should use Background Pages in order to get “Live-Rooms” connections and share/receive data, but also to fetch new messages from the “Message Box” when scheduled time arrives.

In case of regular web applications, no background synchronization will happen, but only while application is opened in browser tab.

3 Implementations

This section will describe actual technical implementation of each project.

"Live-Rooms", "Message Box" and "Client Library" projects will be discussed. As described above, they are distinct projects, so I created 3 separate projects that don't share any code. This will make sure to reduce coupling and enable extending functionality without modifying other projects.

The implementation can be viewed at: *https://github.com/Ioane5/ms_thesis*

The main Git [14] repository project is set up with submodules: "message-box" and "live-rooms". The reason for this set up is Heroku Cloud Application [12] platform. Since Heroku requires a separate Git repository in order to enable seamless cloud deployment. That means that while writing code, with a simple Git command, code will be deployed in a Heroku cloud platform.

Also, by nature they are distinct projects and do not share the code, such modularization helps to reduce coupling between projects and enables better extensibility in the future.

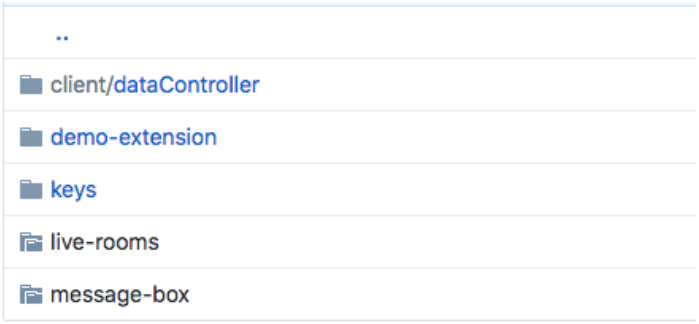

Figure 4. Repository Structure

3.1 Live-Rooms

"Live-Rooms" is built with "Socket.io" library. It was chosen because it is widely used and designed to work with web applications, therefore it has 2 separate parts: Web Client and Server Client. Here it's discussed "Live-Rooms" server part only.

"Live-Rooms" is written with JavaScript Node.js framework. While other implementations can be done with different JavaScript web frameworks, this is the easiest and most popular choice among the developers [15].

Server is implemented with "socket.io" library, instead of other popular choices like REST for this particular problem. Because server and client need to send bidirectional events. That means that client might need to send several messages and wait for an event

from a server. Particular case is when user is waiting in his "Room" for other peers to connect. In that case server might need to send connection request to a user.
For this reason, client needs to have a connection with server, so that server can notify client when other peer requests connection. Other solutions like pooling could be applied in this scenario, but the problem is that clients will have to send several signaling messages through the server, because of that managing several signaling messages with pooling approach [16] would be less efficient and complicated. Socket implementation directly solves the problem where many bidirectional messages can to be transferred.
"Socket.io" library has a notion of events that occur. The first event that happens when user connects the server is event named *"connect"*. It means that user has a ready socket that can be used to send a message or receive a message from.

3.1.1 **Message Contract**

After User is connected with server, service expects to several types of events from user. Every message sent by user has to be a JavaScript object having those attributes:

- fromPublicKey
- toPublicKey
- data

"*fromPublicKey*" means a public key of a message sender. This is required to be present in all messages to correctly identify sender.
"*toPublicKey*" is only required when message needs to be redirected to the other user.
Finally, any other attributes can be added, for example data. Note that server does not care about other attributes, so as far as client can read the data it can be encrypted for security.

3.1.2 **Events**

After user is connected, server listens to socket events.
First event expected after connection is called "*enter_my_room"*. This event fires when user is ready to give information about itself i.e. it's public key. Server joins user to the room named with user public key. "Room" is a "Socket.io" notation that enables us to label sockets. We need to label sockets in order to know which socket is corresponding to particular user. This helps to check if peer is online and redirect messages from other peers. In this simple way it is possible to have multiple parallel signaling between peers.

Imagine if Alice wants to contact Bob and Charlie first she will join room Alice and signal rooms Bob and Charlie. If users are waiting in their rooms, server will redirect Alice signaling messages to those users.

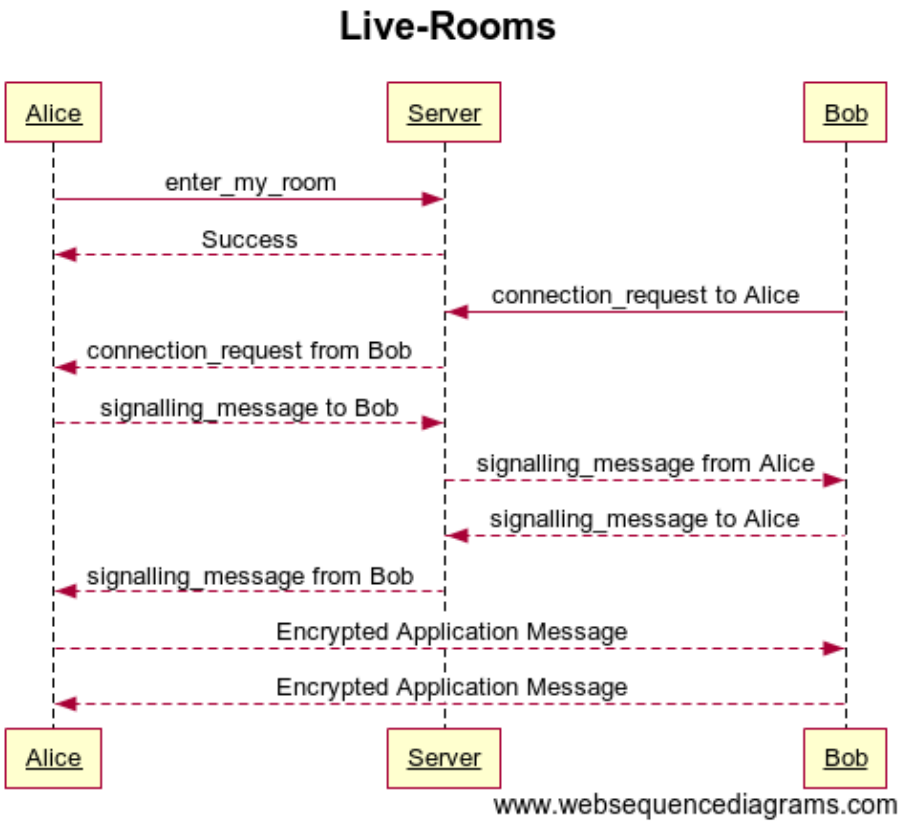

Figure 5. P2P Network Establishment

If any event fails, server usually send an event "*error*" to notify client about failed action. Server also sends a failed message, so that client can reset state and try again if needed.

