

Project Group 2012

A Peer-to-Peer Framework for Social Networks

Documentation

Payman Alavi, Lars Bremer, Deepak Chandra Veena, Jens Janiuk, Liu Kangping, Subramanya Kuduvally Gurumurthy, Ahmad Luqman, Alexander Mäcker, Yevgen Mexin, Alexander Morast, Bahareh Mousavi Jazayeri, Chintan Jayesh Parekh, Anastasia Petkau, Jan Marcel Sander, Hongying Zhao

University of Paderborn
Faculty for Electrical Engineering, Computer Science and Mathematics
Supervisor: Dr.-Ing. Kalman Graffi
April 30, 2013

Contents

1 Introduction **1**

2 Goal Definition **2**
2.1 Current Status of the p2p Framework . . . 2
2.2 Use Cases . . . 3
2.2.1 E-Learning . . . 3
2.2.2 Collaborative Work . . . 4
2.2.3 Social Networking . . . 4
2.3 Plug-ins . . . 4
2.4 GUI Challenges . . . 6
2.5 Framework . . . 7
2.5.1 Storage . . . 7
2.5.2 Security . . . 10
2.5.3 Large File Transfer . . . 11
2.5.4 Search and Cataloguing of Plug-ins . . . 12
2.5.5 Data Structures . . . 12
2.5.6 Summary . . . 13
2.6 Overlay . . . 13
2.6.1 Security . . . 14
2.6.2 Capacity Awareness . . . 16
2.6.3 NAT Traversal Issue . . . 16
2.6.4 Changeable Configuration at Run-Time . . . 17
2.7 Planned Status at the End of the PG . . . 17
2.8 Time Plan . . . 19

3 Plugins **21**
3.1 Groups . . . 21
3.2 Abstract Browser . . . 24
3.2.1 Communication between Java and Javascript . . . 24
3.2.2 jQuery . . . 24
3.2.3 BBCode . . . 24
3.2.4 StringTemplate . . . 24
3.3 Forum . . . 25
3.4 MultiView and Topic Chat . . . 25
3.4.1 Multiview . . . 25
3.4.2 Topic Chat . . . 25
3.5 FileHosting . . . 26

3.6 Voting . . . 27
3.7 FileTransfer . . . 28
3.8 Appstore . . . 28
3.8.1 Requirements . . . 29
3.8.2 Basic idea . . . 29
3.8.3 Implementation . . . 32
3.8.4 Evaluation . . . 35
3.8.5 Appstore Guide . . . 37

4 Graphical User Interface 44
4.1 Perspectives . . . 44
4.2 Futher GUI improvements . . . 44

5 Framework 46
5.1 Distributed Tree . . . 46
5.1.1 Mechanism Description . . . 48
5.1.2 Example . . . 48
5.2 Role Based Access Rights . . . 48
5.2.1 Existing Approach . . . 48
5.2.2 Basic Enhancements . . . 49
5.2.3 Retrieval of Keys . . . 50
5.2.4 Update of Keys . . . 50
5.2.5 Rolemanagement . . . 51
5.3 Write Protection . . . 51
5.3.1 Remote Operations . . . 52
5.3.2 Distributed Set and Distributed List . . . 52
5.3.3 Groups and Roles . . . 55
5.3.4 Evaluation of Distributed List . . . 55
5.4 Voting . . . 58
5.4.1 Public Voting . . . 59
5.4.2 Private Voting . . . 60
5.4.3 Conclusion . . . 60
5.5 AbstractStorageItem - a basic model class . . . 60

6 Overlay 62
6.1 Security . . . 62
6.1.1 Security Threats . . . 62
6.1.2 Related Work . . . 63
6.1.3 Solution and Implementation Strategies . . . 65
6.2 Capacity Awareness . . . 68
6.2.1 Related Work . . . 69
6.2.2 Model . . . 69
6.2.3 Protocol . . . 71
6.2.4 Capacity . . . 73
6.2.5 Heterogeneity Configuration . . . 75

6.2.6 Testing of Heterogeneity . . . 75
6.3 Handling of Mobile Nodes . . . 78
6.3.1 Routing Table . . . 79
6.3.2 Best Routing Candidate Selection . . . 79
6.3.3 Protocol . . . 80
6.3.4 Testing . . . 80
6.3.5 Problems . . . 81

7 Testing **82**
7.1 Unit Testing . . . 82
7.2 Pool Area Testing . . . 83
7.2.1 Test Setup . . . 83
7.2.2 Problems . . . 85
7.2.3 Monitoring and Result . . . 86
7.3 Summary . . . 88

8 Summary and Outlook **89**
8.1 GUI/Plugins . . . 89
8.2 Framework . . . 90
8.3 Overlay . . . 91

A StartLifeSocial Guide **92**
A.1 Eclipse configuration . . . 92
A.2 LifeSocial configuration . . . 92

B Plugin development guide **93**
B.1 Package structure . . . 93
B.1.1 Domain logic . . . 93
B.1.2 User interface . . . 94
B.2 Models . . . 95
B.3 Testing / Commands . . . 99
B.4 Monitoring . . . 100
B.5 Tips . . . 101
B.5.1 Asynchronous programming . . . 101
B.5.2 Providing extension points . . . 101

C Test Guide **102**
C.1 Using the Test Plugin . . . 102
C.2 Tips and Hints . . . 103
C.2.1 Storage Folder Removal . . . 103
C.2.2 Multiple Instances on One Machine . . . 103
C.2.3 Send Friend Request Instability . . . 103
C.2.4 Overloaded System . . . 103

Bibliography **104**

1 Introduction

In times of Web 2.0, online social networks are some of the most visited websites [1]. Especially the fact, that people can find their friends in such a network and communicate with them is the reason of using social networks. Furthermore people want to be up-to-date about the newest trends and the activities of their friends. By that people share their personal data including private pictures and further personal information. On the one hand this is a quite handy way of communication and personalized data sharing, but on the other hand security issues occur. The data, that is uploaded by the user of the social network, could be misused by the provider of the social network for their own purposes, e.g. selling it to other companies. In addition, if the servers of the online social networks are temporarily offline or even permanently down, all the effort of the user maintaining their profiles is not accessable or even lost.

To overcome these problems, the idea of peer-to-peer (P2P) technology came up. With it, no more servers are necessary to create a online social network because all the data that is provided in the social network is stored on the devices of the users. Moreover the bandwidth of the users will be shared in order to provide fast access to the required data in the network.

So, the application LifeSocial is combining the P2P technology with the online social network in order to use the benefits of P2P and provide a more secure and efficient online social network. LifeSocial itself is built on top of a P2P-framework, which is providing common social network functionalities like messaging and sharing pictures. In addition, LifeSocial is made of Eclipse RCP, so that the application is a Rich Client Application and can be extended easily.

Finally online social networks are different and could be used for different specific purposes. In Chapter 2, we will discuss the current state of the P2P-Framework and LifeSocial and we will present different opportunities based on that. We will introduce several use cases for LifeSocial and talk about the different research areas for improving the P2P-framework. In Chapters 3, 4, 5 and **??** we will present our results concerning LifeSocial, the P2P-framework and Pastry. Eventually we will talk about testing in Chapter 7 and give an overview for further research in Chapter 8.

2 Goal Definition

Social networking has received a blooming success in the past decade and so is the need for sharing information. Statistics show that the majority of the Internet traffic is dedicated for the use of social networking. There are many reasons for the success of these networks. Users, for example, have the opportunity to publish their own content, or search for content shared by other users in the social network. Furthermore, they can interact with each other by chatting or Voice over IP (VoIP).

