TAMPEREEN TEKNILLINEN YLIOPISTO

JUHO JOKELAINEN

LIQUID SOFTWARE USING WEBRTC

Master of Science Thesis

Examiner: Prof. Kari Systä
Examiner and topic approved by the Council of the Faculty of Computing and Electrical Engineering on 3rd of June 2015

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
JUHO JOKELAINEN: Liquid Software using WebRTC
Master of Science Thesis, 52 pages, 6 Appendix pages
November 2018
Major: Computer Science
Examiner: Prof. Kari Systä
Keywords: liquid software, application transfer, WebRTC, P2P

Users of today's applications have more and more diverse computing devices. Previously a person might have had a single desktop or a laptop, but now they not only have those, but a couple of tablets and a smart phone or a smart watch. Even the smart refrigerator is not only a joke anymore. With the increasing number of devices users have, keeping all the important applications available all the time is becoming a chore.

Liquid software tries to solve this problem. It is a vision of applications that "flow" from one device to another with minimal effort from the user, while leaving them in charge of all their data, applications and devices. In the utopian world of liquid software all the devices a user ever uses form a hive mind of applications and data, always ready for use.

In this thesis, a framework that helps developers create liquid software is implemented. To provide the communication between devices, the framework uses WebRTC. What makes the framework special is that it is the first piece of software to apply peer-to-peer networking to liquid software. This approach gives the developer more tools when designing how the liquid applications move and synchronize between the devices. The resulting *LiquidRTC* framework is able to transform simple single-page applications into liquid software and transfer and sync those between any device that has a modern web browser installed.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
JUHO JOKELAINEN: Liukkaat ohjelmistot käyttäen WebRTC:tä
Diplomityö, 52 sivua, 6 liitesivua
Marraskuu 2018
Pääaine: Ohjelmistotiede
Tarkastaja: Prof. Kari Systä
Avainsanat: liukkaat ohjelmistot, ohjelman siirtäminen, WebRTC, P2P

Ihmisillä on nykyään yhä useampia ja monipuolisempia tietoteknisiä laitteita. Aiemmin henkilöllä saattoi olla työpöytä tai kannettava tietokone, mutta nykyään näiden lisäksi häneltä saattaa löytyä myös muutama tablettitietokone ja älypuhelin tai -kello. Edes älyjääkaappi ei ole enää pelkkä vitsi. Jatkuvasti lisääntyvien laitteiden keskellä kaikkien tarvittavien ohjelmistojen pitäminen mukana voi käydä työlääksi.

Liukkaat ohjelmistot pyrkivät ratkaisemaan tämän ongelman. Termin takana on visio ohjelmista, jotka "valuvat" laitteelta toiselle mahdollisimman vähäisellä käyttäjän vaivannäöllä, samalla antaen hänen määrätä niin ohjelmistaan, datastaan kuin laitteistaankin. Liukkauden ohjelmistojen utopistisessa maailmassa kaikki henkilön koskaan käyttämät laitteet muodostavat yhden kokonaisuuden, jonka mukana kaikki tarvittavat ohjelmat ja data kulkevat.

Tässä diplomityössä kehitetään ohjelmistokehys helpottamaan liukkaiden ohjelmistojen rakentamista. Tämä ohjelmistokehys käyttää WebRTC:tä tarjotakseen kommunikaatiokanavan laitteiden välille. Kehitetyn ohjelmistokehyksen erityispiirre on se, että kyseessä on ensimmäinen järjestelmä, joka yhdistää vertaisverkkokommunikaation liukkaisiin ohjelmistoihin. Näin ohjelmiston kehittäjälle tarjotaan laajemmat työkalut suunnitella kuinka liukkaat ohjelmistot liikkuvat ja synkronoivat tilaansa laitteiden välillä. Työn tuloksena syntyvällä *LiquidRTC*–ohjelmistokehyksellä voi vähäisellä työllä muokata yksinkertaisesta yhden sivun verkko-ohjelmistoista liukkaita ohjelmistoja ja siirtää niitä ja niiden tilaa minkä tahansa modernin selaimen sisältävien laitteiden välillä.

PREFACE

This thesis was done as part of the research of liquid software at the laboratory of Pervasive Computing at Tampere University of Technology.

First of all I want to express my gratitude to my supervisor Professor Kari Systä for his excellent feedback and suggestions, as well as perseverance with what seemed like an endless task. I would also want to thank M.Sc Anna-Liisa Mattila for filling me in with the background, terminology and ideas behind liquid software, as well as the recommending me for this work. A special thank you goes to M.Sc Jari Voutilainen for diving back into his old research just to help me with my thesis. Without him I would not have had anything to compare my work against.

The work was finished while working at Vincit Oy. I could not have wished a better place to finish my thesis. Vincit has provided me with a very flexible schedule and allowing me to even leave a project with a looming deadline to finish my thesis in time.

Of my colleagues there I wish to particularly thank M.Sc Juha Simola for his time going through my work twice a month, providing feedback and more importantly forcing me to continuosly work.

I also want to thank Riikka for supporting me through all of this. It's been a stressful last few months, and yet you not only put up with my quirks, but also helped with the writing process.

Most importantly I wish to thank my grandma: Kiitos mummu että jaksoit uskoa minuun ja työhöni. Anteeksi että tässä kesti näin kauan. Tämä työ on omistettu sinulle.

Tampere, November 19th, 2018

Juho Jokelainen

TABLE OF CONTENTS

LIST OF FIGURES

LISTINGS

1. INTRODUCTION

Liquid software, as defined in Liquid Software Manifesto [1], is a design principle where "Multiple device ownership should be as casual, fluid and hassle-free as possible." The long term vision is to have any number of devices of different form factors and user interfaces sharing applications and their state seamlessly, with minimal effort from the user.

In the last ten years, companies have started moving in on this goal from multiple fronts. Google Docs [2] have shared their state between multiple computers and Apple has launched projects such as Continuity [3], which allows their laptop to use a nearby phone to e.g. make phone calls. The new smart watches are taking this diversity of co-operating devices even further.

While these are steps toward liquid software, they all still have shortcomings. Google Docs is a split group of specialized software, most of which have been developed the desktop in mind, leaving other devices mostly to just play the part of a passive viewer. Baton [4] and Continuity, on the other hand, include software that handle differing user interfaces better, but are even more restrictive on the ecosystem than Google. The likely reason behind this is that these applications are built by large companies, where vendor lock-in is a thing to strive for.

WebRTC [5] is a new feature added to many modern browsers. WebRTC allows two browsers to directly communicate without the need for a server in between. It has been used to create video conferencing applications, file sharing sites and even multiplayer games.

The goal of this thesis is to test the feasibility of using WebRTC as a transfer layer for liquid software and to find out if there are any considerable advantages or disadvantages of this approach compared to other possible solutions. Probably the most interesting feature of the choice of WebRTC is that it is the first time a liquid software uses peer-to-peer networking. To analyze the WebRTC–based approach, a concrete proof-of-concept was

implemented. An example application is also developed and subsequently liquified using the newly developed framework.

The liquified version of the application is tested and by analyzing it the capabilities of the framework are assessed. The test application gets also liquified with another library and the differences of the framework and library are compared. Being the first liquid software system with peer-to-peer capabilities, the strengths and weaknesses of that are estimated.

The following chapters first take a look at both liquid software in chapter 2 and the browser in chapter 3 to provide some context for this work. Then, the chapter 4 gives an introduction to WebRTC, what it is and how does it work. Chapter 5 describes a program that will be used to test how well the framework liquifies applications. The actual LiquidRTC framework implementation is provided in chapter 6 and it is put to test against the test application in chapter 7.

In chapter 8 the LiquidRTC framework is reflected against another liquid software library and their differing standpoints are discussed. Considering the limitations that have come up, chapter 9 lists different aspects that should be improved upon in the the future frameworks. Finally the chapter 10 outlines the lessons learned in this project.

2. LIQUID SOFTWARE

Liquid software does not have an exact definition and it is up for debate if a piece of software is liquid or not. A simplistic description could be "Software that is so effortlessly available on multiple devices, the users do not have to think through which device they are using the software." However, many facets of liquid software have been defined. This chapter discusses the history of the term, the development around it and different aspects that area considered to lead to liquid software.

2.1. History

The term “liquid software” first appeared in “Liquid Software: A New Paradigm for Networked Solutions” [6] back in 1996. The authors noted that the then still young World Wide Web had revolutionized accessing data and their goal was to do the same for functionality—software whose physical location is irrelevant to the end user who needs to use the functionality.

> *Thus, while location-independent data access is a wonderful thing, it is only the tip of the iceberg of possibilities opened by the Web. One can imagine location-independent (mobile) code— code that is not tied to any particular location in the Web, and whose actual location is not a concern of the users. [6]*

As an example of liquid software, the authors built a search tool that could be run on one machine that pushes the actual code to other devices in the network, runs the search on those machines, and finally returns the results back to the device the user operates. The machines had to run a special service that receive program code from the network, compile and run it and communicate its status back.

In 2014, Antero Taivalsaari, Tommi Mikkonen and Kari Systä breathed new life into this almost forgotten term, by making a list of requirements for liquid software in "Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture" [1]:

1. *In a truly liquid multi-device computing environment, the users shall be able to effortlessly roam between all the computing devices that they have.*
2. *Roaming between multiple devices shall be as casual, fluid and hassle-free as possible; all the aspects related to device maintenance and device management shall be minimized or hidden from the users.*
3. *The user's applications and data shall be synchronized transparently between all the computing devices that the user has, insofar as the application and data make sense for each device.*
4. *Whenever applicable, roaming between multiple devices shall include the transportation / synchronization of the full state of each application, so that the users can seamlessly continue their previous activities on any device.*
5. *Roaming between multiple devices shall not be limited to devices from a single vendor ecosystem only; ideally, any device from any vendor should be able to run liquid software, assuming the device has a large enough screen, suitable input mechanisms, and adequate computing power, connectivity mechanisms and storage capacity.*
6. *The user shall remain in full control regarding the liquidity of applications and data. If the user wishes certain functionality or data to be accessible only on a single device, the user shall be able to define this in a simple, intuitive fashion.*

2.2. Example

Liquid software is—in its core—about the user experience. It is programs that are always with the user in all of their devices, so that they never need to think about it. It is like the devices form a hive mind. To give an idea what a liquid application would seem like, here is an example:

Consider Alice. Alice is an artist. She makes music, does digital paintings among other things. One day she is in a park, drawing on her tablet, when suddenly, that she realizes she is going to be late from her band rehearsal. She hurriedly packs her stuff and jumps in her car. On the way an inspiration hits and she comes up with a great melody, but her phone and tablet are somewhere in the back seat in the backpack. Fortunately, her car also incorporates a computer in the form of the central console. While waiting at the red lights, she fires up the recording application she always uses and hums the melody to the hands free microphone in the car.

She arrives at the location, grabs her backpack and runs to meet the rest of the band. She grabs her phone, starts the same app as in the car, and is immediately able to play back her recording to her friends. They find the melody catchy. Bob, another member of the band suggests an application he has on his laptop for recording the session. He sends the application to Alice's phone and the microphones in the two devices are used for recording.