Once user is entered its room, server is ready to notify peer connection requests. User also can initiate peer connections at this stage.

In order to initiate connection request, event "*connection_request*" should be sent. Note that this event also requires peer public key, so that peer can get an event called respectively "*connection_request*". At this stage peer can either accept or ignore "*connection_request*".

By accepting request Client will start sending events named "*signalling_message*". Signaling messages are redirected from one peer to another. "Live-Rooms" server does not check its content, so it's up to a client to decide how connections will be made. The process of sending WebRTC signaling messages leads to establishing P2P connections.

3.2 **Message-Box**

In this section we describe implementation details of message box. For ease of implementation JavaScript framework called Express.js [17] is used. It helps developers to quickly build a web application. It would be possible to use other web application frameworks like: Spring, Node.js, Ruby on Rails etc. but Express.js for its ease of use and easy deployment capabilities with Heroku cloud application platform [12] is the best fit. Message Box is a REST API [11] and has these HTTP methods:

- List all messages for user
- Get Message by message Id
- Save Message
- List all public messages by key
- Get Public Message by Id
- Publish a Message

Message Box Documentation is deployed on Online API documentation tool (Figure 6. Message-Box API), that provides possibility to easily describe API, add sample responses and mock the functionality.

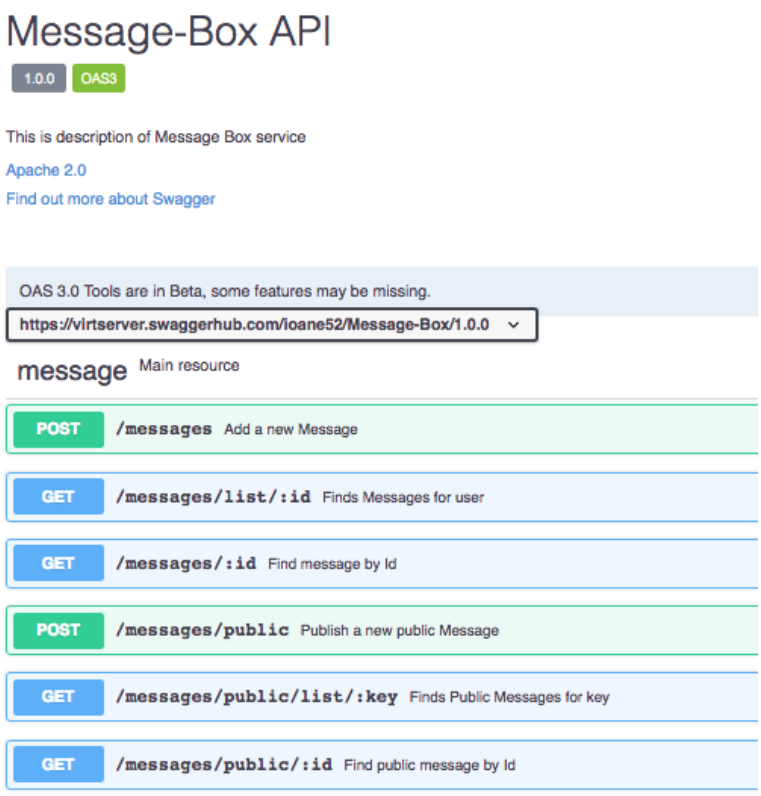

Figure 6. Message-Box API

3.2.1 **Storing Messages**

For storing messages, Mongo Database i.e. MongoDB is chosen. MongoDB is NoSQL database that has simple way of storing and retrieving documents. MongoDB Documents are lot like JavaScript objects, that is why it is very easy to connect JavaScript applications with MongoDB server.

Mongoose JavaScript library [18] is used to simplify database interactions. The message database model is defined for mongoose (Figure 7) library. With this schema, mongoose can validate JavaScript objects, cast and create data accessor functions for the database.

```
var messageSchema = mongoose.Schema({
    message: String,
    sharedWith: [String],
    key: String,
    public: Boolean,
    author: String
});
```

Figure 7. Message Database Schema

3.2.2 **Deployments**

To make services available in network, web application was deployed on Heroku [12] and separate database platform called mlab [19] was connected with "Message-Box" service. The reason is that mlab, provides a free disk space (limited to 500mb), so that it is possible to host a free version of "Message-Box" service.

To connect mlab database with Heroku, environment variables was set in Heroku dashboard (Figure 8) and different environment variables in the local repository.

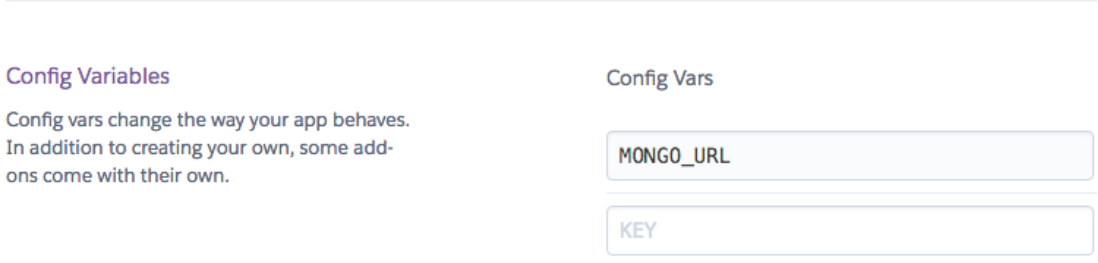

Figure 8. Heroku Database Variable

With this way it is possible to have a test database on development machine and use a production database when deployed.

3.3 **Client**

The most complicated part of the project was client implementations. Client application has to manage correct state, communicate with servers when needed and retry failed actions.

It has to run in both Chrome app and regular web application environment.

3.3.1 **DataController**

DataController is a main class that handles data. It holds state and delegates functionality to the different data Controllers, that are: Local, Live and Cloud Data Controllers.

Local Data Controller's mission is to save/query data in local storage.

Live DataController interacts with "Live-Rooms" service and can save/receive data via peer to peer connection, it hides all the logic of peer connection and data retrieval.

Cloud DataController interacts with "Message Box" respectively.

