TECHNISCHE UNIVERSITÄT DRESDEN

Master Thesis

Federated event propagation in distributed social networks

Marc Löchner

December 9, 2015

Advisor:
Dr. Josef Spillner

Supervisor:
Prof. Dr. rer. nat. habil.
Dr. h. c. Alexander Schill

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Systemarchitektur, Lehrstuhl Rechnernetze

AUFGABENSTELLUNG FÜR DIE MASTERARBEIT

Name, Vorname:	Löchner, Marc		
Studiengang:	Medieninformatik '10	Matrikelnummer:	2951518
Forschungsgebiet:	Service and Cloud Computing	Forschungsprojekt:	DaaMob
Betreuer:	Dr.-Ing. J. Spillner	Externe(r) Betreuer:	---
Verantwortlicher Hochschullehrer:		Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill	
Zweitgutachter:		Dr.-Ing. Daniel Schuster	
Beginn am:	01.05.2015	Einzureichen am:	08.10.2015
Thema:	Federated Event Propagation in Distributed Social Networks		

ZIELSTELLUNG

Distributed and in particular federated infrastructures are increasingly common on the Internet. As opposed to centralised systems, they pose challenges concerning the shared access to data and information. Federated social networks are widely affected due to the high dynamics in content production or modification and churn among the participants.
A representative problem is the propagation of event invitations among the participants of instances (pods) of the social network software Diaspora. This is a major obstacle for potential users compared to having such functionality integrated in centralised social networks. In this master thesis, the RSVP event invitation mechanism shall be analysed and implemented for Diaspora. The associated challenges and solution approaches shall be generalised for other kinds of user-created content in federated systems.

SCHWERPUNKTE

- Analysis of current distributed event invitation protocols
- Concept for a network-integrated RSVP functionality
- Implementation of RSVP in Diaspora
- Elaboration on the generalisation to other content and other networks

----

Unterschrift des verantwortlichen Hochschullehrers
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Declaration of Authorship

I herewith declare that I have produced this thesis without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This thesis has not previously been presented in identical or similar form to any other German or foreign examination board.

The thesis work was conducted from May 1, 2015 to December 9, 2015, under the supervision of Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill at the Department of Computer Networks, Technische Universität Dresden.

Marc Löchner

Abstract

Currently common social network services lack a reasonable amount of privacy due to their centralization. Distributed social networks are considered an alternative to these services. However, due to their limited feature set, current implementations struggle to keep up with their mostly commercial competitors. Limitations derive from scarce financial resources as well as challenges due to the decentralized network topology and consequential complexity to propagate user generated data.

A representative problem is the propagation of social events within a distributed social network. While commercial services provide this feature today, currently available software to establish distributed social networks have not been able to advance. Deploying social event functionality to them is challenging, as its rather complex data structure meets its federation within a distributed network.

This thesis introduces a concept to establish social event functionality in distributed social networks, including its relations to multiple users as well as basic administrative aspects. Furthermore, it provides a proof-of-concept implementation using the Ruby on Rails application Diaspora as a representative software for distributed social networks. The solution is evaluated in terms of usability, privacy and performance, summed up with a portrait of ambivalent participation experiences in the free and open source software developer community.

Contents

Chapter 1

Introduction

Common social network services like Facebook or Twitter run their services from a number of data centers around the world, controlled by either themselves or other commercial providers. The data stored in those data centers are either synchronized or loaded from their source when needed. Still, they are stored on hardware controlled by the provider (Menon, 2012).

One can consider this as a disadvantage regarding privacy aspects of their users. Those services are usually run by commercial enterprises, whose primary goal is to gain profit. As a trade-off for providing their services to their users free of charge, the companies analyze the data supplied by the users, their access patterns and general activities to categorize them and generate substantial digital profile images of them. Additional to the gaining of explicit knowledge about a person, more implicit knowledge can be inferred via the person's contacts. It is possible to precisely determine habits and needs not only of a single person, but of the entirety of the service's user base (i.e. Abel et al., 2011).

The possession of that knowledge makes those companies very powerful. On the one hand, they can advertise third-party products to very specific target groups, on the other hand, they can determine behavioral patterns among the society (Anderson, 2008) and even to predict future behavior (Rogers, 2008). This is very problematic for a social coexistence, as it provides large potential to be abused, mainly to gain even more knowledge of the user base, undermine potential competitors, and, thus, develop an awkward concentration of power.

As the Snowden revelations unveiled, not only commercial enterprises are seeking for deep knowledge of the habits and needs of the world's population, but governments as well (Landau, 2013). A government supplied with such deep knowledge of their people has the power to investigate and control social structures. It may lead to targeted spreading of manipulated information, with the intent to provoke fear, uncertainty and doubt among the people. It may suppress minorities or protest movements in early stages and makes executives gain ultimate control over the society (Chambliss, 1995). In theory, the opposite should be the case.

Pointing out the fact that storing personal information of a large user base in a centralized structure is very dangerous, it is obvious that decentralized data structures can help to preserve informational self-determination[1] of a single user as well as to keep basic principles of democracy in place (Kayes and Iamnitchi, 2015).

Being decentralized, free and open, e-mail has been the primary communication platform for about two decades. The rise of smartphones as the primary communication medium has unveiled its downsides and let other technologies like social networks succeed. Unfortunately, during the last decade, centralized services have gained more attention due to financial backing of venture capital investment (Dam, Nelson, and Lozinski, 2008) and the network effect (Varian and Shapiro, 1999).

To maintain informational self-determination for individuals and protect the society from overwhelming governmental or commercial power, it is crucial to advance development of free and open source distributed social networks (DSNs) to be a desirable alternative to centralized services.

1.1 Presumption

A representative of free and open source DSN platforms is Diaspora. Not being owned by anyone in particular, Diaspora consists of a number of independent instances called *pods*, which interact with each other to become a social network (see figure 1.1).

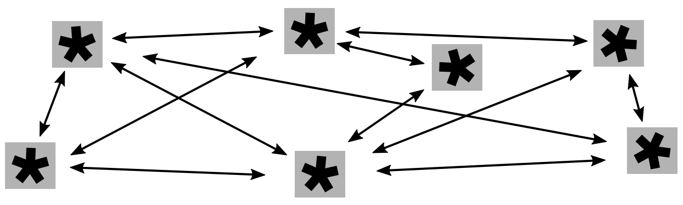


Figure 1.1: Independent Diaspora instances form a DSN

The fundamental characteristics of a DSN is the decentralization of hosted data. Unlike centralized services, in DSNs like Diaspora the user-specific data is stored only on the instance, a user has signed on to. Users as well as administrators of other instances can only access data that is offered to them.

[1]Informational self-determinations is the right of the individual to decide what information about himself should be communicated to others and under what circumstances. Westin (1970)

Hosted data must be specified among their dedication. Besides trivial content like the profile page of a user with any directly user-related data, there are more sophisticated data structures that are addressed to be shared with other users.

The most common type of that data is commonly known as *post* or *status update*. More specified data may be shared images, videos or any other type of extended media. Those are usually implemented as attachments to a post and are owned by their dedicated author.

Another common type of data structure, that may be shared between users, is a social event. The characteristics of an event as a sharable data structure may be defined as a complex set of individual data, as there are date and time, a location, a title and description, a poster, entry fees etc. Furthermore, there is a wide range of data addressing relations between events and people, as there are invitations, attendance, and even more special dedications regarding administration issues, as figure 1.2 shows exemplarily.

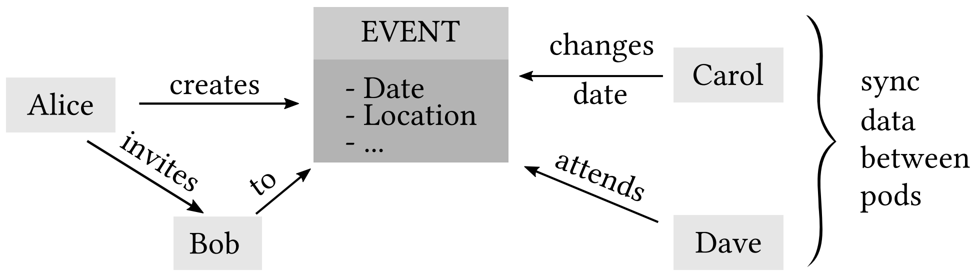


Figure 1.2: Event data structure

While posts are the most common sharable data structure among any of todays major social networks, events are less common and today mostly seen on Facebook, which makes it one of their most frequently used features (Constine, 2015). Regarding privacy concerns, it is desirable to have events become a common data structure in DSNs as well.

Deploying social event functionality to DSNs is challenging, as the complexity of its data structure meets its federation among a distributed network topology. Sophisticated conception to design relations between event data and user data as well as eligible protocols to distribute the data among individual instances within the DSN are required.

This work is dedicated to design a concept to deploy social events to a DSN and develop a proof-of-concept implementation. As a representative for a DSNs, Diaspora is utilized as an example application. It is to be extended with an event feature, that can be deployed and operated between individual instances. Any gained findings should subsequently be generalized for other data structures to be shared in any DSN.

1.2 Outline

Within the remaining chapters of this thesis, the user is guided through the entire progress.

Chapter 2 gives an overview of DSN related work and introduces fundamental basics of modern web software development, as they are required to contribute to Diaspora.

A close software analysis of the Diaspora DSN, its key features and current status will be reviewed in chapter 3.

Based on this analysis, chapter 4 introduces the concept to integrate events into Diaspora in detail. All important aspects including relations and federation as well as alternative and revoked approaches are discussed.

In chapter 5 the implementation of the concept is documented. It focuses on challenging aspects during the development and ties on fundamentals mentioned in chapter 2.

The evaluation of the implementation work is discussed in chapter 6. It examines the functionality of the implemented work, while focusing on features elaborated in chapter 4.

Chapter 7 holds overall conclusions, lessons learned, outlook and a summary of this thesis.

Chapter 2

Fundamentals

As the work on this thesis is also dedicated to expose contribution to free and open source software (FOSS) projects, besides introducing DSN related fundamentals this chapter also discusses elementary FOSS-related aspects: reasons to develop open source software, the workflow on social coding platforms, models for development using version control, and test driven development are covered.

2.1 Distributed social networks

Baran (1964) first introduced a differentiation of the terms *centralized*, *decentralized* and *distributed* regarding computer networks (see figure 2.1).

Peeters (2013) then did some research on the differences between the terms *distributed* and *federated*. He points out that there is not really a clear distinction to make, as different researchers provide different and yet contradictory views about the terms. Within this work, the term *distributed* will be used to describe the network topology, while *federation* will refer to the diaspora feature, that is discussed in section 3.3.

Boyd and Ellison (2010) define social network sites as web-based services that allow individuals to (1) construct a public or semi-public profile within a bounded system, (2) articulate a list of other users with whom they share a connection, and (3) view and traverse their list of connections and those made by others within the system.

Publishing individual content to a defined or undefined subset of members of the network, commenting, replying, direct messaging, like/faving, updating status etc. (Dhekane and Vibber, 2011) should be added to that definition.

DSNs differ from centralized networks in the fact, that the term *network* is not just virtually focused on the relationships between their users, but actually describes the structure of the hosted data.

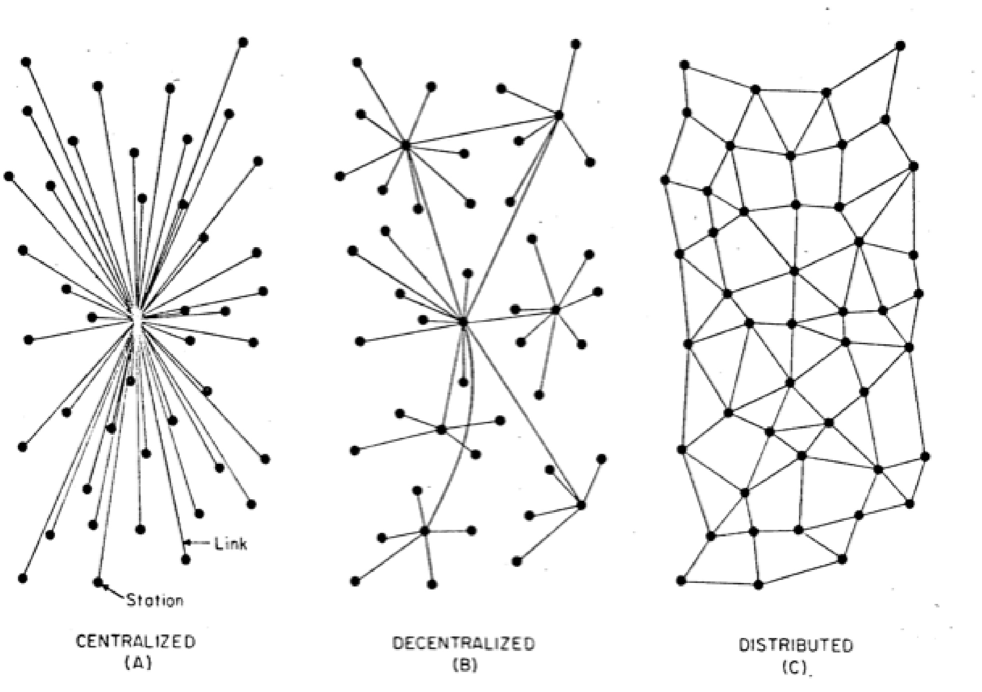


Figure 2.1: Types of networks (Baran, 1964, p. 4), reference via Peeters (2013)

There is an unlimited number of servers running a software that interacts with other instances of the software via given protocols on other servers spread all over the world participating in that network.

Users may connect to other users not just by their usernames but rather like `user@domain`. That form is familiar from e-mail, a similar structured distributed communication network.

Although it is not a crucial feature, distributed social network platforms usually ship as free and open source software (see section 2.3), the protocols used tend to be free and open as well.

2.2 DSN related work

Social network services have become a major tool for communication and collaboration worldwide. The ever growing interest in them and evolving use cases brings new challenges and practical research problems. Massive adoption of mobile and wireless devices, and evolution of peer-to-peer solutions raise attention to partially or fully decentralized social networks. This section draws attention to this progress by introducing recent research results on DSN related subjects.

Maka (2011) used the open source software stack BuddyCloud (2015) to build up a DSN that implements Federation using the Publish-Subscribe feature of the Extensible Messaging and Presence Protocol (XMPP). He stated that the protocol stack is suitable for that task, until it comes to distribution of binary data, where other protocols like the Hypertext Transfer Protocol (HTTP) work better.

Dhekane and Vibber (2011) first introduce a model for friend finding on federated social networks called Talash using the open source software StatusNet (2015). They showed a mechanism how to find contacts on that network through the so-called Friends-of-a-Friend information. Clustering a DSN into communities, each centered around one user, allows expansion of that user's DSN in any other domains.

Liu et al. (2012) define and investigate event-based social networks (EBSNs). Their unique characteristics is stated as the correlation between *online* and *offline* social interactions within the network. As event invitations are made online and the actual attending to an event obviously happens offline in the real world, the members of a certain network may have closer relations as members of other social networks. They also analyzed the information flow depending on the size of the audience to an event invitation as well as the distance between the user's and the event's location, which resulted in various models for attendance prediction.

