Decentralized Social Networking Site

Arend Koopmans

June 17, 2015

**Supervisor(s):** Reginald Cushing (UvA) & Adam Belloum (UvA)

**Signed:** Adam Belloum (UvA)
Abstract

As megacorps control more and more of our data on the Internet, the demand for a more decentralized Internet arises. An incentive for a decentralized website is a social networking site. There are already several decentralized social networks, but none of these networks use the less complicated, browser built-in HTML5 technologies. In this thesis the HTML5 technologies, WebRTC, Web sockets, Local storage and Web workers, are discussed in detail. Followed by an implementation combining these technologies into a decentralized social networking website. This implementation is tested by measuring the performance of a single user, the performance of a full-mesh network and the performance of a full-mesh network spread over multiple locations to simulate real latency. The results and implementation show that, although the website is not yet market-ready, it is relatively easy to create a decentralized social networking site with nothing but browser technologies. This could open the door for web developers to create a range of decentralized websites, which in turn, could contribute to the decentralization of the Internet as a whole.

Github repository - https://github.com/aijkoopmans/decentralized
Demo - http://decentral.herokuapp.com/
# Contents

1 Introduction .................................................. 4
   1.1 Research question ...................................... 4
   1.2 Method .................................................. 4
   1.3 Related work .............................................. 5
      1.3.1 Diaspora ........................................... 5
      1.3.2 Friendica ........................................... 5
      1.3.3 Tent ............................................... 5
      1.3.4 Synereo ............................................. 5
      1.3.5 Summary: Why another social network? ............. 6

2 HTML5 Technologies ........................................... 7
   2.1 WebRTC ................................................... 7
      2.1.1 Usage .............................................. 7
   2.2 Web sockets .............................................. 8
      2.2.1 Usage .............................................. 8
   2.3 Local storage ........................................... 9
      2.3.1 Web Storage ....................................... 9
      2.3.2 IndexedDB ......................................... 9
      2.3.3 File Access and File System ....................... 9
      2.3.4 Usage ............................................. 9
   2.4 Web Workers ............................................ 10
      2.4.1 Usage ............................................. 10

3 Implementation .............................................. 12
   3.1 Global plan ............................................. 12
      3.1.1 The role of the server and user .................... 12
      3.1.2 Use of HTML5 Technologies ......................... 12
      3.1.3 Ruby on Rails .................................... 12
      3.1.4 Phases ............................................ 13
   3.2 Local phase ............................................. 13
   3.3 Synchronization phase ................................. 15

4 Experiments ................................................ 19
   4.1 Testbed ................................................ 19
   4.2 Single user experiments ............................... 19
      4.2.1 Approach .......................................... 19
      4.2.2 Results receiving posts ......................... 20
      4.2.3 Results sending posts ............................ 20
   4.3 Breaking point experiments ........................... 21
      4.3.1 Approach .......................................... 21
      4.3.2 Results .......................................... 21
   4.4 Real-time latency experiments ....................... 23
      4.4.1 Approach .......................................... 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2</td>
<td>Results</td>
<td>23</td>
</tr>
<tr>
<td>4.5</td>
<td>Real-time latency experiments (2)</td>
<td>24</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Approach</td>
<td>24</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Results</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>In conclusion</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>Discussion</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>Future work</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>Conclusion</td>
<td>27</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

The Internet started out as a decentralized collection of computers. Technologies such as protocols were built from ground up with decentralization in mind [1]. Unfortunately what we are experiencing over the past years is an increasing centralization of the Internet where a growing percentage of Internet communication flows through megacorps such as Facebook, Google and others. Back in 2013 Internet traffic was said to have plunged by 40% as Google suffered a blackout. In other words, at that time, 40% of all traffic was owned by Google [2].

The utopic idea of the Internet is an Internet where data is not bound and controlled by policies and megacorps. Fortunately technology allows us to take back some of our freedom. With the latest advancements in browsers technology, more specifically in HTML5, decentralization becomes more easily accessible [3].

One incentive for a server-less website is a social networking site where any data you wish to share with your friends is only hosted on your browser and your friends’ browsers. This ensures your data is only hosted in a place you personally trust [4][5].

This research project entails leveraging latest browser technologies to write a basic decentralized social networking website. This website serves as an example of the possibilities in creating a decentralized website by implementing these HTML5 technologies.

1.1 Research question

To what degree can a web based social networking website be decentralized using latest advancements in browser technology?

