



NTNU – Trondheim
Norwegian University of
Science and Technology

Geo-enhanced routing in DHT with WebRTC

Using WebRTC to build geographically aware
Distributed Hash Table between browsers.

Rolf Erik Heggem Lekang

Master of Science in Computer Science

Submission date: June 2015

Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Geo-enhanced routing in DHTs with WebRTC

Using WebRTC to build geographically aware
Distributed Hash Tables between browsers.

Author:

Rolf Erik H. Lekang

Supervisor:

Svein Erik Bratsberg

Department of Computer and Information Science
Norwegian University of Science and Technology
2015

Abstract

Peer-to-peer systems have earlier made it possible to do more with the networks available than the typical server-client architecture allows. In the last few years, websites have developed from very static pages to complex dynamic applications. The web continues to evolve and WebRTC introduces peer-to-peer communication to the browser and the web applications we run in these browsers. The web applications can utilize this technology in different ways. This project moves a known peer-to-peer algorithm, Chord, to the browser as a proof-of-concept. Moreover, it looks into using geographical location data to enhance the routing in a Distributed Hash Table (DHT). This project investigates the how to build a peer-to-peer system between browsers and how to test the system during development and after. The project proposes to use simulation and a scaled test with real browsers to test WebRTC based peer-to-peer applications. In this project, the browsers ran within Docker containers in data centers around the world. Both simulations and tests with real browsers can be used in development as well as evaluation testing. The Docker based real browser tests were helpful in indicating bugs, however, the environment were hard to debug. The results from the evaluation tests indicate that the routing enhanced with knowledge of geographical locations gives a boost in routing performance.

Sammendrag

Peer-to-peer systemer har tidligere gjort det mulig å oppnå mer med nettverkene som er tilgjengelig enn det man får til med en klient-tjener arkitektur. De siste årene har man sett nettsider utvikle seg fra statisk innhold til avanserte dynamiske web-applikasjoner. Utviklingen fortsetter, og WebRTC introduserer peer-to-peer kommunikasjon mellom nettlesere og web applikasjonen vi kjører i disse nettleserne. Web applikasjonene kan utnytte denne teknologien på forskjellige måter. Dette flytter en kjent peer-to-peer algoritme, Chord, til nettleseren som et proof-of-concept. I tillegg ser prosjektet på bruk av lokasjonsdata til forbedring av rutevalg i en distribuert hash-tabell. Videre, undersøker prosjektet hvordan man kan utvikle et peer-to-peer system mellom nettlesere og hvordan man kan teste et slikt system. Prosjektet foreslår å bruke simulasjoner og skalerte tester med ekte nettlesere til testing av peer-to-peer systemer. I dette prosjektet kjørte de ekte nettleserne i Docker containere i datasentre omkring i verden. Både simulasjonene og testene mellom flere nettlesere kan bli brukt under utvikling og til evalueringstesting. Testene med ekte nettlesere i Docker var nyttig til å indikere feil eller dårlig feilhåndtering, men det var utfordrende å feilsøke i det miljøet. Resultatene fra evalueringstestene indikerte at ytelsen av systemet øker ved bruk av geografiske lokasjonsdata til å forbedre rutevalg.

Acknowledgement

I would like to thank Professor Svein Erik Bratsberg at the Department of Computer and Information Science at Norwegian University of Science and Technology. He has guided the project and given valuable feedback along the way.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem definition	2
1.3	Readers manual	2
2	Background	3
2.1	Peer-to-peer systems	3
2.1.1	Structured vs Unstructured peer-to-peer systems	4
2.2	Distributed Hash Tables	5
2.2.1	Consistent Hashing	5
2.2.2	Churn	6
2.2.3	Chord	6
2.3	Web technology	10
2.3.1	Geographical location	10
2.3.2	WebSocket: real-time communication between client and server	11
2.3.3	Web Storage	11
2.3.4	Offline mode	12
2.4	JavaScript	13
2.4.1	Asynchronous code in JavaScript	13
2.4.2	Promise pattern	13
2.4.3	Prototype object	15
2.4.4	Node.js and io.js	16
2.4.5	Support in browsers	16
2.4.6	Browserify and CommonJS	17
2.5	Docker	17

2.5.1	Containers and virtual machines	17
2.5.2	Docker as a tool in research	18
2.5.3	Docker registry	19
2.5.4	Headless browsers within docker	19
2.6	WebRTC	20
2.6.1	Media communication	20
2.6.2	Data communication	20
2.6.3	Signaling	20
2.6.4	Browser support and frameworks	21
2.6.5	Testing WebRTC applications	23
3	Design	27
3.1	Levels of abstractions	27
3.2	Chord on WebRTC	28
3.2.1	Structure	29
3.2.2	Routing	30
3.3	Changes in Chord	30
3.4	Error handling	32
3.5	Testing and simulation	33
3.5.1	Logging and testing utilities	35
3.5.2	Dashboard	36
4	Experiments	39
4.1	Simulations	39
4.1.1	Dataset generation	40
4.1.2	Experiment setup	40
4.1.3	Retrieval simulation	40
4.2	Real browser experiments	44
4.2.1	Experiment setup	44
4.2.2	Retrieval experiment	44
5	Discussion	47
5.1	Geographical enhanced routing	47
5.2	Building distributed systems with WebRTC	48

<i>CONTENTS</i>	xi
6 Conclusion	51
6.1 Further work	53
A Software packages	59
A.1 peerjs-rpc	59
A.2 peerjs-mock	60
A.3 peerjs-rpc-mock	61
A.4 rpc-dashboard	62
B List of technologies	65

List of Figures

2.1	The highlighted part of the circle shows the key-space assigned to peer N2	7
2.2	The two figures shows the lookups necessary to find an item stored on N8 from N1	7
2.3	A permission prompt shown to ask for the users approval of geographical location access.	10
2.4	Example code showing the difference of promises and callbacks .	14
2.5	Sequence diagram for a connection process in WebRTC	22
3.1	The development stack of Chord implementation.	28
3.2	The relations between prototype objects.	29
3.3	The two figures shows the lookups necessary to find an item stored on N8 from N1	33
3.4	The RPC dashboard	36
4.1	Cumulative timings of 1000 different key location lookups with evenly distributed geographical location	41
4.2	Cumulative timings of 1000 different key location lookups with random geographical location	42
4.3	Cumulative timings of key location lookups with 40 nodes random geographical location	43
4.4	Average query time queries of different bit distances	43
4.5	Cumulative timings of key location lookups with 40 running in docker containers	45
A.1	Sequence diagram for a sending process in peerjs-mock	60
A.2	Example of using peerjs-rpc-mock directly	61
A.3	Example of using peerjs-rpc-mock with require-mock	62
A.4	Example of using rpc-dashboard	63

List of Tables

4.1	Data on round trip times from [1]	40
4.2	Relevant data from simulations with 1000 nodes	42

List of Acronyms

API Application Programming Interface.

CDN Content Delivery Network.

CI Continuous Integration.

CPU Central processing unit.

CSS Cascading Style Sheets.

DDoS Distributed Denial-of-service attack.

DHT Distributed Hash Table.

DOM Document Object Model.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

JSEP JavaScript Session Establishment Protocol.

JSON JavaScript Object Notation.

NAT Network address translation.

OS Operating System.

REST Representational State Transfer.

RPC Remote Procedure Call.

RTT round-trip time.

SLA Service Level Agreement.

TTL Time To Live.

VoIP Voice over IP.

W3C World Wide Web Consortium.

WebRTC Web Real-Time Communication.

Xvfb X virtual framebuffer.

1 | Introduction

Peer-to-Peer systems have been used for multiple purposes for several years. However, most peer-to-peer systems to this date are developed to run directly on an Operating System. WebRTC introduces a way to build peer-to-peer systems within the browser. Thus, the browser is a sandboxed environment suitable for peer-to-peer applications. WebRTC is divided in two separate use cases, streaming media or sending data. To this day, WebRTC implementations mostly utilize the media-channel. There are existing solutions for video-conference calls and streaming of live video. However, the data-channel of WebRTC has had a lack of attention. The attention is picking up as the WebRTC specification and libraries around it is further developed. Thus, more proofs-of-concepts are built to show the benefit from using WebRTC to transfer data.

Most of the proof-of-concepts that have surfaced based on the WebRTC data channel focuses on privacy, e.g. file-sharing applications that allow users to send files directly to each other without storing the files on a server. Furthermore, there is new text chat application, called friends¹, which communicate between the people using it with WebRTC.

Web Real-Time Communication (WebRTC) will probably change the way users of applications on the web interacts with each other. It is a young technology with a lot of promise. This research project investigates the possibilities of the technology with a proof-of-concept implementation of a known peer-to-peer application, the Chord Distributed Hash Table (DHT). Moreover, the project proposes to use the geographical location API in modern browsers to enhance routing performance. The research has a focus on testing and evaluation of WebRTC based peer-to-peer applications.

1.1 Motivation

There are two main motivations for this research project. The first is to create a proof of concept of a peer-to-peer application with WebRTC between web-browsers, from which requires research on different implementations of WebRTC

¹<http://moose-team.github.io/friends/>

and frameworks created for WebRTC and research on how to test the peer-to-peer application. The second motivation is to look into how geographical location might be used to enhance routing in peer-to-peer applications. Both of these points are related to how the web is evolving with client side applications that in some use cases could benefit from using peer-to-peer communication, and furthermore, could benefit from higher performance based on geographical location.

1.2 Problem definition

Research the possibilities for creating distributed systems with WebRTC by implementing a proof of concept as a Distributed Hash Table. Furthermore, use that implementation to check if knowledge of geographical location can help enhance the routing performance. A part of the research of the use of the WebRTC stack to build distributed peer-to-peer applications is to find good strategies for testing the application in terms of correctness of functionality and performance wise.

RQ1 Is WebRTC suitable for a DHT algorithm, such as Chord?

RQ2 Can the geographical location enhance routing performance in a Distributed Hash Table?

RQ3 How to build a reasonable test environment for a peer-to-peer application built with WebRTC?

1.3 Readers manual

The rest of this report is divided into five chapters. Chapter two describes background knowledge and the state of the art of each concept and technology that is relevant to this project. Chapter three describes the design of the WebRTC version of Chord developed in this project. Chapter four describes the experiments performed in this project. The design of the experiments, the execution of the experiments and the results of the experiments. Chapter 5 discusses these results, limitations and similar discoveries of the project. Chapter 6 concludes the information in this report and the results from the research experiments. In addition to those chapters, the appendices contain relevant information that is not directly in the scope of the project. Appendix A contains information about Open-Source JavaScript modules created for this project and their documentation. Appendix B contains a list and short descriptions of technologies, frameworks and modules used in this project.

2 | Background

2.1 Peer-to-peer systems

Peer-to-peer systems evolved from the rapid growth of the internet and computer networks. A peer-to-peer system embraces the use of a large quantity of computing nodes requiring the same information to share that information[2]. In a peer-to-peer system, each peer that needs to send or receive information communicates directly, or indirectly through a similar node, with the node it sends to or receives from.

One of the best-known types of peer-to-peer applications is Voice over IP (VoIP), which makes it possible to call directly peer-to-peer. This kind of services are very popular for video calls, and Skype[3] one of the best-known peer-to-peer calling services as well as one of the best-known peer-to-peer applications. Nevertheless, the peer-to-peer architecture has been utilized in different ways. The applications includes file-sharing[4], web-caching[5][6], distributing sharded information[7][8][9] and games[10].

Peer-to-peer communication has been utilized when there is no need for a central server or a central server is not wanted in the architecture. There are several reasons why a central server architecture is unwanted. One of those reasons is avoiding a single point of failure, which could be achieved by having several sites with central servers. However, there are situations when that is not feasible because the resources might not be available. Moreover, peer-to-peer communication can be a viable option if there is critical that computing nodes continue to work and coordinate when the central server goes down.

Furthermore, there are certain applications that would benefit from not having a central location, which handle communication and coordination. One use-case that has had problems because of the central architecture is free speech movements. In addition, platforms that enable free speech movements to do important work are target of DDoS attacks and blockage in government controlled networks. Events in recent years have shown that there are those who will go to those lengths in order to shut down and suppress free speech move-

ments¹. Furthermore, it has been seen that services like Twitter has played an important role in work against oppression. However, they can be easily blocked by a governmental firewall, because of its architecture based on central servers.

On the other hand, creating tools to avoid governmental firewalls has its negative factors. In many countries, the governmental firewall is used for good purposes. They stop criminality and other bad things from happening. However, peer-to-peer systems exist and most of those who want to avoid governmental firewalls with peer-to-peer technology can already do that. Furthermore, creating peer-to-peer technologies that work inside web pages makes it more accessible especially for people with lower resources which free speech movements.

Transferring substantial amounts of data is another situation where peer-to-peer communications are more applicable than a server-client architecture. In a server-client architecture file transfer between two clients can be slow, because the files need to be uploaded to the central server from the first client before the second client can download it from the server. Thus, it takes more time and uses more of the network resources than it would if the file were transferred directly. This is especially true if the clients are sharing files are in closer proximity together, in terms of network latency, than the clients are with the server. Furthermore, if they are on the same local network transferring files peer-to-peer would be faster than transferring through a server. Companies who deliver content with a large footprint in file size(e.g. operating systems, games and game updates) often utilizes peer-to-peer technologies to deliver their content. The cost of delivering downloads of updates to a game with high-quality graphics can be significantly lower when using peer-to-peer communication to use shared bandwidth and computing power.

Nonetheless, peer-to-peer technologies have been fundamental to online piracy and other criminal activities. Especially peer-to-peer file sharing is widely used to share files illegally without regards for ownership rights and licensing of materials. However, developing technologies like peer-to-peer file-sharing enables the society to use the existing network to deliver services and content that would not be possible at an affordable level without it. There is little indicating that not having peer-to-peer technologies would stop online piracy. Thus, it is possible to argue that the technology enables more good than harm. Furthermore, the legal services that provide content that compete with the piracy channels in the digital world depend on the same technology.

