Zwiebelnetz

-Tech Report-

Dario Brandes, Thies Johannsen,
Paul Kröger, Sergej Mann,
Roman Naumann, Sebastian Thobe

September 19, 2014

Contents

Todo list

1 Introduction

The rise of online communications and social networks affect the way we interact today. A great deal of personal information is stored on electronic devices and transmitted over the Internet. While the recent revelations by Edward Snowden showed the extent of government agencies spying on personal data, it was understood before by the security community that there is a need to better protect privacy in online communications.

A number of promising projects, such as Tox [8], were started to thwart prying on user data. We found, however, no project for social networking with a satisfactory focus on privacy.

The Zwiebelnetz, described in this tech report, is a prototype of a social network which seeks to protect user data.

2 Overview

The Zwiebelnetz currently provides basic features for social networking, such as contact requests, circle management, posts, comments and profile information.

The data published in the network is hosted in a distributed fashion; each user has an embedded system connected to the internet which stores and serves the data she published.

Figure 1 shows the main components of the Zwiebelnetz:

Syncer Daemon Keeps databases of several systems in sync, see section 3

Tor Damon Serves as a proxy to the Internet, see subsection 3.2.

Web Daemon Serves the web frontend to the local network (see section 5).

Database An SQLite3 [1] database storing all user content, see section 4

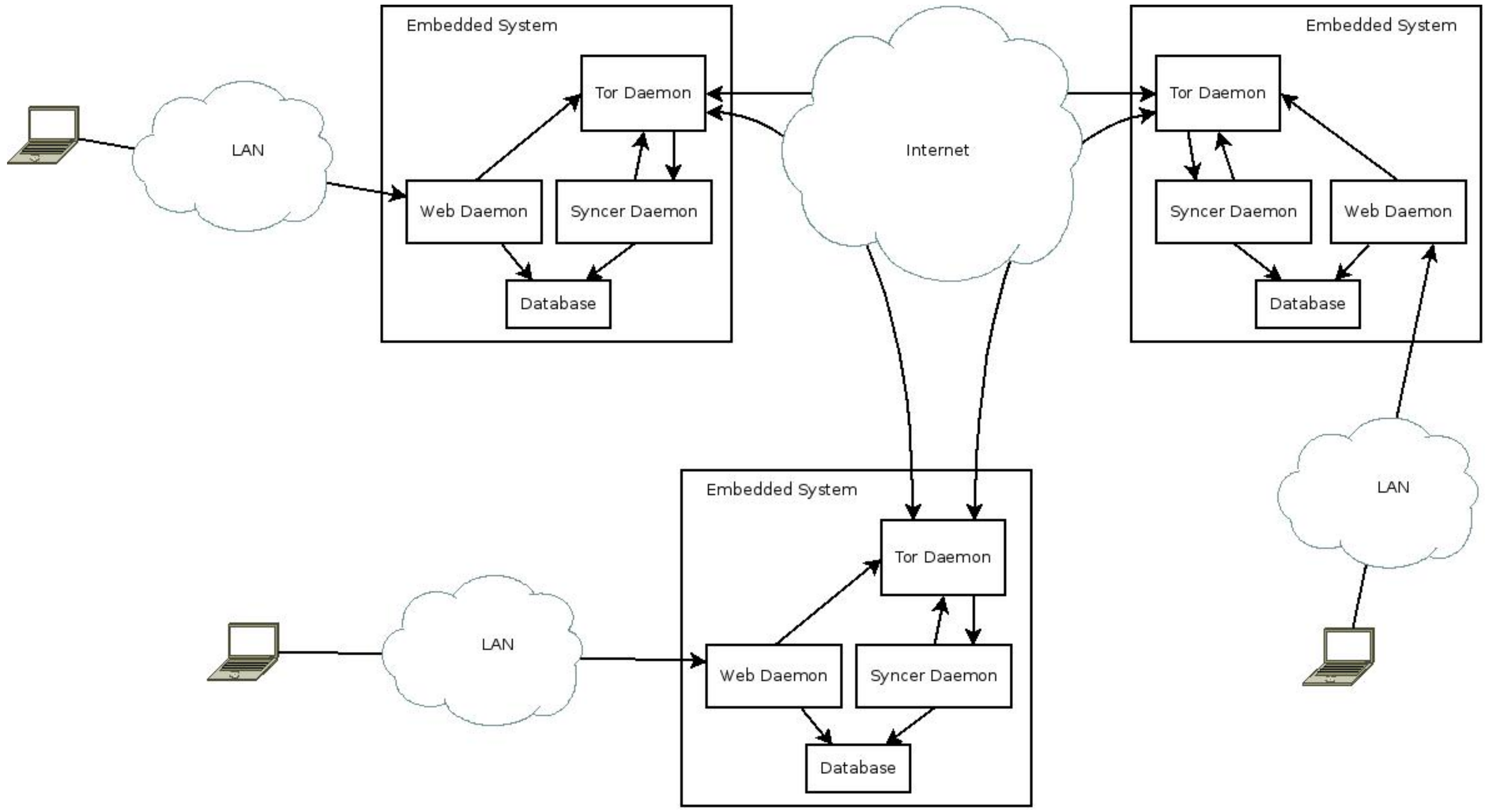


Figure 1: Architecture Overview

3 Network Protocol

3.1 Terms

The following terms will be used in this section:

syncer daemon The system process which provides the synchronization hidden service and regularly pulls updates from other syncers.

web daemon The system process which serves the web interface to the local network.

client Despite the peer-to-peer nature of the protocol, it is useful to distinguish between the connecting entity (giver) and the serving entity (taker). we describe the *connecting* syncer daemon as the client.

server The server describes the syncer daemon which handles a connection of a client.

3.2 Tor Usage

The `syncer daemon` runs on the embedded system, it listens on port 3141 on localhost only. Other systems do not communicate directly with the syncer over the Internet, but connect to it via Tor hidden services [7]. Syncer access the Tor daemon uses the `SOCKS4a` protocol [4], an extension to the `SOCKS4` protocol [5] adding support for hostnames [1].

