Technische Universität Berlin
Fakultät IV

Diplomarbeit in Informatik

A Peer-to-peer Infrastructure for Social Networks

Doris Schiöberg

17. Dezember 2008

Aufgabenstellerin:	Prof. Anja Feldmann, Ph. D.
Betreuer:	Dr. Sonja Buchegger
	Prof. Anja Feldmann, Ph. D.
Abgabedatum:	18. Dezember 2008

Die selbständige und eigenhändige Ausfertigung versichert an Eides statt.
Berlin, den

Unterschrift

Abstract

Over the last years online social networks have become more and more popular. For example several million are now participating in Facebook, one of the most famous platforms. People store a lot of data within such networks. This entails giving the data to a company as all these networks are server based. This data can then be sold to other companies. As a consequence no one knows anymore where his data is and what has been done with it. To give users back the control over their data, we propose an approach for building an online social network using a peer-to-peer infrastructure. Our approach includes encryption of the data. In this manner, the data can be distributed, is not under the control of a company, and the user has full control about who can access this data.

Sonja Buchegger and Anwitaman Datta proposed in [PSON] to build a peer-to-peer system for social networks, called PeerSON. In this thesis we describe the design and implementation of the peer-to-peer infrastructure for social networks without central servers, including topology and protocols for information exchange.

Zusammenfassung

Soziale Netzwerke im Internet wurden über die letzten Jahre zunehmend beliebter. Beispielsweise hat allein Facebook, eine der beknntesten Plattformen, mittlerweile mehrere Millionen Mitglieder. Die Menschen legen enorme Datenmengen in solchen Nezwerken ab, was gleichzeitig bedeutet, diese Daten einer privaten Firma zu geben, da alle diese Plattformen serverbasiert arbeiten. Eine Konsequenz daraus ist, dass praktisch niemand mehr weiss, wo all diese Daten tatsächlich landen. Wir stellen in dieser Arbeit einen Ansatz vor, mit dem die Kontrolle über die Daten wieder an den Benutzer zurückgegeben werden soll; ein soziales Netzwerk, basierend auf einer Peer-to-Peer Infrastruktur, in der die Daten verschlüsselt werden. Durch dieses System werden die Daten im Netz verteilt, werden der Kontrolle einer einzelnen Firma entzogen und der Benutzer erhält die volle Kontrolle darüber, wer Zugriff auf seine Daten hat.

Sonja Buchegger und Anwitaman Datta schlagen in [PSON] vor, ein System für soziale Netzwerke zu entwickeln, das auf Peer-to-Peer Prinzipien beruht. Der Name des Systems ist PeerSON. In dieser Arbeit wird das Design und die Implementierung einer Peer-to-Peer Infrastruktur für soziale Netzwerke vorgestellt, die ohne zentrale Server arbeitet. Diese Arbeit bezieht sich vor allem auf die Topologie und die Protokolle für den Informationsaustausch.

Contents

1 Introduction

Online social networks (OSN) – this term describes a part of the so called "Content Based Web", sometimes also called Web 2.0. Why Web 2.0? The innovation is that in contrast to the traditional web, where a web site is filled with content by the owner of the web site, all or most of the content is contributed by the users, the clients. The success of a web 2.0 page is highly dependent on the community of contributing users. If many contribute the content may be sufficiently broad and reach a certain quality.
Along with this new principle the possibility of communication between users by means of a web service is used by more and more people. This has resulted in platforms such as Facebook [FB], Orkut [OK], or the German version StudiVZ [SVZ]. Their content is mainly information about the users themselves and the abilities to find and to talk to other users. Such platforms are referred to as online social networks.
Today Facebook has millions of users and many more features than just uploading the profile of a user and finding others to write them messages. One of the features Facebook provides includes a mini-feed, a list of small entries, where users can put what they are doing right now. Another one is the feature called Wall, one for each user, which offers a quite to guest books on private home pages. Furthermore, everyone can upload arbitrary data-files, music, videos, or the photos, e.g., of the last party.

1.1 Motivation

This implies that users of social networks are releasing a lot of personal data to the social networks. These, namely the servers belonging to Facebook, Orkut, or StudiVZ, have control over this data. As a consequence of using a social network, the users have to live with artifacts such as getting personalized advertisements filling their mailboxes, having their data sold to third party companies, etc. Other companies are using the data for studies with more severe consequences. For example human resources companies have detected social networks as a good source to compile a personality profile. Health insurances may study, e.g., the photos from many parties, that a user posted, and may conclude, that a person with those drinking habits is too high a risk for the insurance company. In addition, there have already been some awkward incidents, e.g., the story about the Facebook Beacon. With the Facebook beacon it is possible to track a user's buying behavior, e.g., on Amazon, and place a note about that in the user's mini-feed. This enables friends to see what the user bought. This includes the person, in particular, that was meant to receive the present as a Christmas gift, see [BEACON]. Therefore it is understandable that

some users stop using such platforms. They may want their data back under their control. Many others, though, are continuing to use social networks, because this way of social life is becoming a substitute for mailing lists, open discussion, and announces web sites, and even sometimes for real-life communication. But they are getting worried about their privacy. For example, if a student resists to register at StudiVZ, she may not get some relevant information about homeworks, learning meetings, and other events not officially organized by the university. Nowadays many of these events are only communicated via such platforms.
Even worse there is now a discussion going on, if the data a user posts on, e.g., Facebook remains the property of the user or if it belongs to Facebook. This is, for example, relevant regarding the question, if a user is allowed to take out his own social graph from one OSN and can transfer it to another OSN.
In conclusion, we note that current social networks have a privacy leak.

1.2 Sealing the privacy leak of social networks

Our proposal for tackling the privacy leak is to give the user back the control over his data by decentralizing the system and using certain encryption mechanisms to make the data accessible only to authorized people. We do so by replacing the server in an online social network with a distributed system of a peer-to-peer network. This immediately implies that there is no single point of access to the data anymore. Furthermore, we encrypt each data item and the user can choose whom to grant the right to access each data item. This ensures that even if someone were to collect all the data from the system it would not help him.
The challenge is to develop an infrastructure that gives us the possibility to keep as many of the features of existing OSNs, such as Facebook, while eliminating all privacy problems. With our approach we address the following problems: First of all we keep the ownership of the data with the user. By encrypting the data, the user has to make a decision for every data item to whom he wants to give it. This way a user's wife will only know about her Christmas gift if he enables her to know. The only issue we cannot resolve with this approach is the one of trust between people. We cannot assure with technical means that a person who has got a key and decrypted, e.g., a photo, does not send it to others, for example after an argument with the person on the photo.
To keep all the features a web-based OSN has, is more challenging. With our proposed P2P application we are able to keep features such the Wall (guest book functionality), and to share files with one's friends. As a first step, we chose to implement our system as a pure P2P system, which forced us to implement an application that has to be run on a user's computer. That means we forgo to use, e.g. Ajax, to send the code on the demand through a web browser, since, for now, we do not want to assume every client is a web server. Unfortunately this implies, we cannot keep the feature of pure web platform, which enables a user to enter the system from everywhere, including Internet Cafes.
We also do not yet implement keyword search. This feature can be added on top of the P2P system, but is beyond the scope of this thesis.

This thesis focuses on the underlying peer-to-peer infrastructure. It does not cover the encryption part, which is the topic of the thesis by Youssef Afify [ENC]. The P2P system includes building a topology in terms of who is connected to whom, deciding where meta information is stored, and where, how much, and how often the user data is stored to achieve redundancy.

1.3 Structure of this thesis

In Chapter 2 we introduce necessary background for building a P2P infrastructure for a social network. Because of the complexity of the topic, we use this chapter as well to explain details on some of our design decisions.
In Chapter 3 we describe in detail how our protocols work and how our implementation works.
Chapter 4 is about the test setup and data sets we need to evaluate our system.
Chapter 6 summarizes this work and gives an overview about the possibilities to continue this work.

1.4 Terminology

The following terms and abbreviations are used throughout the reminder of this document.

DHT Distributed Hash Table. A system in which every node is a hash entry, that stores values mapped on this value.

P2P Abbreviation for peer-to-peer.

P2P system A P2P system is the underlying architecture that defines how peers communicate with each other

Peer A peer in our system is the process run by the user, that can talk to a DHT and to other peers. When talking to the DHT the relation is equal to a client server relation, as peers send requests to the DHT. Nevertheless, we will always talk about peers.

Location A user can run a peer on different machines. To distinguish the machines we use the term location. The string identifying a location is the locator.

PeerID The peerID is the combination of peer and locator to uniquely identify a user at a specific location

Super-peer Super-peers are special peers that form a distributed lookup service.

GUID Globally Unique ID (GUID) is an ID that uniquely identifies a peer.

hGUID We store a GUID as a MD5 hash, hGUID is the hashed GUID.

OSN Online social network.

SN Social network.

User The term user in this thesis is used to refer to the person hat runs our application or uses a web service.

2 Background information and design choices

In this chapter we discuss first how certain Web OSN features work and how they can be transferred into a P2P environment. Then we use a set of typical P2P applications as an example to show why our system is unique and why we cannot completely reuse existing systems. Furthermore, we discuss the background and our related design decisions on possible topologies, the boostrapping problem in P2P systems, and finally the topic of storage.

2.1 Features of Web based OSNs

In this section we discuss the typical features server based OSNs support, using Facebook as an example.

2.1.1 Joining from everywhere

Facebook is a Web service. As such it can be used anytime and at any place as long as the user has a browser and an Internet connection available. The user can log in from everywhere, including Internet cafes. It is OK, if the user forgets to log out, goes somewhere else, and logs in again. The server-based systems have full control over the user's sessions and can handle this behavior. In addition it is OK, if a user closes the browser without logging out.
In our approach we do not have such a server. So we must find a different solution using the support offered by different P2P systems. More details on this can be found in Chapter 3.2
One thing that cannot be changed is the need for a separate application for the OSN. This application can be realized as a browser plugin or as an independent application to be installed on a user's machine. Both will in most cases not be installed on arbitrary computers, e.g., in an Internet cafe. Unless our system becomes very popular, it will not be possible to log in from everywhere.
Nevertheless it is quite feasible to build it in a way that it can be run from everywhere, once the source code can be reached.