To provide such a social network different factors are important. On the one hand the providers are interested in attracting new users to increase the earnings by involving advertisements into the social network. But with the increased number of users comes a rising bandwidth which leads to high operating costs. Therefore, social network operators aim to reduce the operational costs represented by expensive Internet connections, to be able to solve all the requests with smallest delay, just as the costs for the mass of servers, that have to be provided and maintained. Also another important barrier is that the data is always stored on a server and hence complete removal of content is not guaranteed.

Peer-to-peer (p2p) systems have grown simultaneously. P2p basically offers direct communication with all the machines and hence remove the central dependencies. P2p networks like Napster, Gnutella, Skype, and many more have proved the potential of those systems for file sharing and Internet telephony applications. Having a completely distributed p2p system for communication and storage could benefit a social network in many areas.

2.1 Current Status of the p2p Framework

To overcome the problems mentioned above, LifeSocial was developed. Since 2007 Kalman Graffi and students at the Technical University of Darmstadt and the University of Paderborn have worked on this application with the goal to create a completely distributed social network. To achieve this goal of complete distribution, the decision was made to implement the online social network on top of a p2p framework.

FreePastry reference

Yet LifeSocial is able to provide the basic functionality for social networking. The current framework is based on the FreePastry overlay. It is a quite mature overlay now, but there are still some unresolved security issues and it does not support heterogeneity, i.e. the participation of devices with different resources. On the other hand, the current GUI of LifeSocial does not offer the usability a social network requires. Also, there are several further use cases which can be considered in the framework.

The current software architecture consists of three layers where each layer communicates with the other through well defined interfaces. The entire architecture is shown in the Figure 2.1. The

p2p overlay layer is at the bottom of the architecture. LifeSocial uses the FreePastry for this layer. In FreePastry the users and objects are mapped into a common identifier space. Therefore, two metrics are used in FreePastry: the ID distance and the physical distance. The overlay layer connects the nodes and allows routing functionality among them.

The p2p framework layer is located on top of the overlay. It provides p2p mechanisms that allow developers to create exciting applications. Among those mechanisms are data structures that allow to store data in Lists, Sets, or in a Prefix Hash Tree. Additional functionality for streaming and publishing/subscribing is already provided.

Refer to the existing papers about the LS architecture (though it might have changed a bit)

The framework layer manages the communication between the plug-ins and provides security for the data storage such as messaging over the p2p network. It enables authentication of peers and protection of the integrity of delivering messages by applying a public key infrastructure. In addition, the framework layer offers a global aggregation mechanism that aggregates locally collected attributes over the whole network. This aggregation is very useful to monitor the network.

Refer to the existing papers about the LS security (though it might have changed a bit)

OSGI reference

The LifeSocial application is at the top of the framework and it consists of several OSGI bundles (in the following also referred to as *plugins*. There are some basic bundles which contain the major domain-logic functions. Additional functionality and the UI are contained in further different bundles. The UI code is thereby seperated from the domain logic code. The decision for using OSGI was made by the previous project group to overcome the restrictions due to the lack of modularization support in Java.

2.2 Use Cases

To improve the functionality of LifeSocial, the PG discussed different use cases in which the system might be used. The functionality required by them can be split into different plug-ins. In the following we first describe possible use cases for the social network and identify additional plug-ins that might be useful for these purposes. Afterwards some further challenges and tasks concerning the GUI are listed.

2.2.1 E-Learning

This use case is to support students at school or university by providing a collaborative learning environment. This includes communication, writing, and drawing together and sharing files. As a basis we propose to enhance the functionality of groups with user management and access rights to be able to distinguish between different roles like teachers, students and administrators. As useful plug-ins we identified file sharing, permanent file storage, wikis, forums, group text/audio/video chat, and collaborative editors (text/whiteboard).

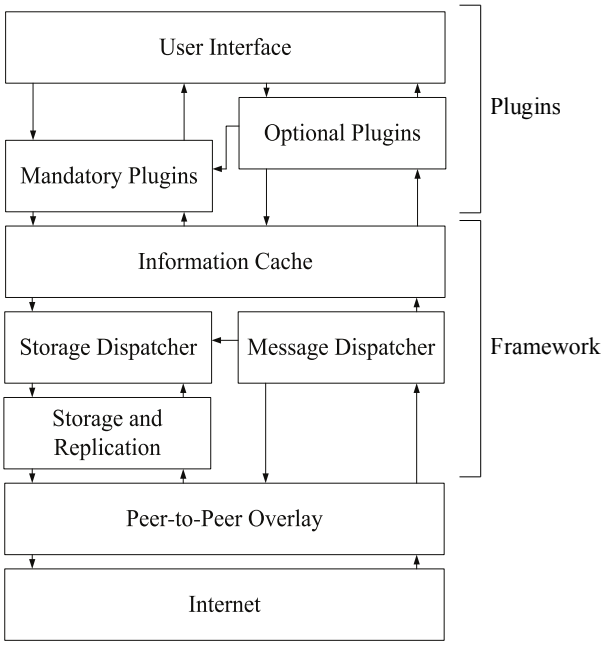


Figure 2.1: Layered Architecture

2.2.2 Collaborative Work

The e-learning use case can be adapted to a collaborative working environment with a similar functionality. In a company's internal network each department could have an own collaborative working group to support the communication between the co-workers. In addition to the plug-ins used for e-learning we propose a calendar.

2.2.3 Social Networking

This use case presents the traditional intention of social networks. It includes connecting with friends, colleagues, or a customized group of people. In comparison to the previous described use cases it is moreover meant to be used for personal and non work-related tasks like sharing profiles, photos, and participating in private or public discussions. Additional plug-ins could be a friend graph, a gallery, and an activity wall.

2.3 Plug-ins

To support the above use cases the functionality has to be implemented as plug-ins which will require to also add additional functionality to the framework. Afterwards, these plug-ins can be graphically arranged to propose a perspective for each use case for the users (Section 4).

In the following we present a list of plug-ins we propose to implement. In the actual implementation each plug-in can be divided into multiple OSGI plug-ins for the functionality and the GUI.

1. *Enhanced groups:* We propose to enhance the existing group plugin to improve the communication and working between group members. The groups can be either public, which means every user can join the group, or private, which means that the group is restricted to specific members. Inside the group, the users can use additional functionality that is provided by other plug-ins. For structuring this functionality in a group, we propose a tree menu which organizes all contents. This can include files, collaborative editors, forums, wikis, etc. Furthermore, different access rights have to be introduced which can be given to the group members. The creator of the group is initially the only administrator and can permit any permissions to users using a role-based permission system.

2. *App store*: An online repository can be used to search, download and publish additional plug-ins. By supporting third-party plug-ins, we intend to get a lot of users to develop new functionalities and hence get a huge variety of plug-ins.

3. *Group chat*: Enhance the existing chat plug-in to support multi-user chats.

4. *Activity wall*: The activity wall gives an overview over the activities in the social network of a particular group of people. When a user starts LifeSocial it would be useful to show a list of activities of all of his contacts and groups. In a group, the activity wall should only contain the activities of the group members. Concrete activities can be provided by other plug-ins. As an example, the forum could inform about new topics and the profile plugin could inform about changes that were made. Also the text messages, that are currently implemented as the *wall* should be included in the activity wall.

5. *File sharing*: One possibility is to share local files directly similar to traditional file sharing applications like BitTorrent. These files are provided by the original publisher and every user who has downloaded it. In addition to the traditional approach, the plugin must provide some encryption-based mechanisms to restrict the access to some users, like friends and family.