After the training session she rides home, she slumps onto her chair next to her computer, where the draft she drew in the park as well as the new recording application with the actual recording are waiting for her.

2.3. Aspects of Liquid Software

Architecting liquid software [7] presents an excellent and thorough view of the broad scope of what features comprise liquid software. Below is a brief summary as well as some additions of my own of the aspects I consider meaningful to the work:

Liquid software may be used *sequentially*, meaning only a single device is actively being used to interact with the application; or *simultaneously* where a user may, for instance, use one device as an input device and another for its computing power or as a display. In either case, there may be just one user, or multiple.

In the example story, Alice drawing on her tablet and finishing the work on her desktop is a single user sequential use case, whereas recording of the training session is collaborative simultaneous example.

Granularity, in the context of liquid software, means the level on which the software running on different devices are liquid.

- On the top level is *operating system (OS) level*, in which the complete Operating system is shared between devices.
- A step down is *virtualization level*, which covers setups in which a virtualized machine is the unit that is transferred or synchronized. This kind of setup is used in data centers, but for reasons that do not match the goals of liquid software.
- The most intuitive one works at *application level*, where each application is its own complete unit.
- In *component level* granularity smaller parts of an application are transferred or synchronized independently.

The Liquid Software Manifesto [1] did not make a distinction between *application transfer granularity* and *synchronization granularity*, but I believe it's an important to clarify these two need not to be the same. As an example, it might make sense to clone a virtual machine as a whole, but only synchronize some of the applications.

To be able to work together, two devices need some way of communicating. Before they can communicate, they need a way to *discover* and connect to each other. This discovery can be done in many ways: straight via Bluetooth or a Local Area Network or through a third party, such as a web server which the clients can connect to at a shared URL.

Being able to find other devices supporting liquid software is worth nothing, if the applications themselves don't support any sort of *Liquid User Experience* (LUE). That is to say they would not be able to synchronize anything. The original paper considers *primitives*—the way the application instance move from once device to another:

- *Forwarding*: Using one device as an UI for the application running on another device.

- *Migration*: Moving the running application instance from one device to another.
- *Forking*: Copying the running application instance from one device to another, leaving two identical copies.
- *Cloning*: Copying the running application instance from one device to another and starting an automated synchronization.

In addition to these, I present another property to consider: what causes the Liquid User Experience to begin: the *initialization trigger*.

- *Push*: User manually selects the target devices and the primitive, with which they wish to initialize LUE.
- *Pull*: User manually selects the source device, application, and the primitive, with which they wish to initialize LUE.
- *Presence*: Allowing two devices to *discover* each other automatically triggers LUE.

Topology is another important design choice for a liquid software network. First, there is the actual network level topology, which describes how clients are connected to each other. This is a question that is covered in more detail in the section 2.5.

Next there is the question of where the applications are stored—the *application source topology*. The system can be built using a *single repository* server, that is the only source of applications for all the potential clients. Another option is to use *multiple repositories*, maybe sharing some of the applications, but some are unique to that repository. Also *client repositories* are an option, forgoing an authoritarian server altogether leaving the clients responsible for providing the applications to run.

The last topology question is that of the state synchronization. The options are similar to those of application source topology: *Single master* (originally *Master-slave*) where a single server has the authority over the whole state and each client trying to change it has to request the server to do so. The server is then responsible for relaying the state change to other clients. The opposite of this is *Multiple masters*, where the application state is decentralized and the different masters (often every client is a master) need to cooperatively decide the correct state. One example of an algorithm solving this complex problem is presented in [8].

A question relating to the state synchronization is that of *partitioning* or *layering*: How is the workload of the application divided between a client and a server. It is not a clear-cut divide, but a gradient from a *thin client* to a *thick client*. On the thin client end there are clients that might only work as a user input device that only sends the user inputs to a server with no additional computation. The server then may instruct the client to change the interface according to the state change in the server end. One such example might be a TV-remote app on a mobile phone.

On the other end of the spectrum, the thick client might do all the work, and only send the necessary state change data to the server. The graphical application Alice used would likely be a thick client, because if the user inputs would be handled on a server and the updated images then sent back to her device to show, the latency would make the application very unpleasant to use.

Since liquid software has to work on a plethora of platforms while retaining its usefulness, its UI has to adapt to different devices. A smart watch has a very different user interface than a desktop PC. Of course, some applications cannot be used with such a limited input capabilities, and sometimes it makes sense to also adapt the functionality of the application to these limitations, or simply not support a device. An example might be an email application that does everything when run on a phone, but on a smart watch only allows reading (and maybe transcribing dictated) emails.

If this concept is taken far enough, it might be argued that the application shared between the devices is not the same any more. If a full blown video player on a TV, shared to a smart watch is only a remote to control the playback, it would make sense to not share the complete application to save resources on the less capable device. I call this asymmetric liquid software, in contrast to the more straightforward symmetric liquid software. It's important to note that a symmetric liquid software can have distinct look and differing functionality, if the application is identical on both devices, but only the UI adapts to the device.

Probably the most dreaded aspect of liquid software is its *security*. A malicious actor who is able to connect to a liquid software can cause havoc by sending arbitrary states to the users. If he is able to send new applications to the users' devices, the situation becomes catastrophic. The only proposed solution seems to be to make sure unauthorized

users cannot connect to the liquid software. In the example story Bob was able to just force new software on Alice's phone and record audio through it. In the real world there is a need for a system to control such access.

2.4. Compared to Cloud software

If the example with Alice and Bob seemed far fetched and the use cases specialized, the reason was to find liquid software functionality that has not yet been developed using cloud infrastructure.

Applications such as the Google Docs [2] suite can already be seen to accomplish most of the requirements set for liquid software: they are discoverable via shared URL, provide sequential and simultaneous use on multiple devices and they provide seamless state synchronization via single master. Moreover, transitioning from one device to another requires no extra steps from the user, and the applications' user interfaces adapt well to the devices' ecosystem variances.

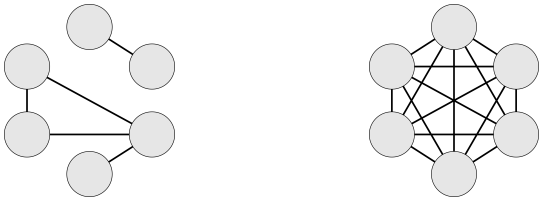
Google Docs is, however, hardly a unique example. One, maybe a little less known liquid software system is Apple's Continuity [3] that supports a set of preinstalled applications on different Apple devices. With a feature called Handoff these applications can have their states synchronized between the user's devices.

A slightly different application that has liquid software –like features is Spotify [9]. Like the other examples, it supports multiple different kinds of devices and allows sequential use. The extra feature that makes Spotify an interesting example is that it supports forwarding: One device can be used to select songs, play and pause or control the volume, while another device is used to play back the music.

What all cloud based solutions have common is that the discovery happens over the internet, often through a shared URL through which the LUE initialization then happens. This also means that in the users perspective there is a single software repository and a single master that controls the state.

However, most cloud based software falls somewhat short on the requirements set by the liquid software Manifesto [1]. Often the problem is with point 6: The application does not allow the user to control their data, but rather sends all the data to the servers without an explicit request from the user. Another is point 5, as vendors try to lock users into

(a) Star **(b)** Mesh



(c) Disjoint **(d)** Complete

Figure 2.1. Different network topologies

their device ecosystem. Nevertheless, these cloud applications are often referred as liquid software in literature, so it seems that the listing is not a strict requirement. [10]

Liquid software is a broader concept compared to cloud software; for example liquid software does not require a server or even network connectivity, as long as it is able to somehow communicate with other devices.

2.5. Topology

All previous instances of liquid software have been using a server-centric approach for connections, meaning there has been a single master node in the network that is used to transmit data between the clients. The design presented in this thesis is the first to look at peer-to-peer liquid software, which makes it possible to freely choose the way different clients are connected to each other. These different ways of arranging connections are called topologies and all of these have their own merits and drawbacks [11]. The topologies that are most interesting from the standpoint of this work are shown in figure 2.1.

A Star network (figure 2.1a), which is what client–server systems use, is simple to set up and use to route traffic, but if the one peer that is used as the central point is lost, the whole network breaks down. In the client–server model this central node is the server and it is expected to stay available, so for that case this risk is minimal, but in a peer-to-peer network the nodes cannot be trusted to stay connected.

The other extremity of the spectrum of the topologies is a complete (fully connected,

N-to-N) network (figure 2.1d), where each peer has a direct connection to all other peers. Because of this property it has no need for routing traffic; and if any peer disconnects, the rest of the network is still forms a complete topology network. The downside of this topology is that it is resource heavy. Each connection itself requires resources, and if a peer has to send data to the network, it needs to duplicate the same data transfer for all of the other peers.

In between these two, there are mesh networks. In a mesh topology network, peers form routes dynamically through each other and try to span to all the clients. The routes may change during the lifetime of the network if the peers disconnect or a specific connection between peers is severed. [12]

A mesh network may be unable to keep all peers connected, for example, when one peer disconnects or loses connection to another peer, the network may break into parts. An interesting case in the context of this work is a peer that moves between two otherwise distinct networks, moving information between them.

3. BROWSER AS A PLATFORM

With ever increasing capabilities of a browser, it can run many of the everyday applications people use. One of the more prominent milestones in the transition of applications to the Web was probably Google Spreadsheets (first part of what became an office suite called Google Docs) in 2006. The success of this family of applications encouraged others to follow and the Web hasn't been the same since. Today there exists SketchUp for 3d modelling, WeVideo for non-linear video editing, Overleaf [13] for LaTeX typesetting, games developed with Unity [14] can be compiled for browser and the list goes on. Google even experimented with browser-only OS with its Chrome OS [15], which left no doubt about it: browser is a very capable platform.

This chapter provides the background on how the Web has developed to lay the groundwork before moving on to describing how WebRTC changes things in the next chapter. It also describes the reasons behind some of the limitations of the framework implemented in this thesis.

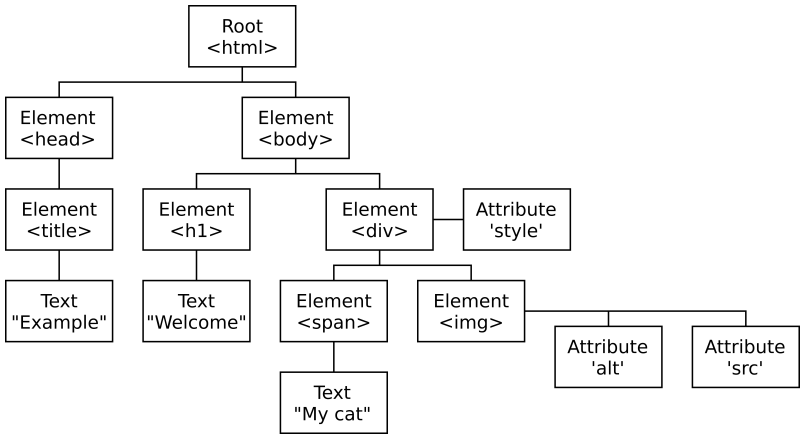


Figure 3.1. A DOM for a simple HTML page.