2.1.2 Leaving messages

In Facebook users can leave messages for other users or write to their Wall (the guest book equivalent in Facebook). This works at any time, even if the recipient is offline at the time the message is sent. Each message is stored on the web server and is visible to the recipient as soon as he goes online. For the rest of this work we refer to such messages as asynchronous messages in contrast to instant messaging, for which both communication partners have to be online.
For centralized services, the asynchronous way of messaging is very easy to handle

as the server has, as said, full control and the capacity to store the messages. For a decentralized system, this task is a harder one. In Chapter 3 we see how to tackle these tasks.

2.1.3 Uploading myself – my own profile

One of the key features of Facebook is the possibility of creating and editing a profile. They enable each user to publish any information about himself that he's willing to share. In addition, a user can upload all kinds of media files, like music, videos, and photos, to document himself and his life online. When a user is not online, this information is still available to others, because all data items are stored on the OSN server.
This for a centralized service very simple task is another challenge for the decentralized approach. In Chapter 2.3 we give a detailed description on how to solve this issue and its side effects within a distributed system.

2.1.4 Finding friends

The typical Facebook "career" starts with an invitation mail by a friend, classmate, fellow student, or colleague. But how does one find more friends, e.g., former classmates? Facebook offers a search functionality that, for example, allows to issue a query with the name of the school and then returns a list of all Facebook users that have entered this school in their profile. A user can in a similar way find people from his work environment, his university, or even people who are in his address list of a web mail service, such as Yahoo!. This is a special topic for any P2P-system. More on that will be given in Chapter 3.

2.2 Differences to other P2P applications

Many P2P systems and applications hav been built for different purposes. As such it is natural to ask, why it is necessary to design yet another P2P system for OSNs. In this section we review the features supported by different P2P systems.

- Searching files
  This is equivalent to searching for content in the system; something that many P2P systems provide, including collaboration and file sharing systems, except maybe Internet telephony and instant messaging. File sharing, distributed cooperation systems, P2P backup systems, all need the feature to search for a specific file. This kind of search does not mean the search for, e.g., old classmates or people by some form of losse specification, rather the search is for a filename by exact-match-search.

- Access rights – access groups
  We need to provide the ability to set different access rights for each file. Moreover we need the capability to define groups of peers that can execute these access rights. Traditional file sharing does not need that. Backup systems on

	Feature supported by PeerSON	File sharing	P2P backup systems	Collaboration platforms
Searching files	yes	yes	yes	yes
Managing different access rights for file	yes	no	no	yes
Searching specific peers for communication	yes	no	no	yes
Continuously changing groups for access rights	yes	no	no	no
Sending/receiving messages	yes	no	no	yes
Changing files	yes	no	no	yes

Table 2.1: Overview about key features and their support in existing P2P systems

a P2P basis are usually meant to manage the user's own files, so these systems have only one access privilege. Nobody but the file owner is able to decrypt the data. Collaboration systems need to handle different access rights and they also allow to change the group membership for access permissions, but in collaboration systems it is usually a fixed set of persons that changes not very often. The purpose of collaboration systems is different to social networks and so the underlying structure is different.
Part of our project is that it should be possible to change the access rights on files. In the scope of this thesis we do not work on this part as we work mainly on the infrastructure concerning topology and underlying protocols.

- Changing files
  That files – keeping the same file name – change their content over time is special to social networks. Collaboration platforms do that as well, but they do not need to maintain changes over the whole network, but only within a specific group.

- Sending/receiving messages
  Such functionality is only provided in some collaboration platforms. This feature is not needed for P2P file sharing or P2P backup systems.

Table 2.1 gives an overview about the above explained features and how well they are supported by typical P2P systems.

In summary, our system is very different from the traditional P2P systems that handle files. It comes very close to collaboration platforms. But it still differs. Collaboration platforms are built to have a common space for a relatively small and fixed group and are not meant to have a continuously changing set of users accessing different files.

2.3 Storage

This section is about how to handle the actual data of the users. By “data“ we mean the actual content that a user uploads to Facebook, such as pictures, personal data, etc. In Facebook all the data is on the servers of the company that owns Facebook, namely Microsoft. The data is always available and when storage becomes short, they can just add some new hard discs. In our system we want to be as close to “always available“ as possible, so we need to think about where, what, and how many copies to store.
The first and easiest way to do it, is to do nothing. A user has his own data and his friends have this data as well after a while. So the data of someone is only spread among those people who are at least interested in his content and allowed to read it. Depending on how many friends a user has, there will be as many copies and the data will be more available. But in times when the user and all his friends are offline, there is no possibility to find the data anymore.
A first idea, to increase the availability of a file, is that everyone participating in the system provides a small amount of his disc space and each file is certain number of times copied to peers who are not friends.

This leads to more questions, that are beyond this thesis.
How many copies do we need that a file is always available?
How should the peers be chosen, to send a copy to? For this we may also need data about how often and how long a peer is online in average. We also should take time zones into account. If all copies are in the same timezone there might be a time of day when even the most reliable peers are offline.
Do we need each part of a user's profile to be always available? Are there parts, which are rarely asked? What can be deleted, when the system runs out of space? How to deal with users who waste too much space by uploading all their videos and music?

In addition to everyone contributing some part of his disc space, we also have the following idea. Since most DSL connections usually come with a small home router, which have now often also some disc space – some with flash storage, others with the possibility to mount an USB stick, or even with a normal notebook hard disc – we can use them to store copies on. The advantage is that these machines are often always online and change the IP address – at least in Germany – only every 24 hours. The disadvantage is, that for this approach additional work by the users must be put to install the necessary applications on their home router to make that work. But many other projects work with collaboration and input by a community. So it might be worth a try.

2.4 Bootstrapping

To get started, every P2P system needs a way for a peer to contact the system for its first connection. This problem is called the bootstrapping problem. For this problem, there exist several approaches, e.g., using a well-known web site to get a set of IP addresses of peers which are known to be stable and “always“ on.

Another approach is to use caching. The IP addresses of neighbours in the P2P network are stored and the next time a peer wants to connect he first tries the IP addresses he was connected to the last time. Sometimes a list of IP addresses is "hard coded" in the application and is down loaded together with the source code or binary. Often a combination of these approaches is used, e.g., in later version of Gnutella implementations.

All these mechanisms have some weaknesses. Reusing an IP address can be problematic. First of all, most DSL providers at least in Germany have a 24-hour reset. Therefore the IP address which was used the last time is likely to have changed since the last connection. In the worst case, all the IP addresses in the cache have changed or a peer has to connect to a neighbour on the other side of the globe because it is the only one reachable anymore. The caching approach also has a bootstrapping problem. Where should the initial connection go to when the cache is naturally empty?
A website to get a list of IP addresses violates the idea of P2P networks. Ideally a P2P system is completely free of any central component. If the website is the only source to get IP addresses of potential neighbours and is taken down, no new peers can connect anymore. Furthermore, such a website has to be continuously updated. A similar problem is faced by using IP addresses that come with the application, such a list should be updated very frequently or contain very stable peers.

For our prototype we rely on OpenDHT, see Chapter 3. OpenDHT uses a web site on which all active nodes are listed. This provides the advantage that we have always a set of online nodes for testing our prototype.

Our idea for this issue is to have a decentralized P2P lookup service similar to the Domain Name Service (DNS). As underlying structure we suggest to use a DHT in which every node is reachable via a web-like URL, so that the IP addresses can be resolved via a DNS query. How do the peers know the URL, especially many URLs, each lookup node would have its own? For this question, we think of a principle that is similar to Jabber. Jabber allows everyone to set up a Jabber server to support the Jabber network. The URL of such a server is always in the form anyserver.jabber.somedomain. We could establish a network of bootstrapping servers which are connected to each other. The URL of each server is then, e.g., myserver.bootstrapping.de, so every URL contains a fixed part like bootstrapping. The knowledge URLs would then spread like the URLs of jabber servers do. This approach does not only work for P2P OSNs, it could be one big systems for all P2P networks and applications.

2.5 Topology

In general, there exist two basic possibilities how to organize a P2P system: in a structured like Chord and in an unstructured way like Gnutella. In this section we take a look at some of the differences and explain our design choices.

2.5.1 Unstructured topologies

One example of an unstructured P2P system is Gnutella. Unstructured systems result in a random neighbour graph as it has no rules on how to choose a neighbour to connect to. There is also no general rule how to manage data. Searching for files is usually done by flooding queries and, because of that, sometimes quite slow.
Unstructured P2P-systems have proven to fit the needs of e.g. file sharing or video streaming systems quite well. A social network has different demands from those. In such a system, we naturally have a few copies of each file, namely at the owner's machine and at his friends' machines, at least as long as we do not introduce extra copies into the system. Additionally and specific to our case, files are changing over time. We have to be able to handle different versions of the same file. Indeed there may be many versions, e.g., the mini feed of Facebook is updated by some users very frequently. It may change every few minutes. In this case, it would mean to flood all the information necessary to maintain the wall feature, which would swamp the system and lead to errors.
Because of this, we very soon came to the conclusion, that for this special purpose an unstructured P2P system does not meet our demands.

2.5.2 Structured topologies

A structured P2P system, e.g., one with an underlying DHT, is in contrast to the unstructured solution a topology, in which every node gets its own ID. Based on this ID the node is then responsible for a certain set of data bound to that ID. When searching data a query is formulated and then forwarded to the node with the ID that is nearest in relation to the data. This way of directing a query can make structured topologies much more efficient.
As mentioned before, it is rather clear to see that a good solution would be to use a DHT or any other kind of structured layout. But using such a structured system introduces new effects we have to take care of. The traditional DHT system puts a lot of responsibility and managing overhead on each P2P node. Yet as devices like mobile phones and pocket PCs become more and more popular, we have to take the limited resources of such machines into account. So not every device in our network should be forced to incur the overhead imposed by the use of a DHT.
One solution to that is a system of super-peers doing the DHT jobs. The other users can then connect to that DHT-system built by super-peers, sending requests and messages. More details on this design can be found in chapter 3.
That means that we logically divide our system into a control and a data plane, where the control plane is organized as DHT that is seen by the data plane as a kind of transparent service which allows to send requests and updates to without knowing the structure. The second part, the data plane, are the users who have the data and are seen by the DHT as clients. The super-peers that build the DHT behave depending on the rules of the DHT in terms of joining, leaving, connecting to each other, managing the data.
With such a system, we can additionally say that only reliable (always on, fixed IP, maybe even with a Web address) machines are allowed to get the super-peer status.

This system – each user decides for himself whether to be a super-peer – also avoids problems, that came up in the Gnutella [GNU] super-peer system. Gnutella gave peers the super-peer status automatically, which had the effect, that a user going online from a broadband link became a super-peer without knowing about it, which can lead to problems in terms of performance or when using the uplink of, e.g., a university.