2.1.1 Structured vs Unstructured peer-to-peer systems

There are two types of peer-to-peer systems, structured and unstructured systems[2]. Both have advantages and disadvantages that affect when it is appropriate to

¹The attack on Github(<https://www.eff.org/deeplinks/2015/04/china-uses-unencrypted-websites-to-hijack-browsers-in-github-attack>) and the blockage of Twitter in Turkey(<http://www.bbc.com/news/world-europe-26677134>) are recent examples.

use either one. It exists well known implementations of structured systems, e.g. Chord[7] and Dynamo[8], and unstructured systems, e.g. Squirrel[5] and Gnutella[6].

The structured peer-to-peer systems have a policy that governs node distribution and the general topology of the peer-network. In those kinds of systems, there are guarantees of correct node lookup and a bound on the time complexity. Furthermore, the policy lowers the message overhead, because of the rules of the network topology the amount of messages needed to lookup a node is lower compared to a system without such rules. However, the rules and enhanced routing comes at a cost. Such systems need to maintain information about parts of the network. Each node needs to know something about other nodes and that information needs to be maintained according to the policy to perform as the system guarantees.

On the other hand, there are unstructured peer-to-peer systems. Compared with structured systems it does not have the same level of topology maintenance since the system is self-organizing. In particular situations with high churn, the lowered maintenance cost can be necessary to avoid that the peers use a significant amount of computing resources for maintenance. Furthermore, the self-organizing feature makes an unstructured system more resilient to node failure because it is designed to depend on certain policies. However, this can result in routing errors and flooding of network messages. Since there is no clear policy on the network topology, the system can not deliver a guarantee of routing correctness or routing performance. In some system design, this can be a downside, especially systems that need to honor a SLA. However not all systems needs those guarantees. Unstructured systems are appropriate whenever guarantees of delivery and network consumption can be traded for a more churn resilient system.

2.2 Distributed Hash Tables

2.2.1 Consistent Hashing

Consistent hashing is a technique in order to map keys and node IDs to a circular key-space. The circular key-space ranges from the minimum hash value to the largest hash value applicable to the given key-space. The hash is calculated with a hashing function, which is deterministic, in order to map a key or the node id to a hash-key within the circular space.

The circular key-space needs to have a range of at least m bits in order to avoid key-collision. Karger et al.[11] proved the following: *"For any set of N nodes and K keys, with high probability: (1.) Each node is responsible for at most $(1 + \epsilon)K/N$ keys and (2.) when an $(N + 1)^{st}$ node joins or leaves the network responsibility of $O(K/N)$ keys changes."* In this setting the term *high probability* is given that there is a good distribution on the hash values used.

There are multiple strategies that would result in a good distribution in most cases. However, there will always be some unique cases that will result, in the worst case, in a skewed distribution. In such a situation, the statement above will not be valid.

2.2.2 Churn

In peer-to-peer systems, Distributed Hash Table especially, Churn is a concern for the stability of the system and the network created by the system. Churn refers to Churn rate, which is the rate a measurement for the amount of peers a system [12]. There are different reasons in play when peers leave a peer-to-peer network. In terms of a system designed for maximal uptime like a peer-to-peer based database(e.g. Cassandra), the prominent reasons are peer failure, network failure or network isolation. On the other hand, systems like file-sharing networks are prone to peers leaving the network because their users decide to leave or choose to turn off the host machine. Hence, the probability of a high Churn rate differ significantly from system to system. Moreover, different systems adopt different strategies for recovering after nodes leave the network. There are two primary strategies for detecting nodes that have left the network for some reason. The first is by using a heartbeat signal, where the each peer sends a message to the other peers that are interested to know about the given peer. The other is by pinging, where the interested peers ping the given peer to check if the peer is accessible.

2.2.3 Chord

Chord is a Distributed Hash Table built on the concept of consistent hashing. [7] It uses a circular key-space of m -bit size. The keyspace starts at the lowest sha1 hash and ends on with a hash of the start hash plus m bits. It uses the circular key-space and the features of consistent hashing to guarantee correctness in routing of the peer network. The Chord algorithm works in two different modes, simple key location and scalable key location. The difference of the two nodes affects lookup time. The simple key location has N messages worst case lookup effort while scalable key location uses $\log N$ messages in the worst case. In both cases, N is the number of nodes in the network. The differences between those two approaches are described in more detail in sections later on.

Each peer has the responsibility of a range of that key-space, that range stretches from the peers predecessors key to the peers key. Figure 2.1 shows the key-space of peer N_2 highlighted. The key-space ring is self-preserved in the way that there is no central organization of the ring. Each node has a hash value and by asking known nodes for its successor the node finds the correct place in the ring. Chord runs maintenance tasks at regular intervals to maintain the successor and predecessor connections when new nodes join the ring. Thus, there is no need for a joining node to announce itself directly after joining as

the stabilize task will announce it to the correct node eventually. The stabilize task runs at regular intervals on each node and makes sure that each node that should know of the new node knows.

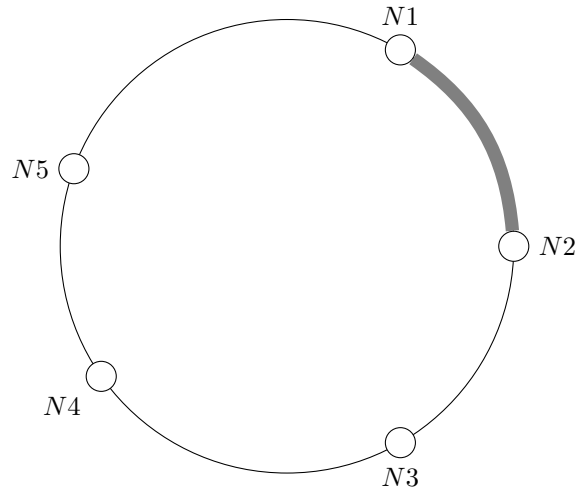


Figure 2.1: The highlighted part of the circle shows the key-space assigned to peer N2

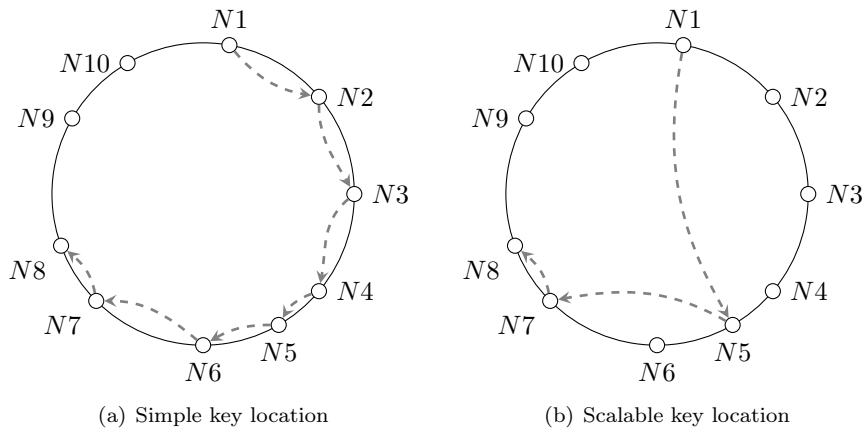


Figure 2.2: The two figures shows the lookups necessary to find an item stored on N8 from N1

Simple key location

This is the first of two key location lookup algorithms. It is slower than the *Scalable key location* lookup in terms of number of nodes lookup necessary to get from a key location to another. It has a worst case scenario of $O(N)$ lookups. On the other hand, it requires very little maintenance since each node only needs to know its successor because a correct lookup is guaranteed if each node knows its successor.

Algorithm 1 Successor lookup of key in simple key location

```

function findSuccessor(hash)
  if  $hash \in (n, successor]$  then
    return  $successor$ 
  else
    return  $successor.findSuccessor(hash)$ 
  end if
end function

```

The successor lookup in this location strategy is listed in pseudocode in figure 1. It checks whether itself has the range in which the key belongs, if not it requests the successor to do the same through a RPC-call. Thus, when the right node is found, the RPC-call will return the value.

This approach is not scalable and will only be suitable for a small number of nodes. However, if the number of nodes never exceeds the given threshold that fits the performance requirements, it could be a valid trade-off in order to avoid extra maintenance. Furthermore, if having less information per worker is more valuable than enhanced performance this approach could still be a suitable choice.

Scalable key location

Simple key location can result in N number of lookups with N nodes, which might result in a low performance. In order to work around this the number of lookups needs to be significantly less with high numbers of nodes. The scalable key location is a strategy to lower the nodes that need to be asked in a location lookup.

The scalable key location maintains a finger table at each node with information about knowledge of certain nodes in the ring. It has an entry for 2^n bit increase in the key, which means that entry n in the table refers to the key of the node, holding the finger table plus 2^n bits. The finger table makes it possible to lookup farther away in the keyspace without asking each node in between. This will significantly lower the number of nodes a node need to ask to lookup a key location. It will lower the worst case from $O(N)$ to $O(\log N)$ in a network of N nodes.

Algorithm 2 Successor lookup of key in scalable key location

```

function n.closestPrecedingNode(hash)
  for  $i = \text{finger.length} - 1$  to 1 do
    if  $\text{hash} \in (n, \text{successor}]$  then
      return  $\text{finger}[i]$ 
    end if
  end for
  return  $n$ 
end function

function n.findSuccessor(hash)
  if  $\text{finger}[i] \in (n, \text{successor})$  then
    return successor
  else
     $n' = \text{n.closestPrecedingNode}(\text{hash})$ 
    return  $n'.\text{findSuccessor}(\text{hash})$ 
  end if
end function

```

The jump increases the farther along the ring the node is placed, thus for each entry in the finger table the keyspan between the referring and the referred node increases. Since the difference is in a power of two, the range will double for each entry in the finger table. This has several benefits: Looking up closer node has a higher probability of reaching the correct node with one lookup, which is good because then it is possible to lookup approximately correct node and get from that node to the correct with a few more lookups. The benefit of increasing the range for every step farther away is lowered maintenance. Thus, it is a trade-off between detailed knowledge and maintenance, and because a node is more likely to look up a closer node more often it is acceptable with a higher level of maintenance closer to the node. This strategy inherits the guarantees of *simple key location* because the first row in the finger table will always contain the successor current node.

The finger table is continuously updated and maintained. A task, which runs periodically, updates an entry in the finger table by trying to lookup the key of that place in the finger table.

Algorithm 3 The maintenance functionality for the finger table

```

function n.fixFingers
   $\text{next} = \text{next} + 1$ 
  if  $\text{next} \geq \text{finger.length}$  then
     $\text{next} = 1$ 
  end if
   $\text{finger}[\text{next}] = n.\text{findSuccessor}(n + 2^{\text{next}-1})$ 
end function

```

2.3 Web technology

The client side of the web technology stack consists of HTML and ECMAScript, in addition, there is the APIs defined by browsers that are an extension of the HTML specification and the ECMAScript specification. ECMAScript is a scripting language defined by the ECMA-262 specification[13]. There are different implementations of ECMAScript, the most popular and the one with widest support in browsers is JavaScript, see section 2.4 for more background on JavaScript. The HTML 5 specification adds numerous features to the web technology stack. A few of those features are relevant and can be useful for distributed systems in the browser. The sections below introduces the most relevant ones.

2.3.1 Geographical location

The Geolocation API defines a way to retrieve the geographical location of a browser[14]. Its specification defines the structure of the API, by which it is possible to retrieve the current location and updates whenever the location changes. Modern web clients can range from watches to desktop computers. Thus, the location of the device might change while the application depending on geographical location is used, which might require the client using the geographical data to listen for change events. Moreover, in order to retrieve the geographical location, the user needs to approve the access. This is in most modern browsers implemented as a prompt-alert as shown in figure 2.3

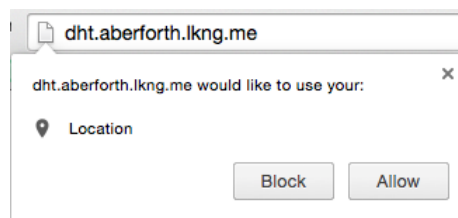


Figure 2.3: A permission prompt shown to ask for the users approval of geographical location access.

It is possible to retrieve longitude, latitude and accuracy from the API. Those data make it possible to pinpoint the location of the machine the browser is running on. Furthermore, with two locations it is possible to calculate the distance in between the two with Lambert's formula. It calculates the spherical distance based on the radius of the earth. Based on the formula and the data types available in JavaScript the precision of the distance will be on the meter scale, thus comparing in kilometer scale should be accurate given that the geographical location can be detected with acceptable accuracy.

2.3.2 WebSocket: real-time communication between client and server

Communication between client and server through HTTP has a significant overhead, by which real-time communication will suffer. There are three different ways of handling real-time communication between client and server: HTTP polling, HTTP long-polling and WebSocket[15]. The two first work by sending HTTP requests to the server when the client application assumes that there is some new information. The difference between the two is the frequency of the polling. The polling of information that might be ready requires HTTP requests on regular intervals, which can be effective and a good solution if the data requested is accessible on regular intervals. In the case when the data becomes available sporadically and in an unpredictable manner, polling will waste resources in the form of network traffic and increased latency. WebSocket on the other hand, is designed for the latter use case. WebSocket is a full-duplex communication between client and server. Thus, when the connection is created, both parties can send information to the other party. This can be useful for notifying the client of new data or regularly sending data one or both ways or streaming data, e.g. logs, from the server to the client. A W3C specification[16] defines the WebSocket API and communication.

2.3.3 Web Storage

The Web Storage API specification defines an interface for accessing and storing data in web clients outside cookies[17]. There are two types of storage defined in the API, which are `SessionStorage` and `LocalStorage`. Both are created to address needs that cannot be addressed by using cookies.

In addition to the Web Storage API, there is another way to store information in web clients: cookies[18]. Cookies have been a way to store state in web browsers for more than two decades. Cookies are accessible both to the client and server side as they are attached to all HTTP requests. The original use case for cookies was to create a shopping cart for e-commerce websites. The requirements for storing of state in web browsers has since grown into many different use cases. There are several use cases where cookies will affect the web application in a negative manner. There is also a security concern when using cookies, especially over an unencrypted connection. Since the cookie is sent with every request it is possible to listen to network communication, from which it is possible to get cookies of other web users on the network. The Web Storage API, makes it possible to avoid this in the cases where the state is only useful in the client and there is no need to send it to the server with each request or to send it to the server at all.