6. *Distributed file storage*: The second idea, which is probably more applicable to small files, is to store the files distributed in the p2p network to make them always available. While the approach of the *File sharing plug-in* can be used for less important purposes like sharing songs or videos with friends and family, this approach is very useful for collaboration. A lecturer who wants to share documents with his students is probably not always online. Though the documents have to be available all the time.

7. *Forum:* A plug-in for thread-based discussions that are public or restricted to group members. For implementing this, a tree data structure will be needed for storing the structure of the discussion.

8. *Live stream:* Live streaming of video and audio. This can be for example used for chatting, conferences and webinars.

9. *Friend as graph network:* Enhancing the existing graph representation of the friend network in terms of performance and functionality. This will mainly concern the GUI but might also require some additional work on the framework for showing more hops, i.e. friends of friends.

10. *Wiki:* Collect knowledge in documents that contain hyperlinks to each other. Additionally it supports versioning, i.e. the storage of all changes that were made to the documents. As storage format we plan to use HTML.

11. *HTML Viewer:* For showing the wiki pages, and in the future probably other content, an HTML viewer is needed. For reusability we will implement it as an own plug-in.

12. *Calendar:* Required to maintain appointments for users. This includes creating events, inviting user to events and automatically assembling the users' calendars. Furthermore, users in a group can share common calendars, whose dates are also shown in the users' personal calendars.

13. *Voting:* Support online polling that can either be public, restricted to friends, or a single group. The creator of a poll can specify a topic, possible answers, a time period in which the votes are allowed and specify which users or groups of users are allowed to vote.

14. *Collaborative text editing:* Online collaboration tool for editing text. This plug-in enables multiple users to simultaneously work on single documents by using a Changeset algorithm that needs to be implemented in the framework.

15. *White board:* Beside a collaborative text editor, a similar functionality for drawings can be implemented. This can be very useful in e-learning and collaborative work for sharing ideas as graphical sketches.

16. *Message Folders:* Currently, LifeSocial does not support the ability for users to manage the messages and organize them into folders. The email plug-in in LifeSocial has only two folders for incoming and outgoing messages. It would be useful for the users to make it possible to customize the structure and the contents of the folders.

This list is meant as an ordered collection of ideas to prioritize the tasks in the development process. However, because of the variety of different possibilities the development needs to focus on the extensibility of LifeSocial. That means, that we choose the plugins to implement, so that we have examples for collaborating plugins. For example, the *forum* could extend the *groups* in a very flexible way. Another reason for choosing plugins to implement, is the research we are doing in the framework and the overlay. These results should have example applications that are related to practice.

2.4 GUI Challenges

Besides the tasks on the GUI that are related to the plug-ins described above, we also face more general challenges. By introducing a huge amount of new plug-ins that can also be extended by the app store, we get confronted with the problem of organizing them inside the GUI. A task for our project group is to investigate how this can be done while maximizing the usability. One way could be to differentiate between built-in plug-ins and third-party plug-ins like it is done in Facebook: The built-in plug-ins have consistent places in the menus while only the latest x used third-party plug-ins are shown directly. All other plug-ins can be used by opening an additional

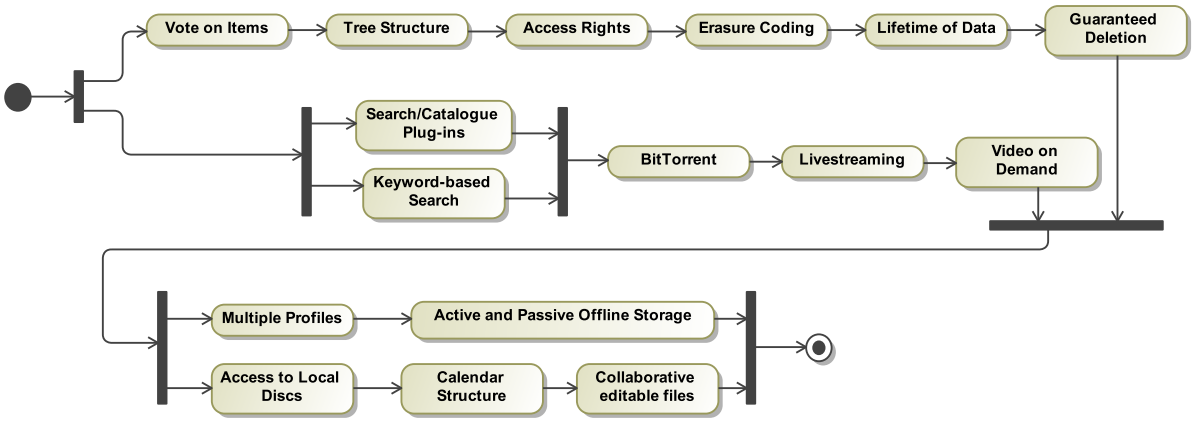


Figure 2.2: Team Framework Work Plan According to Priorities

menu. Another solution could be a tree in which plug-ins are sorted in categories. This would allow browsing installed plug-ins in a more structured way, e.g. listing all multi-user strategy games.

For providing a more flexible and customizable GUI we are also planning to make the views collapsible. In addition, we want to provide predefined layouts of *views*, i.e. the graphical interfaces of the plug-ins, for different use cases. In the context of the Eclipse Platform these layouts are called *perspectives*. They are responsible for the arrangement of containers which include the different views separated by tabs. The perspectives are fully customizable and can also be stored and loaded by the user.

2.5 Framework

In this section the challenges and goals of the team framework are described. The framework of LifeSocial is based on the functionalities provided by the p2p overlay. It is supposed to implement several mechanisms that in turn can be used and combined to develop useful plug-ins for the online social network. Therefore, the framework is the basis of the functionalities the social network can offer. The more functionality it provides the richer the design of possible plug-ins can be.

We identified five different topics that should be considered and which are described in more detail in the following.

Figure 2.2 shows the different challenges to solve according to their priorities. We are starting with two parallel workflows containing the higher prioritized challenges. After finishing both workflows we will continue working on the lower prioritized challenges again in two different workflows.

2.5.1 Storage

In this section we discuss how to improve the storage functionalities of the framework.

Collaborative Editable Files

In every social network users share data items like personal information, pictures, and calendars. But also sharing documents like text documents or even presentations became very popular. If we share a text document with other users, for example, we may want to edit and work on this document simultaneously, as it is known for example from Google Docs. Furthermore, we can think of a whiteboard where multiple users can add items like text, pictures, and videos while chatting or even video-chatting with each other.

The basic challenge for a collaborative editable file editor is to synchronize the content in the file between all active users in real time on the right place without overwriting data from other users. For example, if multiple users edit the same line in a text document, it has to be secured that conflicts are solved without data loss. In addition, such collaborative files have to be secured with access rights for different users or different user groups. Therefore Access Rights Based on Hierarchical Roles, described later on, have to be implemented first. The rights are applied top down in a tree structure 2.5.5. The root have most rights and the childhoods can have less rights.

Define Lifetime of Data

Usually, the data in social networks is stored for an unlimited time after the publication. Such an approach is useful for those types of data, which do not lose their urgency, for example photos, multimedia files, and text documents. However, data such as messages and comments usually lose their importance through some time interval, for example in one month after the publication. Besides, the user can be dissatisfied with the fact that his comments and messages still remain accessible to users for an unlimited time period. A special mechanism, that will allow the additional lifetime property to define how long the data will be stored in the network, is useful.

