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 Networks and Systems
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 Networks and Systems
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 Networks and Systems
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Göteborg, Sweden 2014

Abstract

This thesis describes a cryptographically secured decentralized peer-to-peer file sharing system, bundled as a JavaScript developer library called *Rymd*. This includes evaluating current web technologies and determining if they are sufficient to implement such a system.

The problem domain includes how to make the system secure, decentralized and modular using web technologies. Such a system would give control of traditonally centralized services, such as messaging or storage and sharing of files, back to users. By making the system highly modular, we manage to achieve satisfying results in reliability and technical agnosticism with regards to underlying implementations.

Default implementation modules of the core functionality in Rymd are given. Alongside Rymd a web application, *Shuttle*, that injects these modules into Rymd is developed. IndexedDB is used for persistent data and key storage while peer-to-peer communication is achieved through WebRTC. Common cryptographic services such as encryption, decryption, signing and more are done through the Web Cryptography API. The Namecoin blockchain is used for storing user identities mapped to their public encryption keys, which in turn are used to verify peer identities.

Even though this project succeeds with its goal of creating a client-side file sharing platform, the technologies used are in such a premature state that they cannot fully satisfy security goals. We are confident that the standard bodies and browser vendors will continue their work on bringing open web technologies up to speed with their native counterparts.

Keywords: peer-to-peer, distributed, cryptography, file sharing, JavaScript, IndexedDB, WebCrypto, Namecoin

Sammanfattning

Detta projekt syftar till ta fram ett modulärt bibliotek i JavaScript som är avsett för utvecklare och går under namnet *Rymd*. Biblioteket tillhandahåller säker filöverföring direkt mellan webbläsare. Projektet innefattar utvärdering av moderna webbtekniker för att säkerställa de mest lämpade alternativen.

Huvudproblematiken kretsar kring hur systemet kan göras säkert, decentraliserat och modulärt med hjälp av moderna webbteknologier. Ett sådant system kan användas för att öka användares kontrol över traditionellt centraliserade tjänster, såsom chatt och filsynkning.

Som demonstration av Rymds funktionalitet skapades även en webbapplikation vid namn *Shuttle*, där givna implementationer för kärnfunktionaliteten i Rymd är givna. För datalagring används IndexedDB medan peer-to-peer-kommunikationen utnyttjar WebRTC. Vidare används det ej färdigställda Web Cryptography API för kryptografiska operationer såsom kryptering, dekryptering, signering, med mera. För att lagra kryptografiska nycklar används kryptovalutan Namecoins så kallade *blockchain* där publika nycklar som används för verifiering av identiteter kan hämtas ut med hjälp av användaralias.

Projektet mynnade i slutändan ut i en fungerande fildelningsplattform, med vissa brister i säkerheten. Dessa brister kan direkt härledas till det tidiga utvecklingsstadiet i de webbteknologier som används. Då webben utvecklas i en rasande takt av både webbläsare och standardiseringsorgan är vi dock säkra på att detta rättas till så småningom.

Terminology

General terminology and abbreviations

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. (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.

CRUD Create-Read-Update-Delete. A set of actions to be taken on data collections.

Cryptocurrency A network transaction system that uses a fully distributed cryptographically secured ledger

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.

IETF Internet Engineering Task Force. An organization with the purpose of improving the internet by creating standards.

JSON JavaScript Object Notation, a lightweight alternative to XML for exchanging data (often over different APIs). JSON has become a common way for formatting data, and most languages have native implementations for parsing and serializing JSON.

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 between clients.

PKCS A group of standards for public-key cryptography devised and published by RSA Security Inc.

PKCS#8 Private-Key Information Syntax Standard, used to carry private certificate keypairs and provide a way to construct private key certificates in ASN1.

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.

REST Representational State Transfer, a style used for structuring data APIs by putting constraints on the different URL endpoints. If an API uses REST style, it is often referred to as *RESTful*.

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.

SPKI Simple public key infrastructure, a successor to X.509. It was designed with the goal to eliminate overcomplication and scalability problems.

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.

XHR XMLHttpRequest is a web browser JavaScript API used for sending asynchronous HTTP requests directly from the client.

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

X.509 an ITU-T standard for a public key infrastructure (PKI).

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.

W3C World Wide Web Consortium. A standards organization for the World Wide Web.

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 [1] [2], there is a growing need for services which let users easily store, access, synchronize, and share personal files. The internet makes it 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 reached 200 million users [3]. Google Drive is integrated across a large number of Google's products, as is the case with Apple's iCloud, which stores and synchronizes personal preferences across devices and applications [4].

The existing mainstream services mentioned above are centralized, which means that the files and resources stored with them are placed on central servers somewhere on the internet. This report describes the results of developing a file sharing system that runs independently on each participant's computer and doesn't rely on any central server - a decentralized system. The system will build on the preconception that direct connections are only made between peers that have pre-existing knowledge of each other. Such networks could still allow friend-to-friend propagation of data and searches in several steps to connect users to vast networks of resources through chains of friends of friends. Focus will be 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 [5]. On the application layer, the hosting of large files are to a growing extent conducted in a distributed peer-to-peer 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, users, developers, and businesses alike are moving more and more data to an arbitrary *cloud* on the internet. Companies such as Google and Dropbox provide servers for data storage: An approach 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 a users' Hotmail inboxes [6], 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.

Simultaneously, 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 applications* and *native applications* 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 defined by open standards. Web applications are becoming 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 synchronization, media sharing, and private communication.

1.2 Purpose

The purpose of this project is a cryptographically secured distributed peer-to-peer system for storage and communication of data resources (e.g. files). It should be bundled as a developer library with focus on modularity, decentralization, and security. The goal is to determine if such a system can be implemented solely with web technologies and become usable for scenarios such as file sharing between clients. Such a system should fulfill the following requirements:

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 cannot be detected by users.

Security Encryption in each step, from resource storage to data transfer. Only users with explicit access to a resource should be able to read it.

Reliability If any server goes down and cannot 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 implementation 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.

1.3 Problem

With background of the purpose and requirements, there are a set of concrete problems that a web-based file sharing project needs to address:

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

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 central server. The functionality of the system should not rely on the availability of any specific server. If servers are needed for any reason, they should not store persistant or sensitive data and be easily replacable with new servers running the same software. Temporary downtime can be accepted in this case. Since clients do not know what software their peers are running, all information from them must be considered untrusted until verified.

1.3.2 Peer identity verification

The stated requirement of privacy can only be assured if the identities of peers can be verified. This is generally done with public-key cryptography, where each user is associated with a pair of asymmetric cryptographic 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 [7]. 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. In 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 exists 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).

1.3.3 Resource storage

Usability, security and adherence to public web standards are three highly prioritized properties that make the question of how to locally store resources on clients a difficult one. The FileSystem API [8] enables access to the local filesystem. Some browsers have simple implementations in place, but the standard is now considered dead [9]. Local file access could be very useful – but without cross-platform support it is considered out of the question. A secure way to store the encryption keys for encrypted resources also needs to be determined.

1.3.4 Resource identification

It is desirable for resources to have identifiers that are memorable, secure, and unique. Resource checksums will have to be communicated and verified by peers before accepting a transfer of resource data.

1.3.5 Communication flow and transfer initiation

In order for nodes to be able to share data, they need a way to connect to each other. They also need to do this in a secure manner in order to prevent potential vicious third parties listening on a connection from making any sense of retrieved data. In other words, critical parts should not be sent in raw form but rather be encrypted. When also 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 system's integrity?

1.4 Scope

The project has developed two end products: *Rymd* and *Shuttle*:

Rymd is the main outcome and end goal of the project. It should solve authentication and data transfer between nodes, while also providing 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. Shuttle should be a working example of a file-sharing application that leverages Rymd to provide all these features.

The system will not deal with version management, synchronization, merging resources, or history. Neither will the issue of leaking of certain kinds of metadata be addressed. This includes information on who is communicating with whom, since this is a very difficult issue far beyond the scope of this project. Unfortunately, network operators will likely be able to make a rough estimate of resource size based on the amount of data transferred. This is considered a reasonable privacy-performance tradeoff as long as transfers are padded enough so that the estimation can only ever be so vague.

Rymd will only handle connections and transfers of data between peers with pre-existing knowledge of each other, and the issue of searching for files hosted by unknown peers will therefore not be within the scope of this project. The system should, however, leave the door open for implementing applications to construct propagating search to allow peers connected through some degrees of separation to exchange files.

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

1.5 Structure

Chapter 2 describes the different parts of this project methodology wise. The main part of this report can be viewed as a three step process: a theoretical background; a design, evaluation and analysis chapter; and an implementation chapter.

In chapter 3, fundamental information about relevant theories and technologies is given in order to supply the reader an understanding of the field. Chapter 4 surveys the current landscape and positions this project among other related work. An analysis and overall system design is described in chapter 5 where the underlying modules are specified. The low-level implementations of these are described in chapter 6.

The final part of this report involves discussion and conclusion in chapter 7 and 8, where the results of the project are presented and discussed.

2 | Methodology

In this chapter we describe how the project was executed by splitting it in two parts: an evaluation phase and an implementation phase. During the evaluation, research was made regarding relevant technologies, and included rapid prototyping to quickly test the technologies for the project's use cases. Finally, the methodologies and modularization of the final implementation are briefly presented.

2.1 Evaluation of technologies

The goal of the evaluation phase was to map out the landscape of relevant technologies. Different options were compared with 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 done concerning open web technologies, mainly those 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 for web development. Thus research was done in order to survey the landscape of existing technologies in order to determine which, if any, fulfilled the requirements so that the end product actually could be developed using them. The result of this research is presented in chapter 3 with the resulting analysis and discussion in chapter 5.

A set of areas was created, where each area was 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 How to store data locally on the client.

Communication Possibilities for communicating and sending data with peer-to-peer technology between two nodes.

Authentication and Permissions How to solve authentication between nodes.

Prototyping The development of a rough test case for sending a file from one node to another.

2.1.1 Prototyping

The aim for the prototyping phase was to quickly decide if it was in any way possible to achieve the requirements with the technologies chosen. Therefore a rough prototype of Rymd and Shuttle was 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 the particular technologies used. The choices that proved successful were carried over to the next step.

2.2 Implementation

At an early stage it was decided that the implementation process would apply light 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.

All source code was managed by the distributed source versioning system git[1] and hosted at the online service GitHub[2] (links to all source code is available in appendice A).

2.2.1 Modularity

As stated in section 1.2, developers should be able to easily incorporate their own preferred implementations of the system's core functionality. For modularity to be properly fulfilled, features that could have alternative implementations had to be clearly identified and separated into individual code repositories referred to as *modules*.

Accomplishing this would allow developers to not only supply more fitting modules to their own end products but also easily exchange existing ones if better alternatives were to be released. This has been particularly important in Rymd since it utilizes technologies at the web's furthest frontier; unfinished drafts in constant change.

[1] http://git-scm.com/

[2] https://github.com/rymdjs

3 | Technical background

This chapter gives a theoretical foundation and an overview of the current state of the field for the technical domains that are of relevance for this project: client-side storage, distributed storage, communication, and cryptography. In all fields, standards and technologies for web applications are being rapidly developed and the boundaries for what is possible to achieve in a web application are being continuously pushed by browser vendors and standardization groups. The goals of Rymd has indeed become technically viable in a web environment as of very recently.

