CHALMERS

Rymd

Distributed encrypted peer-to-peer storage

Bachelor's thesis in Computer Science

NIKLAS ANDRÉASSON
ROBIN ANDERSSON
JOHANNES RINGMARK
JOHAN BROOK
ROBERT EDSTRÖM

Department of Computer Science and Engineering
Division of Software Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2014
Bachelor's thesis 2014:01

BACHELOR'S THESIS IN COMPUTER SCIENCE

Rymd

Distributed encrypted peer-to-peer storage

NIKLAS ANDRÉASSON
ROBIN ANDERSSON
JOHANNES RINGMARK
JOHAN BROOK
ROBERT EDSTRÖM

Department of Computer Science and Engineering
Division of Software Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2014

Rymd
Distributed encrypted peer-to-peer storage
NIKLAS ANDRÉASSON
ROBIN ANDERSSON
JOHANNES RINGMARK
JOHAN BROOK
ROBERT EDSTRÖM

Bachelor's thesis 2014:01
ISSN 1654-4676
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Göteborg, Sweden 2014

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Keywords: Some stuff, More stuff, Stuff

Sammanfattning

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Terminology

General terminology

Adobe PhoneGap A software enabling development of cross-platform hybrid smartphone applications - applications developed using web technologies, but packaged as native smartphone binaries.

API Application Programming Interface. An interface that software developers can use to get easy access to data and/or particular functionality for their software. Typically exposed as a software library or HTTP service.

Bitcoin The first and biggest widespread cryptocurrency.

CA Certificate Authority. A third party that is trusted to verify the validity of public keys and certificates.

Chrome Apps Similar to Adobe PhoneGap, but for desktop applications running in a Google Chrome sandbox.

Cloud storage A service that hosts data externally with seamless access over the internet.

Cryptocurrency A network transaction system that uses a fully distributed cryptographically secured ledger (called *blockchain*) to track transactions. The vast majority of cryptocurrencies are forks off Bitcoin.

Centralized system A system which has several nodes connecting to and depending on one or a few central endpoints.

Decentralized system A system where responsibilities are shared across the nodes and does not depend on a single, central endpoint.

DHT Distributed Hash Table. A notion in computer science of a distributed key-value store.

Firefox OS A smartphone operating system where all applications are web-based.

GUID Globally Unique Identifier. Used as pseudo-unique identifiers, such as keys in a database. Usually 128-bit values stored as 32 hexadecimal in groups separated by hyphens.

NoSQL All database systems which are not modelled in tabular relations. Examples are graphs, trees, and key-value stores.

OpenPGP A standard for data encryption and signing, originally coming from the proprietary software Pretty Good Privacy (PGP) and widely spread through the free implementation GPG.

P2P Peer-to-peer. Distributed, direct communication.

PKI Public Key Infrastructure.A system that associates (unique) user identities with their public keys. Typically implemented as a Web of Trust or with one or several CAs. Typically, trusted parties use their private keys to sign the public keys of users to verify the connection between an identity and a public key.

RSA A widely used public-key cryptosystem. As such, it builds on pairs of private and public keys where encryption with one can be reversed by decrypting with the other. This system is asymmetric and the security relies on the practical difficulty of factoring the product of two large prime numbers.

RDBMS Relational Database Management System, a popular type of database management system based on the relational model.

SQL Structured Query Language. A language for managing, querying and manipulating data in relational database systems.

SQL injection A technique for injecting malicious SQL code into the executing database queries in order to, for instance, dump the contents of the database.

XSS Cross Site Scripting is the technique for injecting client side scripts into web pages.

Web of Trust A type of distributed PKI that builds on peer-to-peer trust. The idea is that if Alice trusts Bob, then Bob is trusted introduce new identities and public keys for Alice.

Terms with specific meaning in the Rymd project

Identity A unique, memorable string identifying a user within the network.

Module A delimited area of interest and functionality in system architecture.

Node A client in the network (such as a web browser).

Resource A file or folder in the network (a thing that can be shared between nodes)

Rymd The developer library for web based peer-to-peer sharing – the main product. Is also the Swedish word for *space*, which encompass the main ideas of the project.

Shuttle A web based file sharing application implemented using Rymd to demonstrate the basic capabilities of the project.

Contents

1 Introduction

In a day and age where people all over the world own more than one digital device, there is a growing need for services which let users easily store, access, sync and share personal files. Due to the Internet it is possible to store data *in the cloud* and access it from a web browser or a designated client. Instead of storing files on physical media, it is today a common practice to use services such as Dropbox, Google Drive or Apple's iCloud for home and business matters. In November 2013, Dropbox hit 200 million users [1]. Google Drive is integrated across a large number of Google's products, as is the case with Apple's iCloud, which stores and syncs personal preferences across devices and applications.

The existing mainstream services mentioned above are centralized, which means the files and resources stored with them are placed on central servers somewhere on the Internet. This report describes the results of developing a decentralized file sharing system with focus on security and open web standards.

1.1 Background

Most of the existing technologies and protocols that constitute the internet, as well as the services running on it, are by design decentralized and promote the design of distributed systems. On the application layer, transfers of the hosted large files in peer-to-peer scenarios are to a growing extent conducted in a distributed fashion using the Bittorrent protocol, to the point where it is becoming the de facto approach for many use cases and areas. In others, the transition to distributed transfers and storage of data is still in its infancy. In technology and developer communities, *distributed* and *decentralized* have become buzzwords. The norm today is to use distributed approaches for things such as source version control, data storage, heavy computations and content delivery. However, both users, developers and businesses are moving more and more data to an arbitrary *cloud* on the internet. Companies such as Google and Dropbox provide servers for storage of data: A practice that poses several security concerns. Recent news on government infiltration of these services, as well as the revelation of Microsoft defense of private investigations in users' Hotmail inboxes, raises the issue of centralized storage beyond users' control. The internet itself has always been decentralized, and resembles a conventional graph structure with nodes and paths. By centralizing information and giving up a *thin server-fat client* concept, one deviates from the fundamental idea of a decentralized network.

There is currently a clear transition of user-space applications and services from native binaries to web applications running inside a web browser. Recent initiatives such as Google Chrome Apps, Adobe Phonegap and Mozilla Firefox OS are starting to bridge the gap between *web apps* and *native apps* even more, both for mobile and desktop environments. These applications are implemented using what is casually referred to as *HTML5* or, more accurately, the open web stack – an umbrella term for technologies such as HTML, CSS and JavaScript, which are based on open standards. Web applications are becoming ever-increasingly powerful in areas of software engineering and computer science, even though many standards are still in their infancy. Browser implementation and support is still unstable in many areas, leaving much to cover. Even so, technologies for functionality that was earlier exclusively for native applications are now available for any developer to use in modern, cutting edge web browsers. Notable examples are peer-to-peer video chat, local file storage, powerful encryption methods, and real time full-duplex communication.

Following these trends, a natural consequence is a peer-to-peer distributed data-syncing protocol implemented purely on the open web stack utilizing cryptographic keys for access control. This would hopefully act as a stepping stone facilitating the development of user-friendly and convenient, yet secure and privacy-protecting distributed implementations of services such as personal file syncing, media sharing and private communication.

1.2 Purpose

Development of a distributed peer-to-peer encrypted system for storage and communication of resources packaged in a library, with a web browser as the only client-side dependency has been conducted. In this report, the results are presented along with the state of modern web standards as these technologies are researched and analyzed in terms of how they can facilitate realization of such as system.

Rymd is the main outcome and end goal of the project. It will solve authentication and data transfer between nodes, while it also provides encryption and storage of resources locally. It should be usable as a drop-in module by any web client side code, such as a regular front-end web application, browser extension, or widget.

Shuttle is a proof-of-concept prototype using Rymd to show its functionality. It can be seen as an executable evaluation of Rymd and will be briefly discussed in this report.

Below are the main goals and requirements of Rymd:

Privacy . Only users that are given explicit access to a resource should be able to deduce anything useful about its content. No central entity, such as a server administrator or network operator, should be able to extract incriminating information about a client. Users should be able to trust that they know who they are communicating with. No network operator or server administrator should be able to forge identities in a way that can not be detected by a user.

Security . Encryption in all layers, from resource storage to data transfer.

Reliability . If any server goes down and can not be recovered, no damage should be done to the network as a whole as long as anyone can host a new server using the same source code.

Modularization and agnosticism . The system as a whole should not depend on particular implementations for resource storage, key storage or which protocol to use for data transfer. If a developer wants to, they should be able to easily plug in their own alternative implementation module.

Shuttle should be a working example of a file-sharing application that leverages Rymd to provide all these features.

1.3 Problem

There are several sub-problems in a system of this nature that the project needs to address.

- Decentralization of system logic
- Source code integrity
- Peer identity verification
- Resource storage
- Resource identification
- Communication flow and transfer initiation

Below, these main points affecting the architecture are presented.

1.3.1 Decentralization of system logic

In a truly distributed system, it is necessary to avoid having crucial system logic and data on a server. The functionality of the system should not rely on availability any central server. Temporary downtime can be accepted as long as servers do not store crucial data and can be replaced with new servers running the same software. Since clients can not know what application their peers are running, all information from other peers must be verified and considered as untrusted.

1.3.2 Source code integrity

Since the system logic is run in a web browser, how can one trust the fact that the client code is not altered between executions? This issue is one of the reasons why a large part of the online community is considering client-side JavaScript encryption to be a generally bad practice. At the time of this writing, major web browsers have no way to verify client-side code or resources the way that native binaries or Java applets can be cryptographically signature checked.