We use Tor for three reasons:

1. Network Address Traversal (NAT) and Name Resolution
2. Cryptographic Guarantees
3. Hidden Social Graph

NAT and Name Resolution

The Tor network serves a distributed hash table (DHT) of hidden services' onion addresses. Given an onion address, a client can use the Tor daemon

[1]The hostname support is required since Tor resolves onion addresses, not IP addresses

to establish a connection to the hidden service. No DNS name or static IP address is required, all information needed can be found in the Tor DHT. It is common for home users' Internet connectivity to have dynamic IP addresses without a hostname bound to those addresses.

Modern routers as found in home environments commonly use NAT to map the addresses and ports used on the local network to the IP addresses and ports used on the internet. The Tor daemon which provides the hidden service establishes several introductory points through the Tor network and keeps TCP connections to the first node of each path open. This effectively bypasses the NAT barrier so that users of the system do not need to configure their routers to forward specific ports.

Cryptographic Properties

The tor hidden service protocol provides secrecy, data integrity and authentication [3]. Due to the nature of tor - being an anonymous communication service - the server syncer does not know which system the connection originated from. We therefore implemented a one-directional challenge response protocol, described in section 3.4, to authenticate the connecting client.

Hidden Social Graph

The Zwiebelnetz builds an overlay network with a friend-to-friend topology. The social graph of users can easily reconstructed from such connection (meta-)data. Tor's onion routing protects the connection data by attackers being able to observe and manipulate parts of the Internet. The use of Tor as a means to build the friend-to-friend overlay hides the social network from those attackers.

3.3 Syncer States

The syncer accepts new connections and waits for further instructions. The Figure 2 shows the state machine followed by each syncer thread handling a new connection. Terminal states close the connection after the commands of the state were executed.

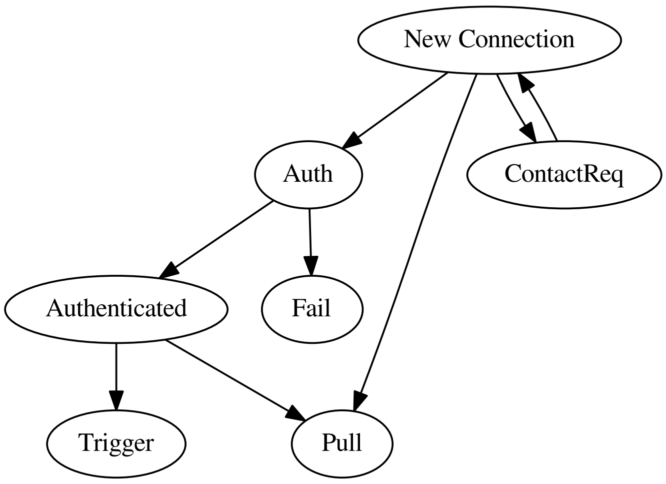


Figure 2: Architecture Overview

Each state is entered upon receiving one more more network messages, see 3.7 for a detailed description of each message. **??** shows how the different components work together in typical use cases like posting comments.

3.4 Authentication

Some commands require (or behave different after) successful authentication of the client. Authentication is done via a encryption based public key challenge response protocol [6].

The client requests authentication by sending an `Auth` package to the server containing the client's `DER`-encoded public key.

The server calculates the onion address of the public key from the `Auth` package and checks if a verified [2] contact with this onion address exists in the database. If no verified contact is found, the connection is closed.

If a verified contact is found, the server generates a 256 bit long random number r and sends a `Challenge` message to the client. The challenge consists of the following three parts:

$H(r)$ The `SHA-256` hash of r

[2] A verified contact has a status of `following`, `pending` or `success`, but *not* `open` or `blocked`

PK_s	The DER-encoded PublicKey of the server
Enc_{PK_c}	A message which is encrypted using RSA-OAEP mode with SHA-224 as the hash function.

The encrypted message contains the following two fields:

r	The 256 bit long random number r
$H(PK_s)$	The SHA-256 hash of the public key of the server [3]

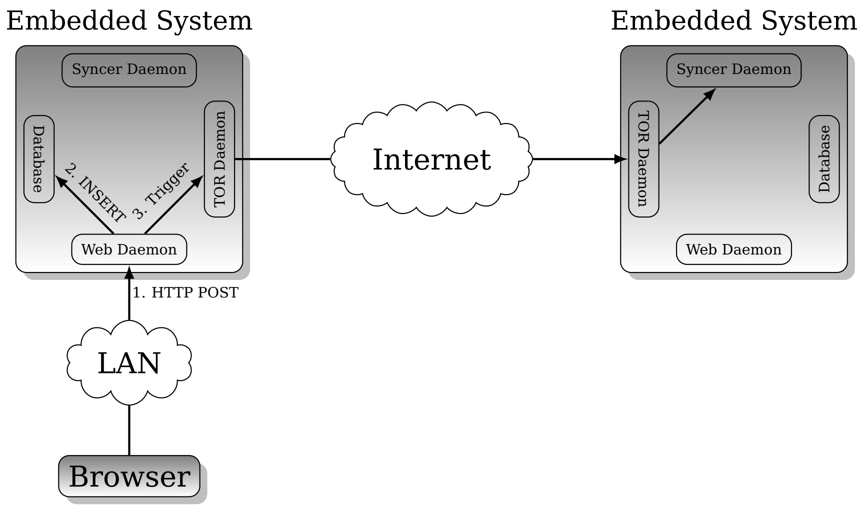
The client decrypts the ciphertext and makes a number of checks before sending a response:

1. the onion address of PK_s is the same to which the client connected in the first place.
2. Hashing PK_s from the plaintext gives the same hash as the $H(PK_s)$ from the decrypted ciphertext.
3. Hashing r from the ciphertext gives the same $H(r)$ as from the plaintext.