The development of client-side data storage in HTML5 is an area that has become more stable and supported across browsers and vendors. Web applications can utilize offline storage such as databases (WebSQL and IndexedDB), key-value stores (Web Storage), and even access to the local file system (FileSystem). Under section 3.1, all of these technologies are presented.

Some of the new *cryptocurrencies* can be utilized for distributed storage of data such as cryptographic keys. Namecoin and Ethereum are notable examples. In section 3.2, they are presented together with Keybase, a service specialized for this purpose that uses a different approach of verifying identities through links at social media accounts.

There has been a steady progression in the introduction of communication protocols available for web applications. From client-server HTTP requests (XMLHttpRequest) via client-server full-duplex TCP connections (WebSocket), it has recently become possible with peer-to-peer real-time communication (WebRTC). In section 3.3, these technologies are presented. Also, issues with NAT traversal in peer-to-peer communication and how they are addressed in WebRTC are explained.

Also of interest, the still unfinalized WebCrypto API [10] has become available in an experimental stage in recent months. In section3.4, the basics of public-key cryptography and certificates are explained, together with the Web Cryptography API and the Advanced Encryption Standard as part of a symmetric encryption scheme.

3.1 Client-side storage

Client-side storage is how arbitrary data can be persisted on disk, accessible through an API, by a web browser. This is a general term for several separate but related APIs:

Web Storage [11] is a simple key-value store in the HTML5 specification.

WebSQL Database [12] is an embedded SQL relational database.

FileSystem [8] is an API providing direct file system access.

Indexed Database [13], or *IndexedDB*, is a NoSQL asynchronous data store.

All of these technologies offer ways of storing data on the user's hard drive instead of on a remote server. There are two main reasons for this: to make the application available offline and to improve performance (fewer requests to a server). Older storage techniques include cookies, plugin based storage (Java Applets, Flash, Google Gears) or browser-specific features.

All four APIs tie data to a single *origin*, a practice referred to as *Same Origin Policy*. An origin is defined by the transfer protocol, the domain, and the port number of a website. Thus every data store 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* attacks (XSS). XSS is a general term for when a middleman injects malicious code in a web page viewed by others. Note that the Same Origin Policy in the data storage layer is no prevention against XSS holes in the other parts of the application, since a user might be attacked from malicious scripts injected elsewhere in the application.

In order to prevent malicious flooding of users' hard drives, browsers impose limits on storage capacity. If the application exceeds that limit, the browser typically show a dialog in the interface to let the user increase the the limit. This quota is separated for each origin and storage mechanism. For instance, `sub.domain.com` may be allowed to store 5MB of Web Storage and 25MB of IndexedDB data, while `sub2.domain.com` may have other restrictions.

Both of the database-centered technologies, IndexedDB and WebSQL, support *transactions*. This ensures the integrity of the database by preventing of *race conditions*, a phenomenon where two sequences of operations are applied at the same time, leading to unpredictable results and a database state of dubious accuracy. This is done by locking the database for writing until a sequence of commands are finished.

Most of the storage formats support synchronous and asynchronous modes. Synchronous mode is blocking, meaning that the storage operation will be executed and completed before the next line of code is executed. Asynchronous operations are non-blocking, performed in the background while the rest of the code is executed, and may be completed at a later stage. Generally a *callback function* is provided and called on completion. This is the traditional approach to work with asynchronous operations in JavaScript, where events and callbacks are used heavily. The JavaScript code snippets in listings 3.1 and 3.2 show the difference between synchronous and asynchronous calls.

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

Listing 3.1: 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 3.2: Asynchronous call

3.1.1 Web Storage

Web Storage is for persisting data in key-value pairs through a single object in web browsers. The API is as simple as attaching values as strings on properties of the global `localStorage` object as seen in listing 3.3. The `localStorage` object persists data through browser sessions, while the `sessionStorage` object will clear all data when the browser tab or window is closed.

```
// Save an item in the local store
localStorage.foo = "bar";
// or
localStorage.setItem("foo", "bar");

// Retrieve item
var val = localStorage.foo;
// or
var val = localStorage.getItem("foo");

// Delete item
localStorage.removeItem("foo");
```

Listing 3.3: Use of Web Storage

3.1.2 WebSQL

WebSQL is the only client-side storage technology mentioned that tries to mimic a traditional SQL relational database. It comes with tables, indexing, transactions, keys, and support for schemas. Regular SQL expressions are used to interact with the database, which means the developer can rely on the vast research that has been made in SQL query optimization (see listing 3.4). WebSQL is high-performing thanks to indexing. Developers who are used to work with traditional databases can start using it in a familiar manner. However, the use of SQL makes the database vulnarable to SQL injection attacks, unless this is taken into account by the developer.

```
// Create or open database with name, version, description and size of 2MB
var db = openDatabase('testdb', '1.0', 'Test database', 2 * 1024 * 1024);

// Create table and insert data
db.transaction(function(tx) {
  tx.executeSql('CREATE TABLE IF NOT EXISTS NAMES (id unique, first, second)');
  tx.executeSql('INSERT INTO LOGS (id, first, second) VALUES (1, "Johan", "Brook")');
});

// Retrieve all names and print them
db.transaction(function(tx) {
   tx.executeSql('SELECT * FROM NAMES', [], function(tx, results) {
     var len = results.rows.length, item;

     for (var i = 0; i < len; i++){
       item = results.rows.item(i);
       console.log("Name: " + item.first + " " + item.second);
     }
  }, null);
});
```

Listing 3.4: Use of WebSQL

3.1.3 FileSystem

The FileSystem API allows for read and write access of files and folders on the user's hard drive. Currently, only Google Chrome has a working implementation of the API. FileSystem is suitable for storing larger binary files, and has good performance thanks to its asynchronous structure. There is no support for transactions or indexing. Its API includes methods for manipulating files and folders, as one would expect from a standard file system (see listing 3.5).

```
// Requesting a sandboxed, persistent file system with a size of 2MB and success callback
window.requestFileSystem(window.PERSISTENT, 2 * 1024 * 1024, function(fs) {
  console.log("Opened filesystem: " + fs.name);

  // Create a text file
  fs.root.getFile("test.txt", { create: true, exclusive: true }, function(fileEntry) {
    console.log("Created " + fileEntry.name + " in " + fileEntry.fullPath);
    // => "Created test.txt in /test.txt"
  });

  // Reading a file
  fs.root.getFile("test.txt", {}, function(fileEntry) {

    // Get a File object and read its contents with FileReader.
    fileEntry.file(function(file) {
      var reader = new FileReader();

      reader.onloadend = function() {
        var contents = this.result;

        console.log(contents);
      };

      reader.readAsText(file);
    });
  });

  // Other methods of FileEntry:
  fileEntry.getMetadata(successCallback, opt_errorCallback);
  fileEntry.remove(successCallback, opt_errorCallback);
  fileEntry.moveTo(dirEntry, opt_newName, opt_successCallback, opt_errorCallback);
  fileEntry.copyTo(dirEntry, opt_newName, opt_successCallback, opt_errorCallback);
  fileEntry.getParent(successCallback, opt_errorCallback);
  fileEntry.toURL(opt_mimeType);

  fileEntry.createWriter(successCallback, opt_errorCallback);
});
```

Listing 3.5: Use of FileSystem

3.1.4 IndexedDB

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 [13].

Basic structure

Due to IndexedDB's object-oriented nature, a database includes a set of *object stores*, which act similarly 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 objects 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 objects 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 has patterns that might be daunting and seem complex to developers 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 3.1.4). A positive side of the rejection of SQL is the prevention of SQL injection attacks. This comes at the cost of a steeper learning curve for database developers already experienced with more traditional databases. Queries to the database will not yield the resulting data set. Instead requests are returned, which will trigger *events* when the operation is finished. When an event is triggered a callback can be passed to handle the scenario and use the data. See figure 3.6 for common use of IndexedDB.

```
var request = indexedDB.open("testdatabase");

// On database version migrations
request.onupgradeneeded = function(event) {
  var db = event.target.result;

  // Create an objectStore for this database
  var objectStore = db.createObjectStore("store");
};

// When the database is ready to use
request.onsuccess = function(event) {
  var db = request.result;

  // Insert "foo" in the store
  var insertTransaction = db.transaction(["store"], "readwrite").objectStore("store").put("Foo");

  insertTransaction.onsuccess = function(evt) {
    console.log("Inserted ’foo’");
  };

  // Read value with key "key" from the store
  var readTransaction = db.transaction(["store"], "read").objectStore("store").get("key");
  readTransaction.onsuccess = function(evt) {
    console.log("Found " + evt.target.result);
  };
};
```

Listing 3.6: Use of IndexedDB

Security and reliability

IndexedDB is built on a transactional model. This means that all commands run inside a transaction context. Transactions have a certain lifetime and cannot be used after they expire. This transactional model is especially useful for when several instances of a client application are 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 roll back the database to the state it was in before the transaction was started, should an error occur.

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* [14]:

- 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 that it is the developer's responsibility to sanitize and encrypt sensitive data before insertion into the store. Encryption is vital for the scenario where the contents of the database are compromised, since the attacker would need access to the encryption key in order to read the information in plaintext. Validation is needed in order to prevent malicious content, such as XSS, from being inserted as the data fields in the store (without the user's knowledge). Properly crafted code could otherwise pose a security risk since vulnerable applications could execute it at a later stage.

Code analysis is divided into *static* and *dynamic* analysis. Static analysis is the analysis of the to-be inserted data in order to detect malicious material. Dynamic analysis is the analysis of executed programs, and can be done by checking the call from the web application to the database. On success, the database operation s allowed to fully execute [14].

3.2 Distributed storage

With the inherent problems of trust in CAs in a PKI, several approaches to publicly available distribution of cryptographic keys have emerged in recent years. Some use a blockchain-based approach derived from Bitcoin. This implies distributing a cryptographically based ledger over an entire network and taking it beyond that of a monetary currency to systems that can be used for a wider range of applications. There are also other ideas on how to solve the trust issue.

3.2.1 Namecoin

A phenomenon that has been on the rise during recent years is that of cryptocurrencies such as Bitcoin [15]. A cryptocurrency is a virtual currency that builds on cryptographical principles to ensure integrity and consistency. Each participant in the Bitcoin network keeps a ledger of all transactions throughout the history of the network. This ledger is called the *blockchain*, because it is a chain of *blocks*. Each block constitutes of the hash of the previously generated block, a number of other recent transactions and a salt. In order for a transaction to be deemed valid, it needs to be included in one of these blocks. This inclusion is done by volunteering nodes that perform a brute-force search for a salt that generates a block hash of a specific form. When such a salt has been found, the block is included in the blockchain. This search for salts is called *mining* and constitutes the work done by *miners* to keep the network running. As an incentive, each verified block also includes a reward to the miner that finds the salt. 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.

Namecoin [16], another cryptocurrency, 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 [17]. 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.

3.2.2 Ethereum

Bitcoin and its derivatives have a built-in scripting language that runs on top of the blockchain. It has limited functionality and is mainly used to set up contracts and multi-party-signed transactions that enables escrow-like functionality. Ethereum [18] is a novel crypto-currency built from scratch that extends this idea by building on *smart contracts* using a Turing complete domain specific language. This makes it a platform for buildig arbitrary distributed system. Users running code pay a small fee of the internal currency *ether* for each computational step. Ethereum could therefore work not only as a DHT, but also execute parts of system logic. Development of Ethereum was annunced at the end of 2013 and has a planned first release in late 2014.