1.2 Method

The HTML5 technologies that will be reviewed are WebRTC, Web sockets, Web workers and Local storage. WebRTC allows real-time communication between browsers. This capability opens the door to peer-to-peer communication between browsers without the necessity to relay data through servers [6]. Web sockets allow direct communication between a browser and a server the way TCP (Transmission Control Protocol) does. With this direct communication Web sockets are able to pull data from the browser without user interaction [7]. Local storage allows browsers to save persistent data which can survive browser restarts [8]. Web workers allow scripts in the browser to run on multiple threads [9].
1.3 Related work

The HTML5 technologies used in this project are relatively new. However, the concept of a decentralized social network or just any decentralized website preexisted. Since 2010 these type of social networks receive more and more attention. The biggest incentive for these networks are privacy and freedom from corporations. The following sections cover a few of the biggest decentralized social networks [4][5].

1.3.1 Diaspora

The biggest decentralized social network is Diaspora. It is a nonprofit, user-owned, decentralized social network that is based upon the free Diaspora software. The project was initiated in 2010 by four students of the Courant Institute of Mathematical Sciences at the University of New York.

Diaspora allows its users to host their data with a traditional web host, a cloud-based host, an Internet service provider, or a friend. The framework, which is built on Ruby on Rails, is free software and can be experimented with by external developers [10][11].

Since it launched, Diaspora has grown significantly. At the time of writing this thesis, Diaspora has more than one million users [12]. In comparison with other networks Diaspora is the biggest decentralized social network, but still, the chance of knowing someone on the network is very small. The network struggles to reach the mass [13].

1.3.2 Friendica

Friendica has an emphasis on extensive privacy settings and uncomplicated server installation. On top of that it aims at federating with as many other social networks as possible. Users can integrate contacts from Facebook, Twitter, Tumblr, StatusNet and other services. Communication with these services occur, when possible, in both directions [14].

The architecture includes a collection of distributed nodes which are able to act as a client or server to other nodes in the network and communicate with each other on the users their behalf. Each node is hosted on a server [15].

1.3.3 Tent

Tent differs from previous mentioned decentralized social networks. It aims much higher by decentralizing the Internet as a whole. Instead of building the tools their app need, it focuses on what their protocol should be able to do and what people can potentially do with it. With Tent, users can choose any hosting provider (or host their own Tent server) and any apps they want to use. Tent handles storing their data and sending it to their friends. It is built around posts. Each Tent server stores a single users posts and sends copies to the user’s subscribers [16].

If you look at Diaspora and Friendica in comparison to Tent, they could both exists within Tent as an app.

1.3.4 Synereo

Synereo differs from the other decentralized social networks by not only focusing on privacy, but also offering the opportunity for users to earn money through their accounts. With current social networks, the content, and thus the value a user creates, is owned by the social network itself. The social network decides whether to sell this content to a third party. With Synereo, the value the user creates stays in the hands of the user. It is up to the user whether it decides to sell this information to a third party. The currency Synereo uses is called an ”AMP”. This currency is deployed on top of Bitcoin’s blockchain [17].
The information flow within Synereo is based upon neural networks. There is no center of control and all messages are relayed through peers on the network. All of the users data is distributed across the community of Synereo users, where it is safely encrypted. This makes the data inaccessible to anyone but the entitled users [17][18].

1.3.5 Summary: Why another social network?

So, with all these decentralized social networks described in the previous subsections, the obvious question would be, why build another one? Firstly, they all use extensive software and technologies to create their network. Not one of these networks use the less complicated, browser built-in HTML5 technologies. Secondly, Diaspora, Friendica and Tent still use server nodes to provide data availability of their users. Thus users continue to rely on these servers to maintain the security, integrity, and reliability of their data. Thirdly, not one of these social networks has successfully won over the mass. In the eyes of the average Internet user they are unappealing and too complicated; they lack the approachability a social networking site needs [19][20].
CHAPTER 2

HTML5 Technologies

2.1 WebRTC

WebRTC provides browsers and mobile applications with Real-Time Communications (RTC) capabilities via simple API's. The WebRTC components have been optimized to best serve this purpose. This capability opens the door to peer-to-peer communication between browsers without the necessity to relay data through servers. As real-time communication is time dependent, WebRTC uses UDP (User Datagram Protocol) to transport its data. In contrast to TCP (Transmission Control Protocol), which is used for all other browser communication, UDP does not make provisions for verifying whether data has arrived and is intact. On top of UDP, WebRTC implements ICE (Interactive Connectivity Establishment), STUN (Session Traversal Utilities for NAT) and TURN (Traversal Using Relays around NAT). These protocols are required in order to establish and manage a peer-to-peer connection over UDP. To secure data transfers between peers, DTLS (Datagram Transport Layer Security) is used. SCTP (Stream Control Transmission Protocol) and SRTP (Secure Real-time Transport Protocol) are used to provide partially reliable delivery [6].

