UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Ioane Sharvadze

Implementation of WebWeaver Platform over P2P Network

Master's Thesis (30 ECTS)

Supervisor(s): Vishwajeet Pattanaik, PhD

Tartu 2018

Title

Abstract:

In the era of Web 2.0 we are using numerous applications that store and manage user data. Switching one application with competitor seems a problem, since we are losing data. While there are numerous approaches to this problem, we propose P2P solution, that gives users real ownership of the data. The goal of this thesis is to show how we enable to have similar functionality of centralized database while storing data on user's devices.

This thesis will show solution for this particular application WebWeaver, that enables users to add a comment on a web page and share with other users, without website having this feature.

Keywords:

Social blogging, Liquid democracy, Digital divide, Peer-to-peer network

Table of Contents

1 Introduction

1.1 Goals

This thesis is about enabling Peer to Peer data sharing between users. The goal is to remove centralized server as a data store and empower users with real data ownership. Ultimately, we would like to remove server at all and make users able to communicate with each other without any middleware, but later thesis will explain that this is impossible.

This thesis will concentrate building solution for a specific application, called WebWeaver. This is a tool that enables users to weave data into specific web element without knowledge and support of the website. WebWeaver is currently being developed as Chrome Extension, that adds this feature to all websites. This thesis will create simple library, that enables application to store data on user's device and directly share it with specific users. As later described for establishing communication between users, server is necessary, but actual communication and data transfer happens on Peer to Peer network.

1.2 Importance

This section will explain the importance of this research.

There are two major problems associated with the centralized data management. Data Ownership and privacy. Whenever using any well-known applications such as Facebook, Google Plus, Twitter etc. we see that they own and manage our data and it is impossible to continue using other application without losing account data [1]. Imagine after years of using Facebook, user wanted to use another application, that would have similar features, but different interface, or provide extended/better solution. User would not be able to transfer its data to the new website.

Data privacy is another concern for users. While people send messages to each other, their data is stored and by application owner. Especially after recent user information leak [2], users start to worry about their private or limited (shared only with friends) data.

However, there are numerous proposals to tackle this problem, yet none of them are widely adopted [1].

Area of decentralized data management remains in active research. Therefore this topic should be considered as important.

1.3 Requirements

As stated above, this thesis will build data management solution for a particular application, called WebWeaver platform. To model adequate solution, we need to exact requirements that WebWeaver platform [2] has stated for data management. For that we need to understand what application does.

WebWeaver is an annotation tool that enables users to leave a comment on a web comment. They can mark comment as public, private or share it to specific users. WebWeaver uses robust algorithm to detect anchors on web pages [3]. Unlike other web annotators like *hipothes.is* [4] or *genius* [5] WebWeaver aims to improve anchoring strategies and algorithms in case of dynamic content.

To explain the anchoring problem, let's describe dynamic web page like Facebook. Once I enter the page I will see a different content then the user with different account. In this case if I annotate any element, the other user (or even me on next day) might not see the same element at all. This is still an open issue for annotators, because of its complexity. But WebWeaver tries to solve it using similarity calculation [3].

This thesis aims to provide data management mechanism for WebWeaver, so anchoring algorithms will not be discussed further. But as for requirement, proposed database solution will have to transfer data of a small size, as for anchoring algorithm metadata is relatively small (under 10 kilobytes) and average comment will require less then 10 kilobytes in size. That means that data partitioning and parallel downloads like BitTorrent [6] protocol does, will not be needed, because parallel downloads only matter when data, that needs to be transferred is large enough (much more than 1 megabyte). Establishing Peer to Peer connection is itself time consuming [7] due to the NAT traversal, that will be described below.

WebWeaver is a social application, that means that it can share data to other users, make it public or leave it as a private. That means security and privacy should be considered during development.

Below are scenarios that describe particular use cases of the data management.

1.3.1 Private Share

Imagine Michael added a private annotation on a web page on his one computer. In this case, only Michael should be able to see own annotation.

1.3.2 Public Share

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 the annotation.

1.3.3 Private Between friends

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.