3.1. Basics

A browser is a tool for viewing interlinked HTML-documents spread across different computers in a network. When a user browses to a web page, the browser sends HTTP-request for the corresponding file, parses the response and creates an *Document Object Model* (DOM). This DOM is a representation of the content on the page, of the elements and their relations as a tree. The browser also finds references from the HTML to other files, such as style sheets and images, fetches them and uses them when rendering the DOM.

A server that only sends files from a file system is called a static server. The opposite of that is a dynamic server—a server that has an internal state and can give different responses, depending on its state and the request. A special case, which works much in the same way as a static web server, is when a browser is used to view files in the local file system without a web server at all.

In addition to rendering HTML, the browser also allows user interaction and computation via JavaScript. Many of the user inputs, such as key presses, drag-and-drop events, mouse hovers and others are passed into JavaScript and the code on the page can then react to these events. This code can alter the view of the web page by editing the DOM which the browser then renders.

In its early days, JavaScript wasn't as prevailing as it is today. It could be used to create some special input fields or to validate user input before enabling the button that would allow the user to submit the data to the server. Most of the time there was no need to use JavaScript at all.

One of the most significant breakthroughs of the Web was in 2005 when *AJAX* (Asynchronous JavaScript And XML) was introduced. The way AJAX works is that a piece of JavaScript can create a HTTP request on its own and handle the response in the background without the rest of the browser stopping. This special kind of HTTP request is called *XMLHttpRequest* or XHR.

A common use case is depicted in fig 3.2:

1. The user does something that requires an data from the server.

2. The browser sends the XMLHttpRequest.

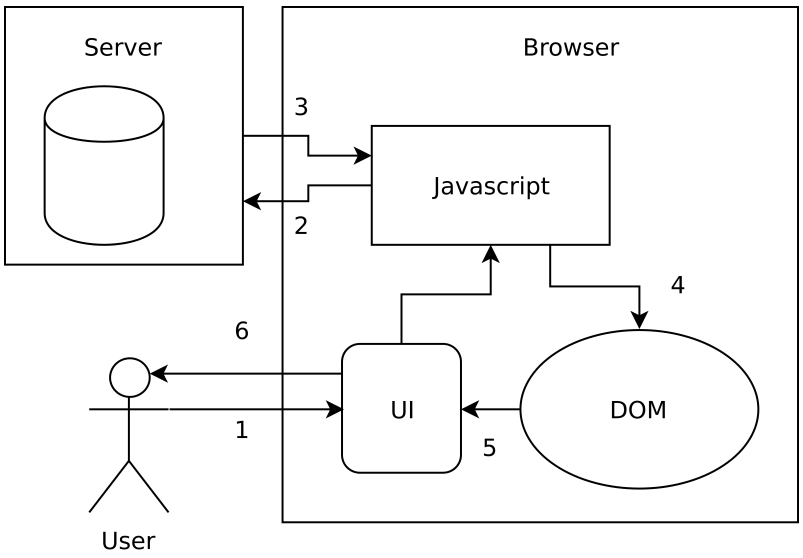


Figure 3.2. Asynchronous JavaScript and XML (AJAX).

3. The server receives the request and processes it.

4. The browser receives the response.

5. Instead of refreshing the window with the content of the response, JavaScript modifies the DOM to show the new content.

6. The user sees the updated page without a need for page reload.

AJAX completely changed the way web sites are designed. Instead of each subpage being their own separate HTML document and the browser loading and rendering them one after another, the parts can be loaded dynamically as needed and the browser can update only the part of the view that is needed.

A good example could be an image viewing application. The HTML-page itself contains a frame, a list of image names and required scripts. When the user clicks an image name, the scripts then use AJAX to load the actual image and to insert it into the frame.

Not only did browsing the Web become much smoother experience thanks to this, but it saved network bandwidth and server resources. Taken to the extreme, AJAX paved the way to *Single Page Application*s (SPA).

A Single Page Application is a web application that does not use links in the traditional way. User interaction that requires the server to be notified instead is handled with XHRs. This provides a user experience much more like that of common desktop software, as

there is no reloading of the page after each user interaction. The result is a application that feels much more responsive with faster response times.

Many, if not most, of today's Web applications are SPAs, including GMail, Twitter and Youtube. Also frameworks such as React and AngularJS are designed toward SPA development.

Still, AJAX has its problems: Every request has to be initialized by the browser; the server is not able to send any data to the browser without the browser specifically asking for it. This can be circumvented by having the client poll, or periodically ask the server for messages.

In 2011 WebSockets were introduced and provided a better solution to this problem. With WebSockets the browser can initialize a bidirectional communication channel between itself and a server supporting WebSockets protocol. After the connection has been established both the client or the server can send messages through the connection. This can be used, for example, in live sport results: The server can send a message when a team scores, and the browser then update the page accordingly. This removes the need for polling for the results, decreasing network traffic and latency.

3.2. HTML5

Web technologies as a whole move forward at an astounding pace, making more and more features that have previously been available only for desktop applications accessible to the Web. In 2014 a new version of HTML was published, called HTML5 [16]. At its core, the markup language itself received a slight face lift, but more importantly, it brought new elements to the HTML specification as well as a lot of related JavaScript functionality.

For example, there was no support for video or audio before HTML5. Any player had to be done with some external plugin, such as Java applets or Adobe Flash.

HTML5 also introduced Local Storage, which allows web sites to store data on the client side browser. Unlike cookies, local storage data is not sent to the server with each request. This makes a great place to save data only the client needs, such as a state of a SPA. A way of writing files on the client machine was also considered, but this File system API did not make it into the standard.

4. WEBRTC

WebRTC (Web Real-Time Communication) [5] is a modern addition to the Web ecosystem, making browser-to-browser communication possible for the first time. The project was started by Google, and was moved to open source development in 2012. As of this writing, WebRTC is a work in progress in Internet Engineering Task Force (IETF), World Wide Web Consortium (W3C) and Web Hypertext Application Technology Working Group (WHATWG).

4.1. Background of WebRTC

The driving force behind WebRTC's design and development was to facilitate real-time audio and video transfer between Web users. To create an application like Skype in the Web, any previously possible solution required an intermediary server to relay the audio and video stream. Not only did this setup cause additional latency, but more importantly, required more often than not dedicated servers and a lot of network capacity to route the audio and video signals. WebRTC was designed to solve this problem.

Due to the origins of the protocol, and the problems it was designed to solve, much of the documentation and specification concentrate heavily on handling audio and video. However, WebRTC does also include the possibility to transfer arbitrary data. This has been used to create, for example, peer-to-peer file transfers applications [17] and simple real-time multiplayer games [18]. Since this work concentrates in sending applications and their state between clients, we only examine the arbitrary data transmission and skip the details of audio and video streaming.

While WebRTC is designed as a part of the Web ecosystem, there are efforts to allow a native application to act as a peer in a WebRTC connection such as librtcdc [19]. This can be useful especially in cases where a server-like peer is desired.

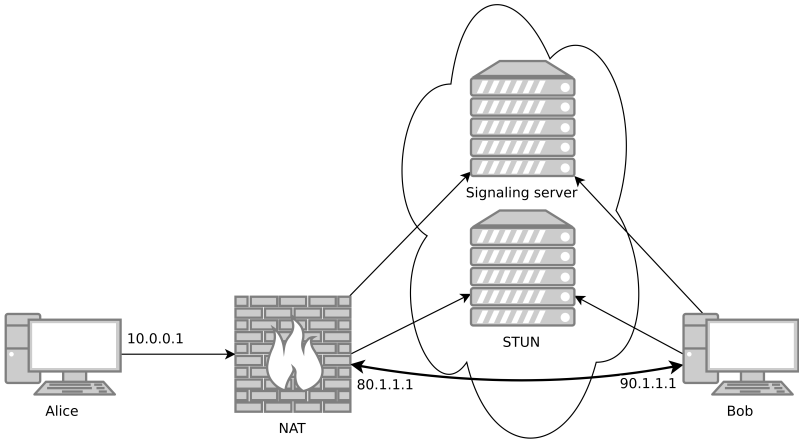


Figure 4.1. Example network for WebRTC connection.

4.2. Technical Background

WebRTC capable browsers provide a set of JavaScript *Application Programming Interfaces* (APIs) to the developer. The most important one, which controls the creation, configuration and closing of a WebRTC connection, is called `RTCPeerConnection` and is defined in JavaScript Session Establishment Protocol [20]. A more in depth specification of the inner working of WebRTC can be found in the W3C's WebRTC Candidate Recommendation [21].

Any application that needs to connect two peers in today's global internet will run into problems with *Network Address Translation* (NAT). In NAT, a network device (often a router) that passes network traffic though modifies the traffic so that the destination sees the router as the source of these packets. When the router then receives packets back from the destination, it remembers who was the original source and forwards the packets to that client. The problem arises, when this client needs to receive a packet first, as the router can't know where the incoming packet should be forwarded. To combat this problem, WebRTC uses *Interactive Connection Establishment* (ICE) [22], which itself then uses first *Session Traversal Utilities for NAT* (STUN) [23] to find any potential IP addresses that might reach this peer (see figure 4.1).

If all else fails, usually because both peers are behind strict NATs, WebRTC can fall back to using *Traversal Using Relays over NAT* (TURN) [24], which uses a relay to forward messages in case a peer is unable to create a direct connection to another peer. This,

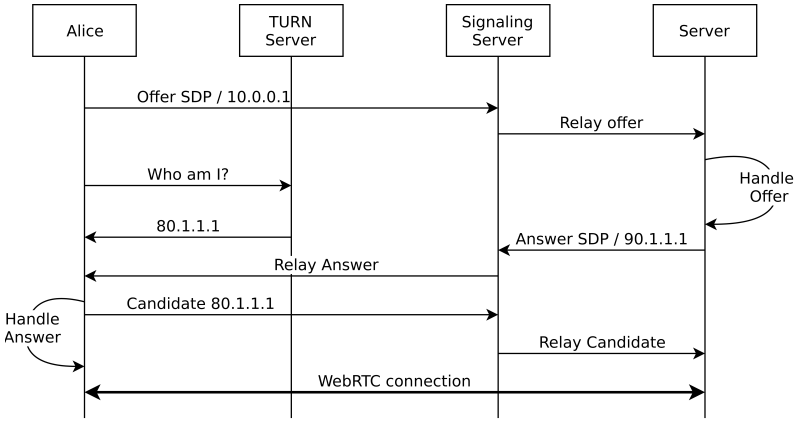


Figure 4.2. WebRTC Connection establishment.

of course, makes the connection by definition not peer-to-peer, but is a useful fallback method for the times its needed.

These protocols are familiar to those working with Session Initialization Protocol (SIP) [25] used in Voice over IP systems (VoIP). This is not a coincidence; WebRTC was designed to work with existing systems. However, this has not gone without criticism. ObjectRTC (ORTC) is another technology that aims to provide the same functionality as WebRTC, but on a lower level API without SDP. This work concentrates on WebRTC due to its broader support in the browser market.

4.3. Connection Establishment

Opening a connection between two clients is a classic catch-22: two clients want to connect to each other, but how can they agree on how to connect, if there is no way for them to communicate. If one peer acted as a server—that is, it would be listening on incoming connections on a known address and a port—things would be easy. However this is not what WebRTC does.