The first storage in the Web Storage API, `SessionStorage`, fills the need for storage in the client that is accessible just to the current session. Isolating a storage to the current session can be important in applications that a user would,

with high probability, use in several windows or tabs simultaneously. Especially if the data stored by one session might interfere with another session and not behave as the user expects.

The second, `LocalStorage`, is a persistent storage that can store large amounts of data, compared to `SessionStorage`. `LocalStorage` is not isolated to the current session. Thus, all sessions can access the data at any time. Furthermore, the data will not be removed before it is deleted by the application or the user. The data will not be transferred to the server with each HTTP request as opposed to how cookies works. This gives the ability to save larger amounts of data without slowing down HTTP requests the server.

Both storage types are applicable as a storage backend in a distributed hash table implementation. However, there is positive and negative consequences of using either of them. The session-based storage will loose all data when a session ends, which makes failure recovery and re-joins more complicated. However, a re-join would be like a normal join and would not require handling stale data. `LocalStorage` will have persistent data between sessions, and thus it will be able to serve data from earlier sessions. This makes the impact of each failure and re-join less in terms of resources used to synchronize data. Furthermore, the data will be accessible to all instances running in the same browser, which would make it insufficient to open a new browser window in order to start another virtual node. However, if it is a requirement that the data should only live for one session data can be stored with the session id as key prefix. Storing with a session prefix would also solve the problem of running several instances in different windows in the same browser. Thus, with application logic it is possible to achieve the features of `SessionStorage`, which make the `LocalStorage` more applicable.

2.3.4 Offline mode

Application cache manifests enables web application to run even when the browser is offline. This feature can be useful for several purposes in a peer-to-peer system. It allows applications that do not need a connection to the server that hosts the application to work. Hence, the user can visit a web application one time and still use it even if the server is down as long as it has the necessary data. In the peer-to-peer setting with WebRTC, this means that a group of peers can create a peer-to-peer network if they have the application in their browser and can configure signaling servers(see section 2.6.3).

2.4 JavaScript

JavaScript is an implementation of the ECMAScript specification. The language was initially developed by Brendan Eich. It is a dynamically typed programming language and also defined as a prototype-based scripting language. In today's browsers, it is the de-facto standard for client side programming on the web stack[19] as it has been for several years. Moreover, JavaScript can run both in a web browser and directly in an OS. Not only can the language run in both situations, but in various cases the same code base can both run in a browser and on the directly in the Operating System.

2.4.1 Asynchronous code in JavaScript

In JavaScript, there is often need to write asynchronous code. Especially when one works with network requests or IO operations. Until a few years back there was one way to handle asynchronous in JavaScript. That was callbacks, which can be achieved by sending callback functions as an argument to other functions[20]. The callback function should take the of the asynchronous operation as arguments. Thus, the function doing the asynchronous operation can call the callback function when it is done. This is a good pattern when you have one code block waiting on another. However, if there is several asynchronous operation that needs to be performed in a given order the nesting of callback can become large. Nesting in several layers with asynchronous dependencies can be error prone and make the code hard to read. In the callback pattern, there is the need for error handling in each callback function, from which wrong handling of errors can occur. Furthermore, if there is no interest in handling errors at any level lower than the outermost level, there would be a need for code that populates the errors up to the last level.

2.4.2 Promise pattern

Promises is another pattern for handling asynchronous code in JavaScript. It builds on the notion of a chain of events that return either a value or a promise of a value. A promise can either be resolved with a value or rejected with an error. All errors thrown inside a promise chain will reject the promise, and the rejection is passed down the chain. Thus, they are only handled in the place in the chain where it is appropriate. If an error is thrown, the chain is broken, and the next error catch function is called. Furthermore, a function calling another function that returns a promise can return the promise. Thus, if there is no need to handle the promise within the function, it can be returned. Besides, promise libraries have the functionality to wait for several promises to return and then resolve a list of return values.

Figure 2.4 shows the differences between a callback approach and a promise-

```
1  // callback version
2  function loadWithCallback(arg1, callback) {
3      asyncOperation1(arg1, function (error, result) {
4          if (error) return callback(error, null);
5          asyncOperation2(result, function (error, result2) {
6              if (error) return callback(error, null);
7              asyncOperation(result, function (error, result3) {
8                  if (error) return callback(error, null);
9                  callback(result3);
10             });
11         });
12     });
13 }
14
15 load(function(error, result) {
16     if (error) {
17         // handle error
18     } else {
19         // handle success
20     }
21 });
22
23 // promise version
24 function loadWithPromise(arg1) {
25     return asyncOperation1(arg1)
26         .then(function(result) {
27             return asyncOperation2(result);
28         })
29         .then(function(result) {
30             return asyncOperation3(result);
31         });
32 }
33
34 load
35     .then(function(result) {
36         // handle success
37     })
38     .catch(function(result) {
39         // handle error
40     });
```

Figure 2.4: Example code showing the difference of promises and callbacks

based approach of a load function that contains three ordered asynchronous. The promise code is more readable and has less boilerplate code for error handling. Line 4,6 and 8 only passes the error on correctly and makes sure that the rest of the callback does not run if an error occurs, which the promise chain handles in the promise example.

Several test frameworks in JavaScript has built-in support for promises. This makes it possible to return a promise within a test function. The test will then wait for the promise to resolve before concluding the test. One of the nice features of returning a promise in a test is failure handling. If the promise chain throws an error and it is not caught before the end. The test framework will mark the test as failed.

Benchmarks In discussions on the topic of callbacks versus promises, performance has been a downside of using promises. The promise library called bluebird changed that. Bluebird is efficient in both memory consumption and the time it takes to perform x operations. The performance of bluebird have been compared to other promise libraries and callbacks in a benchmark.² Bluebird is insignificantly slower than callbacks, however, significantly closer to the performance of callbacks than any other promise implementation.

2.4.3 Prototype object

In JavaScript, there is only one construct, Object. All types derive from an object. Thus, implementing inheritance and class like data structures must be based on objects. The closest concept to a class that is available in JavaScript is prototype objects. Each object has an internal link to a prototype. The prototype object also has an internal link to a prototype object, unless it is null. Thus, it is possible to create inheritance with prototype objects.

The prototype chain is a chain of linked prototype objects which ends with null. All prototype objects is a set of properties as with regular objects. Lookups of properties in an object first checks if the object has an own property with the given name before checking the prototype chain closest to farthest away. Moreover, since objects store functions as properties, this makes it possible to inherit and override functions by extending the prototype chain.

ES6, the next version of ECMAscript, introduces a new keyword to the language specification: class. It is syntactical sugar for the process of extending objects and prototype chains to achieve inheritance.

²The benchmark mentioned can be found on <https://github.com/petkaantonov/bluebird/blob/f114841282193642484ddb8dc315fc45355bbdb0/benchmark/README.md> (retrieved 26.04.2015)

2.4.4 Node.js and io.js

Node.js is a platform, on which it is possible to run JavaScript directly in a operating systems instead of in a browser[20]. This enables creation of server side JavaScript applications, as well as, running JavaScript on a local computer. Node.js is based on Chromes V8 JavaScript engine. Thus, the same language interpreter is used in Google Chrome and Node.js.

Furthermore, the framework is built around the event based pattern described in section 2.4.1, from which all I/O and network operations are asynchronous and event based. There are, however, sync wrappers around several functions(e.g. `readFile`³ has `readFileSync`⁴). These synchronized versions blocks Node.js' event loop in order to make the code synchronous. Thus, all other asynchronous code will have to wait for the sync operation before they can call their callback functions or resolve their promises. The blocking of the event-loop can therefore result in significant holdup and slow down in concurrent code.

The Node.js ecosystem is built around npm, which is a packet manager for Node.js and io.js. It is not restricted to Node.js packages or JavaScript for that matter. However, it is created for JavaScript packages for Node.js/io.js and Browserify(see section 2.4.6). The require functionality in Node.js will look up JavaScript modules in `node_modules`, which is where npm install dependencies, or local files with a specified path. Dependencies in Node.js is hierarchical, which means that if a module depends on another module it will be installed in `node_modules` inside the module, e.g. if `a` depends on `b` then `b` would be installed in `node_modules/a/node_modules/b/`. The hierarchical dependencies makes it possible to have several packages that depend on different versions of the same package without getting conflicts or broken modules.

io.js is a fork of Node.js[21]. A few of Node.js' core contributors decided to create a fork of Node.js because they were unhappy with the status quo in the development of Node.js. The development of Node was, and is still, lagging behind on the support of newer versions of the V8 engine. Furthermore, one of the goals of io.js is to have a predictable release cycle that strictly follows semantic versioning.⁵ The upgraded V8 version and faster releases with bug fixes and performance increases makes io.js a more suitable choice in some situations. Especially when performance is key to application. io.js will eventually be merged back into node, it was decided in May 2015 to merge Node.js into io.js and the merged code base will become Node.js.

2.4.5 Support in browsers

An ongoing problem with the web-stack is the differences between modern browsers. The different browsers supports different parts of the specification

³https://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback

⁴https://nodejs.org/api/fs.html#fs_fs_readfilesync_filename_options

⁵<http://semver.org/>

of HTML, CSS and ECMAScript. Furthermore, parts of the specifications define the functionality, not the APIs. Thus, the naming of certain functionality in the API differ from browser to browser.

2.4.6 Browserify and CommonJS

Browserify makes it possible to require dependencies in a similar fashion to Node.js[22]. It is a tool to build JavaScript application for the browser and inject dependencies. It uses a require function similar to Node.js require[23] to define dependencies of a file. As a result, the dependencies must be on the same form as Node.js packages, which means that Node.js packages that do not require server-side dependent code can be used in the browser. Furthermore, it makes it possible to use npm⁶ to install dependencies.

In addition to making the pattern used in Node.js and Node.js packages available code written with Browserify can run in Node.js. Thus, Node.js based test runners and tools is compatible with the code. This makes it possible to run tests without a full browser.

2.5 Docker

Docker is a platform for virtualization. Its goal is to be an easy way to build, deploy and move instances of distributed applications. The key idea is that processes are run within a container that isolates the process from the host operating system, by which makes the host not as vulnerable from the application running inside the container[24]. Moreover, it gives the possibility to run several similar applications on the same host without the risk of the applications interfering with each other. One of the key features of docker is its portability, it is possible to move a container from one host to another without much delay.

2.5.1 Containers and virtual machines

The features mentioned above are features that other virtualization technology, e.g. vagrant, has had for a long time. However, the overhead of running a regular virtual machine versus running a container is significantly different. The virtual machine runs a complete operating system within the virtual machine, on which resources like CPU, memory and disk space is consumed. The footprint in disk space of a simple web API can be below 10MB. However, the virtual machine has a footprint of often as high as 5GB or more. Furthermore, the virtual machines need to reserve resources for both memory and disk. The reservation means that the resources cannot be used by another application. Thus, even if an application uses 500MB of RAM at average and up to 1GB of RAM under load

⁶<https://npmjs.com>

it still needs to reserve 1GB of RAM from the host system. Therefore, running the same application under docker one would be able to run more instances on the same hardware compared to regular virtual machines.

The time necessary to start a virtual machine and install the application could take a while. Docker tries to avoid that by creating the image used for a container one time and running it several times. Building the image takes approximately the same time as installing the application on a regular machine, then starting the container takes an insignificant amount of time. This makes docker more suitable than regular virtual machines in the cases where there is a need to start several instances simultaneously or to start an instance in a short amount of time.

Docker can affect the host system more than a regular virtual machine because it does not have the same restraints on resources. There are restraints in place when using Docker, but they are more liberal than and not as strictly defined as they are in regular virtual machines. However, it is possible to limit the resources that a given Docker container can use, e.g. set the limit of use of RAM to 1GB.

Fat and thin containers

In the Docker community, there is some controversy about whether Docker should be used as fat or thin containers. A fat container is one, in which several applications and services are installed. An example could be using one container for a website. The docker container would need to install a database, cache server, web server and application server. In the same context, a thin container strategy would have one container for each of those applications. The controversy is about how one should use Docker and defining best practices.

Both strategies have positive and negative factors. The thin container strategy makes it easy to scale up specific parts of the application. In the website example, one could add a container for a new application server if that is the current bottleneck. However moving the whole application around requires downloading or building and running several containers that can be time-consuming and unpractical. Fat containers on the other hand can easily be moved around, but is more difficult to scale on specific parts of the application because it requires rebuilding a large container as well as starting a new instance of everything.

2.5.2 Docker as a tool in research

In research, reproducibility is important for the credibility of the work. It makes confirmation of the results feasible. Docker is a great tool to make reproducibility achievable because if the application, algorithm or system can be run with a docker container and the test data set is available then the conditions in form of setup should be the same when someone tries to reproduce[25]. There is

still factors that change between different runs, but several of them are controllable (e.g. the amount of resources in terms of CPU, RAM, etc. available to the test).

2.5.3 Docker registry

The Docker registry is a service for hosting of reusable docker images. It makes it possible to download prebuilt images. The service makes it possible to upload a Dockerfile or to set up automatic pulling of a Dockerfile from a version control server on updates. The Dockerfile is then built, and the docker software can download the image. The time needed to download a Docker image in order to start a container can often be significantly lower than building it locally, especially on servers with a good network connection. A container can be downloaded and started on a new machine with docker with one command.

2.5.4 Headless browsers within docker

Browsers need a screen to attach their graphical user interface, on which they display everything one can see when one uses the browser. In order to create many isolated instances of browsers it is necessary to run them within a virtual machine or container. In the case of running them inside Docker, there is no screen to attach them to. This will also be the case when running them on headless servers, which is appropriate when running automated tests. Thus, it is necessary to emulate a screen so that the graphical interface can be rendered. However, it will be rendered on a virtual canvas that will not be rendered itself. One way to achieve this is to use Xvfb as the window manager, in which the browser is started. Xvfb is a virtual version of the X window management system used on multiple UNIX-based systems.