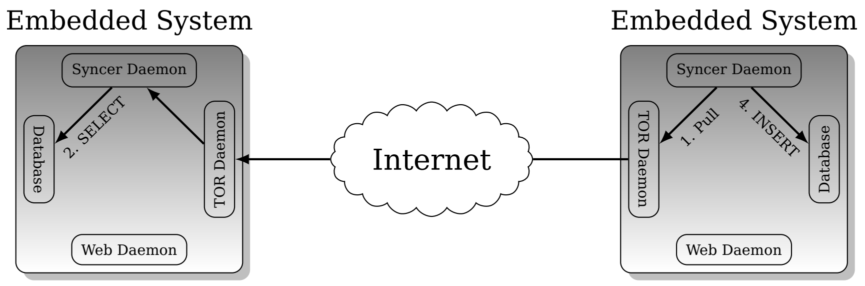
If any of those checks fails the client considers the authentication to have failed and will not produce a response.

Note that $H(R)$ is used as a witness so that the client can verify that the server knows the answer to the challenge. Otherwise it could be possible for the server to abuse the client as a decryption oracle. $H(PK_s)$ in the encrypted message on the other hand serves as a protection so that the server cannot forward a challenge from another system to the client.

The response only contains r'. The server checks that $r = r'$ and sends a `Success` packet to the client. If $r \neq r'$, the authentication failed and the server closes the connection.



(a) Server side post publication



(b) Client side post transmission

Figure 3: Post publication

3.5 Tasks of the individual components involved in the main use cases

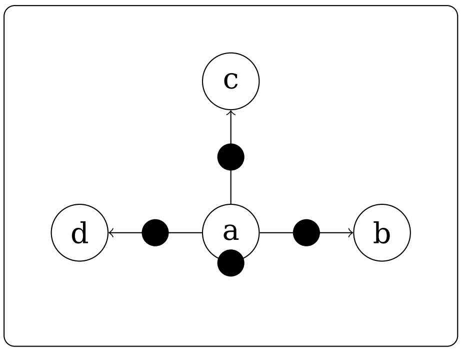
3.5.1 User published a posts

When the user posts a message the web browser sends a HTTP POST request to the REST-API. The REST-API server inserts the post into the database and sends trigger messages to all contacts that are part of the circles the post is publihed in (see figure 3 (a)). A client who is triggered scans the database for the latest timestemp (`remote_published_at` for posts and `changed_at` for profiles) of a post or profile from the contact that send the trigger message. Afterwards the client sends a pull request with the determined timestemp to the syncer daemon who send the trigger message. The syncer daemon who

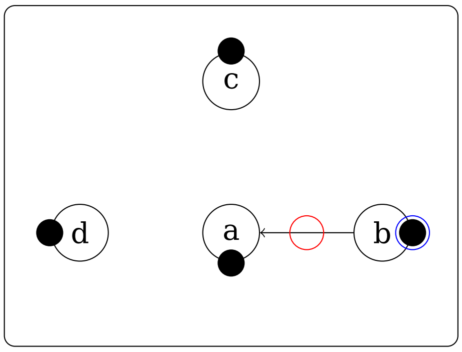
[3]The Handbook of Applied Cryptography sends the full public key in the ciphertext instead of it's hash. We only send to hash because of size limitations of `RSA-OAEP`; we only need to encrypt one block.

received the pull request scans its database for posts with a `published_at` timestamp greater then the timestemp received with the pull request and sends all posts meeting that condition to the client. The client saves all received posts in its database. For eachs post, the received `published_at` timestamp is locally saved as the `remote_published_at` timestamp (see figure 3 (b)).

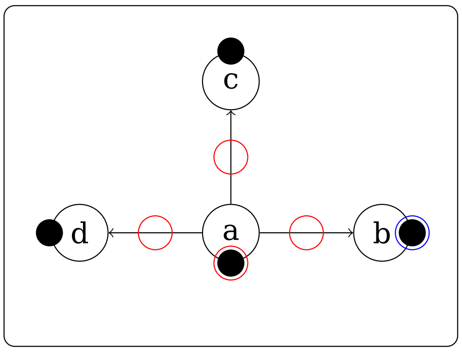
3.5.2 User comments on a post



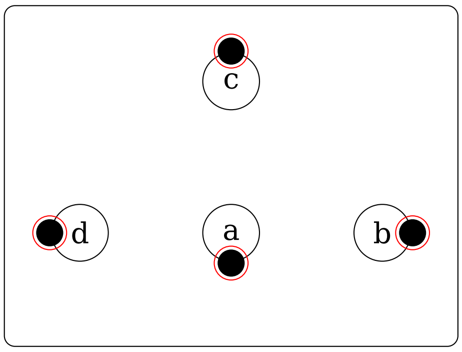
(a) Node a publishes a post to nodes b, c and d



(b) Node b published a comment on the post. The comment is pulled by node a. Node b marks the comment as pending since it has not been republished by node a



(c) Node a publishes the comment to nodes b, c and d



(d) As node b receives the comment from a, it clears the pending state

Figure 4: Comment publication

If a user comments on a post, the comment is send back to the originator of the post. The message format and database table for posts and comments is the same. A comment is a post with the `parent_id` field set to the id of the post the comment refers to. On different systems the post ids differ. To refer to the same post on different machines a SHA-256 hash is used. The hash is computed based on the timestemp, the onion address of the auhor and the content of the post. After receiving the comment, the post originator distributes the comment to all contacts the post was send to, if the comments TTL field is greater than zero (see figure 4).

3.5.3 Contact requests

If a user adds a new contact, a contact request packet is send to the contacts onion. The new cotact inserts the requesting contact into its database with its status field set to `OPEN`. If the requested user accepts the contact request, the REST-API server sends a trigger packet to the requesting syncer. Afterwards the triggered syncer tries to authenticate and pull from the triggering syncer. If the authentication is successfull the client syncer updates the contacts state to `SUCCESS`.

3.5.4 Profile information

Profile information are, like posts, send in reply to a pull request. If the pulling syncer receives the first profile key-value pair it deletes all profile information in the database from the onion it pulls from, so that fields that are outdated are deleted. Afterwards the received profiles are inserted into the database.