Since the first DHT systems, e.g., Chord [CHORD][CHO], were proposed, a lot of different DHT systems have been developed. Therefore we can now choose from a set of different system with different advantages and disadvantages.
The basic functionalities of most DHT systems are quite similar. They provide put and get methods to store and find data. The differences are in how they manage the data, how well they can handle churn, and how efficient they are in finding data, inserting and deleting data, and so on.
As seen, there is not yet an application or system that has the features we want, so we finally have to design our own protocol along with building the topology. But before we choose a DHT system, we have to answer another question of a design principle for our system. We can decide to design a system with smart super-peers, doing all necessary management of meta-data or build a system, in which the peers have to manage most of the work themselves.

Such an intelligent DHT needs some extra abilities apart from storing key value pairs. It needs to be able to differentiate semantically between key value pairs and react to them, e.g. when a user uses a second machine to enter the system, but is already noted as online on the first machine, the DHT needs to exchange all related key value pairs, giving the first machine the state inactive and setting the user online for his second machine.
Keeping user state is only one example, another one is that such a DHT node would need to be able to know if a new connection is a non-super-peer, a normal peer, or a new DHT node joining the DHT with all implicated steps to integrate the new DHT node.
This can be done by extending a DHT or designing a new one.

The second possibility is to put more intelligence into the peer application and find a method that enables peers to find out by themselves the state of other users, versions of files, and so on. The disadvantage is, that we can only use the base abilities of the DHT, namely *put*, *get*, and for some DHTs *remove*. This increases the complexity of the peer protocol and its implementation. One of our goals was to enable also thin clients like mobile phones to participate in our system, so the peer application has to be as simple as possible. The advantage, that we do not need to design or extend a DHT and especially do not have to deploy yet another DHT.
Which of these possibilities to choose depends also on the available systems.

- Chord
  Chord was one of the first DHTs and has evolved since its first implementation. The project itself is still there, but nowhere deployed.
  The Chord project website [CHORDCOM] has the following comment about the code: "At this point no official release for Chord is available. You may access our latest source in two ways: you can download an automatically gen-

erated snapshot of our working code, or you can browse or download the source code using the Mercurial distributed version control system. The anonymous CVS and CVSweb interface is no longer recommended or updated, but is still available.

This version is experimental, under active development, and may frequently be broken. The Chord HOWTO describes in more detail how to download and compile the software. The code base is licensed under an MIT/X11-style license.“

The advantage of Chord is that it is simple and easy to understand. But unfortunately this genius simple system has some weaknesses. Chord is static, a node computes its node-key before entering the system and is responsible for data near to its node-key. The result is that there is no easy functionality for load balancing. Furthermore, the original version of Chord also had no redundancy. Every time a node leaves without adhering to the leave-protocol, it removes data from the system.

- Hierarchical DHTs
  There exists a set of hierarchical approaches for DHTs which come close to what we need. Usually the main difference to our system is, that the hierarchical approach is used to enable better load balancing, to reduce the amount of messages in the whole system, and similar ideas. But it seems that there are no implementations of such a DHT publicly available. So if we decide to build our own, these hierarchical approaches can be an inspiration to avoid reinventing everything from scratch.

- Igor - A Keybased Routing System
  Igor [IGOR] is an active project and the people working on it have a strong interest in cooperation to see their system in productive use. The only, but big disadvantage of Igor is that it is only a routing system. Any further DHT functionality – load balancing, managing data or keys, etc – must be implemented by us. As such, we would need to develop our own DHT system or reimplement an existing one.

- P-Grid
  P-Grid [PGRID][PGR] is a special variant of a DHT, it has a “trie“ structure without hierarchy. The data is stored in the leafs of a binary tree. Moreover, the data can be searched by a prefix, e.g., a file name starts with A, the branch responsible for A is searched.
  An important feature of P-Grid is that it allows range queries. P-Grid has also an implementation but it is not deployed. So using P-Grid, we would at least need to deploy their code and eventually extend it to fit to our idea of a smart DHT. The reason we did not choose to use P-Grid, was that the code was, at the time we had to come to a decision, not ready for usage. We would have had to debug it first, which would have needed some time.

- OpenDHT/Bamboo
  OpenDHT [ODHT][OHASH] is a project that deployed the Bamboo [BBO][BAMB] DHT system on PlanetLab, already running to test our ideas on it. The idea behind that project is to provide a system that is always online and to provide a simple interface. The whole project is for research purposes and is meant to be the basis for client applications being implemented on top of the system.
  This project has some strong advantages. First of all, we can use the deployment of OpenDHT and just have to write a client, the peer application that talks to the DHT-nodes. The problem with that is we cannot change the implementation that is deployed on PlanetLab. Therefore this only works for the variant that puts all intelligence into the peers.
  Additionally, we can do the necessary changes on Bamboo, deploy it in a test environment and do tests with that system. The Bamboo code is in Java, but a peer application could be written in any language. In contrast to other implemented DHT systems, OpenDHT provides an additional remove method, so key value pairs cannot only be put and searched in the system but also removed if necessary.

- Pastry
  Pastry [PASTRY][PAST] was also one of the early projects working on the idea of a DHT for P2P systems. Because of that, the theory behind Pastry is strong and the model has evolved through the years.
  Pastry exists as a open source Java implementation, which one can change to put the intelligence into the DHT or just deploy it and build our protocol around it.
  The advantage on Pastry is, there is a huge tutorial on how to use and write code for it. And there is also a set of scripts to get it running on PlanetLab. Unfortunately the scripts are quite old and need some adjustments.

There are many systems, which come close to providing what we need. For the scope of the thesis, we decided to use the system that was usable without changes and already deployed, OpenDHT. We use OpenDHT to realize the version of a non-smart DHT of super-peers. For more details on the design and implementation see Chapter 3.

3 Design and implementation of a P2P protocol for social networks

As we decided to implement the version with non-smart super-peers, we need to design a protocol for the peers that offers all key features of an OSN. For the protocol we designed, we need at least three methods, put, get and remove to make the system work. We do that using OpenDHT – that is the only deployed system, that provides all three methods – as a network of super-peers that provides a lookup service. We implemented an application for the non-super-peers that works using OpenDHT.
This chapter describes how our protocols work and what we implemented so far. First, we explain how globally unique IDs are chosen and managed. Then, we explain our lookup service built by a set of super-peers, first the version of smart super-peers as a concept, then the version with non-smart super-peers, which we also implemented. The rest of this chapter explains the details on all protocol parts and additionally how features, e.g., instant messaging, work from the point of view of the protocol logic.
Figure 3.1 shows the view on our system for an example use case.

3.1 Globally unique IDs – GUID

A globally unique ID is equal to a user name. We do not have a central instance where a user can sign in with a user name, that ensures that this ID is not yet in use. There exist approaches for finding unique names in a distributed way, which we did not implement. Rather we decided to use e-mail addresses, at least for the prototype. Our reason is, that today almost everyone has an e-mail address that is naturally unique. Moreover services like Facebook and StudiVZ assume that every user has an e-mail account, where they can send at least the confirmation mail to to finish the registration process. The only problem with e-mail addresses is that they have a certain value e.g. for spammers. If we put them into a DHT, the owner of the DHT node can extract the list with all the addresses on this node. Since we use OpenDHT, which is under a centralized control of a researcher, this is not as bad as having a DHT in which anyone can become a node, but it remains an issue. But this is only the case for the prototype.
The solution we use in our prototype is that we use MD5 hashes to “encrypt“ the e-mail addresses. Hash collisions can happen there, but are sufficient rare. So each GUID is the MD5 hash of an e-mail address.

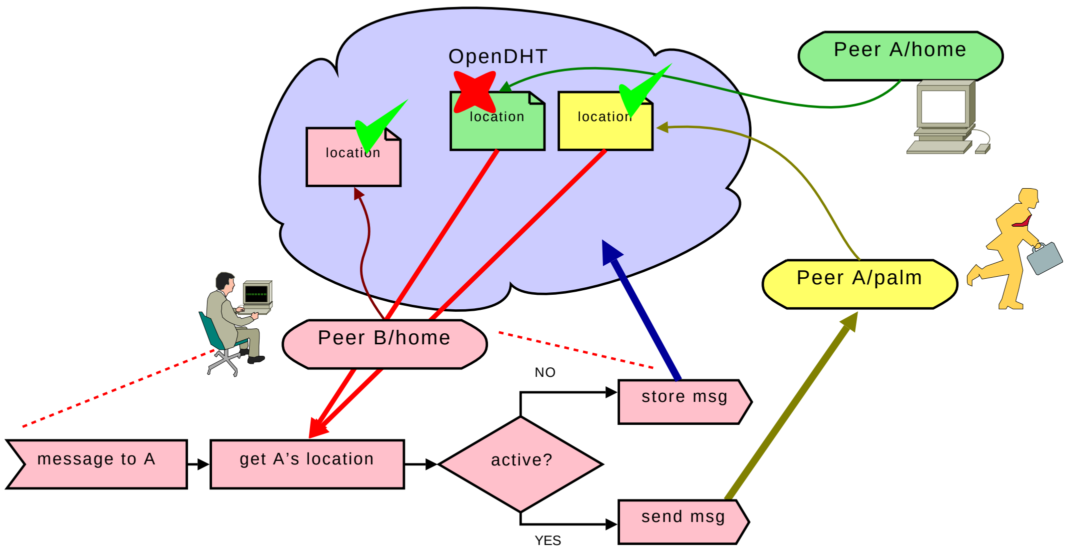


Figure 3.1: Example: abstract view of the whole system in the case a peer B wants to send a message to peer A.

3.2 DHT of super-peers as lookup service

As said before, we think about a system, where a DHT built of super-peers plays the role of a distributed lookup service. We decided to use OpenDHT, because it is already implemented and deployed on PlanetLab, which reduces the amount of work to be done for prototype testing. This decision implies that we cannot choose to put the intelligence into the DHT as we cannot change how OpenDHT works. Nevertheless we want first to explain, how an intelligent DHT would work and then explain what OpenDHT does in detail. So the next section describes how we would build a special DHT for the needs of an OSN, an OSN-DHT.

3.2.1 Smart super-peers – OSN-DHT

In a system with simple peers, the DHT, and with that the super-peers, have to manage a lot of data and maintain it. For now we assume a traditional DHT that brings its own node-ID to data mapping, load balancing, and so on, which we can extend to fit our needs. In the following section we describe the needs of an OSN-DHT and our ideas to solve them.