Provide Guaranteed Deletion

The user of social networks leaves a lot of private information about him in the network, for example comments, photos, personal data, contacts and information about relations with other users. At any point in time the user can wish to remove this information partially or completely from the network. At present, almost all centralized social networks do not give a possibility of guaranteed and definitive removal of user data. They only allow to block the access to the undesirable data for other regular users, but they still keep them on their servers. Thus, though the access to the information for other users is closed, the arbitrary (on demand of the authorities) or involuntary (as a result of attack on the server) information transfer to a third party remains possible. Administrations of social networks usually give the reason for such approach - too big expenses for removal of the data and for defragmentation in their data banks. However, the p2p networks have not centralized data storage and the data is more or less evenly distributed between all users. The expenses for deletion of the data from local discs of users can be not taken into consideration. However, the p2p paradigm generates another problem the data can be distributed between a lot of other users and can be duplicated. It is necessary to implement the approach, that will allow sending of delete requests to all network participants and will consider the temporary inaccessibility of the offline peers.

Provide Active and Passive Offline Storage

Here our goal is to implement an active and a passive local storage of data. Such data can be used to support delay tolerant networks. The main idea is to save files or profiles on the users local disc in order to make them available when the user is offline so that he can view or change them offline. The challenge is to decide which files should be stored and which the user is allowed to change offline without causing any consistency issues. We also want to support the possibility that a group of users can use the social network although they do not have a connection to the Internet and are only connected via a local network infrastructure like LAN or Bluetooth. The challenge here is that these users should be able to exchange their offline stored data and that offline changes are only performed once after one of the users is online again.

Support Access to Local Discs of Friends

Here we want to give users the possibility to provide friends a direct access to some local files without storing it in the network. Such a functionality could be used in plug-ins for example to enable a user to play some music located on a friend's computer. We can decide between different use cases:

- Grant access to anyone who is interested by using for example BitTorrent 2.5.3 (traditional file sharing), the data is stored on the publisher's local disc.
- Grant access to file system for only one friend for example during a chat conversation.
- Grant access to a list of selected friends where the data is still stored on the publisher for example during group chats.
- Grant access to a list of selected friends, where the data is also stored on the selected friends, thus the publisher might go offline and the data is still available, at least as long as the friends are online.

Erasure Coding for Redundancy

Due to churn, the leaving and joining of nodes, it is important to introduce redundancy into the system to ensure that files are still available if the responsible node is offline or has lost the file. Main properties that should be taken into consideration are high availability of content and low storage overhead. Until now file replication is employed, and therefore several copies of a file are stored at different nodes. This approach has mainly two drawbacks in comparison to erasure coding where files are split into smaller chunks and coding techniques are utilized to add redundancy. Erasure coding overcomes the problem that large files cannot be stored because a single node has not enough storage space. Furthermore, erasure coding allows to provide a higher file availability while using the same amount of storage space. Here the main challenge will be to handle the higher complexity of chunking and coding because to retrieve a file several chunks has to be gathered. The aim is to provide usual replication for data that could be changed later on and also erasure coding, which will be applied for data that could not be changed later on, because using erasure coding, update procedures might be expensive to execute.

Reputation of Data/ Persons - Vote on Items

In online social networks like Facebook, votes as for example a ”like”-button are popular and heavily used. Therefore, it is necessary to implement some kind of voting system in our framework. The ”like” button lets users vote on content like comments or photos with friends on Facebook.

Overall the users vote, if they like or do not like the content. In general this functionality can be provided by an overall voting functionality where you can choose your own voting elements like for example smile, good, cool, etc. The challenges are to define the topics to vote on, to store the users votes to let every user only vote once, and to link it to any kind of data items like comments or photos.

Keyword-based Search

Due to the fact that the framework is based on top of a structured p2p overlay it is basically only possible to retrieve documents by knowing their fileID. However, it is desirable to allow users to retrieve a document without knowing its file ID in advance. Therefore, we aim at designing a possibility to search for certain documents by some kind of keyword search. For example it should be possible to search for people from a certain city or with a certain age. The keyword-based search mechanism can be used for different types of data. The users profiles usually contain certain personal information such as country, city, age, profession etc. Users in the social networks can be grouped using this private information. The music files usually contain additional meta-data about genre, singer, album etc. Other data such as text documents or binary files can be described by means of additional meta-files, in which the users can add the optional tags. These tags can be used as keywords for the advanced search Furthermore, such a search is also reasonable for an appstore to find plug-ins that provide a certain functionality or belong to a certain category.

2.5.2 Security

In the following we describe two different approaches to enhance the framework in terms of security.

Access Rights Based on Hierarchical Roles

The framework obviously should offer the user a possibility to define access rights for its data. So far this is only possible to give such rights to a list of other known users that should be able to access a certain data item. However, it might be reasonable to be able to provide access to whole groups of users (users that have a certain role like ”staff of company XY”) that are not necessarily known individually. Furthermore, our goal is to design such roles in a hierarchical manner, i.e. for example the role ”member of management of company XY” also implies the role ”staff of company XY” and therefore, if an item is shared with ”staff of company XY” also the users with roles ”member of management of company XY” have access to this item. It will also be useful to have the possibility to store special rights for users or groups of users. For example the right to read, edit or delete. The challenge here is to offer these access right completely

based on cryptographic methods, since we cannot trust a central access control in a peer-to-peer system.

Support Multiple Profiles per User

Usually, each user has only one profile in the social network containing the personal user data which are accessible to all users or only to users from the friend list.

The p2p framework of LifeSocial should allow participants to create several profiles for different purposes. One profile, for instance, could be used for business contacts whereas a second profile is used for the user's social life.

2.5.3 Large File Transfer

This section describes framework improvements in terms of large file transfer such as BitTorrent or video transfer.

BitTorrent

BitTorrent is a file sharing protocol, which is based on p2p technology. It is used for distributing large files or data over the Internet. The main differences between BitTorrent and normal p2p lookups are: In BitTorrent the source file is split into several pieces to enable simultaneously download of missing pieces. BitTorrent uses a tracker to keep track of all active peers in the network, it holds information about the file, the pieces of the file, and its hash values for validation It additionally uses a tit-for-tat principal to solve the fairness problem. That means, if a peer stops or reduces his upload, all its neighbors immediately stop or reduce uploading to this selfish peer.

We will use this concept for a rapid and efficient exchange of large files between the individual peers in LifeSocial.

Video on Demand

Using video on demand a user starts downloading a video file and simultaneously starts watching it. The user can also use functionalities like pause, quick / slow forward, or jump to different positions of the video.

The framework will benefit from such a video on demand functionality in multiple ways. Users are able to upload videos and share them with other users without the requirement to stay online and multiple users can watch the video file simultaneously without stressing a single node. The challenges of a p2p based video on demand system are to store the video file in the network and make it available for multiple users at the same time. For this the users watching the same parts of the video have to forward the data.

Live Streaming

To have the possibility of live streaming in the Framework would offer great advantages. Users are able to video-chat with each other or even broadcast live streams to large groups of users.

A peer-to-peer live streaming solutions using functionalities of for example Skype or SopCast can be integrated into the framework. More challenging is the functionality to broadcast live streams towards the huge amount of users. Such a system has to be robust against frequently joining, leaving, or even failing nodes. It needs a short delivery delay, a low data miss rate, it has to balance different upload capacities, and provide fairness.

2.5.4 Search and Cataloguing of Plug-ins

Plug-ins can be used in LifeSocial for expansion of its base functions as well as simple and fast addition of various new features. LifeSocial supports many useful services, but the necessity to expand the available functionality of the program can arise at any moment. For high-grade work with the plug-ins the program shall contain mechanisms for search and cataloguing of plug-ins.