Networking in Docker containers is by default isolated from the host. All Docker containers share a local virtual network, from which they connect to the internet through a bridge. The bridge requires NAT to open connections back to processes running in Docker containers. However, it is possible to turn this off by starting the Docker container with the option `-net=host`, which will make the process in the Docker container access the network connections of the host OS and thus work as an process running directly on the host OS. Furthermore, this affects browsers running in Docker because if one would want to use a Docker container to test connections without NAT it is necessary to set the `-net` option to host.

2.6 WebRTC

WebRTC, or Web Real-Time Communication is an addition to the ECMAScript API specification[26]. The goal of WebRTC is to enable peer-to-peer communication directly between browsers and other internet devices through a well-defined API[27]. Earlier one had to use plugins to enable communication between browsers, however as a result of the HTML 5 API enhancement. It naturally followed to add an API for peer-to-peer communication. The WebRTC specification and API has two separate types of communication.

There are other implementations of WebRTC than browsers, even though this is meant for browsers. To mention a few, there are reusable frameworks for iOS and Android. Those exist for different reasons. Firstly, the mobile browsers have not adopted WebRTC fully, and a few have not adopted it at all. Furthermore, there are applications that benefit from having access to native APIs on mobile devices. In addition to the mobile platform implementations, there are multiple open-source unofficial implementations of WebRTC in different programming languages.

2.6.1 Media communication

The video streaming capabilities makes it possible to send video streams between peers from browser to browser. The research and development in this area have mostly been focused on video-calls⁷ and video-on-demand services. [28] This thesis will not focus on the media streaming part of WebRTC.

2.6.2 Data communication

WebRTC has support for transferring data in peer-to-peer between browsers or other implementations. This is the part of the WebRTC specification that can be utilized in a distributed hash table implementation. The data communication channels support sending data in various formats, e.g. byte-arrays or JSON-objects.

2.6.3 Signaling

WebRTC is based on peer-to-peer communication, but since it is designed to be used by consumers without configuring networks beforehand there is a need for connection broker[29]. The modern networks that are combined as the internet are built of multiple small networks. The small networks can have firewalls blocking certain traffic or have one IP-address. In cases where a whole network has one public IP-address, the router controlling the network needs to translate

⁷Services like appear.in and talky demonstrate this.

packages sent to the network for it to reach the correct destination. In most cases, this is handled by a technology called Network address translation. To be able to connect directly between peers from different networks behind NAT and firewalls, it is necessary to have a common broker to negotiate the connection. In other words, WebRTC is peer-to-peer but still needs a server to connect in some cases. However, if the peer is publicly available and it knows to listen for incoming messages, there is no need for signaling through a server.[30] Thus, the two peers can negotiate the creation of a data transfer channel directly.

The signaling specification is defined by JavaScript Session Establishment Protocol[31] rather than the WebRTC specification because it is intended to be more general and make it possible for the implementation to make the signaling decisions. Thus, the requirements of the protocol are specified in its document.

Figure 2.5 shows a sequence diagram of the process of signaling between two peers. They both have a connection to the signaling server that is used to communicate offers and answers. *Peer1* tries to connect to *Peer2*. The process starts with the creation of an offer on *Peer1* and attaches a local descriptor. The descriptor contains meta-information about *Peer1* and is set as the local descriptor on *Peer1*'s data-channel object as well. Moreover, *Peer1* sends the offer to the signaling server through the connection already established. Then the signaling server forwards the offer to *Peer2* if it has a connection to *Peer2*. Otherwise, the signaling server reports back to *Peer1* with a message that *Peer2* is not available. If *Peer2* is available and receives the offer and wants to connect, it creates an answer set local descriptor on it and itself, as *Peer1* did with the offer. Furthermore, it must set the local descriptor of the offer as the remote descriptor on its data-channel instance. Then *Peer2* sends the answer to the signaling server, by which sends it to *Peer1*. After receiving the answer *Peer1* sets the descriptor attached to the answer as the remote descriptor on its data-channel instance and the WebRTC data-channel connection is open and can be used to send data between the two peers. Also, the answer and offer payload might contain interesting metadata like browser type and version, which can be useful when sending data since different browsers support different ways of serializing payloads.

2.6.4 Browser support and frameworks

The implementations of the specifications in the different web browser differ in several ways. There are browsers that do not support WebRTC. It is mostly newer versions of Chrome, Firefox, Safari and Opera that support WebRTC. However, how you interact with the communication layer differ. The different browsers have their names for API functions, namespaces and variables. To make interaction between applications and the WebRTC API easier there exist frameworks that give an abstraction layer with a single API. A few of these frameworks also add an abstraction layer on top of the signaling described in section 2.6.3. Peer.js and SimpleWebRTC are described in more detail in the

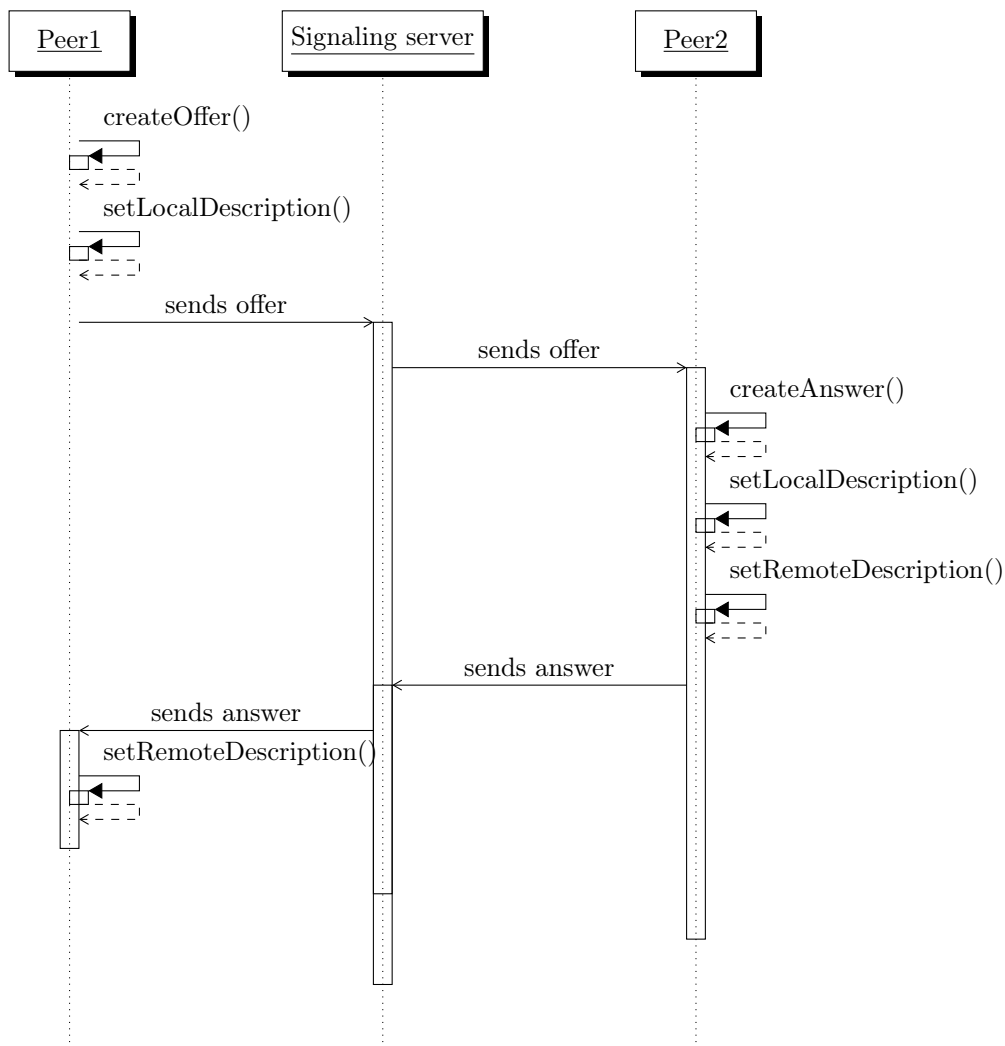


Figure 2.5: Sequence diagram for a connection process in WebRTC

sections below.

Peer.js

Peer.js is a framework with assets that are useful on the client side and the server side. On the server side, they have a package for Node.js which can be used as a signaling server. The team behind Peer.js also offers this as a service from peerjs.com, which is a hosted version of the node-package. On the client side,

there is a package that handles signaling and abstraction of browser differences. Thus, it is possible to use the same code in different browsers as long as they are supported by Peer.js. The largest benefit of using Peer.js is the handling of differences between browsers. In addition, connecting between nodes and error handling is handled in a good way and keeping that part of the code separated from rest of the code base lowers the risk of regression errors in connecting and handling of errors. Furthermore, since it defines an API, it adds a layer of abstraction that can be mocked. Mocking communication between nodes makes it possible to write isolated tests to test the correctness of functionality in the routing algorithm in a simulated environment. `peerjs-mock`, described in section A.2, and `peerjs-rpc`, described in section A.1, both utilizes this to enable isolated testing. The ability to write isolated tests decreases the feedback loop significantly compared to starting multiple browsers and connecting them before being able to test the given functionality.

SimpleWebRTC

SimpleWebRTC is framework created to make it easy to start a new WebRTC application[32]. It takes care of signaling and connecting to other clients. The framework is designed around the concept of rooms, to which clients connect. Thus, there is little control over the client-to-client connections. SimpleWebRTC will connect to each client in the given room and accept connections from each client in the given room. Furthermore, the abstractions in this framework hide functionality that an advanced application, as a DHT would need. However, as a starting point for a proof of concept of simple applications utilizing WebRTC it would be a good choice.

SimpleWebRTC is built mainly for the media streaming channel of WebRTC because it was written to build Talky, a video conferencing application based on WebRTC. The only use of the data channel that this framework supports is file transfers. However, the API for file transfer is limited.

2.6.5 Testing WebRTC applications

Tests for WebRTC applications for the browser requires a browser. Using browser emulator and virtual DOMs, as might be custom for testing of client side code for browsers, will not work. None of the available browser emulators has support for WebRTC. Thus, it is necessary to use either Chrome or Firefox with a webdriver like Selenium. The webdriver makes it possible to control the web browser from code and automate testing. Selenium supports fine-grained control of the browser with clicks and other behavior that are expected of the user of a web browser. However, in the case of testing a WebRTC application with minimal user interaction it may be only necessary to be able to control the navigation. Moreover, Selenium supports waiting for a given condition for an amount of time. In the case that the condition becomes fulfilled, the sele-

nium script will move on. Otherwise, the script will raise a condition. Thus, it is feasible to run a browser with selenium until a condition in the JavaScript application is fulfilled and then quit the browser.

Headless browsers

The headless browser makes it possible to run tests that need a browser on a server without a graphical interface. The browser software needs a graphical interface to render web pages. They are created to be used with graphical interfaces not to run on servers. However, it is possible to run a normal browser as a headless browser by attaching its graphical interface to a virtual X-screen. X is a window system used on Unix-like systems and there exists a virtual version of it called Xvfb. Thus, it is possible to run a browser, e.g. Firefox or Chrome, on a Linux server, without attaching it to a screen. Furthermore, running browsers with Xvfb is helpful when running several browsers inside different Docker containers as described in section 2.5.4.

Testing for correctness

Testing of correctness can be divided into two parts. The first is testing of correctness in the code of the project and the second is testing for correctness with the whole stack. The latter would be an integration test. If one assume that the underlying abstractions work as expected testing for correctness in the project code should be enough, however it is always better to test with integrations in place because software might not always work as expected. The first part is achievable by mocking the WebRTC API or the API of the abstraction of WebRTC that is used. Mocking WebRTC will also make the tests run outside a browser with Node.js testing tools like mocha. Thus, the tests run faster and can run on continuous integration without a headless browser.

Testing in at scale

In the development of peer-to-peer applications with WebRTC, there is a need to test in real browsers at approximately the target scale. Thus, if one creates an application designed for 20 peers. There should be performed testing on approximately the same amount of peers. Testing with two-three peers will not be satisfyingly checking the requirements of the application.

Simulations will only test parts the application. It can be a great way to find logic errors in certain situations and ensure that regression errors do not reappear. However, it is necessary to use actual browsers to be able to evaluate the performance of the application when running in browsers and to test their integrations.

Testing with many browsers can be complicated and an issue of orchestrating

different actions in different browsers. Therefore, running browsers with scripts is wise to be able to get a predictable behavior of the browser. Furthermore, running the browsers headless, as described above, gives the ability to run browsers on several nodes and control them from one, which makes it possible to run more browsers and larger tests. Besides, it is important to test that browsers running on different machines and sub-networks can communicate and work together as the application expects. Also, Docker makes it feasible to start the same browser in a sandboxed environment multiple times on the same host or several hosts without much overhead.

3 | Design

The implementation in this project is separated into smaller modules. There is a module for the Chord algorithm, a module for using the algorithm in a web browser, a module for viewing statistics about the Chord network in real-time. In addition to those modules, there are helper modules for RPC, mocking of Peer.js and mocking of the RPC module.

3.1 Levels of abstractions

Implementations of software are always on a given level of abstraction. In distributed systems, the communication often has impact on which abstraction levels that are appropriate. The WebRTC environment, especially, gives restrictions on the lower end of the abstraction levels. However, the different WebRTC-frameworks also defines another level of abstraction. Section 2.6.4 describes some differences between browsers and implementations of WebRTC. Thus, as described in section 2.6.4, an abstraction on the different APIs into a single interface is helpful. The solution described here is based on Peer.js[33], a framework described in section 2.6.4.

Furthermore, the implementation will communicate similarly to known Remote Procedure Call (RPC) implementations from other languages and platforms. Therefore, the project would benefit from an RPC implementation in peerjs. There were not any acceptable RPC implementations on top of WebRTC or Peer.js when this project started. Thus, developing a RPC-module was necessary as a part of the project. The module, peerjs-rpc, can be installed from npm and is available to the general public. The module is described in more detail, in terms of design and API, in section A.1 in the appendix. Figure 3.1 shows the abstraction levels and where in those layers the Chord implementation and the RPC-module are placed. The communication is handled by the web browser and the WebRTC API. Thus, the communication is never handled directly by the implementation.