What does an OSN-DHT need to offer, to make it work? This depends on what users do in such a system.
First of all, peers ask for other peers, their state, IP address, port, and other information. To provide that, the DHT has to maintain a list of all GUIDs of its peers. If a GUID is seen for the first time, the DHT adds it to the list. GUIDs stay there, even when the user is offline. In this case, an entry is added to the list with the offline state and the IP address is deleted from the list. It is still open what happens when a peer does not follow the logout procedure, but just kills the peer process. In that case, the DHT would give a wrong address to another peer about that user,

then the peer would try to connect to his other peer and fail. Usually a peer will try more than once to establish a connection before he gives up, noticing that the address is wrong. In that moment the peer knows more than the DHT. Maybe we can use that fact and introduce a functionality that a peer reports his knowledge back to the DHT.

Second, peers will ask for files, who has which version of a file. So the DHT also has to maintain a list of mappings of files and versions to GUIDs. We have a list in mind that has as search entry the combination of file name and version, sorted by latest version. Every entry in that list has as second column all GUIDs that have this version of the file. Nevertheless, the peers have to do some work and send a message to the DHT each time they have a new file or a new version of a file.

In summary, the nodes of an OSN-DHT have to store a set of lists and maintain them. They need to react to the messages they get from non-super-peers, which means they need to be able to react to the semantic content of a message, e.g., understand when a user sends the logout message and react by changing the user's state on the referring list.

To realize a system like this, we at least need to extend the code of an existing DHT system.

3.2.2 Using OpenDHT as lookup service

The following section describes how we use OpenDHT as a lookup service and how we realize on top of it a system with the version of a non-smart DHT.
OpenDHT provides similar functionalities as most other DHT systems, it mainly works with key value pairs, that can be stored in OpenDHT, searched and downloaded, and – this is quite special – deleted. Apart from that, OpenDHT has some limitations and special properties.
OpenDHT is a centrally maintained system of up to 200 nodes that can be accessed and used by everyone. Each key must be a byte array of maximum 20 bytes. For us, a key is either a GUID or file name, so the 20 bytes would be a problem, but as all keys are hashed it is enough. Values in OpenDHT are limited to 1024 bytes, which led to some problems during the development, e.g. see Chapter 3.10. But this limitations also show that OpenDHT either was meant to be used as lookup service or only for research purposes where it is usually not necessary to store bigger amounts of data with one key. Because of that, it is not possible to store, for example, a video file in the DHT. It is also possible to add a secret to every put message, a string that serves as a password. If a secret is set, a value can only be deleted when the remove request contains the same secret. A special feature in OpenDHT is that keys can be used twice as long as the value is different. That means that a key-value pair can put twice to OpenDHT without overwriting the old one if the values are different. We will explain in Chapter 3.7, when explaining the log in procedure, how we make use of this.
Furthermore, OpenDHT works with timeouts. Each key-value pair has a *time-to-live* value, that can be at maximum set to a week. That implies two things. Any key-value pair disappears from thhe DHT when the *time-to-live* is over. For us, this means a peer that is offline for more than a week disappears from the DHT. That

means he is not found anymore instead of giving the answer that the peer is offline. It also implies that, to do a refresh, we send messages for all files a peer has to the DHT at every single login. With this design a GUID that is offline for a week can no longer be found. But the files of a peer can still be found if other peers have copies of his files.
Another effect is that when deleting a key-value pair, the *time-to-live* of this pair has to be greater than the *time-to-live* of the actual key-value pair. But that also means, that the same key-value pair cannot be put into the OpendDHT until the delete times out. Because of that we had to introduce timestamps in our system as well, see Chapter 3.7 for details. For bootstrapping, OpenDHT uses the fact that it is a "centralized" system. A list of all active DHT nodes is stored on a web site and continuously updated.

3.3 Further methods to connect to neighbours

It is clear how the super-peers connect to each other. They just execute the Bamboo protocol. The peers open a connection to one super-peer and use it as server, similar to a web server. But how do the peers connect to each other?
A peer is not a super-peer, although a user – the person in front of a machine – can run both, a peer and a super-peer at the same time, but the peer and super-peer will always be logically divided. So this section is about how peers connect to each other.
Traditionally, P2P works over the Internet. A peer connects over its provider's uplink to the P2P system. We suggest to include also other links. We reserve fields in our protocol to inform not only about IP address and port of a peer, but also to give other peers information about alternative methods to connect. For example, if a peer A is participating in a wireless mesh project like Berlin Freifunk [BFF], he can put this information together with his other information into the value that belongs to his GUID key. If another peer B wants to connect to A, he asks the DHT for the connection information of A and receives together with IP address and port also the ID and name of A within the mesh and can then connect directly via the mesh instead of going the whole way through the Internet. Alternatively, a user can choose the geographically nearest peer by comparing the peer's GPS data with his own.
Additionally, we plan to include some features of delay tolerant networks, e.g., two peers with mobile phones meeting, e.g., at the university not having an Internet connection at that moment can exchange their profile via Bluetooth. In the moment the DHT does not know about the new location of some files. The synchronisation is done the next time the peers have Internet connectivity.
There are as many possibilities for alternative connections, as methods for data transfer between two pieces of hardware exist.

Our prototype – protocol and implementation – contains only the fields for wireless meshes and GPS data. At this point, it does not yet do anything with the data, but it can be easily extended. The idea is to keep this part interactive, so that a user can decide each time if he wants to use the Internet or any other connection.

3.4 One GUID – many machines

Today, a person usually has more than one PC. There is a PC at home, one at work, a handheld, a mobile phone, etc. But if a user has a set of video files, it might be a problem to store them on his mobile phone because of space limitations. So, if the system tells peer A that B has the desired file this does not necessarily suffice, e.g., B has the searched file on his PC but is at the moment online with his mobile phone.
To get around this problem we introduce an extended GUID, a peer is not only defined by his GUID but by a combination of GUID and a location. This is similar to the way the Jabber protocol does it. The key stored in OpenDHT is still the hashed GUID, but the value contains a field that is called location, so that B's PC can be distinguished from his mobile phone. Further details on this value is given in the section about the login procedure 3.7.

3.5 Announcing content – file management

As denoted before, we need the ability to tell the DHT what files a peer has stored. This section describes the protocol and how we implemented it.
To keep the choice of which file gets which access rights most flexible, we use a single file, e.g. city, name, etc. for every part of the personal data. The following protocol is about the communication between peers and super-peers, how the peers store the information, which files they have within OpenDHT.
Our keys are equivalent to the file name, and the file name is constructed using the hGUID and the type of the file:

```
     key  ::=  ⟨filename⟩
filename  ::=  ⟨hGUID⟩@⟨type⟩
    type  ::=  in|list|fl|fn|ne|bd|ci|wi|we⟨index⟩
   index  ::=  INTEGER
   hGUID  ::=  md5(⟨GUID⟩)
```

As OpenDHT supports multiple values per key, we will possibly receive a list of values. These values contain the information, which of the peers store which version of the file. Hence the user can then choose a peer to download the actual content of the file directly.

```
 value  ::=  ⟨peerID⟩#⟨version⟩
peerID  ::=  ⟨hGUID⟩#⟨locator⟩
```

A peerID consists of the hGUID of the user, that runs the peer, and the locator. For the logical architecture of the system it is both valid, GUID and hGUID. Together with the locator it always denotes uniquely a peerID. The difference is, that OpenDHT knows for its keys only hGUIDs, where as in the human readable context, the GUID in clear text is more important.

Except for wall index, wall files, list, and index, the abbreviations, such as *ne* for name are not needed by the actual application. These abbreviations are only important for the user to formulate the right search string, when searching for a certain file, because the keys need an exact match to get a result by OpenDHT. So these abbreviations are a question of standardization.

Type	Description
in	Index file. This file contains a list of all files which belong to a user's profile
list	List file. This file contains a list of all files stored at the location of a peer, including profile files of other users.
fl	List of friends of a peer.
fn	The file containing the first name of a user.
ne	The file containing the last name of a user.
bd	The file containing the date of birth of a user.
ci	The file containing the city, in which a user currently lives.
wi	Another index file, the wall index. This file contains a list of all wall entries a user has.
we⟨index⟩	This file contains one wall entry. The number of the wall entry is successive and defined on the receiver's side. So if peer A sends peer B a wall entry for the first time, but B already has ten other wall entries, A's message gets the number 11 not 1.

Table 3.1: Implemented filetypes of reference implementation

We defined only a small set of file types for the prototype, the list can easily be extended. See Table 3.1 for a list of which file types we have already defined.

3.6 Interaction between peers to retrieve a file

After a peer has all necessary information about a file, she initiates a connection to the peerID she wants to get the file from. The following protocol is about, how peers (not super-peers) communicate with each other.
The get request:

get request ::= get#⟨filename⟩#⟨version⟩

When a peer receives a get request it answers:

get reply ::= getreply#⟨code⟩#⟨filename⟩#⟨version⟩\n[⟨filecontent⟩]
code ::= 200|404|405

The codes are leant on HTTP reply codes to make it more easy to keep them in mind, see Table 3.2.

Figure 3.2 shows an example for the procedure to get a file. Peer A wants to get the file B@in, the index file of peer B, so it asks the DHT and gets the answer that B has the file in version 2 at location home and work, and peer C has the file in version 1 at location home. 2 is the highest version, so A wants to get the file from B and asks the DHT for the login value of B to find out that B is currently online at location home. Online means that the user behind the GUID B is currently not at his PC, but for receiving a file that does not matter. A sends the get request to B and first receives the get reply followed by the actual content. The example does not show all options that are sent.

Code	Description
200	OK. File will be sent
404	File not found. In that case, only the code, the requested filename and version number is replied.
405	Different version. This means, the requested file exists, but has a different version than the requested one. In this case the newest version available is sent.