Instead, WebRTC uses a bootstrapping technique called signaling. In this signaling stage, the two peers use a arbitrary method (almost always a common server) to relay signaling messages. The goal of the signaling stage is to share the required information between the peers so that they can create a direct connection. In this phase, the two peers exchange messages that define the different streams, their codecs and network protocols to use, as well as encryption schemes and how the two can connect to each other.

This section covers a simplified view (figure 4.2) of the use of ICE in the context of

WebRTC. A more in depth and accurate description can be found in [22]. Figure 4.1 shows how the network is set up in the example.

The WebRTC Candidate Recommendation [21] states that the signaling needs to work according to the offer/answer model and that the offer and answer are in Session Description Protocol (SDP) format [26, 27]. The JSEP does not mandate how these messages are transmitted, so the channel, over which these are sent, is left up to the user to decide. In theory the signaling could be carried out over any bi-directional channel, but in practice this is almost always handled by a script that communicates over XMLHTTPRequests or WebSockets. These protocols require an intermediate server to forward the signaling messages between the browsers. The reason the signaling channel is not defined, is to allow intercompatibility with previous systems, most notably SIP [25].[20]

4.3.1. Example

To clarify how a WebRTC connection is usually created, this section will go over the signaling phase covering how the WebRTC API is used. The network setup for this example is presented in figure 4.1 and figure 4.2 is a sequence diagram of the following description.

The signaling phase in WebRTC connection begins with the creation of an SDP Offer. First Alice creates a `RTCPeerConnection` to hold the state of the connection. Then she creates the offer with a call to `RTCPeerConnection.CreateOffer`, passing information on what kinds of data streams she wishes to send and receive, and what protocols and codecs to use. She saves this offer as her side of the connection with `RTCPeerConnection.SetLocalDescription`. The offer is then sent to the Bob via a signaling server.

After receiving the offer Bob makes the decision of whether or not to allow the connection. If Bob decides to continue the process, he also creates an `RTCPeerConnection` object and saves the offer he received with `RTCPeerConnection.SetRemoteDescription`. An answer to the received offer is created with `RTCPeerConnection.CreateAnswer`. Bob then saves this answer with `RTCPeerConnection.SetLocalDescription` and sends it to Alice through the signaling server. After this Alice saves the answer with `RTCPeerConnection.SetRemoteDescription`.

The WebRTC frameworks of the two browsers started collecting candidates according to the ICE protocol after the creation of offer and answer. For every local candidate (ones that describe the peer itself) that a peer can collect, WebRTC will call `RTCPeerConnection.onicecandate`. The developer has to set a function to handle these calls and to send the candidates to the other peer using the signaling channel. When receiving a remote candidate from the signaling channel, a peer only needs to register it using `RTCPeerConnection.addIceCandidate`. The WebRTC framework will handle the rest of the connection forming in the background.

The WebRTC framework in the browser starts testing for connectivity when the first candidate is added. Because of NATs, firewalls and other reasons, many of these candidates may fail to create a communication channel. The WebRTC system continue gathering candidates using STUN and notifies the user code when these are found. In the example Alice learns from STUN that she is behind a NAT and that her global IP address is 80.1.1.1. She messages this candidate to Bob using the signaling server.

The last resort for making a connection, when both peers are behind a NAT, is *Traversal Using Relays around NAT* (TURN) [24]. In TURN a client that is behind a NAT sends a request to a server in the public internet and requests for a port lease. The TURN server responds with an address–port -pair to the client and keeps the connection open for further communication. Any data sent to the server to that given address–port -pair is then routed through the open connection to the client. Once the peer receives the address–port -pair from the TURN server, they can then send that as a candidate to the other peer.

4.4. Security of WebRTC

All WebRTC communications are encrypted by design. To secure media streams, WebRTC uses Secure Real Time Protocol (SRTP), and for arbitrary data either Datagram Transport Layer System (DTLS) in case of UDP, or *Transport Layer System* (TLS) when TCP is used [28]. These security mechanisms guards the sent data from tampering or eavesdropping, even when a TURN server is used. As WebRTC does not define the signaling method, its security is also left up to the user to choose.

One security concern, however, has been touted by a wide audience: WebRTC can be used to leak IP addresses—local and global, and even ones that should be hidden due to

VPN use—to the Web [29]. A web page may contain a script that gathers identifiable data of the visitors. This script, when loaded in a browser, initiates a RTCPeerConnection and sends out an offer. Even without receiving an answer, the browser starts to gather candidates, and the malicious script can then send these to the attackers server. Because of this, some users have blocked WebRTC altogether, limiting the usefulness of the technology.

5. TEST SETUP

To better understand the framework presented in the next chapter, this chapter will explain the test application that is later liquified. This application is a simple SPA, that at this point does not have any code for communicating with another peer, transferring the application or synchronizing its state. It is meant to be a blank slate for the liquification tests.

5.1. Test Application

The test application is a simple note taking SPA, shown in figure 5.1. It consists of a button to create a new note, a list of all notes and a text editor area. The editor area can be used to edit the note that is selected from the list of notes.

The application consists of a single HTML5-file that describes the elements and the structure of the overall view of the application; a CSS-file that manages to look of the application, and a JavaScript file that contains the application logic. The application also loads jQuery [30] from the web to both ease the development and to work as a test for how the framework handles external resources.

The application is designed to allow taking multiple notes, each note having its own content. The *New note* button opens a dialog asking for a name for the note. After providing that, the new note appears in the box on the left and is automatically selected.

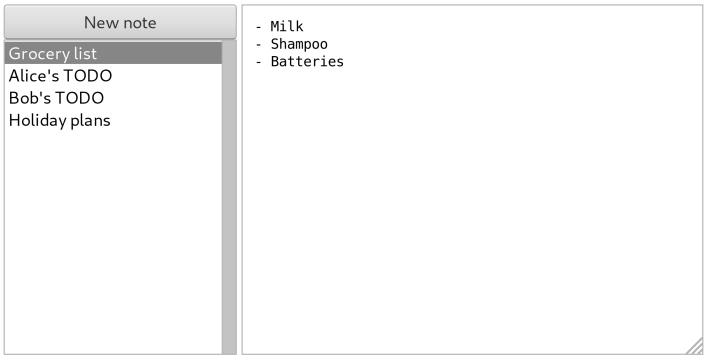


Figure 5.1. Test application.

These notes are held inside an JavaScript object `notes`, where the name of the note is the key, and the note content the value. The contents of the selected note are shown in the editor window, and whenever the content is changed (by writing, pasting text or any other reason), the new content is saved into the `notes` array.

Listing 5.1. Standalone editor.

```
var notes = {};
var currentNote = undefined;

var createNote = function(n) {
    notes[n] = "";
    $('.noteList').append($('<option />').html(n));
};

var selectNote = function(n) {
    currentNote = n;
    $('.noteList').val(n);
    $('.editor').val(notes[n]);
};

$('.editor').on("change keyup paste", function() {
    if(notes[currentNote] != $(this).val()) {
        notes[currentNote] = $(this).val();
    }
});

$('.noteList').change( function() {
    currentNote = $('.noteList').val();
    $('.editor').val(notes[currentNote]);
});

$('.newNoteButton').click(function() {
    var newNoteName = prompt("Name for new note:");
    if(newNoteName == null) {
        return;
    }
    createNote(newNoteName);
    selectNote(newNoteName);
});
```

This application does not include any code for communicating with another peer or a server. This means, not only, that all application logic is in the client JavaScript, but that the system that is used to liquify the application has to provide the communication channel.

The JavaScript code of the test application is shown in listing 5.1. The HTML- and CSS-files are omitted as trivial.

5.2. Test Method

The test application is designed to be liquified using the framework implemented in this thesis. The way this liquid application is envisioned to work, is to allow simultaneous editing of different notes and automatic synchronization of the contents of the notes that are being edited.

6. IMPLEMENTATION

To test the viability of using WebRTC in developing liquid software and to facilitate the creation of liquid software, a small framework was implemented. This framework, *LiquidRTC*, is the first peer-to-peer capable framework for creating liquid software. To complement the peer-to-peer nature of the system, the decision was made to not fetch applications from a server, but rather to use client repositories. This, in turn, forces the framework to support application transfer from device to another—a feature that drove much of the design. Last, but not least, the framework is designed to be as easy to understand as possible for future work—as a proof-of-concept for future WebRTC based liquid software.

To concentrate on these properties of liquid software, many had to be left with little or no attention. Most visible feature that was not considered is the UI adaptation. Since the work is done on the Web platform, a plethora of tools exist for creating reactive web applications, and therefore it didn't make sense to dedicate time for it.

This Framework consists of roughly 4 parts: *Program serializer* needed for making the applications transferable, *Networking* to provide the framework a way to distribute applications as well as the applications themselves to share data with other peers, the *API* that's provided to the developer to handle synchronization and other communication between peers, and *UI extension*, which allows end users to interact with the liquid aspect of the application. In addition to these, the implementation also includes a landing page, which ties these together. In production this landing page would likely be replaced with a more purpose-built version. The following sections will go through these components in order.

Figure 6.1. The framework landing page.

6.1. Landing page

The provided landing page is a minimal web page that is meant only to serve as a way to bootstrap the rest of the framework. To the end user, it looks like the page shown in figure 6.1. The only element on the page is a square containing the instructions to drop the application files in it. The landing page runs scripts for receiving the application files from the user, the program serializer, networking handling and a way to launch programs, either from provided files or ones coming over the network.

Everything on the landing page is run completely on the client side and it can actually be loaded from local file system instead of a server.

6.2. Program serializer & launcher

As discussed earlier, the browser is designed to view HTML-documents and linked other files. These files are pointed to with URLs and these URLs either point to a file on a HTTP-server or a local file system. Since one of the focus points of this thesis is to share the applications between users directly in peer-to-peer fashion, using an HTTP-server to provide the files is not acceptable.

Sending the contents of all the files from one user to another one by one is possible, but there is no way to render them: The contents of these files refer to one another with a file name, but after receiving the data over the network these contents are just strings in JavaScript, not actual files with a location or a filename and, as such, a URL has no way to point to them. A way to write files to disk in JavaScript was suggested, but is not widely accepted or implemented [31].

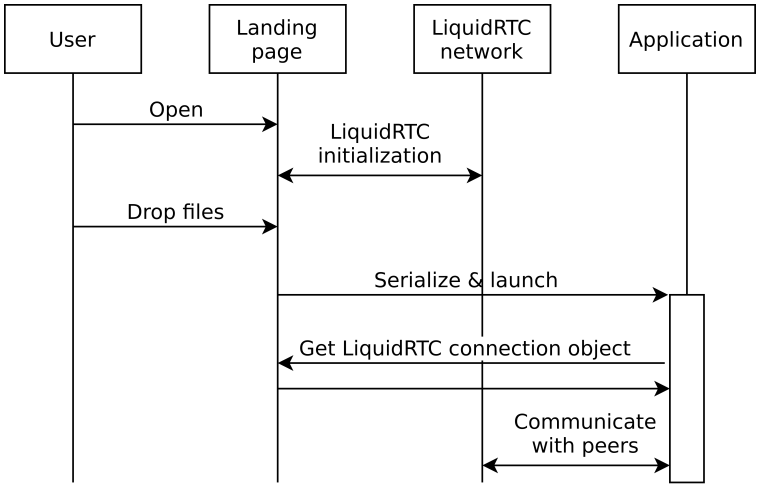


Figure 6.2. Launching an application from user provided files.