The RPC module is built using the promise pattern, see section 2.4.2 for information on the pattern. Therefore, it is possible to use function calls in between

nodes as promise chains, and it can also be a part of a bigger promise chain. Using promise chains for all asynchronous task makes sure that all asynchronous events happen in the correct order. Furthermore, since the RPC module populates the error between nodes and reject promises as if the error happened locally, the correct node handles errors in the correct place. Thus, the error is handled on the closest place to its origin where it is understood. In most cases in Chord, that place is on the node that initiated the original sequence of events like a get-call or a call to findSuccessor.

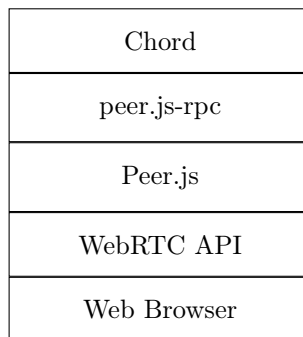


Figure 3.1: The development stack of Chord implementation.

3.2 Chord on WebRTC

This section will describe the implementation of the regular Chord algorithm, see section 2.2.3, while section 3.3 describes the changes needed to be able to enhance the routing based on geographical location.

The implementation of Chord on WebRTC needs to be designed to work in a modern browser with support for WebRTC. It is important to have a dependency management system, by which can solve dependency injections of JavaScript libraries, as well as local files. The choice for this project was Browserify, see section 2.4.6. It makes it possible to run the code in Node.js or io.js in addition to browsers as it leverages CommonJS dependency injection. The ability to run the code in Node.js or io.js as well as browsers makes it possible to run tests and simulations without using browsers. Thus, the feedback loop from unit tests becomes smaller since unit tests can be run without a browser or browser emulator. Most of the unit test needs to mock WebRTC/Peer.js to restrict the scope of a given test. Thus, mocking Peer.js and running unit-tests in Node.js with Mocha results in a better development process. Furthermore, the ability to run simulations in Node.js makes it possible to test the performance of the routing in given environments and situations, in addition to, the ability to create more sophisticated tests with a whole chord ring running inside a Node.js script.

3.2.1 Structure

The implementation is divided several prototype objects. There is four main classes **Chord**, **Node**, **Task** and **Storage**. The sections below describes them in detail. Figure 3.2 illustrates the relations between prototype objects. Each line represents references between instances, the object above has a reference to the object below. Also, the RPC prototype object in the figure is from peerjs-rpc.

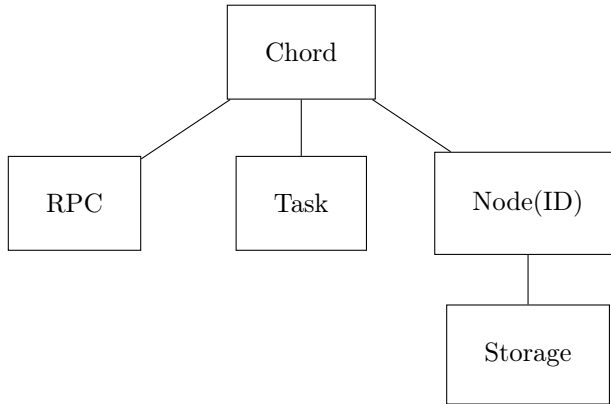


Figure 3.2: The relations between prototype objects.

Chord This object represents an instance of the algorithm and represents a peer. It has an instance of the RPC-object from peerjs-rpc. It contains all necessary functionality of the Chord algorithm, e.g. `findSuccessor` and `stabilize`. All communications between peers are between instances of this object through their instance of the RPC-object. Furthermore, on the creation of the Chord object it will pass itself as scope to the RPC-instance. Thus, other nodes can invoke methods on the given instance and access attributes on the given instance. An instance of this object contains a reference to an instance of **Node**, in which it stores information about itself.

Node The Node object is a representation of a node in the Chord-ring. It inherits ID, so it has all the functionality of that object as described below. Also, the Node object contains information about the predecessor, successor and the finger table. Furthermore, a reference to the storage is also within this object, which is an instance of Storage described below. The predecessor, successor and each node in the finger table are also instances of the Node object. However, unknown attributes like the finger table are not set in those instances.

ID This object stores the hash of the node as hex-string and byte arrays and has the functionality to compare with other ID-objects. Furthermore, this

object also can check whether an ID is within in the range between the two other ID objects. It is in its object in order to use the functionality without using separate this functionality from the Node object, described above, was

Task The task object maintains the running of scheduled tasks, e.g. `fixFingers`. It encapsulates the built-in `setInterval` in the JavaScript language. It handles starting and stopping of tasks and error handling within tasks. In addition, it stores an counter of how many times the task have run for debug purposes.

Storage The Storage object is an interface for different storage methods(e.g. cookies, Web Storage), and it has a basic in object storage for testing purposes. Using the in object storage makes the storage class store the data in a regular JavaScript object, which will exist for as long as the storage object exists. Thus, it is great for testing as creating a new Storage object or a new Node object will create a new object to store the values in and the old will be garbage collected.

3.2.2 Routing

Chord has two different types of key location lookup, from which there are two different implementations of `findSuccessor`, as described in section 2.2.3 and section 2.2.3. The implementation in this project solves this by adding an argument on calls to `findSuccessor` that defines the key location lookup strategy. Thus, the result is a merged version of the two shown in algorithm 1 and algorithm 2 in section 2.2.3 The updated `findSuccessor` function is shown in algorithm 4. Using an argument on the lookup functions instead of a separate prototype object for each of strategies makes it possible to change on the fly. Thus, it is possible to do a key location lookup with different strategies without creating a new ring, which necessary to test comparatively between the different strategies.

3.3 Changes in Chord

In order to benefit from the knowledge of geographical location, the Chord algorithm needs changes. The Chord algorithm uses a finger table to enhance routing performance in the scalable key location version, as described in section 2.2.3. The proposed algorithm will build on the scalable key location version and has a few changes to utilize the geographical location of the node.

Multiple parts of the chord algorithm need changes. Firstly, all parts communicating information about nodes need to pass the geographical location of the nodes. Thus, instead of sending a single hash string, when resolving `findSuccessor` and other functions that return nodes, it is necessary to send an object containing the hash and the geographical location. Furthermore, the

Algorithm 4 The findSuccessor implementation that can choose strategy

```

1: function n.findSuccessor(hash, strategy)
2:   function STRATEGIES.scalable(hash)
3:     return successor.findSuccessor(hash, "simple")
4:   end function

5:   function STRATEGIES.scalable(hash)
6:      $n' = n.\text{closestPrecedingNode}(\text{hash})$ 
7:     return n.findSuccessor(hash, "scalable")
8:   end function

9:   if finger[i] ∈ (n, successor) then
10:    return successor
11:   else
12:    return STRATEGIES[strategy](hash)
13:   end if
14: end function

```

finger table and predecessor- and successor-fields have to store the geographical location information. Secondly, the routing must be changed to use the location information to route more efficiently. In order to use the location of a node to route better, the algorithm has changes in the closest preceding node lookup in the finger table.

Algorithm 5 shows the new closest preceding node lookup. The difference from the original lookup is the when the loop finds the closest node in the finger table, see line 3. The new lookup will check if the node that is one step farther away, in the keyspace, in the finger table is geographically closer. If it is, the next closest node in the finger table will be chosen instead of the closest. Figure 3.3 shows the lookup routing from $N1$ to a key that belongs to $N8$, in which figure 3.3(b) is the lookup with geographical enhancement while figure 3.3(a) is the same as figure 2.2(b) from section 2.2 shown for comparison.

Furthermore, the findSuccessor needs changes in order to call the correct closest preceding node function. Algorithm 6 contains changes necessary, in which line 9 to 12 contains a new function that calls the geographical aware closest preceding node lookup. Since line 16 calls the correct strategy function based on the strategy argument it is possible to decide whether to use simple(see section 2.2.3), scalable(see section 2.2.3) or geographical enhanced lookup on each separate call to a lookup function.

The approach described above preserves the correct lookup guarantee of the original Chord implementation. The guarantee states that if all nodes have the correct successor the lookup will always return the correct node. The nodes returned by the closest preceding node lookup in the finger table always will return a node that is between the node asking and the node containing the

Algorithm 5 Changes to the lookup to account for geographical distance

```

1: function n.geoClosestPrecedingNode(hash)
2:   for  $i = \text{finger.length} - 1$  to 1 do
3:     if  $\text{hash} \in (n, \text{successor}]$  then
4:       distance1 = geographicalDistance(n, finger[i - 1])
5:       distance2 = geographicalDistance(n, finger[i])
6:       if  $\text{distance1} \leq \text{distance2}$  then
7:         return finger[i - 1]
8:       end if
9:       return finger[i]
10:    end if
11:  end for
12:  return n
13: end function

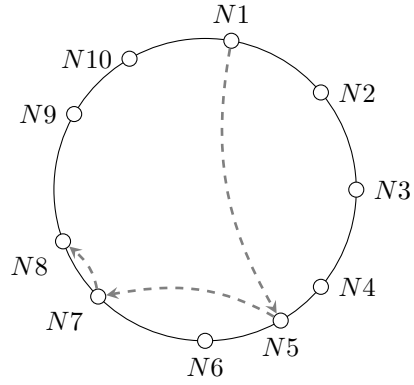
```

requested key location. Thus, it will at some point reach the predecessor of the node containing the requested key location. Therefore, the **findSuccessor** function will return the successor of the node asked, as shown on line 14 in algorithm 6.

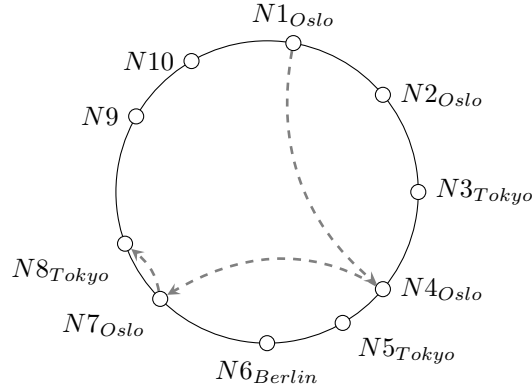
3.4 Error handling

The error handling strategy for used in this project is based on the promise pattern, from which custom placing of error handling is achievable. In section 2.4.2, it is mentioned that promise chains will break on error, and the error populates to the first catching element in the chain. Thus, in asynchronous distributed chains of events the error will still be handled at a place where the error the application has a large enough context to understand how to handle the error. Furthermore, the intention is to handle the error as close to the origin of the error as possible. However, if one invoke of a function on another peer throws an error the peer that throws the error might not know what the error affects. Thus, sending it back to the invoking peer is the wanted result, because that peer has the context to handle the error. The promise chain from the RPC module handles this so that the invoking node gets the error.

In some cases, the error needs to be handled locally. It is possible to handle errors locally on the invoked peer by adding a catch on the local promise chain. The catch block handles the error and then rethrow the error, or it throws a new error after handling the original error. Thus, both the invoking peer and the invoked peer is aware of the error and can handle it appropriately.



(a) Scalable key location



(b) Geographical key location

Figure 3.3: The two figures shows the lookups necessary to find an item stored on N8 from N1

3.5 Testing and simulation

This project involves several types of testing: unit testing, simulations of routing and real world scenario testing.

The first is an important part of the development process and is two-sided: unit tests running in node and unit tests running in the browser. However, it is the same test suite run in different environments for different purposes. The unit tests for this project runs with a test framework called mocha¹. The tests are as described above scoped down to test specific parts of the algorithms and utilities used by the algorithms and their interaction with the RPC-module. Thus, the tests will not call any WebRTC functionality as this set of tests assumes that

¹<http://mochajs.org/>

Algorithm 6 findSuccessor after adding geographical key location lookup

```

1: function n.findSuccessor(hash, strategy)
2:   function STRATEGIES.scalable(hash)
3:     return successor.findSuccessor(hash, "simple")
4:   end function

5:   function STRATEGIES.scalable(hash)
6:      $n' = n.closestPrecedingNode(hash)$ 
7:     return  $n.findSuccessor(hash, "scalable")$ 
8:   end function

9:   function STRATEGIES.geographical(hash)
10:     $n' = n.geoClosestPrecedingNode(hash)$ 
11:    return  $n.findSuccessor(hash, "geographical")$ 
12:   end function

13:  if  $finger[i] \in (n, successor)$  then
14:    return successor
15:  else
16:    return STRATEGIES[strategy](hash)
17:  end if
18: end function

```

the RPC-module works as documented. Instead, it uses mocking libraries that mock the WebRTC-communication between nodes and enables several nodes to communicate with defined side-effects. Thus, since all usage of browsers specific API's are mocked, the tests can run in Node.js or io.js. However, the mocha tests can run within a browser if the test runner loads the tests correctly. To be able to run the tests in a browser, the tests must be bundled with Browserify and loaded in a browser window. Karma² is a tool that can accomplish this by using plugins for mocha and Browserify. These tools make it possible to run the tests in the background while developing and load them into a browser for an extra check or to debug issues that only appear in a browser.

The second type, the simulations, are designed to be able to run the distributed algorithm in a controlled environment. Section 4.1 in the experiments chapter describes the simulation setup and the results in detail. The simulations run in Node.js or io.js and can simulate a network of Chord nodes by using a mock module for peerjs-rpc called peerjs-rpc-mock. The module support simulating delays on the nodes on a connection based level. Thus, it is possible to define delays between all nodes both ways. Using simulations in this way can be a powerful tool for evaluating the algorithm and improving its performance and for comparing different versions of algorithms.

²<http://karma-runner.github.io/>

The last type, the real browser network tests, are designed to evaluate how the system performs in as real as possible situation. Section 4.2 in the experiments chapter describes the experiment and its results in detail. The tests run in a real browser. Furthermore, to achieve an as real situation as possible there must be several physical nodes included in the network and several browser instances. In addition to those requirements, since one version of the algorithm uses geographical location there must be nodes running in different locations. The tests must run from data centers on different continents. Browsers are, as mentioned in section 2.5, intended to render web pages. Thus, they do not work on servers without a window management system. These tests, therefore, run in browsers attached to a virtual window management system, by which makes the browser believe they render the content on a screen.