2.1.1 Usage

To reduce the complexity of WebRTC and focus on the peer-to-peer communication between browsers, an API called PeerJS is used. PeerJS wraps the browser’s WebRTC implementation into a complete, configurable peer-to-peer connection API. Equipped with nothing but an ID, a peer can create a data connection to a remote peer. PeerJS has the BinaryPack serialization format built-in. This means it is possible to send any JSON type as well as binary Blobs and ArrayBuffers [21].

The following code snippet displays a usage example of PeerJS. A peer connects to a remote peer. When a connection with the remote peer is established, it starts listening for incoming messages from this connection. The peer can also send messages to this connection.

```javascript
1 // Create a peer
2 var peer = new Peer('your-ID', host);
3
4 // Connect to another peer
5 var conn = peer.connect('another-peers-ID');
6
7 conn.on('open', function() {
8     // Receive messages
9     conn.on('data', function(data) {
10         console.log('Received', data);
11     });
12     
13     // Send messages
14     conn.send('message to send');
15 });
16
17 // Disconnect connection
18 conn.close();
19
20 // Clear events
21 conn.off();
```

7
With WebRTC and the PeerJS API, peer-to-peer connections can be made and data can be transferred between peers. This is an indispensable part of creating a decentralized social networking site. At the time of writing, WebRTC is solely supported in newer versions of Firefox, Chrome and Opera [22].

2.2 Web sockets

The web has been largely built around the so-called request/response paradigm of HTTP. A client requests a web page and in response nothing happens until the user navigates to the next page. Around 2005, AJAX started to try and make the web feel more dynamic. AJAX offers the possibility to create asynchronous web applications. With AJAX, web applications have the ability to send data to and retrieve data from a server asynchronously. This happens without interfering in the behaviour of the existing page. Nevertheless, still all HTTP communication was steered by the client which requires user interaction or periodic polling to load new data from the server [23].

Work-arounds that enable the server to send data to the client in the very moment it knows that new data is available have been around for quite some time. However, all of these work-arounds share one problem: They carry the overhead of HTTP, which does not make them well suited for low latency applications [7].

The Web sockets specification defines an API, which establishes "socket" connections between a web browser and a server. In simple terms: there is a persistent connection between the client and the server. Both parties can initiate sending data at any given time [7].

2.2.1 Usage

Web sockets can establish a low latency, real-time connection, between the client and the server. With this, it can be used to update social streams in real-time. At the time of writing, all new versions of the major browsers support Web sockets [24].

The following code snippet displays a usage example of Web sockets using socket.io and Node.js. The server listens to the socket and transmits all incoming messages back to every connection on the socket. The client also listens to the socket and prints incoming messages. When a client sends a message to the server through the socket, all users connected to this socket receive this message from the server.

The server side.

```javascript
var io = require('socket.io')(http);

// Listen to chat messages of clients and transmit chat messages to all clients
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

The client side.