LifeSocial needs a mechanism for the distribution and discovery of plug-ins in the p2p network, that later will expand its functionality. However, OSGi does not include the technology for the distributed sharing and search of the components. We need to define and implement the solution for powerful storing and discovery of OSGi bundles in the LifeSocial. It will be helpful to implement the access to plug-ins not only by routing, but also by searching for keywords or descriptions and functionalities of plug-ins. This feature corresponds to the challenge ”Keyword-based Search” 2.5.1.

2.5.5 Data Structures

Data structures are important to handle data in an appropriate way. So far only a distributed list, a distributed set as well as a prefix hash tree are implemented. In agreement with the GUI and plug-in team and according to our challenges the following additional data structures have to be implemented in the framework.

Tree Structure

A distributed tree structure might be useful for several applications. It is, for instance, needed for hierarchical forum or folder structures. Therefore, we want to develop a tree structure for the framework.

Calendar Structure

LifeSocial needs a special format for storing and exporting of distributed calendar information, because shared calenders can become to large to be stored in one file on a single node. In general we need a distributed hash table who maps an object to a set of objects to make use of this data structure even in other use cases.

In case of the calendar we can map a time object containing for example the year, the month, and the day to a set of events, while each event is presented by an special event object containing for example the timespan, the place, and the organizer. This can be implemented using the basic distributed hash functionality of Pastry for example by mapping calendar identifier and dates to a set of events (distributed hash table).

Using the developed structure the user will be able to publish and to share his calendar on the network or to send meeting requests to other peoples, who can import the events into their own calendar. The calendar structure must contain different information such as title, summary, start and end time of the calendar events. There are some existing calendar formats, that can be used as sample (e.g. iCalendar (*.ics)) and we need to decide which information will be stored in our calendar format and how it will be done.

2.5.6 Summary

In this Section we have presented a lot of different improvements to the framework in the areas Storage (Section 2.5.1), Security (Section 2.5.2), Large File Transfer (Section 2.5.3), Search and Cataloguing of Plug-ins (Section 2.5.4), and Data Structures (Section 2.5.5). We will work on the challenges out of these areas according to the priority plan shown in Figure 2.2 as introduced at the beginning of this Section.

2.6 Overlay

The key element of any p2p system is an overlay network which operates on top of the network layer. The overlay organizes the nodes in the network in a specific way to allow routing and other operations.

LifeSocial is implemented on top of FreePastry which is a prototypical implementation of Pastry developed by the Rice University. As team responsible for the overlay, we identified four major topics that must be addressed.

One of these topics is security. Security can be defined as fault free working environment in the present of attackers. The previous project groups have provided secure storage and communication in LifeSocial using asymmetric keys. However, the overlay still remains vulnerable for routing attacks. Attackers may achieve the attack by pretending other identities, getting information by forwarding messages, or partitioning the network for instance. Here we identified some major attacks which need to be considered. Firstly, we discuss Sybil, Eclipse, and routing attacks. Lastly under this topic we discuss user anonymity which allows/disallows users to send or receive messages anonymously. Solving such issues allows us to publish LifeSocial in an unrestricted user base where malicious users are expected. For that reason we believe that solving the security concerns must have our highest priority.

Within the topic of security we will first address Sybil attacks because they lead to all the other defined attacks and hence is the root of the attacks. This is followed by eclipse attacks, routing attacks and anonymity issues.

Heterogeneity is a second very important field that we will assign the second highest priority after the security. Capacity awareness and uniform load distribution are some of the major goals in this field. It allows users with various categories of devices to easily access LifeSocial without the danger of any devices being overloaded.

NAT traversal issues play an important role in the real world. The main goal is to allow machines behind such a system to communicate with other nodes in LifeSocial. It is the most important task after the Security and Heterogeneity were addressed. Lastly, we study the impact

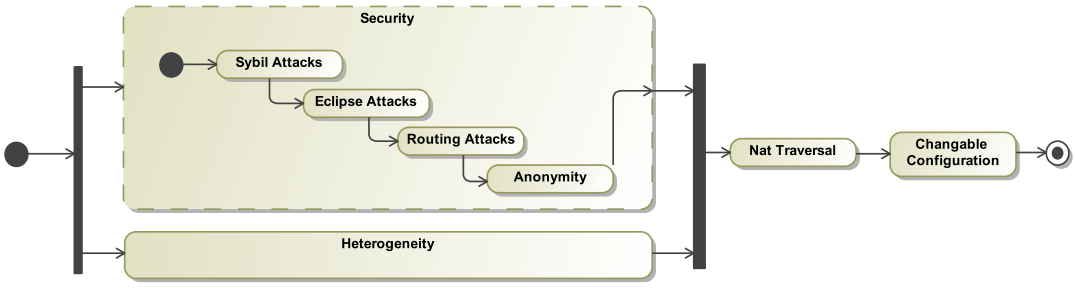


Figure 2.3: Team Overlay Work Plan According to Priorities

of changing configuration on a p2p system at runtime and discuss various approaches which could help us implementing a changeable configuration in LifeSocial.

Figure 2.3 shows that we intend to work on the security issues and heterogeneity in parallel. The two topics with least priority are addressed in sequential order when security and heterogeneity were successfully implemented and tested. In what follows in this section, all topics are described in more detail. They are arranged in descending order according to their priority.

2.6.1 Security

Security issues are an important aspect when designing and refining the p2p overlay. A number of possible attacks caused by malicious users and the meaning of an anonymous p2p system is described in this section.

Sybil Attacks

If the system cannot control the relation between entities and identities, then the adversary can create numerous identities and deceives the other nodes. In Pastry, if an adversary creates many fake identities, it becomes easier to pollute the routing table of honest nodes. With many virtual identities, the adversary gets a better view of the network and can control other nodes more easily. In addition, when all the fake nodes join the network the bootstrapping action causes a lot of stabilization traffic in the network. The problem will get worse if the overlay design, needs to transfer a lot of data in the bootstrapping phase. On the other hand, when all the forged nodes fail suddenly it causes to decrease the robustness of the network dramatically. These fake nodes potentially have the power to partition the network and create a parallel network to hijack the newly arrived nodes.

Eclipse Attacks

Every node in Pastry maintains a neighbor set. When a group of malicious nodes could control a larger fraction of these neighbors than the legitimate node is said to be eclipsed by this set and hence the name. This attack may lead to various other routing attacks and which makes preventing such an attack very important. For example, in Pasty, a routing table is built based on prefix matching so malicious node could provide a false update therefor poisoning the table. There are various solutions existing in the p2p world to prevent such an attack and we plan to

implement and test some of the good solutions to find a better solution and implement it for LifeSocial.

Routing Attacks

In order to forward a message to its destination in Pastry several nodes are responsible. In each step of the routing the responsible node forwards the message to another node which is closer to the target. If a malicious node appears as one of the intermediate nodes between the origin of the request and the destination, the routing system can face a problem. One of the important reasons is that the initiator has no control over the intermediate nodes. As a routing attack, the adversary can forward a request to a wrong destination or simply stop forwarding the messages. It can also pretend that it is the destination node.

Anonymous p2p network

An anonymous p2p network means nodes or participants in the system are anonymous achieved by hiding its identity. It is getting more and more popular these years. The main reason for such a network is to communicate without revealing one's identity. Anonymous p2p networks can be used to protect privacy and prevent possible attacks. However, there are also malicious motivations, such as distribution illegal content, escape censorship or fear of retribution.