The solution the framework implements, is to serialize the files required into a single string containing the complete application. This string can then be loaded as a HTML document the browser can render as described in section 3.1..

The program serializer starts by going through the files given by the user, looking for an HTML file. While doing this, the serializer also catalogs all other supported files to key–value -pairs, key being the file name and the contents of the file being the value. Next, the serializer steps through the contents of the HTML file and examines URLs it finds. If an URL is absolute—that is, it points to a file on a specific host—it is left as is. This is done so that if a developer wishes to use resources from the internet (such as scripts or media from a CDN), they can. It is also possible to provide all required files to the serializer and not to depend on a connection to the internet. If, however, a URL is relative, the serializer looks for a file that would match it in the catalog that it built earlier and inline that file if found. If the file is not found the program serializer fails.

This serialization not only limits the application to just one HTML file, but also makes it the only file that can reference other files. That is, no URLs in CSS or JavaScript files can be replaced with the contents of the pointed file. This is usually not an issue at the moment, but since the newest iteration of JavaScript includes support for importing modules [32] the situation may change in the future. Another missing feature is support for images. Support for these should also be straightforward to implement with the use of data URLs [33].

Listing 6.1. Serializing and launching a liquified app (simplified).

```
function handleDroppedFiles(evt)
{
  var extra_scripts = ['js/ui.js', 'js/getDS.js'];
  var f = evt.dataTransfer.files;

  serializeProgram(f, extra_scripts, function(html) {
    var new_win = window.open();
    new_win.my_source = html;
    new_win.document.open();
    new_win.document.write(html);
    new_win.document.close();
  });
}
```

After the dropped files have been processed into one string, a new window is opened in the browser and this string is injected in as its source. The source is also provided for the new window in a special variable called `my_source`, so the application can then send this string in future when it needs to replicate itself to new peers. It also injects the scripts that create the UI extension and copy the `liquidRTC` object to the launched application. The code that implements the program launch is shown in listing 6.1.

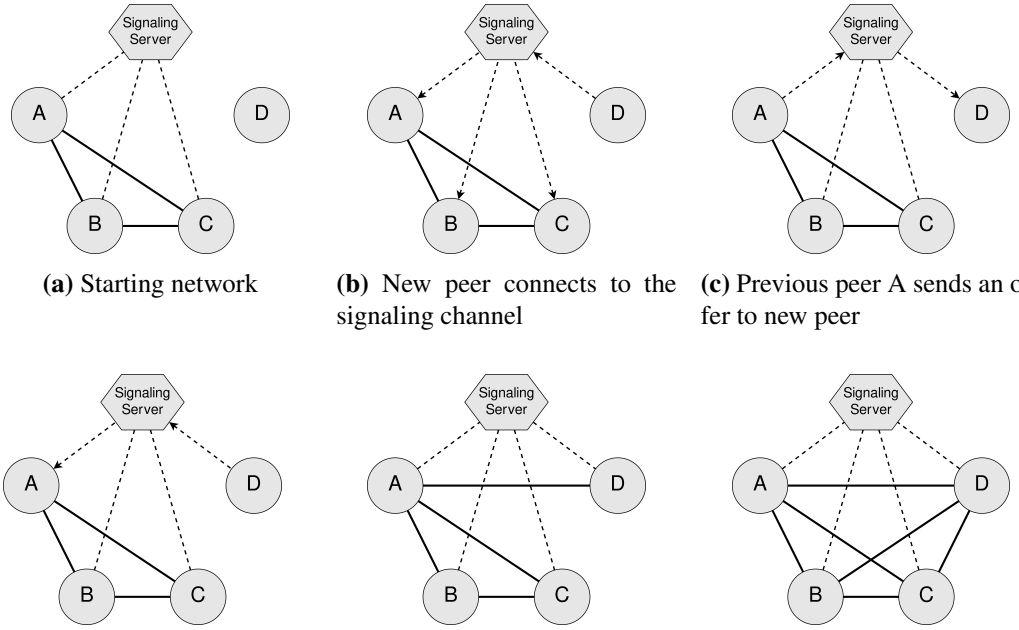
The `GetDS.js` script fetches the `liquidRTC` object from the landing page into the newly opened application and initializes it. This copying of an object between the browser windows is a very unusual and forces developers to write code that is very peculiar.

When writing applications that use LiquidRTC, the developer has to use the functions presented by the `liquidRTC` object. However, at the time the application code is written, this object does not exist and there are no clues as to where it is coming from. It is only at runtime, when this object gets injected into the application.

6.3. Networking

Liquid software cannot exist without some sort of connection between the devices, through which the software or state can be transmitted. The signaling in LiquidRTC is done via a signaling server that is provided as part of the framework. The server is a simple Node.JS application that is mostly used to pass messages between the peers. The signaling server and the peers use a simple JSON-based protocol for routing the messages.

To keep the setup and routing as simple as possible, the complete network topology



(a) Starting network

(b) New peer connects to the signaling channel

(c) Previous peer A sends an offer to new peer

(d) New peer sends a response to A

(e) New peer opens WebRTC connection with peer A

(f) New peer has successfully joined the network

Figure 6.3. Joining the LiquidRTC network.

(figure 2.1d) was chosen. This allows all peers to directly communicate to any and all peers without the need to pass messages through another peer. A device can belong to multiple such networks, or move between different LiquidRTC networks, but the current implementation does not support transferring applications between different networks.

To make the developers' work easier, the functionality to automatically create a network of WebRTC connections between the browsers that open the landing page was implemented as part of the framework. A script on the landing page first creates the WebSocket connection with the signaling server. The server then facilitates the creation of connection between the peers.

6.3.1. Automatic connection creation

The framework automatically connect peers into a LiquidRTC network. This process is pictured in figure 6.3. A solid line represents a WebRTC connection and a dashed line a WebSocket connection. A message being passed around is shown as an arrow.

When the user opens the landing page, a piece of JavaScript on the page creates a LiquidRTC object that in turn initiates a WebSocket connection the signaling server. The page also prompts the user for an identifying name that is later used to show the differ-

ent connected peers. This name is sent to the server over the newly created WebSocket connection.

The server keeps a list of all peers that have connected to the network so far. When a new peer connects (D in figure 6.3b) the server will send a `getOffer` message to all the existing peers notifying them that a new potential peer wishes to join the network. This message contains the name of the user who just connected, telling the earlier connectees who it is that wishes to join.

To avoid confusion, the example assumes that peer A is faster in communicating with the new peer and is able to create the WebRTC connection before peers B or C send their offers. In reality, the connection establishment runs concurrently for all peers.

Assuming the WebSocket connection is alive, the browser of A will receive the `getOffer` message. Without notifying the user, the framework will automatically create an offer as described in 4.3.This offer is then bundled with information about the sender A and the recipient D in a `offer` message and sent over the WebSocket connection to the server. When the signaling server receives an offer, it unpacks the message, checks who is it for and tries to redirect the message to the correct peer. This step is shown in figure 6.3c.

Once received, D handles the offer as described in section 4.3.1. and sends an `answer` message containing the SDP, sender and target information to the signaling server. The server once again unpacks the message to check who is the target of the message and relays it if possible (figure 6.3d). Any candidates the peers send to each other are also relayed through the signaling channel in the same fashion, combining the candidate with information about the intended recipient.

After the handshakes are finished, the browsers form the WebRTC connection if they find suitable candidates (fig 6.3e). Once the connection establishment has been completed between the new peer and all previous peers, this new peer has successfully connected to the LiquidRTC network (fig 6.3f). After this point, the signaling server will not interfere with the communication between the connected clients any more. Only time the server will contact the peers is when a new client informs it wishes to join the LiquidRTC network.

In this implementation the signaling messages are not authenticated. This means any

peer can send any of these messages to the server. For example C can send an offer to D stating the sender is A. It would be simple for the server to check that the sender is who they claim to be. This was not done in order to keep the code as simple as possible.

6.3.2. Synchronization

There are multiple ways to handle the state synchronization of the applications. First option is to automatically share the whole application state—an example of application level state synchronization granularity as discussed in chapter 2.3. If the synchronization is limited to what is rendered, the application synchronization can be done by synchronizing the whole DOM of the application between peers. DOM synchronization is somewhat restrictive in that it does not allow partial synchronization. Another problem is that if the application needs to react to the changes of the DOM. Therefore, this solution does not provide the level of flexibility that the framework aims for.

Another problem is that often with the DOM-synchronization based approach is that some of the state is often hidden in JavaScript variables. These are not directly rendered, but may effect how the application functions. As such, these must also be synchronized in some fashion.

An automated way to solve this issue would be to give the application a list of JavaScript objects that need to be synchronized. The framework would then keep track of these objects, and when one of them changes, its contents get sent to other peers that copy the change into their state. This allows component level granularity state synchronization, but still does limit itself to only sending information about the current state, not things such as events.

The synchronization primitives LiquidRTC includes (described in the next section) provide the developer the most freedom, which on the other side of the coin means they contain the least amount of automation. The framework can be used to send arbitrary messages over the LiquidRTC network to one or all of its peers via the provided API and handle the received messages in any way they wish. These messages may be the complete state of the application, a component on the page, the difference between previous and new state of some component or an event that is not supposed to be saved as part of the state. One such event could be to used to show a notification to the user.

6.4. API

For the user of the framework—the developer of future liquid software—most of the details listed above are insignificant. What is important for them, is the API the framework provides. Like the framework as a whole, also the API is designed to be minimal, with as much flexibility as possible, while providing easy access to the most crucial communication features. This API is encapsulated in the `LiquidRTC` class and usable through the `liquidRTC` object that is copied to every application by the program launcher.

The functions the `liquidRTC` object provides are rather bare and minimal. `broadcast` takes a `message_id` string and a `message` JavaScript object and sends those two to all peers in the LiquidRTC network. `send` takes `peer` as an extra argument and uses that to find a recipient of the same name and sends the data to only that peer. `on` is used to register a handler for a message. This function takes a `message_id` string and a `handler` function. After registering a handler with the `on` function, whenever the peer receives a (`message_id`, `message`) pair that has a matching `message_id`, the `message` gets passed as a parameter to the registered handler. `broadcast` is simply a a specialized version of send that sends the same message to all peers. The usage is practically identical to Socket.IO [34], a well known messaging library for JavaScript works.

This interface also resembles the event-driven system used often in JavaScript. The application sets a listener with the `on`-function to react when it receives a message, the same as registering an event handler. Later any peer may send a matching message to trigger the call for the registered handler. The sent message can pass arbitrary data with it.

In addition to these functions, the framework also provides the developer with access to all the `RTCPeerConnection` objects. The connection with the signaling channel is not available to the developer, as it is intended as a implementation detail and should not be used in the applications themselves.