1.3.4 Offline Peer

Application should be able to send annotations even if peers are offline. It should be noted that not all the peers are online at all the times.

1.3.5 Saving the data

The application should save annotations locally, so that it is always accessible when user downloads at first. 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.

1.3.6 Security, Integrity

While network is distributed, it should be possible to validate by another user that annotation that was downloaded is unequivocally created by the specified author. Also, private annotation should not be accessible from other users.

1.3.7 Technical

Technical requirements are that, library should work well with Chrome browser and should be able to be integrated in Chrome Extension, as far as WebWeaver is a Chrome Extension. It should be able to run in Background page, so that application can receive connections even if chrome extension is not opened.

1.4 Comparison & Overview

This section will now overview different approaches for decentralized data management and will show in what sense is our approach better.

1.4.1 Musubi

Mobile social application platform Musubi [8] proposes Trusted Group Communication Protocol to send encrypted data with server relay. This is very mobile application-based solution that uses server to transfer data, because some of the mobile networks (3G), do not allow incoming connections [8], so that establishing direct peer to peer connection is impossible.

Musubi users can send their data only after public key is shared among other users. Its goal is to provide a Framework to develop server less mobile apps. While it's goals seem to be similar to ours, there are subtle differences

Musubi does not support public data sharing. It only has a group sharing. Also, our thesis aims to have a possibility to transfer data without storing on server (only peer to peer network), while Musubi is always using servers for data relay.

1.4.2 Linked Data, Solid

Solid is very interesting platform that was built specifically to target data ownership. It uses Web ID to identify users. User should choose the service that will host its data. Service can be third party or self-hosted, but it should implement Solid interface to support all the features. It uses RDF-based resources to link data [1].

The reason why we don't want to go with Solid is that it still stores data on non-user device. It can be hard for user to set up service or find any free hosting service. So, it could be better if without any configuration, user could start using application, but also have its own data on a personal computer.

1.5 Limitations

Initial research showed 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 devices are hidden behind Network Address translators (i.e. NAT). They give user a temporary address (IP and Port) for communication, they only allow incoming traffic form the address that user has requested information.

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 for us that in some cases it is not even possible to send data to peer without server. That case should be always considered and as a fallback mechanism, relay service should 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 Proposal

In this section, I propose solution, while in next section, implementation will be started 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:

P2P connection establishment

To make a peer to peer connection we need a signaling server, because peers must know when another peer wants to connect. Before establishing connection, client's ports should be opened, communication metadata transferred. This topic has been long researched, that's why Google standardized peer to peer connection and created WebRTC (web real time communications) protocol [9]. WebRTC is a set of APIs that is implemented by most of the contemporary browsers. It has been implemented first in Google Chrome browser and thus it fulfils our 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, I propose to use *Socket.io* [10] because it is simple, well documented and widely popular library.

Public data holding

In order to share public data, it needs to be stored somewhere on the server, otherwise for example if user requests annotations for website it has to ask to all users. It's obvious that asking all users is not a feasible solution.

Sharing data when peer is Offline

Another reason for using server-side application is that users might not be online when data is 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

Goals

Before starting discussing server, I want to build it 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. As we want to give users freedom and data ownership, server should have least knowledge of application specifics. To support that ultimately, we can hide that 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.

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 [8] 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.

Idea

To accomplish requirements of Peer to Peer data sharing, we need to create a service 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.

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.

Our services use unique key in order to distinguish between users.

2.1 Message Box

Functionality & Technology

The first step of implementing server is a message box. As explained above it’s a simple database where it holds messages.

Server will have three 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.

So, we can see that “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 key for any public data.

To model server, I used most commonly used client-server architecture - REST [11]. 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.

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.

Basically, any extension should be possible unless they provide basic minimal functionalities.

2.2 Live Room

As Explained before, Live Room is a place where data can be sent P2P. 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 server will ever hold the data. But still data encryption is absolutely 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.

2.3 Client

In order to integrate functionality in application, library client should be written. The library should be compiled in single JavaScript file that can be added in html file as a script file. In this way, library user can use functionality with simple API.