Table 3.2: Reply codes for get replies

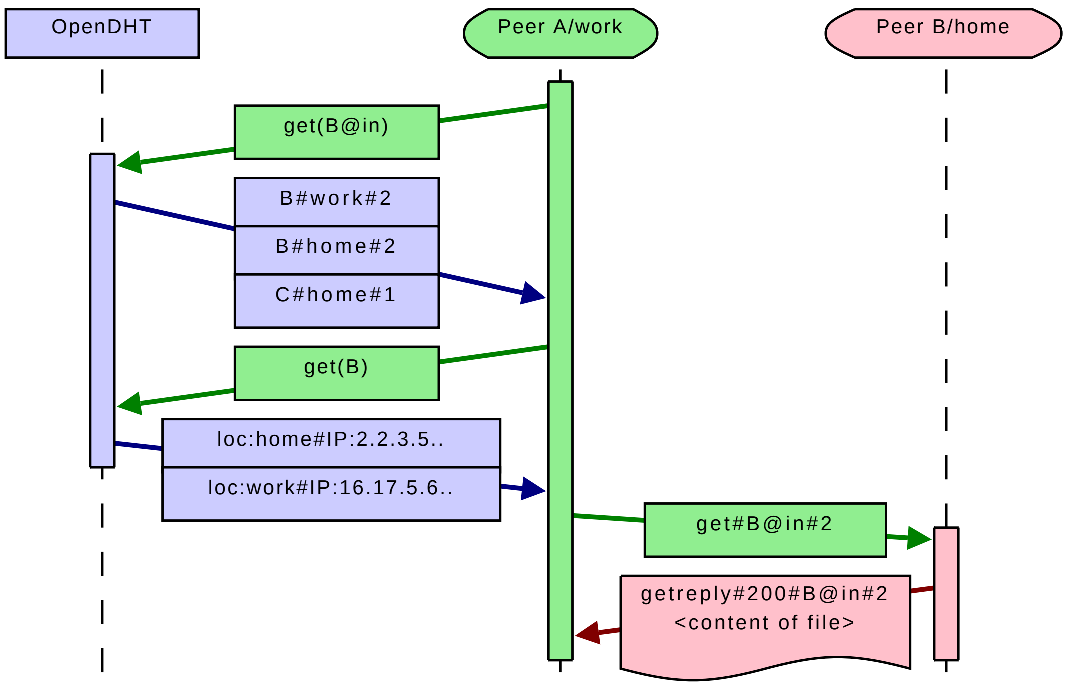


Figure 3.2: Example: peer A wants to get the file B@in. (Pseudo code)

3.7 Log in/log out

In this section we explain how user sessions are handled and how the log in and log out procedures work and are implemented.

3.7.1 Handling user sessions

Questions, that have to be answered, include:
What should be done if a peer stays online at his home PC, goes out without logoff and wants to connect from somewhere else?
How does PeerSON know at which location the peer is right now, e.g., for private messages?
These two questions are partly addressed by the already mentioned location field. That field helps first to distinguish different machines of a user. But when a peer wants to communicate directly, how will it know to which location to connect? And also, a user leaving PC 1 and moving over to PC 2 makes the user unavailable on PC 1 but not the files there, if the machine is still online and the application running. To solve these situations we introduced the following procedure.
A peer can have three states, *active, online, and offline.* Each time a user starts the application on a machine, all values related to one GUID are changed, so that each peer can always find out to where he can connect to.

3.7.2 The protocol

Before the actual login, the application sends a request to OpenDHT to get all values related to the GUID that is to be logged in. If one of those values contains the state active, the state for this values is changed to online. After that the value for the current location is built containing the state active. When the user changes machines again, the whole procedure has to start again.
When all values have the right state, the application sends each key-value pair for all locations to OpenDHT, and the old values are removed.
Because of the *time-to-live* property of OpenDHT key-value pairs, we added to each value related to GUIDs a time stamp to keep also same values different. When later searched for a GUID at a certain location, always the biggest number in the time stamp-field is used. Usually using time stamps can lead to problems, as machines need to be kept in sync. But we do not rely on exact time stamps, we use this only to avoid the *time-to-live* problem when removing and putting again the same value. When we need to compare the time-stamps, we just sort by the size of a number originating from the same PC, and one machine should usually stay in synch with itself.
This can only cause problems if the clock of one machine wraps backwards. But that is implicitly taken care of by removing outdated values in the login procedure.

This procedure generates a lot of messages, and also redundant messages, as the values with state *offline* are also sent again. This does not matter since we need to refresh the DHT entries from time to time anyway. An example of a complete login procedure is given in Figure 3.3.

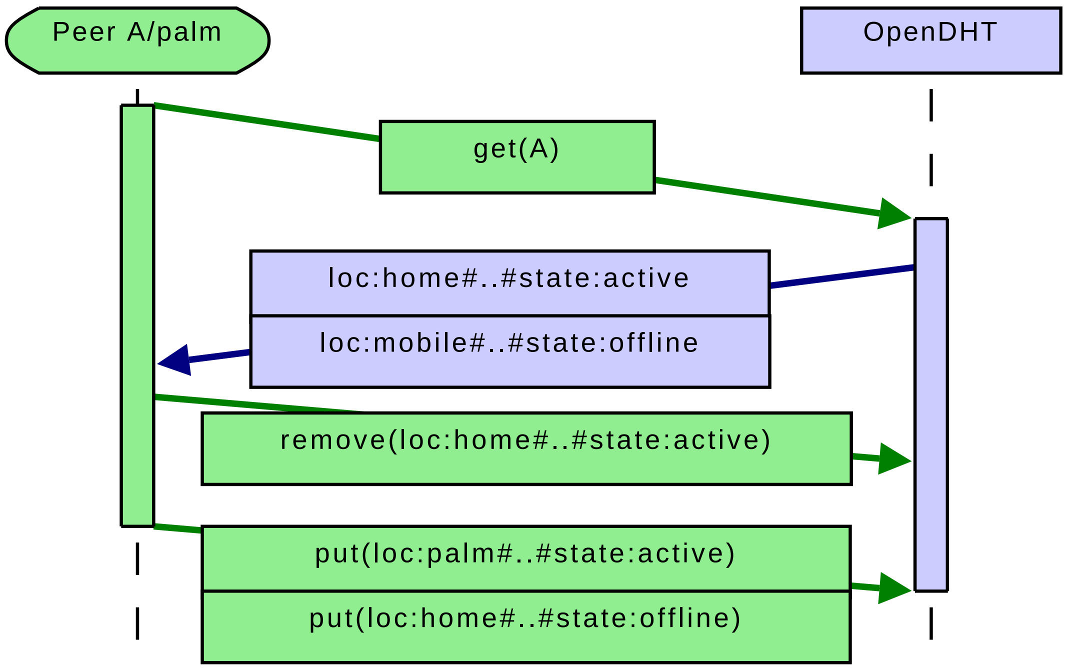

Peer A/palm
OpenDHT
get(A)
loc:home#..#state:active
loc:mobile#..#state:offline
remove(loc:home#..#state:active)
put(loc:palm#..#state:active)
put(loc:home#..#state:offline)

Figure 3.3: Example for the login procedure of a peer with GUID A on location palm

The example is without the options of OpenDHT such as *time-to-live* value for put operations.

3.7.3 Format of key value pairs

This part of the protocol is again between the peers and OpenDHT. The syntax for the login procedure is as follows:

key(⟨GUID⟩)	::=	⟨hGUID⟩
value(⟨GUID⟩)	::=	locator:⟨locator⟩#IP:⟨IP⟩#port:⟨port⟩#State:⟨state⟩-#meshID:[⟨meshID⟩]#GPS:[⟨GPS⟩]#⟨timestamp⟩
locator	::=	STRING
IP	::=	⟨IP address⟩
port	::=	⟨port⟩
state	::=	active|online|offline
meshID	::=	STRING
GPS	::=	STRING
timestamp	::=	INT

3.7.4 Problems when leaving suddenly

At the moment, we do not implement any solution for peers leaving without following the logout protocol, e.g., just killing the process or unplugging a cable. But we provide two possible solutions. The actual issue when leaving without the logout is, that the DHT gets out of sync; it cannot deliver the right information anymore.

Code	Description
locator	A string defined by the user, e.g., home or laptop, to tell on which machine the user is currently active.
IP	The IP address of the machine the user is currently active on.
port	The port on which incoming connections are accepted.
state	A peerID can be in one of three states. Only one peerID can be active at a time.
meshID	Optional. If a user is participating in a city mesh project, e.g., Berlin Freifunk, he can announce how he reachable within the mesh.
GPS	Optional. In this field, a user can put his GPS coordinates to support coordinate based P2P neighbour selection.
timestamp	Time in seconds since 01.01.1970. This field is used to work around the problems with *time-to-live* fields in OpenDHT and to find out which value is the newest if there is more than one possibility.

Table 3.3: Fields of the login value

We propose two solutions to get rid of wrong information in the DHT as soon as possible:
The first variant is that a peer that notices that he got wrong information, sends the right key-value pair to the DHT in order to set the state to *offline*. The problem with that solution is that it would be a security issue. As said, OpenDHT allows to add to every put request a secret that is necessary to delete the value. This is special in OpenDHT and it does not avoid that someone puts a slightly different value for another peer, e.g. just a newer time stamp that would make all older values obsolete. As mentioned before, in this work we do not take care of security implementations and assume that all peers are well behaved, but allowing everyone to change the state or other connection data of other peers is nevertheless a serious issue.
The second method we have in mind, is to set the *time-to-live* of a peer's contact key-value pair to a very short time, e.g. five minutes and then do a refresh short before the value times out. This would keep the system consistent in range of five minutes, but would destroy the possibility to see that a peer exists, but is offline at the moment. In that case, a peer would only find another peer if the other peer is at least online.

3.8 Synchronous messages: chat

Chat or instant messaging is a popular feature not only in platforms like Facebook, but also in web forums, and other web sites like web radio. Some provide their own within the service, others link to instant messaging services or IRC channels.
Instant messaging or chat is only possible if both partners of a communication are online at the same time. So this is in terms of protocols one of the most easy tasks, peer A asks the DHT if peer B is online, if so peer A starts a conversation with

B. We did not implement a chat or instant functionality for the prototype, because it cannot easily be tested with an automated simulation and it is an easy task to extend the implementation with that functionality, but would enlarge the code to handle without providing a use for the testing phase.

3.9 Searching new friends

One of the strengths of services like Facebook or StudiVZ is, that they have a search functionality to find old class mates, people from the same university, or with similar interests. Such a feature is difficult to handle in a system that has the goal to save the privacy of its users. To search for another person by a key word, e.g. get a list of all people studying the same subject at the same university, assumes that all the persons appearing on the list agree to make their GUID, university and their study field public. In our prototype, we allow only exact matches, so people need to be friends before searching each other in our system, at least they need to know each other's GUID.