A recent research project by Konforty et al. (2015) on a DSN is called Synereo. They present social content that is relevant and actionable based on the user's own estimation of value. Furthermore, they discuss the relationship between attention, value, and social agency in order to motivate the central mechanisms for content flow on the network. A network model showing the mechanics of the network interactions, as well as the compensation model enabling users to promote content on the network and receive compensation for attention given to the network, is introduced.

In November 2015, a workshop on DSNs has taken place for the first time in Miami, Florida (DeSN15, 2015) , that is to serve as a forum for researchers or professionals from both academia and industry to exchange new ideas, discuss new solutions, and share their experience in the design, implementation, analysis, experiment or measurement related to decentralized social networks. Papers accepted to this workshop cover gossiping on browsers without a server (Carvajal-Gómez et al., 2015), service discovery for spontaneous communities in pervasive environment (Nejma et al., 2015) and a network and application framework for spontaneous and ephemeral social networks (Boutet et al., 2015).

Besides this related academic work, there is a wide range of DSN software currently in development. A broad comparison list is available on Wikipedia (2015). This is discussed particularly in section 3.1. However, none of the previously published work or released DSN software has discussed or featured federated social events yet. It is the premise of this thesis.

2.3 Free and open source software

Free software enables users to have the freedom to run, copy, distribute, study, change and improve the software (Free Software Foundation, 2001). It refers to the term *freedom* and is often explained by the distinction of sharing the meaning with *free speech* as opposed to *free beer*.

The term *open source* refers to a paradigm, that describes the source code of a software being available to the public for general usage, examination and, depending on the actual licensing model, modification and redistribution. It aims at the collaborative approach, that programmers share the original source code with others, not directly involved with the primary conception of the software, usually referred to as the community. Within the community, any further improvement and creativity that is submitted back to the project will be examined and conceivably merged.

The aspect of openness is very important for the credibility of a software. As Hoepman and Jacobs (2007) already stated, openness of software will increase its security. While the overall quality of the code can be examined, rated and eventually improved by anyone, the development of the software may be accelerated through the wider input. Developers and users can make use of tools to validate the source code. Users are free to judge independently about the security of the software security.

Another important aspect for software sources to be open is the ability to work in a team of independent developers. Dabbish et al. (2012) examined the value of transparency for large-scale distributed collaborations and developing communities in practice. When studying development behavior and results of software projects on Github, they found that the technical skills and reputation of the developers gained through the influence of team mates in a surprisingly high amount.

There are a number of free software licensing models, that explicitly define the rights granted to share and modify a certain software. While classic licenses like the GPL (Free Software Foundation, 2007a) mostly address computer programs in general, other licenses refer to more certain use cases. For example, the AGPL (Free Software Foundation, 2007b) is explicitly designed to cover the usage of a software via a computer network, such as web applications like Diaspora. There are also free licenses for user-generated content, like the CC licenses (Creative Commons, 2002).

Still, software licensing can not enforce privacy. Therefore, technical measures like the use of DSNs have to be taken. Only a combination of DSNs and free software licenses ensure privacy in the long run.

2.4 Distributed software development

When the subsistence of a software is focused on privacy, distributed software development is crucial. As opposed to classic development approaches of centralized software development, in distributed version control systems like Git (2005), there is no canonical reference code repository by design. Repositories may be cloned infinitely across any number of systems. Communication between them and synchronization may be automated or performed manually.

In case of the development to move in a wrong direction, this keeps control over the software code itself for other developers. It also prevents loss of code, in case a repository hosting platform may become unavailable. Concerning privacy, this ensures transparency to the code base, as it can not be compromised and redistributed without any backup or comparison possibility.

Participating in software development with distributed version control requires a certain amount of conduct for developers. The following subsections explain the collaboration work flow on social coding platforms, as well as different concepts of version control development models, and the necessity for test driven development.

Social coding platform development

When software projects are hosted on social coding platforms (Vasilescu, Filkov, and Serebrenik, 2013) like Github, usually a certain collaboration work flow is applied (see figure 2.2).

As not everybody is granted write permissions to any project, a contributor first may *fork* the project to his own profile. Within that fork he may follow any desired development model, but should leave the branches remain the same.

To work on the project, the developer is to *clone* the repository from the *origin* of the forked project into a *local* development environment. Changes are *pushed* back to origin.

Once the contribution is complete, the contributor is to file a *pull-request* in the development branch of the *upstream* project, where the project originally has been forked from. The maintainers of that project then may merge the pull request into the original project.

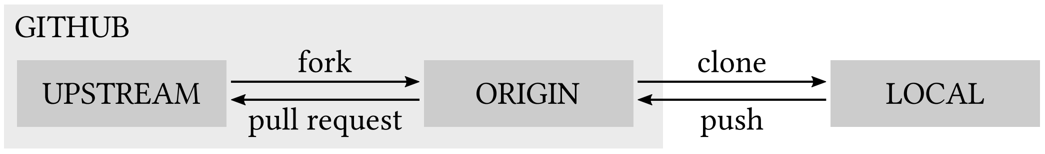


Figure 2.2: Social coding workflow

Version control development models

Hammant (2013) describes a development model called *Trunk based development* as shown in figure 2.3. This model is based on one central branch called the *trunk*. While any developer may use other branches within their local development infrastructure, on the server side all development is made on the trunk.

Once the state of the software is ready for a release, then a branch is created from the trunk and called after the release number. Bug fixes are made on the trunk and then merged into the release branches. Usually developers do not have write permissions to them, only the release engineer does. So this is a quite constrained and hierarchical development model.

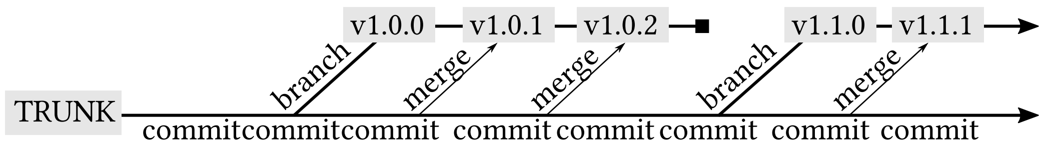


Figure 2.3: Trunk based development

Driessen (2010) describes another development model called *Git Branching* (see figure 2.4), as it is adopted and applied by the Diaspora community (Tilley, 2012a). As one of the main enhancements of Git compared to older version control systems is the ease of branching. Making use of branches is a core part of the development work flow. Within the model there are a number of default branches, that every project should consist of:

master is the current state of the project. Releases are tagged on it. No active development is done on that branch.

develop should be the branch for daily work. New branches are created from here and any nightly builds would be taken from it. This branch is where external developers should issue their pull requests against.

feature branches are created, once a new feature is added to the software. This is shown by naming the branch after the new feature. Once the feature is completed, the branch is merged back into the development branch.

release branches are created from the development branch right before a production release. Last minor bug fixes can be made as well as version numbering adjusted inside the code. It is then merged into master, where it is tagged as the current version number. Finally it is also merged with the develop branch to include any minor fixes into future releases as well.

hotfix finally comes into account, if there is a severe bug in a production release, that needs to be fixed immediately. The branch will be created from the master branch. Besides that it does not differ from a release branch, as it will be merged back into the master and the develop branch.

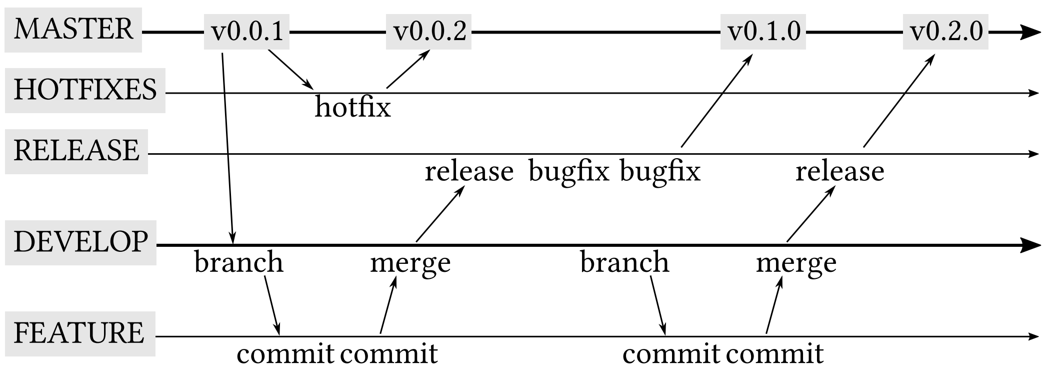


Figure 2.4: Branching model

Test driven development

As Beck (2003) first describes, test-driven development (TDD) is a key method to improve overall code quality of any software project.

Within many software development environments there are testing frameworks available, that run certain test cases created by the developer on the actual software code, like RSpec for Ruby or Jasmine for JavaScript projects. Running tests is a standard procedure in any software build operation.

Following TDD means to actually write a test case, before even implementing the code for the software itself. The idea is to use the test case to describe, what the software code is expected to do. The test case fails, as long as the code does not fulfill the desired functionality. Once the code passes the test, it can be refactored to improve overall code quality, while keeping the test passing. Finally the feature will be deployed, and before the next feature is implemented, another test needs to be written therefore (see figure 2.5).

Within the Diaspora community, TDD is applied within the development process and any contributor is encouraged to utilize it (Tilley, 2012b).

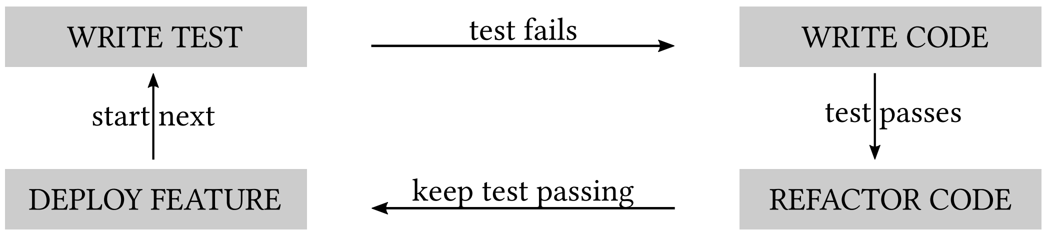


Figure 2.5: Test driven development

Chapter 3

Diaspora DSN Analysis

Diaspora is a distributed social network (DSN) that consists of multiple independent instances. The DSN is not owned by a particular person, enterprise or any other corporation, because each instance is run by individuals. This keeps the network from being subject to advertising, data mining and commercial takeovers. The software which the instances forming the network are run with, is based on a free and open source software that is called Diaspora as well.

The term *Diaspora* refers to a historical expression for movements of people away from their home country. Being of Greek origin, the term is well-known from the expulsion of Jews from Judea, Greeks fleeing away from Constantinople and other historical occasions.

The dispersion away from a bad origin has been adopted as a metaphor for the movement from centralized services to a DSN. Started as a private approach by Zhitomirskiy et al. (2011) it later became a free and open source software community project (Zhitomirskiy et al., 2012). The goal of the project is to address privacy concerns of users by design.

3.1 Comparison to other DSN software

There is a wide range of DSNs being in active development. A broad comparison list is available on Wikipedia (2015). As most of them share major characteristics, as being actively developed, using free and open source software development kits, released under free software licenses as the AGPLv3 license and their maturity is in stable state, the choice for Diaspora as reference software for this thesis has been made by the number of active users in the particular network.

Table 3.1 on page 18 shows a comparison of active users on three of the current major DSNs. Since Diaspora is by far the most actively used DSN, it is utilized to demonstrate the goals of this thesis.

Diaspora	Gnu Social	Friendica
666.617	24.500	8.575
(The Federation, 2015)	(GSTools, 2015)	(Friendica Directory, 2015)

Table 3.1: Active users in 2015 on current major DSNs

3.2 Architecture overview

Unlike centralized services, where there is only one logical server, the Diaspora DSN consists of an unlimited number of single instances. An instance of Diaspora is called a *pod*, which is assumed to be derived from the biological term for a group of aquatic mammals (Wordnik, 2015).

Any user of the network belongs to a certain pod. As figure 3.1 shows, the user account is represented by its unique identifier formed as `user@domain`, where the `user` corresponds to the login name on the pod and the `domain` to the domain of the server, the pod is hosted on. This unique identifier is called the *Diaspora handle*.

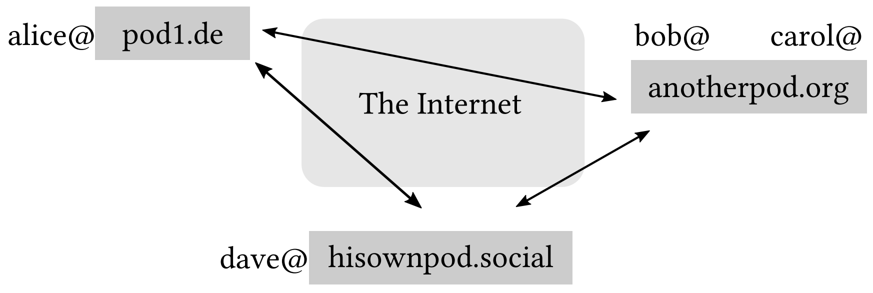


Figure 3.1: The Diaspora DSN

The pods interact with each other via the Internet. This is done by a background service, so that it does not interfere with the front end. When a user submits a post, it is first stored on the local pod and later transfered to other pods asynchronously (see section 3.3).

The friendship concept is asymmetric in Diaspora. Unlike in a symmetric friendship, where both parties see any content of each other instantly, if Alice starts sharing with Bob on Diaspora, she has decided to *send* posts to Bob, rather than receiving his content. Bob in turn gets notified of this and now receives her posts. He may now choose to also share with Alice, but he is not required to do so. If he does not, Alice will only see Bobs content, that he has marked as public.

3.3 Federation

The fundamental aspect of DSNs is the act of sharing data between different pods. Within Diaspora (and some other DSNs as well) this is called *federation*. The technology behind this feature consists of a number of open protocols, that are explained in this section.

Once a new Diaspora pod is set up, it has no knowledge of any other users or pods, so the user will not see any content outside of his own pod. Also, there is no central server that has this knowledge.