1.3.3 Peer identity verification

Each user of the system will be associated with a self-generated private-public pair of cryptographic RSA keys. With knowledge of the public keys of their peers, there are standardized identity verification protocols used on a session-to-session basis. Regardless of the authentication protocol used, there is always a chicken-and-egg problem with the distribution of public keys and how to tie them to identities. In order to trust the validity of the key provided from another entity, the user puts trust in that entity. Traditionally, there are two types of Public Key Infrastructures (PKIs) with different ways to address this:

- A Web of Trust, as often utilized in OpenPGP [2]. Here, a user has a list of peers that they trust - trusted introducers. If they receive a public key and associated identity signed by one of their trusted introducers, they will know that the trusted introducer has verified the connection between the identity and the public key. This way, an active user will steadily grow their network of trusted introducers. One needs to have a network of dependable and active peers in order to successfully participate in a Web of Trust.
- A PKI centered around one or several Certificate Authorities (CAs). Here, there is a predefined list of authorities that are trusted to sign participants public keys. This creates a centralized network and puts a lot of trust in the CAs. SSL utilizes this approach and there are several historical examples of when this trust has been broken (more recently in the Diginotar hack of 2011).

1.3.4 Resource storage

Usability, security and adherence to public web standards are three priorities that make the question of how to locally store resources on clients a difficult one. There are proprietary technologies for mapping resources to files on the local filesystem, which could be very useful – but without cross-platform support, they are considered out of question.

1.3.5 Resource identification

It is desirable for resources to have identifiers that are both memorable, secure, and unique - Zooko's triangle is a concern here as well. However, there are practical limitations on using any current cryptocurrency blockchain here. There is a monetary cost associated with the insertion of a new value, and updates can take a significant amount of time to propagate over the network. These practical issues would make such a system practically unusable. File names are not even close to unique and disclose unnecessary information, should an adversary without the corresponding secret key get hold of an encrypted resource. Since

resources are communicated peer-to-peer, the issue of malicious resource identifier collision attacks becomes negligible since users would have first-hand contact with peers that they trust and can verifiy the identity of. Resource checksums will have to be communicated and verified by peers before accepting a transfer of resource data.

1.3.6 Communication flow and transfer initiation

Once peers have verified each other's identities, the question of how two peers start the transfer of a shared resource arises. This is also an implementation detail in applications using the library and depends on what problems that particular application is intending to solve.

1.4 Scope

The project is scoped to encompass two parts: *Rymd* and *Shuttle* (see 1.2)

The system will not deal with version management, syncing, merging resources, and history. Neither will the upload of the users' keys (used for encrypting data) to the blockchain be solved by Rymd, since it involves payment logistics – a subsystem not in our scope.

Leaking of certain kinds of metadata will not be addressed. Namely, information on who is communicating with whom, since this is a very difficult issue far beyond the scope of this project. Also, network operators will likely be able to make a rough estimate on resource size based on the amount of data transferred, since this is considered a reasonable privacy-performance tradeoff as long as transfers are padded enough that the estimation can only ever be so vague.

Some of the technologies involved in this project are quite recently developed, which means that some of even the latest browsers might lack support. The final product will most likely not work on all types of devices and browsers.

1.5 Similar technologies

There is a plethora of disparate technologies for distributing and syncing data between peers that at first glance may look very similar to Rymd. Below are some more well-known and similar pieces of software. The features described will hopefully highlight how they relate to each other and Rymd.

Bittorrent Sync [3] is a distributed peer-to-peer multi-way file syncing software using the Bittorrent protocol for file transfers. Synced folders are mapped directly to the underlying file system, and each folder is encrypted using a shared secret key. Public-key cryptography is not employed, and there are only closed-source binary clients using Bittorrent, Incs network available. While they do have a developer API, it requires developer keys issued from Bittorrent Inc.

RetroShare [4] markets itself as a Friend-2-Friend decentralized communication platform. It uses GPG to create a Web of Trust between peers. It is, however, a very large project: The application provides file-sharing, instant messaging, discussion forums, e-mail, VoIP and group chat. It is open source and distributed as cross-platform binaries.

ShareFest [5] is a peer-to-peer one-to-many file-sharing web based software using WebRTC data channels. ShareFest can be seen as a more limited and primitive version if what Rymd aims to be: ShareFest can share files over WebRTC channels, but does not accommodate for authentication, persistence or local encryption. It does, however, operate on a mesh network similar to Bittorrent. Other similar WebRTC-based P2P file sharing web applications but without additional cryptographic properties include RTCCopy and ShareDrop.

Freenet [6] is one of the first *darknets*, consisting of a distributed, decentralized data store that uploads files with strong anonymity across a network. Each node in the network also acts as a cache for the content stored in the network. Files are generally split up in parts that are distributed, and when fetching files it is unfeasible to determine the origin and sender of the files. Focusing on anonymity, free speech and plausible deniability, the encryption is done in the communication and storage layers. Because of this design, Freenet is quite slow. Files can be retrieved using the cryptographic key used to upload it. Freenet is free software built with Java.

Tahoe-LAFS [7], or Tahoe Least-Authority Filesystem, is a distributed, encrypted and redundant file system. It distributes encrypted files across a predetermined set of servers and allows sharing of both mutable and immutable files. There is a web-interface, but equal to all other user-interfaces it has to go through a *gateway* where encryption and server-communication is performed. Users will typically run their own gateways and will thus need to accommodate for hosting for them.

Bitmessage [8] is a P2P distributed messaging system intended to replace e-mail. Public keys of all participants are distributed over the entire network, and can be retrieved using their fingerprints (which are used as addresses). In order to send a message, it is encrypted using the receiver's public key and sent to the entire network. Participants try to decrypt every message, and will so be able to retrieve the ones they can decrypt. Messages are stored in the network for two days. There is thus no way to tie messages to senders and recipients.

2 | Methodology

Initially the project was split into two parts: an evaluation phase and an implementation phase. This chapter intends to further detail the results of each phase and how it relates to the end product.

2.1 Evaluation of technologies

The goal of the evaluation phase was to map out the landscape of relevant technologies. Different options were compared to each other in order to analyze strengths and weaknesses in regards to a set of given parameters:

Suitability . How well does the technology suit the needs and demands of the job? Are there any technical limitations?

Maintenance . Is the technology actively maintained? If not, does it pose an issue? What are the future scenarios?

Industry support . Are some unsupported browsers neglible? What are the industry's current opinions?

Research was made about open web technologies, mainly belonging to the HTML5 standard. The usage of open web technologies was a requirement of the project's end product, Rymd, which meant that no native code could be written as part of the system and that the quality of the product would be completely dependent on the state of existing APIs and tools of web development. Thus research was made in order to survey the landscape of existing technologies at the time in order to determine which, if any, fulfilled the requirements so that the end product actually could be developed using them.

A set of areas were created, where each area connected with one or several core problems stated within the project. In each area, evaluation and comparisons were made, which included researching APIs and prototyping actual test cases implementing isolated forms of future system features. The research areas were divided as follows:

Data storage . Investigated how to store data locally for each node. This was crucial in order to meet with the overall requirement of building a decentralized system.

Communication . Reviewed the possibilities for communicating and sending data with peer-to-peer technology between two nodes.

Authentication and Permissions . How to solve authentication for nodes.

Prototyping . Rapidly produced a rough test case for sending a file from one node to another.

2.2 Prototyping

The aim for the prototyping area was to quickly decide if it was in any way possible to achieve the requirements with the technologies chosen. Thus was a rough prototype of Rymd and Shuttle created, which implemented two basic test cases: storing a file in the chosen data storage implementation, and sending that file to another node where it was stored in that node's local data storage. The prototype worked successfully, which validated the choice of those special technologies.

2.3 Implementation

During the implementation phase, which stretched from the end of the evaluation to the end of the project, the product was implemented according to the requirements. It was early decided that the implementation process would lightly apply agile methodologies. For this project, this included having a Product Backlog with User Stories, working in sprints, and having bi-weekly Scrum-meetings where current state and eventual problems were brought up. For managing the stories in the backlog, the online management tool Pivotal Tracker was used.

All source code was managed by the distributed source versioning system git[1] and with the online tool GitHub[2]. It was decided to split up the Rymd library into several smaller code repositories ("modules"), which all resided on GitHub. See section 4 for details regarding the technical implementation.

[1] http://git-scm.com/
[2] https://github.com/rymdjs

3 | System design

This chapter describes the technical foundations of the system and how the problems described in section 1.3 have been addressed in Rymd.

3.1 Peer identity verification

For a truly decentralized system, it is not acceptable to adapt a CA-entered approach. While a Web of Trust is interesting, it might be too cumbersome for users. This issue is addressed in "Zooko's Triangle" (See figure 3.1), stating that no system assigning names to participants in a network can have the property that names are secure, decentralized and meaningful at the same time. This conjecture has since been proven false by the design of systems such as the blockchain of the cryptocurrency Namecoin, which effectively acts as a cryptographically secured distrubuted hash table (DHT) with unique keys. Users can reserve a name and assign to it a value of their choice by the cost of a small amount of the Namecoin currency (at the time of this writing 0.01 NMC [9], which equals roughly 0.03 USD [10]).

Rymd therefore utilizes a DHT for storage of keys to achieve all of these goals: The distributed nature of cryptocurrencies makes it decentralized; peers can choose their own names (identities), giving meaningful names; the small monetary fee required to register a name makes it both secure and prevents massive name-squatting by malicious third parties.