3.6 Message Format

Messages are transmitted over a `TCP` connection established by the `TOR` daemon. The messages are encoded using `JSON` [2] notation. `JSON` only specifies single messages, we therefore use a simple, binary message format specifying type and length of messages to allow sending multiple `JSON` messages over a single `TCP` connection.

bytes	field
1	message type
4	message length n in bytes
n	json encoded message

3.7 Individual Message Formats

Auth

message type:	$0x41$ ('**A**')
description:	`Auth` messages are sent by the client to request authentication. They initiate a challenge response authentication
encoding:	`JSON` **data**

`JSON`:

type	description
binary	`DER` encoding of the client's public key

Challenge

message type	$0x43$ ('**C**')
description	`Challenge` messages are sent by the server after receiving an `Auth` messages of a verified contact.
encoding	`JSON` **struct**

The `JSON` encoding is a **struct** with the following fields:

name	type	description
HR	binary	hash of the server's public key
PubKey	binary	`DER`-encoded server's public key
Enc	binary	ciphertext (see subsection 3.4)

Response

message type	$0x52$ ('**R**')
description	`Response` messages are sent by the client in response to a a `Challenge` message.
encoding	`JSON` **data**

`JSON`:

type	description
binary	decrypted 256 bit long random number r

Success

message type	$0x53$ ('**S**')
description	`Success` messages are sent after successful `Pull`, `Trigger`, `Auth` and `ContactRequest` messages.
encoding	none (no payload)

Trigger

message type	$0x54$ ('**T**')
description	`Trigger` messages are sent to a server to indicate that a subsequent pull of the server syncer to the client syncer will yield new information.
encoding	none (no payload)

Pull

message type	$0x50$ ('**P**')
description	A syncer can request all posts, comments and profile updates he may see with a `Pull` message. Without prior authentication, only `Public` information will be sent to the client. With prior authentication `Public` information *and* information, which is accesable to any `circle` of the contact, will be sent.
encoding	`JSON` **data**

`JSON`:

type	description
number	timestamp (only data newer than this timestamp will be sent by the remote syncer)

PushPost

message type	$0x51$ ('**Q**')
description	`PostPost` messages contain a post or comment, they are sent in response to a `Pull` request.
encoding	`JSON` **struct**

The `JSON` encoding is a **struct** with the following fields:

name	type	description
Message	text	Markdown text of the post or comment.
PostedAt	number	UTC Unix-Timestamp when the post or comment was created.
PublishedAt	number	UTC Unix-Timestamp when the
TTL	number	Time-To-Live of the post or comment, may not be forwarded
Author	text	The author's onion address of the post or comment
Hash	text	H(Message, PostedAt, Author's Onion) (`SHA-256`)
ParentHash	text	The empty string for a post; the hash (see above) of the post to which the comment refers for comments

PushProfile

message type	$0x55$ ('**U**')
description	`PushProfile` messages contain a single profile entry, they are sent in response to a `Pull` request.
encoding	`JSON` **struct**

The `JSON` encoding is a **struct** with the following fields:

name	type	description
Key	text	The name of the profile entry.
Value	text	The value of the profile entry.
ChangedAt	number	UTC Unix-Timestamp when the entry was created or changed.

ContactRequest

message type	$0x42$ ('**B**')
description	`ContactRequest` messages cause the server to create a new contact for the client with status `Open`.
encoding	`JSON` **struct**

The `JSON` encoding is a **struct** with the following fields:

name	type	description
Message	text	The contact-request message which will be shown to the user of the server.
Onion	text	The onion address of the client.

4 Database

This section describes the most important database tables. Tables that are only needed to represent relations are omitted here. All database tables described in this section have an id field which is not shown here.

4.1 Onion

An onion represents a tor hidden service onion address.

Column Name	Type	Description
Onion	text	Tor Onion Address

4.2 Contact

A contact extends the onion address with personal information. It has the following columns.

Column Name	Type	Description
Nickname	text	The Nickname of the contact
Alias	text	The name that the local user gave this contact
Trust	int	Describes how sure the local user is that this onion belongs to the person it pretends to
Status	int	Describes the relationship to this contact.
RequestMessage	text	Message used for contact requests

The status field contains one of the following values.

Value	Status	Description
0x00	BLOCKED	Contact is blocked and will not be triggered or pulled
0x01	OPEN	A contact request was received from this contact but not yet accepted
0x02	PENDING	A contact request was send to this contact but not yet accepted
0x03	SUCCESS	Bidirectional relationship
0x04	FOLLOWING	Unidirectional relationship. The syncer will periodically pull posts from this contact

4.3 Post

Column Name	Type	Description
Message	text	Content in Markdown format
created_at, update_at	time	Used by Database wrapper
deleted_at	time	If not '0001-01-01 00:00:00' Post was deleted by the user at the given time
t_t_l	int	Time to live. Decremented at every transmission.
published	bool	if false, post will not be transmitte to anyone
posted_at	time	Time when the post was written
published_at	time	Time when the post was published
remote_published_at	time	Time when the originator published this post
hash	text	base64 encoded SHA-256 hash of message, timestamp and author

A post has one originator and one author. The author is the person (onion) who wrote the post. The originator is the onion address the post was received from. A post is published to an arbitary amount of circles. If a post has a parent, the post is a comment on the parent post.

4.4 Profile

The users profiles consists of an arbitrary number of key-value entries. The entries are assigned to users by the `onion_id` field. Each entry of the local user is assigned to an arbitrary number of circles.

Column Name	Type	Description
key	text	Profile key
value	text	Profile value
deleted_at	time	If not '0001-01-01 00:00:00' Profile was deleted by the user at the given time
changed_at	time	Time the profile was created/modified
onion_id	int	Onion Id of the profiles owner

There are two keys that need to be handled differently by the user interface. A profile entry with the key `picture` contains the base64 encoded profile picture of the user the entry belongs to. Profile entries with an empty key are used by the syncer to signal the deletion of an entry to other syncers and should not be displayed.