```javascript
var socket = io();
```
2 // Listen to server for new chat messages
3 socket.on('chat message', function(msg) {
4     console.log(msg);
5 });
6
7 // Send message
8 socket.emit('chat message', 'hi');

2.3 Local storage

Local storage provides a way for web applications to store data locally within the user's browser. Before HTML5, application data could be stored in cookies that are included in every server request. Local storage is more secure, and large amounts of data can be stored locally without affecting the website's performance. In contrast to cookies, the storage limit is far larger (up to multiple GBs) and information is never transferred to the server. Local storage is per domain. All pages, from one domain, can store and access the same data [8].

There are four different types of Local storage: Web storage, IndexedDB, File Access and Web SQL Database. Since Web SQL Database is deprecated, it will not be discussed as an option [25].

2.3.1 Web Storage

Web storage simply provides a key-value mapping, for example: localStorage["name"] = username. Present implementations only support string-to-string mappings. Therefore it is required to serialize and de-serialize other data structures [8].

2.3.2 IndexedDB

IndexedDB provides a structured, transactional, high-performance NoSQL-like data store with a synchronous and asynchronous API. The API permits you to create databases, data stores and indexes, handle revisions, populate data using transactions, run non-blocking queries, and traverse data sets using cursors [26][27].

2.3.3 File Access and File System

File Access is an API for reading file content in JavaScript. A file that has been added to a file input element can be read as a URL. A File System API can be used for storing the file corresponding to the URL [28].

2.3.4 Usage

Web storage lacks functionality, but since it uses key-value mapping, it is easy to implement. At the time of writing, all new versions of major browsers support web storage [29]. IndexedDB has more functionality but the API is more difficult to implement. IndexedDB is not yet fully supported in all browsers. Newer versions of Firefox, Chrome and Opera offer full support. Safari and Internet explorer are not yet fully adjusted, but do give some support [30].

The community around Local storage currently recommends IndexedDB [31][32][33]. There are a few wrappers available which facilitates implementation. One of these wrappers is Dexie.js [34].

The following code snippet displays a usage example of Dexie.js. A new database is created with a table friends. The process of adding a record and retrieving records is illustrated.
// Make a database
var db = new Dexie('MyDatabase');

// Define a schema
db.version(1).stores({
  friends: 'name, age'
});

// Store a friend
db.friends.add({
  name: 'Camilla',
  age: 25
});

// Find some friends
db.friends.where('age').below(50).each(function(friend){
  console.log(friend.name);
});

For the purpose of storing files, and more specifically photos, Chromstore offers a convenient solution. Chromstore provides support for persistent file storage in the browser [35].

The following code snippet displays a usage example of Chromstore. A photo is created on the local File System.

// Retrieve or create a file
cs.getFile('photo.jpg', {create: true, exclusive: true}, function(fileEntry){
  console.log('File created');
});

2.4 Web Workers

Web workers ensure the availability of threads in the browser. Threading is important when doing compute intensive tasks. Traditionally any script in the browser ran on one single thread, therefore any computation interfered with the responsiveness of the website itself [9].

2.4.1 Usage

In this case Web workers can divide the tasks of WebRTC, Web sockets and Local storage over multiple threads. At the time of writing, all new versions of major browsers support Web workers [36].

The following code snippet displays a usage example of a Web worker. The main script initiates the worker. The main script listens to the worker and sends a message to the worker. The worker also listens to the main script. When a worker receives a message, it send back a message to the main script.

Main script.

// Create the worker
var worker = new Worker('doWork.js');

// Listens to the worker
worker.addEventListener('message', function(e) {
  console.log('Worker said:', e.data);
}, false);
9 // Transmit data to worker
10 worker.postMessage('Ping');

---

doWork.js (the worker).

1 // Process incoming data
2 self.addEventListener('message', function(e) {
3   // return data to the listener
4   self.postMessage('Pong');
5 }, false);
CHAPTER 3

Implementation

3.1 Global plan

Taken into account that the time for implementation is limited, the system remains partially centralized. Servers will still be used for the discovery of friends and updates. This ensures the implementation is less complex and therefore achievable within the given time frame.

The website should offer users the possibility to create a profile with a name and profile picture. Users have the ability to add other users as friends. They can create posts consisting of text and optionally an image. These posts appear on the user’s profile as well as the global timeline. Users have access to their friends’ profiles and posts.

3.1.1 The role of the server and user

The server will keep track of the authentication of users and the friendships established between users. The server does not store any personal data, it only indexes which user hosts which data. Friendship between users implies a certain amount of trust. Therefore personal data of friends, as well as the user’s own personal data, will be locally stored on the user’s browser. This ensures personal data is only stored in a trusted place.

3.1.2 Use of HTML5 Technologies

IndexedDB and File Access, in combination with a File System, locally store the data of the users. WebRTC establishes a real-time connection between users. This connection allows the users to exchange data. Because users update each other social streams in real-time, through WebRTC, a persistent connection between the server and user is therefore redundant. This makes the use of Web sockets unnecessary. AJAX can be used to send and retrieve data from the server without interfering with the behaviour of the existing page. Web workers can be used at a later stage to distribute the tasks over multiple threads, which in turn increases the stability of the browser.

3.1.3 Ruby on Rails

The application will be built in Ruby on Rails. Rails is a model-view-controller structured web-framework that abstracts and simplifies common repetitive tasks. Rails is written in Ruby as Symfony and Zend are written in PHP and Django is written in Python.

The main principle of Ruby on Rails is convention over configuration. In other words, the programmer does not have to spend a lot of time setting up and configuring files. Rails comes with a set of conventions which speed up the development. As stated before, Rails uses the
programming language Ruby. Since Ruby gives the possibility to program on a meta-level, and Rails uses this, the application will be very readable and self-documenting. On top of this Rails has a large community and a great set of third-party libraries called Gems [37].