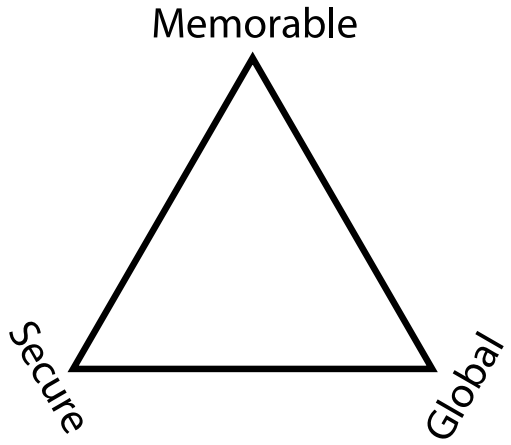

Figure 3.1: *Zooko's Triangle, with the edges representing the achievable combinations of features [11]*

Once a key distribution scheme has been established, an authentication scheme needs to be determined. There are several schemes for authentication using public-key cryptography. Among these are Otway Rees (not mutual, MITM attack exists), Wide Mouth Frog (depends on timestamps), and Needham-Schroeder. Of the ones surveyed, Needham-Schroeder stood out as simple to implement since it does not utilize symmetric keys or timestamps, while it provides mutual authentication.

In 1995, Gavin Lowe described a man-in-the-middle attack on the protocol where an adversary that can initiate a session with one party can then pose as that party when communicating with a third party. Schroeder also proposed a fix to this vulnerability, and this amended *Needham-Schroeder-Lowe protocol* presented below is what Rymd utilizes for authentication.

3.2 Decentralization

Assuming that the system can utilize a DHT such as a cryptocurrency blockchain for storage of the public part of RSA key pairs, the issue of how to interface a web application with the blockchain in a way that allows for verification of identities without putting too much trust in the HTTP/cryptocurrency gateway also needs to be addressed. Additionally, as previously stated, the initial insertion of the key requires monetary resources, and is something that should be solved outside of Rymd. Users can either provide their existing keys and identity to Rymd or let Rymd generate a new pair of keys and manually insert the public part in the DHT of choice. While the public key can be stored in a DHT, private keys need to be stored securely on each client, preferably without giving client code any direct access to the raw keys. Finally, a secure way to store the encryption keys for encrypted resources needs to be addressed.

3.3 Creation and transfers of resources

Full access to a resource implies possession of three things: The encrypted resource data, they cryptographic key used to encrypt said data and the metadata describing the resource. The creation of a new resource goes like this:

1. Metadata is generated. Metadata consists of resource name, author identity, MIME type, a randomly generated GUID, incrementing file version (always 1 in the case of new resource) and a timestamp.
2. A resource-specific symmetric cryptographic key is generated. In the default implementation, a 256 bit AES-CBC key is used.
3. The resource data is encrypted using the resource key.
4. A resource hash is calculated based on the metadata and encrypted data.
5. Add the hash to the metadata.
6. Combine the metadata and the encrypted data to form the internal representation of the resource.
7. Save the key in a local key store.

To save a resource locally, a user-specific symmetric *master key*, generated at the time of first access, is used to encrypt the metadata before saving it and the encrypted resource data to a local resource store.

Generally, the metadata and the encrypted data are transferred separately - maybe even from separate peers. Therefore the hash is important to verify the integrity of the encrypted data. Resource IDs and hashes could also be communicated via trusted channels outside of Rymd, for example on web pages or via e-mail. Once again, verification of resource integrity and authenticity can be achieved by verifying the hash. It is vital that a secure enough hashing algorithm is used; MD5, which used to be the de-facto standard for generating file checksums, has been proven to be weak and contain vulnerabilities to the extent where checksum collisions are too easy to generate. SHA-1 has for some time been recommended for verification of data integrity, but due to theoretical collision attacks and advances in computational capabilities, the U.S. government currently recommends against SHA-1 for applications that require collision resistance[12]. In the default Rymd implementation, the superceding SHA-256 algorithm is used. This gives a strong protection while avoiding the computational overhead of e.g. SHA-512. Note that hashing provides message integrity, but not authentication (establishment of author). This could be established by letting the author of the resource sign the hash with their private key. Peers on the receiving end could then verify the signature using the author's public key. Establishing resource authentication has not been considered a main goal of Rymd and this functionality is therefore not (yet) implemented.

The metadata and resource data are handled separately in Rymd and the communication flow will differ depending on the implementing application. Generally, the metadata will be shared with trusted peers to allow them to decrypt the resource. The encrypted resource can be shared freely since possession of the key is required to make anything useful from it. This way untrusted peers could help facilitate transfers of resources in distributed systems. In the example implementation Shuttle, file sharing is initiated from the sharing end. Consider the case when Bob wishes to share a resource with Alice (assuming both Alice and Bob are already connected to the network):

1. Bob requests Alice's public key and endpoint ID's from the DHT.
2. Bob initiates a connection with Alice and they are mutually authenticated. This process is described below in 3.1.
3. Bob sends the metadata and key for the resource to Alice.
4. Alice creates a new resource like described above, but without the encrypted data, and saves it to her data store.
5. Alice requests the resource data from Bob.
6. Bob sends the encrypted resource data to Alice.
7. Alice adds the encrypted data to the resource and saves it to the data store.
8. When Alice wants to access the resource, she decrypts it.

3.4 Modularity

As stated in section 1.2, developers should be able to easily incorporate their own preferred implementations of the system's core functionality. Accomplishing this would allow developers to not only supply more fitting modules to their end products, but also exchanging existing ones if better alternatives were to be released. This has been particularily important in Rymd since it utilizes technologies at the web's furthest frontier, meaning unfinished drafts in constant change. While these technologies caused a lot of problems, e.g. certain features suddenly breaking because of an update, they supplied the neccessary functionalty required for Rymd's key features to work.

For modularity to be properly fulfilled, these features had to be identified and separated into individual modules; Currently, keys, metadata and resource data need more work to be completely separated. However, data storage, cryptography, communication and DHT usage are provided dynamically through dependency injection.

Furthermore, a system with separate interchangeable parts allowed for both advantages and disatvantages. Easier testing was one of the former, where instead of a suite covering the whole system, tests could be limited to each module. Something that proved to be more difficult was debugging flow of events covering multiple modules. Data would travel through connecting end points between modules and then be sent far down call hierarchies (perhaps triggering new events or having more information appended to itself) before bubbling back up in the call chain. Any error occuring in such a flow were hard to pinpoint.

4 System implementation

This chapter details the underlying implementation of the modules of Rymd and Shuttle (Data Storage, Communication, Authentication, Encryption), as well as documenting and motivating the different choices of required technologies. The latter is a crucial part in the development of web application systems: composing appropriate technologies and tools for the goal based on given criterias.

Standards and technologies for web applications are being rapidly developed, and the boundaries for what is possible to achieve in a web application is being continously pushed by browser vendors and standards groups. Web based applications are today a viable alternative, unless the application is for mobile and is in need for heavy graphics performance, like for photo/video editing and games, or basically everywhere where sophisticated memory management is required. In *Why Mobile Web Apps Are Slow* [13] it is argumented how Javascript is not yet suitable for heavy performing applications, but this is not an issue for this project, as it is not included in the product's nature or scope at all.

Of relevance to Rymd, the Web Real-Time Communications (WebRTC) protocol [14] has become usable for arbitrary data streams in major browsers in 2014. Also of relevance, the still unfinalized Web Crypto API [15] is currently available in an experimental stage in recent months at the time of this writing. The goals of Rymd has thus become technically viable in a web environment as of very recently.

The development of client side data storage in HTML5 is also an area that has become more stable and supported across browsers and vendors. Web applications can utilize offline storage like databases (WebSQL and IndexedDB), key value stores (LocalStorage), and even offline caching with Application Cache [16] in order to build completely offline applications with no requirements for internet access.

Furthermore, there is currently a lot of work underway in the field of distributed secret communication. Notable projects that share similar ideas or have inspired Rymd are mentioned under section 1.5. Also worth noting is the new field of cryptocurrencies, which work by distributing a cryptographically based ledger over an entire network. The first and most well-known is Bitcoin [17], but there are also subsequent currencies that extend the original idea beyond that of a traditional currency to a system that can be used for a wide range of applications. The first worth noting is Namecoin [18], which adds a global and secure key-value store. Another interesting initiative is Ethereum [19], a cryptocurrency with contracts that not only allow storage of arbitrary data in the blockchain, but can also be scripted with a Turing complete programming language and can therefore be used to implement arbitrary systems. A system like Ethereum could be very interesting to explore for a project like Rymd, but it is still in such an early stage that it is deemed too unstable to be useful at this point. Namecoin is currently considered a good candidate for key distribution. Keybase [20] is another recent initiative that intends to solve the distribution of public keys. It is essentially a HTTP-interface that maps keys to identities. While commandable, it again raises the issue of centralized storage. Systems relying on Keybase put a lot of trust on the availability and integrity of their service.

4.1 Data storage

Storage of data is, suffice to say, crucial for any file sharing system. Since the data store was to be used by several parts of the application the demands for the module's interface had to be as general as possible, adhering to a standard CRUD[1] interface, including methods for creating, fetching, updating and deleting records in the store.

There are essentially four alternatives for persisting data on the client:

- LocalStorage
- IndexedDB

[1] Create–Read–Update–Delete

	IndexedDB	WebSQL	File System	LocalStorage
Google Chrome 34	Yes	Yes	Yes	Yes
Mozilla Firefox 29	Yes	No	No	Yes
Apple Safari 7	No	Yes	No	Yes
Opera 20	Yes	Yes	Yes	Yes
Microsoft Internet Explorer 11	Yes	No	No	Yes

Table 4.1: Browser support for selected HTML5 APIs at the time of writing

- WebSQL
- FileSystem API

LocalStorage is included in the HTML5 WebStorage specification [21], and is a basic key-value store with a simplistic API. It is supported across all major browsers and has a maximum storage limit of 5 megabytes. The latter was a deal-breaker, since the product would have to support larger files than could possibly fit into that space. LocalStorage further does not support complex structures and indexing, and storing different data types is complex and needs manual serialization and deserialization. Thus this solution was rejected at once.