We analysed and divided anonymity into 2 major sectors of use:

1. Routing Anonymity: The routing anonymity is a defense of traffic analysis. It means the route between two peers should not be invariant. Usually after the initialization of p2p network, peers communicate via unchangeable route paths. If peer A routes the message to peer B unchangeably, it may have a high chance that the attacker is able to get the knowledge of the relationship between A and B when an attacker exist in this route path. The attacker can perform certain attacks to decrypt messages between A and B. The challenge here is to make the route path unique and random every time. Since usually there are lots of peers in a p2p network, randomness raises the possibility of uniqueness. However, total randomness will cause a huge delay and too much overhead of routing, so the randomness is limited to the initiator of routing(the first peer that message hop to). Provided that routing in FreePastry is based on the peer distance metric of identifier (message is always trying to be routed to the nearest peer), the first idea we consider is to not route the message to the nearest peer, instead, a random peer which is a little far away from the sender. Certainly this distance between sender and initiator should be chosen not only randomly, but also reasonably, otherwise it will cause much overhead

2. Content Anonymity: Content here covers various sections like messages, comments, and shared files. There are various users who do not want to reveal their identity when they make a comment, send a message, or share a file. Every node is responsible for a set of objects and in turn every object has a responsible node assigned to it. In such a case having a universally anonymous id to be responsible for the content which are hidden is a really big challenge. Having such a feature would be advantageous in many situations, for example in a polling system where identities are supposed to be kept hidden.

The combination of the two approaches above would be a challenge. However, anonymity would be more of a feature compared to other routing attacks, hence we believe that it has the lowest priority. However, it would always be good to have it.

2.6.2 Capacity Awareness

Mobile user devices like smart-phones, tablet computer, and notebooks allow to use Internet services at any time. This freedom in usage is a big gain for the user but it introduces challenges for application developers. Especially designers of p2p systems have to be aware of the different capacities of the nodes participating in the system. Nodes in a p2p system are the consumer and provider of services at the same time. FreePastry tries to balance the load equally in a way that each node in the system is assigned the same workload. However, due to the heterogeneity of the node set, such a system is unfair and unbalanced. Nodes with little capacity like smart-phones may be overloaded with the assigned fraction of the workload. On the other hand, stronger machines may still have free capacity available that is unused.

Two general types of capacity awareness can be distinguished. One capacity aware strategy allows nodes to not provide any service. These nodes may be very weak or unreliable machines like smart-phones which usually only offer low bandwidth, storage, or processing capacity. They are not participating in data storage, processing, or in routing.

A second type of capacity awareness reduces the id range a weak peer is responsible for. These nodes are still participating in the p2p network, however, their storage and processing workload is reduced since fewer elements are assigned to those nodes due to the reduced id space. Strong nodes, on the other hand, are assigned a greater id space and are therefore responsible for more load.

The capacity awareness will allow weak machines to participate in LifeSocial without overloading them. It will additionally create a fair system where the load in the system is balanced in a way that strong machines get more workload assigned than it is the case for weak nodes.

2.6.3 NAT Traversal Issue

In the common networking interactions, servers are implemented and maintained to supply the clients with the services. So all the network relies on powerful servers to provide a lot of client services. In this network architecture NATs, Network Address Translation, came to conserve all the clients as the private networks and to provide them with the one way connectivity to the public network for technical and security reasons.
P2p networking as an autonomous environment counts on the distributed nodes around the network instead of powerful servers to handle all networking tasks and communications. Hence, every node in the network must be accessible to serve a specific service. On the other hand, NAT is the major obstacle in two connectivity of nodes to the whole network. As the first applicable solution, the existing FreePastry configuration to traverse NAT/Firewall is going to be considered, this must lead to the enabling port forwarding on the NAT before a new node joins the overlay. Still Freepastry enables the port forwarding by UPnP which my not be available on every machine so in this case it asks user to enable port forwarding manually.Obviously existing functionality is not a comprehensive one to perform NAT traversal in all situation so in

addition STUN (STUNT) protocol should be supported by overlay, as STUN servers are needed to define whether there exists a NAT, and the NAT type. By defining superpeers hole punching and relaying mechanisms should be managed in the network. In this case a node even behind a symmetric NAT can communicate to the network just by connecting to another node in the overlay as relay node.

2.6.4 Changeable Configuration at Run-Time

When the p2p overlay is established, the identifier and the routing table are automatically assigned to each peer once it joins the network. The identifier of node and object is arranged in a predefined key space. The structure or size of routing table depends on the type of overlay. However, since everything is configured automatically, it will provide flexibility if configuration can be changed at run-time Because each node's identifier or routing table has a dependency with other nodes', which kind of configuration could be changed at run-time will be discussed. It will be also useful for the test of LifeSocial. The main usage of this changeable configuration is to enable developers to be able to test different configuration parameters, such as routing tables size and frequency of pings(aliveness checks), routing table maintenance, cache emptying under a different work load or network size. Another issue that will be decided in the future is whether to implement a graphical user interface or leaving the configuration in the source code. It might be interesting if a user can configure its own run-parameters by a GUI. However, there must be limitations to the allowable input configuration of the user. Otherwise users may crash the network by deploying absurd configurations. Such a GUI will also be convenient for developers to test LifeSocial.

2.7 Planned Status at the End of the PG

The goal of the PG is to enhance the p2p framework, overlay and LifeSocial(plugins and gui). Each of the three teams work on the tasks described in this proposal.

The PG is scheduled for twelve months, with a seminar phase (4 months) and implementation phase (8 months). The implementation phase can be divided into two iterations: In the first iteration the proposal is written and a common understanding of the running code is gained during the first weeks of the project group. After these initial steps, the members of the PG are divided into three teams: GUI, P2P Framework and Overlay. Each of these teams works on their individual work tasks that are described in the remainder of this proposal.

The implementations designed by the individual groups are merged after an extensive testing period as shown in Step 5 in Figure 2.5. Subsequently, a series of tests is executed that proves the compatibility of the new implementations.

The second iteration of the project group starts with Step 7. First, a common understanding of the new version of LifeSocial is achieved. Subsequently the group is once more divided into teams that work on problems that have not been addressed yet.

The GUI and plug-in team aims to improve the usability of GUI and to provide additional functionality by developing new plug-ins. To support these new functions, the Framework team will improve the framework by extending the available interfaces and underlying mechanisms.