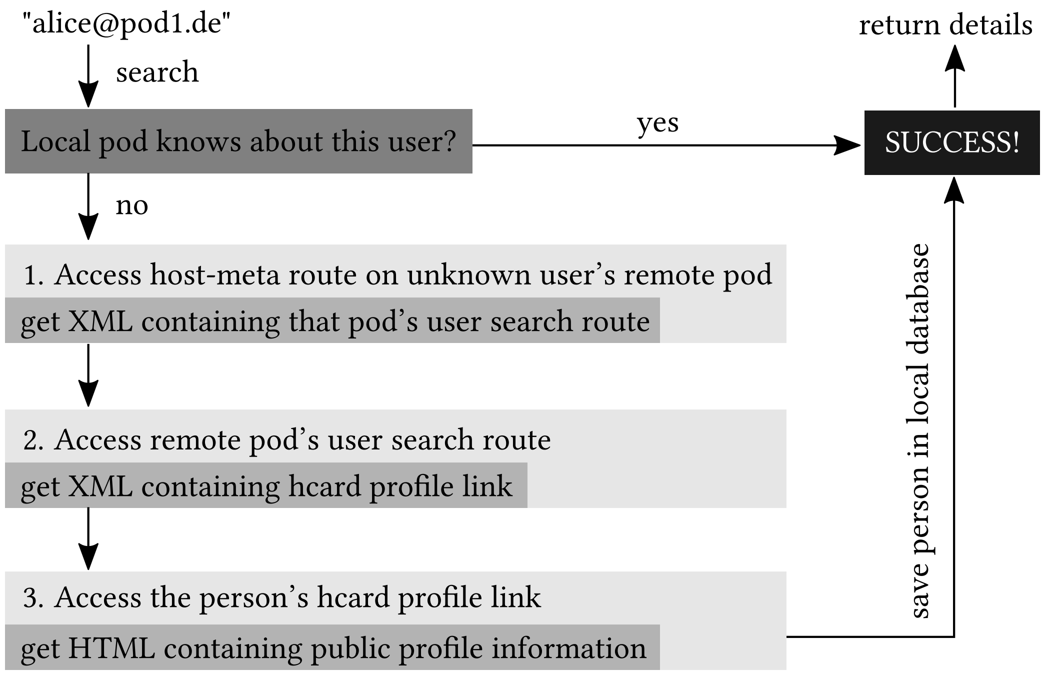

"alice@pod1.de"
search
Local pod knows about this user?
yes
SUCCESS!
return details
no
1. Access host-meta route on unknown user's remote pod
get XML containing that pod's user search route
2. Access remote pod's user search route
get XML containing hcard profile link
3. Access the person's hcard profile link
get HTML containing public profile information
save person in local database

Figure 3.2: WebFinger user discovery

When a user adds a contact from a remote pod, the pods establish a connection to that remote pod. The user discovery is shown in figure 3.2. It starts by searching for the unique identifier called *Diaspora handle*.

If not already in the local database, the pod first starts to figure out where information of the remote user can be found. Diaspora uses the standard protocol *WebFinger* (Norris, 2014) therefore. Using WebFinger, the remote pod is assumed to provide information about how to ask for users at the *well-known URL*:

```
http://remotepod.example/.well-known/host-meta
```

It returns XML-formed information about the search template, which should be the actual *WebFinger URL*:

```
https://remotepod.example/webfinger?q=bob@remotepod.example
```

Requesting this Uniform Resource Locator (URL) returns XML-formed data about the user like the profile page *hCard URL*:

```
https://remotepod.example/hcard/users/b4a21a409a35b368
```

The hCard of a user (Çelik and Suda, 2013) provides more detailed content formed in Hypertext Markup Language (HTML), that the local pod can display to the user (see figure 3.5). The local pod stores all the information retrieved by the remote pod in its own local database. Now the local pod has knowledge of this remote pod and the remote pod in turn knows about the local pod as well.

The Diaspora software uses the *Salmon* protocol (Engestrom, 2013) to send content from one pod to another. It is a standard protocol for comments and annotations to *swim upstream* to update original sources.

Figure 3.3 shows a sequence diagram of the Salmon Real Time Commenting Flow. First, a new entry is posted on the source, published to subscribers via *PubSubHubbub* (Fitzpatrick et al., 2014), and re-published by an aggregator. Next, a new comment called *relayable* (Tilley, 2012c) is posted on the aggregator. It is pushed back upstream to the source using Salmon. Finally, the source pushes the *relayable* comment to all subscribers.

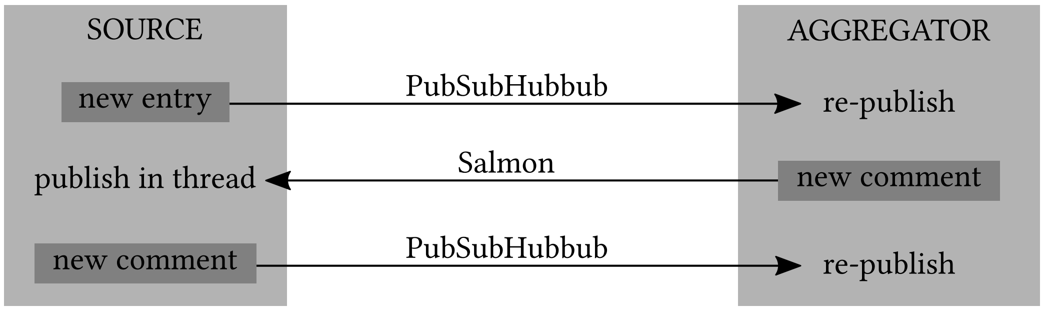


Figure 3.3: The Salmon Real Time Commenting Flow

Sending a message is divided into three tasks: construct the message called *slap*, construct the URL for the remote salmon endpoint, and actually posting the message. Messages in Diaspora are sent encrypted using the public RSA key of the remote user, that is announced through WebFinger. Diaspora extends the structure of the payload by meta data about how to handle the encryption. The constructed slap holds *base64* (IETF, 2006) double-encoded data with data-type and encoding parameters besides the payload. It is signed with the private RSA key of the local user before being ready for delivery.

The URL of the remote endpoint is constructed using the pod URL trailed by the `/receive/users/` route and the user's `guid`, which is a globally unique identifier of the user account (see section 5.6). Sending the message is done by a POST request to that URL. It requires the data to be assigned as `application/x-www-form-urlencoded` MIME-Type and differs from the original Salmon protocol by adding `xml=((double urlencoded salmon slap))` to it.

Receiving a salmon slap is basically identical to sending in reverse order. Since nothing about the author is sent in clear text within the slap, first the payload must be decrypted to find out who the message is from. The payload contains the Diaspora handle of the sender, which is needed to get the public key through the WebFinger protocol to verify the signed message.

Before persisting the received content in the database, the receiving pod also verifies, that the Diaspora handle of the sender of the content is identical to the Diaspora handle of the content's author. This implies that every entity that should be federated must have a user account associated with it as its author.

3.4 Applied software components

The core components of the Diaspora software (Haß, 2013):

Application framework The application framework used by Diaspora is *Ruby on Rails*. It is designed as a Model-View-Controller (MVC) design pattern and communicates through a routing mechanism following the REpresentational State Transfer (REST) paradigm. It features the usual environment modes *production*, *development* and *test*.

Database *MySQL* or *PostgreSQL* are available as database servers to store persistent data like users, posts, comments etc.

Webserver *Unicorn*, the HTTP server for Ruby on Rails environments, generates the dynamic content.

Template engine To render HTML for the user interface, the templating engine *Haml* is used. Its core advantage to embedded Ruby is to avoid mixing up HTML and Ruby code inline.

Client Application Framework In the browser the JavaScript framework *Backbone.js* is responsible for structured data handling. It is also based on the MVC paradigm and provides a RESTful Application Programming Interface (API) in JavaScript Object Notation (JSON).

Background processing Tasks that need more time for computation are done by the background processing system *Sidekiq*. This includes communication to other Diaspora pods or fetching remote profiles.

Process communication To handle communication between Unicorn and Sidekiq, the key-value store in-memory database *Redis* is integrated.

3.5 User Interface

The Diaspora user interface is very similar to other social network services.

Figure 3.4 shows the main view called the *Stream*. It holds status updates, reshares and other postings by people who share with the account of the current user. Each entry has handles to like, reshare and comment on it. On top there is a submission form for the current user to post new content.

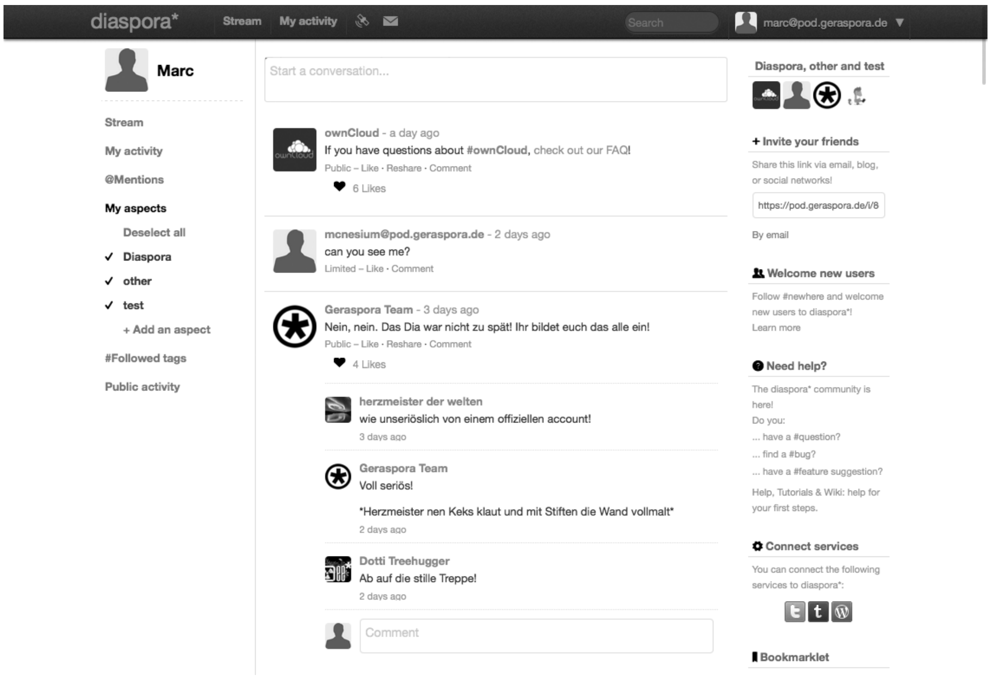
There are two sidebars on the left and on the right. The left sidebar holds the main menu, where the content of the stream can be modified. Different user aspects can be selected or deselected to filter the corresponding postings. The right sidebar holds helpful support links, information about friendship connections and other meta-content, that is not directly related to the stream elements.

There are other views for a single post, search results (figure 3.5), messaging, notifications and settings. The interface features HTML5 elements and is responsive for small- and large-scale screen resolutions. There is also a separate mobile view, that only applies on handheld devices. It features less functionality, but acts as foundation for an Android application called *Diaspora Native WebApp*.

3.6 Development status

The Diaspora DSN software is currently in active development. Following the Git Branching model (see section 2.4), releases are separated into distinctions in the form `major.minor.hotfix`, as described by Preston-Werner (2013). There is a fixed schedule for minor releases scheduled to every six weeks. A minor release will contain fixes, features and changes that can be applied to a pod running in production mode without following a migration guide or a downtime for more than a few minutes (Faldrian, 2015). As of December 9, 2015, version 0.5.4 is the latest stable release.

The Diaspora development, code release, issue tracking and pull request handling is done on Github at `https://github.com/diaspora`. Further information for installation, development and other knowledge is held in a separate Wiki at `https://wiki.diasporafoundation.org`. Project management, discussions and decision making is undertaken on a social discussion platform called Loomio (2015), short discussions and minor consultations mostly on Internet Relay Chat (IRC) in the Freenode network, channel `#diaspora-dev`.



Start a conversation...

ownCloud - a day ago
If you have questions about #ownCloud, check out our FAQ!
Public – Like · Reshare · Comment
6 Likes

mcnesium@pod.geraspora.de - 2 days ago
can you see me?
Limited – Like · Comment

Geraspora Team - 3 days ago
Nein, nein. Das Dia war nicht zu spät! Ihr bildet euch das alle ein!
Public – Like · Reshare · Comment
4 Likes

herzmeister der welten
wie unseriöslich von einem offiziellen account!
3 days ago

Geraspora Team
Voll seriös!

Herzmeister nen Keks klaut und mit Stiften die Wand vollmalt
2 days ago

Dotti Treehugger
Ab auf die stille Treppe!
2 days ago

Comment

Diaspora, other and test

+ Invite your friends
Share this link via email, blog, or social networks!
https://pod.geraspora.de/i/8
By email

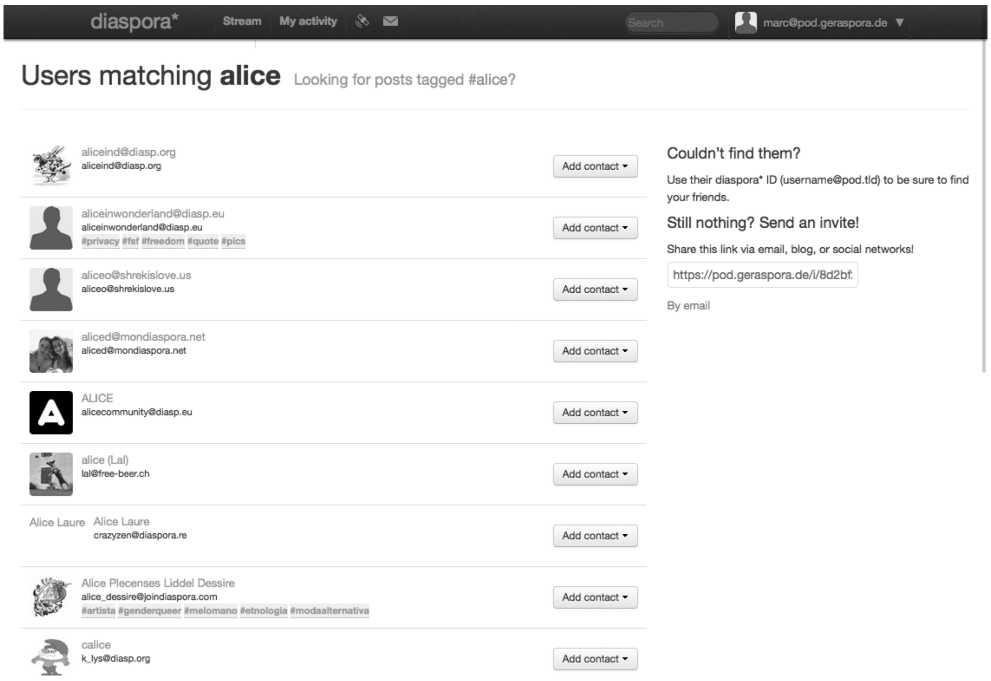
Welcome new users
Follow #newhere and welcome new users to diaspora*!
Learn more

Need help?
The diaspora* community is here!
Do you:
... have a #question?
... find a #bug?
... have a #feature suggestion?
Help, Tutorials & Wiki: help for your first steps.

Connect services
You can connect the following services to diaspora*:

Bookmarklet

Figure 3.4: Diaspora Stream



Users matching **alice** Looking for posts tagged #alice?

aliceind@diasp.org aliceind@diasp.org	Add contact
aliceinwonderland@diasp.eu aliceinwonderland@diasp.eu #privacy #faf #freedom #quote #pics	Add contact
aliceo@shrekislove.us aliceo@shrekislove.us	Add contact
aliced@mondiaspora.net aliced@mondiaspora.net	Add contact
ALICE alicecommunity@diasp.eu	Add contact
alice (Lal) lal@free-beer.ch	Add contact
Alice Laure Alice Laure crazyzen@diaspora.re	Add contact
Alice Plecenses Liddel Dessire alice_dessire@joindiaspora.com #artista #genderqueer #melomano #etnologia #modaalternativa	Add contact
calice k_lys@diasp.org	Add contact

Couldn't find them?
Use their diaspora* ID (username@pod.tld) to be sure to find your friends.
Still nothing? Send an invite!
Share this link via email, blog, or social networks!
https://pod.geraspora.de/i/8d2bf
By email

Figure 3.5: Diaspora Search Results

Chapter 4

Concept

The goal of this thesis is to show the ability of a software to be extendable by abstracting models and evolving them to other functionality. The DSN software Diaspora (see chapter 3) will be used to demonstrate the proposal.