For the authentication a Gem called Device is used. Device is a flexible authentication system that makes it easy to create, update and access users. It is very secure and widely used [38]. AJAX handles the navigation between pages and submitting forms. Instead of implementing this per page, the Gem Ajaxify automatically makes the application load content in the background via AJAX [39]. Gems RSpec and Capybara give the ability to test the website [40][41].

3.1.4 Phases

The implementation can be roughly split into a local phase and a synchronization phase. The local phase focuses on the local storage of the user’s data. The synchronization phase focuses on synchronizing the users information, profile pictures and posts through a peer-to-peer network using WebRTC.

3.2 Local phase

The local phase implements a basic authentication system for users. The previously mentioned Gem Device is used for this purpose. The user model has a complete set of attributes including an email, password, sign_in_ip, sign_in_at.

The first thing a user need is a name. A model called information is created. Anticipating a user will need more basic information in the future (e.g. an address or phone number). A user has one information and an information belongs to one user. When an information object is created it is stored on the server with an id, user_id, created_at and updated_at attribute. The attribute name is only stored locally in the IndexedDB.

The user also needs a profile picture. A model called profile_picture is created. A user has one profile_picture and a profile_picture only belongs to one user. When a profile_picture object is created it is stored on the server with an id, user_id, created_at and updated_at attribute. The image itself is only stored locally in the File System using File Access.

Now that the users have a basic profile, they should have the ability to become friends with one another. In order to achieve this a model called friendships is created. A user has many friendships. Friendships contain a user_id, friend_id and a boolean confirmed. If a user A invites a user B to be friends, a friendship is created with user_id of A and friend_id of B. The boolean confirmed is stating false. If user B confirms this invitation there is another friendship created with of user_id B and friend_id of A. The boolean confirmed is true. The boolean confirmed of the first friendship is now also true.

Finally the user needs to be able to create posts with some text and optionally an image. A model posts is created. A user has many posts and a post only belongs to one user. When a post is created it is stored on the server with an id, user_id, created_at and updated_at attribute. The text attribute is only stored locally in the IndexedDB and the image is only stored locally in the File System.

Figure 3.1 displays the database diagram of these models and their relation to each other.
The processes of creating, updating and deleting informations, profile pictures and posts are quite similar. Since creating and sharing posts is the most extensive, all examples that illustrate implemented processes will be based on posts.

To create a post a user has to submit a form. This form has a text field and a file input element for the image. When a user submits this form a post is created on the server. Once the post is saved to the server, the post is locally created. The image is stored on the File System and the text attribute is saved to the IndexedDB. Finally the HTML is updated. As the whole site uses AJAX this process finishes without reloading the page.

Figure 3.2 displays the sequence diagram corresponding to the process above.

Figure 3.1: The database diagram for the local phase with relations between the server and client tables
### 3.3 Synchronization phase

To enhance the availability of users their data, the moment a user signs in, it synchronizes all possible data of his friends. This happens regardless whether a user visits the profiles of these friends or not.

A user can make a connection to his friends as well as to friends of friends. This allows user $A$ to get data from user $B$ by directly connecting to him. If user $B$ is offline, user $A$ can still get data of user $B$ by connecting to a friend of user $B$. For example user $C$. This user $C$ is not necessarily a friend of user $A$. This also enhances the availability of users.

Figure 3.3 and Figure 3.4 illustrate the processes above.

![Figure 3.3: B's data from B](image)

![Figure 3.4: B's data through a friend of B](image)

There are two ways of synchronizing data. The first way consists of making a connection to a user without knowing if the user has useful data. As data is hosted by friends of users, the user has to connect to all online friends of his offline friends, to synchronize the data of these offline friends. Figure 3.5 displays an example of a user $A$ with four offline friends ($B$, $C$, $D$ and $E$). These four friends each have another four friends. These friends are online. When user $A$ wants to get the data of users $B$, $C$, $D$ and $E$ it has to connect to all online friends of users $B$, $C$, $D$ and $E$ to make sure it has the most recent data. The second possible way to synchronize data consists of preparing a *to do* list before connecting to other users. This *to do* list contains which data is not yet synchronized and which users host this data. The user will now only connects to useful users. Instead of connecting to all friends of users $B$, $C$, $D$ and $E$, user $A$ only connects to the most useful friend of users $B$, $C$, $D$ and $E$. This results in connecting to only four users, one friend of user $B$, one of user $C$, one of user $D$ and one of user $E$. When user $A$ connects to one of these users, it knows directly what useful data is hosted with this user.

![Figure 3.5: The network of user A with four offline friends, who have three online friends each](image)
If we assume a network with an average node degree of $n$, where all immediate $n$ friends are offline, the amount of connections to be made in the first case equals $n(n - 1)$. The amount of connections to be made in the second case equals $n$. The only downside of the second method is that the server has to store index information containing which users host which data and what version of this data.

To allow the server to index which data is hosted at which user, version tables are needed on the server side. The moment a user creates a post, updates his information or his profile picture, there also is a record created in the corresponding version table. When a user $A$ creates a post $X$, a post_version is created with post_id of post $X$ and user_id of user $A$. This means that user $A$ hosts his own post $X$. The created_at and updated_at attributes are a form of version tracking.

Figure 3.6 displays the database diagram with the version tables and their relations.

The moment a user signs in to the website it requests the post, profile_picture and information tables of the server side and compares it to his own post, profile_picture and information tables. If an item, for example a post, is out of date the user does a second call to the, in this case, post_versions table. From this table the user can see which user hosts the specific post. This is stored in the to do list of the user.

The following pseudocode describes the process above.