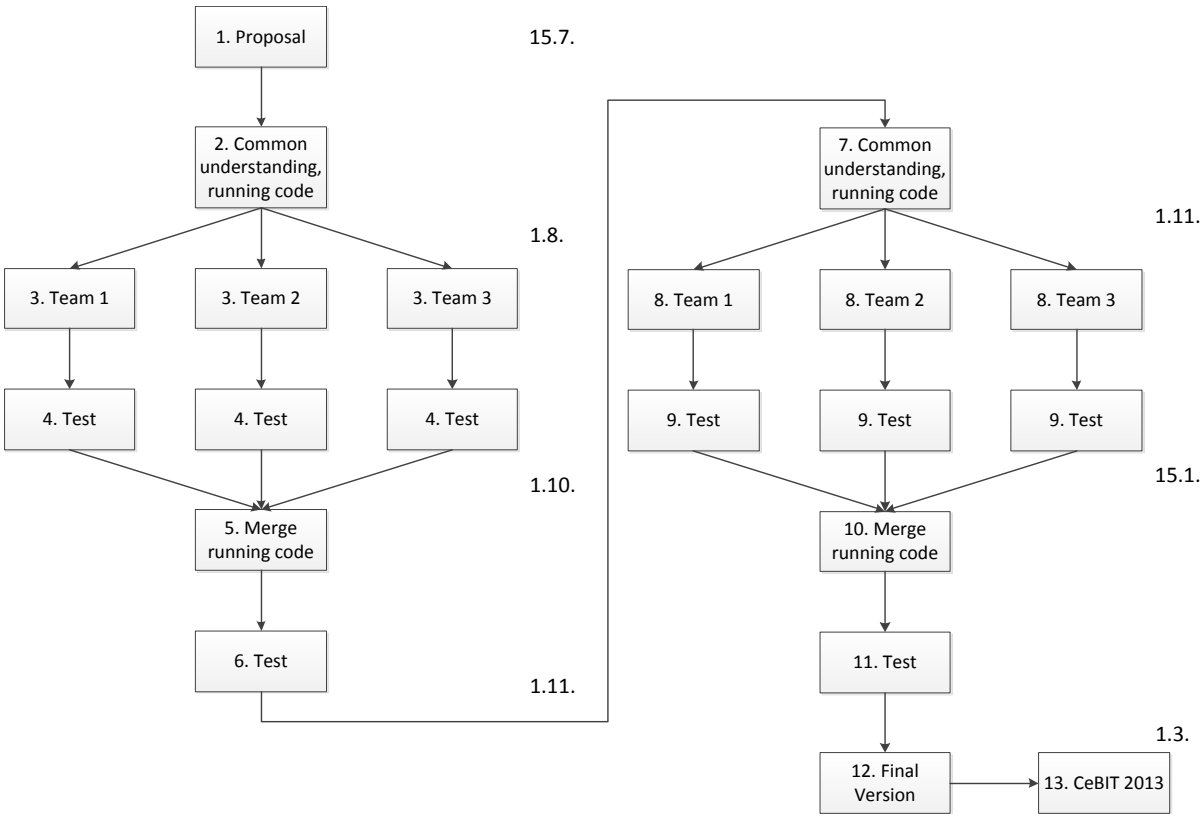


Figure 2.4: Overall time plan

The overlay team will focus on security to increase the robustness of the system. This will allow us to deploy LifeSocial to a bigger audience where malicious users are expected. The overlay group will additionally work on heterogeneity. The goal is to allow different groups of user devices with different capacities to participate in the social network while no node is over- or underloaded.

The changes of the groups are merged twice during the project group. Each of these merging processes is followed by an extensive test period. In each test period, LifeSocial will be executed on multiple machines that execute a predefined series of functions. After the evaluation of the automated test series we will start an alpha test phase where the system is tested among the team members.

At the end of the PG a beta test phase is started to test the final version of LifeSocial. Its purpose is to identify the last bugs in the system and to give us the chance to remove them before the new version of LifeSocial is presented during the CeBIT.

2.8 Time Plan

this is alread contained in the previous section?

The PG is scheduled for twelve months, with a seminar phase (4 months) and implementation phase (8 months). The implementation phase can be divided into two iterations: In the first iteration the proposal is written and a common understanding of the running code is gained during the first weeks of the project group. After these initial steps, the members of the PG are divided into three teams: GUI, p2p Framework and Overlay. Each of these teams works on their individual work tasks as described in the previous sections of this proposal.

The implementations designed by the individual groups are merged after an extensive testing period as shown in Step 5 in Figure 2.5, the goal of the test period is to evaluate the quality of the solutions. The second iteration of the project group starts with Step 7. First, a common understanding of the new version of LifeSocial is achieved. Subsequently, the group is once more divided into teams that work on problems that have not been addressed yet, later these solutions are merged, tested and evaluated, then the new final version of the implementations is created. This version will be then presented to the public at CeBIT 2013.

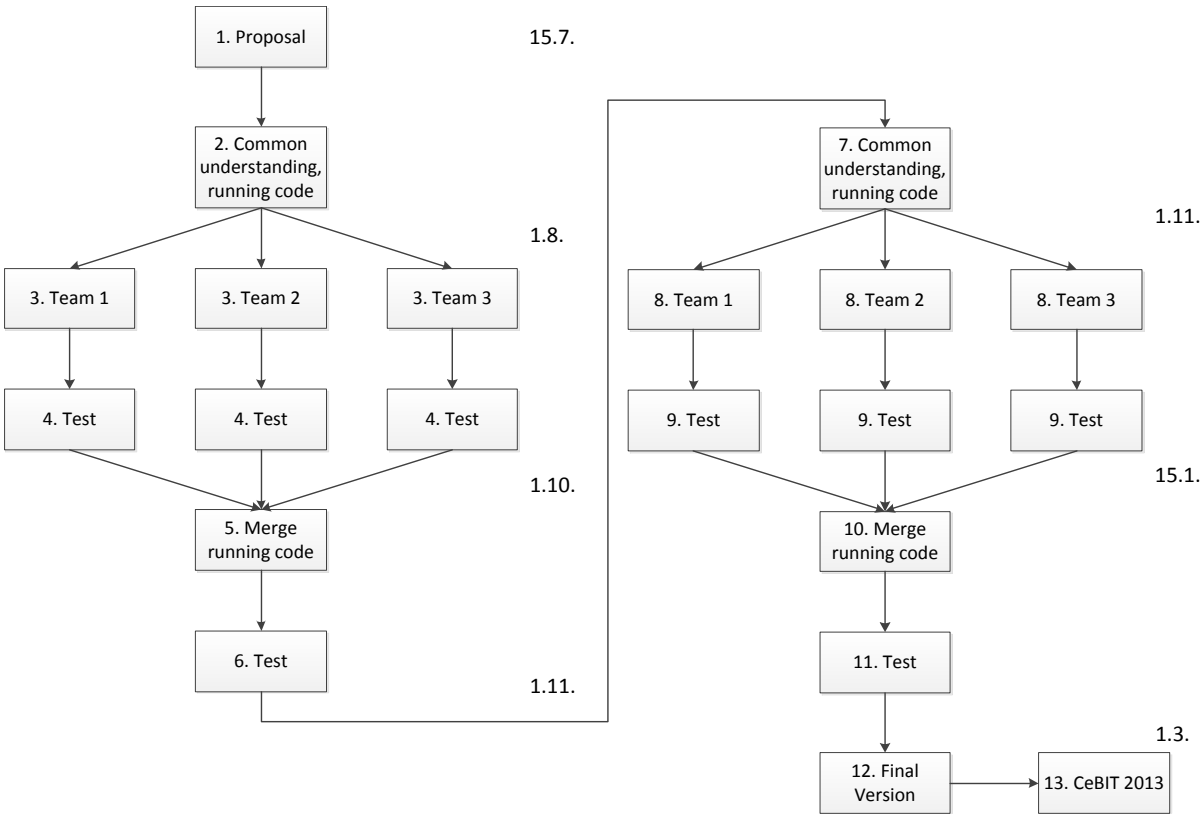


Figure 2.5: Overall time plan

3 Plugins

Since LifeSocial is based on the OSGI Platform, LifeSocial itself consists of many so-called bundles. This fact allows one to add or remove bundles to the application at runtime. Especially for social networks this software architecture is very suitable for further extensions like games and cooperative work.
In the previous PG's each plugin of LifeSocial was divided into an OSGI bundle, which contained the functionality, and an OSGI bundle, which contained the GUI elements. For the communication in-between the bundles, all functions were provided by using a *Command Interface*. By that, each GUI bundle was able to invoke all provided commands of any other bundle. The access was implemented using Strings for the command names and their parameters.
We recommend to change this programming convention to not using the Command Interface for all inter-bundle communication anymore. The usage of strings for all parameters prevents the developers from using hardly needed concepts like callbacks. A programming guideline for plugins can be found in Section B.