Client Library should not expose inner implementation to the user, but provide an abstraction to save, query public and local data.

Configurations

Library user should be configurable in order to satisfy different kinds of application needs. Above, I already mentioned that there can be different kinds of servers that extend basic functionality and provide more security and robustness, that is why we need a way to change 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 [9] 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. The services will be maintained by me on Heroku Cloud Application platform [12], thus they will have strict limitations on 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 server URLs, that help to establish peer to peer connection. This might be needed to be changed in case library user wants to use its own STUN server to improve performance and security. Otherwise application will use default STUN server maintained by Google.

API/Usage

This section describes Client Library API and explains it's behavior.

In order to start using library in web application, user has to import library script file using JavaScript Script tag.

User will be able to use those methods:

- sync()
- getPublicDataByKey()
- 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.

Get public data by key is useful when library user wants to read public data. The library will try to fetch the "Message Box" for any available public data, but also will check local storage in order to provide the data.

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 call back. This should be a good place to query local database again and display fresh results if needed.

Background Sync

Library should be designed to work both with simple Web applications and Chrome Applications. The difference is that Chrome Applications 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” 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.

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 we discuss “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 [14].

I chose to implement server 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 [15] 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.

Message Contract

After User is connected with server, we listen 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.

Events

After user is connected, we listen to socket events.

First event to expect 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 check if peer is online and redirect messages from other peers. In this simple way we can 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.

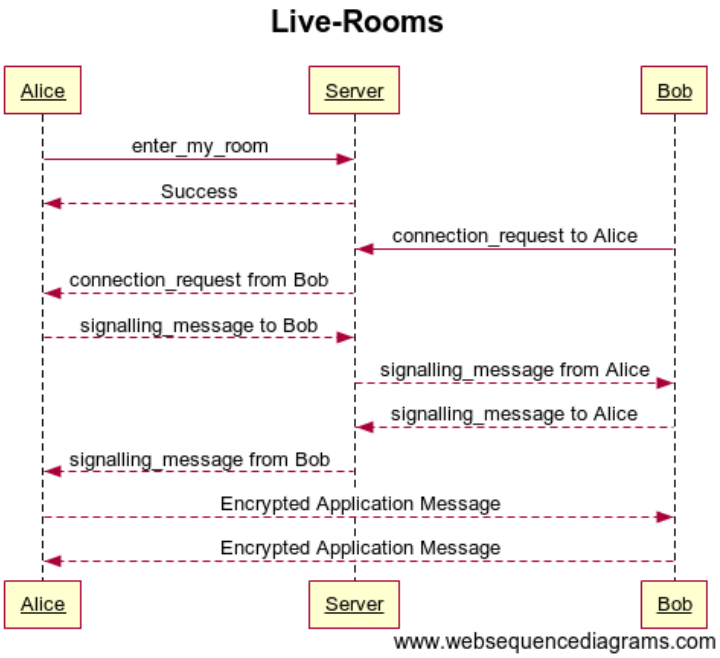


Figure 1

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. In our case (described later) signaling messages lead to establishing WebRTC connections.

3.2 Message Box

In this section we describe implementation details of message box. For ease of implementation I used JavaScript framework called Express.js [16]. 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 I chose Express.js for its ease of use and easy deployment capabilities with Heroku cloud application platform [12].

Message Box is a REST API [11] and has three http methods:

- List all messages for user
- Get Message by message Id
- Save Message

Message Box Documentation is deployed on Online API documentation tool (figure 1), that provides possibility to easily describe API, add sample responses and mock the functionality.