3.2.3 Keybase

Keybase [19] is another recent initiative that intends to solve the distribution of public keys. It is essentially an HTTP-interface that maps keys to identities. While keys themselves are stored centrally at Keybase's servers, it utilizes social media for proofs. The idea is that a Keybase user will put proofs of their Keybase identity on public social media services such as Twitter or Github. The Keybase client will refer to these to make sure that the given key corresponds to the user of these social media accounts. It ties identities to keys as long as a user's Keybase and social media accounts are not all compromised.

3.3 Communication

In the beginning of the World Wide Web's history, web browsers performed full page loads in order to render web pages. Further down the road, techniques such as AJAX (Asynchronous JavaScript and XML) enabled data to be fetched asynchronously from servers through the XMLHttpRequest API, thereby unlocking the creation of dynamic web applications. Since then, we have witnessed the introduction of the WebSocket protocol which enables persistent two-way communication between client and server. These techniques are centered around communication between a client and a server, but with the recent initiative of WebRTC (Web Real-Time Communication) which enables peer-to-peer communication there are some new possibilities in the field. In order to understand how these techniques work, some inherent problems with the infrastructure of the internet will be discussed in the NAT Traversal section.

3.3.1 XMLHttpRequest

XMLHttpRequest (XHR) is a browser API which enables data to be fetched asynchronously from servers. This means that a web page can retrieve new updates from the server without a full page reload. The browser is responsible for the construction of HTTP requests according to parameters passed to the API. In the use case where real-time updates from the server are desired, a common technique is to poll the server at regular intervals as the possibilities for streaming are limited. Although the name of the API suggests that data transfers are limited to XML, this is not the case and the name is nothing more than a remnant of the past [20]. Today it is much more common to transport data in the form of JavaScript-serialized objects, also known as *JSON* (JavaScript Object Notation).

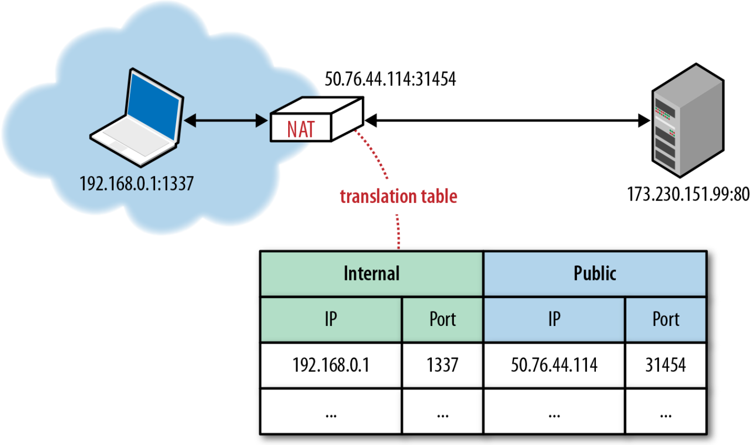
3.3.2 WebSocket

WebSocket (RFC 6455) [21] is a protocol which was standardized in 2011. The protocol allows two way communication between a server and a client through persistent connections. The protocol runs on top of the TCP protocol and is independent from the HTTP protocol. The implications of this is that the server does not need to open a new TCP connection for every incoming message as in XHR, and the high overhead from HTTP is eliminated which eases server workload.

3.3.3 NAT Traversal

Network Address Translation (NAT) [22] was first introduced as as a short-term solution to the problem of IP address depletion in IPv4. The idea was that by utilizing NATs, several hosts in a private network could share a single public IP address.

A NAT is responsible for maintaining a table of entries that map an internal IP address and port to a public IP address and port and dropping these entries when they are no longer of relevant. 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(See figure 3.1) [23].



Internal		Public	
IP	Port	IP	Port
192.168.0.1	1337	50.76.44.114	31454
...	...	...	...

Figure 3.1: *A NAT maintains a table mapping internal IP and port, to a public IP and port. [24].*

The presence of NATs can pose a problem to applications leveraging the UDP protocol for transportation of data and network communication in general. The different techniques that can be used to resolve problems caused by NATs are often referred to as *NAT Traversal* techniques.

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 it 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 since they cannot reach the source. In the case of TCP, which constitutes a well defined state, it is inherently simple to determine when an entry should be dropped.

A technique commonly used to solve the problem with UDP entries expiring and being dropped is to utilize keepalives at regular intervals. This is is commonly referred to as *UDP hole punching* [25].

Another problem is that internal hosts know their internal IP address but not their public one. If a host runs an application that communicates the IP address as a part of the payload to hosts oustide of the private network, there would obviously be a missmatch if the internal address is chosen. Therefore it is common to utilize a protocol called STUN (Session Traversal Utilities for NAT) [26]. STUN enables hosts to obtain their public IP addresses with the help of an external STUN server (See figure 3.2).

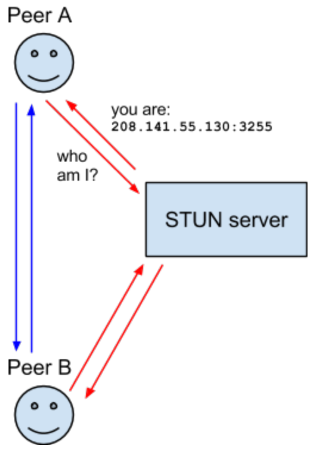


Figure 3.2: *STUN servers let peers in a private network behind firewalls discover their public IP-addresses [27].*

These techniques are not always adequate because of the fact that STUN does not work with all types of NATs. In some cases, UDP traffic might be blocked by a firewall. This is where the TURN (Traversal Using Relays around NAT) protocol [28] can be leveraged. TURN establish a TCP connection with a relay server when UDP has failed. 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 3.3).

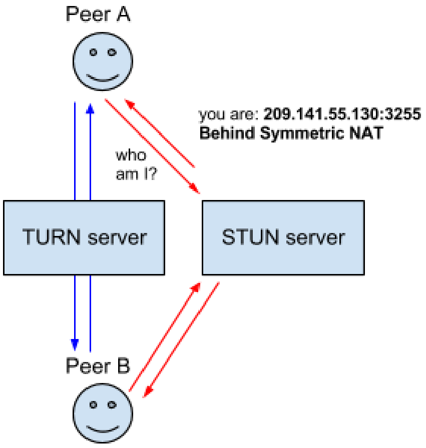


Figure 3.3: *If a peer-to-peer connection cannot be established, a relay through a TURN server could be used. All peers send their packets through the relay which makes it more costly. But at least the connection works [27].*

3.3.4 WebRTC

Up until recently, browsers have lacked support for direct peer-to-peer communication. Then in 2011 Google released a project called WebRTC, with the purpose of enabling real-time voice and video in the browser [29]. Since then, WebRTC has evolved to enable more general real-time data communication between browsers [30]. It has become usable for arbitrary data streams in major browsers in 2014 [31] [32].

Since the announcement of WebRTC, the organizations W3C and IETF have worked together on standardizing protocols and drafting APIs. The major browser vendors Google, Mozilla and Opera support the project [33]. While Microsoft supports the concept of WebRTC and contributes to the W3C WebRTC working group, it does not support Google's (or nowadays, W3C's and IETF's) version of it [33]. Microsoft doesn't want to support the new technology until it has become a standard and does not fully agree on some constraints placed on it [33]. Microsoft states that one of their issues with the current WebRTC version is that it has predetermined paths for choosing codecs and ways of sending media over the network – sort of a black box. This hinders application developers who want 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 attempts to address these issues.

WebRTC's functionality is abstracted into three different APIs: *MediaStream*, *RTCPeerConnection* and *RTCDataChannel* [34]. 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, it utilizes techniques such as *jitter buffering* and *echo cancellation* to ensure a high standard even in unstable networks. For the intent of file sharing, the RTCDataChannel API is the most relevant.

At the transport layer WebRTC makes use of UDP which can be motivated by the fact that timeliness is vital for real-time communication [35]. The UDP protocol alone is not enough to construct efficient peer-to-peer applications. For this very reason, WebRTC leverages a number of additional protocols built on top of UDP (See figure 3.4).

WebRTC's usage of UDP makes peer-to-peer communication inclined to suffer from connectivity problems in the presence of NATs (see section 3.3.4). This problem has been taken into account and is relieved by utilizing the ICE protocol (see section 3.3.4).

For the sake of security, data is encrypted according to 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 reliable transport protocols such as TCP.

RTCPeerConnection

The RTCPeerConnection API handles the creation of peer-to-peer connections and takes care of connectivity problems caused by NATs (see section 3.3.3) by utilizing the Interactive Connection Establishment (ICE) protocol [23]. The ICE protocol handles *NAT Traversal* and is used for establishing peer-to-peer connections. It makes use of STUN and falls back to TURN when no other alternatives exists (See figure 3.5).

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 depends on two things:

- The existence of a signaling channel - where a connection should be negotiated
- The choice of signaling protocol - which protocol that should be used for the negotiation

[1] http://creativecommons.org/licenses/by/3.0/legalcode

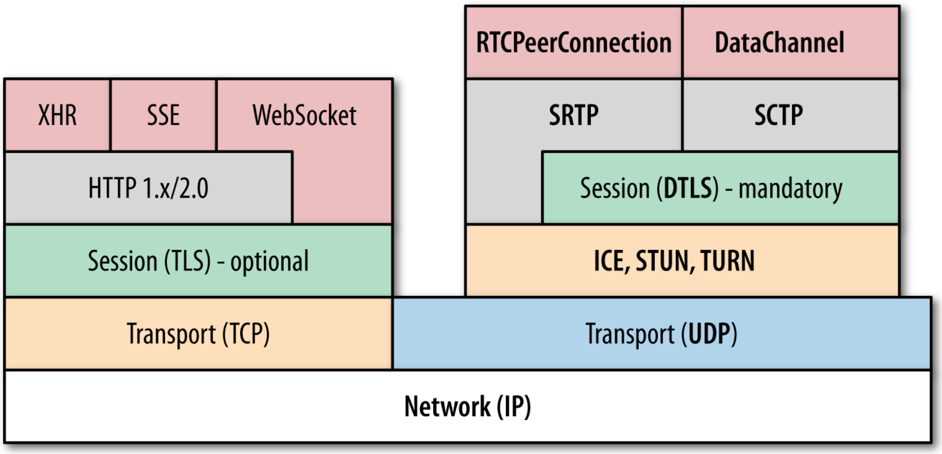


Figure 3.4: *WebRTC's underlying protocols [36]*

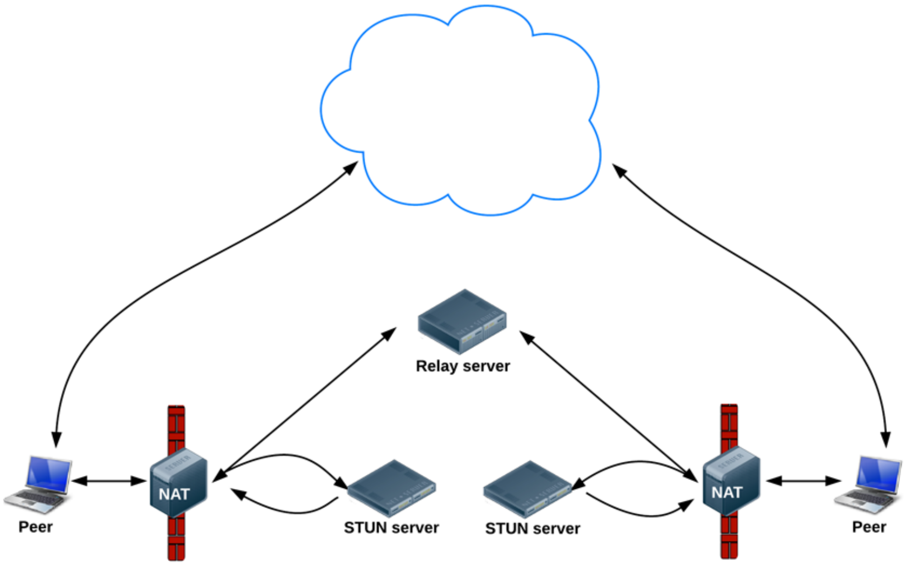


Figure 3.5: *The different ways for ICE to find network interfaces and ports, titled* Finding connection candidates *[34], available under a Creative Commons Attribution 3.0 Unported License*[1]

Regarding the choice of signaling channel, a dedicated signaling server is often used. That is, a server which relays connection offers from one peer to another. Although this is the most common choice of signaling channel, examples of a more serverless approach can be found [37]. For the choice of signaling protocol the standard does not provide any recommendations - this is for developers to decide.