IndexedDB and WebSQL are both client side databases and more sophisticated storage solutions than LocalStorage. WebSQL is supported by Google Chrome, Apple Safari (desktop and iOS), Opera and Android. The specification is not longer maintained by W3C [22], and will probably be deprecated on all browsers in the future. IndexedDB is supported by all major browsers except for Safari (desktop and iOS), and is a Candidate Recommendation by W3C [23]. Arbitrary types of data can be stored in the database, such as strings, numbers, Javascript objects, and raw binary data.

The FileSystem API is a collection of methods for reading and writing to a sandboxed filesystem from a browser with client Javascript code. It is a very early standard, and is currently only supported by Google Chrome and Opera, and has the status of Working Draft by W3C [24]. While FileSystem has good performance for larger files and a well-performing asynchronous API, it lacks support for indexing and search. Mozilla seems to have no plans on implementing FileSystem in Firefox [25].

All of the mentioned technologies are sandboxed: the data is tied to a single origin (*http://test.domain.com* for instance). All future access to the data must come from that domain (this includes the protocol and port number as well). The browsers also limits the maximum allowed storage size – the quota. The quota is different for each storage mechanism, and the browser typically asks the user with a dialog if they want to let the app exceed the quota.

The conclusion was to use IndexedDB for persisted resource storage. It was chosen because of its support by Google Chrome, Internet Explorer and Firefox, and due to the fact that its actively maintained (while WebSQL is not). Users of the Safari browser will not be able to utilize the product, but in respect to the project's overall direction in regards to experimental technologies, this is negligible.

4.1.1 IndexedDB

The IndexedDB is a transactional, indexed client side database capable of storing different types of data structures with an asynchronous API. IndexedDB is actively developed and implemented in the latest versions of Mozilla Firefox, Google Chrome, Microsoft Internet Explorer and Opera: its specification is a Candidate Recommendation by the W3C, as of July 2013[23]:

> This document defines APIs for a database of records holding simple values and hierarchical objects. Each record consists of a key and some value. Moreover, the database maintains indexes over records it stores. An application developer directly uses an API to locate records either by their key or by using an index.

A query language can be layered on this API. An indexed database can be implemented using a persistent B-tree data structure.

Basic structure

Due to IndexedDB's object-oriented nature, a database includes a set of *object stores*, which act similar to tables in relational database management systems. An object store can hold *objects* of different types, including binary data and Javascript primitives and objects. Each object has a *key* (either specified by the developer from the object's properties, or automatically generated and managed by the database) that is used for indexing and retrieving records. One or several *indexes* can be created on a store from an object's properties for quick querying. A *cursor* is used to iterate on the resulting set of objects from a query on the store.

The asynchronous API might include complex patterns if the developer is not used to NoSQL structures. Unlike WebSQL, IndexedDB does not support SQL, and instead exposes ways for querying and manipulating data via *requests* and *transactions* (see section 4.1.1). A positive facet of the rejection of SQL in favor of NoSQL is the prevention of SQL injection attacks, but with the cost of a steeper learning curve for already experienced database developers. Queries to the database will not yield the resulting data set: instead requests are returned, which will trigger *events* for when the operation is finished. When an event is triggered a *callback* can be passed to handle the scenario and use the data. This goes well with the asynchronous nature of Javascript, where events and callbacks are used heavily. Using asynchronous passing of callbacks prevents the program execution to block at a line when a potentially heavy operation is called. The Javascript code snippets in listings 4.1 and 4.2 show the difference in synchronous and asynchronous calls.

```
// Fetch a record with id 10 from a database and store in variable
var result = DB.find(10);
```

Listing 4.1: Synchronous call

```
/*
  Request a record with id 10 from a database, continue execute other code,
  and handle result of the database operation in callback when it has finished
*/
var request = DB.find(10);

request.onsuccess = function(evt) {
  var result = evt.result;
};
```

Listing 4.2: Asynchronous call

Security and reliability

IndexedDB is built on a transactional model, which implicates that all commands runs inside a transaction context. Transactions have a certain lifetime, and cannot be used after its expiration. This transactional model is especially useful for when several instances of a client application is using the same database and issuing commands: without transactions, concurrency problems and other collisions might occur with data loss as a result. Transactions are able to abort and be rolled back to the state of the database before the transaction was started.

Kimak, Ellman and Laing highlight four important aspects of securing a IndexedDB driven application in their *An Investigation into Possible Attacks on HTML5 IndexedDB*

and their Prevention[26]:

- Client side data encryption
- Input validation
- SOP (Same-Origin Policy)
- Code analysis

The database in IndexedDB does not include any kind of bundled encryption or validation, which means it is the developer's responsibility to sanitize and encrypt sensitive data before inserting into the store. Encryption is vital for the scenario where the contents of the database is compromised: the attacker must have access to the encryption key in order to read the information in plain text. Validation is needed if malicious content, such as Javascript, is inserted as the data fields in the store and then will be executed at a later stage.

The *Same Origin Policy* is used in IndexedDB. An origin is the transfer protocol, the domain, and the port number. Thus every database is associated with an origin, which implicates certain security aspects: an application in *http://domain.com/subdir* may retrieve data from *http://domain.com/subdir/dir* since they have the same origin, but cannot retrieve data from *https://domain.com:3000* due to the different protocol and port number. This is a layer of protection against Cross Site Scripting (XSS) attacks, even though there is no prevention against XSS holes in the other parts of the application (the database might be compromised due to malicious scripts injected elsewhere).

Code analysis is divided into *static* and *dynamic* analysis. Static analysis is the analysis of the to-be insterted data in order to detect malicious material. Dynamic analysis is the analysis of executed programs, and can according to [26] be done by checking the call from the web application to the database and on success, the database operation can be performed.

4.1.2 Implementation

The main task for the data storage module was to abstract away the low-level methods in IndexedDB (the backing store used, see section 4.1.1). By the use of the concept of *promises*[2], the asynchronous, callback-based methods in the IndexedDB API could be made very streamlined and simple to manage. An example can be found in listing 4.3

```
var Store = new IndexedDbStore("myStore")

// Fetch all records as an array
Store.all().then(function(records) { ... })

// Create a record
Store.create("A record").then(function(record){ ... })

// Insert a record
Store.save("A record").then(function(guid){ ... })

// Fetch a record by GUID
Store.get(guid).then(function(record) { ... })

// Delete a record by GUID
Store.destroy(guid).then(function(record) { ... })
```

Listing 4.3: Common database operations

[2]http://en.wikipedia.org/wiki/Promise_(programming)

The largest challenge came to the edge cases when storing files, or as they are called in web browser: *Blobs*. Since only Firefox can as of now store blobs directly in IndexedDB, an alternate route had to be taken for other browsers. Initially the module used conditionals and converted incoming data to and from *ArrayBuffers* (the browser construct for raw byte streams). But since ArrayBuffers are just the raw data, all metadata for the blobs (such as filename, timestamps, size) would be lost when saving as an ArrayBuffer. In early versions of the data storage module this metadata would be stored in a separate store in the database, but this was too tight coupled and was removed. The final implementation is storing data as-is – any metadata must be saved explicitly in a separate operation.

The data storage module is a separate repository and the source is available at https://github.com/rymdjs/data-storage.

4.2 Communication

In order for nodes to be able to share data, they need a way to connect to each other. They also need to to this in a secure manner in order to prevent potential vicious third parties listening on a connection from making any sense of retrieved data. I.e. critical parts should not be sent in raw form but rather be encrypted. When considering security aspects there are essentially three questions that need to be answered regarding the issue of connecting nodes:

- How can a node find another node to begin with (peer discovery)?
- When a node has been found, how can a connection be established?
- What data needs to be encrypted in order to ensure the integrity of the system?

Answering these questions and finding the corresponding best technology was the focus of this research area. In recent years there has been a trend driving more and more of tools and services on the web towards user collaboration. A natural step in this trend is initiatives such as WebRTC and CU-RT-Web which enables direct peer-to-peer communication.

WebRTC seeks to be a common standard for browsers with W3C drafting client side APIs and IETF developing the protocols and peer-to-peer communication[27]. There are also security aspects built into WebRTC. The major browser vendors Google, Mozilla and Opera support the project [28]. While Microsoft actually supports the concept of WebRTC and contributes to the W3C WebRTC working group, it does not support Google's (or nowdays, W3C's and IETF) version of it[28].

Microsoft warns about supporting the new technology until it has actually has become a standard and also doesn't fully agree on some constraints put on the technology[28]. Microsoft explains that one of their issues with the current WebRTC version is that it has predetermined paths of choosing codecs and ways of sending media over the network – sort of similar to a black box. This hinders application developers wanting to optimize to suit their own needs. Microsoft's answer to this is their own CU-RTC-WEB (Customizable, Ubiquitous Real Time Communication over the Web) which tries to addresses these issues.

All in all Microsoft remains optimistic that a common standard will eventually be established[28]. They do however stress that more participants need to get involved besides Google and Mozilla. Since all major browser vendors (except Microsoft) support WebRTC at this time we chose this technology for the project.