According to the task definition, an overall concept for federated events is introduced in this chapter. This includes different approaches to construct an event model and its properties, relations between users and events and the administration aspect. Further, challenges concerning Federation are discussed, and the feature set of the application module is specified. Concluding the concept some mockups for the user interface in the browser are introduced.

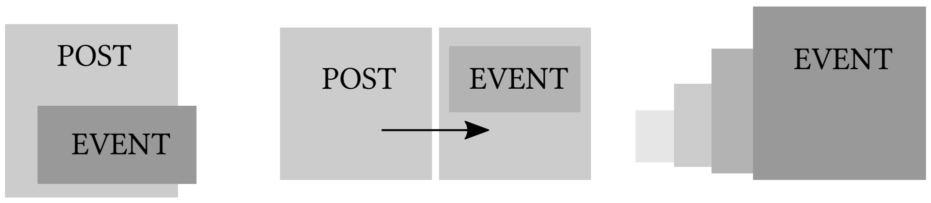
4.1 Model development strategy

A number of different approaches to add events to the feature list of Diaspora are to be concerned.

As a first option, an event could be handled as an extension of a regular post (see figure 4.1a). This would be easy to implement, as it would not be much different than the existing poll function, that Diaspora already features. As a downside, this approach would closely connect an event to an individual user.

Another approach would mirror the post model and extend it with event-like features (see figure 4.1b). This would be closely related to the post model, but feature certain important differences. As the most important one, this way an event can be connected to multiple users. As personal experiences from other event portals show, it is a very important feature to have multiple users be able to administrate an event. This top-down strategy is most likely to produce ready-to-play results, but requires deep knowledge of the software characteristics.

A third strategy introduces an event as an individual model and a bottom-up development technique (see figure 4.1c). In this case, a clean model with only a basic feature set would be implemented and slowly evolve and equip with features that integrate into the Diaspora environment. This strategy is focused on stable features and interoperability with software other than Diaspora as well. This follows requirements according to the task definition and therefore it is the favored strategy for this thesis.

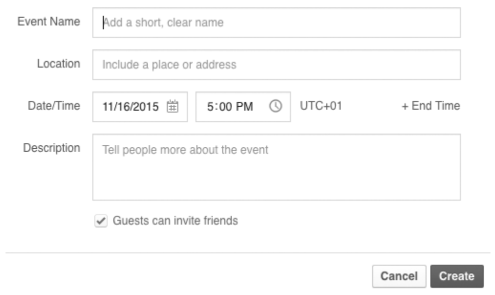


(a) Extend a post (b) Copy post model (c) Evolve event model

Figure 4.1: Backend model development strategies

4.2 Properties

To ensure a familiar user experience with social events, it is reasonable to consult other social network services like Facebook (see figure 4.2) for certain properties, a social event might have, and adopt them in the concept for Diaspora events.



Event Name	Add a short, clear name
Location	Include a place or address
Date/Time	11/16/2015 5:00 PM UTC+01 + End Time
Description	Tell people more about the event
	✓ Guests can invite friends

Figure 4.2: Facebook event form revealing properties of a social event

- *name* string, that holds a short and clear name to act as a title for the event
- *datetime start* and an optional *datetime end* object
- *location name* string, that holds a place or address to point to
- *description* text, that holds any particular event information

Additionally and also a common property of an event on social network services is an *image* for event posters or other extending graphics. A Diaspora event should include the option to add an image to it, which can be visualized next to the essential event data within the stream or an individual view.

The *description* in Diaspora events may act as a regular text field of a status message. This includes that it can hold tags, so that events can be categorized with the Diaspora built-in taxonomy.

Diaspora already features locations, that may be attached to posts. If so, the web application gets the current location from the HTML5 Geolocation API (Popescu, 2014) and utilizes the search engine of Open Street Map (Twain, 2015) to match a name, which will then automatically be attached to the post. For events, the location search function can be convenient for known locations, but should not be automatically attached to events. Instead, the user may search for known location names and choose between search results or enter one manually, if no sufficient location was found.

4.3 Administration

As stated above, the administration of an event should be possible for multiple users. Granting write permissions to an event to remote users is a challenging effort with respect to federation (see section 4.5).

There are a number of different approaches to implement that functionality. A very lightweight possibility is to keep the administration feature for events open just to users on the same pod. This would purge the complexity of remote administration, but also fully contradicts the federation concept, so it is not regarded as a serious option.

Another approach would keep a list of authorized users on the pod of the creator. When a remote person tries to edit an event, that list is being called to verify the authorization of the remote person to edit an event. That would imply this pod to always be available for any other pod. This contradicts the concept of individual instances in a DSN, so it should not be the basis of the concept either.

Instead, the concept makes use the fact that publishing an event basically copies the event object to the database of the remote user's pod. The list of editors will be a relation of a person to the event and thus, a *relayable*. The remote pod itself implements write permission to the user to edit the event locally.

Because any federated entity must have its own author in Diaspora federation, the remote edited event is not sent back to the original pod itself, but instead an *EventUpdate* is created, whose author is the editing user. This update is sent back to the original pod, where the original event is updated using the update's properties (see section 4.5).

4.4 Relations to an event

Defining relations between an event and any corresponding user is a challenge within this thesis. Two different approaches have been attempted and evaluated.

As a user creates an event, he should automatically be its owner. Thus a first approach defined, that owning an event forms a relation to it. Another relation is the claim to attend to an event. When a person invites another person to an event, this would be a third alternative relation, that must be stored.

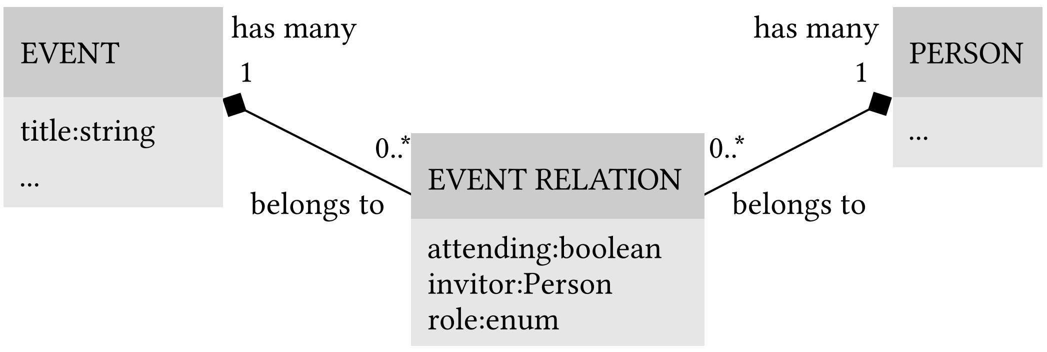


Figure 4.3: First attempt event concept class diagram

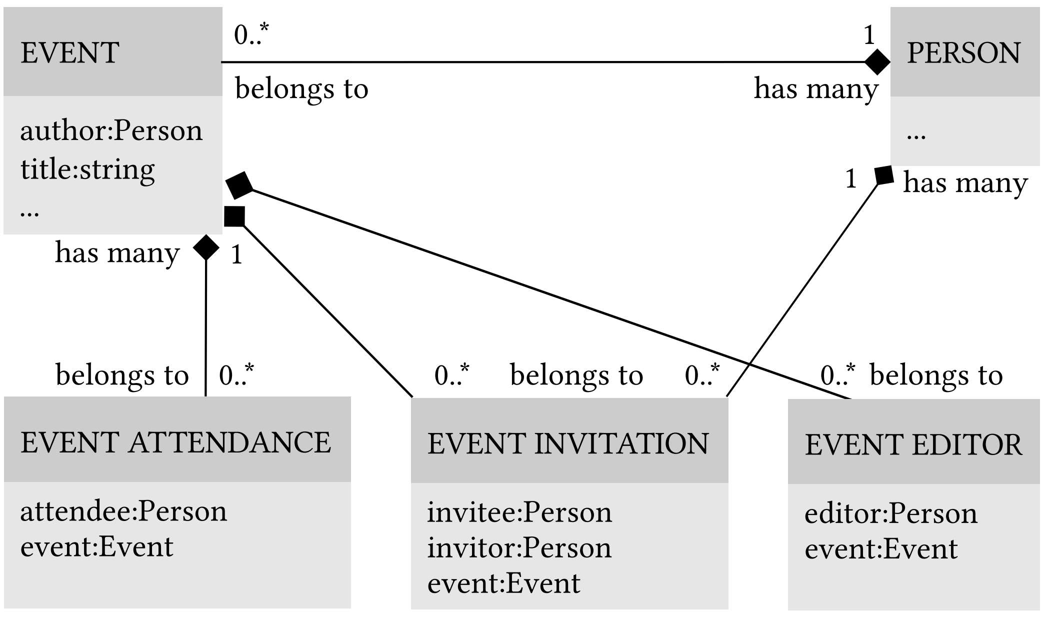


Figure 4.4: Class diagram of the relations to an event

Figure 4.3 shows a class diagram of an event with its direct properties (see section 4.2) and one entity with its properties defining the relations of a person to an event. That acts as *relayable* to the event entity and holds its attendances, invitations, and also a relation to a person who is either owner of or otherwise allowed to edit the event (see section 4.3). The latter would be stored as an *enumerable* featuring three different states *owner*, *editor*, and *guest*.

This approach had to be reverted for multiple reasons. As the Diaspora Federation implementation assumes any entity that should be federated to have its author as a direct property of it (see section 3.3), storing such a relation in another entity would be redundant.

The obligatory author also affects the other properties of the relation entity. When Alice invites Bob to an event, Alice is the author of the created relation to the event. When Bob accepts the invitation and attempts to store his attendance to the event, he must edit the relation entity with his attendance, but this updated relation can not be federated, because the sender of the entity, Bob, is not its author, which is Alice.

The consequence of these findings is, that the full stack of relation information to an event can not be stored within one entity.

Realizing the complexity of this case, the concept had to be refactored with respect to these obstacles. The second approach introduces a setup, that consists of one event entity, as well as multiple *relayables* for attendances, invitations and editors of the event, each storing their person-event relation individually.

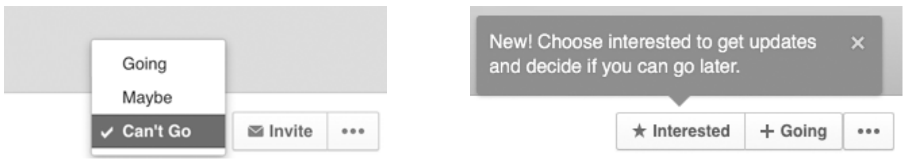
Figure 4.4 shows the class diagram for the refactored concept. It indicates the direct relation between an event and a person through the event's author attribute. The relations to the event are represented by individual entities, each being connected to the event through the event attribute. Further attributes are described in detail, as follows:

attendance The *attendee* is inherited from the event's author, so it does not have an individual relation to a person itself

invitation A relation between to persons and an event. In analogy to the attendance relation, the *invitee* is inherited from the event's author, but the *invitor* as a different person is referenced directly.

editor A mixture of both the two first relations, as there is only a direct relation between a person, the *editor*, and an event through this entity

On other social networks, there are more options to create a relation to an event, like the ability to *not attend*. As opposed to this, there will not be such an option within this concept. Personal experiences show, that a sufficiently high amount of users tend to just ignore an event, if they are not going to attend. In October 2015 even Facebook adopted this with a relation called *interest* as a replacement for the old behavior (see figure 4.5 on page 30).



Going
Maybe
✓ Can't Go
Invite
•••
New! Choose interested to get updates and decide if you can go later.
×
★ Interested
+ Going
•••

Figure 4.5: Facebook attendance

4.5 Federation

Federating an event is done with *PubSubHubbub* functionality of the Salmon protocol (see section 3.3). The relations to an event make use of the *relayability* concept, that is also part of the Salmon protocol. It is also used for comments and likes in Diaspora.

A remote user creates an attendance entity to an event, so his pod sends a *relayable* to the pod of the original event, which in turn will determine according to the list of related pods, what other remote pods will need to see the attendance, and relay the attendance to those pods (see figure 4.6).

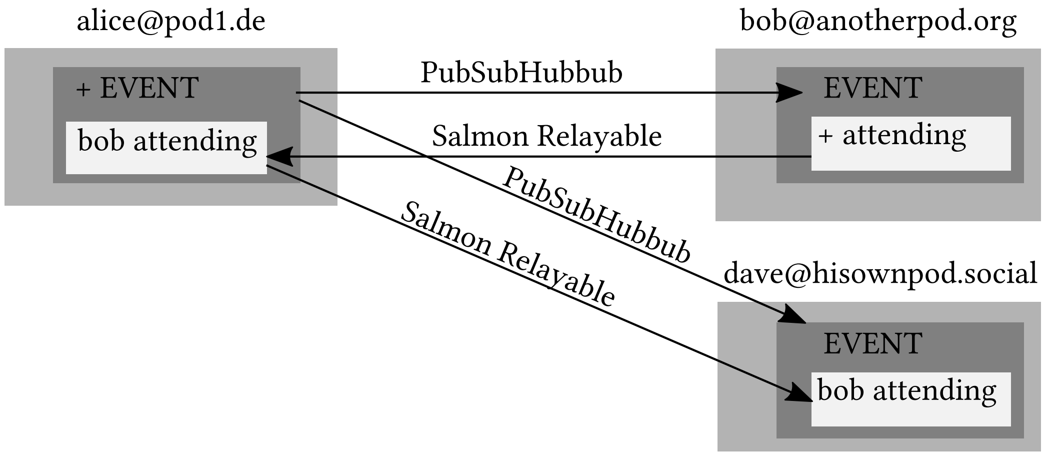


Figure 4.6: Event relation relayabilty

This procedure is insufficient when it comes to editing an event (see section 4.3). As already stated in section 4.4, any federated entity must have a dedicated author. When another person edits any entity and tries to federate it to its original pod, the verification of the author fails, because it is not congruent with the sender in that moment. So there must be an alternative approach with event updates.

Figure 4.7 shows, how this is achieved: another entity called an *EventUpdate* is introduced, that is created upon editing an existing event. It will hold the same properties as an event itself, but will not be persisted in the database. Instead, it is only used to send it to other pods.

As the author of the update is certainly the remote editor, the receiving pod can verify it as the sender. In advance, it uses the properties to update the original event in the local database, and then just discards the received event update.