RTCDataChannel

The RTCDataChannel API allows for arbitrary data to be sent peer-to-peer by leveraging the RTCPeerConnection API. As UDP itself, which WebRTC runs on top of, is not suitable for reliable data transportation the RTCDataChannel API utilize the SCTP (Stream Control Transport Protocol) protocol. SCTP can be configured in terms of reliability and message order, and it provides some additional services such as congestion and flow control [35].

3.4 Cryptography

Encryption is the process of running data through an algorithm with the goal of making the information unreadable for unauthorized parties. The output of an algorithm depends on the input data and the *encryption key* used. There are two types of key schemes: symmetric-key schemes and asymmetric-key schemes (also known as public-private). In symmetric cryptography schemes such as AES, the same key is used for both encryption and decryption. Because of this, both sender and recipient must possess the same key, and the key must therefore somehow be communicated through a secure channel. Asymmetric key-schemes such as RSA, on the other hand, work with pairs of keys where the encryption of one corresponds to the decryption of the other. One of these keys is called *private* and should only be known to the person generating the key-pair, and the other is called *public* and can be shared freely. In this way, there is no problem with how to transmit keys. Generally, encryption and decryption in asymmetric schemes are much more computationally demanding than that of symmetric schemes, so a common approach is to use symmetric encryption for data and asymmetric keys for the communication of the symmetric key.

Until recently, the practice of performing cryptographic operations in a web browser environment has been considered bad practice by security professionals [38]. 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[2], as well as raw private and secret key data. With the advent of the new Web Cryptography API, or *WebCrypto*, these issues are being addressed [10].

3.4.1 Web Cryptography API

WebCrypto is an open standard for implementation of cryptographic primitives accessible through web client code [10]. Basically, all primitives and raw material (the underlying keys and algorithms) would be blackboxed for the client application and executed natively in the web browser. However, the API is still in an early stage and at the time of this writing only Chromium [39] and Microsoft Internet Explorer [40], out of the major browsers have implemented more than a basic pseudo-random number generator [41]. The exact implementation of the different features may come to change drastically over time.

[2] The basic functions in a cryptographic scheme such as hashing, encryption, decryption, signing, verification and generation of keys

3.4.2 Certificates

Certificates are used to confirm the validity of users' keys [42]. Instead of requesting keys directly, which could have potentially been compromised by malicious third parties, certificates are retrieved from trusted CAs. In essence, a certificate contains a user's public key along with user data and information about the certificate. The CA signs the certificate and a hash is embedded as proof that the certificate has not been unlawfully modified.

These certificates are structured according to a certain format. One commonly used format for certificates is the X.509 standard [43]. In the X.509 system, a CA issues a certificate binding a public key to a name such as an e-mail address or a DNS entry.

3.4.3 Advanced Encryption Standard

The Advanced Encryption Standard [44], *AES*, is one of the most widespread symmetric encryption schemes in use today. It is a specification established by the U.S. National Institute of Standards and Technogy in 2001 and based on the Rijndael cipher [45], which in turn is based on the idea of substitution-permutation [46]. The algorithm distorts the inputed value by means of replacement and uses a structure with a fixed block size. AES was intended as a replacement for DES [47] [48]. Encryption is performed in rounds with the number of rounds depending on the length of the key.

For reasons of performance and security, symmetric algorithms such as AES encrypt data in blocks of fixed size. There are different ways to make these blocks relate to each other, depending on the type of data and application. One of the more common modes is Cipher Block Chaining, or CBC, where the encryption function for each subsequent block is fed the encrypted version of the previous block. This is done to make blocks with the same input plaintext indistinguishable [49].

4 Related work

There is a plethora of technologies for distributing and synchronizing data between peers that at a 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 [50] is a distributed peer-to-peer multi-way file syncing software using the Bittorrent protocol for file transfers. Synchronized 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 the only available clients are closed-source binary applications using the network of the creator, Bittorrent Inc. While they do have a developer API, it requires developer keys issued from Bittorrent Inc.

RetroShare [51] markets itself as a Friend-2-Friend decentralized communication platform which uses GPG to create a Web of Trust between peers. The Friend-2-Friend search works by propagating search and file transfers recursively to all friends of friends to a certain degree of separation. In this way, users can search for files from a huge network while still only staying directly connected to people that they trust. It is, however, a very large project: The application provides file-sharing, instant messaging, discussion forums, e-mail, Voice over IP (VoIP) and group chat. It is open source and distributed as cross-platform binaries.

ShareFest [52] 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 of what Rymd aims to be: ShareFest can share files over WebRTC channels, but does not accommodate 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 [53] 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 them. Freenet is free software built with Java.

Tahoe-LAFS [54], 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 similar 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 hosting for them.

Bitmessage [55] 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.

Rymd differentiates by being the only project so far that combines the properties of open source, purely web based, decentralized and cryptographically secured. Additionally, it is a developer-geared library rather than a user-oriented application.

5 | Analysis and System design

This chapter gives a closer analysis of the technologies brought up in chapter 3 in relation to the overall problems presented in section 1.3. First, we go through the client-side storage alternatives for web browsers and present the motivation behind the choice of IndexedDB as the default data store for Rymd. WebRTC as a means of peer-to-peer communication in Rymd is also briefly presented. This is followed by an explanation of the authentication scheme in Rymd and how the cryptocurrency Namecoin is employed in key distribution. A brief explanation of the implications these design choices have on the decentralization aspect of the system is laid out, followed by the resource data structures in Rymd and how they are communicated between peers in a file sharing scenario. This concludes in the high-level design and modules of Rymd and Shuttle.

5.1 Data storage

Storage of data is 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 (Web Storage)
- IndexedDB
- WebSQL
- FileSystem API

LocalStorage is included in the HTML5 Web Storage specification [11] 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. Further, LocalStorage does not support complex structures and indexing, and storing different data types is complicated and needs manual serialization and deserialization. Thus this solution was immediately rejected.

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 no longer maintained by W3C [12] 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 [13]. Arbitrary types of data can be stored in the database, such as strings, numbers, JavaScript objects, and raw binary data.

The last alternative, The FileSystem API, is a collection of methods for reading and writing to a sandboxed file system from a browser with client JavaScript code. It is a very early standard, being currently only supported by Google Chrome and Opera, and has the status of Working Draft by W3C [8]. 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 for Firefox [56]. In April 2014 it was announced on the Web Applications Working Group mailing list that the specification should be considered dead, since other browser vendors have had no interest in implementing it [57].

[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 5.1: Browser support for selected HTML5 APIs at the time of writing

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 browser 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 it is actively maintained (while WebSQL is not). Users of the Safari browser will not be able to utilize the product, but considering the project's overall direction with regards to experimental technologies, this is negligible.

5.2 Peer-to-peer communication

Even though there are two projects which seeks to enable peer-to-peer communication in the browser - WebRTC and CU-RTC-Web - the only viable alternative for the time being is WebRTC. The project is under development by the organizations W3C and IETF and is supported by the current versions of Chrome, Firefox and Opera. CU-RTC-Web on the other hand is not supported by either W3C or IETF and is not implemented in any browsers.

For the project's intent of file sharing, the RTCPeerConnection and RTCDataChannel API were found to be most relevant. By using these APIs in conjuction arbitrary data can be sent peer-to-peer. Relating back to the project goals regarding privacy and security it is also convenient that data transfers are encrypted according to the DTLS protocol.

5.3 Peer identity verification

One of the main issues to be resolved in a project of this nature is that of distribution of cryptographic keys. For a truly decentralized system, it is not acceptable to adapt a CA-centered approach, because of the high level of trust that is put in central authorities. While a Web of Trust is interesting, it might be too cumbersome for users. This issue is addressed in *Zooko's Triangle* (See figure 5.1), 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 [58]. The 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 distributed hash table (DHT) with unique keys. Users can reserve a name and assign to it a value of their choice at the cost of a small amount of the Namecoin currency (currently 0.01 NMC [59], which is roughly equivalent to 0.03 USD [60]).

Ethereum, which was announced just a couple of weeks before the start of this project, extends this by their scripting language which not only allows 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 similar to Ethereum could be very interesting to explore for a project such as Rymd, but its development is still in such an early stage that it is deemed too unstable to be useful at this point.

Rymd therefore utilizes Namecoin 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 blockchain-based approach makes it secure. Additionally, the small monetary fee required to register a name prevents massive name-squatting. However, there are practical limitations and consequences associated with this approach. The monetary cost associated with the insertion of a new value means that key insertion needs to be handled outside of Rymd. That updates can take a significant amount of time to propagate over the network (up to several hours) is another issue, but since insertion or updates of keys should happen very seldomly, this should be acceptable. Most importantly, since Namecoin (or any other currently existing cryptocurrency for that matter) communicates using their own binary data protocol [61], web applications can not interact directly with the blockchain to fetch this information before a mutual peer-to-peer authenticated connection is established. As a consequence, a service that acts as a bridge between Rymd and the blockchain is needed.

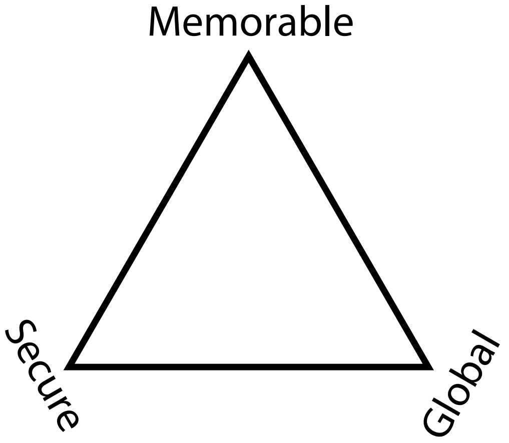


Figure 5.1: *Zooko's Triangle, with the edges representing the achievable combinations of features [62]*

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, attacks exist [63]), Wide Mouth Frog [64] (depends on timestamps), and Needham-Schroeder [65]. Of the ones examined, Needham-Schroeder stood out as simple to implement since it does not utilize symmetric keys or timestamps, while it provides mutual authentication. Consider the scenario where A wants to authenticate to B, assuming that they have already exchanged public keys K_{PA} and K_{PB}. Then the Needham-Schroeder protocol flows as shown below:

$A \rightarrow B : \{N_A, A\}_{K_{PB}}$ A generates a random nonce N_A, encrypts it together with their identity and sends it to B.