DataController unites all of those and hides functionality, with this approach, other parts of application do not need to know, how data is handled.

DataController also handles configuration of different services and passes corresponding parameters when needed.

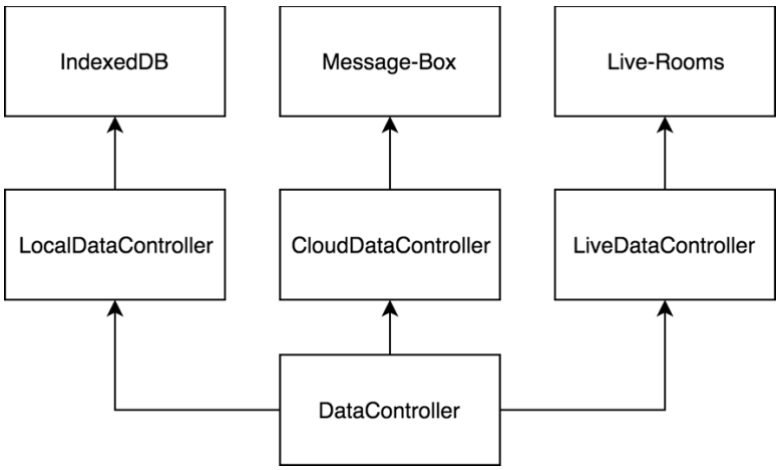

Figure 9. DataController Structure

Here is how it behaves in different stages:

- When initialization, it creates all the controllers.
- When Syncing it tries to fetch data from Cloud DataController

- “getByKey” and “getByAuthor” function delegates to Local Data Controller, since user is searching current data.
- “saveData” first saves data with Local DataController, then tries Live DataController, if peer is not online, message is saved to Cloud DataController.
- “listenDataChanges” waits for data update from Live and Cloud DataControllers.

Internally DataController handles Unique Id creation, so that library users do not need to create Ids for messages.

3.3.2 **Live DataController**

Live DataController as described above, is responsible for using “Live-Rooms” service. Upon creation, it connects user to its *room* and listens to “*connection_request*” and “*signaling_message*” events. When “*connection_request*” is received, it starts peer to peer connection establishment process.

Connection establishment process starts with gathering ICE (Interactive Connectivity Establishment) candidates. ICE candidates should be sent to remote peer using “*signaling_message*”. With ICE candidates are needed in order to perform NAT traversal [7].

By default, ICE candidates configured to be free STUN services provided by Google (Figure 10).

```
'iceServers': [{
    'urls': ['stun:stun.l.google.com:19302', 'stun:stun.2talk.co.nz:3478']
}]
```

Figure 10. WebRTC configuration

When remote peers get signaling message, it also starts to gather ICE candidates and sending them to peer using “Live-Rooms”.

After ICE candidates are shared, data channels are opened, and users can start sending data using peer-to-peer connection.

The Live DataController holds opened data channels, so that it does not have to create new connections every time user sends an information. When data channel is broken, new connection establishment process will start, if multiple errors occur, it will return null callback to the sender, so that other service can try sending information.

3.3.3 **Cloud DataController**

Cloud DataController handles “Message-Box” service interaction. It has several public functions: sync, save, publish, and fetch public by key.

When sync message is fired, it connects to message box listing endpoint, gets list of messages and downloads them one by one.

Save and publish functions on the other hand, simply send a message to the server, save needs list of public keys, that will have an access on the message. Publish will make message publicly available and associate with the key.

Fetch public by key, will search for public messages associated with key and download them.

3.3.4 **Local DataController**

For implementing Local DataController IndexedDb [20] is used. The reason for this chose is good browser compatibility and flexible API that helps to store information on local disk and ability to query using different attributes.

As name suggests, it can index data using keys to provide fast retrieval of the information. Because framework requirement is to provide two queries, by key and by author, two indexes were created. Both keys are not unique, so API returns a list of results sorted by creation date, or null if error occurs.

Before saving data, Controller checks if both key and id are present. Note that as mentioned above, Id is generated by DataController.

3.3.5 **Background**

When running in Chrome app, client has to manage state in background page. In this case application will be able to synchronize messages even when program does not run in foreground. This is important part of requirement, because clients are not always running application.

Initial implementation tried to construct “DataController” instance on background page and then access it directly from foreground Application for querying and saving. Unfortunately, this approach does not work, since Foreground and Background Pages are running in different contexts, and while it is possible to access primitive variables using “getBackgroundPage”, Chrome browser can only send JSON-serializable types between pages. Because in “DataController” we are using socket object, that cannot be properly JSON-serialized.

For this reason, it is impossible to have “DataController” in Foreground page. For overcoming this problem “DataControllerClient” and “DataControllerReceiver” were created. Those two classes reside in different pages/contexts of application. Because it’s impossible to directly interact with objects that are in Background Page, JSON messages (that contain action information) are sent to the background page, then it receives and executes actions with DataController.

To pass information from foreground page to background page, I use Chrome message passing [21], where “DataControllerClient” sends a JSON object with “action” and “params” attributes. “action” attribute points to which DataController method should be called, “params” are parameters that need to be passed to the method. “DataControllerReceiver” listens to the message on a background page, reads “action” and executes method in same context with provided “params”.

“DataControllerClient” is for application use, that’s why it should be created on front page. It has very same API as “DataController”, but the difference is that it delegates functionality to “DataController” that resides in “DataControllerReceiver”. Figure 11 shows that behavior in case “DataControllerClient” is instantiated in Chrome Application.

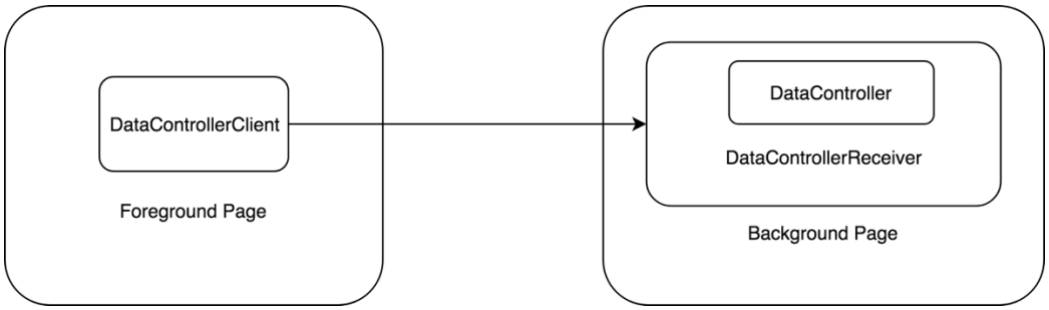

Figure 11. DataController in Chrome Extension

If “DataControllerClient” is created in regular web application, then there is no Background Page, that is why, client creates “DataController” inside class and executes actions locally.