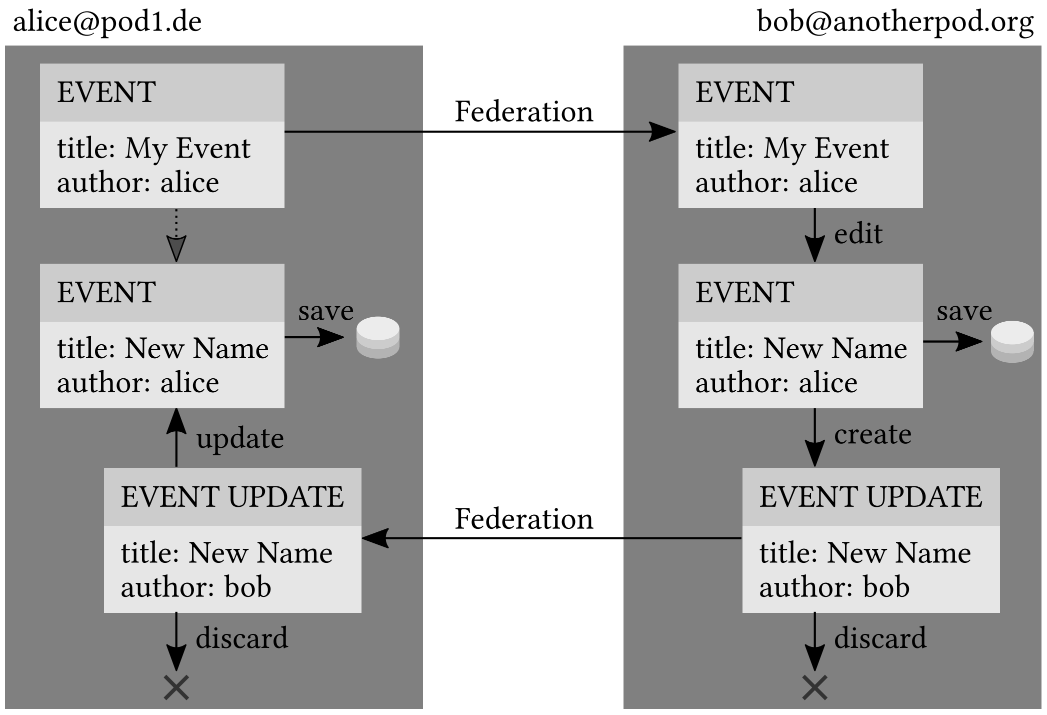


Figure 4.7: Event update federation

4.6 Feature set

This section describes the feature set, that the event feature should incorporate. According to the task definition, this should not relate too closely to the Diaspora software, in order to be deployed on other DSNs systems as well.

The features will be introduced separately in the following, while not elaborating on technical details. See chapter 5 for technical implementation specifics.

Get all events

This feature should return all events stored in the local database of a pod. Only direct properties of each event should be returned. This is useful for the construction of a calendar view. Assuming that all events are public, there is no further distinction regarding privacy aspects as a part of this concept. Further, this call should not require a valid user login, so that also guests may be able to use it.

Get one event

As opposed to all events, calling one event should return all known details of an event, including any attendance as well as any invitation to an event. This is useful for a single event view, which includes the presentation of all invited and attending people. This call should not require user authentication either.

Create an event

When it comes to creating a new event, a valid user account is obligatory, as an event requires an author as a direct property (see section 4.4). So a call to this feature must include authentication. It must provide all information about direct properties stated in section 4.2 as well. Any information must be provided within one call to the application.

Attend to an event

A user who is to claim his attendance for an event must of course provide the event to attend to. Further, user authentication to the application must be provided, so the application can associate a user account to the created attendance relation to the event.

Unattend from an event

This feature is considered the reverse function of the previous. A user has to provide an event as well as authentication to be able to remove the corresponding attendance relation from the event.

Invite a person to an event

When a user wants to invite another person to an event, some kind of identification for the invitee must be provided besides the event. Obviously, user authentication is required for the application to be able to create a relation between the invitor, the invitee and the event. There is no such feature as un-inviting a person from an event, as it is assumed that the person itself will remember the existence of an event in any case.

Enable a person to edit an event

Allowing other people to edit an event should only be allowed to the owner of the event, so authentication is mandatory. Further, the event as well as a person to be nominated as editor of the event have to be provided.

Disable a person to edit an event

This again is the reverse function to the previous. It should also only be allowed to the owner of an event, and therefore requires user authentication. As an editor of an event is a distinct relation to an event, which needs to be removed from the database, only the ID of that relation is required.

Edit event properties

Updating an event can basically include all information, that is needed to create an event. It is not required to provide all information with every edit call, though. Only those properties that are updated, suffice the call. User authentication is mandatory, to ensure that only authorized users may edit an event.

4.7 Alternative approach using iCalendar

A very different approach would be to delegate the task of dealing with events to a calendar server, that serves events in a standard format like *iCalendar* (IETF, 2009). Such a calendar server would take control over the storage of any event, access control and concurrency issues. There are two options to approach this.

One would be to have a separate implementation like Davical (McMillan, 2014), Baïkal (Schneider, 2014) or even ownCloud (Karlitschek et al., 2015), where the Diaspora application itself acts as a client to store and load the actual events from. This would in turn mean to have a dependency for the Diaspora application outside of Ruby on Rails, which is very inconvenient for a straight-forward installation process (see appendix A.2) and will therefore approximately never be integrated in a stable release.

Another approach would require to implement a calendar server application into Diaspora itself. This in turn would mean to advance the software with an enormous feature set, that is actually not absolutely required for the purpose of being a DSN software set.

Both of those options would imply lots of functionality, that the general Diaspora software would barely make use of. As already stated in section 4.3, the administration logic can be achieved by extending already built-in functionality. The KISS-Principle (Hanik, 2007) applies here.

4.8 User Interface

While most of the thesis deals with so-called backend functionality, the concept work to extend Diaspora with social event functionality can also benefit from a draft for an interface design for the frontend user. The Diaspora user interface can be considered a more or less distinct application from the server implementation of ruby on rails, as it is implemented as a Backbone.js featured JavaScript browser application (see section 3.4). Communication between this frontend application and the backend will be realized with a RESTful API (see section 6.1).

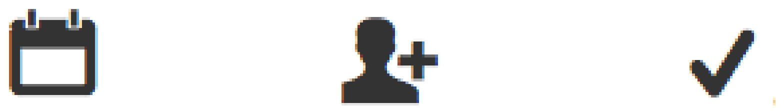
Developing event functionality within this application will be out of scope of this thesis and therefore any implementation will not be realized. To get an idea of a possible user experience of Diaspora events, this section briefly shows a design concept of a user interface for the event feature within Diaspora.

In the stream view the sidebar will hold another section called *My Events* in it as shown in figure 4.9. There will be a small monthly calendar table view with days highlighted, on which there are events the user has stated to attend. Below there is a button *Create new event.* Being a section inside the sidebar, it follows the responsive grid system of the Bootstrap framework, so on a small screen it will be below the stream.

A shared event will appear in the stream between any other content (see figure 4.9). The template therefore must be slightly altered as compared to a regular post, as the essential data *name*, *date*, *location* and *image* should be placed more outstanding than any regular content.

The single view for an event is shown in figure 4.10. It is held closely to the single post view. On the left side of the screen the essential data for the event is shown. To the right, there will be a list of users who announced to attend to that event and those who are invited by others. Below there are comments to the event.

There are a couple of new icons to be introduced (see figure 4.8). They have been taken from the Entypo Font Icon Library (2015) which are used in Diaspora already.



(a) event (b) invite (c) attend

Figure 4.8: New event icons. Source: Entypo Font Icon Library (2015)

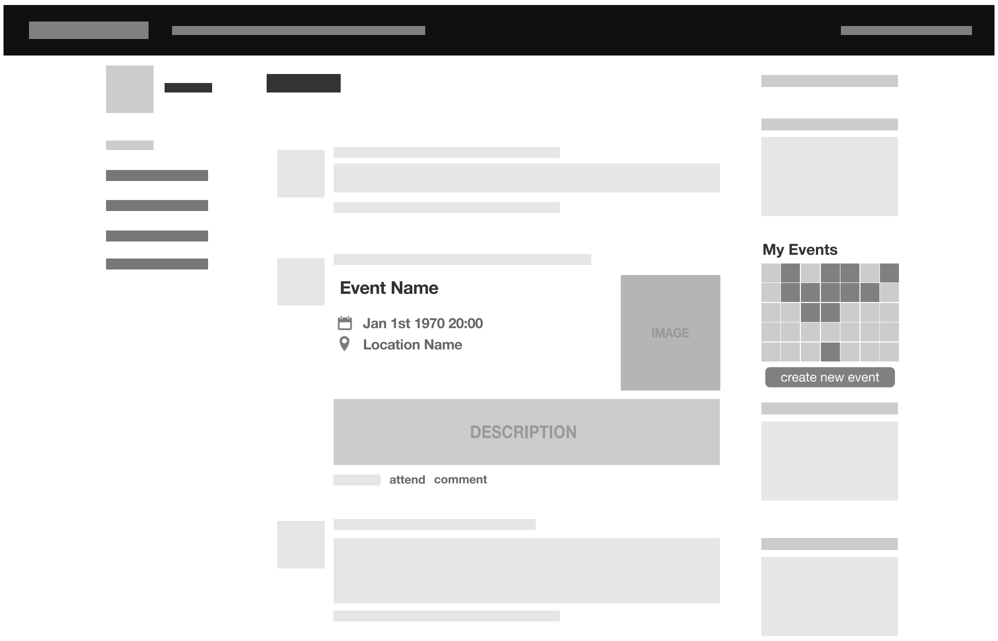


Figure 4.9: Mockup for the stream view including the side bar section

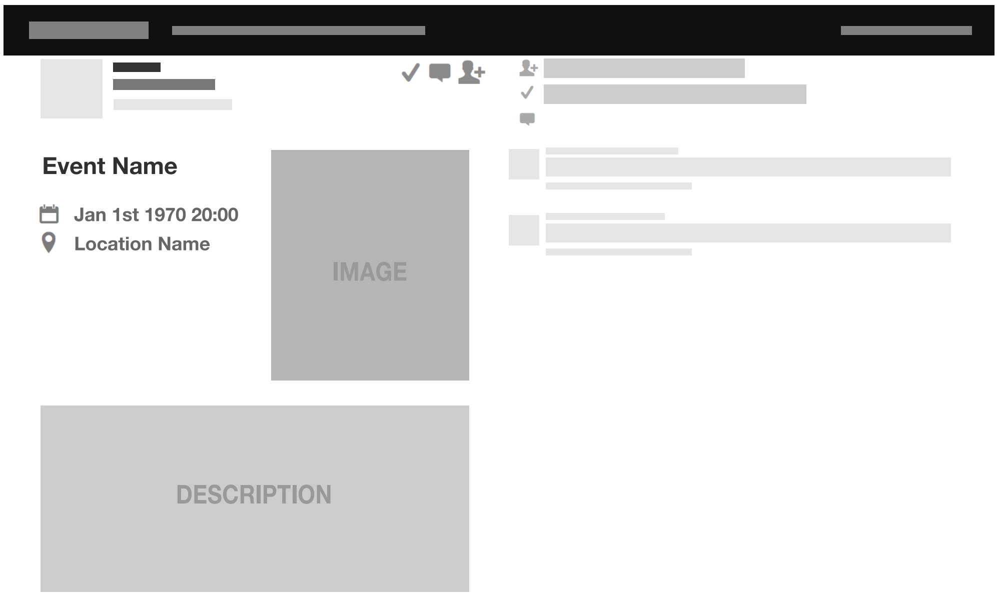


Figure 4.10: Mockup for the single view

Chapter 5

Implementation

This chapter discusses the key aspects of the implementation process. Major concerns about integrating events into the MVC environment as well as key concepts of database migration are explained. The aspect of configuring routes in Ruby on Rails is touched and some examples regarding TDD are presented.

5.1 MVC Integration

The implementation of the concept discussed in chapter 4 should fully integrate into the Diaspora application framework Ruby on Rails. As this is designed following the MVC design pattern, so will the event feature have to be.

Substantial component of any entity within an MVC application resides within the *model*. It manages the data, logic and rules, describes the domain and behavior of it as well as the connection to other entities. The *controller* component is responsible for dealing with user input and manages the output respectively.

The work of this thesis does not include the integration in the Backbone.js featured JavaScript frontend application, so there will be no discussion about the *view* component of the MVC pattern. Instead, there will be a RESTful API provided (see section 6.1), that is invoked in the controller.

As an example for the performance of the MVC paradigm, the according elements of the event entity are described in the following paragraphs.

The *Event* model is the center of the implementation. In listing 5.1 is shown, how it describes the connections to other models using the very readable English-language-like Ruby on Rails syntax. The model also holds a method *receive*, that is invoked on the receiving side of a federation process. As federation only sends information to other pods, the receiving pod is still responsible to persist the information in the local database. This is what `self.save` does.

```
class Event < ActiveRecord::Base

  belongs_to :author, :class_name => 'Person'
  has_many :event_attendances
  has_many :event_invitations
  has_many :event_editors

  def receive(user, person)
    self.save
  end

end
```

Listing 5.1: Event Model

Listing 5.2 shows the *EventsController.* As a remarkable detail, Ruby on Rails conventions name controllers using a plural notation, while the model is named using singular notation. The events controller holds methods that correspond to the different calls to the application following the REST paradigm, that Ruby on Rails is founded on.

In particular, the `index` method returns all events known to the local application. `show` returns just one event and therefore needs the parameter `params[:id]` as an identifier. The `create` method shows, how a new event is created using the given parameters and setting the current user as the author of the event. In advance, it is sent to the federation mechanism using the `Postzord::Dispatcher` module. In the `update` method an event is looked up in the database using the passed identifier, edited and saved. Next, a new instance of an *EventUpdate* is created, using the same properties as the original event has, plus the globally unique identifier (GUID) of the event (see section 5.6), which is in turn sent to the federation mechanism using the `Postzord::Dispatcher` module. The event's `destroy` method uses the retract method of the `current_user` class, which takes care of the local removal of the event and any corresponding relations as attendances, invitations and editors, as well as notifying remote pods of the removal using the federation mechanism.

```
class EventsController < ApplicationController
  before_action :authenticate_user!, :only => [:create, :update]

  def index
    render :json => Event.all
  end

  def show
    render :json => Event.find(params[:id]).to_json(
      :include => [:event_attendances,:event_invitations]
    )
  end
  def create
    event = Event.create(
      author: current_user.person, title: params[:title]
    )
    Postzord::Dispatcher.defer_build_and_post(current_user, event)
    render :json => event
  end

  def update
    event = Event.find(params[:id])
    event.title = params[:title] || event.title
    event.save
    event_update = EventUpdate.new(
      event: event.guid, title: params[:title]
    )
    Postzord::Dispatcher.build(current_user, event_update).post
    render :json => event
  end

  def destroy
    if Event.find(params[:id])
      current_user.retract(event)
    end
  end

end
```

Listing 5.2: Events Controller

Table 5.1 shows models, controllers and the methods, each entity provides for the RESTful API. The related entities do not feature any `index` or `show` methods, as that information is provided when a single event is shown. As a notable difference of the *EventUpdate* to the others, this entity only consists of a model. This is not necessary, as it will never be called using REST, but merely through the `EventsController` (see listing 5.2).

Model	Controller	Methods
Event	EventsController	index, show, create, update, destroy
EventAttendance	EventAttendancesController	create, destroy
EventInvitation	EventInvitationsController	create
EventEditor	EventEditorsController	create, destroy
EventUpdate	-	-

Table 5.1: Models, controllers and methods of the event feature

5.2 Database migration

Besides the integration into the MVC environment, the database must be extended as well. The development environment has utilized the *MariaDB* relational database management system (Widenius, 2015). Ruby

Ruby on Rails ships with a command line interface to alter the database environment. Using the `rake` command, the database migration is called on a file, that is also created automatically and can be altered afterwards.

```
rake db:migrate db/migrate/20150917141231_create_events.rb
```

The datetime string at the beginning of the file name is used to sort multiple migration files by date. Migrations may depend on other migrations, that have been created at an earlier point of time.