4.5 Circle

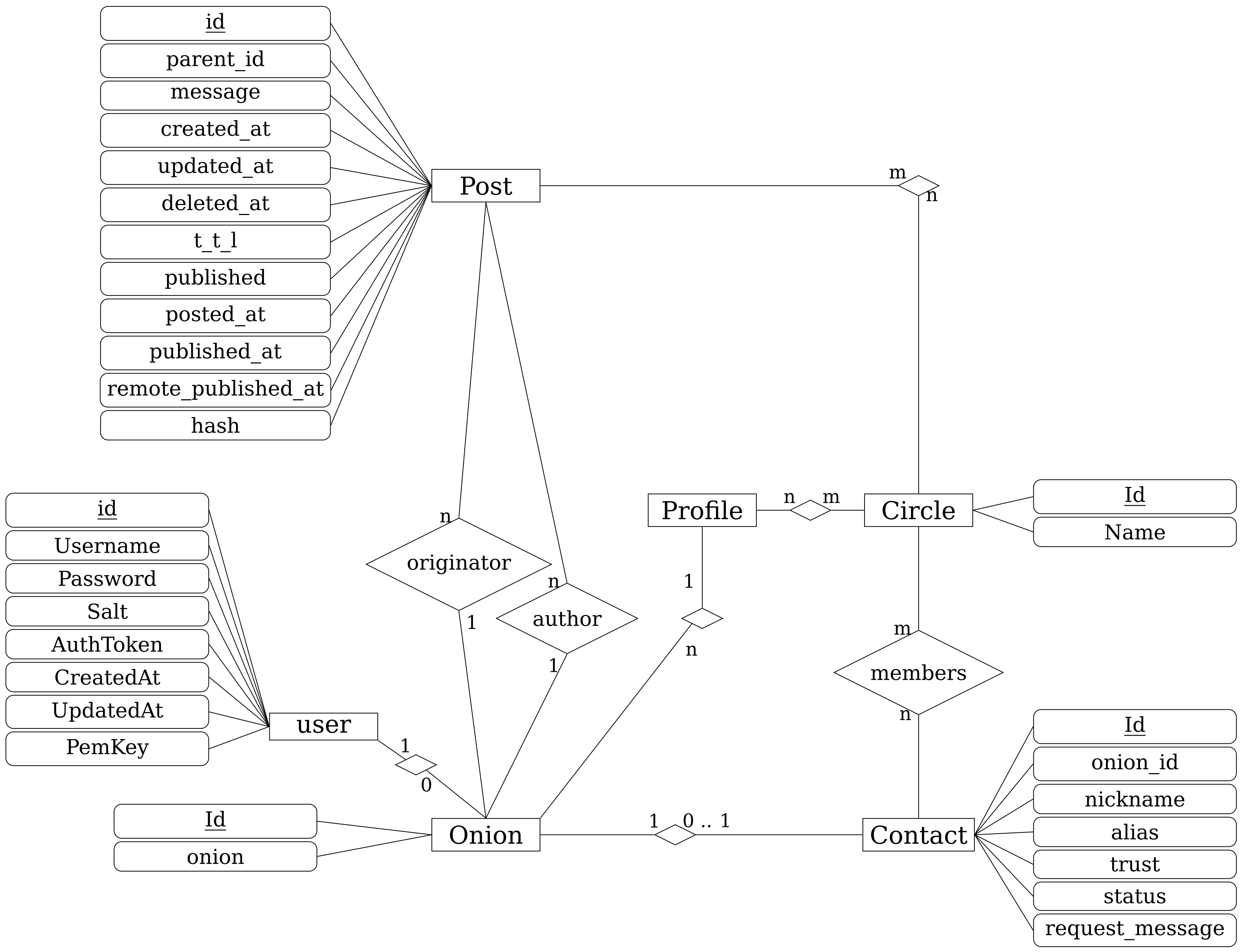
Circles allow the user to group contacts. An arbitary number of contacts can be assigned to a circle. A contact can be assigned to multiple circles.

Column Name	Type	Description
Name	text	Circle name
Creator	int	Is the circle user(1) or automatically(0) created

4.6 User

The user table contains information about the local user. Generally there is only one entry in this table.

Column Name	Type	Description
Username	text	Login name
Password	text	Password hash
Salt	text	Password salt
AuthToken	text	HTTP Request authentication token
CreatedAt	time	Time the user was created
UpdatedAt	time	Time the table was updated
OnionId	int	The users onion address
PemKey	text	PEM encoded Hidden-Service private key


id
parent_id
message
created_at
updated_at
deleted_at
t_t_l
published
posted_at
published_at
remote_published_at
hash
Post
m
n
Profile
n
m
Circle
Id
Name
originator
n
1
author
n
1
1
n
members
m
n
id
Username
Password
Salt
AuthToken
CreatedAt
UpdatedAt
PemKey
user
1
0
Id
onion
Onion
1
0 .. 1
Contact
Id
onion_id
nickname
alias
trust
status
request_message

5 Frontend

5.1 Overview

The frontend consists of a JavaScript framework which is connected to and served by a REST API written in Go (golang). Syncer and API communicate using the database.

5.2 Ember

Ember.js is an open-source client-side JavaScript model-view-controller framework for web applications. It is delivered to the client as a single-page application by the HTTP-Server. Within the application routes are used to maintain applications state and to preload data. They will call the specified controller which is responsible for loading and generating the application view using HTML templates as well as handling actions and user input. Objects are stored in an internal data store which can be persisted using different adapters including the HTML5 local storage and REST.