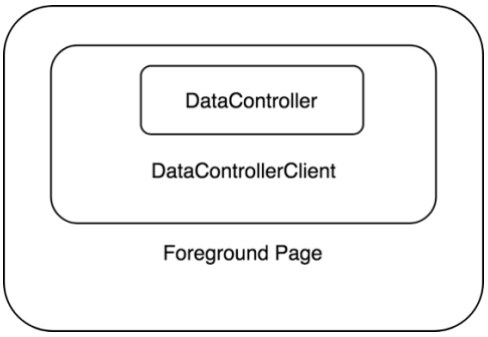

Figure 12. DataController in Web Application

For this reason, application can use “DataControllerClient” and run the same code as Chrome or Web application, “DataControllerClient” will handle both cases without changing the code.

3.3.6 **WebPack**

Because Data Controller has many classes in the project (Figure 13) it would be hard to publish client library as a multi file project. If so client would need to import multiple scripts in order. That would make library much harder to use for developers, because it’s hard to order libraries in this way.

That is why “WebPack” [22] is used, a simple library that exports multi file JavaScript project into single JavaScript file.

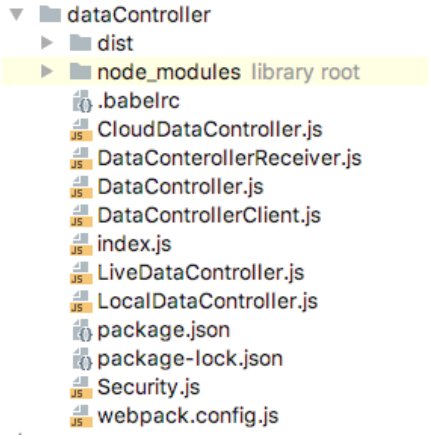

Figure 13. DataController File Structure

Now that library is packed as a single file, so user can import a single “DataController.js” file and start using library.

WebPack configuration file (Figure 14) describes how should library should be packaged and where it should output single compiled file.

```
module.exports = {
    entry: './index.js',
    output: {
        path: path.resolve(__dirname, '../demo-extension/library'),
        filename: 'dataController.js',
        library: 'DataController',
        libraryTarget: 'umd',
        umdNamedDefine: true
    },
```

Figure 14. WebPack Configuration

4 Security

Application has several services that manage user data, that is “Message-Box”, “Live-Rooms” and Client library. Client Library is sending data through P2P network or storing it in “Message-Box” to make it available for the user. Client library should treat services as untrusted, since they might be maintained by third parties.

Because of that it is vital to encrypt data and make it unavailable for adversary who might want to temper or read the information.

4.1 User Public Key

In order to maintain security, public key encryption is used. By having peer public RSA key [23], it’s possible to encrypt data for a specific user. Only user holding private RSA key, will be able to decrypt the data.

User id is treated as a public key, since it’s unique per user and it directly helps to encrypt data for a specific user id.

The limitation with public key encryption is a data size. Encrypted data cannot exceed the key length, that is why first client will encrypt data with AES-CBC encryption [24] that can encrypt unlimited size of data. After this AES-CBC key pair will be encrypted for specific user, so he/she can decrypt a key and then decrypt a whole message.

4.2 Data Encryption/Decryption

As mentioned above, data encryption itself should be done with AES-CBC encryption, with random initialization vector and key. Random Initialization vector provides with a randomness even if messages are same.

If Alice is sending Bob same message twice, the adversary will not be able to deduct that messages are same, because every message is encrypted with a different Initialization vector [24].

In order to decrypt Alice's message, Bob will need AES-CBC key pair (key and initialization vector). That is why, Alice will need to encrypt those keys with Bob's public key and provide with an encrypted message.

Before decryption, Bob will find it's keys decrypt a key pair, then decrypt a message and verify authenticity.

4.3 **Verification**

If framework do not have any kind of verification of authenticity, it will be possible to send a message in name of another peer. This could be achieved by hacking "Message-Box" service, or directly tempering traffic.

That is why public key signatures are used, so that peers can verify message senders.

If Alice wants to ensure that Bob receives authentic message, she can sign a message with her RSA private key. When receiving a message, Bob will use Alice's public Key in order to verify sender.

4.4 **Implementation**

For JavaScript cryptography, JavaScript library *Forge* [25] is used, that implements common set of tools for encryption/decryption, including key generation and signatures.

Before integrating security into application, *Security.js* class is defined. This helps to create abstract API for the application, so that it is easy to use it in different parts of the application. It is good idea to hide logic separately, so whenever change is needed, it can be done in one convenient place.

Security.js should be initialized with a client's public and private keys, in order to encrypt and decrypt the data. It also defines two public methods:

- Encrypt
- Decrypt

4.4.1 **Encrypt**

Encrypt method should receive string data that needs to be encrypted and access keys. Access keys is an array of a public keys, that can later decrypt and verify data authenticity. It should be public keys of users that author wants to share data with, also author should include his own public key, if he wants to read data later.

Before encrypting the data, SHA-1 hash [26] of a data is signed.

Hashing data is necessary in order to reduce data in size, because it is impossible to sign data that is longer then public key.

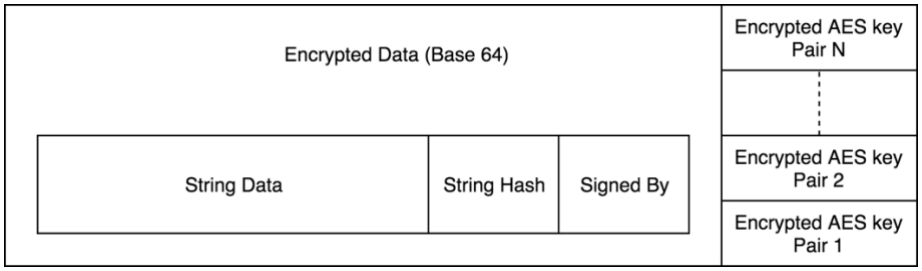

Figure 15. Encrypted Data Structure

After data is signed, public key of a signer is attached to the data.
After signature, whole JavaScript object is converted to string, and encrypted.
Forge Library encrypts String and outputs a binary data. Handling binary data is very inconvenient in JavaScript and is impossible to send via HTTP, because HTTP is a text protocol, that is why It was decided to encrypt data into Base64 encoding [27].