In this project, two docker images was used one with Firefox and one with Chrome. Detailed descriptions of the Docker images is listed in appendix B. The Chrome docker images was unsuitable to use with geographical location because it was not feasible to avoid the prompt to allow access to the browsers location(see section 2.3.1). Thus, the experiments was performed using the firefox image as it was possible to load a settings profile with geographical location access set to always accept.

3.5.1 Logging and testing utilities

The Chord module is a Browserify module, from which it is possible to import all the components described in section 3.2.1. The test page contains a loading script written in JavaScript that imports the Chord algorithm and configures everything necessary to run the test. This script is the entry-point of the Browserify-bundle created for the test. The loading script instantiates a new Chord object and tells it to join an existing node.

A Node.js based server serves the script that initiates the Chord algorithm. The server is written in a JavaScript and Node.js and serves a page with the test page and a REST API view with a list of clients. The purpose of the API view is coordinating of tests, and the view would not be necessary in a real use case of an DHT with WebRTC. Furthermore, the same Node.js server runs the server-side part of Peer.js, which handles signaling and negotiation of a new WebRTC connection. Also, the node server listens to a WebSocket(see section 2.3.2) channel for logging and coordination of the browsers connected to the Chord application.

Furthermore, the loading script configures some logging utilities. The headless browser tests must run on several servers as mentioned above. The headless browsers do not give access to the logging console in an easy way, see section 2.6.5. Thus, custom logging is necessary to know what was going on during tests and after the tests. The wrapper script that initiates the chord algorithm, therefore, sends log messages over WebSocket to the node server serving the test site. The node server logs all messages it receives over WebSocket. Thus, the

Node.js server works as a central logging service for the test environment.

The Node.js server can also transmit messages through WebSocket channels to all browsers currently participating in a test running on the given server. Thus, it is possible to send commands. Such commands are helpful in controlling and coordinating the test nodes since there is no direct control over the web-browsers. Examples of implemented commands are refresh, by which all browsers would refresh the page, and quit, by which all browsers would result in the quitting all the browsers.

3.5.2 Dashboard

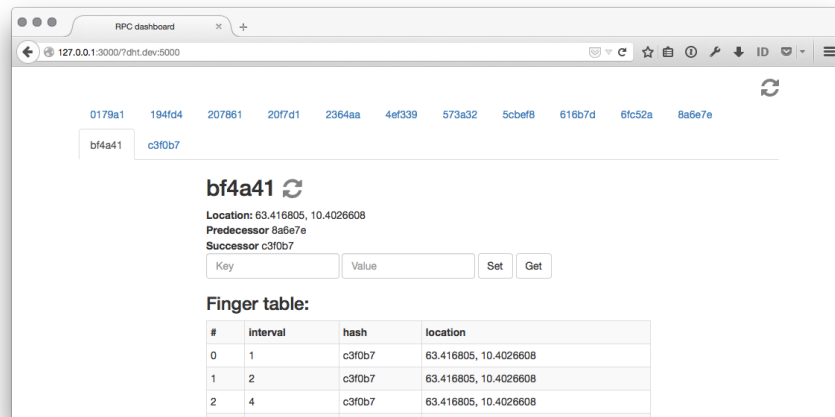


Figure 3.4: The RPC dashboard

In the real-scenario tests, it is necessary to communicate with the Chord ring. Querying nodes with get- and set-request is easy to do with the RPC module. Thus, as long as a peer ID and the signaling server is available it is possible to invoke get and set on the given peer, to which the ID belongs. There are two ways to accomplish the invoking. However, both are in a browser since the RPC-module need Peer.js to load. The first is to add the test queries to the loading script described in section 3.5.1. The second is to build a standalone utility to query with that creates a RPC instance and use it to query Chord peers. The first would be the quickest and easiest to create. However, it would be tightly coupled with the test script. The tight coupling is unwanted because if changes to the querying mechanism are required a new Chord ring must be created and stabilized before it would be possible to query again. Furthermore, as a standalone application the querying module could be used with other applications utilizing the RPC-module. Therefore, a test-dashboard was created.

The dashboard is a simple one-page JavaScript app. The dashboard does two things. Firstly, it connects to all nodes in a list of peer IDs and invokes the same function on all of them and stores the result in a list. The dashboard passes the result list to a template that render the information as HTML, before the rendered template is shown to the user. Secondly, it is possible to invoke a function on a given node.

Furthermore, the dashboard is built generically, by which makes it usable for other applications that use the peerjs-rpc module. The dashboard designed to accept custom templates and JavaScript code that creates listeners for forms in the template. Also, the name of the function that the dashboard invokes on all nodes can be set as an option.

4 | Experiments

This project had two types of experiments. A simulation ran with `io.js` on one server, and an experiment ran in real browsers. It is just the latter that had access to WebRTC. Thus, the simulation uses mocked communication with predefined delays. Setup and results of both experiments are described in the following sections:

4.1 Simulations

The simulation ran in `io.js`, in other words it ran directly on a machine, not within a browser. As described in section 2.4.4, `io.js` makes it possible to run JavaScript on a machine outside the browser. This makes it possible to evaluate the performance of the routing in predictable conditions. Even though the conditions are predictable it does not mean they are necessarily optimal conditions. It makes it possible to define the conditions to fit the test in question. Thus, it is possible to add latency, delayed and faulty messaging.

In a test between browsers in a real-life situation there are multiple changing factors that affect the performance of routing and messaging in general. Those factors are beyond the control of the application and the WebRTC framework. The abstraction level in which this application is operating has no control over network or the browser. Thus, when the message is handed to the browser to be sent to another peer there are many factors that weigh in on the correctness and speed of the delivery.

This experiment is designed to evaluate the changes to the Chord implementation in known conditions. The geographical aware routing will be tested by adding a fictional delay to the RPC-module. Thus, all messages sent with the RPC module will be delayed by a given time based on the given distance between the sending peer and the receiving peer. Each node will have a preset location, with latitude and longitude as the Geolocation API would have returned.

The delays are based on data from publications of statistical data of delays on geographical distances. [1] shows that the round-trip time for a given distance is as shown in table 4.1. The RTT data is from statistics from the Akamai network,

which handles delivery of streaming of live video feeds from one location to the whole world. Thus, the data is trustworthy as it is taken from statistics of real use-cases where RTT is a key performance metric. Hence, using that data to configure latency in the simulations should generate a realistic view of the latency impact on the RTT in the Chord network. However, this will not account for the delays in browsers and computation, which will be present in the real browser test in the next section.

The locations chosen for the experiment are Berlin, Paris, Honolulu, Oslo, Tokyo and Washington. The distances between those location ranges from 500 km to 10 000 km. All those distances yield additional noticeable latency as they are longer than the local distance approximation in [1].

Table 4.1: Data on round trip times from [1]¹

Distance	160 km	800 - 1600 km	5000 km	10 000 km
RTT	1.6 ms	16 ms	48 ms	96 ms

4.1.1 Dataset generation

A dataset generator was developed for this project. It generates all possible hash values in the keyspace, from 000000 to FFFFFFFF, and takes a sample from those hash values. The sample can either be evenly distributed or chosen more randomly. Then, the generator takes the sample and generates a Chord ring by setting the correct predecessor and successor on each node. Furthermore, it populates the finger-table with the correct data and sets the location either based on a modulo based selection or randomly choosing a location. Finally, the generated Chord ring is converted to a JSON array to make it possible to load it into a simulation. The JSON array can be saved to a JSON-file, from which it is possible to load the generated Chord ring multiple times. Thus, it is possible to verify a simulation by running it several times with the same Chord ring.

4.1.2 Experiment setup

The simulations experiments ran on a Ubuntu 14.04 server running on a Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz with 16 GB of RAM. Furthermore the simulations ran in io.js version 1.8.1.

4.1.3 Retrieval simulation

This simulation is designed to test end-to-end time of gets in a prepared chord ring. All runs of this simulation will be based on chord rings which the test

¹The paper has these values in miles, therefore the values are converted.

loads from JSON-files. The files contains information about each node as well as their predecessor, successor and finger table. Thus, the test can run directly without waiting for the network to stabilize. Furthermore, it makes the test reproducible because it possible to load the same ring several times. Thus with the code checkout from version control and the correct data file, the simulation should give the same result when run several times on the same hardware.

Long key distance

In one simulation 1000 queries was performed on two different 1000 node Chord networks. The first had an even distribution on geographical location, each node had location based on $n \bmod \text{LOCATIONS.length}$, where n is the nodes placement in the Chord ring. The second had randomly generated geographical locations. They where assigned based on $\text{Math.random}(0, \text{LOCATIONS.length})^2$. Figure 4.1 shows the cumulative timings of the queries in the first Chord network and figure 4.2 shows the cumulative timings of the queries in the second network.

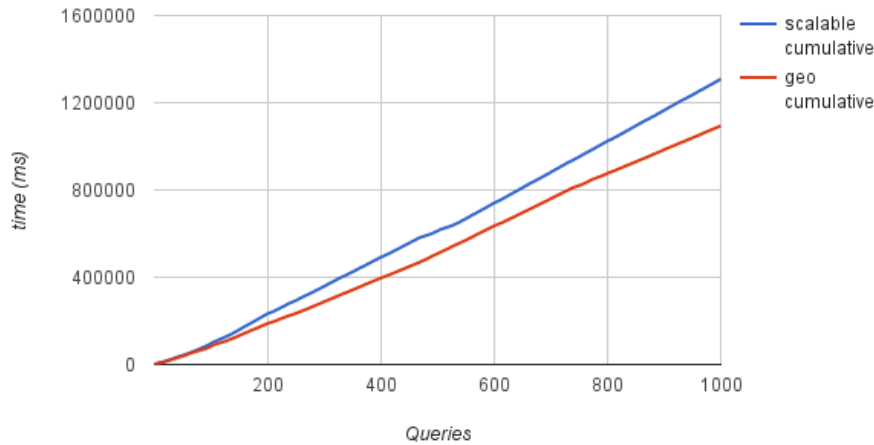


Figure 4.1: Cumulative timings of 1000 different key location lookups with evenly distributed geographical location

A lookup of the farthest away from the peer doing the query takes at average 6-7 lookups in a network with 1000 nodes. Hence, if all lookups have the longest distance the lookup will have 1344 ms in added latency. The simulations had

²https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

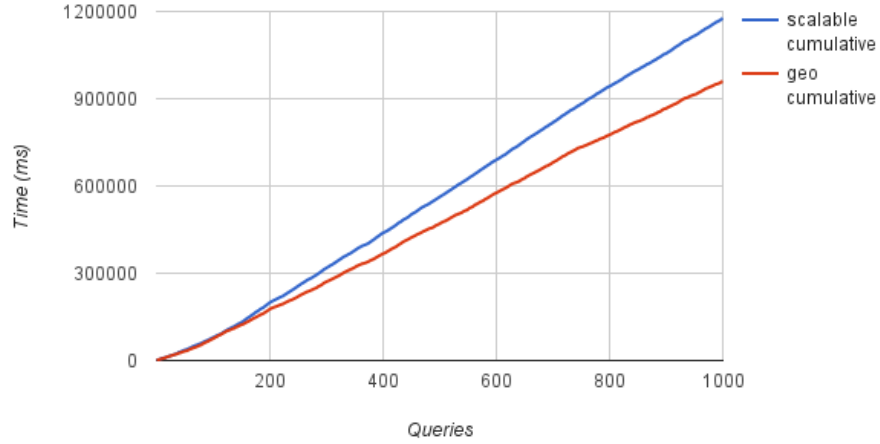


Figure 4.2: Cumulative timings of 1000 different key location lookups with random geographical location

an average query time below that, as expected since not all connections are over the longest geographical distances. Table 4.2 lists the average and standard deviation of the simulations of both the scalable lookup and the geographical lookup.

Table 4.2: Relevant data from simulations with 1000 nodes

Distribution of locations	Even		Random	
	Scalable	Geographical	Scalable	Geographical
Average query time	1308 ms	1093 ms	1176 ms	959 ms
Standard deviation	308	288	314	325

Variable key distance

In another simulation designed to compare the benefit of geographical enhanced routing in different key location distances, the simulation ran a query for each bit distance from 1 to 22 and ran the query from each node in the Chord network. Figure 4.4 shows the result from the simulation. It shows that longer distances gives higher rewards, which is to be expected as the longest distance will have $O(\log N)$ lookups while the shortest have $O(1)$ lookups. Thus, the longer the distance the more chances for the geographically enhanced to affect the routing. Also, the figure shows that the geographical strategy at least performs as good as the Chord scalable lookup strategy.

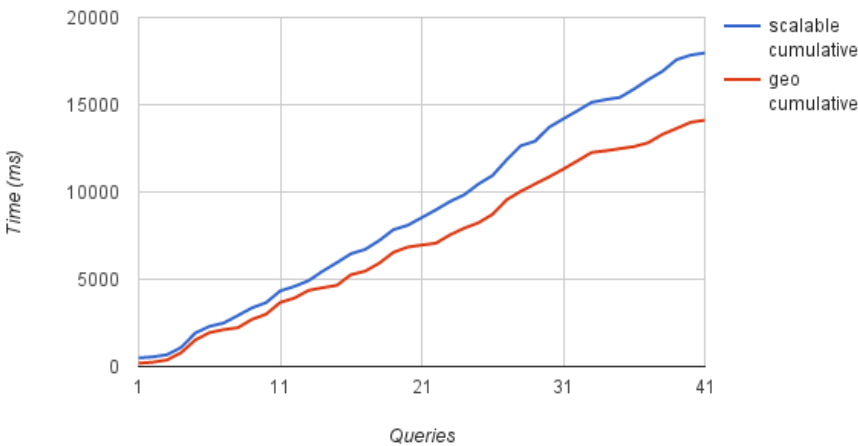


Figure 4.3: Cumulative timings of key location lookups with 40 nodes random geographical location

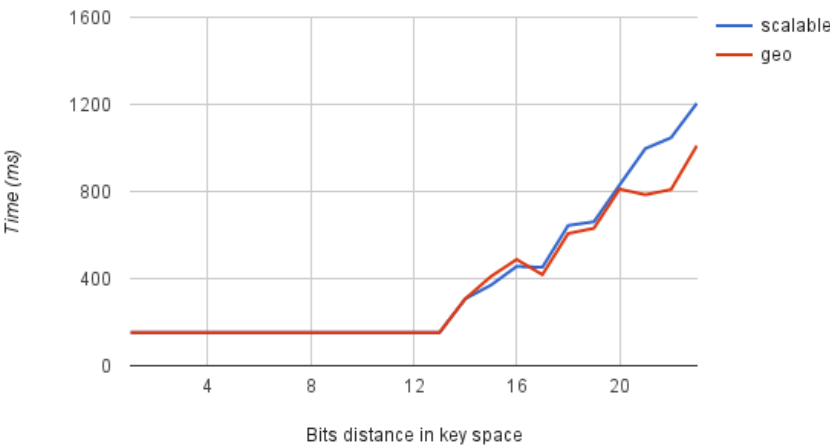


Figure 4.4: Average query time queries of different bit distances

4.2 Real browser experiments

The real browser experiments are designed to measure how the algorithm works and performs in a real scenario. An instance of `rpc-dashboard` performs and controls the experiments. The dashboard, which has the functionality to do multiple queries as in the simulations, run on a regular machine that have a regular browser, from which it is possible to interact with the dashboard. Furthermore, the dashboard connects to all peers and can invoke any function of the Chord algorithm on any of those peers. The peers run on headless browsers inside docker containers as described in section 2.5 and 3.5.