This decision ,however, has a considerable effect on security of the network: every application can receive all messages sent through the LiquidRTC network. This means that a malicious application, once on a target browser, can listen to the messages that are meant for other liquid applications and then possibly send them to the attacker through

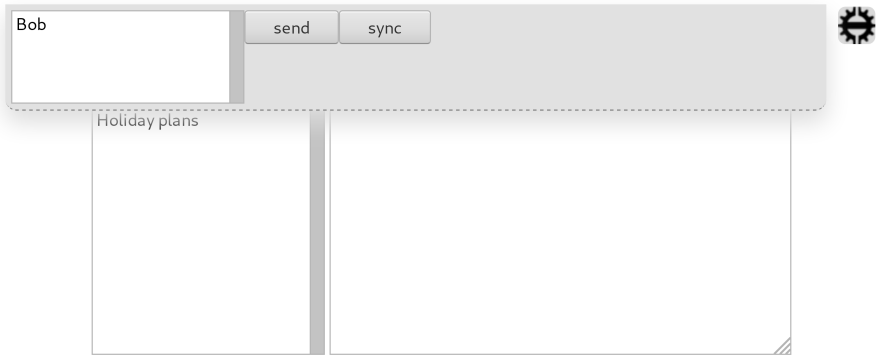


Figure 6.4. UI shown.

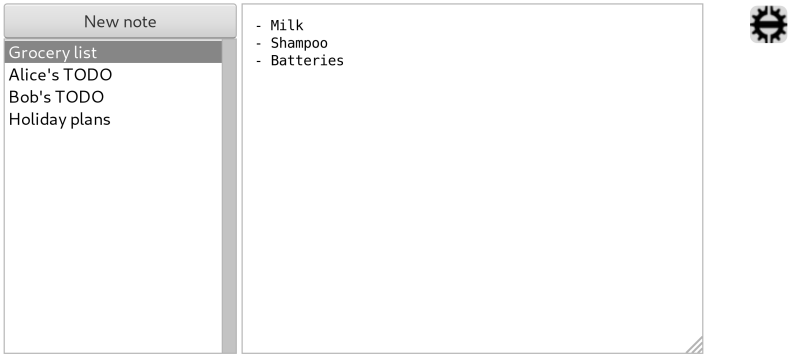


Figure 6.5. UI hidden.

some other channel. As the security of the whole LiquidRTC in the current proof of concept state relies on trusted peers and applications at this state, this problem is not considered.

6.5. UI extension

While the Framework is as unobtrusive as possible, it's not completely invisible to the user. One of the requirements of the manifesto [1] is that the user is in control of their program synchronization—they must have a way to select peers with which to share the programs and their state.

To fulfill this requirement, the framework implements and injects a small UI extension to all applications during the serialization and launch phase. The UI is shown in figure 6.4: It contains a list of all other peers and buttons to push the application to the selected peer. It is hidden by default and shown only after user clicks the button on the top right corner of the page. The test application with the UI hidden is shown in figure 6.5. This is to keep it out of the way when its not needed. Other than this small addition, the user provided

SPA is identical to the original in appearance and function.

In this implementation, the look and functionality of this panel is defined by the framework. In a real world scenario this should not be the case, but rather the application developer should be able to modify the panel to best fit the specific application.

The only function the UI currently has is to allow the user to transfer the application to another peer. The application transfer is simply a special `message_id` that has the serialized application code as the `message` content. Clicking the *send* button uses the `liquidRTC.send` method to send the message above to the selected peer. The landing page has set up the listener for these messages that opens the received application.

7. EVALUATION

To evaluate how well the framework from the previous chapter meets the set goals, the application described in chapter 5 was liquified using the framework. The liquification process can be seen as two distinct steps:

- Automatic connection sharing and UI injection handled by the framework.
- Application specific synchronization handled by the developer's code.

The first one was covered in the previous chapter. This chapter will go through the latter using the test application from chapter 5 as an example. As stated before, the goal is to transform the given application into liquid software with the ability to move and synchronize its state between devices.

7.1. Liquifying the Test Application

The code in listing 7.1 sets up the message handler for a message type `editor-state`. As the `message`, this handler expects an object with a field `note` that contains the name of the modified note, and `msg` that holds the text content of that note. The function this handler calls checks if the received note exists on this peer, creates it if it does not and saves the content in the `notes` object. If the note is active, the editor area is updated to display the updated note.

Listing 7.1. Handling received application state.

```
liquidRTC.on('editor-state', function(msg) {
  update_note(msg.note, msg.msg);
});

var update_note = function (n, t) {
  if(notes[n] === undefined) {
    createNote(n);
  }
  if(currentNote == undefined) {
    selectNote(n);
  }
  notes[n] = t;
  if(n == currentNote) {
    $('.editor').val(notes[currentNote]);
  }
};
```

Listing 7.2. Sending the test application state.

```
var syncNote = function (n) {
  var msg = {
    note: n,
    msg:  notes[n]
  };
  if(typeof liquidRTC !== 'undefined') {
    liquidRTC.broadcast('editor-state', msg);
  }
};

$('.editor').on("change keyup paste", function() {
  if(notes[currentNote] != $(this).val()) {
    notes[currentNote] = $(this).val();
    syncNote(currentNote);
  }
});

$('.newNoteButton').click(function() {
  var newNoteName = prompt("Name for new note:");
  if(newNoteName == null) {
    return;
  }
  createNote(newNoteName);
  syncNote(newNoteName);
  selectNote(newNoteName);
});
```

This application always broadcasts every change in the `notes` object to all peers. This means the `send` function is never used, because there is no need to send a message to a selected peer. The broadcast first constructs the required object containing the note name `note` and its content `msg` and then broadcasts that with the message type `editor-state`. The synchronization function is called when the content of a note is changed as well as when a new note is created. The code in listing 7.2 covers the code that does all this. The handlers for button press and note edits are the same as in listing 5.1, but with an additional call to `syncNote`.

Like described in 6.2., the API is available through the `liquidRTC` object that gets injected into the program at the program launch phase. To allow the application to function correctly either within a LiquidRTC system or outside it, the developer has to surround the references to `liquidRTC` in checks that such object exists. These checks are visible in both the code listings 7.1 and 7.2.

7.2. Example Use

Alice opens the landing page by pointing the browser to the server holding the landing page. She then drag-and-drops the three files making up the editor software into the landing page. The editor opens in a new tab in the browser. She creates a new note, “Alice’s TODO” and writes down some text.

She clicks the icon on the top right corner of the page to open the LiquidRTC UI extension. There, in the box she finds the names Bob and Charlie. She selects Bob, clicks send and then does the same for Charlie. Quickly thereafter, she sees two new notes appear in the note list. She doesn’t mind them, but continues writing down her own notes in her TODO list.

Bob has created a TODO list of his own and is typing some notes in there at the same time Alice is writing down hers. Charlie has started writing down ideas in a note called “Holiday plans”. Each of the notes are being edited at the same time, with each browser keeping a full copy of all the notes all the time.

Alice is interested in the plans for the holiday, so she selects the note. She comes up with a new idea and when she sees Charlie has stopped typing, she adds her own contribution to the note. Charlie can see the updating text in real time on his browser.

Suddenly Bob's browser crashes while he is writing a note. He restarts the browser and reopens the landing page. He then tells Charlie what happened and asks him to send the application back, which he does. Bob receives the application and the complete state, including the note he was writing and continues from where he left off.

7.3. Liquid Software Aspects

LiquidRTC tries to balance between usefulness and flexibility for different kinds of liquid software. Assessing the liquified test application against the characteristics of liquid software listed in 2.3. provides an estimate on how well the implemented framework is suited for creating different kinds of liquid software.

LiquidRTC does not differentiate between sequential or simultaneous use, nor between single or multiple user use cases, simply because it does not try to handle the state, but rather gives the developer the tools to do synchronization as fit for the application. However the tools do not include support for concurrent modifications to the same data, making simultaneous use case support somewhat questionable.

The peer discovery is done via a shared URL, or a local file. These are used to connect to the common signaling channel. Because the LiquidRTC framework works on the Web platform, there are no real alternatives.

The application transfer granularity in LiquidRTC is locked at application level and the applications are always symmetric—the application always sends its own source when transferring to another device. The state synchronization part is a little vague. In the strict sense, there is no state synchronization in the framework, but the developer is given tools to provide component level synchronization granularity.

LiquidRTC is designed to use push as the Liquid User Experience initialization trigger: The end user is given the tools via the UI extension to select a target device to which to send the application. The application developer can use the provided API to provide the presence initialization trigger, but the framework is not designed for such use. The pull initialization trigger is not possible with the current design.

The LUE primitive that LiquidRTC as a framework implements is forking, but as shown with the example application, the tools provided make it easy to implement cloning. Migration would also be trivial to implement as forking and then closing the

original copy of the application.

As one of the main focuses of this work was sharing applications from the end users machine and not from a web server, the example are uses client repository application topology. It is still possible to use server repositories by providing the applications from a server like any traditional web application. Since this skips the serialization and application transfer features of LiquidRTC and only uses the UI extension and WebRTC connection establishment, it is strictly less interesting as an example. However, in production work such setup may have its benefits, as server repositories are more reliable for application storage.

Because the peer-to-peer nature and symmetric applications, the only possible state topology is multiple master. This is somewhat unfortunate, as multiple peers doing simultaneous changes can lead to cases where a consistent state is hard to maintain.

As there is no server interacting with the application built on top of LiquidRTC, the applications are thick. Of course an application specific server can be built and each peer can communicate with it, but it would be the question why use LiquidRTC in the first place.

The framework has no notion of security: each peer implicitly trusts every other peer—that is everyone who can see the landing page and open the signaling channel. Any peer can force an application to open on any other peer on the network. This application can do anything a normal browser window can.

7.4. Other

Transferring the applications and their state works well with modern browsers. Various versions of Firefox and Chrome were tested in a PC and mobile devices. Back in 2015 WebRTC had problems with interoperability between different browsers, but these seem to have been resolved.

The way the applications are opened into a new browser window is considered as a pop-up by every browser tested. This feature makes it somewhat more cumbersome to use the framework, as receiving an application requires user interaction to allow pop-ups and may cause an application to be lost.

Because of the way the framework handles application transfer and serialization, it is

limited to strictly single page applications. In some cases this can be a limitation. Using some other method for transferring the applications between peers may provide a less restrictive alternative.

Looking back at the Liquid Software Manifesto [1], the framework is a mixed success. Starting from the less successful points: The manifesto calls for effortless roaming and minimizing the effort the end user has to make. The manual push method of transferring applications does mean that the user has to know which device they are going to use next, and the fact that many browsers blocks the transferred applications as pop-ups forces the user to have both devices at hand while transferring the application.

The requirement for synchronizing state transparently is harder to judge. On one hand LiquidRTC does allow very precise control on what to synchronize, when and how to react to those events. However it also evades the responsibilities of automating the synchronization on the argument that it is not not possible to do so while providing the necessary flexibility of synchronization.

Since the framework works on the web platform it can run on almost any device. Some less popular browsers on some platforms may not support WebRTC, but this limits the device support only minimally. The framework also puts the end user in control of if an application should be moved to another device.