$B \rightarrow A : \{N_A, N_B\}_{K_{PA}}$ B responds by generating their own nonce N_B, encrypts it together with N_A and sends it back to A. By replying with N_A, they prove that they possess the private key corresponding to KP_B.

$A \rightarrow B : \{N_B\}_{K_{PB}}$ A replies with N_B. The proof works in the same manner as for B. A and B are now mutually authenticated.

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 [66]. Lowe also proposed a fix to this vulnerability and this amended *Needham-Schroeder-Lowe protocol* presented below is what Rymd utilizes for authentication.

$A \to B : \{N_A, A\}_{K_{PB}}$

$B \to A : \{N_A, N_B, B\}_{K_{PA}}$ B also includes their identity to make sure that this message can not be reused by other parties posing as someone else.

$A \to B : \{N_B\}_{K_{PB}}$

5.4 Decentralization

Since the system utilizes the Namecoin blockchain for storage of public keys, there is an issue of how to interface web applications with the blockchain without putting too much trust in the HTTP/cryptocurrency gateway. Users could host their own gateways or retrieve or verify keys manually through their own Namecoin clients. 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.

5.5 Creation and transfers of resources

First, we address the question of how to identify resources. That identifiers should be memorable, secure and unique holds not only for users, but also for resources. Since Namecoin is being used for storage of public keys of users, it is therefore natural to consider using a cryptocurrency for resources, too. However, the practical limitations on insertion and updates becomes a much bigger issue here, since resources are created and updated in a much higher frequency than users register or update their public keys. These practical issues would make such a system practically unusable. Therefore, the idea of using any current cryptocurrency blockchain for storing anything resource-related was discarded. 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. Therefore, resources are simply identified by a random GUID generated at the time of resource creation.

Full access to a resource implies possession of three things: the encrypted resource data, the cryptographic key used to encrypt said data and the metadata describing the resource. The creation of a new resource is done as follows:

Generation of metadata 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.

Generation of resource-specific symmetric cryptographic key In the default implementation, a 256 bit AES-CBC key is used.

Encryption of the resource data using the resource key

Calculation of resource hash based on the metadata and encrypted data The hash is then added to the metadata.

Creation of resource Metadata and encrypted data are combined into the internal representation of the resource.

Saving of key to 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.

The exact flow involved in a transfer depends on the implementing application. Generally (and in Shuttle), 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. An interesting possibility is that of *Friend-2-Friend search* in the Retroshare network (described under chapter 4), which makes it possible for users to search for files in a huge network while still only being directly connected to their trusted friends. The separation of storage, encryption and transfers of metadata and file data in Rymd allows Rymd-based applications to realize similar functionality.

Once again, verification of integrity and authenticity of resources are achieved by verifying the hash. It is vital that a sufficiently secure 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 [67]. 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 the use of SHA-1 for applications that require collision resistance [68]. 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. In this way untrusted peers could help facilitate transfers of resources in a distributed fashion. In the example implementation Shuttle, file sharing is initiated from the sharing end. Consider the case where Bob wishes to share a resource with Alice (assuming both Alice and Bob are already connected to the network and know each others' identity names in the DHT):

1. Bob requests Alice's public key and endpoint IDs from the DHT.

2. Bob initiates a connection with Alice and they are mutually authenticated. This process is described in 5.3.

3. Bob sends the metadata and key for the resource to Alice.

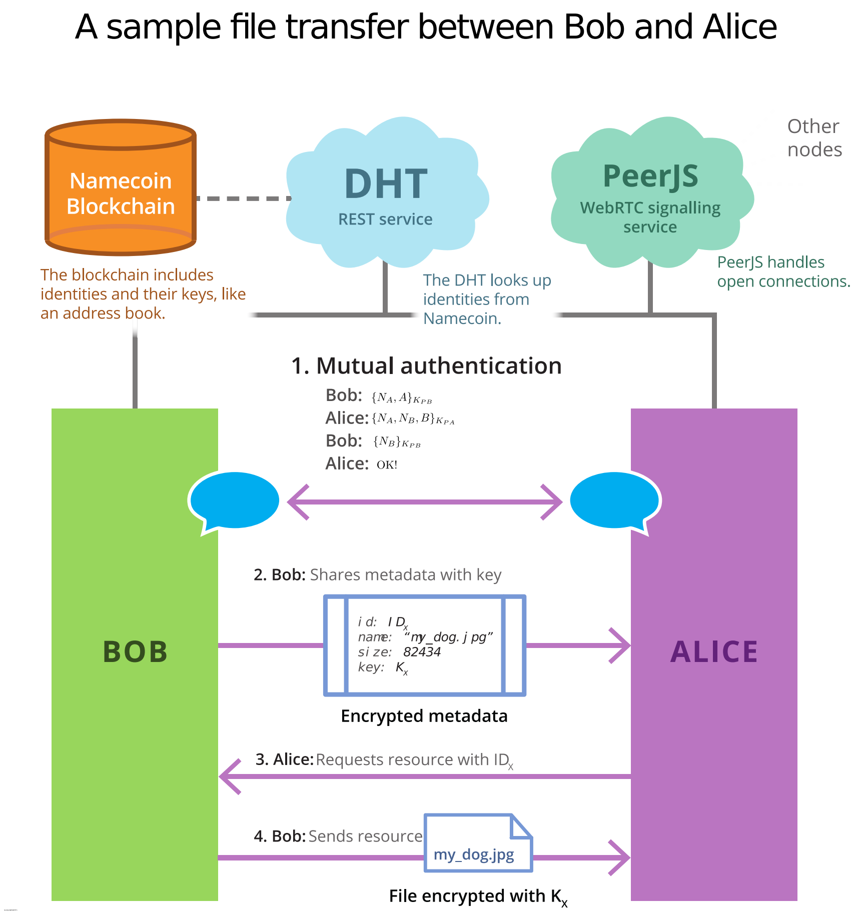
4. Alice creates a new resource as described above, but without the encrypted data, and saves it to her data store.

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.



A sample file transfer between Bob and Alice
Namecoin Blockchain
DHT
REST service
PeerJS
WebRTC signalling service
Other nodes
The blockchain includes identities and their keys, like an address book.
The DHT looks up identities from Namecoin.
PeerJS handles open connections.
1. Mutual authentication
Bob: $\{N_A, A\}_{K_{PB}}$
Alice: $\{N_A, N_B, B\}_{K_{PA}}$
Bob: $\{N_B\}_{K_{PB}}$
Alice: OK!
2. Bob: Shares metadata with key
id: ID_x
name: "my_dog.jpg"
size: 82434
key: K_x
Encrypted metadata
BOB
ALICE
3. Alice: Requests resource with ID_x
4. Bob: Sends resource
my_dog.jpg
File encrypted with K_x

Figure 5.2: *The flow involved in Bob sharing a resource with Alice*

5.6 Modules in Rymd

The core Rymd library contains no implementation meant to deal with a problem area but rather acts as the central hub with workflows that binds all parts together. The library is divided into modules that separates complex areas of the application – areas that are most likely to change independently over time – into well defined and easily exchangeble *modules*. Rymd is designed to have the modules supplied dynamically through dependency injection. With key features being implemented with technologies from everchanging fields at the web's forefront, this modular design was imperative in order to make the system maintainable (as explained in section 2.3.1).

The items shown below represent the general high-level responsibilities behind each module. Please see chapter 6 for specifications of the technologies used and accounts of the modules' actual implementations.

Cryptography Handles encryption, decryption, signing and verification as well as key generation and hashing.

Peer-to-peer communication Responsible for setting up initial contact with another peer and maintaining the connection. It also deals with securing this connection through end-to-end encryption.

DHT interaction Used to interface with a Distributed Hash Table in order to retrieve peers' public key and ID.

Data storage Deals with persisting file data and metadata (as described in section 5.5) on disk, and this module also provides means of accessing and manipulating the data.

Key storage Solves storing and providing access to cryptographic keys.

In reality, no distinction was made between the data storage module and the key storage module. Furthermore, it would have been appropriate to split the data storage module into two parts, one for the actual file data and one for the metadata. As it stands, metadata could not be stored if a storage solution that stores data directly on disk would be injected.

6 | System implementation

With the background of the high-level foundations presented in chapter5, this chapter details the underlying implementation of the modules of the whole system. All source code has been written in JavaScript. `DHT` is run as a Node.js server and `Shuttle` runs as a client-side web application. Rymd is the platform independent core module containing the business logic of the system. All other modules contain the specific implementations for each problem area (see figure 6.1). Links to the source code for all modules are available in chapter A.

In section 5.6 the overall areas of responsibility for the system modules were defined. This resulted in the following concrete modules described in this chapter:

Rymd The business logic for the system. Has references to the other modules through dependency injection.

Shuttle The front-end prototype – a client-side web application.

DHT Client Module which looks up records in the Namecoin blockchain through a NodeJS web service (see A).

RymdCrypto Module for cryptography.

IndexedDBStore Module for data and key storage.

PeerJS Connection Module for communication with the PeerJS service.

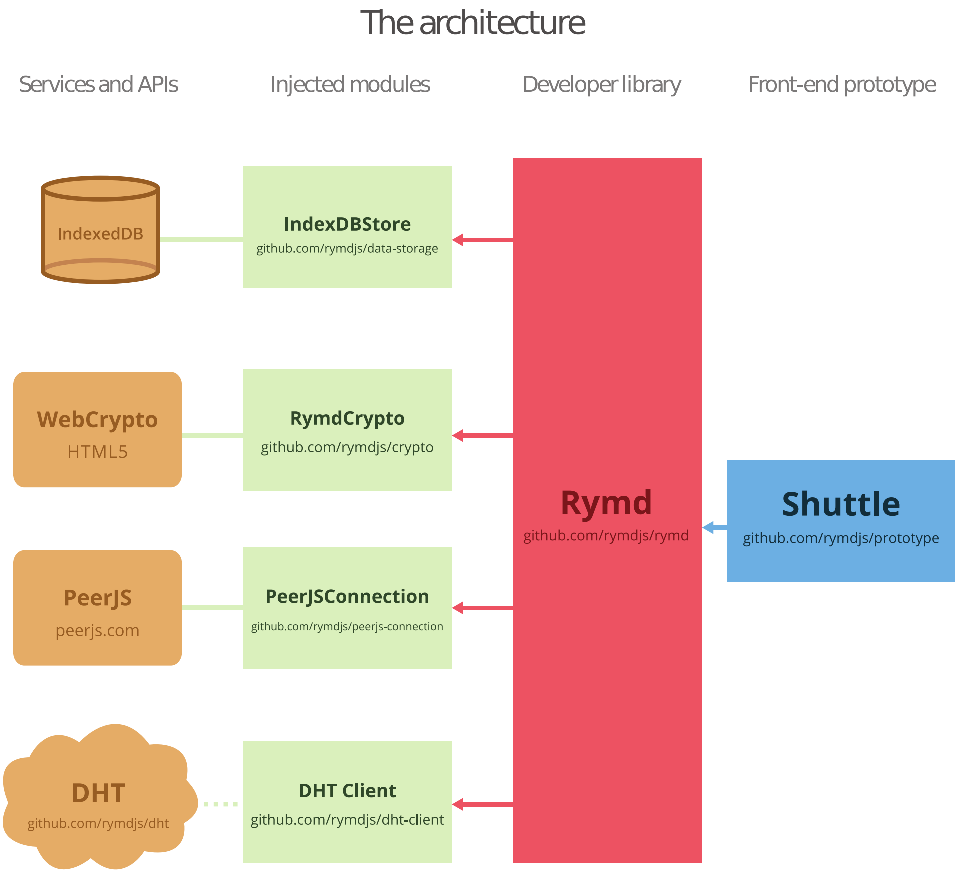


Figure 6.1: *The modules and external APIs utilized in Rymd and Shuttle*

6.1 Rymd

The Rymd library is the only truly implementation-agnostic module and should be runnable on any platform as long as implementation modules are dependency injected. It defines the business logic for behaviour such as:

- The structure of resources and metadata
- The communication and authentication flow
- The *resource store*: How resources are saved and identified independent of implementation
- Management of chryptographic keys
- Session management

Applications utilizing Rymd will generally instantiate a `RymdNode` object, which is the main entry point of the library and inject implementation modules. The RymdNode object will then act as the interface between the application and the Rymd library. The public interface of `RymdNode` is listed in figure 6.1 along with the method signatures and return values.

```
// Method signatures on the form <name>: [(<parameter>:<type> ...)]:<Return type>

// Initialize network and connection handlers from an identity
init: (identity:String):Promise

// If the RymdNode is initialized with an identity or not.
isAlive:Boolean

// Set the private key for an identity
setPrivateKey: (key:Key, identity:String):Promise(guid)

// Get the private key for an identity
getPrivateKey: (identity:String):Promise(key)

// Get the public key for an identity
getPublicKey: (identity:String):Promise(key)

// Get the RymdNode's current identity name
currentIdentity:String

// Connect to another identity (RymdNode)
connect: (identity:String):Promise(connection)

// Share a resource with a guid with another identity
shareResource: (guid:String, identity:String):Promise

// Request (download) a shared resource with a guid
requestResource: (guid:String):Promise

// Destroy a resource with a guid
destroyResource: (guid:String):Promise
```

Listing 6.1: Public methods of `RymdNode`

RymdNode triggers certain events on itself, which can be used for handling incoming resource sharing proposals, resource download requests, and more. The complete list of events are listed in figure 6.2. These events might initially be triggered inside some injected module, but RymdNode listens to events and triggers them on itself in order to present a coherent external event interface. For instance, the request event is triggered inside the PeerJS Connection module (section 6.6) and bubbled up to RymdNode.

```
App.rymdNode.on('init', function(identity) {
  // This RymdNode is finished initializing with a given 'identity'
});

App.rymdNode.on('resource', function(peerName, resource) {
  // Incoming resource from 'peerName'
});

App.rymdNode.on('request', function(peerName, data, connection) {
  // Incoming request for a resource (whose metadata are in 'data') from 'peerName'
});

App.rymdNode.on('share', function(peerName, data, connection) {
  // Incoming share request from 'peerName' with resource metadata in 'data'
});
```

Listing 6.2: Events triggered on RymdNode

6.2 Shuttle

The front-end prototype was built in order to test and implement the features of Rymd. Early versions included a minimum viable interface for adding, showing and sending files. This was further iterated over, ending up letting the user:

- Add files through form control or drag-and-drop
- Share, view and delete files
- Login (verifies with the DHT, see 6.3)
- Get in-app notifications for incoming sharing requests
- Download remote files that have been shared by other users
- Add custom encryption keys

Shuttle uses Rymd's functionality by instantiating a global RymdNode object (see 6.1). This effectively makes Shuttle a node in the network. In-app notifications are shown by listening to certain events on the rymdNode object. Shuttle uses RymdNode's external interface in order to control the application logic, such as responding to and initiating resource sharing requests (see figures 6.3 and ??).

```
// Listen to incoming request events and immediately send the wanted resource
App.rymdNode.on('request', function(peer, data, connection) {
  // Access the rymdNode object's resource store
  App.rymdNode.store.getResource(data.guid, true)
    .then(connection.sendResource.bind(connection));
});
```

Listing 6.3: Incoming download request

```
var guid = <guid fetched from the application interface>;

App.rymdNode.store.getDecryptedResource(guid).then(function(resource) {
  // Create URL to file with correct MIME type and show in new window
  var objectUrl = Rymd.Utils.toObjectURL(resource.data.data, resource.metadata.type);
  window.open(objectUrl, "_blank");
});
```

Listing 6.4: Show a downloaded file

6.3 Authentication (DHT, DHT Client)

Since the default authentication implementation utilizes Namecoin, which can not be accessed directly from a web application, a gateway service needs to be used. Therefore, the domain of authentication spans over several parts in separate systems:

DHT A NodeJS[1] based server that looks up entries in the Namecoin blockchain. It is also used to keep track of session-based IDs, as described under 3.3. This is the only module that runs outside of the Rymd library.

DHT-Client Client-side interface module to the DHT.

ConnectionHandler Submodule in the Rymd library. Implements the Needham-Schroeder-Lowe authentication business logic.

The idea with this separation is that the authentication algorithm is part of the core library, while derivative projects should be able to replace DHT and $DHT-client$ with implementations using other stores such as Ethereum or Keybase without having to consider writing a secure authentication protocol, should they so desire.

6.4 RymdCrypto

The implementations of WebCrypto in Chromium supply key generation, but there is no support for persisting keys between sessions or even exporting private keys. W3C - the organization behind WebCrypto - have announced their intention to handle persistent key storage in an upcoming API called WebCrypto Key Discovery [69]. However, Google has no intention of implementing this in Chromium before the WebCrypto API is finalized. WebCrypto Key Discovery API was originally intended to be a part of the WebCrypto API but was extracted in order to decrease implementation complexity.

[1] http://nodejs.org

6.4.1 Algorithms

Since Rymd uses asymmetric keys for authentication, RymdCrypto should supply asymmetric signing and encryption schemes using the same keys. Considering the support in WebCrypto, this leaves RSA based schemes as the only alternatives (See table 6.1). There are only two RSA based encryption schemes that are part of WebCrypto: $RSAES-PKCS1-v1.5$ and $RSA-OAEP$. $RSAES-PKCS1-v1.5$ is currently the only one that is implemented in Google Chrome, and is therefore the current choice for Rymd. Rymd uses $RSASSA - PKCS1 - v1.5$ for signing, mainly because it is recommended by the working group behind the WebCrypto API.

For encryption of resources, a symmetric algorithm is used. AES-CBC is a fast and simple to use symmetric key algorithm based on AES using cipher block chaining [46].

Finally, a good hashing algorithm is needed in order to avoid collision of hashes, in other words to avoid having two generated hash values being exactly the same (similar to the *birthday problem* [2]). For hashing purposes RymdCrypto uses SHA-256 as explained in section 5.5.

Algorithm name	Type	Encrypt	Decrypt	Sign	Verify	ImportKey
RSAES-PKCS1-v1_5	ASYM	x	x			x
RSASSA-PKCS1-v1_5	ASYM			x	x	x
RSA-PSS	ASYM			x	x	x
RSA-OAEP	ASYM	x	x			x
ECDSA	ASYM			x	x	x
AES-CTR	SYM	x	x			x
AES-CBC	SYM	x	x			x
AES-CMAC	SYM			x	x	x
AES-GCM	SYM	x	x			x
AES-CFB	SYM	x	x			x

Table 6.1: Some WebCrypto API algorithms

Commercial RSA certificates are more widely deployed compared to DSA certificates. Furthermore, the asymmetric keys are wrapped in PKCS#8 and SPKI certificates while the symmetric key exist in raw format. *The Private-Key Information Syntax Standard* (PKCS#8) 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 [39]. See listing 6.5 for a complete rundown of Rymd's algorithms.

[2] http://statistics.about.com/od/ProbHelpandTutorials/a/What-Is-The-Birthday-Problem.htm

```
 var generationAlgorithmSign = {
   name: "RSASSA-PKCS1-v1_5",
   modulusLength: 2048,
   publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
   hash: {
     name: "SHA-256"
   }
 },

 generationAlgorithmEncrypt = {
   name: "RSAES-PKCS1-v1_5",
   modulusLength: 2048,
   publicExponent: new Uint8Array([0x01, 0x00, 0x01])
 },

 aesAlgorithmKeyGen = {
   name: "AES-CBC",
   length: 256
 },

specFormats ={
   public: ’spki’,
   private: ’pkcs8’,
   secret: ’raw’
 }
```

Listing 6.5: Algorithms implemented

6.4.2 Libraries

Until the WebCrypto Key Discovery API is available, the RymdCrypto module generates pseudo-random keys through the external library *bignumber-jt*[3]. To do low-level key generation directly in JavaScript is bad practice and is performed in order to make Rymd runnable until a more solid solution is available.

Furthermore, IndexedDB is used as makeshift storage since the WebCrypto API lacks functionality for storing keys between sessions, or even exporting private keys. In order to use IndexedDB for the keys they need to be parsed to certificates. This is also handled by bignumber-jt. All certificates have a static key size where asymmetric keys are of $1024bits$ and symmetric keys are of $256bits$; none are explicitly generated via the API.

Hashing is handled by the external library *crypto-js*[4].

6.4.3 Interface

In short, the RymdCrypto module exposes functions handling:

Generation of asymmetric keys Generates RSA key pair

Generation of symmetric keys Generates AES key.

Exporting and importing keys

Encryption and decryption For AES and RSA

Signing and verification For RSA

[3] https://www.npmjs.org/package/bignumber-jt

[4] https://github.com/evanvosberg/crypto-js

Hashing Using SHA-256

6.5 IndexedDBStore

The main task for the data storage module was to abstract away the low-level methods in IndexedDB (the backing store used, see section 3.1.4). An API example can be found in listing 6.6. The module supports use of multiple object stores and auto-generation of GUID keys.

```
var IndexedDbStore = require("indexeddbstore")

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 6.6: Common database operations

The largest challenge came to the edge cases when storing files, or as they are called in web browser: *Blobs*. Since at present only Firefox can 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 tightly coupled and was removed. The final implementation is storing data as-is – any metadata must be saved explicitly in a separate operation.

The asynchronous API of IndexedDB is relatively verbose and complex. It makes heavy use of event driven programming and thus the developer communicates with the database with callbacks. By the use of *Promises* [70], the asynchronous, callback-based methods in the IndexedDB API was made streamlined and simple to manage.

6.6 PeerJS Connection

Rymd leverages the open source project PeerJS[5], which simplifies sending peer-to-peer data between clients. This module was therefore constructed, in line with the project guidelines regarding modularity, to contain PeerJS-specific code and to

[5] http://peerjs.com

provide an independent interface which does not reveal the details of PeerJS's inner workings. If changes in Rymd's requirements makes the choice of PeerJS obsolete, then the changes will be isolated and one should still be able to depend on the same interface.

PeerJS utilizes 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, that is the signaling protocol, PeerJS utilizes both WebSockets and XMLHttpRequest [71]. After a connection has been setup between two clients, the server is no longer needed in order for them to communicate.

The following steps explain how PeerJS brokers connections:

1. Two clients connect 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 connects 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 order to map the identity registered in the Namecoin blockchain to the ID supplied by the PeerJS server, the DHT service is used. After a client has connected to a PeerJS server, it supplies the data regarding the server IP and the generated ID to the DHT. When another client wants to create a peer-to-peer connection, it queries the DHT service for the Namecoin identity, which then returns the IP for the PeerJS server and the ID for the client.

To ensure the integrity of users it is vital that the communication with the PeerJS server and the DHT service is encrypted. The current implementation utilize the WebSocket protocol and the XHR API, which are not encrypted per default, for communication over these channels. Therefore an implementation running on top of the SSL/TLS protocol has to be in place before the project can be considered to have reached a releasable state.

6.7 Testing

Automatic unit tests have been implemented where possible. Thanks to the use of isolated modules test suites were able to be short and concise. Mocha was used as test runner, with Chai as assertion library (please see section B for links to external libraries used).