```plaintext
1 function prepare_to_do_posts()
2  for server_post in server_posts do
3      id = server_post.id
4      # the post exists local
5      if server_post.version > local_post.version
6          # the post is outdated
```
After preparing the to do list the next step consists of establishing connections to certain users. A user always opens up a connection with his online friends. As well as synchronizing data that the user missed while it was offline, it creates the possibility to send out real-time updates to friends through this connection. On top of this, the user opens up a connection to useful friends of friends that are added to the to do list.

The following pseudocode describes the above process.

```python
function connect_to_peers()
for peer_id in friends
    self.connect(peer_id)
for peer_id in useful_users
    self.connect(peer_id)
```

When a user A makes a connection to a user B, a request with the id of, for example, a post is sent. User B replies with an update message containing the sensitive data of the post. User A then updates his IndexedDB and local File System. Subsequently a post version is created at the server database, implying that user A hosts a post of user B. Finally the to do list is updated so user A does not connect again to user B or any other user to synchronize this specific post.

The following pseudocode as well as figure 3.7 illustrate the above process.

```python
function listen()
if type == request
    # send the post to the requesting user
    post = local_post.get(post.id)
    peer.send(update, post, self)
else if type == update
    # update or create the post locally
    local_posts.put(post.id, post.text, post.image)
    # update or create a post version for the current user
    server_post_versions.put(post.id, post.version, self.id)
    # update the to do so the current user knows it has been completed
    to_do_posts.update(post.id)
```
When a user creates a post, the local phase, where the post in created, is completed. Subsequently to this phase, the user tells the server it hosts its own post. In addition the user sends out a real-time update to his currently online friends.

Figure 3.8 displays the full sequence diagram of creating a post for a user.

Users are now able to synchronize data with other users in a smart way and create data in real-time relative to the other online users. They also provide availability for other users, even the other users are not online.
This chapter covers experiments to see how well the implementation from the previous chapter performs.

The performance is measured by the total amount of posts synchronized and the average time a post takes to synchronize. The time a post takes to synchronize is measured between the moment the post is created at one user and the moment the post version of the receiving user is created. Every measurement of every experiment is repeated at least three times to guarantee reliable results.

4.1 Testbed

The testbed consists of multiple virtual machines created with Docker [42]. Each instance of Docker launches an instance of Google Chrome using Selenium [43]. The virtual machines are divided over hubs. In the first two experiments all virtual machines are hosted on one server in Amsterdam. To simulate latency in the third experiment, a second location is added by using Amazon EC2 (Elastic Compute Cloud) in Ireland. In addition a third location was added using a home computer situated in Amsterdam.

The website is hosted through Heroku. Heroku is a cloud platform as a service (PaaS) supporting several programming languages, including Ruby on Rails. Heroku works with Dynos. Together they are the unit of computing power on Heroku, providing lightweight, isolated containers that run an application [44].

With Heroku, the server and database can be scaled easily for a short period of time to boost the performance. This is necessary to make sure the server and database are not the problem in case of failure.

During the experiments the server is upgraded to 1024 MB RAM. The amount of so called Dynos is increased to four and the database is also upgraded to 1024 MB RAM. In terms of money the server would now cost 250 dollar a month.

4.2 Single user experiments

4.2.1 Approach

In these experiments there is one online user with an increasing amount of online friends. These friends do not have any other friends apart from this user.

The first experiment tests how many posts the user can handle receiving simultaneously. The
friends of this user all post a single post at the exact same time. The amount of posts the user has to synchronize is equal to the amount of friends the user has.

The second experiment tests how well one post of the user synchronizes to his friends. The user sends out a single post to each of his friends at a given moment.

4.2.2 Results receiving posts

Figure 4.1 displays the percentage of posts synchronized relative to the amount of friends. Figure 4.2 displays the average time for a post to reach the receiving user.

![Figure 4.1: Percentage of posts synchronized](image1)

![Figure 4.2: Average time for posts to synchronize](image2)

The results show that after the point of receiving 128 post at the same time, not every post reaches the receiver. A possible explanation would be the inability of a browser or virtual machine to handle this many posts simultaneously. The average time of synchronization grows linear to the amount of users. This is to be expected.

4.2.3 Results sending posts

Figure 4.3 displays the percentage of posts synchronized relative to the amount of friends. Figure 4.4 displays the average time for a post to reach his friends.

![Figure 4.3: Percentage of posts synchronized](image3)

![Figure 4.4: Average time for posts to synchronize](image4)

The results show that after the point of sending 128 posts at the same time, not every post reaches the receiver. A possible explanation would be the inability of a browser or virtual machine to handle this many posts simultaneously. The average time of synchronization grows approximately linear to the amount of users. This is to be expected.