4.2.1 WebRTC

WebRTC is a project which enables real-time communication between browsers[14]. Through the project, developers are able to create different types of applications which leverage peer-to-peer technology.

To obtain and transfer streaming data WebRTC's functionality is abstracted into three different APIs: *MediaStream*, *RTCPeerConnection* and *RTCDataChannel*[29]. MediaStream, or *getUserMedia*, handles synchronized media streams, i.e. synchronized video and sound from

a computer's camera and microphone. RTCPeerConnection manages reliable and efficient communication of the data streams. The API utilize techniques such as *jitter buffering* and *echo cancellation* to ensure a high standard even in unstable networks. Furthermore, when a peer connection is present arbitrary data can be sent by leveraging RTCPeerConnection with RTCDataChannel.

RTCDataChannel

The RTCDataChannel API demonstrated the capabilities that Rymd desired. Data transfers are secured with the DTLS (Datagram Transport Layer Security) protocol. The DTLS protocol is based on the TLS (Transport Layer Security) protocol, the main difference being that DTLS is constructed for datagrams while TLS is used for more reliable transport protocols such as TCP.

Before a connection can be initiated between peers, one of two parts must extend an offer which contains data describing the connection to the other part - this is often referred to as the signaling phase. The signaling phase requires a channel where the offer can be negotiated - it is usual for the channel to be a dedicated signaling server, but examples of a more serverless approach can be found[30]. The standard does not provide any recommendations regarding the choice of signaling channel and protocol - this is for developers to decide. The connection phase is then handled by RTCPeerConnection which manages the ICE (Interactive Connectivity Establishment) workflow.

NAT, STUN, TURN and ICE

A problem with the construction of peer-to-peer applications using WebRTC, is the presence of NATs (Network Address Translation). NATs were first introduced in RFC 1631 as a short term solution to the problems regarding IP address depletion in IPv4. The idea was that by utilizing NATs, several hosts in a private network could together share a single public IP address [31].

A NAT is responsible for maintaining a table with entries that maps an internal IP address and port, to a public IP address and port; and dropping these entries when they are no longer of relevance. When a host behind a NAT wants to communicate with an external host, the NAT creates an entry in the table, which can then be used to route the response back to the internal host.

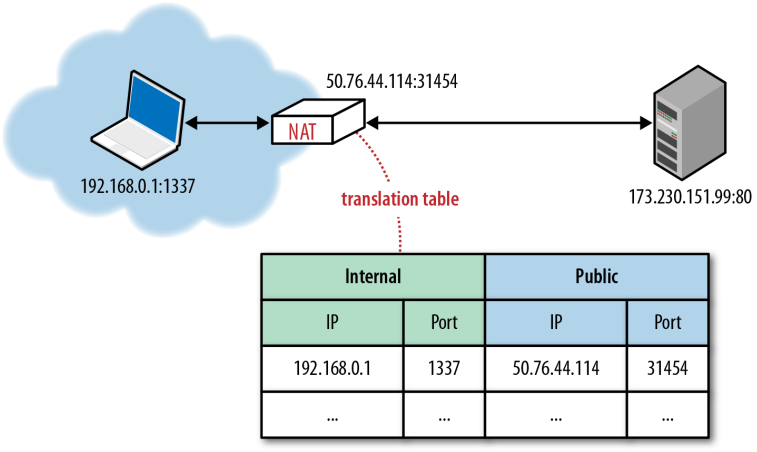

Figure 4.1: *A NAT maintains a table mapping internal IP and port, to a public IP and port. [32].*

The reason to why the presence of NATs can pose a problem for peer-to-peer communication in the browser, is that WebRTC mainly leverage the UDP protocol for transportation of data. The underlying issue with the UDP protocol is the absense of state, as opposed to the TCP protocol. The statelessness of the UDP protocol makes is difficult for a NAT to

determine when a table entry is no longer relevant and should be dropped, which leads to the fact that UDP routing entries are expired based on time. If an entry is predeterminedly expired, it will cause inbound packets to be dropped, as they can't reach the source. In the case of TCP, which got a well defined state, it is inherently simple to determine when an entry should be dropped.

A technique commonly used to solve the problem regarding UDP entries getting expired and dropped, is to utilize keepalives at regular intervals, which is commonly referred to as *UDP hole punching* [33].

An additional problem is that internal hosts know their internal IP, but not the public one. If a host runs an application, which communicates the IP as a part of the payload to hosts oustide of the private network, there would obviously be a missmatch if the internal IP is chosen. Therefore it is common to utilize a protocol called STUN (Session Traversal Utilities for NAT), which was introduced in RFC 5389. The protocol enables hosts to obtain its public IP address, with the help of an external STUN server (See figure 4.2) [34].

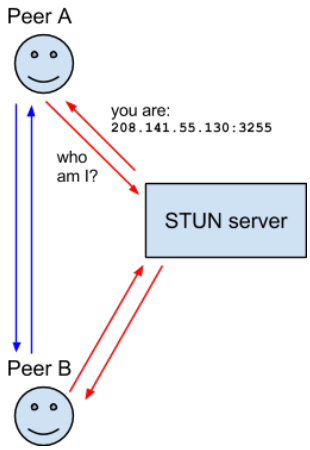

Figure 4.2: *STUN servers lets peers in a private network behind firewalls discover their public IP-addresses[35].*

These techniques are not always enough though, which is caused by the fact that STUN does not work with all types of NATs, and in some cases UDP traffic might be blocked by a firewall. In these cases the TURN (Traversal Using Relays around NAT) protocol, specified in RFC 5766, can be leveraged. TURN establish a TCP connection with a relay server in those cases where UDP fails. The relay server is used to tunnel data to the other host, which means that there is no longer a direct peer-to-peer connection (See figure 4.3) [36].

In the case of WebRTC, the techniques desribed above are leveraged through the ICE (Interactive Connection Establishment) protocol which is used by the RTCPeerConnection API. The ICE protocol is specified by RFC 5245 and has the objective of connecting peers in the most efficient way, it makes use of STUN and fallback to TURN when no other alternatives exists (See figure 4.4) [37].

4.2.2 Implementation

Rymd leverages the open source project PeerJS[3], which simplifies sending peer-to-peer data between clients. PeerJS utilize WebRTC and is essentially split into two components: A server which acts as the signaling channel, and a client side API which interacts with the server as well as other peers. The server only handles the brokering of connections, which implies that only the data necessary for negotiating a connection is sent through this point. For communication between the server and the clients, Peer.js utilize both WebSockets and XMLHttpRequest [38]. After a connection has been setup between two clients, the server is no longer needed in order for them to communicate.

[3]http://peerjs.com

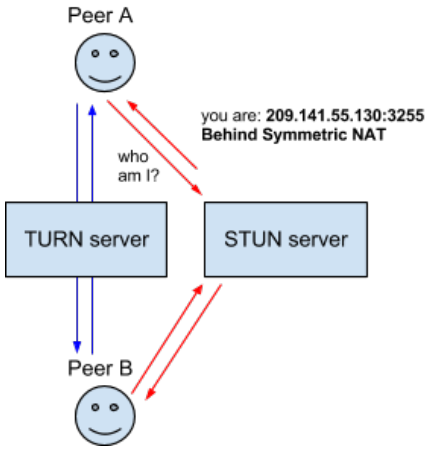

Figure 4.3: *If a peer-to-peer connection cannot be established, a relay through a TURN server could be used. All peers send their packets through the relay which makes it more costly. But at least the connection works[35].*

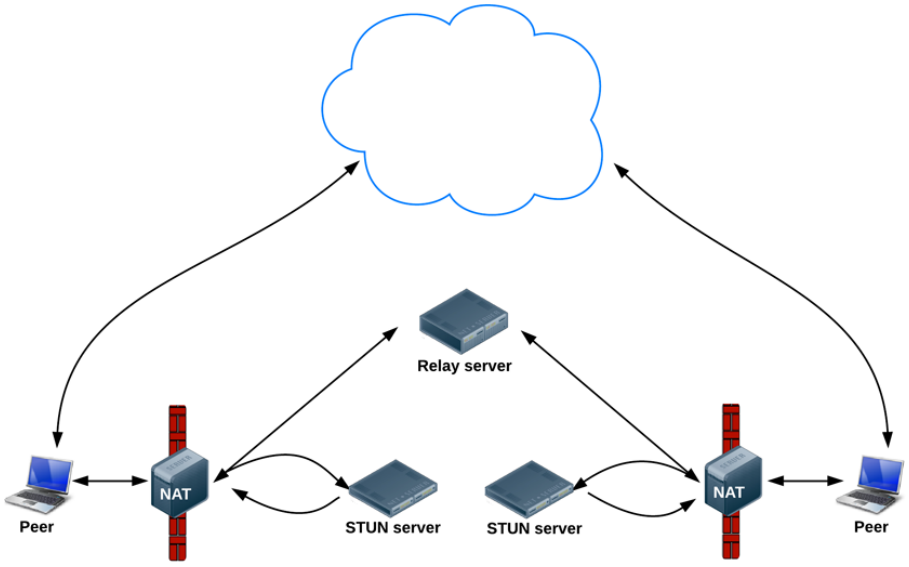

Figure 4.4: *The different ways for ICE to find network interfaces and ports [29]*

The following steps explain how PeerJS broker connections:

1. Two clients connects to the PeerJS server, using the client side API.
2. The server returns unique IDs for each of the clients.
3. One of the clients connect to the other using the client side API, where the unique ID for the other one is provided. The PeerJS server then forwards the information needed to set up a peer-to-peer connection to the other client.