```
describe("IndexedDBStore", function() {

  var db;

  beforeEach(function() {
              db = new IndexedDBStore({
                     dbName: "test"
              });
       });

  it("should have a database name", function() {
              db.name.should.equal("test");
       });

  it("should save a record and return an id", function() {
              return db.save({foo: "bar"})
              .then(function(id) {
                     id.should.be.a("String");
              });
       });

  it("should retrieve a given record", function() {
              return db.save({foo: "bar"})
      .then(db.get.bind(db))
      .then(function(record) {

                     record.should.be.an("Object");
                     record.data.foo.should.equal("bar");
              });
       });
});
```

Listing 6.7: Sample test suite

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.

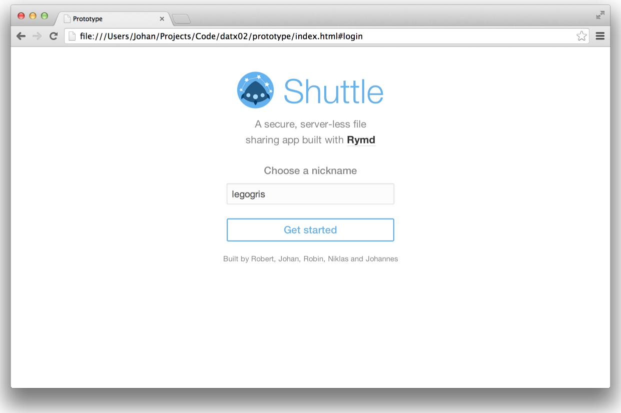
7 | Results and Discussion

Here, we lay out the outcome of the project and some of the difficulties that were met in the process. We then 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.

7.1 Outcome

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 5.1), cryptography (section 6.4) and communication (section 6.6) 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 can add, list, view and delete 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 be downloaded to their local data store.


Prototype
file:///Users/Johan/Projects/Code/datx02/prototype/index.html#login
Shuttle
Get started

A secure, server-less file sharing app built with **Rymd**

Choose a nickname

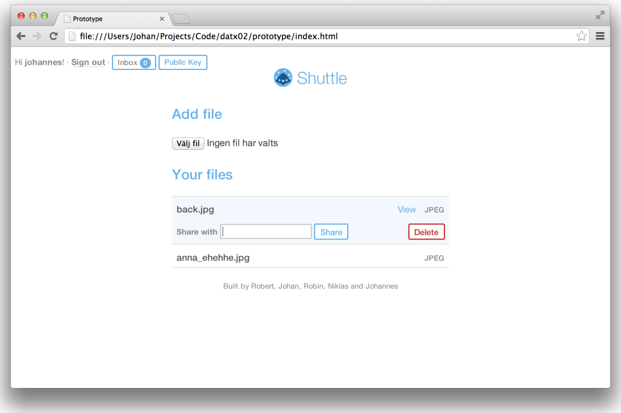
legogris

Built by Robert, Johan, Robin, Niklas and Johannes

Figure 7.1: *The Shuttle login view*

7.2 A decentralized system

One of the main initial goals for 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 are, however, two parts where communication with central endpoints still needs



Built by Robert, Johan, Robin, Niklas and Johannes

Figure 7.2: *Files listing in Shuttle*

to be done: connecting peers through WebRTC ICE and the interaction with the Namecoin blockchain.

7.2.1 WebRTC ICE

As described in section 6.6, establishing 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 for 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 considered that the latter was 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.

7.2.2 Accessing the DHT

The Namecoin blockchain is used to tie identities to their public keys and PeerJS endpoints. 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. In theory, each user could even run their own gateway. On May 6th, at the time of writing of this report, a public and more general HTTP/Blockchain interface at *chain-api.com* [72] 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 such as 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 such as 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.

7.3 Cryptographically secured data

In the application layer, all communication over WebRTC is DTLS encrypted. As stated in section 5.5, 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. When the WebCrypto Key Discovery API becomes finalized and implemented across browsers, separate key storage options will become available and can be implemented in Rymd.

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 - anonymous file sharing where peers are only connected to peers they trust, but data is relayed through chains of connections to facilitate propagation of data. In these cases it is imperative that resources can be transmitted separately from their keys and metadata so that intermediate peers can relay 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.

7.4 A modular and implementation agnostic system

As much of Rymd uses and relies on 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 alternatives. 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 ownprojects 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 altogether even if the current implementation puts all three side by side in IndexedDB.

Furthermore, a system with separate interchangeable parts allowed for both advantages and disadvantages. 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 the debugging of events that flow through multiple modules. Data would travel through connecting endpoints between modules and then be sent far down call hierarchies (perhaps triggering new events or having more information appended to it) before bubbling back up in the call chain. Any error occuring in such a flow were hard to pinpoint.

7.5 Trusting a distributed web based application

Currently, major web browsers have no way to verify client-side code for web applications the way that native binaries or Java applets can be cryptographically verified using signatures. Since system logic is run in a web browser, users can not know for sure whether the client code is altered between executions or not. 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 [38]. If the application is delivered in form of a web browser extension, it can be signed using a certificate. This means, however, that the application has to be specifically bundled for each web browser which limits the practical portability of the code. These extensions are currently mainly available for desktop browsers. In the case of Rymd, it all depends on how the developers of implementing applications uses it.

7.6 Ethical aspects

Rymd was originally conceived from an ethical issue: 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.

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 applies 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 a contributor in the movement of free communication on the web.

7.6.1 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 those mentioned in section 4 that are currenly used for these purposes - Rymd does not enable risks that are not already present.

Inevitably, the question of whom is to hold responsible for malicious activity is raised when a project of this nature is realized. Some argue that developers can be held responsible since they are enabling this behaviour with less risk of repercussions. Others argue that as long as surveillance and data mining are increasingly abused by authorities and private organizations over the world, the access to private and secret communication is becoming ever more important. Technology exploring new frontiers is always met with a critical gaze; laws and regulations have merely briefly halted advancement. One way or other, technological evolution has always prevailed once the genie is out of the bottle.

Most importantly, however, it is our opinion that 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

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

issue as a technical one and will let the readers of this report decide for themselves where they stand and how Rymd relates to this.

8 Conclusion

The scope of functionality that is included in open web standards is constantly widening, and with this so do the potentials and capabilities of modern web browsers. Some standards are still in their infancy. Most notably, the Web Crypto API is still in a experimental state and lack vital parts such as secure key storage. More progress is needed before secure authenticated peer-to-peer file sharing can be done by web browsers.

Even so, Rymd and Shuttle show that this future might not be that far away. Even if the resulting code is not yet ready for production from a security perspective, we feel that our ambition to create a starting point for a web-centered secure file sharing platform has gone well. It is our hope that Rymd will get feedback from the community in order to inspire and play a part in a new wave of file sharing technologies on the web.

The web is evolving in a incredibly fast rate. This project demonstrates a fraction of what will soon be possible. We are watching with eager eyes what the future may hold.

"This is not the end. It's not even the beginning of the end. But it might be the end of the beginning." - Winston Churchill

References

[1] OFCOM. (Sep. 2005). Digital Television Update - 2005 Q2, [Online]. Available: http://stakeholders.ofcom.org.uk/binaries/research/tv-research/q2_2005.pdf (visited on 05/19/2014).

[2] ——, (Apr. 2005x). The Communications Market 2005 2 Radio, [Online]. Available: http://stakeholders.ofcom.org.uk/binaries/research/cmr/part2.pdf (visited on 05/19/2014).

[3] 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/ (visited on 05/19/2014).

[4] L. Wang, G. von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu. (Apr. 2010). Cloud Computing: a Perspective Study, [Online]. Available: http://link.springer.com/article/10.1007/s00354-008-0081-5 (visited on 05/19/2014).

[5] J. Crowcroft, T. Moreton, I. Pratt, and A. Twigg. (Apr. 2010). Peer-to-Peer Systems and the Grid, [Online]. Available: http://www.cl.cam.ac.uk/teaching/2003/AdvSysTop/grid-p2p-paper.pdf (visited on 05/19/2014).

[6] J. Frank. (2014). Strengthening our policies for investigations, [Online]. Available: http://blogs.technet.com/b/microsoft_on_the_issues/archive/2014/03/20/strengthening-our-policies-for-investigations.aspx (visited on 05/19/2014).

[7] 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 (visited on 05/19/2014).

[8] E. Uhrhane. (Apr. 2014). W3C File API: Directories and System, [Online]. Available: http://www.w3.org/TR/file-system-api/ (visited on 05/19/2014).

[9] ——, (Apr. 2014). Web Applications Working Group mailing list: [fileapi-directories-and-system/filewriter], [Online]. Available: http://lists.w3.org/Archives/Public/public-webapps/2014AprJun/0010.html (visited on 05/19/2014).

[10] R. Sleevi and M. Watson. (Apr. 2014). W3C Web Cryptography draft, [Online]. Available: http://www.w3.org/TR/WebCryptoAPI/ (visited on 05/19/2014).

[11] I. Hickson. (Apr. 2014). W3C Web Storage draft, [Online]. Available: http://www.w3.org/TR/webstorage/ (visited on 05/19/2014).

[12] ——, (Apr. 2014). W3C Web SQL Database draft, [Online]. Available: http://www.w3.org/TR/webdatabase/ (visited on 05/19/2014).

[13] N. Mehta, J. Sicking, E. Graff, A. Popescu, J. Orlow, and J. Bell. (Apr. 2014). W3C Indexed Database draft, [Online]. Available: http://www.w3.org/TR/IndexedDB/ (visited on 05/19/2014).

[14] S. Kimak, J. Ellman, and C. Laing. (Apr. 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 (visited on 05/19/2014).

[15] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System May 2009, May 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf (visited on 05/19/2014).

[16] Namecoin.info. (Apr. 2014). Namecoin, [Online]. Available: http://namecoin.info/ (visited on 05/19/2014).

[17] D. Gilson. (Jun. 2013). What are Namecoins and .bit domains?, [Online]. Available: `http://www.coindesk.com/what-are-namecoins-and-bit-domains/` (visited on 05/19/2014).

[18] Ethereum.org. (Apr. 2014). Ethereum, [Online]. Available: `https://www.ethereum.org/` (visited on 05/19/2014).