A valid *migration* to create entries for Diaspora events is shown in listing 5.3. For every entity a separate table is created. The columns of each table are created within each creation block.

Each column entry can be enhanced with some constraints that are passed to the database automatically by Ruby on Rails. `t.belongs_to :author` tells the database that the corresponding column will be of type `int`, as it will hold the identifier of another entity from the database, which it belongs to. Therefore, Ruby on Rails automatically adds `_id` to the name of the column, so that the name of the column in this example is `author_id`. During runtime, it automatically detects this column, when referencing attributes called `author` within the code (see section 5.6 for special characteristics of the identifier). Further constraints may be attached to the column creation, for example `:null => false`, if the database entry must exist.

As it is required to search for relations between events and their corresponding entities, it is suitable to create a database index for those relations. As an example, when removing an attendance, the following code is called:

```
EventAttendance.find_by_id_and_attendee_id(
  params[:id], current_user.person.id )
```

To avoid time-consuming database operations, indices for relations like those are created, so that the application can process the request in a reasonable amount of time.

The up method describes the creation of the database structure. The Ruby on Rails interface also supports a reverse operation, that uses the down method and is called using `rollback` instead:

```
rake db:rollback db/migrate/20150917141231_create_events.rb
```

```
class CreateEvents < ActiveRecord::Migration

  def up
    create_table :events do |t|
      t.belongs_to :author, :null => false
      t.string :title, :null => false
      t.string :guid
      t.timestamps null: false
    end
    create_table :event_attendances do |t|
      t.belongs_to :attendee, :null => false
      t.belongs_to :event, :null => false
      t.string :guid
      t.timestamps null: false
    end
    add_index :event_attendances,
              [:attendee_id, :event_id], unique: true
  end

  def down
    drop_table :events
    drop_table :event_attendances
  end

end
```

Listing 5.3: Database migration file

5.3 Routing

The interface to a Ruby on Rails application like Diaspora is usually a URL. To create interfaces to the different entities within an application, Ruby on Rails provides a routing mechanism, that form the PATH part of a URL. For basic entities that follow the REST paradigm, creating a route to the entity requires only one command to a file called `config/routes.rb` within the application. The event related entities are created as shown in listing 5.4. These entries create routes as described in section 6.1.

```
Diaspora::Application.routes.draw do
  resources :events
  resources :event_attendances
  resources :event_invitations
  resources :event_editors
end
```

Listing 5.4: Database migration file

5.4 Branching

As discussed in section 2.4, development on version control systems in the Diaspora project utilizes *Git branching*. The implementation part of this work tied on to this strategy and established a similar Git branching tree.

As shown in figure 5.1, the main branch was branched off of the upstream branch *develop*. Being a feature branch, the newly created branch *events* was defined to be the master branch of this thesis work. Further branching and merging was to be sourced from this branch.

The first implementation, represented by the two branches *create* and *federate* was to be truncated, as discussed in section 4.4. The resulting code has not been discarded though, but merged back into *events* for future documentation.

By the time of the refactoring, another version update had been released (see section 3.6), so the event feature branch was updated to the latest version as well. After the update pull from the upstream development branch and the creation of another sub-feature branch *more-relations*, any single relation feature has been created on their own branches.

After completing a relation feature, the code has been merged back to *more-relations*, and finally back to the main project branch *events*.

The code for the event feature has not been merged into the upstream repository by December 9, 2015.

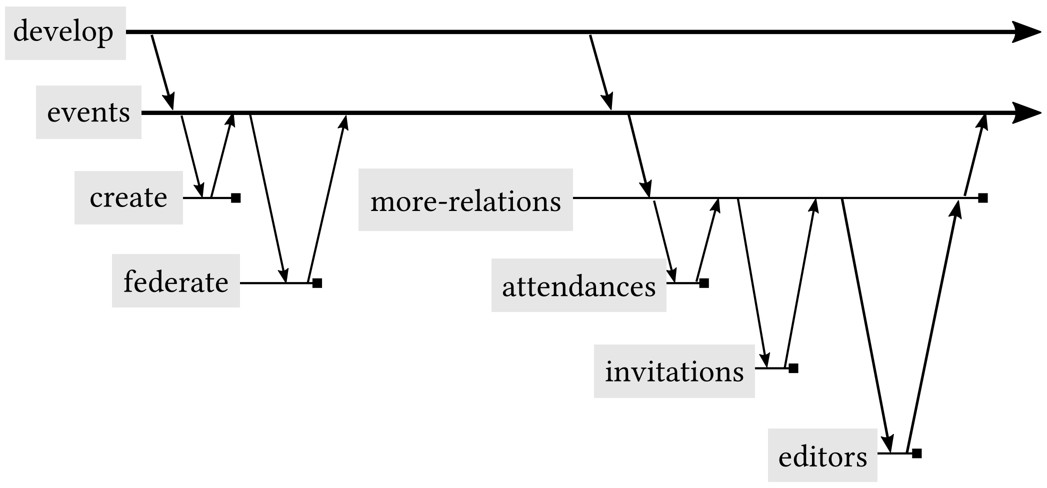


Figure 5.1: Git branching tree of this work

5.5 Testing

As described in section 2.4, testing code is a key method to improve overall code quality of any software project. Diaspora does not use the Ruby on Rails testing feature. Instead, it uses the more sophisticated testing suite *RSpec* (Chelimsky et al., 2014). Listing 5.5 shows the RSpec code for a test on the creation of an event. It tests for a newly created event object to pass the validation credentials, set in the event model, which is accomplished by the method call `.to be_valid`.

```
describe Event, :type => :model do

  before do
    @event = Event.new( title: "My Event", author: alice.person )
  end

  describe 'validation' do

    it 'should create an event' do
        expect(@event).to be_valid
    end

  end

end
```

Listing 5.5: Event creation test

5.6 About GUIDs

In early implementation states the API was to be used not with the presented local identifier (ID) parameters but with a globally unique identifier (GUID) instead.

This GUID is a hexadecimal string with at least eight hexadecimal digits (Tilley, 2012d), that is given to any entity in Diaspora, that needs to be unique across all Diaspora pods worldwide. It is used by the Diasporas Federation protocol (see section 3.3) and thus was to be applied with the events API as well.

While for example for persons this is mandatory for security reasons (i.e. an attacker can not crawl all users of a pod by just incrementing `/people/[id]`), all event data assumed to be publicly available. So there is no necessity to obfuscate them.

In turn, the Ruby on Rails framework is intended to automatically make use of calls to the local ID of an entity. This is explained best with the following code example taken out of the event invitation model. This is just assigning an event object to a variable. Using the GUID requires to explicitly define the attribute to search for:

```
event = Event.find_by_guid(params[:event])
```

When just relying on the ID, the explicit call to the attribute may be left out and Ruby on Rails will automatically know what attribute to look for:

```
event = Event.find(params[:event])
```

While this is just a minor improvement, it gets more perspicuous in the next example. This code declares the associations between the data models of an event and the event invitation. Using GUIDs would cause to explicitly address the primary and the foreign key of an event:

```
belongs_to :event, :primary_key => :guid,
                   :foreign_key => :event
```

Using the ID instead will make the code just as small and readable as this:

```
belongs_to :event
```

Ruby on Rails is intended to provide easy readable code, so it is desired to follow that convention to use IDs where possible.

As any event that is dealt with via the API is known to the local pod and therefore filed in the local database, it does have a local ID and may be addressed with it. There is no need to use the globally unique identifier in the API.

Chapter 6

Evaluation

This chapter evaluates the concept and implementation of this thesis. First, the functionality of the API will be examined, followed by a load test with a large number of events. Furthermore, the general assembly of the concept is discussed along with the evaluation of the development environment and overall experiences with developing free and open source software.

6.1 RESTful API performance

While the Ruby on Rails framework provides an integrated module to output HTML using `format.html`, the user interface of Diaspora is mainly build by the Backbone.js client side application framework.

For this framework to be able to get access to the main application, this thesis work contributes an API to handle events within Diaspora. This API may be called using standard HTTP request methods, which the REST paradigm is based on. The input is done via standard POST request messages with POST variables in the message body carrying the payload. Answers to GET request messages are formatted as JSON string, that may easily be processed by the client application.

In analogy to the features discussed in section 4.6, the corresponding API calls are introduced and explained in this section. The command line interface application *httpie* by Roztočil (2015) is assumed to demonstrate commands for each call, that provides a command `http` to make HTTP calls as shown below.

The listings in this section do not show the entire output. To save room, elements with marginal relevance have been excluded, as there are the properties added by Ruby on Rails automatically, `created_at` and `updated_at`, as well as the GUID, which is only important for federation.

As stated, authentication is important for most of the API calls. Diaspora uses the HTTP state management mechanism called *cookies* (IETF, 2011). To use cookies in the command line environment, a bash variable `$C` is created for the entire HTTP POST variable:

```
C="Cookie:\ _diaspora\_session=bTdJZ1hNbUFpazEJOUh3kz3lQS4dlN3PT0"
```

To keep the presented examples descriptive, line breaks in the printed document are avoided by creating another variable `$S` for the scheme, host and port part of the call in the same way:

```
S="http://pod.diaspora.example:3000"
```

Get all events

This call uses the GET method on the `events` route without any other parameters. Ruby on Rails defaults this call to access the `index` method of the events controller.

```
http GET "$S"/events
```

The answer to this request returns any known events from the database (see listing 6.1). The payload is a JSON-formatted array of events, each itself having their primary properties listed. Besides the author of the event, there are no further relations to it listed.

```
[ {
    "event": {
        "id": 1,
        "author_id": 1,
        "title": "Alices␣Event"
    }
  }, {
    "event": {
        "id": 2,
        "author_id": 2,
        "title": "Bobs␣Event"
    }
} ]
```

Listing 6.1: Array of events

Get one event

This call includes a parameter, which Ruby on Rails interprets as identifier and calls the `show` method of the events controller.

```
http GET "$S"/events/1
```

Listing 6.2 shows the answer to this call. The JSON object again shows the primary properties of the event, but in addition, it includes related attendances and invitations to that event, each as a subsidiary object with its identifier and the related people.

```
{
    "event": {
        "id": 1,
        "author_id": 1,
        "title": "Alice␣Event",
        "event_attendances": [
            {
                "id": 1,
                "event_id": 1,
                "attendee_id": 1
            }
        ],
        "event_invitations": [
            {
                "id": 1,
                "event_id": 1,
                "invitee_id": 2,
                "invitor_id": 1
            }
        ]
    }
}
```

Listing 6.2: Event 1 with its relations

Create an event

This call requests the POST method on the `events` route. Ruby on Rails automatically refers to the `create` method of the events controller on POST requests, and passes the parameters. As this call should only be allowed to a registered user, the method expects a session cookie, which is passed as the bash variable `"$C"`.

```
http POST "$S"/events "$C" title="Bobs␣Event"
```

The answer to this call is shown in listing 6.3. It shows the newly created event as a JSON object with its properties. As this event has just been created, there are no relations to it yet.

```
{
    "event": {
        "id": 2,
        "author_id": 1,
        "title": "Bobs␣Event",
    }
}
```

Listing 6.3: Newly created event

If the call has not passed any session cookie and thus, was not authorized, an error message as shown in listing 6.4 is returned along with the associated HTTP status code (IANA, 2015).

```
HTTP/1.1 401 Unauthorized
{
    "error": "You␣need␣to␣sign␣in␣or␣sign␣up␣before␣continuing."
}
```

Listing 6.4: Result of an unauthorized request

Edit event properties

This call uses the PATCH method on the `events` route, that causes Ruby on Rails to select the event with identifier 81 from the database. According to the REST paradigm, an existing entity is referenced directly within the address. Ruby on Rails automatically refers to the the `update` method of the events controller and passes the parameters along. Any existing parameter will be processed and the corresponding event data will be updated.

```
http PATCH "$S"/events/81 "$C" title="Bobs␣new␣event␣name"
```

The answer to this call is equivalent to listing 6.3, except of course, it shows the updated properties.

Attend to an event

This call creates a new entity of an `EventAttendance`, as it sends a POST request to the `event_attendances` route, passing an identifier in the event variable with it, to create an attendance relation to the corresponding event entity.

```
http POST "$S"/event_attendances "$C" event="1"
```

The answer is similar to the sub-objects of the call for a single event shown in listing 6.2, just the `event_attendance` part with the identifiers for the event and the attendee. In case, the event parameter is not passed, an appropriate error message is returned along with the associated HTTP status code (listing 6.5).

```
HTTP/1.1 422 Unprocessable Entity
{
    "error": "'event'␣required"
}
```

Listing 6.5: Newly created event attendance

If the client application has not recognized an already existing attendance and tries to create it again, the error message shown in listing 6.6 is returned.

```
HTTP/1.1 409 Conflict
{
    "error": "attendance␣exists"
}
```
Listing 6.6: Newly created event attendance

Unattend from an event

This call deletes the `EventAttendance` entity with the identifier 1 in the database, which is referenced directly within the address. The call passes only the cookie variable as a further parameter.

```
http DELETE "$S"/event_attendances/1 "$C"
```

If the call is correctly authenticated and there is an attendance record to delete, it will be processed and answered with a JSON-formatted success message shown in listing 6.7. Otherwise, a `404 Not Found` or `401 Unauthorized` status code will be returned.

```
{
    "success": "attendance␣deleted"
}
```
Listing 6.7: Newly created event attendance

Invite someone to an event

This call creates a relation `EventInvitation` for the person with the identifier 2 to the event with identifier 1 in the local database.

```
http POST "$S"/event_invitations "$C" event="1" invitee="2"
```

As the attendance creation answer, the returned answer to this call is equivalent to the one shown in listing 6.2 as well. Analogous is error handling, as discussed in the attendance section.

Enable a person to edit an event

This call creates a relation `EventEditor` for the person with identifier 2 to the event with identifier 1 in the local database.

```
http POST "$S"/event_editors "$C" event="1" editor="2"
```

Success or error messages replied to this call are compliant with those to create invitations and attendances.

Disable a person to edit an event

This call removes the entity of the relation `EventEditor` marked with identifier 1 from the database.

```
http DELETE "$S"/event_editors/1 "$C"
```

Success and error messages replied to this call are equivalent to the DELETE call to remove an attendance.

6.2 Performance

Performance of the implementation can be tested in terms of the time consumption of the federation process. Several tests have been executed between three different pods (see appendix A.1), creating events and different relations to it. Exemplarily, the creation and federation of one, ten, a hundred and a thousand events is presented here. It has been created using the `time` tool within following shell command:

```
I=0; time while [ $I -lt 1000 ]; do http POST "$S"/events "$C"
    title="Carols␣$I␣Event"; let I=I+1; done;
```

This command only measures the creation on one pod, so the time differences to the creation on the remote pods have been calculated by comparing the log output of the Ruby on Rails servers. It turned out, that there was no time difference between sending and receiving side. This is resulted in the fact that every event is sent individually within the loop, so there is only a very small amount of data to be transfered with every request.

Figure 6.1 shows the results of the experiment. It unveils, that the time consumption graph rises linearly to the created events.