In line with the project guidelines regarding modularity, Peer.js specific code is separated into a module of its own. If the requirements are to change, then those changes are contained to a single module.

4.3 Authentication

4.3.1 Namecoin

A phenomenon that has been on the rise during recent years is that of cryptocurrencies such as Bitcoin [39]. Each participant in the Bitcoin network keeps a ledger of all transactions throughout the history of the network. In order for a transaction to be deemed valid, it needs to be included in a cryptographically signed block together with a salt by one of the nodes, with a hash that has a special format. It is this brute-force search for salts that generate these hashes that are called *mining* and constitute the work done by *miners* to keep the network running. As an incentive, each verified block also includes a reward to the miner that first finds it and submits it to the blockchain[40]. In effect, all transactions ever made are publicly available and tracked so that anyone can confirm their validity. This prevents forgery and double-spending of bitcoins.

Another cryptocurrency is Namecoin[39]. Namecoin is essentially a fork of Bitcoin with new transaction types that allows its blockchain to be utilized as a distributed key-value store. Although similar in nature to Bitcoin, its main purpose is to be used as a decentralized domain name system (DNS), rather than as a monetary currency.

With a decentralized DNS such as Namecoin, top level domains (such as *.com* or *.se*) can exist without being controlled by any central authority [39]. Also, the DNS lookup tables where domain names and their IP addresses are stored are shared in a peer-to-peer manner. The only necessary condition for these domains to be accessible is that there are participants willing to run the DNS server software. Although mainly intended to be used as a DNS, it contains several namespaces where arbitrary strings such as public cryptographic keys can be stored.

4.4 Encryption

Until recently, the practice of performing cryptographic operations in a web browser environment has been considered bad practice by security professionals[41]. One reason for this is that it has been impossible to verify the integrity of client side source code between executions - something that has now changed with the advent of signed browser extensions. Another issue is the internal openness of JavaScript - any cryptographic implementation would unavoidably expose all their primitives, as well as raw private and secret key data. This is being addressed by the new Web Cryptography API[15], an open standard for implementation of cryptographic primitives in web clients accessible through client code. Basically, all primitives (such as hashing, encryption, decryption, signing and generation of keys) and raw key material would be black boxed for the client application and executed natively in the web browser. However, it is not yet finished and at this time of writing only Google Chrome out of the major browsers have implemented more than a basic pseudo-random number generator. Therefore, at this stage Rymd only runs in the very recent development versions of Google Chrome or Chromium. Even if the implementations provided by these browsers supply key generation, there is no support for persisting keys between sessions or

even exporting private keys. Until this is implemented, the Rymd crypto module therefore generates keys on its own; doing this is bad practice and is only performed in order to get a prototype running.

The Encryption uses two algorithms, namely the $RSAES - PKCS1 - v1.5$ and the $AES - CBC$. AES-CBC is a symmetric key AES based using cipher block chaining; it is the only symmetric algorithm implemented at the time of writing, besides it is fast and simple to use. $RSAES - PKCS1 - v1.5$ - an AES implementation based on the RSA algorithm. The fact that it uses RSA is inline with our guidelines since it allows the key to be reused for both signing and encryption. Considering DSA algorithms: the DSA is faster for signing and key generating but slower when it comes to signature checking and it also lack the ability to preform encryption with Regarding accessibility, both algorithms are included in the Web Crypto API and are therefore provided by the browser as black boxes. In commercial terms as for our project, RSA is clearly the winner; commercial RSA certificates are much more widely deployed than DSA certificates. As previously mentioned, the Web Crypto API lacks functionality for storing keys between sessions or even exporting private keys. In order to support key storage between sessions, the key generation is moved out of the API and handled by a library *bignumber-jt* [42]; The key is then parsed to a certificate. All certificate have a static key size where asymmetric keys are of $1024bits$ and symmetric ($AES - CBC$) are of $128bits$; none are explicitly generated via the API.

Furthermore, the asymmetric keys are wrapped in PKCS8 and SPKI certificates while the symmetric key exist in raw format. The Private-Key Information Syntax Standard (PKCS8) - defines a way to store the private key and the Simple public key infrastructure (SPKI) - defines a way to store the public key. All three standards was chosen because they are the only three formats that is currently supported by Chromium [43]. W3C - the organization behind the Web Crypto API, have announced their intention to implement the Web Crypto Key Discovery API [44] a secure key store which would allow Rymd to persist keys between sessions in a secure manner. Web Crypto Key Discovery API was originally thought as a part of the Web Crypto API was extracted in favor overall of decreased implementation complexity.

4.5 Testing

Automatic unit tests have been implemented where possible. No integration or functional tests have been written for testing larger parts of the system. This is due to the fast iteration of the library's interface and constant change in implementation.

5 | Results

This project has resulted in a modular peer-to-peer developer library for sending data, encrypted with public key encryption, over a secure connection to another client. The library, *Rymd*, has composed several modules for specific areas such as data storage (section 4.1), cryptography (section 4.4) and communication (section 4.2.2) in order to create a foundation for sending files without central file storage.

For demonstrating the capabilities of Rymd, a sample prototype web application has been created, named *Shuttle*. This front facing client provides a user interface for showing local files, sending files to other users registered in the blockchain, and managing encryption keys. In Shuttle, the user is capable of adding, listing, viewing and deleting files in their local data store, where the files are encrypted and stored along with their metadata. By knowing a recipient's identity, the user is able to share a file with the recipient over an encrypted P2P connection. When sharing a file, the receiving end will instantly show a notification with a remark that the sender wants to share a file. If the recipient chooses to accept the sharing request, the file will download to their local data store.

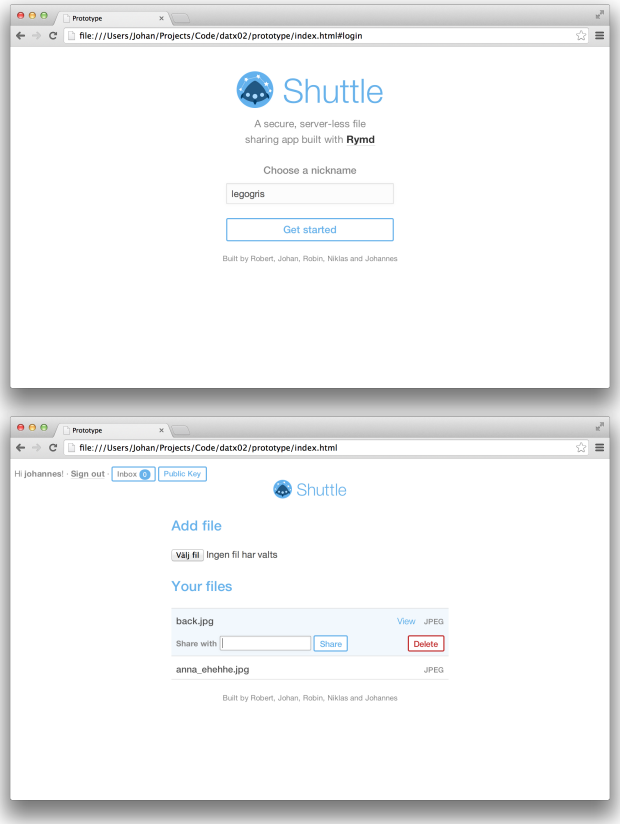

Figure 5.2: *Files listing in Shuttle*

5.1 Source code

Due to the modularity of the system, a number of repositories exist to hold the source code of the different modules. All source code is available at the project's GitHub page: https://github.com/rymdjs. All relevant repositories can be found below:

Rymd https://github.com/rymdjs/rymd

Shuttle https://github.com/rymdjs/prototype

Crypto https://github.com/rymdjs/crypto. Cryptography module.

Data Storage https://github.com/rymdjs/data-storage. IndexedDB adapter.

PeerJS Connection https://github.com/rymdjs/peerjs-connection. Connection adapter for PeerJS for doing P2P.

DHT Client https://github.com/rymdjs/dht-client. Client module for Namecoin lookups.

DHT https://github.com/rymdjs/dht. Node.js HTTP REST adapter for lookups in the Namecoin blockchain.

Rymd Logger https://github.com/rymdjs/rymd-logger. Custom debug and flow logger module.

Rymd Utils https://github.com/rymdjs/rymd-utils. Globally used utility functions.

6 | Discussion

Here, we go through the main goals with Rymd and see how well they could be met, where and why there are shortcomings and how these could be addressed. Finally, we explore the ethical motivation behind Rymd and the implications it could have in a non-technological sense.

6.1 Could a truly decentralized system be achieved?

One of the main initial goals with Rymd was to make the system truly decentralized and reliable independently of the availability of certain services. To a large extent, Rymd is successful in this area. Any application, including the prototype Shuttle, can be downloaded and executed locally. Therefore there are no dependencies on web servers, since all data transfers between clients are performed in a peer-to-peer fashion. No central database outside of the local client stores persistent data. There is, however, two parts where communication with central endpoints still needs to be done:

6.1.1 WebRTC ICE

As described in section 4.2.2, establishing of WebRTC connections still relies on the availability of a STUN or TURN server. This makes implementing applications depend on the availability of such a server. However, there are several public ICE servers available and in the case of downtime it is trivial to set up a new one and make the application use the new server instead. Also, since verification of identities of peers is performed locally and all data is end-to-end encrypted, there is no possibility of the administrator of these servers to spoof identities or deduce anything about shared resources. The two things that do leak are identity names (since these are needed to deduce who to connect to whom) and, if TURN is used, estimated size of data transferred. We found no way around the former and decided the latter to be a fair tradeoff - future implementations that care about leaking of resource size could solve this by also transfer redundant padding data regardless of resource size.