5.3 REST API

The REST API, written in Go (golang), is used to deliver Ember to the client and persist Embers internal store. REST (Representational State Transfer) is a programming paradigm which defines access to a resources using the CRUD (Create Read Update Delete) policy. Ressources are encoded as JSON documents and send via the HTTP protocol. REST defines ressource access as follows:

Method	URL	Action
GET	/ressource	Returns all entities of this ressource
GET	/ressource/1	Returns the entity with the ID 1
POST	/ressource	Creates an entity of this ressource from the data in the request body
PUT	/ressource/1	Updates the entity with ID 1 with the data in the request body
DELETE	/ressource/1	Destroy the entity with ID 1

5.4 Architecture Decisions

The architecture of the service is designed to be as efficient as possible to be able to run on a Raspberry Pi. Using a client-side JavaScript framework seemed to be the best solution because the heavy-lifting is outsourced to the client. Most common web frameworks are built on either interpreted languages or require a runtime, which is too much overhead to be efficiently hosted on a Raspberry Pi. The same consideration can be applied to the REST API. From the field of compiled languages we chose Go (golang) because it is fast and lightweight language, with a rich standard library and fast to develop. We choose Sqlite for the database because of our small dataset size requirements and the least overhead. NoSQL solutions are not appropriate because of our relation-heavy dataset.

5.5 Events

In order to push new content to contacts in ”realtime“ the creation of posts, comments and profile informations will send trigger packets to all affected contacts. The emission of the trigger packets is done by the REST-API server.

5.6 Security

Even though the security is not in the focus of the frontend, basic measures are in place to deny unauthorized access. As described in the API documentation requests are authorized using an authorization token provide by the API after a successfull login. This token is stored as a cookie.

Loading the application in an iFrame from a different server should not expose any cookies. Using a second Tor hidden service the UI can easily made accessible from outside the LAN.

5.7 Building a different UI

Using the following API documentation the UI can be changed.

6 API Documentation

TODO We tried to be as RESTful as possible but added some non restful routes and fields to requests to remove complexity from Ember

6.1 Status/Error Codes

Status Code	Description
200	Successfull
400	Bad Request, is return if the request is missing data or has missmatching data types
401	Unauthorized, is return if Auth-User or Auth-Token header is not set or invalid
404	Not found, is return if a requested resource can not be found or updated
500	Internal Error, Something unexpected happened (for example database errors)

6.2 User - Create

Request	Request data	Response	Description
POST /users	Listing 1	HTTP-Status: 200	Creates a user with username and password

Listing 1: Request: Create a user

```
{"username": String, "password": String}
```

6.3 User - Authorize

Request	Request data	Response	Description
POST /authorize	Listing 2	Listing 3	Authorizes a user with username and password

After passing the correct login information to the API a authorization token is generated and added to the API response. To be able to access and modify data each request must include both following headers.

Header	Value
auth_user	username
auth_token	auth token from the API response

Listing 2: Request: Authorize a user

```
{"username": String, "password": String}
```

Listing 3: Response: Authorize a user

```
{"auth_token": String}
```

6.4 Post

Listing 4: Post Object

```
{
    "id": Integer,
    "message": String,
    "createdAt": Time,
    "updatedAt": Time,
    "deletedAt": Time,
    "postedAt": Time,
    "ttl": Integer,
    "originator": Integer,
    "author": Integer,
    "circles": []Integer,
}
```

6.5 Post - GetAll

Request	Request data	Response	Description
GET /posts	empty	Listing 5	Returns an array of all posts

Listing 5: Response: Get all posts

```
{
    "posts": []Post
}
```

6.6 Post - Get

Request	Request data	Response	Description
GET /posts/:id	empty	Listing 6	Returns the post with given id

Listing 6: Response: Get post with given id

```
{
    "post": Post
}
```

6.7 Post - Create

Request	Request data	Response	Description
POST /posts	Listing 7	Listing 8	Create a post from POST data, updated object is returned, syncer emits event to notify all contacts in selected circles

Listing 7: Request: Create a post

```
{
    "post": {
        "message": String,
        "createAt": Time,
        "postedAt": Time,
        "updatedAt": Time,
        "ttl": Integer,
        "author": String,
        "originator": String,
        "circles": []String,
    }
}
```

Listing 8: Response: Create a post

```
{
    "post": {
        "id": Integer,
        "message": String,
        "createdAt": Time,
        "updatedAt": Time,
        "deletedAt": Time,
        "postedAt": Time,
        "ttl": Integer,
        "originator": String,
        "author": String,
        "circles": []String,
        "comments": []String,
    }
}
```

6.8 Post - Delete

Request	Request data	Response	Description
DELETE /posts/:id	empty	HTTP-Status: 200	Deletes the post with given id

6.9 Comment

Listing 9: Comment Object

```
{
    "id": Integer,
    "message": String,
    "createdAt": Time,
    "updatedAt": Time,
    "deletedAt": Time,
    "postedAt": Time,
    "isRemotePublished": Boolean, //TODO: explain function
    "ttl": Integer,
    "originator": Integer,
    "author": Integer,
    "post": Integer,
}
```

6.10 Comment - GetAll

Request	Request data	Response	Description
GET /comments	empty	Listing 10	Returns an array of all comments

Listing 10: Response: Get all comments

```
{
    "comments": []Comment
}
```

6.11 Comment - Get

Request	Request data	Response	Description
GET /comments/:id	empty	Listing 11	Returns the comment with given id

Listing 11: Response: Get comment with given id

```
{
    "comment": Comment
}
```

6.12 Comment - Create

Request	Request data	Response	Description
POST /comments	Listing 12	Listing 13	Create a comment from POST data, updated object is returned, syncer emits event to notify all contacts in selected circles. TODO: update to correct description

Listing 12: Request: Create a comment

```
{
    "comment":{
        "message": String,
        "createdAt": Time,
        "postedAt": Time,
        "updatedAt": Time,
        "ttl": Integer,
        "isRemotePublished": Boolean, //TODO
        "author": String,
        "originator": String,
        "post": String
    }
}
```

Listing 13: Response: Create a comment

```
{
    "comment": {
        "id": Integer,
        "message": String,
        "createdAt": Time,
        "updatedAt": Time,
        "deletedAt": Time,
        "postedAt": Time,
        "isRemotePublished": true, // TODO: explain comment
        "ttl": Integer,
        "originator": Integer,
        "author": Integer,
        "post": Integer,
    }
}
```