A protocol that provides a search functionality but respects everyone's privacy preferences would look as follows.
We already have a fine-grained file structure with one single file per profile field, e.g. name, birthday, or city of birth. Our users can set the access rights for each of those files differently, they can also decide to make parts of their profile public. Using that, we can introduce a third kind of key-value pair, that contains as keys keyword categories like name, city of birth, and so on. The value then is, respectively, the name of the file that contains the information. As each filename is in the form of ⟨GUID@⟩⟨file abbreviation⟩ there will be for each peer a different value for a key, so that also same names do not lead to a collision as long as there is no hash collision.

3.10 Asynchronous messages: Wall, mail

As seen in Section 3.8, instant messaging in the context of P2P is quite a simple problem, as all communication works only if both partners of a conversation are online at the same time, which is a synchronous communication. In Facebook, StudiVZ, and other social network platforms, it is a feature to be able to leave messages for other users that are offline at that moment, a kind of e-mails. Facebook provides a message-leaving system that is similar to a guest book and is there called wall. These kind of messages work all asynchronous. We use the term wall as a reference for all kinds of asynchronous messages.

In this section, we discuss why the wall is, from the protocol point of view, the hardest problem in a P2P system – along with the bootstrapping problem – , give some possibilities on how to solve and explain what we implemented and why.

3.10.1 Why asynchronous messages are the hardest problem

We use file sharing as an example to explain why it is such a difficult problem to provide the possibility of asynchronous messages in a P2P system. In a file sharing system, when a peer asks the network for a file, he gets as answer all the other peers that have that file, but only the peers that are online right now. The network does not know anything about peers that also have the file but are offline, so a peer can always only communicate with other online peers.

In the case of asynchronous messages, a peer wants to leave at least the message in the system that he wants to send a message for some other peer that is currently offline. In the worst case, both peers are never online at the same time, he even wants to leave the message itself somewhere, that the other peer can pick it up. Hence we need to add a memory – not storage – to our P2P system.

3.10.2 Possibilities to solve that problem

There are some possibilities for dealing with that issue, depending on what the underlying infrastructure is.
First, we could make other peers responsible for that. If a user is offline at the moment there is a message for him, a set of other peers is checked if one of them is online. Every peer found being online that has been made responsible for keeping messages for this certain user then keeps the message. This set of peers is called delegates. When the addressed peer comes online again, it asks each of its delegates if there are new messages. The disadvantage is that we do not know how many delegates we need to make sure that, first, at least one of them is always online to take over a message and second, there is also always at least one of them online to deliver the message. To make that system more reliable, the delegates should possibly exchange all messages they get for their peer. To realize this protocol, we need some data about the behavior of the peer, to find out how many delegates we need and which peers are good candidates for being delegates.

The second possibility is to divide the work between delegates and the DHT. If a message shall be sent, the sender notifies the DHT, that means he sends a key-value pair with a certain key that can be asked for and identified by the receiver when he comes online again. The value tells what the name of the message file is and which peers have the file. The improvement here is that the peer at least knows that there is a message and can try to catch the peers storing the messages.

The third possibility addresses the case that we build our own smart DHT. Then we can introduce an additional functionality that the DHT collects all arriving asynchronous messages. The smart DHT will also notice when the receiver of the messages comes online again and can then deliver all messages. In that case we only would have to deal with the size of messages and limited storage on super-peers. But for the peers it would feel like a regular mail server.

3.10.3 What we implemented

The questions that is left is how many super-peers we will need to make the lookup service scalable. To get an idea about that, we will need to get more experience with the whole system and it is not relevant for the description how such a system should work in theory. Working with OpenDHT means we do not have a smart DHT that we can change. So we implemented a slightly different version of the third possibility

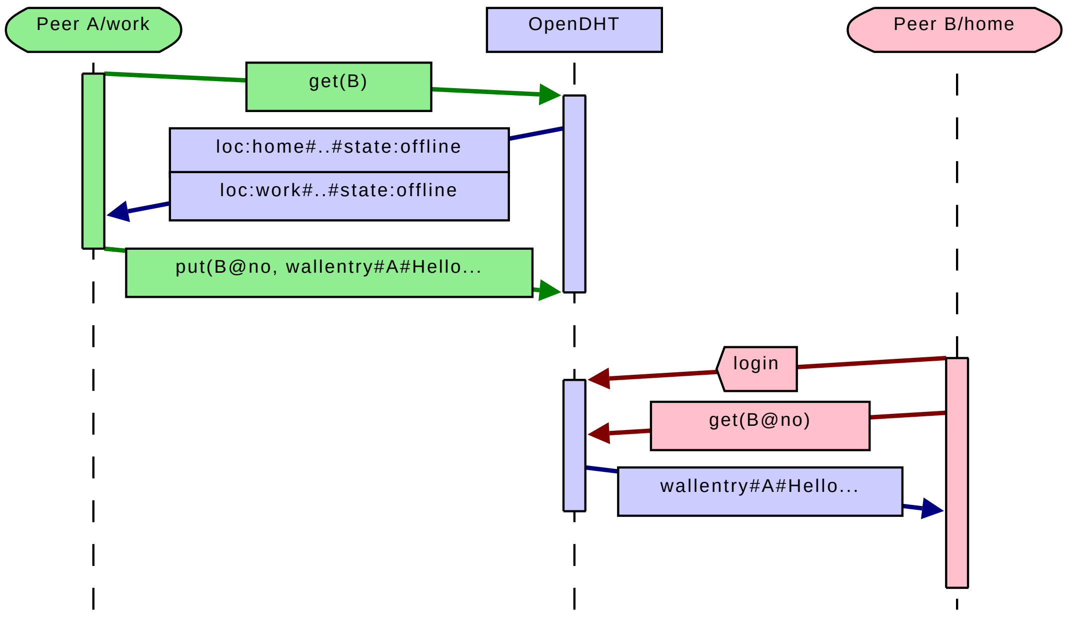

Peer A/work
OpenDHT
Peer B/home
get(B)
loc:home#..#state:offline
loc:work#..#state:offline
put(B@no, wallentry#A#Hello...
login
get(B@no)
wallentry#A#Hello...

Figure 3.4: Example: A wants to send a wall message to B while B has no active location

mentioned. When peer A wants to send peer B a wall entry, but B is offline, it sends a certain key-value pair to OpenDHT. OpenDHT does not know anything about the state of B, it just stores key value pairs. So when B does the login procedure it also queries for a specific key that is used to store all the messages arrived since his last login and after that B removes these values to avoid getting messages twice.

Because of the limitations in OpenDHT for the size of values, we have to limit all asynchronous messages to pure text with fewer than 800 letters. To keep the prototype more simple we limit all wall entries to 800 letters.

3.10.4 Format of the messages

To notify about asynchronous messages, we use the following key value syntax, to send the information from the peers to OpenDHT.

key(⟨notification⟩)	::=	⟨hGUID(receiver)⟩@no
value(⟨notification⟩)	::=	wallentry#⟨GUID(sender)⟩#⟨wall entry content⟩

Figure 3.4 shows an example for the operation to send a wall message. In that example B is online at one location, but sending data to a peer is only allowed when the user is there to notice that data is sent to his computer. Because of that, A cannot directly send the message and will leave it in the DHT. The next time B finishes the login procedure he will ask at last the DHT for notifications and will receive A's message from the DHT. This example is again without the OpenDHT options.

4 Evaluation

This chapter describes the setup, parameters, and values, which are necessary to prove that our system works properly and is scalable.
We distribute our application on PlanetLab to do the experiments there. Beside an additional question on the lookup DHT, we discuss three major questions, first what we need to measure to know if our system works and scales, second how to simulate real users, and third how to simulate their behavior.

4.1 The experiment setup

PlanetLab is an open research platform. It is a set of currently 932 nodes distributed over the world and connected to each other. It is possible to allocate resources on these nodes in the form of virtual machines to implement, install, and run any experiment. Because of the virtualisation, it is not a problem that OpenDHT as well is deployed on PlanetLab. The only influence will be that if we let our experiment run on a node that also runs OpenDHT, we may get faster answers to requests. But the time-consuming part in the communication with OpenDHT is not the connection to an OpenDHT node, but as in any DHT the search process of data among all DHT nodes. As such it is very unlikely to see an actual improvement in the speed of answers.

The idea is to create a social network from a small graph, see chapter 4.3, and for each node of the graph a standardized profile, a set of user files including the application. Then we generate for each node an event log, a text file that contains all the events a user can do, like log in, find the newest files of a friend, and so on. The generation of these events is at the moment randomized on the basis of an uniform distribution, as we still do not have appropriate knowledge about the user behavior in social networks. We plan to determine the right distributions by studying trace data of social networks in future work. To execute the events we have a parser for the event logs that is the same for all users. This strategy produces a lot of small event logs that can be kept locally for later reproduction of each experiment.

The whole experiment has several steps. The first one is an experiment with ten nodes in a complete graph – everyone knows everyone – running locally on one machine to get a first insight. After that the ten node graph tested on ten manually chosen PlanetLab nodes. Then we do a first setup with a small social graph. All nodes are distributed on PlanetLab and simulate a small network of users of our system.

To approach reality, we decided to test some extreme cases in addition to experiments with real data. First, we use a setup in which all peers are always online and exchange data in a randomized way. The other extreme is a setup in which the online/offline state of users is also randomized. Both scenarios are far from reality, but give an insight for best case and worst case behavior of the system. This allows us to later put the results with real or estimated behavior data in relation to these two extremes.

Unfortunately, OpenDHT has been very unreliable for a few months, so it was not possible to execute our test setup until now.

4.2 What to measure to know if the ideas work

To know if our system works, we first need to know what to measure and what to compare the results with. This section discusses which values can be gathered and what that tells us about our protocol and its implementation. In general we use Facebook as a reference, especially on everything related to time, e.g., the time until the profile of a peer is available compared to the time it takes on the Facebook website to open a user's profile.

4.2.1 Latency

Another indication for how good our system is, is the time it takes to find files, peers, and so on, compared to the time it takes to, e.g., open a profile of a friend in Facebook. For example, we measure how long it takes to search a file and finally get it. This time contains two components. First, the answer time of the used DHT, which is also depending on the load on the DHT and the general search speed of the DHT. Second, the time the implementation needs to process each answer. The expectation is that our system will be slower than Facebook. But the question is if the difference is small enough that it is still usable.

4.2.2 Get the right answers from the system

Getting the right answers means that other peers and the location of files can be found. To find a peer is to get the right value from the DHT, to find a file is to get the right information about peers having the file, their location, and the version that exists at that location. Furthermore, it means that if a wall entry shall be sent, it is sent directly to the peer if the peer is active and sent to the DHT in case there is no active peer. These things concerning the actual functionality are already done manually. We started three instances of the program on the same machine and tried out by hand if it works and right results are given output, e.g., if a file exists after requesting a get operation. These tests were done several times while OpenDHT was still operable.