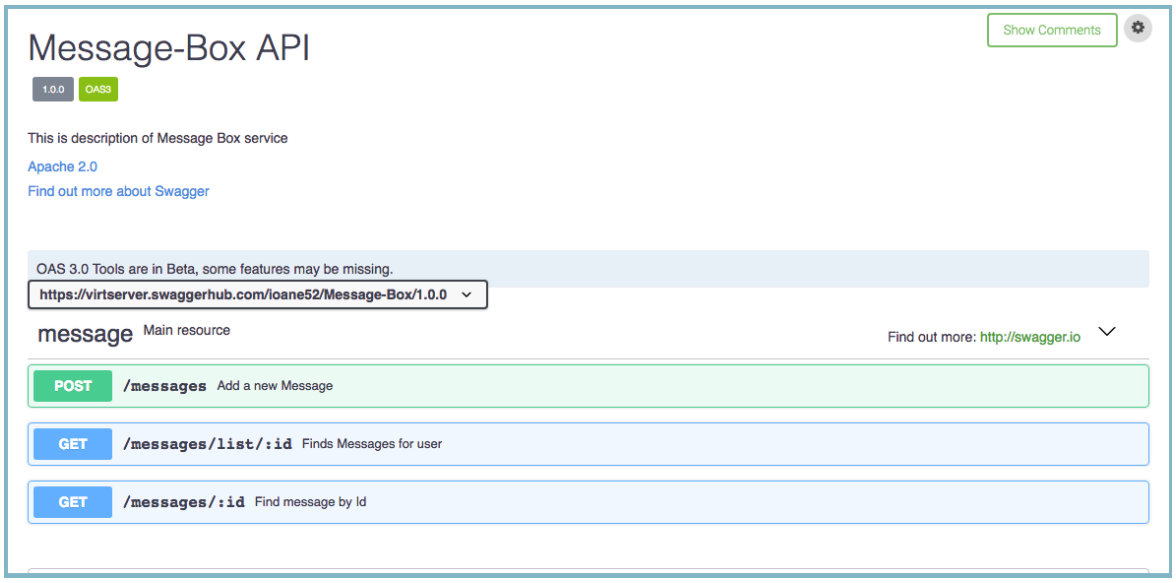


Figure 2

Storing Messages

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

I used Mongoose JavaScript library [17], that helps us to interact with database. I had to define schema from mongoose (figure 2). With this way mongoose can validate JavaScript objects, cast and create accessor functions for database.

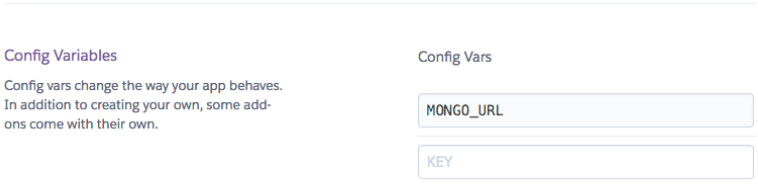
```
var messageSchema = mongoose.Schema({
    message: String,
    sharedWith: [String]
});

messageSchema.index({sharedWith: 1});
```

Figure 3

Deployments

To make everything working I deployed web application on Heroku [12] and deploy database on separate platform called mlab [18]. The reason is that mlab, provides free disk space (limited to 500mb), so that we can host a free version of our service.

To database with Heroku, I set environment variables in Heroku dashboard (figure 3) and different environment variables in local repository.



With this way I can have a test database on development environment and production database when deployed.

Figure 4

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 app environment.

DataController

DataController is a main class that handles data. It holds 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 store.

Live DataController interacts with “Live Rooms” service and can save/receive data.

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.

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.

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 4).

Figure 5

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.

Cloud DataController

Cloud DataController handles “Message-Box” service interaction. It was two public function, sync and save. When sync message is fired, it connects to message box listing endpoint, gets list of messages and downloads them one by one.

Save function on the other hand, simply sends a message to the server.

Local DataController

For implementing Local DataController I used IndexedDb [19]. The reason for this chose is good browser compatibility and flexible API that helps to store information on local disk and query using different attributes.

As name suggests, it can index data using keys to provide fast retrieval of the information. Because our requirement is to provide two queries, by Key and by Author, I created two indexes respectively. 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.

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 we can access primitive variables using “getBackgroundPage”, Chrome can only send JSON-serializable types between pages. Because in Data Controller we are using sockets, it cannot be properly JSON-serialized.

For this reason, “DataControllerClient” and “DataControllerReceiver” were created. Those two classes reside in different pages/processes of application. Because we cannot directly interact with objects that are in Background Page, we send JSON messages 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 [20], 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, reads “action” and executes method 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 4 describes that behavior in case “DataControllerClient” is instantiated in Chrome Application.