Sending posts is 3.4 times faster than receiving posts. A possible explanation could be the distribution of saving versions to the server. The moment a user receives multiple posts, that user has to make multiple calls to the server. When one user sends out a post to multiple users, these calls are distributed over these users.

### 4.3 Breaking point experiments

#### 4.3.1 Approach

Scalability is a well known issue in computer science and certainly in peer-to-peer processes [45]. This experiment searches for the breaking point in the system. For this purpose a full mesh network with an increasing amount of online friends will try to synchronize posts. The number of connections in this network increases with the formula $n(n-1)/2$, as formulated by Metcalfe’s law [46]. Metcalfe’s law has been used to explain the growth of many technologies, and more particularly social networks. It states that the value of a social network increases as the number of users increases, because potential links increase for every user as a new person joins [46].

In this network all users post a single post simultaneously. Since all users are friends in a full mesh network, every user has to synchronize every post, but his own. The amount of traffic containing a post is $n(n-1)$.

The connections and traffic in the network grow exponentially. Figure 4.5 displays the number of connections in this network relative to the amount of users as well as the increase in traffic relative to the amount of users.

![Figure 4.5: The number of connections in this network relative to the amount of users and the increase in traffic relative to the amount of users.](image)

The full mesh network increases with eight users at every step with a maximum of 64 users in total. Increasing the amount of users to 72 resulted in inconsistent data.

#### 4.3.2 Results

Figure 4.6 displays the percentage of posts synchronized relative to the amount of users. Figure 4.7 displays the average time for a post to synchronize to friends of a user. Figure 4.8 displays...
the absolute amount of posts synchronized.

Figure 4.6: Percentage of posts synchronized

Figure 4.7: Average time for posts to synchronize

Figure 4.8: Absolute amount of posts synchronized

The results show that after the point of 24 users, not every post reaches its receiver. This is at approximately 576 posts.

The single user experiments show that a single user can handle receiving up and until 128 posts simultaneously, as well as sending out up and until 128 posts simultaneously. This means the user’s browser or virtual machine should not cause the loss of posts.

All Dockers share the same Ethernet bridge in this setup, which means one location has to process every post. A possible explanation would be the inability of the virtual of network of the server to process this large amount of posts simultaneously.

A possible explanation for the sudden drop in average synchronization time at 64 users would be the drop in the absolute amount of posts synchronized. Figure 4.8 displays that approximately 400 posts do not reach their destination, which could mean the virtual of network of the server has less posts to process.
4.4 Real-time latency experiments

4.4.1 Approach

To simulate real latency the virtual machines are distributed over different locations. In addition to the virtual machines launched from the server in Amsterdam two locations are added. A second location launches virtual machines from Amazon EC2 in Ireland. A third location runs multiple browsers on a home computer situated in Amsterdam.

4.4.2 Results

Figure 4.9 displays the percentage of posts synchronized relative to the amount of users. Figure 4.10 displays the average time for a post to synchronize to friends of a user. Figure 4.11 contains the absolute amount of posts synchronized to display the absolute amount of traffic.

In figures 4.9, 4.10 and 4.11 the blue line represents the performance of the server in Amsterdam. The orange line represents the performance with users divided over two servers. One in Amsterdam and one from Amazon EC2 in Ireland. The red line represents the performance with users divided over the two servers previously mentioned and a third home computer situated in Amsterdam.

The results show that users distributed over two or three locations synchronize less posts com-
pared to users at one location.

This could be due to the fact that WebRTC uses UDP. Even though it uses SCTP and SRTP to provide partially reliable delivery. It is possible that Google Chrome might not yet support reliable data channels [47].

When distributing the users over two or three locations the average time a post takes to synchronize improves by a factor of 3.1. This is to be expected as the Dockers are now spread over multiple Ethernet bridges, giving every location less posts to process.

Due to the complexity of setting up users on different locations, testing it with more than 32 users was not possible within the time.

4.5 Real-time latency experiments (2)

4.5.1 Approach

In addition to the previous experiment which only measured the amount of posts synchronized and the average time for posts to synchronize, an additional experiment was conducted. This experiment provides insight to where posts get lost. For this purpose a full mesh network of 24 users equally distributed over the three, earlier mentioned, different locations will try to synchronize posts.

Note that this additional experiment was conducted on a different day from the previous experiments. This resulted in a performance drop of 12% in the amount of posts synchronized.

4.5.2 Results

Figure 4.12 displays the percentage of posts synchronized per location and between locations. The percentage at the location displays the percentage of posts synchronized between the users at that location. The percentage between the locations displays the percentage of posts synchronized between these locations. The amount of posts to be synchronized between users at one location is equal to 64. The amount of posts to be synchronized between locations is equal to 128.

![Figure 4.12: Database diagram for the synchronization phase](image)