4.3 How to simulate real users

We work with a real system, not a simulation, but we do not have real users by now, so we need to simulate the users and their behavior. This section describes how this can be done and what we have already achieved.

4.3.1 The social graph

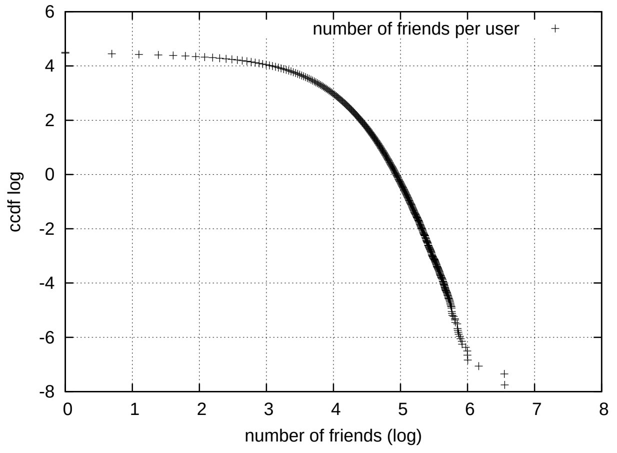
To simulate users, we first need the users and their friends. That can be described by an undirected graph. The nodes are the users, each edge corresponds to the friendship relation between two users.

There exist several theoretical approaches to generate so called social graphs, one of them for example is the small world graph algorithm developed by Jon Kleinberg [KLEINBERG]. But no matter how good the theory is, such an artificial graph is just not a real social graph as Facebook or other platforms produce naturally.

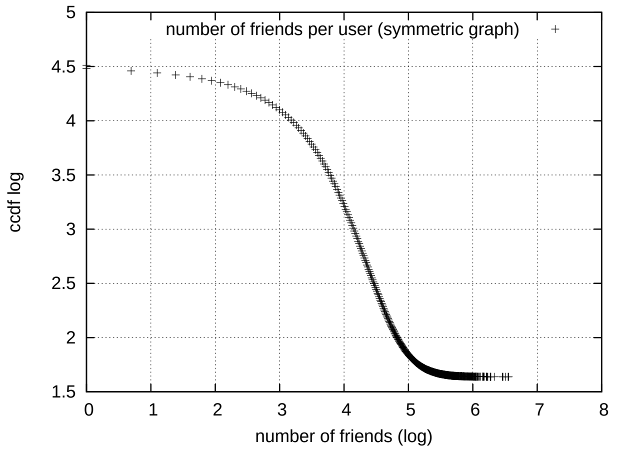
The problem is to get access to this data. The data is owned by the company that owns the platform. This data has a high economical value and the companies protect it. That even led to discussions about wether the user who put his data in the platform still owns the data and is allowed to extract it again. There are several applications to extract ones own data from Facebook or other social networks to transfer it to another platform. Most platforms have now a clause in the user contract that forbids to use such applications. A case that became public was [SCOB].

StudiVZ is the German equivalent to Facebook, has similar features, and even looks similar. It is a platform for students and very successful.
In 2006, Hagen Fritsch, a student from TU Munich started a project with the goal to prove that the platform has some privacy issues and wrote a program that is called a web crawler. He was able to extract a huge part – 1.074.574 profiles – of all user data from StudiVZ, namely all profile data, that the users gave the status public to and also some not-public user profiles. In StudiVZ, public means that everyone who is a member of StudiVZ can access the data. Hagen Fritsch published the analysis results of the data [FRITSCH], there can also the exact number of crawled profiles can be found. This data is anonymized available for research, so we got a graph with around 450,000 nodes. Each node is encoded with an integer, so that we cannot derive any personal data from it. We analyzed this graph and found that the number of friends per user seems to be heavy tailed, see Figure 4.1(a). The graph we got is directed and because of that the friendschip relations are asymetric – node A having an edge to node B does not mean, that B also has an edge to A. In StudiVZ the friendship relation is always symmetric, either two users are friends or not but having a friend which a user is not the friend of is not possible. Because of that, we changed the graph and made it undirected in the way that we added a node A to the friend list of another node B if B was on the friend list of A. In the view from a graph theortic way we made of each directed edge an undirected. Figure 4.1(b) shows that this does not change the properties of the graph that are important for us, the average number of friends stays stable compared to the original graph. From here on we are only working with the symmetric version of the graph.

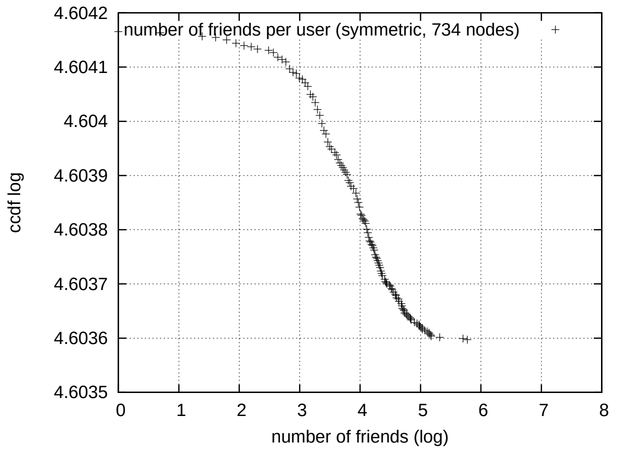
Our interest is to find out how many friends users have and how strongly the graph is connected. How many nodes can be deleted before the graph becomes unconnected?
We found an answer to this when we tried to fit the graph to our test setup. Of course 450,000 nodes are not manageable in a test setup, so we tried to reduce the number of nodes to fewer than 500 but keep the diameter as big as possible, at least more than five. The result is that it is not possible. Finally, we chose a subset of



(a) CCDF of the friendship graph of the StudiVZ data



(b) CCDF of the friendship graph of the StudiVZ data – made symmetric



(c) CCDF of the reduced symmetric friendship graph one node with degree three and followed its friends to a recursion depth of two, that means including the three friend nodes and their friends and so on. This leads to a diameter of only four, but with this one start node the resulting graph had more than 700 nodes. Setting the recursion depth to three to get a diameter of six already results in almost the whole original graph.

How real is the small graph derived from the StudiVZ graph?
The original data is itself only a subset of the real graph, as the crawler did not get all the data and the graph is from 2006 and it is very likely that the data has changed a lot since then. Furthermore, we can produce the same plot, see Figure 4.1(c) of the small graph as we produced for the full graph. It seems to be similar to the symmetric and also to the original asymmetric graph, see Figure 4.1(b) and Figure 4.1(a) but 734 nodes are not enough to be able to give a complete answer. The small graph is, compared to the full graph, not significant in size.

Though it is not clear how good our test graph really is, it is a good enough start for experiment setups.

4.3.2 Online times and actions

Having a graph that can be used to simulate the users is very helpful. But to find out if the system is scalable and well behaved, the users must do something with the system. So we need to simulate not only the users but also their behavior. After searching through papers that analyze values like session length and churn we found that most of them analyze Bittorrent networks or Flickr or something else. But we did not find any paper on the session length and churn in Facbook or StudiVZ. Many of the papers we found give some numbers, but do not focus on the exact data we need.

Fortunately, we have access to an anonymized data set containing information about behavior of real users in Facebook. As this data has been collected at a PoP of a large ISP it spans over a significant customer population to allow representative statements. These insights are in the progress of publication and will be available later on. For further information on this, we refer the reader to [FSCH]. We will be able to extract from this data the parameters for our test setup, e.g., how long users stay online, what is the churn in OSNs, how often users are asking for the profiles of their friends, etc.

4.3.3 How many super-peers are needed?

An additional question that comes up is, how do we ensure we have enough super-peers? And how many are enough?
We cannot ensure that. We assume, that every user can decide to be a super-peer or not. So, if just not enough users become super-peers (in relation to normal users) the system might crash because of too few super-peers handling too many user requests. To find out how many super-peers we need in relation to users, we need to get more experience with the system.

5 Related work

There are many systems and platforms that provide similar or features related to our work. In this chapter we briefly introduce them and explain how they differ from our approach.

5.1 E-post

E-Post is a P2P infrastructure for e-mail. It has some similarities with our approach, but a different purpose. E-Post is based on Pastry, and the principle is that every participant of the system provides some storage to be a part of a distributed mail server. The messages are stored within the DHT until they are delivered.

There are some differences to our system. First, E-Post works without super-peers, so every peer needs to provide all features. Second, E-Post is in some way only a part of what we do, it is so to say the wall, as peers cannot create profiles and search other peers' profiles. But that is not the goal of E-Post. E-Post is still a very active project and it is also productive, so best way to get more information about this is to have a look on their website [EPOST][EP].

5.2 Jabber

Jabber is a server based instant messaging system, using the XMPP protocol. Jabber gave us some inspiration for our system, although it has not much in common with our work as it is a centralized instant messaging system The idea to support different locations per user comes from that protocol, as in Jabber this is also supported. The difference is, that the XMPP protocol works with priorities on the locations, that the user has to set manually for each peer instance. That means that if a user forgets to change the priorities on one location and then leaves for another, messages for the user may be delivered to the old location. Because of that the message may be received too late, e.g., the message that there is a party in one hour. This is the reason we try a different approach building our protocol with locations.
For further information see [XMPP]

5.3 Publish-subscribe systems

To notify others about news, there exist publish-subscribe systems, e.g., a user subscribes to a group and then gets automatically a message every time there is news in this group, similar to news feeds for websites.