After this, AES key pair is converted to Base64, put into JavaScript array, and converted to string. The reason is that each AES key needs to be binary, in order to work with a crypto library [25].

```
{
  "Alice's Public Key": "Encrypted AES key with Alice's Public Key",
  "Bob's Public Key": "Encrypted AES key with Bob's Public Key",
  "John's Public Key": "Encrypted AES key with John's Public Key"
}
```

Figure 16. Access Keys

After that, stringified AES key is encrypted for each access key and put into a JavaScript object, so that peer that receives a message, can find its own AES key with its public key.

4.4.2 **Decrypt**

For decryption, the flow is reverse of encryption. First Bob should find its AES encrypted key in an access key object, by his Public key. After that Bob can decrypt AES key with its private key, since it's encrypted with RSA its public key.
After that, key should be parsed as a JavaScript object and converted to binary from Base64 string.
At this point message can be decrypted with AES-CBC algorithm. After decrypting, message will be converted to string, string converted to JavaScript object.
After having decrypted Data, validity of signer is checked, by using signature and signed by attributes. If data is not verified, method returns null. Otherwise it returns data String and signed by attribute.

4.4.3 **Application Integration**

This section describes how Security.js is used within application. Before sending a message via "LiveDataController" or "CloudDataController" message is encrypted with encrypt method. In access keys public keys of users (who can access the information) and own public key (so that later data can decrypt by sender if needed) is passed.

When application receives a data from "LiveDataController" or "CloudDataController", they both fire method "onDataReceived". This is private method that application uses to insert data in local store and fire user event listeners. For that reason, it's a right place to decrypt data and continue flow.

If data is null, then It's assumed that data did not pass validation test and it is ignored.

5 Demonstration

The idea of a data management framework came from a specific application called WebWeaver, but the framework is generic, and it can work with various type of application.

To prove that framework is generic and provides adequate solution for requirements stated in this thesis, demo application was created.

The demo application itself is a simple chat application that work on Chrome browsers as an extension (Figure 17) and web page (Figure 18). It is built with JavaScript and uses data management framework described in this Thesis.

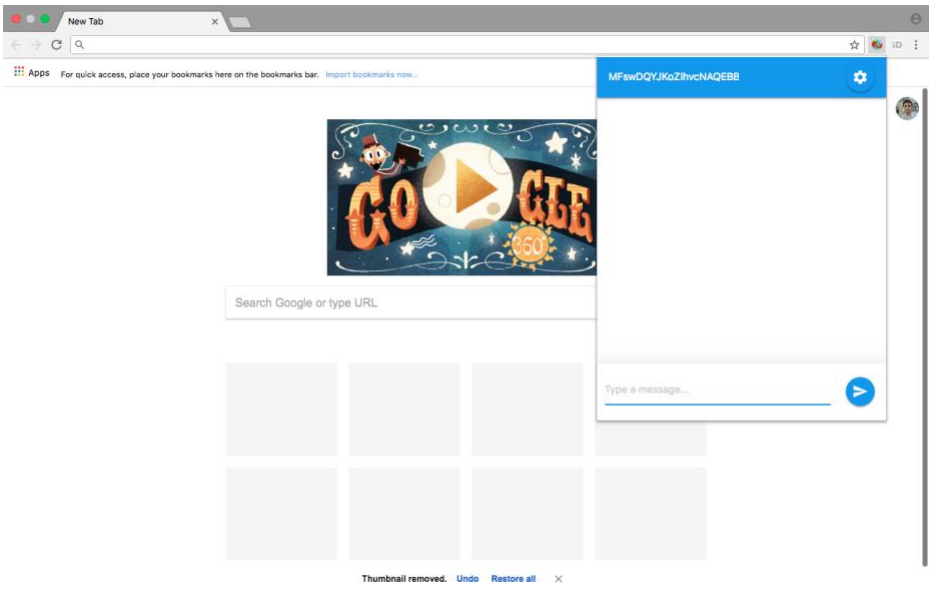

Figure 17. Demo-App Chrome Extension Mode

The application has settings screen, that can configure user details (public and private key) and a chat screen with a peer user id and message inputs.

The application uses all features of framework in order to test its correctness and usefulness.

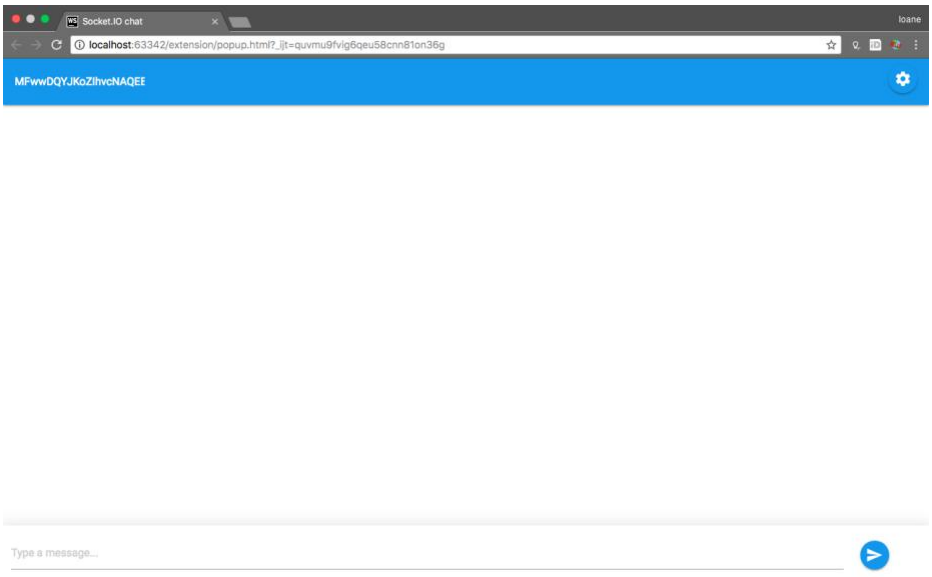

Figure 18. Demo-App Desktop Mode

The application uses default setup of the “Message-Box” and “Live-Rooms” services.