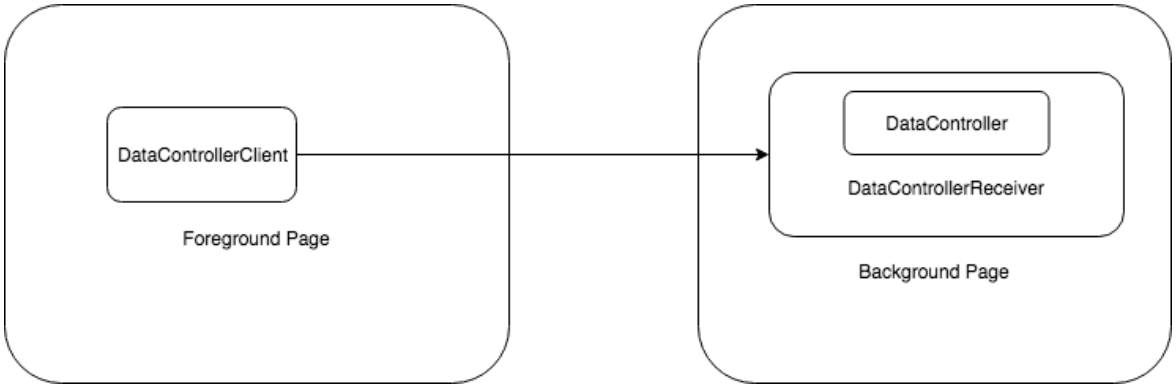


Figure 6

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.

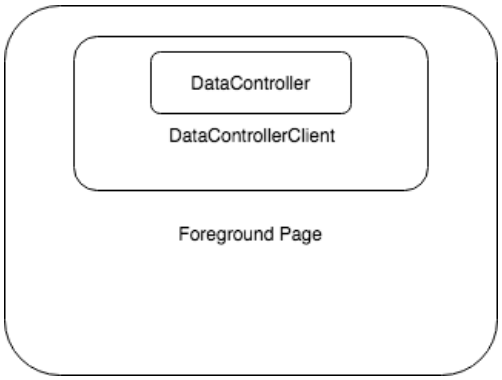


Figure 7

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.

Web Pack

Because I Data Controller has many classes in the project (figure 6) it would be hard to publish client library as a multi file project, because client would need to import scripts in order. That would make library much harder to use, because it’s hard to order libraries in this way.

That is why I’m using “WebPack” [19], a simple library that exports multi file project into single file.

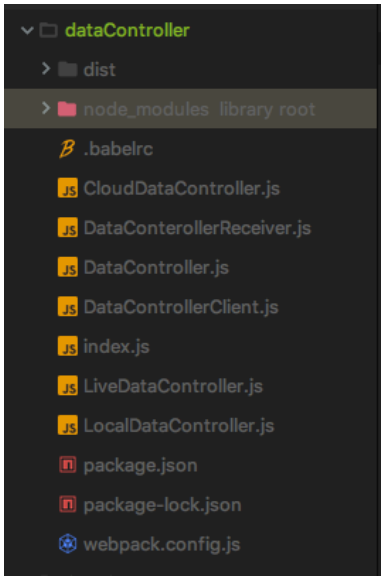


Figure 8

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

4 Sample Project

Sample project was done in order to show the usability of the Library.

5 Security

6 Outcome & Conclusion

[9]

7 References

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

[2] D. Draheim, M. Felderer ja V. Pekar, „Weaving Social Software Features Into Enterprise Resource Planning Systems“.

[3] „Making Web Annotations Dynamically Robust and Semantically Rich“.

[4] „hypothes.is,“ [Võrgumaterjal]. Available: https://web.hypothes.is/.

[5] „genius.com,“ [Võrgumaterjal]. Available: https://genius.com/web-annotator.

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

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

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

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

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

Appendix

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Ioane Sharvadze**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Implementation of WebWeaver Platform over P2P Network,

(title of thesis)

supervised by ,

(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **27.04.2018**