At a first glance, those systems look like the right approach for our asynchronous message notification. But there is a big difference in the logical structure. Publish-subscribe systems work with a one-to-many-relation, many users get messages from

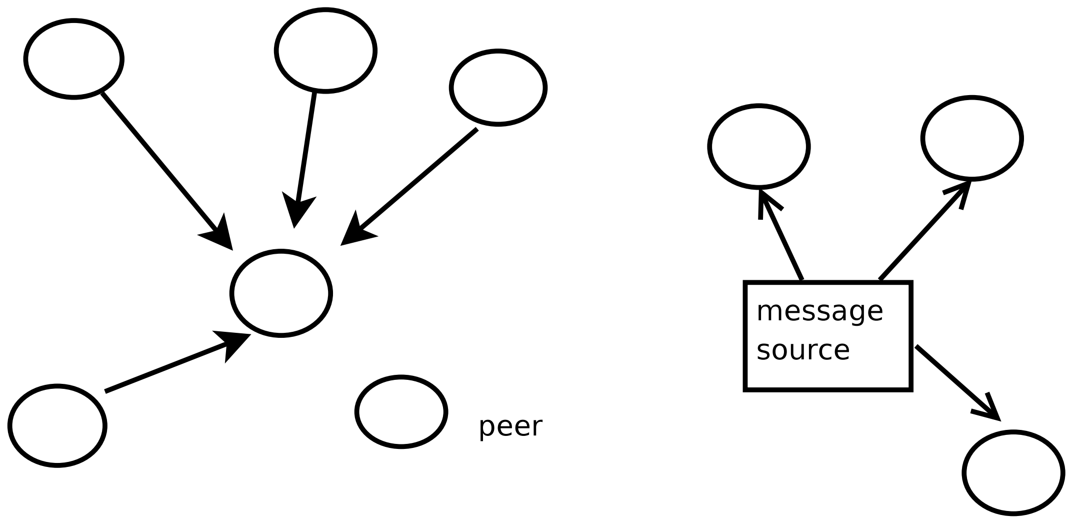


Figure 5.1: a) one peer receives messages from many other peers. b) many clients subscribe to a source of messages, e.g., news

one source, which they subscribe themselves to, see Figure 5.1 b), whereas we need a many-to-one-relation. Each peer can get messages from many other peers, an example for one peer getting wall entries from other peers is given in Figure 5.1 a). So if we try to use a publish-subscribe system, a peer would have to subscribe to all other peers to be able to receive messages from them and each peer would be a subscribe source that has to be managed and maintained in the system. Speaking in the example of a news group, each peer is then a news group and each peer would need to subscribe at least to all peers from whom he accepts wall entries from.

A publish-subscribe system will become interesting for us, when we add groups to our system. Groups in, e.g., StudiVZ can be opened by all users and other users can become members of a group. These groups work in a way similar to a web forum.

5.4 Master thesis about the encryption system

As mentioned before in this work we do not handle any encryption or security issues. A protocol to that was designed by Youssef Afify [ENC], who especially addresses the problem of access control.

5.5 Hierarchical DHTs

There exist different approaches for hierarchically organized P2P systems. First of all, Gnutella has been working since years with a super-peer concept, but it is still an unstructured P2P system. Hierarchical systems including DHTs, such as Chordella [HDHT] are in many cases designed to improve load balancing of DHTs. They do not intend to divide the whole system into two independent parts and use the super peers as lookup service. We could choose to use Chordella instead of OpenDHT but we still would need to extend it using with non-super-peers as clients.

5.6 DHTs as lookup service

A DHT is traditionally always a lookup system, as it is an architecture built with the goal to efficiently find data and content within a P2P network. In contrast, we use the DHT as a super-peer system, to store and look up meta-data, but store the content outside of the DHT. There are some approaches to use DHTs as lookup services, e.g., to improve routing in the Internet, such as scaling down the size of routing tables. For example there are projects at TU Berlin which will be published within the next months, for more information refer to Anja Feldmann [AF] and Luigi Iannone [LI]. These systems usually design very specific DHTs to fit their needs, similar to how we design a specific OSN-DHT for our approach. So we cannot use the systems that were designed in other projects as they will very probably not be able to use our infrastructure. Basic DHT implementations are often meant to be the underlying system. To give them a certain functionality, there must be implemented an application on top of it. In the cases where a common, unchanged DHT is used to implement the desired functionality on top of it, there is still the difference in the application.

6 Conclusions and outlook

In this Chapter we summarize what we did within this thesis and give a brief outlook on how the work can be continued.

6.1 Summary

In this work we developed a P2P infrastructure for an online social network, based on two concepts, a DHT built by super-peers and the peers that use the DHT as a lookup service.

We gave an introduction to the background of our work and especially for the design descisions we had to make. Based on that, we designed a detailed protocol on top of the Bamboo DHT, more specifically for the deployment of it, OpenDHT, and gave an overview over other possibilities to design a P2P system for social networks.

We implemented an application that provides the key features of an online social network and that uses OpenDHT as a lookup service. This includes a protocol to leave messages for other peers in the DHT, and one to announce a peer's connection data and which content a peer has stored.

We analyzed a subset of the social graph of StudiVZ to derive the simulation of users from it for evaluating our system and described our test setup in which we want to evaluate the system.

6.2 Future work

In this thesis we implemented a first prototype to realize our project. This project consists of many parts, which goes beyond the scope of this thesis. First of all we need to complete the setup for the evaluation of our system. The scripts are prepared, but we still do not have enough knowledge about user behavior to simulate this behavior realistically.

Since OpenDHT seems to be gone, we need to deploy our own Bamboo testbed. Before that, we plan to use a single process to simulate the existence of OpenDHT on a single PC. Furthermore, the application needs to be extended by a graphical user interface to make its usage more comfortable. When the first evaluation phase shows a good behavior and we are sure that we can stick with our protocol, the application needs to be extended by more features, e.g., a chat functionality.

Depending on the results of the evaluation, it is even possible to bring this project to the real world and offer a privacy saving alternative for Facebook, Orkut, StudiVZ, etc.

Acknowledgements

First of all others, thanks to Sonja Buchegger, as she had the idea to create a P2P OSN project and gave me the chance to participate in that. She and Anja Feldmann, my two advisors, gave me a great support during the last months, with hints, questions, asking the right questions, coming up with helpful ideas, linking to the right people, making coffee, some pressure at the right time, and a lot of other things.
Further thanks and greetings to the other members of this project in Singapur and Switzerland, Anwitamann Datta and Le Hung Vu, for their hints and questions, especially during the development phase. Also thanks to Thomas Fuhrmann for his insights and explanations.
Thanks to Bernhard Ager who had the idea to hash GUIDs to avoid problems with OpenDHT, which also brought yet some privacy in our system.
Another one is Fabian Schneider who gives me access to important data and, moreover, is always a source of energy as he always brings back the smiling, even when the whole world tries to prevent one from finishing a thesis.
Special thanks go to Harald Schiöberg. He helped me to domesticate Latex, hunting bugs, and getting along with dia. But, much more than that, he was there to support me all the time, reading the first ugly drafts of this thesis, asking mean questions, solacing me, when the thesis seemed to be never ending work, and above all he was just there.
It is not possible to list all the names, as the group around Anja Feldmann grew very fast the last year. But thanks go to the whole INET group. It is a great pleasure to work with such smart, helpful, and friendly people.

Bibliography

[PSON] Sonja Buchegger, Anwitaman Datta: *A Case for P2P Infrastructure for Social Networks - Opportunities & Challenges. In: The Sixth International Conference on Wireless On-demand Network Systems and Services.*; WONS 2009, February 2-4, 2009, Snowbird, Utah, USA.

[IGOR] Bernhard Amann, Benedikt Elser, Yaser Houri, Thomas Fuhrmann: *IgorFs: A Distributed P2P File System. In: Proceedings of the Eigth IEEE International Conference on Peer-to-Peer Computing*; P2P 08, September 8-11, 2008 Aachen, Germany.

[ENC] Youssef Afify: *Access Control in a Peer-to-peer social network. Master thesis*; EPFL, August 15, 2008, Lausanne, Switzerland.

[HDHT] Quirin Hofstätter, Stefan Zöls, Maximilian Michel, Zoran Despotovic, Wolfgang Kellerer: *Chordella - A Hierarchical Peer-to-Peer Overlay Implementation for Heterogeneous, Mobile Environments. In: Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing - Volume 00, pages 75-76* ; IEEE Computer Society, 2008, Washington, DC, USA.

[AF] `www.net.t-labs.tu-berlin.de/~anja`.

[LI] `http://www.net.t-labs.tu-berlin.de/~luigi/`.

[PASTRY] `http://freepastry.org/`

[PAST] A. Rowstron, P. Druschel: *"Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems" In: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pages 329-350*; November 2001, Heidelberg, Germany.

[FSCH] `www.net.t-labs.tu-berlin.de/~fabian`.

[ODHT] `http://www.opendht.org/`.

[OHASH] Brad Karp, Sylvia Ratnasamy, Sean Rhea, and Scott Shenker: *Spurring Adoption of DHTs with OpenHash, a Public DHT Service. In: Proceedings of the 3rd International Workshop on Peer-to-Peer Systems, Springer-Verlag Lecture Notes in Computer Science Hot Topics Series*; IPTPS 2004, February 2004, San Diego, CA, USA.

[BBO] `http://www.bamboo-dht.org/`.

[BAMB] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz: *Handling Churn in a DHT. In: Proceedings of the USENIX Annual Technical Conference*; June 2004.

[BEACON] http://www.efluxmedia.com/news_Facebook_Dragged_into_Class_Action_Lawsuit_over_Beacon_22293.html.

[EPOST] http://www.epostmail.org/.

[EP] Alan Mislove, Ansley Post, Charles Reis, Paul Willmann, Peter Druschel, Dan S. Wallach, Xavier Bonnaire, Pierre Sens, Jean-Michel Busca, Luciana Arantes-Bezerra: *POST: A Secure, Resilient, Cooperative Messaging System. In: Proceedings of the 9th Workshop on Hot Topics in Operating Systems*; HotOS'03, May 2003, Lihue, HI.

[XMPP] http://xmpp.org/.

[FB] http://www.facebook.com/.

[OK] http://www.orkut.com/About.aspx.

[SVZ] http://www.studivz.net/.

[GNU] http://rakjar.de/gnufu/index.php/GnuFU_en.

[CHORD] http://pdos.csail.mit.edu/chord/.

[CHO] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, pages. 149-160*; ACM SIGCOMM 2001, August 2001, San Deigo, CA, USA.

[CHORDCOM] http://pdos.csail.mit.edu/chord/\#downloads.

[PGRID] http://www.p-grid.org/.

[PGR] Karl Aberer, Philippe Cudr-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, Roman Schmidt: *P-Grid: A Self-organizing Structured P2P System*; SIGMOD Record, September 2003.

[BFF] http://berlin.freifunk.net/.

[KLEINBERG] Jon Kleinberg: *The small-world phenomenon: An algorithmic perspective. Proc. 32nd ACM Symposium on Theory of Computing, 2000. Also appears as Cornell Computer Science Technical Report 99-1776 (October 1999).*

[SCOB] http://scobleizer.com/2008/01/03/ive-been-kicked-off-of-facebook/.

[FRITSCH] http://studivz.irgendwo.org/.