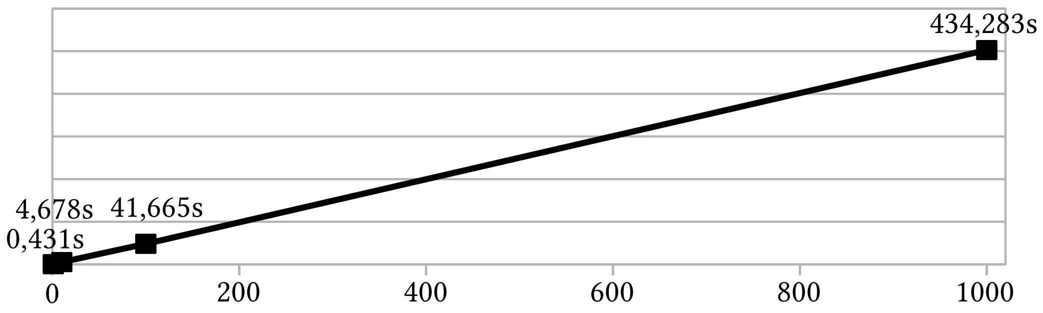


Figure 6.1: Time spent to create and federate 1000 events

However, there are a number of arguments, that lead to the assumption, that the performance of the implementation does not act deterministic. It was found, that the local server response can take up to 20 seconds, if the system has been idle for a while before the first request. This is likely caused by the experiment setup. The three pods run as virtual machines on one host, so the time difference for the data transfer between the pods is negligible. In addition, this experiment has been performed running Diaspora in the development environment. The application behaves differently in production mode, where caching features and other asynchronous operations are taking place. This scenario has not been tested, as a realistic production setup is out of scope of this thesis.

6.3 Privacy

As the implementation resulting from this thesis is just a proof-of-concept, it has assumed that any event is public. This can be demonstrated by get-requesting the `/events` route of any pod, as described in section 6.1. The answer to this request holds all data known to this pod.

Being public data, a pod sends events out to all other pods, it has knowledge of. This also includes, that a pod gets notified of an event, even though nobody shares with people on that pod. This can be considered a downside regarding privacy concerns, but it is a consequence of the event being public data.

In future implementations, this implementation should be extended by the possibility to define certain aspects, an event is published to (see section 7.2). This would imply, that only those pods are notified of an event, that host people the creator of the event shares with. That feature extension would emphasize the attempt to obey privacy concerns of the user base.

Nevertheless, the implementation is integrated into Diaspora, a DSN, which is considered more privacy-protecting by design because of the decentralized data storage. There is no central data storage, that can be compromised, which is a great benefit for informational self-determination.

6.4 Concept assembly

The attempt to extend Diaspora with events has been shared at an early stage with the Diaspora community. There is even a bounty filed on this issue by Jansen (2015). This has also not been the first attempt, but none has reached actual development progress, that has been made public. But there has been some feedback on how to approach this.

A proposal by Robinson (2015) stated the post extension strategy (see section 4.1). This approach would have been fairly easy to implement and promise fast result delivery. However, as the concept also intended to feature administration options for multiple users, this approach became inconvenient to suffice that goal. It seemed more or less like a coding exercise and lacks a certain amount of scientific aspiration.

Instead, the concept of an event to be a separate entity next to a post was focused on. It immediately suggested itself to adopt the post model code to first imitate a post with the same features but another name, and then adapt it to become an event. This would also cause an abstraction layer to be figured out, that would not only work for both a post and an event, but also as a strategy model that could be adapted for other DSNs.

Unfortunately, it came out that the code base of Diaspora did not allow this approach to be successful. The structure of the code is perceived very complex and poorly structured. Section 6.5 discusses the code situation in detail. After some time of analysis this approach had to be discounted as well.

The only way to actually develop at least some basic event functionality was deemed to be the bottom-up strategy. This would leave most of the Diaspora code aside for a moment, and focus on pure Ruby on Rails development. Applying its MVC architecture, the structure featuring an individual event model and another model for each of the relations to the event has been successfully implemented in a reasonable amount of development time.

Ruby on Rails, being a well-engineered development framework for web applications, incorporated well in that process – for the time of developing code that just covers functionality on a single instance. Once it came to federation, it was obvious to go back to rely on what Diaspora already does with posts, and adapt it for events. This implicates the occupation of concepts of the Diaspora code base, which ended up being the most time-consuming part of the entire implementation.

Another obstacle in the assembly of the concept was the fact, that Diaspora does not make use of most of the View-Part of the MVC architecture, Ruby on Rails provides. Instead, it imposes on backbone.js, a framework for rich-client web applications. While in production this may be a good idea, for the purpose of the implementation as a proof of concept for this thesis, it would have been a tremendous amount of work to implement the user interface concept (see section 4.8) just to demonstrate the actual functionality to share events via federation.

The slow development advance resulted in dropping any implementation of the user interface concept. To prove the functionality of the concept, the JSON API is sufficient to fulfill that task.

6.5 Implementation Experience

While Ruby on Rails is one of the major web application frameworks out there, it is somehow different from other frameworks and software environments due to its roots in the Ruby programming language. It is focused on being easy to read, as it was regular English language. Methods may take their parameters not within parenthesis, but attached as a comma-separated list. If the last parameter is a complex object, no braces are needed, it may as well be served just comma-separated.

This can be unfamiliar for users of other programming languages, but if all conventions are strictly maintained, code can be readable and functional. Ruby on Rails in general is well-engineered and production-ready for building web applications – if there is a development concept for a software project, and if conventions are followed. Diaspora does not have that.

Diaspora is build upon Ruby on Rails with those concepts and conventions in mind, but it does not follow them consequently. Besides the default MVC elements `models`, `views`, `controllers` and `helpers`, there are other elements introduced, like `presenters`, `serializers`, `workers`, `services`, `uploaders` and more. None of this is documented in the Diaspora developer wiki (2014), nor explained within the code.

As a prominent example, there is a very central module for federation called `Postzord`, which is most likely a reference to robots in Power Rangers comics (Ryulong, 2013). Regarding the conventions of self-speaking code, this is not helpful at all.

To emphasize the deep nesting of code within Diaspora, the code snippets in appendix B.1 and B.2 and show federation of an event, how it is created, sent, received and stored. Federation is one of the major features of Diaspora, but the implementation is far from being stable. There is a lot of discussion about how to handle public post federation going on until today (Robinson, 2013). The current implementation status might work somehow, but is not reliably useful to easily understand and extend it. This caused the implementation of federated events to take most of the development time.

6.6 Open Source Software Development

The reason for the bad condition of the code base is situated in the history of the project. Back in 2010 there were four enthusiastic college students hacking a software from scratch, financed by a crowd-funded campaign. They were working under pressure and released a hardly functional application early to users (Vijayan, 2010). After two years of further feature implementation and a case of death within the developer team the original developers quit working on the project and the code got released to the public.

From that time it was the job of the developer community to evolve the software. This is hard, when people write code in their free time. Figure 6.2 shows the commit history of the entire project. Since the original developers quit in 2012, the commit frequency decreased immensely.

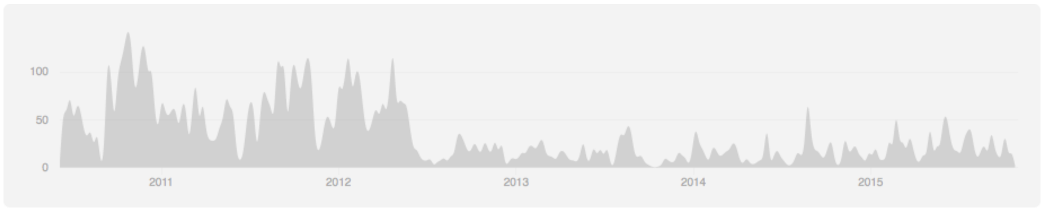


Figure 6.2: Contributions to develop branch from June 6, 2010 to October 31, 2015, excluding merge commits (Github, 2015)

In contrast to other large FOSS projects like for example Mediawiki or Wordpress, there is no company behind Diaspora, that hires developers to work for money on the project. So it is up to the free time of every single contributor, what is done and how it is done. It is obvious, that there are lots of pieces within the software, that have not been modeled after a concept, but just adapted from other pieces. Patches and new components have just been added somewhere and somehow, with the only intention to improve a single feature. Considerations for aspects of professional software development, like following a larger concept, adaptivity, documentation or even testing have been neglected in favor of getting things done.

This is totally reasonable, as people do this in their free time. There are more than 300 contributors to Diaspora (Github, 2015), and many of them have only contributed very little code. There is an issue list on Github, and whoever feels capable of getting hands on it, does it.

Some contributors are considered core developers, who keep track of pull requests and review the code. There is an online community board on Loomio (2015), where discussions on project management are held. But many discussions are open for months without participation of any volunteer. This is also done in their free time, so the reliability of their job depends on the form of the day. When there are questions about the code, which are usually done on IRC, answers very often sum up with the hint to “read the code”. Of course, it is not a support channel run by a professional support staff, so this again is nothing to subject for criticism. But it explains the overall code quality, and the rather rustic development status of the Diaspora software itself, compared to apparently competing services like Facebook.

Chapter 7

Conclusion

This thesis discussed a concept to establish social event functionality in a distributed social network. It has been indicated as a challenging operation, as a social event consists of a rather complex data structure, and its federation within a distributed network has been an attempt that has not been introduced before.

Generally ambivalent circumstances of contributing to free and open source software community projects state reasons for the stated limitations. But as currently common social networks are suspect to be compromised in terms of privacy, due to their centralized structure, distributed social networks are considered more promising concerning privacy aspects.

The findings of this thesis still encourage the approach to continue contributing into free and open source projects. This final chapter summarizes its contributions. A discussion of open issues and future work round it up.

7.1 Summary

The first chapter introduced the topic of federated events in distributed social networks. It stated the motivation and the presumption for this thesis.

Chapter 2 first defined DSN as the fundamental term within this thesis. It provided an overview of DSN related work, explained the significance of free and open source software and introduced social coding platform development, version control development models and test driven development as being fundamental basics of modern web software development, that are required to contribute to Diaspora.

The key features of the Diaspora DSN itself have been analyzed in chapter 3. A comparison to other DSN software stated, why Diaspora was chosen as example application for this thesis. The architecture overview described the basics of the interaction of different pods, before the federation concept of Diaspora was explained. Applied software components, the user interface and the current development status of Diaspora were reviewed briefly, to provide an overall impression of Diaspora.

Chapter 4 introduced the concept to integrate events into Diaspora. A strategy to develop models for social events in a web application was discussed, before the primary properties of an event and its administrational aspect were introduced. The significant aspects of relations and federation of an event were discussed, as well as alternative and revoked approaches, before the feature set of the concept was constituted. Finally, a design mockup for an integrated user interface for events within the Diaspora frontend application has been introduced.

The implementation of the concept is documented in chapter 5. First, the integration of the concept within the MVC architecture of Ruby on Rails was described. Afterwards, further implementation aspects like the database migration, routing, the branching strategy and testing of the implementation were shown.

In Chapter 6 the concept and implementation were evaluated. The functionality API was examined, followed by a load test with a large number of events. Thereafter, the general assembly of the concept has been discussed along with the evaluation of the development environment and overall experiences with developing free and open source software.

7.2 Outlook

In free and open source software development it is obvious to state that a project is never finished. There may be a concept for a given feature, that can be adequately implemented. But there will always be details that can be fine-tuned, unfixed bugs and further feature requests, and there is no doubt, that this is true for this thesis.

The output of this work is a proof-of-concept for federating events in distributed social networks, manifested as JSON API, that can be operated with any client. To actually make this feature able to be integrated into Diaspora for production, a user interface as introduced in section 3.5 must be developed and integrated into the current backbone.js client application of Diaspora.

In future development there may also be features, that pay respect to even more privacy related features. In this implementation, any event is assumed to be public. However, as posts in Diaspora may be either published publicly or restricted to certain aspects, that are defined for and by every user individually, this should also be possible with events. A challenging aspect within this use case will be the relation relayability.

Future implementations of events could also be advanced with more sophisticated features. The location feature that Diaspora already integrates, could be adopted for events. Locations that are used periodically, could be saved and provided as an automatic suggestion. Being an evolved entity, a location could become an entity that may be administrated in analogy to an event, providing contact data, opening times and further general information.

The future development of Diaspora in general is cut into multiple ways. On the one hand, the existing code must be analyzed and refactored immensely. On the other hand, there are many features, that people are used to from commercial social network services, that should be available for Diaspora as well, to make people switch to Diaspora. Both of this is hard to achieve, if developers can only contribute to the project in their spare time.

Developing free and open source software has lots of advantages for security and independence. But as long as companies around the world pay their employees to run Facebook pages, rather than to write code for projects like Diaspora, those projects will not be able to compete, but remain roughly functional free time projects to some enthusiasts, while the big players will keep increasing their influence in everyday life of the people around the world, and the awkward concentration of power mentioned in chapter 1 will not disperse.

Appendix A

Technical documentation

A.1 Development setup

The development setup consists of three virtual machines, hosted on a VMWare environment, each equipped with the following software stack:

- Debian GNU/Linux 8.0 Jessie
- Ruby 2.2.1p85
- Rails 4.2.5
- Bundler 1.10.6
- MariaDB 10.0.21
- OpenSSL 1.0.2d
- git 2.6.1

The instances run inside a virtual Local Area Network without Domain Name System, so the existence of the instances have been configured on each instance as well as on the local development machine, using `/etc/hosts` file:

```
141.76.42.75   madev1
141.76.42.76   madev2
141.76.42.80   madev3
```

The hostname of each instance has been declared in `/etc/hostname`. The instances could be reached from the local development machine via their host name only, for example `http://madev1:3000`

Local development environment used for implementation:

- Macbook Pro v3,1 2,2 GHz 4GB RAM
- OSX 10.11 El Capitan
- Atom 1.0
- git 2.6.3
- httpie 0.9.2
- Iterm2 2.1.4
- Firefox 42

A.2 Installation process

- Install Linux
- Clone Diaspora from a remote repository, i.e. Github

  ```
  cd ~
  git clone git://github.com/diaspora/diaspora.git
  cd diaspora
  ```

  The `cd diaspora` is very important, because it initiates path configuration
- Copy database configuration file and insert the credentials:

  ```
  cp config/database.yml.example config/database.yml
  ```

- Copy general configuration file and insert configuration details

  ```
  cp config/diaspora.yml.example config/diaspora.yml
  ```

  The only important setting is `url: ¨http://madev1:3000¨`), because once a pod has been started for the first time, database entries rely on this setting.
- Install ruby components

  ```
  gem install bundler
  bin/bundle install --with mysql
  ```

- Create database entries

  ```
  bin/rake db:create db:schema:load
  ```

  It is important to use the rake script shipped with with diaspora, as this is is configured for optimal performance.
- Start the server

  ```
  bin/rails server -b 0.0.0.0
  ```

  The `-b` option tells the server to listen to any connection instead only the one configured in `config/diaspora.yml`. It has been found that this is necessary for people search between the different instances.