6.13 Comment - Delete

Request	Request data	Response	Description
DELETE /comments/:id	empty	HTTP-Status: 200	Deletes the comment with given id

6.14 Onion

Alias routes: authors, originators

Listing 14: Onion Object

```
{
    "id": Integer,
    "onion": String,
    "contact": Integer,
    "profiles": []Integer,
}
```

6.15 Onion - GetAll

Request	Request data	Response	Description
GET /onions	empty	Listing 15	Returns an array of all onions
GET /authors	empty	Listing 15	Returns an array of all onions
GET /originators	empty	Listing 15	Returns an array of all onions

Listing 15: Response: Get all onions

```
{
    "onions": []Onion
}
```

6.16 Comment - Get

Request	Request data	Response	Description
GET /onions/:id	empty	Listing 16	Returns the onion with given id
GET /authors/:id	empty	Listing 16	Returns the onion with given id
GET /originators/:id	empty	Listing 16	Returns the onion with given id

Listing 16: Response: Get onion with given id

```
{
    "onion": Onion
}
```

6.17 Circle

Listing 17: Circle Object

```
{
    "id": Integer,
    "name": String,
    "creator": Integer, //  app (0)  user (1)
    "contacts": []Integer,
    "posts": []Integer,
}
```

6.18 Circle - GetAll

Request	Request data	Response	Description
GET /circles	empty	Listing 18	Returns an array of all circles

Listing 18: Response: Get all circles

```
{
    "circles": []Circle
}
```

6.19 Circle - Get

Request	Request data	Response	Description
GET /circles/:id	empty	Listing 19	Returns the circle with given id

Listing 19: Response: Get circle with given id

```
{
    "circle": Circle
}
```

6.20 Circle - Create

Request	Request data	Response	Description
POST /circles	Listing 20	Listing 21	Create a circle from POST data contacts in selected circles

Listing 20: Request: Create a circle

```
{
    "circle":{
        "name": String,
    }
}
```

Listing 21: Response: Create a circle

```
{
    "circle": {
        "id": Integer,
        "name": String,
        "creator": Integer, // app (0) user (1)
        "contacts": []String,
        "posts": []String,
    }
}
```

6.21 Circle - Delete

Request	Request data	Response	Description
DELETE /circles/:id	empty	HTTP-Status: 200	Deletes the circle with given id

6.22 Contact

Listing 22: Contact Object

```
{
    "id": Integer,
    "onion": Integer,
    "nickname": String,
    "alias": String,
    "trust": Integer,
    "status": Integer,
    "request_message": String,
    "circles": []Integer,
},
```

6.23 Contact - GetAll

Request	Request data	Response	Description
GET /contacts	empty	Listing 23	Returns an array of all contacts

Listing 23: Response: Get all contacts

```
{
    "contacts": []Contact
}
```

6.24 Contact - Get

Request	Request data	Response	Description
GET /contacts/:id	empty	Listing 24	Returns the contact with given id

Listing 24: Response: Get contact with given id

```
{
    "contact": Contact
}
```

6.25 Contact - Create

Request	Request data	Response	Description
POST /contacts	Listing 25	Listing 26	Create a contact from POST data

Listing 25: Request: Create a contact

```
TODO
```

Listing 26: Response: Create a contact

```
TODO
```

6.26 Contact - Update

Request	Request data	Response	Description
PUT /contacts/:id	Listing 27	HTTP-Status: 200	Update a contact with given id from POST data

Listing 27: Request: Update a contact

```
{
    "contact":{
        "nickname": String,
        "alias": String,
        "trust": Integer,
        "status": Integer,
        "request_message": String,
        "onion": String,
        "circles": []String,
    }
}
```

6.27 Contact - Delete

Request	Request data	Response	Description
DELETE /contacts/:id	empty	HTTP-Status: 200	Deletes the contact with given id

6.28 Profile

Listing 28: Profile Object

```
{
    "profile": [
        {
            "id": Integer,
            "key": String,
            "value": String,
            "onion": Integer,
            "circles": []Integer
        }
    ]
}
```

6.29 Profile - GetAll

Request	Request data	Response	Description
GET /profiles	empty	Listing 29	Returns an array of all profiles

Listing 29: Response: Get all profiles

```
{
    "profiles": []Profile
}
```

6.30 Profile - Create

Request	Request data	Response	Description
POST /profiles	Listing 30	Listing 31	Create a profile from POST data, updated object is returned, syncer emits event to notify all contacts in selected circles. TODO: update to correct description

Listing 30: Request: Create a profile

```
{
    "profile":{
        "key": String,
        "value": String,
        "circles":[]String,
    }
}
```

Listing 31: Response: Create a profile

```
{
    "profile": {
        "id": Integer,
        "key": String,
        "value": String,
        "onion": Integer,
        "circles": []Integer,
    }
}
```

6.31 Profile - Update

Request	Request data	Response	Description
PUT /profiles/:id	Listing 32	HTTP-Status: 200	Update a profile with given id from POST data

Listing 32: Request: Update a profile

```
{
    "profile":{
        "key": String,
        "value": String,
        "circles": []String,
    }
}
```

6.32 Profile - Delete

Request	Request data	Response	Description
DELETE /profiles/:id	empty	HTTP-Status: 200	Deletes the profile with given id

7 Security Considerations

7.1 Reliance on Tor

The Zwiebelnetz heavily relies on Tor's security. The systems build a friend-to-friend network among contacts, making it easy to extract the social graph from connection data. If Tor traffic can be de-anonymized, the encrypted traffic could potentially reveal private information such as:

- Who is a contact of whom, since communication only happens between contacts.
- When does someone post (or update the profile): Very short message sent to contacts is likely a "Trigger" packet.

- How long is a Post, or how many Profile updates were written: TCP Connections are closed right after a Pull request is served. The more data there is to send, the more encrypted traffic will pass the network.
- Who do you post to or publish profiles entries to: The Trigger messages are only sent to valid recipients if a Post is written or a profile entry is updated.

Many more such information could potentially be extracted if Tor's anonymization was broken.