8. COMPARISON

The LiquidRTC framework was compared with a solution that aims to solve the same problems. However, because liquid software has received limited attention, there are only a handful of libraries or frameworks written. Moreover, the whole notion of liquid software is very broad, and every library has to make decisions on questions like what is the LUE trigger, or what level of granularity to support. This unfortunately means there is no existing system that allows a fair apples-to-apples comparison with the LiquidRTC framework. Most regrettably, no library with support for transferring applications between devices could be found.

8.1. Liquid.JS

Liquid.JS, by Jari Voutilainen, was chosen as the library to compare LiquidRTC against. Liquid.JS is a server–client system, and its design heavily concentrates on the question of state synchronization in an automated fashion, minimizing the work the developer has to do to keep different peers in sync.

Sadly, there seems to be an implementation detail with Liquid.JS or the way it is bundled that makes it incompatible with the way the test application was built. Due to this, the test application liquification with Liquid.JS was only partly successful. However, the theory behind Liquid.JS is sound, and thanks to its developer, a comparison between Liquid.JS and LiquidRTC was possible.

8.2. Networking

Liquid.JS is built on top of a lightweight Socket.IO server. Where LiquidRTC server is used to relay only the signaling messages, in Liquid.JS system the actual data sent between the peers is relayed through the server.

Liquid.JS includes a messaging server very similar to the one used by LiquidRTC.

They both are Node.js applications and pass simple messages in JSON format, and neither includes any communication code that is specific to a single liquid application, i.e. all the liquid application developed on these platforms use the same message formats.

The Liquid.JS server does more than that however, as it is used as the application repository in its ecosystem. It also handles all the synchronization messages passed between the peers, whereas the signaling server in LiquidRTC was used only to forward the signaling messages. Because Liquid.JS server is used to serve the applications, a common scenario would be to run one server for each different liquid application. On the other hand, since the LiquidRTC server is only used for signaling, there can be multiple different LiquidRTC applications in the same LiquidRTC network. This makes it easier for LiquidRTC applications to communicate between different applications through the LiquidRTC network. This may be a useful side effect, or a unwanted security problem, depending on the viewpoint.

8.3. State Synchronization

The LiquidRTC network cannot be said to have a shared state as the framework does not automatically synchronize the state. Instead, the developer is given the tools to do the synchronization by sending and handling messages.

Liquid.JS, on the other, hand takes a completely different approach. When synchronizing, it takes a snapshot of the current DOM of the application (similar to the program serializer in LiquidRTC, but much more sophisticated) of the page and sends the required changes to get from the initial state to the state of the snapshot. The receiver then resets itself to the original state and applies the received difference. This forces the DOMs to always synchronize correctly, without the need for sending and handling messages. This is a very useful feature of application level granularity.

Unfortunately, reloading the DOM causes some parts of the state, such as the attached JavaScript event handlers, to be destroyed. This means that if a script attaches a click handler to a button after the document has loaded, the DOM reload causes this to be lost. The framework does contain a way of registering these handlers so that the framework can recreate them after a reload, but this erodes the benefit of automated state synchronization.

Neither the synchronization message of Liquid.JS nor the messages sent with Liq-

Figure 8.1. Liquid.JS user interface.

uidRTC are automated in the sense that the framework would notice a meaningful change and run the synchronization without being explicitly told to do so. In theory, Liquid.JS could probably detect a change in the DOM (or a part of it that the developer defines) and use that to trigger a synchronization. Due to the possible large DOM dumps and the complete reload of the page, this automation would likely be very distracting.

Liquid.JS provides a similar user interface for synchronizing the state as LiquidRTC does for transferring the application. The interface is shown in figure 8.1.

8.4. Application Transfer

Liquid.JS examples are server centric, resembling more traditional web applications. As such, the example applications are loaded from the server and don't have any method of loading applications from user repository. This in turn means there is no need to transfer an application from one peer to another.

However, as Liquid.JS does handle the state synchronization by transferring the DOM and the registered JavaScript, it seems reasonable to assume it could be adapted for transferring the whole application.

8.5. Application Code Difference

Liquid.JS initialization is done by instantiating an object from the Liquid.JS module. This initialization is functionally equivalent to the serialization and launch steps in LiquidRTC: It adds a UI element for synchronization and creates the required network connections.

To send a synchronization message from one client to others, the created object provides functions `virtualize` and `transfer`. The example code in listing 8.1 shows a minimal example how to add synchronization to take place when a new message is created using Liquid.JS. The code is based on the code in listing 5.1, with only the lines marked as new being added.

Listing 8.1. Application liquification using Liquid.JS.

```
var LiquidJS = require('Liquid');                 // new
var liquidjs = new LiquidJS();                    // new

var createNote = function(n) {
  console.log(n);
  notes[n] = "";
  $('.noteList').append($('<option />').html(n));
  liquid.vdom.virtualize();                       // new
  liquid.vdom.transfer();                         // new
};
```

8.6. Other

Since both Liquid.JS and LiquidRTC are designed to be used in a browser, they both have the same reactive web design support for UI adaptation. Another feature that is tightly coupled with the browser ecosystem is the discovery method: Both Liquid.JS and LiquidRTC based applications discover and connect to each other via a shared URL.

The Liquid User Experience triggers of the two frameworks share an identical design. In both cases, the end user is provided a list of connected peers and the ability to push the application state to the target peer. The difference, that Liquid.JS already has the application open on both peers and only the state is pushed, is irrelevant in this case.

9. FUTURE WORK

This thesis has outlined a proof of concept for a framework for sharing simple applications and their state using WebRTC. As such, many real world requirements and limitations were ignored in order to focus on the main goal. Some of these omissions are easy to include, such as safer signaling and UI adaptation. Others, such as authenticating user in a peer-to-peer network are not. This chapter is not going to list all these, but rather collect ideas that could make a well developed production ready version of LiquidRTC even better.

9.1. As a Browser Extension

The framework could be modified to work as a browser extension. This way, the UI elements for selecting peers and initiating the application transfer/state synchronization would be directly in the browser. Instead of a landing page, to which the user would drag the files, the user would open the app they wish to share. This allows a multi-repository workflow in addition to the user-repository one developed in this work.

To join a network and to be able to receive applications like in the current design, the user would, however, need to open some sort of landing page that creates the parent connection to the signaling channel and the WebRTC connections.

9.2. As a Native Application

Running the LiquidRTC system in the browser has its downsides:

- The user has to have the landing page open in a browser window.
- The signaling methods are limited to the communication channels provided by browsers.
- Applications are limited by the restrictions set by the browser (for instance no way

to access local files).

Implementing LiquidRTC in a native application would fix all of these: The application could run in the background and listen to incoming applications without cluttering a browser session. The native LiquidRTC system could use more specialized signaling, for example, peers in the same local area network could find each other with UPnP or similar and completely eliminate the signaling server. The LiquidRTC system could provide more feature rich environment to the applications built on top of it.

Because the browser technologies, all the way from rendering HTML to WebRTC connection establishment are in the core of LiquidRTC, it is important these are still available in the native version. A framework called Electron [35] sounds like a perfect solution to this problem: it allows building applications as if they were run in a browser, while providing features such as file system access.

9.3. Networking

Different topologies can benefit networking in different situations. The choice of complete topology is mainly because it is easy to implement, and reliable for small networks. For larger networks, the number of connections becomes unreasonable, hindering the whole system performance. In theory, an adaptive mesh network could be a good candidate, but this needs further research.

The idea behind liquid software was hassle free multi-device roaming. For this end, an interesting use case can be for example a user who starts working on a computer at work, moves the application to their phone when leaving using the liquid network at the workplace. Once home, a different liquid network may await. Now, the user may wish to move the application started at work to a new liquid network that was not available at the time the application was first run. In this use case, a device may be a component in multiple liquid networks.

To make this kind of roaming possible, the only addition needed to the framework is to allow the landing page to reconnect to a new LiquidRTC network. Because the `liquidRTC` object is shared with all the applications, the new network would automatically be usable by all already open applications.

9.4. Heterogeneous Clients

It should be noted that WebRTC is not strictly limited to browsers. Therefore it is possible to create native applications that can communicate with rest of the liquid network.

Currently the only client in the LiquidRTC system is the landing page, but this is not necessarily the right choice. There could be WebRTC peers that act like servers within the network , allowing users to store application data and/or state. These server–like peers could be used to have persistent storage, allowing users to completely disconnect all their devices between uses, and not lose their applications or their state. To make use of this, the LiquidRTC protocol would also require the addition of pull LUE trigger to fetch the applications.

9.5. IndexedDB

Late at the development phase of LiquidRTC a web feature called IndexedDB API [36] was brought up. This is a modern web technology that allows saving objects in the browser in such a way they can be referenced in in HTML. Using this technique would allow sending each file of the application separately. This could possibly defeat the need for a program serializer, but some sort of processing would still need to be done to decide how to refer to the files stored in the IndexedDB. Another advantage of this approach would be that the applications that have been opened or received by the browser are saved and can be opened at a later time.

9.6. Authentication

In this implementation, the signaling messages are not authenticated. This means any peer can send any of these messages to the server. For example, C can send an offer to D stating the sender is A. When D receives the offer, it thinks its coming from A and forms the connection. It would be simple to add the capability for the signaling server to check the authenticity of the sender, but the to keep the implementation simple this was not done.

10. CONCLUSIONS

The goal of this thesis was to explore the use of WebRTC as a base for developing liquid software. At first, the ideas behind liquid software were studied and then the possibilities and limitations of WebRTC and the Web as a whole were scoped. To put WebRTC to the test, a proof-of-concept framework *LiquidRTC* was developed. The goal of this framework is to help developers make common web applications into liquid software with as little extra work as possible. Using the created framework a simple example application was liquified and its new liquid properties evaluated.

The *LiquidRTC* framework has shown that WebRTC is not only a viable option for liquid software networking, but due to its peer-to-peer nature, a highly flexible alternative to all previously tried methods. The novel application transfer and launch method allows an unforeseen way to work with user repositories for web applications.

WebRTC also has the advantage of being part of the Web ecosystem, which is probably the fastest growing platform for application development. With it easy, yet powerful tools are readily available for application developers to write their applications.

The implemented framework can work as a step towards better tooling for creating liquid software. While it is not ready for production work with its limited serialization support and security model, it does provide ideas and a proven networking system on which the next steps in the research can be taken.

There are still many unanswered questions about liquid software itself. One of the more critical ones is the one of security: How should liquid software authenticate and authrorize devices in its network. Before this question gets answered a wide deployment of liquid software is not possible.

BIBLIOGRAPHY

[1] Antero Taivalsaari, Tommi Mikkonen, and Kari Systä. Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. http://lively.cs.tut.fi/publications/LiquidSoftwareManifesto-2013.pdf, 2013. [Online; accessed 06-04-2017].

[2] Google docs. https://www.google.com/docs/about/. [Online; accessed 03-11-2018].

[3] Apple Continuity. https://www.apple.com/macos/continuity/. [Online; accessed 06-03-2018].

[4] Nextbit baton is real-time app data sharing, beta coming to cyanogenmod. https://www.androidcentral.com/nextbit-baton-real-time-app-data-sharing-beta-coming-cyanogenmo [Online; accessed 03-11-2018].

[5] WebRTC. https://webrtc.org/. [Online; accessed 23-02-2017].