Appendix B

Code examples

B.1 Send an event

```
event = Event.create(
  author: current_user.person,
  title: params[:title]
)
Postzord::Dispatcher.defer_build_and_post(current_user,event)
```

Listing B.1: `app/controllers/events_controller.rb`

```
def self.defer_build_and_post(user, object, opts={})
...
  Workers::DeferredDispatch.perform_async(user.id, object.class.
      to_s, object.id, opts)
end
```

Listing B.2: `lib/postzord/dispatcher.rb`

```
def perform(user_id, object_class_name, object_id, opts)
  user = User.find(user_id)
  object = object_class_name.constantize.find(object_id)
  opts = HashWithIndifferentAccess.new(opts)
  opts[:services] = user.services.where(type: opts.delete(:
      service_types))

  add_additional_subscribers(object, object_class_name, opts)
  Postzord::Dispatcher.build(user, object, opts).post
rescue ActiveRecord::RecordNotFound # The target got deleted
    before the job was run
end
```

Listing B.3: `app/workers/deferred_dispatch.rb`

```
def self.build(user, object, opts={})
  unless object.respond_to? :to_diaspora_xml
    raise 'This␣object␣does␣not␣respond_to?␣to_diaspora␣xml.␣␣
        Try␣including␣Diaspora::Federated::Base␣into␣your␣object'
  end
  if self.object_should_be_processed_as_public?(object)
    Postzord::Dispatcher::Public.new(user, object, opts)
  else
    Postzord::Dispatcher::Private.new(user, object, opts)
  end
end


def self.object_should_be_processed_as_public?(object)
  if object.respond_to?(:public?) && object.public?
    true
  else
    false
  end
end


def post
  self.deliver_to_services(@opts[:url], @opts[:services] || [])
  self.post_to_subscribers if @subscribers.present?
  self.process_after_dispatch_hooks
  @object
end


def deliver_to_services(url, services)
  if @object.respond_to?(:public) && @object.public
    deliver_to_hub
  end
  services.each do |service|
    if @object.instance_of?(StatusMessage)
      Workers::PostToService.perform_async(service.id, @object.
          id, url)
    end
    if @object.instance_of?(SignedRetraction)
      Workers::DeletePostFromService.perform_async(service.id,
          @object.target.id)
    end
  end
end
```

Listing B.4: `lib/postzord/dispatcher.rb`

B.2 Receive an event

```ruby
def receive!
  if @author && salmon.verified_for_key?(@author.public_key)
    parse_and_receive(salmon.parsed_data)
  else
    logger.error "event=receive␣status=abort␣reason='
        not_verified␣for␣key'␣" \
                 "recipient=#{@user.diaspora_handle}␣sender=#{
                    @salmon.author_id}"
  end
rescue => e
  logger.error "failed␣to␣receive␣#{@object.class}␣from␣sender
      :#{@author.id}␣for␣user:#{@user.id}:␣#{e.message}\n" \
               "#{@object.inspect}"
  raise e
end


def parse_and_receive(xml)
  @object ||= Diaspora::Parser.from_xml(xml)
  logger.info "user:#{@user.id}␣starting␣private␣receive␣from␣
      person:#{@author.guid}"
  validate_object
  set_author!
  receive_object
end

def validate_object
  raise Diaspora::XMLNotParseable if @object.nil?
  raise Diaspora::ContactRequiredUnlessRequest if
      contact_required_unless_request
  raise Diaspora::RelayableObjectWithoutParent if
      relayable_without_parent?
  assign_sender_handle_if_request

  raise Diaspora::AuthorXMLAuthorMismatch if
      author_does_not_match_xml_author?
end

def assign_sender_handle_if_request
  #special casey
  if @object.is_a?(Request)
    @object.sender_handle = @author.diaspora_handle
  end
end
```

Listing B.5: `lib/postzord/receiver/private.rb`

```
def author_does_not_match_xml_author?
  return false unless @author.diaspora_handle != xml_author
  logger.error "event=receive␣status=abort␣reason='author␣in␣xml
      ␣does␣not␣match␣retrieved␣person'␣" \
                 "type=#{@object.class}␣sender=#{@author.
                     diaspora_handle}"
  true
end
```

Listing B.6: `lib/postzord/receiver.rb`

```
def receive(user, person)
  raise 'You␣must␣override␣receive␣in␣order␣to␣enable␣federation
      ␣on␣this␣model'
end
```

Listing B.7: `lib/diaspora/federated/base.rb`

```
def receive(user, person)
  self.save
end
```

Listing B.8: `app/models/event.rb`

Acronyms

API	Application Programming Interface.
DSN	distributed social network.
EBSN	event-based social network.
FOSS	free and open source software.
GUID	globally unique identifier.
HTTP	Hypertext Transfer Protocol.
ID	identifier.
IRC	Internet Relay Chat.
JSON	JavaScript Object Notation.
MVC	Model-View-Controller.
REST	REpresentational State Transfer.
RSA	the Rivest-Shamir-Adleman cryptosystem.
TDD	test-driven development.
URL	Uniform Resource Locator.
XMPP	Extensible Messaging and Presence Protocol.

Bibliography

Abel, Fabian et al. (2011). “Semantic enrichment of twitter posts for user profile construction on the social web”. In: *The Semanic Web: Research and Applications*. Springer, pp. 375–389.

Anderson, Chris (2008). “The End of Theory: The Data Deluge Makes the Scientific Method Obsolete”. In:

Baran, Paul (1964). “On distributed communications networks”. In: *Communications Systems, IEEE Transactions on* 12.1, pp. 1–9.

Beck, Kent (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Boutet, Antoine et al. (2015). “C3PO: A Network and Application Framework for Spontaneous and Ephemeral Social Networks”. In: *Web Information Systems Engineering–WISE 2015*. Springer, pp. 348–358.

Boyd, Danah and Nicole Ellison (2010). “Social network sites: definition, history, and scholarship”. In: *IEEE Engineering Management Review* 3.38, pp. 16–31.

BuddyCloud (2015). `http://buddycloud.com`. Online, Accessed Jul. 15, 2015.

Carvajal-Gómez, Raziel et al. (2015). “WebGC Gossiping on Browsers Without a Server [Live Demo/Poster]”. In: *Web Information Systems Engineering–WISE 2015*. Springer, pp. 332–336.

Çelik, Tantek and Brian Suda (2013). `http://microformats.org/wiki/hcard/`. Online, Accessed Nov. 2, 2015.

Chambliss, William J (1995). “Crime control and ethnic minorities: Legitimizing racial oppression by creating moral panics”. In: *Ethnicity, race, and crime: Perspectives across time and place*, pp. 235–258.

Chelimsky, David et al. (2014). `http://rspec.info`. Online, Accessed Jul. 15, 2015.

Constine, Josh (2015). `http://techcrunch.com/2015/07/29/will-facebook-launch-an-events-app/`. Online, Accessed Sep. 14, 2015.

Creative Commons (2002). `http://creativecommons.org/licenses/`. Online, Accessed Nov. 16, 2015.

Dabbish, Laura et al. (2012). "Social coding in GitHub: transparency and collaboration in an open software repository". In: *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM, pp. 1277–1286.

Dam, Rob van den, Ekow Nelson, and Zygmunt Lozinski (2008). "The changing face of communication". In: *IBM Global Business Services*.

DeSN15 (2015). *1st Workshop on Decentralized Social Networks*. `http://datasets-satin.telecom-st-etienne.fr/aboutet/DeSN15/`. Online, Accessed Oct. 15, 2015.

Dhekane, Ruturaj and Brion Vibber (2011). "Talash: Friend Finding In Federated Social Networks." In: *LDOW*.

Diaspora developer wiki (2014). `https://wiki.diasporafoundation.org/An_introduction_to_the_Diaspora_source`. Online, Accessed Jun. 2, 2015.

Driessen, Vincent (2010). *A successful Git branching model*. `http://nvie.com/posts/a-successful-git-branching-model/`. Online, Accessed Jul. 15, 2015.

Engestrom, Jyri (2013). `http://www.salmon-protocol.org/`. Online, Accessed Nov. 2, 2015.

Entypo Font Icon Library (2015). `http://dsyko.github.io/meteor-entypo/`. Online, Accessed Aug. 27, 2015.

Faldrian (2015). `https://wiki.diasporafoundation.org/Release_process`. Online, Accessed Nov. 3, 2015.

Fitzpatrick, Brad et al. (2014). `http://pubsubhubbub.github.io/PubSubHubbub`. Online, Accessed Aug. 27, 2015.

Free Software Foundation (2001). `https://www.gnu.org/philosophy/free-sw.html`. Online, Accessed Oct. 15, 2015.

— (2007a). `http://www.gnu.org/licenses/gpl.html`. Online, Accessed Nov. 16, 2015.

— (2007b). `http://www.gnu.org/licenses/agpl.html`. Online, Accessed Nov. 16, 2015.

Friendica Directory (2015). `http://dir.friendica.com/`. Online, Accessed Aug. 18, 2015.

Git (2005). `http://git-scm.com/`. Online, Accessed Nov. 16, 2015.

Github (2015). `https://github.com/diaspora/diaspora/graphs/contributors/`. Online, Accessed Nov. 3, 2015.

GSTools (2015). `http://gstools.com/`. Online, Accessed Aug. 18, 2015.

Hammant, Paul (2013). *What is Trunk Based Development?* `http://paulhammant.com/2013/04/05/what-is-trunk-based-development/`. Online, Accessed Jul. 15, 2015.

Hanik, Filip (2007). `http://people.apache.org/~fhanik/kiss.html`. Online, Accessed Aug. 27, 2015.

Haß, Jonne (2013). https://wiki.diasporafoundation.org/Diasporas_components_explained. Online, Accessed Aug. 18, 2015.

Hoepman, Jaap-Henk and Bart Jacobs (2007). "Increased security through open source". In: *Communications of the ACM* 50.1, pp. 79–83.

IANA (2015). http://www.iana.org/assignments/http-status-codes. Online, Accessed Nov. 16, 2015.

IETF (2006). https://tools.ietf.org/html/rfc4648/. Online, Accessed Nov. 2, 2015.

— (2009). https://tools.ietf.org/html/rfc5545. Online, Accessed Nov. 16, 2015.

— (2011). https://tools.ietf.org/html/rfc6265. Online, Accessed Nov. 16, 2015.

Jansen, Nils (2015). https://www.bountysource.com/issues/59061-events-and-rsvp-module. Online, Accessed Nov. 3, 2015.

Karlitschek, Frank et al. (2015). https://owncloud.org/. Online, Accessed Nov. 3, 2015.

Kayes, Imrul and Adriana Iamnitchi (2015). "A Survey on Privacy and Security in Online Social Networks". In: *arXiv preprint arXiv:1504.03342*.

Konforty, Dor et al. (2015). "Synereo: The Decentralized and Distributed Social Network". In:

Landau, Susan (2013). "Making sense from Snowden". In: *IEEE Security & Privacy Magazine* 4, p. 5463.

Liu, Xingjie et al. (2012). "Event-based social networks: linking the online and offline social worlds". In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 1032–1040.

Loomio (2015). https://www.loomio.org/g/EseV9p4X/diaspora-community/. Online, Accessed Nov. 16, 2015.

Maka, Stephan (2011). "Design and implementation of a federated social network". PhD thesis. Saechsische Landesbibliothek-Staats-und Universitaets-bibliothek Dresden.

McMillan, Andrew (2014). http://davical.org/. Online, Accessed Nov. 3, 2015.

Menon, Aravind (2012). "Big data facebook". In: *Proceedings of the 2012 workshop on Management of big data systems*. ACM, pp. 31–32.

Nejma, Ghada Ben et al. (2015). "Service Discovery for Spontaneous Communities in Pervasive Environments". In: *Web Information Systems Engineering–WISE 2015*. Springer, pp. 337–347.

Norris, Will (2014). https://webfinger.net/. Online, Accessed Nov. 2, 2015.

Peeters, Stijn (2013). http://networkcultures.org/unlikeus/resources/articles/what-is-a-federated-network. Online, Accessed Jun. 9, 2015.

Popescu, Andrei (2014). *Geolocation API Specification.* http://dev.w3.org/geo/api/spec-source.html. Online, Accessed Aug. 18, 2015.

Preston-Werner, Tom (2013). https://semver.org. Online, Accessed Jul. 15, 2015.

Robinson, Jason (2013). https://www.loomio.org/d/9vpoe0UR/public-post-federation. Online, Accessed Nov. 3, 2015.

— (2015). https://github.com/diaspora/diaspora/issues/1359#issuecomment-101998582. Online, Accessed Nov. 3, 2015.

Rogers, Adam (2008). "Tracking the News: A Smarter Way to Predict Riots and Wars". In:

Roztočil, Jakub (2015). http://httpie.org/. Online, Accessed Oct. 13, 2015.

Ryulong (2013). https://en.wikipedia.org/wiki/Zord. Online, Accessed Nov. 3, 2015.

Schneider, Jerome (2014). http://baikal-server.com/. Online, Accessed Nov. 3, 2015.

StatusNet (2015). https://gnu.io/social/about/. Online, Accessed Jul. 15, 2015.

The Federation (2015). http://the-federation.info/. Online, Accessed Aug. 18, 2015.

Tilley, Sean (2012a). https://wiki.diasporafoundation.org/Git_Workflow. Online, Accessed Jul. 15, 2015.

— (2012b). https://wiki.diasporafoundation.org/Testing_workflow. Online, Accessed Jul. 15, 2015.

— (2012c). https://wiki.diasporafoundation.org/Federation_message_semantics#Relayability. Online, Accessed Aug. 27, 2015.

— (2012d). https://wiki.diasporafoundation.org/Federation_protocol_overview. Online, Accessed Aug. 27, 2015.

Twain (2015). *Nominatim.* http://wiki.openstreetmap.org/wiki/Nominatim. Online, Accessed Aug. 18, 2015.

Varian, Hal R and Carl Shapiro (1999). "Information rules: a strategic guide to the network economy". In: *Harvard Business School Press, Cambridge.*

Vasilescu, Bogdan, Vladimir Filkov, and Alexander Serebrenik (2013). "StackOverflow and GitHub: Associations between Software Development and Crowdsourced Knowledge". In: *Social Computing / IEEE International Conference on Privacy, Security, Risk and Trust, 2010 IEEE International Conference on*, pp. 188–195. DOI: http://doi.ieeecomputersociety.org/10.1109/SocialCom.2013.35.

Vijayan, Jaikumar (2010). *Facebook wannabe Diaspora hit on security issues.* http://www.computerworld.com/article/2515604/web-apps/facebook-wannabe-diaspora-hit-on-security-issues.html. Online, Accessed Nov. 16, 2015.

Westin, Alain (1970). “Privacy and freedom. 1967”. In: *Atheneum, New York*.

Widenius, Michael (2015). https://mariadb.org/. Online, Accessed Nov. 16, 2015.

Wikipedia (2015). https://en.wikipedia.org/wiki/Comparison_of_software_and_protocols_for_distributed_social_networking. Online, Accessed Aug. 18, 2015.

Wordnik (2015). https://www.wordnik.com/words/pod. Online, Accessed Nov. 2, 2015.

Zhitomirskiy, Ilya et al. (2011). http://blog.diasporafoundation.org/2011/09/21/diaspora-means-a-brighter-future-for-all-of-us.html. Online, Accessed Aug. 18, 2015.

— (2012). http://blog.diasporafoundation.org/2012/08/27/announcement-diaspora-will-now-be-a-community-project.html. Online, Accessed Aug. 18, 2015.