We also rely on Tor's promise to protect data's authenticity, integrity and privacy for data sent to hidden services and the one-way authentication. Tor uses RSA keys with a length of 1024 bit to secure the initial diffie-hellman key exchange upon new connections. RSA-1024 is no longer conisidered secure nowadays.

Also, we rely on the uniqueness of onion addresses. We do not store other system's public keys - we only store the onion address. During authentication (see subsection 3.4), the requested system sends it's public key. We only check that the onion of the public key indeed matches the onion we stored. If it was possible to generate a public key with yields the same onion address, attacker could have us connect and authenticate to his system, allowing to send us posts not written by the contact we think they were. He would also be able to authenticate to us, since the syncer server, as well, only checks if the key passed with the `Auth` package matches the onion address of a contact. This would allow the attacker to obtain any information the impersonated contact could read. Onion addresses in Tor are the *first 80 bit* only of the 160 bit `SHA1`-hash. In general, the security parameters seem to be set rather low with Tor. There are proposals in the Tor project to switch to longer onion addresses and use elliptic curve public key cryptography. Most of our code is compatible with such changes, so that a transition to different onion length or different public key cryptography schemes would affect few components.

Two ideas to improve on security problems caused by Tor are:

- Do not reply on Tor's transport security, tunnel through TLS or similiar. This would require to use some concatenation of onion address and some TLS certificate fingerprint as contact information.
- Store public keys of contacts. If an attacker generates a key pair with

the same onion address in the future, it will not match the stored public key and be rejected.

7.2 Markdown

We support markdown for Posts and Comments. This enables the creation of well formated postings instead of just text messages. The markdown library consists is a Javascript based parser which converts the posts into html in the browser of the viewing user. Security bugs in the markdown library would allow only Contacts to attack the viewing client. While not as critical as attacks which *any* user, not only contacts, could run, an the (unaudited) markdown parser should be disabled if of concern.

One issue we encountered is the inline rendering of external images. It allows malicious contacts to link to images on own web servers with unique urls and later analyze the server logs. This could be used to identify IP addresses of users which keep the attackers as contact of follow him.

This affects the comment functionality. An attacker use this to read the IP addresses of the contacts and followers of a *contact* of his.

An easy way to protect against this attack is to enable Tor in the browser which accesses the Zwiebelnetz, the server logs will only show anonymous connections via the Tor network. The number of connections, however, will not be hidden this way.

7.3 Unaudited Code Base

Our code-base consists of more than five thousand lines of `golang` and several web languages. Only some parts of the code were read by another team member after being written. It is therefore likely, that the code-base contains unknown security issues.

While the syncer and restful API mostly use the standard library of `golang`, the frontend uses a large web framework and several web components. While all of those are commonly used on the Internet, it is not clear to us how much their code can be trusted.

7.4 Frontend Connectivity

The frontend connects to the JSON-API using plain HTTP, not HTTPS. It is therefore suggested to manually set up a second Tor hidden service which is used to connect firstly in a secure manner (see subsection 7.1, though) and secondly from a remote location, outside the local area network, which, in our case, usually is a home environment.

Authentication is done via salted hashes using `SHA-256` as a hash function. In the plain HTTP case, this means that passwords are transmitted as plaintext. While plaintext passwords are only transmitted in the LAN environment, this is still undesirable, the user could connect to his system using shared wireless connection. Again, use a second Tor hidden service for frontend connectivity.

The browser stores an access token in a cookie after authentication. This cookie allows the frontend to authenticate to the JSON-API without the user having to enter his password every few seconds. If the user does not protect physical access to his device, an attacker could use the Zwiebelnetz without loggin in, though, as long as a browser with a Zwiebelnetz session is still open.

8 Used Software, Frameworks and Libraries

8.1 BackEnd

The backend was developed in Go [`https://www.golang.org/`], an open source programming language. The following libraries were used.

- Go-Json-Rest [`https://github.com/ant0ine/go-json-rest/`],
  a quick and easy way to setup a RESTful JSON API
- go-sqlite3 [`https://github.com/mattn/go-sqlite3`],
  sqlite3 driver conforming to the built-in database/sql interface.
- GORM [`https://github.com/jinzhu/gorm`],
  object-relational mapping library for Go.

8.2 FrontEnd

The frontend is web-based and can be used with any conventional browser (Firefox, Chrome, Opera etc.). Techniques like HTML and JavaScript/CoffeeScript and the following libraries were used.

- jQuery [`http://jquery.com/`],
  a fast, small, and feature-rich JavaScript library.
- Bootstrap [`http://getbootstrap.com/`],
  popular HTML, CSS, and JS framework.
- Ember.js [`http://emberjs.com/`],
  a framework for creating ambitious web application.
- Emblem.js [`http://emblemjs.com/`],
  a new templating language that compiles to Handlebars.js. (Handlebars provides the power necessary to let you build semantic templates effectively with no frustration.)
- Moment.js [`http://momentjs.com/`],
  Parse, validate, manipulate, and display dates in JavaScript.
- marked [`https://github.com/chjj/marked`],
  a full-featured markdown parser and compiler.
- Select2 [`http://ivaynberg.github.io/select2/`],
  is a jQuery based replacement for select boxes.

References

[1] Sqlite. `http://www.sqlite.org/`. accessed 2014-09-15.

[2] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), March 2014.

[3] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[4] Ying-Da Lee. Socks 4a: A simple extension to socks 4 protocol. `http://www.openssh.com/txt/socks4a.protocol`. accessed 2014-09-02.

[5] Ying-Da Lee. Socks: A protocol for tcp proxy across firewalls. `http://www.openssh.com/txt/socks4.protocol`. accessed 2014-09-02.

[6] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*, chapter 10 - Identification and Entity Authentication. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[7] Tor Team. Tor hidden services. `https://www.torproject.org/docs/hidden-services.html.en`. accessed 2014-08-29.

[8] Tox Team. Tox - a new kind of instant messaging. `http://www.tox.im`. accessed 2014-09-15.