When Alice wants to send a message to Bob, then “Live-Rooms” service helps users to establish a peer to peer connection and transfer data.

When Bob is not online, Alice leaves a message on a “Message-Box”. Bob checks “Message-Box” once is online and gets Alice’s message. After reading Alice’s message from a “Message-Box” he asks service to delete message from cloud.

Application defines special public key for public messages. If one want to join public Channel, user can type channel with a lower dash “_”. When application detects that first character of public key is a lower dash, then it treats as a public key. With that it sends data not to a peer but publishes it via “Message-Box” (Figure 19).

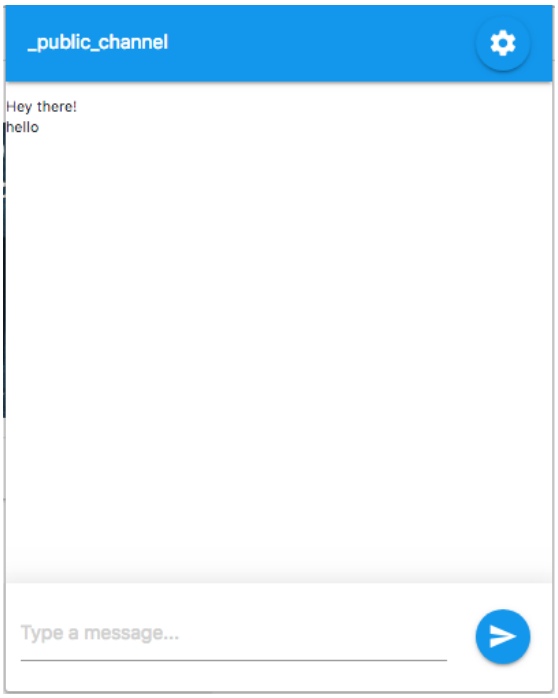

Figure 19. Demo-App Public Channel

Note that public messages are not saved locally with this application, but developer could save them via “saveData” method call and query later with public key

5.1 **Testing P2P network**

In order to verify peer to peer data transfer, it was decided to open applications that are behind different NAT servers. With that it can be proved that application handles NAT traversal correctly.

First application was opened in Estonia and checked public and private IP addresses (using online IP detection tool [28]).

Figure 20. IP address (Estonia)

Because IP addresses are different it can be assumed that local IP (192.168.0.10) is translated into public IP (82.131.13.185) address. Also it should be noted that Internet service providers might use multi-layer NAT network.

Second application is set in Georgia, with help of a remote computer control tool.

Figure 21. IP address (Georgia)

As shown in Figure 21, peer is behind completely different NAT, because it has different public IP. Local IP addresses also differ, but they could be same, because Local IP addresses are not unique.

For simplification below user in Estonia will be called Bob and user in Georgia Alice.

For testing purposes, different RSA key pairs were generated, using Mac OS terminal commands.

```
openssl genpkey -algorithm RSA -out alice_private_key.pem -pkeyopt rsa_keygen_bits:2048
openssl rsa -pubout -in alice_private_key.pem -out alice_public_key.pem

openssl genpkey -algorithm RSA -out bob_private_key.pem -pkeyopt rsa_keygen_bits:2048
openssl rsa -pubout -in bob_private_key.pem -out bob_public_key.pem
```

Figure 22. Generating RSA keys

Generated key is 2048 bits long, required length by application is more than SHA-1 length that is 160 bits.

After key pair generation settings are initialized and entered peer public Ids.

Once message is sent from Alice to Bob, application establishes P2P connection with the help of a framework and sends a message.

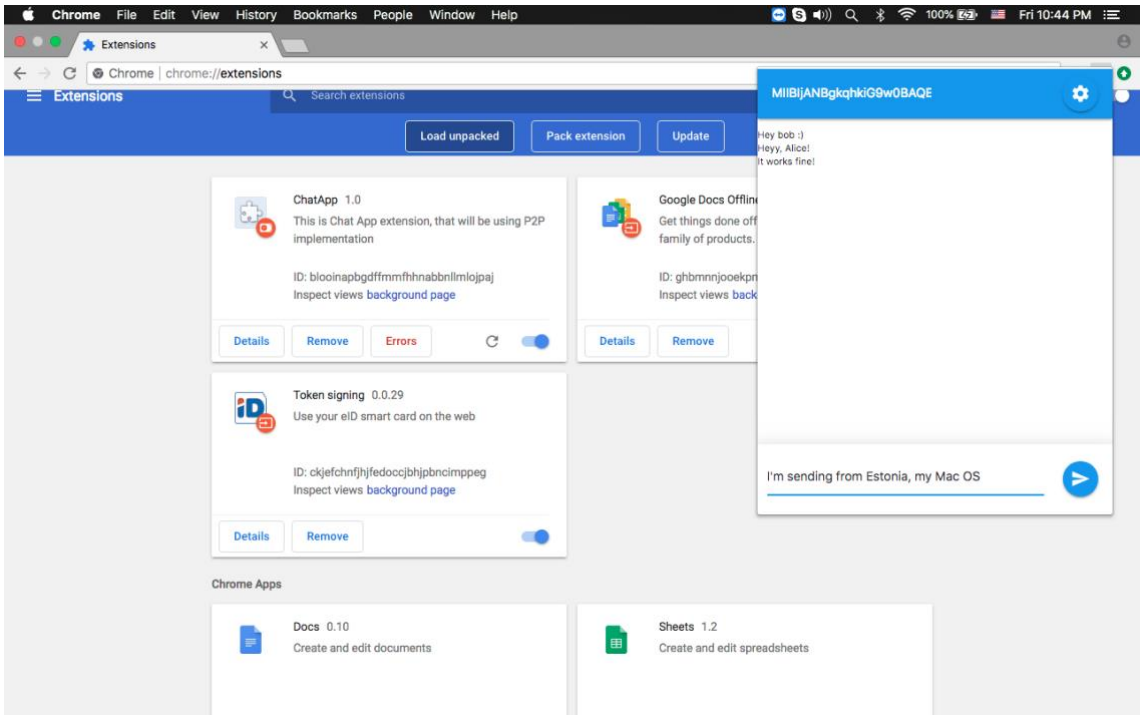

Figure 23. Sending Data (Estonia, MacOS)