6.1.2 Accessing the DHT

The Namecoin blockchain is used to tie identities to their public keys and PeerJS endpoints. Since Namecoin (or any other currently existing cryptocurrency for that matter) communicates using their own binary data protocol[45], Rymd can not interact directly with the blockchain to fetch this information before a mutual peer-to-peer authenticated connectin is established. Therefore, an HTTP gateway running the Namecoin software was developed that acts as a bridge between the blockchain and Rymd nodes. Trust in the operator of this gateway is crucial, since public keys are fetched and verified through it. The paranoid user could, however, easily run their own gateway or manually verify or insert public keys using their own Namecoin client. On May 6th, at the time of writing of this report, a public and more general HTTP/Blockchain interface at *chain-api.com*[46] was released. Currently it only support Bitcoin, but promises future integration with the Namecoin blockchain. Once that happens, it would be trivial to replace the current gateway with Chain if one would like to do so. The buzz around services like these shows that this is an emerging area with more interesting development to come in the near future.

Something that would both solve these issues and be very interesting in many other areas is a blockchain where nodes can communicate through open web protocols. This would mean that web clients could interact directly with the blockchain without going through external gateways like these, at the same time allowing them to contribute to the network. Given the premises stated in the introduction and the rapid development of emerging blockchains and cryptocurrencies, we think it is only a matter of time before this happens.

6.2 Is all data truly cryptographically secured?

In the application layer, all communication over WebRTC is DTLS encrypted. As stated in section XX, for local storage and sending of resources, every resource is also AES encrypted with a resource-specific key. Since all existing browser implementations of the Web Cryptography API are still experimental and partial and there is not yet support for secure storage of cryptographic keys, the keys are stored alongside their encrypted data in IndexedDB. As long as the keys are not passphrase protected, this effectively means that at the level of the local client, the encryption adds no extra protection and can be considered redundant. An adversary gaining access to the database with the encrypted data would also have access to the decryption key. This is most likely only during a transitional period and as the Web Crypto API becomes stable and fully implemented across browsers, separate key storage options will become available and this can be resolved.

Despite this problem, the AES encryption still serves a purpose. Consider an application utilizing Rymd where users communicate through each other in a *darknet* fashion. In these cases it is imperative that resources can be transmitted separately from their keys and metadata so that intermediate peers can facilitate the transfer of resource data without gaining knowledge of the contents.

Also, systems with updateable resources can and should regenerate keys for each version and backward secrecy - the property that access to the key for one version of the resource will not allow decryption of older versions of that resource - will be achieved.

6.3 Is the system modular and implementation agnostic?

As much as Rymd uses and relies some of the latest web technologies, great care was taken during development to not make it rely on any of these implementations, should they be superceded or complemented by other, more fitting alternativces. The main Rymd library itself handles only the business logic of the system and gets the modules implementing data storage, cryptography, peer-to-peer communication and DHT interaction supplied at runtime via dependency injection. Developers who have their own idea of how these needs should be served in their projects could write their own implementation modules. The one area where work needs to be done is that currently, storage of keys, metadata and resource data are tied together. Before Rymd can go stable, this should be addressed by treating these as separate data stores alltogether even if the current implementation puts all three side by side in IndexedDB.

6.4 Ethical aspects

Rymd was originally conceived from an ethical issue: The one that free, private and secret communication should be easily accessible and usable on the web. As it currently stands, truly secret and private communication requires running binary files and/or putting trust in a service provider. Rymd aims to be a step away from that restriction.

6.4.1 Does Rymd meet this goal?

At its current state, Rymd should not be trusted with confidential data. This is mainly because of the limitations stated in the preceding sections that come from the choice of still immature, cutting-edge technnologies. Also, Rymd is still in an experimental stage and should not be considered stable or trusted until it has been exposed to extensive peer review and scrutiny by the community - a reservation that should be held for any project of this nature. However, we are confident that we are going in the right direction and hope that further development could make Rymd to a contributor in the movement of free communication on the web.

6.4.2 Implications

As always when it comes to services enabling private communication, concerns are raised on the issue of what they can be used for. Commonly mentioned are terrorism, drug dealing and child pornography. First, we want to emphasize that Rymd does not in itself provide any anonymity for its users (though it could easily be used in conjunction with anonymization services such as TOR[1]). While Rymd could indeed be used for these purposes, there are already other services such as the ones mentioned in 1.5 that are currenly used for these purposes - Rymd does not enable risks that are not already present.

Inevitably, the question of which party holds responsibility for villainy is raised when a project of this nature is realized. One could argue that we are the ones responsible since we are supplying end users with the possibility to perform malicious actions. Another claim could be that the government is the one to blame for allowing such services to exist. Our own opinion on this matter is that people are solely responsible for their own actions. Technology exploring new frontiers is always met with critical glaze; laws and regulations have merely briefly halted advancement. One way or another, technological evolution always prevails. Meanwhile, people performing atrocities will probably always find other ways of doing their deeds. It therefore seems resonable to try to stop people performing atrocities than the tecnology currently under the microscope.

Mainly, however, it is our opinion that the right to private and secure communication without corporation or government surveillance is a human right, and this right is effectively nonexistant if it requires significant monetary resources and/or technical know-how. Putting this standpoint aside, we have mainly treated this issue as a technical one and will let the reader of this report decide for themselves where they stand and how Rymd relates to this.

[1] https://www.torproject.org

7 | Conclusion

References

[1] J. Constine. (Nov. 2013). Dropbox Hits 200M Users, Unveils New "For Business"Client Combining Work And Personal Files, [Online]. Available: http://techcrunch.com/2013/11/13/dropbox-hits-200-million-users-and-announces-new-products-for-businesses/.

[2] U. Maurer, "Modelling a public-key infrastructure", *Computer Security — ESORICS 96*, ser. Lecture Notes in Computer Science, E. Bertino, H. Kurth, G. Martella, and E. Montolivo, Eds., vol. 1146, Springer Berlin Heidelberg, 1996, pp. 325–350, ISBN: 978-3-540-61770-9. DOI: 10.1007/3-540-61770-1_45. [Online]. Available: http://dx.doi.org/10.1007/3-540-61770-1_45.

[3] BitTorrent Inc. (Feb. 2014). BitTorrent sync, [Online]. Available: http://getsync.com/.

[4] RetroShare Team. (Feb. 2014). RetroShare, [Online]. Available: http://retroshare.sourceforge.net/team.html.

[5] Peer5. (Feb. 2014). Sharefest, [Online]. Available: https://www.sharefest.me/.

[6] I. Clarke. (Feb. 2014). Freenet, [Online]. Available: https://freenetproject.org/.

[7] Tahoe-LAFS. (Feb. 2014). Tahoe-LAFS, [Online]. Available: https://tahoe-lafs.org/trac/tahoe-lafs.

[8] Bitmessage Community. (Feb. 2014). Bitmessage, [Online]. Available: https://bitmessage.org/wiki/Main_Page.

[9] Namecoin wiki. (Feb. 2014). Register and configure .bit domains, [Online]. Available: https://wiki.namecoin.info/index.php?title=Register_and_Configure_.bit_Domains&oldid=36.

[10] CryptoCoin Charts. (May 2014). Nmc/usd - namecoin / us dollar today charts and orderbook from kraken, [Online]. Available: http://www.cryptocoincharts.info/v2/pair/nmc/usd/kraken/today.

[11] Z. Wilcox-O'Hearn. (Oct. 2001). Names: Distributed, Secure, Human-Readable: Choose Two, [Online]. Available: http://web.archive.org/web/20011020191610/http://zooko.com/distnames.html.

[12] R. Blank and P. D. Gallagher, *Recommendation for key management – part 1: general (revised)*, Published as NIST Special Publication 800-57, http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf, 2012.

[13] D. Crawford. (2013). Why Mobile Web Apps Are Slow, [Online]. Available: http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/.

[14] Google Inc. (2014). WebRTC, [Online]. Available: http://www.webrtc.org/.

[15] M. W. Ryan Sleevi. (2014). W3C Web Cryptography draft, [Online]. Available: http://www.w3.org/TR/WebCryptoAPI/.

[16] I. Hickson. (May 2011). HTML5 – Offline Web Applications, [Online]. Available: http://www.w3.org/TR/2011/WD-html5-20110525/offline.html.

[17] Bitcoin.org. (2014). Bitcoin – Open source P2P money, [Online]. Available: https://bitcoin.org/en/.

[18] Namecoin.info. (2014). Namecoin, [Online]. Available: http://namecoin.info/.

[19] Ethereum.org. (2014). Ethereum, [Online]. Available: https://www.ethereum.org/.

[20] Keybase.io. (2014). Keybase, [Online]. Available: https://keybase.io/.

[21] I. Hickson. (2014). W3C Web Storage draft, [Online]. Available: http://www.w3.org/TR/webstorage/.

[22] ——, (2014). W3C Web SQL Database draft, [Online]. Available: http://www.w3.org/TR/webdatabase/.

[23] E. G. A. P. J. O. J. B. Nikunj Mehta Jonas Sicking. (2014). W3C Indexed Database draft, [Online]. Available: http://www.w3.org/TR/IndexedDB/.

[24] E. Uhrhane. (2014). W3C File API: Directories and System, [Online]. Available: http://www.w3.org/TR/file-system-api/.

[25] J. Sicking. (2014). Why no FileSystem API in Firefox?, [Online]. Available: https://hacks.mozilla.org/2012/07/why-no-filesystem-api-in-firefox/.