The results show that users at the home location outperform the users simulated with Docker on the servers. A possible explanation for the lesser performance of Dockers could be found in the connection setup and NAT traversal between Dockers hosted on virtual machines. The connection setup in WebRTC follows ICE which means that the setup tries to do NAT traversal by "punching holes" in the firewall. The fact that Dockers are hosted on virtual machines means
there are at least two NATs in action - one between Docker and the virtual machine and an
other one between the virtual machine and the actual host. This might make the peer-to-peer
connection setup fragile. Which may force the use of relay servers (TURN servers). This would
impact the connection.

The users on the server in Ireland have the lowest rate of performance. This could possibly be
explained by the use of a less performing server.

Due to time limitation, it was not possible to take a deeper look into the NAT traversal between
Dockers.
CHAPTER 5

In conclusion

5.1 Discussion

All experiments had a decent failure rate. Apart from explanations in the previous chapter, this failure rate was to be expected as the setup of the virtual machines through Docker was quite complex and everything was done on either one, two or three locations. It should also be noted that numerous virtual machines crashed occasionally when testing full mesh networks. The amount of crashes ranged from no crashes in networks smaller than 24 users to 5%-10% in larger networks. The results are therefore deviant from a setup with real users.

5.2 Future work

As mentioned in previous sections the testbed is a considerable cause for the failure rate regarding post arrival. An alternative for Docker is Saucelabs [48]. In essence Saucelabs works the same way Docker does, as it also launches a Selenium instance of Chrome on multiple virtual machines. Comparing Saucelabs to Docker could give insight to what extent the testbed was responsible for the failure rate.

To improve the failure rate, it is important to implement a recovery system. A recovery system would guarantee the arrival of posts which could significantly increase the percentage of posts synchronized. Since a recovery system resends posts that would otherwise be lost, the average synchronization time increases. Post delivery is a crucial part of a functioning social network, the expected increase in the percentage of posts synchronized therefore outweighs the additional synchronization time.

By default, Javascript works on one thread. By adding Web workers processes can be distributed over multiple threads. This distribution could increase the stability of the browser.

Web sockets could also attribute to the stability of the browser, by taking over different tasks of the user. The preparation of the to do could be shifted from the user to the server. Assuming a recovery system is implemented, the server could also fulfill the task of ensuring post delivery.

As the website is currently partially centralized, decentralizing the website entirely is an important and crucial part of the future work. By implementing a Gossip Protocol, index information at the server could be distributed over the nodes. Gossip protocols are an efficient way of keeping nodes updated about new information. Through a deterministic or randomized algorithm the nodes communicate and exchange this information at each communication step [49].

An other obstacle that has to be overcome in order to make this website viable, is the availability of the first users. At this point users depend on their friends for the availability of their data.
No online friends means no available data. Having very few friends to start with, this generates a problem. Beacons (or super nodes) could offer a solution by kickstarting the system. Beacons will serve temporarily as friends that are always online. After reaching a point of independence where your profile is available solely through friends, the beacons will be disconnected. Diaspora and Friendica also use this form to ensure the availability of users. In contrast to Diaspora and Friendica, this website disposes the beacons after a certain point, ensuring full decentralization [11].

Finally to make this website market-ready, security measures should be implemented. Securing every connection and every message is an indispensable part of this website. Fully securing the website is an extensive process which will not be further discussed in this thesis.

5.3 Conclusion

This thesis covered the implementation of a social networking website using latest advancements in browser technology. The research question stated to what degree this was possible.

The result is a basic decentralized website. The website offers users the possibility to create a profile with a name and profile picture. Users have the ability to add other users as friends. They can create posts consisting of text and optionally an image. These posts appear on the user’s profile as well as the global timeline. Personal data of users is only stored on the browser of the user itself and on the browser of their friends. This ensures personal data is only stored in a trusted place. The server solely indexes which users host which data. It does not store any sensitive information.

As can be concluded from the experiments and future work, the website is not yet market-ready. Improvements in synchronization, availability and security are essential to create a fully operative version of this website. Finally browser technologies IndexedDB and WebRTC still await full browser support. At the time of writing, the website only runs on Google Chrome and Firefox.

This thesis demonstrated the possibility to create a decentralized website using latest advancements in browser technology. It also shows that the implementation of these advancements proves to be relatively easy in comparison to the technologies used in other decentralized social networks. This straightforward implementation could open the door for web developers to create a range of decentralized websites, which in turn, could contribute to the decentralization of the Internet as a whole.
Bibliography