[6] John Hartman, Udi Manber, Larry L. Peterson, and Todd Proebsting. Liquid software: A new paradigm for networked systems. http://dl.acm.org/citation.cfm?id=893566, 1996. [Online; accessed 16-12-2016].

[7] Andrea Gallidabino, Cesare Pautasso, Tommi Mikkonen, Kari Systa, Jari-Pekka Voutilainen, and Antero Taivalsaari. Architecting liquid software. *Journal of Web Engineering*, 16:433–470, September 2017.

[8] Neil Fraser. Differential synchronization. In *Proceedings of the 9th ACM Symposium on Document Engineering*, DocEng '09, pages 13–20, New York, NY, USA, 2009. ACM.

[9] Spotify. https://www.spotify.com/fi/. [Online; accessed 30-10-2018].

[10] Tommi Mikkonen, Kari Systä, and Cesare Pautasso. Towards liquid web applications. In *Proceedings of the 15th International Conference on Engineering the Web*

in the Big Data Era - Volume 9114, ICWE 2015, pages 134–143, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[11] Thomas A. Funkhouser. Network topologies for scalable multi-user virtual environments. In , pages 222–228, 1996.

[12] R. Bruno, M. Conti, and E. Gregori. Mesh networks: commodity multihop ad hoc networks. *IEEE Communications Magazine*, 43(3):123–131, March 2005.

[13] Overleaf. https://www.overleaf.com/. [Online; accessed 04-11-2018].

[14] Unity. https://unity3d.com/. [Online; accessed 04-11-2018].

[15] Chromebook. https://www.google.com/chromebook/. [Online; accessed 04-11-2018].

[16] Erika Doyle Navara, Theresa O'Connor, Ian Hickson, Silvia Pfeiffer, Robin Berjon, Steve Faulkner, and Travis Leithead. HTML5. W3C recommendation, W3C, October 2014. http://www.w3.org/TR/2014/REC-html5-20141028/.

[17] Sharedrop. https://sharedrop.io. [Online; accessed 29-10-2018].

[18] jump-game. https://github.com/rynobax/jump-game. [Online; accessed 29-10-2018].

[19] librtcdc. https://github.com/xhs/librtcdc.

[20] J. Uberti, C. Jennings, and Ed. E. Rescorla. Javascript Session Establishment Protocol. https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-24, 2017. [Online; accessed 31-03-2018].

[21] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, Anant Narayanan, Bernard Aboba, and Taylor Brandstetter. WebRTC 1.0: Real-time Communication Between Browsers. https://www.w3.org/TR/2017/CR-webrtc-20171102/, 2017. [Online; accessed 11-03-2018].

[22] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, RFC Editor, April 2010. http://www.rfc-editor.org/rfc/rfc5245.txt.

[23] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389, RFC Editor, October 2008. http://www.rfc-editor.org/rfc/rfc5389.txt.

[24] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, RFC Editor, April 2010. http://www.rfc-editor.org/rfc/rfc5766.txt.

[25] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, RFC Editor, June 2002. http://www.rfc-editor.org/rfc/rfc3261.txt.

[26] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566, RFC Editor, July 2006. http://www.rfc-editor.org/rfc/rfc4566.txt.

[27] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264, RFC Editor, June 2002. http://www.rfc-editor.org/rfc/rfc3264.txt.

[28] E. Rescorla. WebRTC Security Architecture. https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-14, 2018. [Online; accessed 12-07-2018].

[29] N. M. Al-Fannah. One leak will sink a ship: Webrtc ip address leaks. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5, Oct 2017.

[30] jQuery. https://jquery.com/. [Online; accessed 11-11-2018].

[31] Arun Ranganathan. FileSystem API. Technical report, W3C, 2016. https://w3c.github.io/filesystem-api/.

[32] import() proposal for JavaScript. https://github.com/tc39/proposal-dynamic-import, July 2018. [Online; accessed 04-10-2018].

[33] Larry Masinter. The "data" URL scheme. RFC 2397, RFC Editor, August 1998. http://www.rfc-editor.org/rfc/rfc2397.txt.

[34] Socket.io. https://socket.io. [Online; accessed 29-10-2018].

[35] Electron. https://electronjs.org/. [Online; accessed 06-11-2018].

[36] Ali Alabbas and Joshua Bell. Indexed database API 2.0. W3C recommendation, W3C, January 2018. https://www.w3.org/TR/2018/REC-IndexedDB-2-20180130/.

APPENDIX A. LIQUIFIED EDITOR

```
<html>
<head>
<title>Editortest</title>

<style>.progContainer {
  width: 600px;
  margin: 0px auto;
}

.noteBar {
  width: 200px;
  height: 300px;
  float: left;
}

.newNoteButton {
  height: 30px;
  width: 200px;
}

.noteList {
  height: 270px;
  width: 200px;
}

.editor {
  padding: 10px;
  border: 3;
  margin: 0px;
  height: 278px;
  width: 374px;
  float: right;
}

</style><script>
window.addLoadEvent = function(fn) {
  var oldonload = window.onload;
  if(typeof(oldonload) === 'function') {
    window.onload = function() {
      fn();
      oldonload();
    };
  } else {
    window.onload = function() {
      fn();
```

```
    };
  }
};

var sendProg = function(targetUser) {
  liquidRTC.peers[targetUser].send('program', my_source);
};

window.addLoadEvent( function() {
  var target_div = document.getElementById('framework_ui_div');
  var togglebtn = document.getElementById('toggle_button');
  var userlist = document.getElementById('user_list');
  // If no div is present, create one and add it to body
  if( !target_div ) {
    target_div = document.createElement('div');
    target_div.id = 'framework_ui_div';
    target_div.style.borderBottom = '0px dashed gray';
    target_div.style.backgroundColor = '#E0E0EE';
    target_div.style.height = '0px';
    target_div.style.transition = 'height .5s, padding .5s, border .5s, opacity .5s';
    target_div.style.marginRight = '45px';
    target_div.style.padding = '5px';
    target_div.style.paddingTop = '0px';
    target_div.style.paddingBottom = '0px';
    target_div.style.opacity = '0';
    target_div.style.overflow = 'hidden';
    target_div.style.borderBottomLeftRadius = "10px";
    target_div.style.borderBottomRightRadius = "10px";
    target_div.style.position = 'absolute';
    target_div.style.width = 'calc(100% - 70px)';
    target_div.style.boxShadow = '0px 10px 20px 0px #dde';
    document.body.insertBefore(target_div, document.body.firstChild);
  }
  if( !userlist) {
    userlist = document.createElement('select');
    userlist.id = 'user_list';
    userlist.multiple = 'true';
    userlist.style.cssFloat = 'left';
    userlist.style.width = '200px';

    target_div.appendChild(userlist);

    var btns = ['send', 'sync'];
    for (var i = 0; i < btns.length; i++) {
      var btn = document.createElement('button');
      btn.id = btns[i] + "_button";
      btn.innerHTML = btns[i];
      target_div.appendChild(btn);
```

```
    }
  }
  if( !togglebtn ) {
    togglebtn = document.createElement('div');
    togglebtn.style.width = '32px';
    togglebtn.style.height = '32px';
    togglebtn.style.background = 'url( [data]');
    togglebtn.style.backgroundColor = '#AAEEAA';
    togglebtn.style.borderRadius = '20%' ;

    togglebtn.style.cssFloat = 'right';
    togglebtn.style.clear = 'none';
    togglebtn.style.position = 'absolute';
    togglebtn.style.right = '10px';
    togglebtn.style.top = '10px';

    togglebtn.id = 'toggle_button';
    document.body.insertBefore(togglebtn, target_div);
  }

  var updateUserList = function() {

    while(userlist.firstChild) {
      userlist.removeChild(userlist.firstChild);
    }

    for(var user in liquidRTC.peers) {
      var opt = document.createElement('option');
      opt.innerHTML = user;
      opt.label = user;
      opt.value = user;
      userlist.appendChild(opt);
    }
  };

  document.getElementById('send_button').onclick = function() {
    console.log("click");
    for (var i = 0; i < userlist.options.length; i++) {
      var user = userlist[i];
      console.log("checking " + user.value);
      if(user.selected) {
        console.log("Sending");
        sendProg(user.value);
      }
    }
  };

  togglebtn.onclick = function() {
```

```
      if(target_div.style.height == '0px') {
        updateUserList();

        target_div.style.height = '';
        target_div.style.paddingTop = '5px';
        target_div.style.paddingBottom = '5px';
        target_div.style.borderBottom = '1px dashed gray';
        target_div.style.opacity = '1';
      }
      else {
        target_div.style.height = '0px';
        target_div.style.paddingTop = '0px';
        target_div.style.paddingBottom = '0px';
        target_div.style.borderBottom = '0px dashed gray';
        target_div.style.opacity = '0';
      }
    };
  });

  </script><script>
  //window.onload = function() {
  (function() {
    window.addLoadEvent(function() {
      liquidRTC = window.opener.liquidRTC;
      if(liquidRTC) {
        console.log("Got liquidRTC from parent");
      }
    });

    init_datasignal = function(fn) {
      liquidRTC = window.opener.liquidRTC;
      fn(liquidRTC);
    }
  })();
  </script></head>
  <body>
  <div class="progContainer">
    <div class="noteBar">
      <button class="newNoteButton">New note</button>
      <select class="noteList" size="20"></select>
    </div>
    <textarea class="editor"></textarea>
  </div>
  <script src="http://code.jquery.com/jquery-2.1.1.js"> </script>
  <script>$(function() {

    var notes = {};
```

```
var currentNote = undefined;

var liquidRTC = undefined;

if(typeof(init_datasignal) === 'function') {
  init_datasignal( function(lrtc) {
    liquidRTC = lrtc;
    liquidRTC.on('editor-state', function(msg) {
      update_note(msg.note, msg.msg);
    });
  });
}

var createNote = function(n) {
  notes[n] = "";
  $('.noteList').append($('<option />').html(n));
};

var selectNote = function(n) {
  currentNote = n;
  $('.noteList').val(n);
  $('.editor').val(notes[n]);
};

var syncNote = function (n) {
  var msg = {
    note: n,
    msg:  notes[n]
  };
  if(typeof liquidRTC !== 'undefined') {
    liquidRTC.broadcast('editor-state', msg);
  }
};

var update_note = function (n, t) {
  if(notes[n] === undefined) {
    createNote(n);
  }
  if(currentNote == undefined) {
    selectNote(n);
  }
  notes[n] = t;
  if(n == currentNote) {
    $('.editor').val(notes[currentNote]);
  }
};

$('.editor').on("change keyup paste", function() {
```

```
      if(notes[currentNote] != $(this).val()) {
        notes[currentNote] = $(this).val();
        syncNote(currentNote);
      }
    });

    $('.noteList').change( function() {
      currentNote = $('.noteList').val();
      $('.editor').val(notes[currentNote]);
    });

    $('.newNoteButton').click(function() {
      var newNoteName = prompt("Name for new note:");
      if(newNoteName == null) {
        return;
      }
      createNote(newNoteName);
      syncNote(newNoteName);
      selectNote(newNoteName);
    });
  });
  </script>
  </body></html>
```