[26] D. C. L. Stefan Kimak Dr. Jeremy Ellman. (2014). An Investigation into Possible Attacks on HTML5 IndexedDB and their Prevention, [Online]. Available: http://www.cms.livjm.ac.uk/pgnet2012/Proceedings/Papers/1569607913.pdf.

[27] H. A. François Daoust Dominique Hazaël-Massieux. (2013). Web Real-Time Communications Working Group Charter, [Online]. Available: http://www.w3.org/2011/04/webrtc-charter.html.

[28] J. Roettgers. (2012). Microsoft commits to WebRTC – just not Google's version, [Online]. Available: http://gigaom.com/2012/08/06/microsoft-webrtc-w3c/.

[29] S. Dutton. (2012). Getting Started with WebRTC, [Online]. Available: http://www.html5rocks.com/en/tutorials/webrtc/basics/.

[30] Chris Ball. (May 2013). WebRTC without a signaling server, [Online]. Available: http://blog.printf.net/articles/2013/05/17/webrtc-without-a-signaling-server/.

[31] P. F. Kjeld Egevang. (May 1994). The IP Network Address Translator (NAT), [Online]. Available: http://www.ietf.org/rfc/rfc1631.txt.

[32] I. Grigorik. (2013). High-Performance Browser Networking - UDP and Network Address Translators, [Online]. Available: http://chimera.labs.oreilly.com/books/1230000000545/ch03.html.

[33] D. K. Bryan Ford Pyda Srisuresh. (Feb. 2005). Peer-to-Peer Communication Across Network Address Translators, [Online]. Available: http://www.brynosaurus.com/pub/net/p2pnat/.

[34] P. M. D. W. Jonathan Rosenberg Rohan Mahy. (Oct. 2008). Session Traversal Utilities for NAT (STUN), [Online]. Available: http://tools.ietf.org/html/rfc5766.

[35] C. M. Laurent Jouanneau Jérémie Patonnier. (2014). Introduction to WebRTC architecture, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Guide/API/WebRTC/WebRTC_architecture.

[36] J. R. Rohan Mahy Philip Matthews. (Apr. 2010). Session Traversal Utilities for NAT (STUN), [Online]. Available: http://tools.ietf.org/html/rfc5766.

[37] J. Rosenberg. (Apr. 2010). Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols, [Online]. Available: http://tools.ietf.org/html/rfc5245.

[38] Peer.js. (May 2014). Peer.js - Github project, [Online]. Available: https://github.com/peers/peerjs.

[39] Crypto Coin Insider. (May 2014). Namecoin, [Online]. Available: http://www.cryptocoinsinsider.com/namecoins/.

[40] Paul Gil. (2014). What Are Bitcoins? How Do Bitcoins Work?, [Online]. Available: http://netforbeginners.about.com/od/b/fl/What-Are-Bitcoins-How-Do-Bitcoins-Work.htm.

[41] M. Security. (Apr. 2010). JavaScript Cryptography Considered Harmful, [Online]. Available: http://www.matasano.com/articles/javascript-cryptography/.

[42] T. Wu. (Apr. 2010). bignumber-jt, [Online]. Available: http://www-cs-students.stanford.edu/~tjw/jsbn/.

[43] eroman@chromium.org. (Apr. 2010). WebCrypto implementation in Chromium, [Online]. Available: https://docs.google.com/a/chromium.org/spreadsheet/ccc?key=0Agiw0cuQZfVGdHNUNXBhZEFkazkyVy1uM1pISnlKRWc#gid=0.

[44] N. Mark Watson. (Apr. 2010). WebCrypto Key Discovery, [Online]. Available: http://www.w3.org/TR/webcrypto-key-discovery/.

[45] Bitcoin Project. (May 2014). bitcoind source code, [Online]. Available: https://github.com/bitcoin/bitcoin/.

[46] Chain. (May 2014). Chain - Simple, powerful Blockchain APIs, [Online]. Available: http://chain-api.com/.

[47] David Gilson. (2013). What are Namecoins and .bit domains?, [Online]. Available: http://www.coindesk.com/what-are-namecoins-and-bit-domains/.

Appendices

A | Communication flow

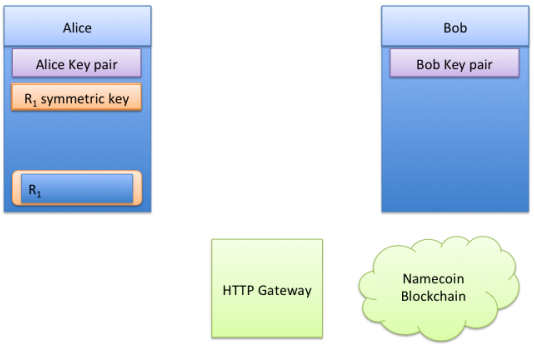

Figure A.1: *Alice and Bob have added each other's identities as* friends. *Alice has a resource R_1 that she wants to share with Bob. R_1 is encrypted with a symmetric secret key.*

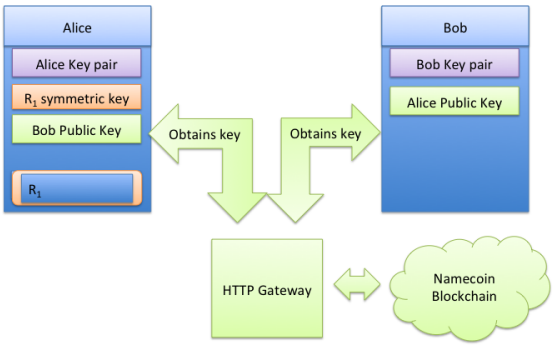

Figure A.2: *Alice and Bob both independently contact an HTTP gateway to get each other's public keys from the DHT. If they do not want to trust a third party, they could set up their own gateway or even add the keys manually.*

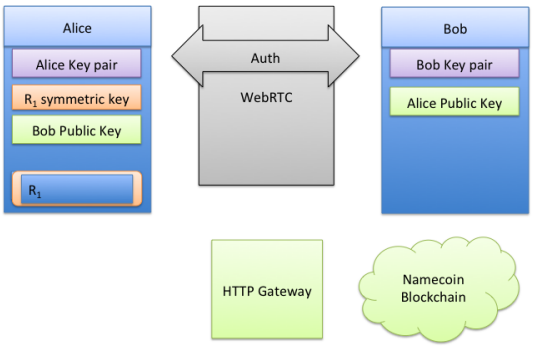

Figure A.3: *Alice and Bob tells a central server that they want to communicate, for example by sending their friend lists. The server sets up a direct WebRTC connection between them. They use this channel to authenticate using standard public key cryptography.*

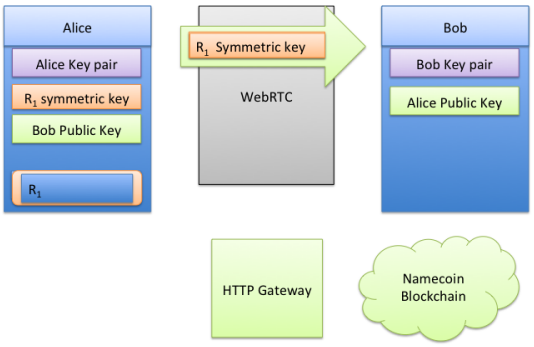

Figure A.4: *Alice wants to share R_1, so she encrypts the key used to encrypt it with Bob's public key and sends it to Bob along with a reference ID that he can use to request it from Alice.*

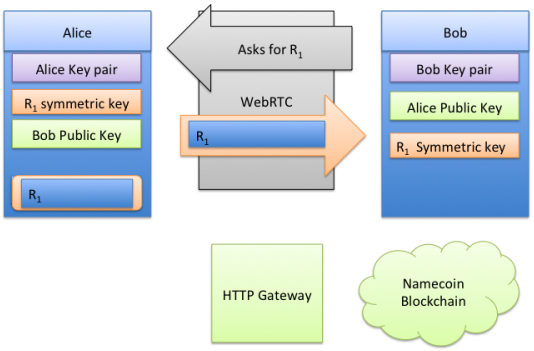

Figure A.5: *Bob asks Alice for R_1. She responds by sending it, still encrypted with the key she just sent to Bob.*

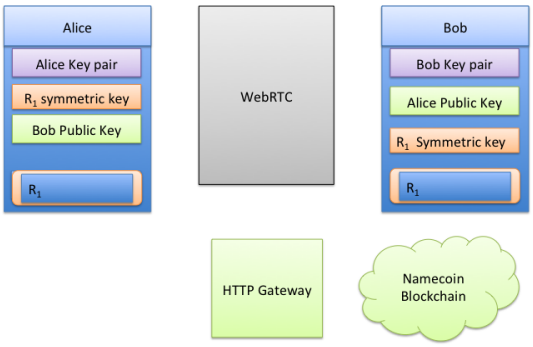

Figure A.6: *The transaction is finished. Bob is now in possession of* R_1 *and the key used to encrypt it.*

B | List of external libraries

Listed below are external tools and libraries that were used in the development of Rymd and Shuttle.

Backbone http://backbonejs.org. Lightweight Javascript frontend framework.

Browserify http://browserify.org. Node style dependency management in the browser.

Chai http://chaijs.com. Assertion library used with the test framework.

gulp http://gulpjs.com. Build system.

jQuery http://jquery.com. Javascript library for DOM manipulation and more.

Mocha http://visionmedia.github.io/mocha. Test framework for Javascript.

PeerJS http://peerjs.com. WebRTC library for P2P communication.

Q http://documentup.com/kriskowal/q. Library for working with promises for asynchronous code.

Underscore http://underscorejs.org. Functional helper toolchain for Javascript.