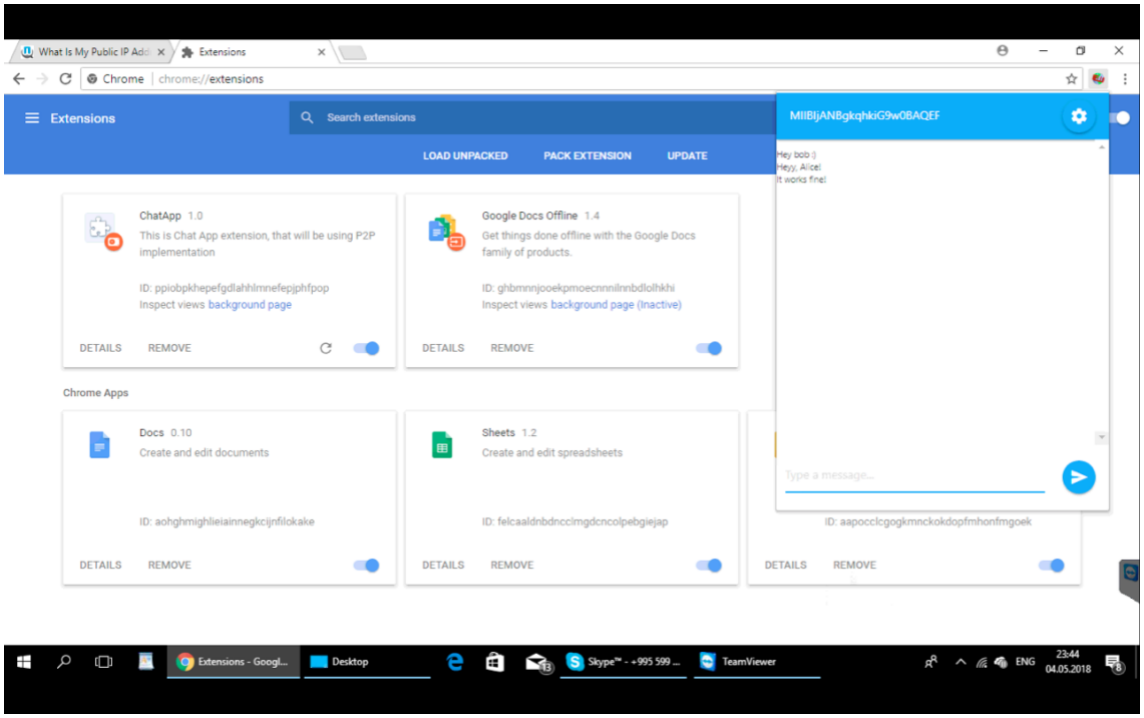

Figure 24. Receiving Data (Georgia, Windows 10)

In Figure 23, it can be seen that Bob is sending message from Estonia, while in Figure 24, it is visible that message is delivered to user residing in Georgia.

To verify that only Peer to Peer connection was used, Message Box service was closed, so that it would not be used.

Test passed, since, after sending data, it was delivered to another peer in less than 1 second.

5.2 **Test Offline Sending**

To test offline following test case is suggested. Alice is gone from a network, meanwhile Bob sends a message. If message is delivered to Alice, after re-joining the network, then test case is passed.

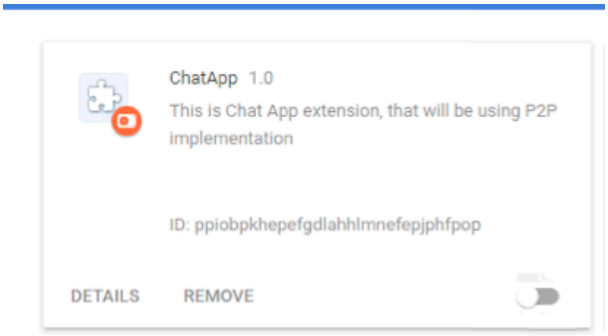

Figure 25. Disabling Chrome Extension

Using chrome extension switch, it is possible to completely disable extension. In this mode no background service will run. Note that current implementation runs background even if Chrome application is turn off, the reason is that Chrome application leaves a process that handles background services, so that just closing browser would not turn off synchronisation.

After disabling Alice and sending message from Bob's computer, Alice extension was turned back on. After a sync with "Message-Box" that happened in a minute after log-in, Alice gets a message, so that it can verified that Offline sending works correctly.

5.3 **Testing Security**

Testing security in real life is very hard, and often requires time and community support to detect vulnerabilities. But still simple tests can be done in order to check if invalid/tempered messages are still delivered.

For testing security, following test scenario was chosen. Alice leaves a message for a Bob, while Bob is being. In this case a message should be stored in "Message-Box" server temporarily, until Bob syncs and gets a message.

While Bob is offline, Alice's Message will be changed in mlab [19] database dashboard, that is "Message-Box" service data store. If Bob ignores Alice message, then test is passed.

In mlab database dashboard there is only single message from Alice to Bob, that is encrypted and therefore impossible to understand the data.

Figure 26 Tempering Message in Dashboard

Test showed that when Bob comes online, it ignores message, so test is considered as passed.

6 Conclusion

By analyzing tests, it can be assumed that the goal of the thesis is reached. The developed framework is a simple tool that can be integrated by various kinds of applications. Demonstration showed its usefulness for a chat application, where real time data sending via peer to peer network is important. The data is secure and impossible to temper with. Application can create public channels where any user can query by application defined keys. Private and group share is supported. Application demonstrates data synchronization even if peer is offline at the sending time.

The main goal of the framework was to enable developers to create applications with a powerful data ownership and privacy features in mind. With this simple framework, developers can achieve these goals.

The framework is built with multiple parts and services, that is why it can be easily extended with a custom functionality. Developers might choose to change a single or multiple part of the framework and extend functionality. Some might prefer implementing custom policies for data handling. Organizations might enforce user authentication on data sharing services, such as “Message Box” and “Live-Rooms” so that only people with a specific access can use a service.

Finally, it will require support of community in order to better understand developers need of data management. Any open source framework will need community guidance in order to evolve more useful tool for developers and users.

7 References

[1] A. M.Kaplan and M. Haenlein, “Users of the world, unite! The challenges and opportunities of Social Media,” *Business Horizons,* 2010.

[2] A. Sambra, S. Hawke, T. Berners-Lee, L. Kagal and A. Aboulnaga, “CIMBA: client-integrated microblogging architecture,” *Proceedings of the 2014 International Conference on Posters & Demonstrations Track ,* 2014.