4.2.1 Experiment setup

The experiments ran on the latest stable version of Firefox(37.0.2) within Docker containers running on Docker 1.6 with Ubuntu 14.04. The Docker containers ran on 20 core virtual machines in Digital Ocean's³ data centers in San Francisco, New York, London and Singapore in addition to one server on the campus of Norwegian University of Science and Technology. The Docker container is available on the Docker registry⁴.

There are some limitation that apply when running these experiments. The most noticeable one is CPU power. Using the WebRTC API can be fairly CPU intensive, especially when several instances of a browser is running inside Docker containers on the same host. The experiments showed that there is a limit on the number of peers that can run on a given machine before the processing time is so significant that the latency, saved from routing based on geographic location, is insignificant. Thus, there is a limit on how many servers it is possible to run a given number of serves from Digital Ocean.

4.2.2 Retrieval experiment

A retrieval experiment ran on several Chord networks. The experiment was performed by creating N browser instances that visited a given url and viewed that page until the TTL expired or condition in the selenium script was fulfilled, which would happen if a certain element became visible. The test page loaded Chord and configured logging and other necessary utilities, as described in section 3.5.1. The first node creates a Chord ring and all other calls join with the first node as the node they wants to join. Thus, all except the first node would invoke `findSuccessor` on the first node.

The dashboard described in section 3.5.2 was used to monitor the Chord network until it stabilized itself and all nodes had the appropriate predecessor and

³<https://www.digitalocean.com/>

⁴https://registry.hub.docker.com/u/relekang/firefox-webrtc/build_id/55395/code/bfmny6kvdzfdbntyx6jktdb/

successor and their finger table was populated. Then the dashboard was used to query several lookups from each node to different key locations.

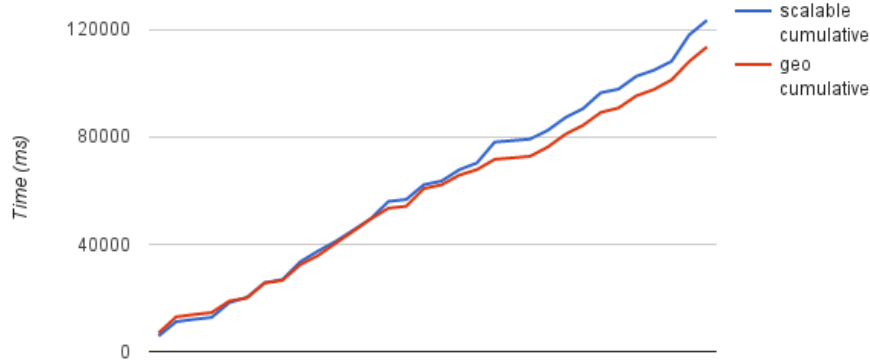


Figure 4.5: Cumulative timings of key location lookups with 40 running in docker containers

Figure 4.5 shows the cumulative time of queries ran on a Chord ring with 40 nodes. It shows that the geolocation enhanced routing also has a benefit when running in a network of real browsers.

5 | Discussion

5.1 Geographical enhanced routing

The experiments show that there is an increase in performance when utilizing geographical location data when routing in a Chord network. The increased performance is not significant compared to the cost of additional lookups. However, that is to be expected that extra lookups cost more than the increased latency on long distances. On the other hand, the geographical key location lookup strategy performs at least as good as the scalable solution, which indicates that choosing the next closest preceding node does not add a significant amount of additional lookups. It can be argued that it indicates that in most cases it results in the same amount of lookups. Thus, running the geographically aware lookup strategy poses little risk of a decrease in routing performance.

Nonetheless, there is an overhead of maintenance and communication of the geographical aware lookup strategy. All maintenance messages that refers to nodes must send the location. There is a significant increase in size when comparing an object with the hash value representing the peer id and location compared to a string of the peer id. Despite the footprint difference, the increased payload size is insignificant compared to the base payload size of the RPC communication. Thus, even though there is a cost, the cost of running the geographically enhanced lookup is not high.

Furthermore, the simulations show that the increased performance of using the geographically enhanced varies based on different factor. The performance increase is larger in a lookup with long key distance compared to a lookup of short key distance. This is to be expected as the closer the destination key is the more dense the finger table. Thus, it will require fewer lookups to find the correct location, whereas the geographical enhancement will not apply often. Furthermore, in the beginning of the finger table there is a higher probability of the same node occupying several rows. Hence, when the lookup of closest preceding node checks the row on $n - 1$, the probability that the distance will be the same is higher. Even so, lookups that use the closest parts in the finger table often result in one or two lookups. Even if there is a long geographical distance between the nodes, the long distance round-trip is inevitable in order

to find the correct node.

Comparing the simulation with the real browser experiments shows interesting results. The real-browser experiment indicates that there is an improved performance in geographically enhanced routing as seen in the simulations. However, the benefit is less significant compared to the total time. The total time of a lookup in the real browser experiment is higher than the time of the simulation. The increased duration of a lookup is to be expected as the actual browser test has several factor affecting the time it takes to do a lookup or send a message. Firstly, sending messages across networks has a base latency not accounted for in the latency listed in table 4.1. Secondly, using a real network connection is prone to have package loss and bad routing or other network related errors that are not seen in a simulation. Thirdly, there is an increased consumption of CPU when sending real messages. Moreover, the browser overhead will also add additional CPU consumption. The CPU consumption is further discussed in section 5.2. Lastly, the impact of shared computing resources from running on a shared data center is also a risk that is hard to avoid and measure, but it should not be significant in the end.

The first two factors is as expected while the performance on the load in each browser is a bit higher than expected. Thus, the lowered increase in the test with an actual browser is expected. Nonetheless, there is an increase in performance. The performance increase could have been higher if each machine ran fewer nodes, from which the CPU consumption on each machine would be lower. Thus, it indicates that using the geographical location to enhance routing in geographically widespread peer-to-peer networks can be beneficial. Furthermore, the strategy for utilizing the geographical knowledge can be refined and further developed to create an even better performance strategies. Different peer-to-peer systems need different strategies to be able to use geographical knowledge.

Moreover, an interesting additional side-effect of lowering the amount of long-distance round-trips is the decreased probability of package loss. It is shown in [1], the publication with the latency statistics, that the probability of package loss is halved from long distance connections to regional connections.

5.2 Building distributed systems with WebRTC

Building a proof-of-concept Distributed Hash Table with WebRTC has uncovered both positive and negative traits of the stack and the technology.

The tests in real browsers show that WebRTC can consume a high amount of the available CPU. In this project, the test environment was a browser running in Docker, in which a WebRTC application send at average two messages per second. The experiences of running this setup indicate that the browser or the WebRTC implementation has a significant overhead. It does not have a

noticeable impact in a typical setting with the user running a single browser regularly. On the other hand, it has a increased load compared to other web applications and is more unstable. In the development and early experiment, running several instances of the Chord algorithm in different tabs within the same browser often resulted in the browser quitting or crashing.

Even if the load is not noticeable in regular use cases, it can easily affect the test environment and the test result in an experiment as the one described in section 4.2. Even if in most cases it is not noticeable to a regular users. There are known issues with browser implementations of WebRTC using a high amount of CPU power. Those issues lead to poor performance of everything running inside the browser in general. The general experience is that running several browsers on one node works until a certain threshold affected by the CPU power. Both the clock rate and number of cores impacts the number of nodes one can run on a given machine. Since each instance of the Chord peer runs inside its browser, it utilizes multicore and several CPUs in a good way.

While the CPU load was high, the Docker experiments showed that the usage of memory was not any higher than expected from running a web browser. The Chord implementation itself is not memory intensive, and the expected load on the experiments running in real browsers indicates that the handling of WebRTC messages is not memory intensive either.

Nonetheless, WebRTC is a new technology, and it has not been supported by browsers for long. The specification is not finished; it is still a draft at World Wide Web Consortium (W3C). Therefore, it is expected that the technology will evolve in the near future. As more and more take use of it the browser implementations will become more battle proven and, thus, have a higher chance of a better performance.

The performance is also affected by the fact that JavaScript is single-threaded and asynchronous operations are handled after the current running block has yielded. Hence, blocking operations affects the asynchronous operation in the way that they can not complete until the blocking operation does. The Chord implementation does not have any blocking operations since everything is wrapped in promises. However the stack below might have blocking operations that even if they are called by asynchronous code it will result in the blocking of the event-loop, which handles event-callbacks. Hence, it can affect the performance of sending and receiving messages, e.g. a background task like `fixFingers` can slow down the query of a key location.

6 | Conclusion

This research project has shown that the concept of distributed systems built on top of Web Real-Time Communication (WebRTC) is feasible, but the technology is still young. WebRTC has the potential to be a platform for a new set of peer-to-peer systems in the future. Furthermore the results from the project indicate that using geographical knowledge can boost performance in such peer-to-peer systems. The sections below evaluates the results in the light of the research questions presented in chapter 1.

RQ1 Is WebRTC suitable for an distributed hash table algorithm, such as Chord?

WebRTC is a good platform to develop distributed peer-to-peer systems. However, not all peer-to-peer systems fit in the web stack, and it does not make sense to put all kinds of peer-to-peer systems in a browser. However, there are use cases that are appropriate to solve in the browser environment WebRTC is suitable. If this applies to a Distributed Hash Table is deemed by the same criteria. There are multiple use cases for Distributed Hash Tables in browsers, and there are situations where they are not so applicable. A use case for an DHT in the web browser environment is coordination of peer-to-peer Content Delivery Network (CDN) or other similar cache networks.

Nonetheless, the technology is still in its early stages and changes fast. Thus, the performance of an application on top of this technology and how to implement them has a high probability of chance. On the other hand, as with all new technology each day brings it closer to a more standardized and stable state.

RQ2 Can the geographical location enhance routing performance in a Distributed Hash Table?

Using the geographical location to enhance the performance of routing in a Distributed Hash Table is possible. The strategy used to utilize the knowledge of geographical location will affect the performance increase in the routing. This project proposed to change the closest proceeding node lookup to use

knowledge of the geographical location to enhance routing performance. The strategy was to choose the peer one step farther away in the finger table in the case where it was geographically closer. The changes in the routing results in fewer long distance round-trips in queries over large keyspaces. Decreased amount of long distance round-trips results in lowered lookup times. Also, it reduces the probability of package loss.

Another related question is whether a web browser is a correct layer for routing optimization. In some cases, it might because the information is not available on a lower level. Also, adding a high-level routing optimizer does not do the job of the ones on a lower level. Both optimizers will work and make the chosen route better. The routing in Chord is on a high level, and it is not as the routing in the network layer it is routing in the network of nodes and their structure in the virtual network. Thus, if there is application logic affecting the routing, the routing optimization can be placed on the same level as that logic. In this case, the `findSuccessor` is that kind of logic, and hence it is appropriate to put the optimization of that routing logic on the same level.

RQ3 How to build a reasonable test environment for peer-to-peer application built with WebRTC?

In the development of peer-to-peer applications with WebRTC, it is important to run different types of tests. There are three major branches of tests that apply: unit-testing, simulations and testing in real browsers. Each of the different types has their benefit, from which a reasonable test strategy can be derived. The unit-tests are to ensure that isolated parts of the application works as expected under different circumstances. The simulation combines integration testing between components of the application with performance testing. The applications can be tested in different situations with known factors and known side-effects. The testing in real browsers can be divided into two different parts: manual correctness tests and full application integration and performance test. The first is to test the application between to local browsers. The second is the most interesting part. Testing integration and performance in a real setting in real browsers running inside Docker containers gives the ability to evaluate how the application performs and behaves when used in a real browser setting. Running those docker containers on fairly cheap virtual servers lowers the cost of a full-blown test of a peer-to-peer application on WebRTC. Furthermore, it makes it possible to test how the application is affected when it runs in browsers on different continents. These tests running inside real browsers in Docker containers are somewhat hard to debug, but having central logging over WebSocket to the server hosting the test was helpful. Also, it was the closest one can get to acquire knowledge of what is going on inside the web browsers in the Docker containers.

6.1 Further work

Better strategy of geographically enhanced routing

The strategy for using the geographical location to enhance routing proposed in this project did not have a large impact on a particular situation. Thus, further work naturally includes researching whether there is a strategy that is even better at exploiting the knowledge of the geographical location to achieve better routing performance. There are several possible options. One is to use a bucket strategy as Dynamo[8] does and replicate buckets to different geographical regions.

Utilizing concurrency of multithreading in browsers

Browsers have one way to achieve better concurrency than the non-blocking event-pattern allows. Web Workers are JavaScript files loaded into and running in their threads. Thus, one script maps to one thread. It might be interesting to research the possibility to make Web Workers handle background tasks.

Contribute to WebRTC and WebRTC libraries

The WebRTC specification and implementations still need a lot of work. The most prominent implementations are open-source, Chromium and Firefox. The knowledge gained from this project could help the development by providing data from the experiments and bug reports of discovered issues.