[19] Keybase.io. (Apr. 2014). Keybase, [Online]. Available: `https://keybase.io/` (visited on 05/19/2014).

[20] I. Grigorik. (2013). High-Performance Browser Networking - XMLHttpRequest, [Online]. Available: `http://chimera.labs.oreilly.com/books/1230000000545/ch15.html` (visited on 05/19/2014).

[21] I. Fette and A. Melnikov. (Dec. 2011). The WebSocket Protocol, [Online]. Available: `http://tools.ietf.org/html/rfc6455` (visited on 05/19/2014).

[22] K. Egevang and P. Francis. (May 1994). The IP Network Address Translator (NAT), [Online]. Available: `http://www.ietf.org/rfc/rfc1631.txt` (visited on 05/19/2014).

[23] 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` (visited on 05/19/2014).

[24] I. Grigorik. (2013). High-Performance Browser Networking - UDP and Network Address Translators, [Online]. Available: `http://chimera.labs.oreilly.com/books/1230000000545/ch03.html` (visited on 05/19/2014).

[25] B. Ford, P. Srisuresh, and D. Kegel. (Feb. 2005). Peer-to-Peer Communication Across Network Address Translators, [Online]. Available: `http://www.brynosaurus.com/pub/net/p2pnat/` (visited on 05/19/2014).

[26] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. (Oct. 2008). Session Traversal Utilities for NAT (STUN), [Online]. Available: `http://tools.ietf.org/html/rfc5766` (visited on 05/19/2014).

[27] L. Jouanneau, J. Patonnier, and C. Mills. (Mar. 2014). Introduction to WebRTC architecture, [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/Guide/API/WebRTC/WebRTC_architecture` (visited on 05/19/2014).

[28] R. Mahy, P. Matthews, and J. Rosenberg. (Apr. 2010). Session Traversal Utilities for NAT (STUN), [Online]. Available: `http://tools.ietf.org/html/rfc5766` (visited on 05/19/2014).

[29] H. Alvestrand. (Jun. 2011). Google release of WebRTC source code, [Online]. Available: `http://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html` (visited on 05/19/2014).

[30] Google Inc. (Apr. 2014). WebRTC, [Online]. Available: `http://www.webrtc.org/` (visited on 05/19/2014).

[31] ——, (Apr. 2014). WebRTC Chrome, [Online]. Available: `http://www.webrtc.org/chrome` (visited on 05/19/2014).

[32] ——, (Apr. 2014). WebRTC Firefox, [Online]. Available: `http://www.webrtc.org/firefox` (visited on 05/19/2014).

[33] J. Roettgers. (Aug. 2012). Microsoft commits to WebRTC – just not Google's version, [Online]. Available: `http://gigaom.com/2012/08/06/microsoft-webrtc-w3c/` (visited on 05/19/2014).

[34] S. Dutton. (Jul. 2012). Getting Started with WebRTC, [Online]. Available: `http://www.html5rocks.com/en/tutorials/webrtc/basics/` (visited on 05/19/2014).

[35] I. Grigorik. (2013). High-Performance Browser Networking - WebRTC, [Online]. Available: `http://chimera.labs.oreilly.com/books/1230000000545/ch18.html` (visited on 05/19/2014).

[36] ——, (2013). High-Performance Browser Networking - WebRTC, [Online]. Available: `http://chimera.labs.oreilly.com/books/1230000000545/ch18.html` (visited on 05/19/2014).

[37] Chris Ball. (May 2013). WebRTC without a signaling server, [Online]. Available: http://blog.printf.net/articles/2013/05/17/webrtc-without-a-signaling-server/ (visited on 05/19/2014).

[38] Matasano Security. (Apr. 2010). JavaScript Cryptography Considered Harmful, [Online]. Available: http://www.matasano.com/articles/javascript-cryptography/ (visited on 05/19/2014).

[39] eroman@chromium.org. (Apr. 2010). WebCrypto implementation in Chromium, [Online]. Available: https://docs.google.com/a/chromium.org/spreadsheet/ccc?key=0Agiw0cuQZfVGdHNUNXBhZEFkazkyVy1uM1pISnlKRWc#gid=0 (visited on 05/19/2014).

[40] Microsoft. (2013). Web Cryptography, [Online]. Available: http://msdn.microsoft.com/en-us/library/ie/dn302338(v=vs.85).aspx (visited on 05/19/2014).

[41] Mozilla. (2013). JavaScript crypto, [Online]. Available: http://msdn.microsoft.com/en-us/library/ie/dn302338(v=vs.85).aspx (visited on 05/19/2014).

[42] M. Arora. (May 2011), [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1279264& (visited on 05/19/2014).

[43] Internet Engineering Task Force. (Jan. 1999), [Online]. Available: http://www.ietf.org/rfc/rfc2459.txt (visited on 05/19/2014).

[44] National Institute of Standards and Technology, *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*, Federal Information Processing Standards Publication, 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (visited on 05/19/2014).

[45] .NET Security Blog. (2006). The Differences Between Rijndael and AES, [Online]. Available: http://blogs.msdn.com/b/shawnfa/archive/2006/10/09/the-differences-between-rijndael-and-aes.aspx (visited on 05/19/2014).

[46] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Fergusonk, T. Kohno, and M. Stay. (Apr. 2010). The Twofish Team's Final Comments on AES Selection, [Online]. Available: https://www.schneier.com/paper-twofish-final.pdf (visited on 05/19/2014).

[47] National Institute of Standards and Technology, "Data Encryption Standard", *In FIPS PUB 46, Federal Information Processing Standards Publication*, 1977, pp. 46–2.

[48] E. Danielyan. (2001). Goodbye DES, Welcome AES.

[49] M. Rouse. (Jun. 2007). Cipher Block Chaining (CBC), [Online]. Available: http://searchsecurity.techtarget.com/definition/cipher-block-chaining (visited on 05/19/2014).

[50] BitTorrent Inc. (Feb. 2014). BitTorrent Sync, [Online]. Available: http://getsync.com/ (visited on 05/19/2014).

[51] RetroShare Team. (Feb. 2014). RetroShare, [Online]. Available: http://retroshare.sourceforge.net/team.html (visited on 05/19/2014).

[52] Peer5. (Feb. 2014). Sharefest, [Online]. Available: https://www.sharefest.me/ (visited on 05/19/2014).

[53] I. Clarke. (Feb. 2014). Freenet, [Online]. Available: https://freenetproject.org/ (visited on 05/19/2014).

[54] Tahoe-LAFS. (Feb. 2014). Tahoe-LAFS, [Online]. Available: https://tahoe-lafs.org/trac/tahoe-lafs (visited on 05/19/2014).

[55] Bitmessage Community. (Feb. 2014). Bitmessage, [Online]. Available: https://bitmessage.org/wiki/Main_Page (visited on 05/19/2014).

[56] J. Sicking. (Apr. 2014). Why no FileSystem API in Firefox?, [Online]. Available: https://hacks.mozilla.org/2012/07/why-no-filesystem-api-in-firefox/ (visited on 05/19/2014).

[57] E. Uhrhane. (Apr. 2014), [Online]. Available: http://lists.w3.org/Archives/Public/public-webapps/2014AprJun/0010.html (visited on 05/19/2014).

[58] 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 (visited on 05/19/2014).

[59] 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 (visited on 05/19/2014).

[60] 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 (visited on 05/19/2014).

[61] Bitcoin Project. (May 2014). bitcoind source code, [Online]. Available: https://github.com/bitcoin/bitcoin/ (visited on 05/19/2014).

[62] Wikimedia Commons. (2006). A diagram of Zooko's Triangle - a theory of the qualities of naming systems, [Online]. Available: https://commons.wikimedia.org/wiki/File:Zooko\%27s_Triangle.svg (visited on 05/19/2014).

[63] G. Wang and S. Quing, Two new attacks against Otway Rees Protocol, *IFIP/SEC2000, Information Security* Aug. 2000, 137–139, Aug. 2000. [Online]. Available: http://www.uow.edu.au/~guilin/papers/SEC00-137-fl.pdf (visited on 05/19/2014).

[64] M. Burrows, M. Abadi, and R. Needham, A logic of authentication, *ACM Transactions on Computer Systems* vol. **8** 1990, 18–36, 1990.

[65] R. M. Needham and M. D. Schroeder, Using Encryption for Authentication in Large Networks of Computers, *Commun. ACM* vol. **21**, no. 12 Dec. 1978, 993–999, Dec. 1978, ISSN: 0001-0782. DOI: 10.1145/359657.359659. [Online]. Available: http://doi.acm.org/10.1145/359657.359659 (visited on 05/19/2014).

[66] G. Lowe, An Attack on the Needham-Schroeder Public-key Authentication Protocol, *Inf. Process. Lett.* vol. **56**, no. 3 Nov. 1995, 131–133, Nov. 1995, ISSN: 0020-0190. DOI: 10.1016/0020-0190(95)00144-2. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(95)00144-2 (visited on 05/19/2014).

[67] C. R. Dougherty. (Apr. 2010), [Online]. Available: http://www.kb.cert.org/vuls/id/836068 (visited on 05/19/2014).

[68] 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.

[69] M. Watson. (Apr. 2010). WebCrypto Key Discovery, [Online]. Available: http://www.w3.org/TR/webcrypto-key-discovery/ (visited on 05/19/2014).

[70] B. Cavalier and D. Denicola. (2014). Promises/A+ specification, [Online]. Available: http://promisesaplus.com/ (visited on 06/05/2014).

[71] Peer.js. (May 2014). Peer.js - Github project, [Online]. Available: https://github.com/peers/peerjs (visited on 05/19/2014).

[72] Chain. (May 2014). Chain - Simple, powerful Blockchain APIs, [Online]. Available: http://chain-api.com/ (visited on 05/19/2014).

[73] Crypto Coin Insider. (May 2014). Namecoin, [Online]. Available: http://www.cryptocoinsinsider.com/namecoins/ (visited on 05/19/2014).

[74] P. Gil. (Apr. 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 (visited on 05/19/2014).

[75] Bitcoin.org. (Apr. 2014). Bitcoin – Open source P2P money, [Online]. Available: https://bitcoin.org/en/ (visited on 05/19/2014).

[76] F. Daoust, D. Hazaël-Massieux, and H. Alvestrand. (Mar. 2013). Web Real-Time Communications Working Group Charter, [Online]. Available: http://www.w3.org/2011/04/webrtc-charter.html (visited on 05/19/2014).

[77] I. Hickson. (May 2011). HTML5 – Offline Web Applications, [Online]. Available: http://www.w3.org/TR/2011/WD-html5-20110525/offline.html (visited on 05/19/2014).

[78] T. Wu. (Apr. 2010). bignumber-jt, [Online]. Available: http://www-cs-students.stanford.edu/~tjw/jsbn/ (visited on 05/19/2014).

[79] B. Kaliski and J. Staddon. (May 2008). Public-Key Cryptography Standards (PKCS) #8, [Online]. Available: http://www.ietf.org/rfc/rfc5208.txt (visited on 05/19/2014).

[80] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. (Sep. 1999). SPKI Requirements, [Online]. Available: http://www.ietf.org/rfc/rfc5208.txt (visited on 05/19/2014).

[81] B. Smith. (Sep. 1999). Additional steps to protect your privacy, [Online]. Available: http://blogs.technet.com/b/microsoft_on_the_issues/archive/2014/03/28/we-re-listening-additional-steps-to-protect-your-privacy.aspx (visited on 05/19/2014).

[82] M. Thomson. (Apr. 2013). Customizable, Ubiquitous Real-Time Communication over the Web, [Online]. Available: http://lists.w3.org/Archives/Public/public-webrtc/2012Oct/att-0076/realtime-media.html (visited on 05/19/2014).

Appendices

A 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

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

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

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

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.

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.

bignumber-jt http://www-cs-students.stanford.edu/ tjw/jsbn/. Library for bignumber arithmetic and RSA component generating.

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

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

Crypto.js https://code.google.com/p/crypto-js/. Collection of cryptographical functions.

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.