[3] A. Sambra, A. Guy, S. Capadisli and N. Greco, “Building Decentralized Applications for the Social Web,” *Proceedings of the 25th International Conference Companion on World Wide Web,* 2016.

[4] A. Guy, R. Verborgh, C. Lange, S. Auer and T. Berners-Lee, “Decentralised Authoring, Annotations and Notifications for a Read-Write Web with dokieli,” *International Conference on Web Engineering,* 2017.

[5] B. Dodson, I. Vo, T. J. Purtell, A. Cannon and M. S. Lam, “Musubi: Disintermediated Interactive Social Feeds for Mobile Devices”.

[6] H. Alvestrand, “Google Releases WebRTC source code,” [Online]. Available: http://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html.

[7] B. Ford, P. Srisuresh and D. Kegel, “Peer-to-Peer Communication Across Network Address Translators”.

[8] “Cambridge Analytica Scandal,” [Online]. Available: https://www.theverge.com/2018/4/10/17165130/facebook-cambridge-analytica-scandal.

[9] D. Draheim, M. Felderer and V. Pekar, “Weaving Social Software Features Into Enterprise Resource Planning Systems”.

[10] “ Real-time bidirectional event-based communication.,” Socket.io, [Online]. Available: Socket.io.

[11] R. Thomas, “Representational State Transfer (REST),” 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[12] “Heroku Cloud Application Platform,” [Online]. Available: Heroku.com.

[13] “Chrome Background Pages,” [Online]. Available: https://developer.chrome.com/extensions/background_pages.

[14] “Git,” [Online]. Available: https://git-scm.com/.

[15] “Most Popular Programming Languages and Frameworks,” [Online]. Available: https://insights.stackoverflow.com/survey/2017#technology.

[16] “Polling,” [Online]. Available: https://en.wikipedia.org/wiki/Polling_(computer_science).

[17] “Express Fast, unopinionated, minimalist web framework for Node.js,” [Online]. Available: https://expressjs.com/.

[18] “Mongoose,” [Online]. Available: http://mongoosejs.com/.

[19] “Mlab,” [Online]. Available: https://mlab.com/.

[20] “IndexedDb,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API.

[21] “Chrome Message Passing,” [Online]. Available: https://developer.chrome.com/extensions/messaging.

[22] “WebPack,” [Online]. Available: https://webpack.js.org/.

[23] W. P. Wardlaw, The RSA Public Key Cryptosystem, 2000.

[24] S. Frankel, R. Glenn, NIST and S. Kelly, “The AES-CBC Cipher Algorithm and Its Use with IPsec,” 2003.

[25] “Forge - JavaScript Crypto Library,” [Online]. Available: https://github.com/digitalbazaar/forge.

[26] “SHA-1 Standard,” [Online]. Available: https://tools.ietf.org/html/rfc3174.

[27] “Base64 encoding,” [Online]. Available: https://tools.ietf.org/html/rfc4648.

[28] “Online IP detection tool,” [Online]. Available: https://www.whatismyip.com/my-ip-information/.

[29] J. Pouwelse, P. Garbacki, D. Epema and H. Sips, “The Bittorrent P2P File-Sharing System: Measurements and Analysis”.

[30] M. Moore, “The semantic web: an introduction for information professionals,” Thomson Reuters, 2011.

[31] E. Mansour, A. V. Sambra, S. Hawke, T. Berners-Lee, M. Zereba, S. Capadisli, A. Ghanem and A. Aboulnaga, “A Demonstration of the Solid Platform for Social Web Applications”.

[32] C. Holmberg, S. Hakansson and G. Eriksson, “Web Real-Time Communication Use Cases and Requirements,” March 2015. [Online]. Available: https://tools.ietf.org/html/rfc7478.

[33] “hypothes.is,” [Online]. Available: https://web.hypothes.is/.

[34] “genius.com,” [Online]. Available: https://genius.com/web-annotator.

[35] “Web ID,” [Online]. Available: http://webid.info/.

Appendix 1 – Developers Guide

This is simple guide intended for developers in order to help framework integration with a chrome extension or web application.

Setup

First developers need to download and copy minified DataController.js file inside your project. This is a single file that needs to be added in HTML header with a script tag.

```
<script src="libraries/dataController.js"></script>
```

If you wish to integrate library in a chrome extension you will need to additionally declare script in applications *manifest.json* file

```
"background": {
  "scripts": [
    "libraries/dataController.js",
    "background.js"
  ],
  "persistent": true
},
```

Make sure to add persistent flag in order to keep synchronisation even if chrome extension is closed.

```
"permissions": [
  "background",
  "storage",
  "alarms",
  "unlimitedStorage"
]
```

You need to declare several permissions in order to use functionality. If you are using background page, you will need to have background permission.

Storage permission is required in order to store local data.

User unlimited storage flag in order to enable more than 10mb storage for application. With this flag, application will be able to store as much information as user's disk space allows.

Enable alarms permission in order to allow timely background syncs with "Message-Box" service.

After configuration is done, now you need to initialize DataControllerReceiver object in *background.js* page. This is necessary in order to work in background page context.

You do not need to do this if you are implementing only web application. If your application supports both web page and extension mode, then this step is necessary.

```
let p2pControllerReceiver = new DataControllerReceiver();
```

You need to initialize DataControllerClient in your front-page script.

```
dataController = new DataControllerClient();
```

After doing this, you can check if controller is already initialized in a background page and query for its public and private keys.

```
dataControllerClient.isInitialized(function(publicKey, privateKey) {

  if(publicKey) {
    // Controller is initialzed
  } else {
    // Controller not initialized
    dataControllerClient.init("userPublicKey", "userPrivateKey");
  }
});
```

In case public Key parameter is null, then it means that controller is not yet initialized, then you should initialize DataControllerClient with user's public and private keys.

It is also correct place to initialized application specific configurations such as "Live-Rooms" service, "Message-Box" service and synchronisation interval.

You can use free "Live-Rooms" and "Message-Box" services, deployed by me.

```
'https://live-rooms.herokuapp.com/'
```

```
'https://message-box2.herokuapp.com'
```

Below you can check full API methods of DataControllerClient.

```
isInitialized(callback)
init(publicKey, privateKey, liveUrl, liveConfig, callback, cloudUrl, syncPeriodInMinutes);
saveData(data, sharedWith, callback)
publish(data, callback)
listenDataChanges(callback)
fetchPublicByKey(key, callback)
getByKey(key, callback)
getByAuthor(author, callback)
```