Furthermore, the WebRTC libraries and frameworks, e.g. Peer.js, also has some issues uncovered by this project. The discoveries have already resulted in contributions to Peer.js. However, there is still unresolved issues with the framework.

Create more peer-to-peer applications with WebRTC

Another way to contribute to WebRTC is to create more applications on top of the API. The best way to test a technology is to use it. The more a technology gets used, the more it is tested and the better it become. Not only will using of WebRTC help the technology forward it will also result in utility modules, e.g. an Remote Procedure Call-module.

Automated tests of peer-to-peer applications

This project has found a good way to orchestrate a test of a WebRTC application with a large set of peers. The natural next step is to automate that process. It should be possible to start a test from one checkout of the code base from

version control with a single script. Furthermore, this could be integrated with a Continuous Integration system, by which would make it possible to evaluate changes to such an application easier. The Continuous Integration system could set up peers in browsers on different nodes and then ran the tests before presenting them to the developer evaluating the change. An automated process like this would be a good tool when developing peer-to-peer WebRTC applications.

Create a new RPC module

The Remote Procedure Call-module created for this project is based on Peer.js because of reasons as time-limit and a need for a signaling server. However, most of the benefits as a unified API for all browsers are benefits that could be provided by an Remote Procedure Call-module. It might be worth researching the possibility to have a Remote Procedure Call-module directly on top of the WebRTC browser stack without any other framework.

Bibliography

- [1] Erik Nygren, RK Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 2010.
- [2] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. International computer science series. Addison-Wesley, 2012.
- [3] Salman a Baset and Henning G Schulzrinne. An Analysis of the Skype Peer to Peer Internet Telephony Protocol. pages 1–11, 2006.
- [4] Johan Pouwelse, Pawel Garbacki, Dick Epema, and Henk Sips. The Bittorrent P2P file-sharing system: Measurements and analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3640 LNCS:205–216, 2005.
- [5] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. *Proceedings of the twenty-first annual symposium on Principles of distributed computing - PODC '02*, (Podc):213, 2002.
- [6] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. *Proceedings First International Conference on Peer-to-Peer Computing*, pages 99–100, 2001.
- [7] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.
- [8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshal, and Werner Vogels. Dynamo: amazon’s highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220. ACM, 2007.

- [9] Avinash Lakshman and P Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44:35–40, 2010.
- [10] Stephan Krause. A case for mutual notification: a survey of P2P protocols for massively multiplayer online games. *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 28–33, 2008.
- [11] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.
- [12] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. *Proceedings of the USENIX Annual Technical Conference 2014*, (June), 2004.
- [13] ECMAScript Language Specification - ECMA-262 Edition 5.1. <http://www.ecma-international.org/ecma-262/5.1/>. Retrieved: 2015-03-03.
- [14] Geolocation API Specification. <http://www.w3.org/TR/geolocation-API/>. Retrieved: 2015-03-31.
- [15] Victoria Pimentel and Bradford G. Nickerson. Communicating and displaying real-time data with WebSocket. *IEEE Internet Computing*, 16(4):45–53, 2012.
- [16] The WebSocket API. <http://www.w3.org/TR/websockets/>, 2012. Retrieved: 2015-05-05.
- [17] Web Storage. <http://www.w3.org/TR/webstorage/>. Retrieved: 2015-04-01.
- [18] Joon S. Park and Ravi Sandhu. Secure cookies on the web. *IEEE Internet Computing*, 4(4):36–44, 2000.
- [19] D Crockford. *JavaScript: The Good Parts*. O’Reilly Media, 2008.
- [20] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.
- [21] io.js - JavaScript I/O. <https://iojs.org/en/faq.html>. Retrieved: 2015-05-03.
- [22] Browserify. <http://browserify.org/>. Retrieved: 2015-04-04.
- [23] Modules Node.js v0.12.2 Manual & Documentation. https://nodejs.org/api/modules.html#modules_module_require_id. Retrieved: 2015-04-04.
- [24] Docker Documentation. <https://docs.docker.com/>. Retrieved: 2015-03-27.

- [25] Jennifer Schommer. Existing Tools for Reproducible Research in Science Docker. pages 1–4, 2014.
- [26] WebRTC 1.0: Real-time Communication Between Browsers. <http://www.w3.org/TR/webrtc/>. Retrieved: 2015-04-04.
- [27] WebRTC. <http://www.webrtc.org/>. Retrieved: 2015-01-27.
- [28] JK Nurminen, AJR Meyn, and Eetu Jalonen. P2P media streaming with HTML5 and WebRTC. In *Computer Communications Workshops*, number 4, pages 63–64, 2013.
- [29] Thomas Sandholm, Boris Magnusson, and Bjorn A Johnsson. On-Demand WebRTC Tunneling in Restricted Networks. *arXiv.org*, cs.NI, 2013.
- [30] Getting Started with WebRTC - HTML5 Rocks. <http://www.html5rocks.com/en/tutorials/webrtc/basics/>. Retrieved: 2015-02-02.
- [31] Cullen Jennings, Justin Uberti, and Eric Rescorla. Javascript Session Establishment Protocol.
- [32] SimpleWebRTC.js. <https://simplewebrtc.com/>. Retrieved: 2015-02-04.
- [33] PeerJS Documentation. <http://peerjs.com/docs/#api>. Retrieved: 2015-04-04.

A | Software packages

The packages and framework described below was developed for this project. They fill a requirement of the project that was not related directly to the research and, therefore, is excluded from the codebase.

A.1 peerjs-rpc

This module is open-sourced and can be found on Github¹ and npm².

Design

This module is designed to work with browserify and exposes its functionality as defined by browserify. It also requires Peer.js with browserify. Thus, Peer.js must be installed via npm. npm will handle this when installing this module.

All methods exposed by this module supports both Promises and callbacks, as described in section 2.4.2. The methods will call the callback with error as the first argument and then the result as the second argument. This is called Error-First callbacks.

RPC(nodeId, scope, options) The constructor takes three arguments: nodeId is the local node id, scope is an object containing functions and variables that should be available to other nodes, options is an object with options for this module and options that are passed on to peerjs.

ping(nodeId, callback) This method will ping the node with a given nodeId and resolve the returned promise and the callback with either **true** or **false** depending on whether the node responds.

¹<https://github.com/relekang/peerjs-rpc>

²<https://www.npmjs.com/package/peerjs-rpc>

attr(nodeId, attrName, callback) This method will fetch the value of an attribute in the given scope on the node with the given `nodeId`. The method will resolve the returned promise and the callback with the value of the attribute. In the case, that the attribute does not exist in the given scope, the promise will be rejected with an error and the callback will be called with the error as the first argument.

invoke(nodeId, functionName, arguments, callback) This method will invoke a function in the given scope on the node with the given `nodeId`. The method will resolve the returned promise and the callback with the value resolved by the promise returned by the function. In the case, that the function does not exist in the given scope, the promise will be rejected with an error and the callback will be called with the error as the first argument. The **arguments** argument should be a list and if no arguments should be passed to invoke it should be an empty list.

A.2 peerjs-mock

This module is open-sourced and can be found on Github³ and npm⁴.

Design

This module makes it possible to test code depending on `peer.js` in an environment without WebRTC. It is designed for Node.js and io.js environments or Browserify based browser environments. It should be applied on import level by using a mocking module for the `require` function in Node.js, io.js or Browserify. It simulates `peer.js` by creating its own `DataConnection` object and keeping a register of available connections. Thus, the connection objects will use the register to find the correct object and call `receive` on that object in order to send data. Figure A.1 shows the internal calls of `peerjs-mock` when sending of data is initiated.

The module has two objects, `PeerMock` and `DataConnectionMock`, that replaces two objects in `Peer.js`, `Peer` and `DataConnection` respectively. The mock of the `Peer`-object implements the original API of the original object, but it uses the mocked data connection object to handle communication.

Figure A.1: Sequence diagram for a sending process in `peerjs-mock`

³<https://github.com/relekang/peerjs-mock>

⁴<https://www.npmjs.com/package/peerjs-mock>

A.3 peerjs-rpc-mock

This module is open-sourced and can be found on Github⁵ and npm⁶.

Design

This module has the same API as peerjs-rpc described in section A.1. It is designed to mock everything beneath peerjs-rpc. Thus, it isolates the algorithm from WebRTC and Peer.js.

The package overrides the sending and receiving mechanism of peerjs-rpc and uses a single object containing references to all nodes. The sending mechanism calls the receiving mechanism on the peerjs-rpc instance that is supposed to receive the message. In other words, the sending mechanism finds the given node in the object containing all instances of peerjs-rpc objects and calls the mechanism in that instance that handles incoming messages directly. The module is wrapped in a function, which takes an object as an argument. The object is used to store the references to all the nodes. This gives the initiator of the mock the ability to control the node references, e.g. remove an reference from the object to simulate peer-failure.

This module gives full control of the communication between peers. Thus, it is possible to create predictable situations by adding delays. The control over delays between different nodes is important when creating a simulation that is supposed to test the performance of changes in an algorithm in environments with different delays between different nodes.

Furthermore, mocking on a higher level than peerjs-mock can be good to avoid unnecessary allocation of resources, which might result in the ability to run simulations with a larger number of nodes.

Usage

```
1   var RPC = require('peerjs-rpc-mock')({});  
2   var n1 = new RPC('n1', scope);
```

Figure A.2: Example of using peerjs-rpc-mock directly

The module is installed from NPM with `npm install peerjs-rpc-mock`. Then it can either be used directly as shown in figure A.2 or put in as mock in require as shown in figure A.3.

⁵<https://github.com/relekang/peerjs-rpc-mock>

⁶<https://www.npmjs.com/package/peerjs-rpc-mock>

```

1   var mr = require('mock-require');
2   mr('peerjs-rpc', require('peerjs-rpc-mock')({}));
3   var YourModule = require('your-module');

```

Figure A.3: Example of using peerjs-rpc-mock with require-mock

A.4 rpc-dashboard

This module is open-sourced and can be found on Github⁷.

Design

The dashboard is designed to be a reusable dashboard renderer, by which can be used to interact with several peerjs-rpc peers. On instance of the dashboard takes a given set of options, as listed below in the usage section. The dashboard will connect to all peers from a list of identification strings and then invoke a function on each peer and store the result in an object. Furthermore, it will render a template passed as an option and call a function **onRendered** passed as an option. After rendering the newly rendered template will replace the content of an element in the DOM. The template will get a list of the outputs of the invoked function from the peers in addition to extra context variables passed as an option.

The dashboard will create an RPC instance, from peerjs-rpc, for the dashboard with a unique ID prefixed with dashboard. Thus, if the network it is connecting to initiate actions on connect, the application should filter peers prefixed with dashboard.

Usage

Figure A.4 shows a code example of a general usage of the rpc-dashboard module.

Options

container CSS selector of the element which should contain the dashboard.

rpcOptions Options-object passed to the peerjs-rpc module.

fetchClients A function that returns a list of RPC-peer identification string or a promise that will resolve such a list.

template A compiled Handlebar-template.

⁷<https://github.com/relekang/rpc-dashboard>

```
1   var createDashboard = require('rpc-dashboard').init;
2
3   var dashboard = createDashboard({
4     container: '#container',
5     rpcOptions: require('./rpc-options'),
6     func: 'toJSON',
7     fetchClients: function() { return ['peer1', 'peer2']; },
8     template: require('./templates/peers.handlebars'),
9     onRendered: function onRendered(utils) {
10       // rewire listeners
11     }
12   });
```

Figure A.4: Example of using rpc-dashboard

onRendered A callback function that will be called when the template is rendered.

peerComparator A comparator function used in `Array.sort()`.

B | List of technologies

This chapters lists short descriptions for every framework and technology used in this project that are relevant to this report.

Docker A platform for deploying and running isolated processes. Section 2.5 describes the platform and how it compares to regular virtual machines.

xvfb Virtual X window management system. It can be used to run application that requires a window management systems on machines without displays, e.g. web-browsers in a Docker image.

Selenium Utility that controls browsers. It can simulate user behaviour in a browser. Selenium controls the test browsers in the docker containers in this project.

JavaScript frameworks

Created for this project

peerjs-rpc Remote Procedure Call module on top of Peer.js. See section A.1

peerjs-rpc-mock Mock of the peerjs-rpc module. A module designed for simulations. See section A.3

peerjs-mock Mock of the Peer.js module. It mocks the connections between two peers. See section A.2

rpc-dashboard A reusable dashboard to view stats from peers using peerjs-rpc. See section A.4

Third-party

Node.js Serverside JavaScript framework. Makes it possible to run JavaScript with Chromes V8 engine directly on an OS instead of in the browser. See section 2.4.4 for more detailed description.

io.js A fork of Node.js. It is for the same purpose as Node.js, but is more up to date. See section 2.4.4 for more detailed description.

Browserify A framework for dependency injection in client-side JavaScript which utilises the CommonJS pattern for dependency management, which is the same as Node.js.

Bluebird A promise implementation in JavaScript. It is described in the Promise-pattern section in chapter 2.4.

Peer.js A wrapper around the WebRTC API that creates a unified interface for all supported browsers and handles signaling of connections. More detailed description is in section 2.6

Mocha Unit test framework for JavaScript. It runs by default on Node.js, but can also run in the browser.

Karma Karma is a test runner that runs JavaScript unit tests in a browser. It can run tests written with several test frameworks in most browsers.

Docker images

This project used two Docker images for browser testing. Both images takes two arguments: an url and a timeout. The browser will visit the url and stay open for the amount of time specified in the timeout.

relekang/firefox-webrtc A Docker image running latest stable version of Firefox, version 37 at the time of the last build of the Docker image. It is available on Github¹ and on the Docker registry².

¹<https://github.com/relekang/docker-webrtc-test>

²<https://registry.hub.docker.com/u/relekang/firefox-webrtc/>

relekang/chrome-webrtc A Docker image running latest stable version of Chrome, version 42 at the time of the last build of the Docker image. It is available on Github³ and on the Docker registry⁴.

³<https://github.com/relekang/docker-webrtc-test>

⁴<https://registry.hub.docker.com/u/relekang/chrome-webrtc/>