Furthermore, we introduce an App Store, which means that the OSGI bundle, responsible for the GUI, and the OSGI bundle, responsible for the business logic, are put together in a so-called *feature*. Now the users of LifeSocial only have to download or update features once from the App Store and do not have to download both bundles manually.

refer to appstore section, will there be one in the plugin chapter too?

In the following the new plugins which were implemented in this PG are described and explained.

3.1 Groups

Groups are an elementary part of social networks since they can be used for different purposes. In groups, users with common interests meet in order to get in contact with each other. Furthermore groups can be used as contact points for different events like courses at universities. There, the user would like to ask questions to the lecturer or discuss them with the other course participants. On the other hand, the lecturer could provide some course related information, just as files like slides or scripts, only to the participants.
Because of the huge amount of different use cases for groups, the plugin requires a very flexible architecture. Therefore, the plugin uses OSGI to provide a mechanism for extending the groups using further plugins (e.g. one for forums and one for a wiki). Using declarative OSGI services, the groups plugin can track all other plugins that provide functionality to the group. In OSGI terminology the group plugin is the *service consumer* for the service *IGroupContentContributor*, i.e. it is notified about plugins that implement the interface

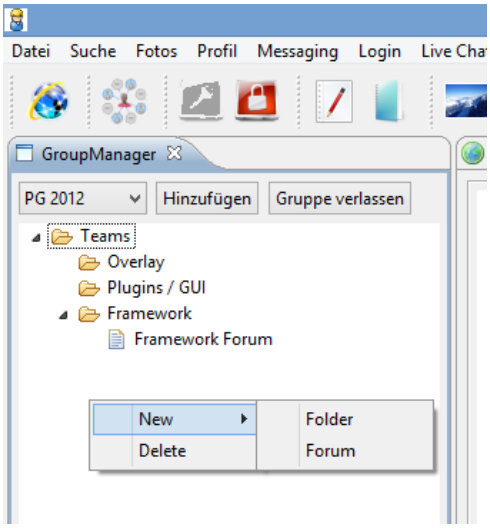


Figure 3.1: GUI of the content manager for groups

org.lifesocial.plugin.groups.api.IGroupContentContributor and declare this fact using OSGI. The latter plugins are called *service providers*.
For achieving great usability for managing the different groups a user participates in and the functions each group provides, we introduce a separate *Group Manager* view. You can see a screenshot in Figure 3.1. The active group can be selected using a drop down list. Afterwards a tree shows the content of the groups. In this example the Group "PG 2012" is active. It contains folders for each team and a forum for the framework team. Items can be deleted and created using the context menu. As you can see, the only possible content type is *forum*, because it is the only plugin that is started and is a service provider for the IGroupContentContributor interface.

In Figure 3.2 you can see a class diagram which shows the extensions mechanism. The items in the tree correspond to *Item* objects which can be divided into *Folders* and *ContentItems*. The service provider implements the *IGroupContentContributor* interface which specifies a list of *GroupContentContributions*. Each *GroupContentContribution* has a name which is used in the context menu for the *new* menu, an ID which is used for mapping the instances of *ContentItems* in the group to *GroupContentContributions*, an association to a Class which extends *AbstractStorageItem* and which is instantiated when creating a new *ContentItem*, and a callback class which defines what happens when a user opens the *ContentItem* in the tree.

The forum plugin for example provides a GroupContentContributor which has the name *forum* and defines the *Forum* model to be instantiated when creating a new ContentItem. When the ContentItem is double-clicked, the forum view is opened and the threads are shown (see Section 3.3).

When communicating in user groups, privacy, and thus cryptography, may be very important. For supporting that, we distinguish between *public* and *private* groups: All users can join every public group by their own. For participating in private groups, a user has to be invited by an admin of that group. The content of the private groups is encrypted symmetrically using the *access rights* feature of the framework (see Section 5.2).

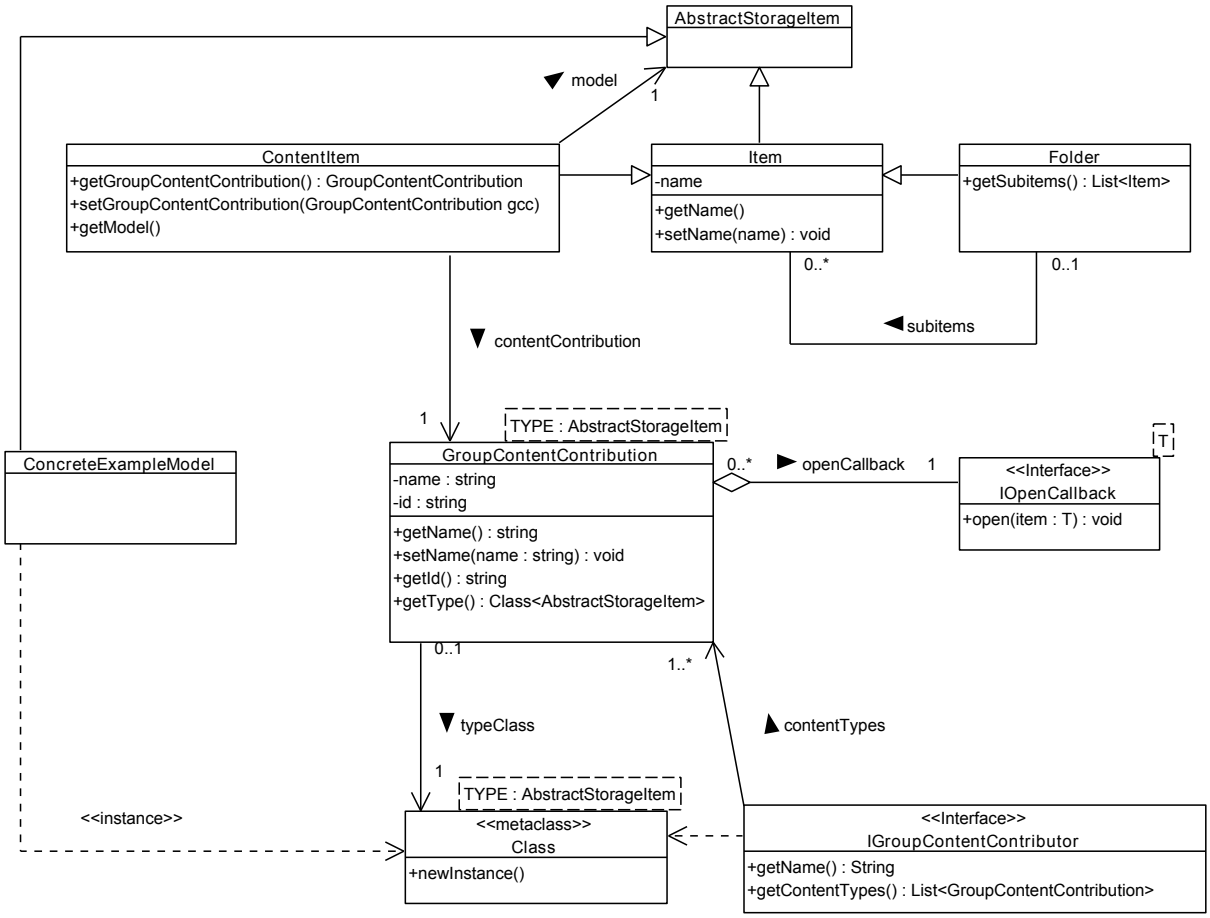


Figure 3.2: Class diagram for group extensions

3.2 Abstract Browser

Several plugins, which were contained in our Goal Definition, have requirements that can be easily fulfilled by using HTML. For example the forum and the wiki benefit from advanced possibilities for text formatting which are implemented using BBCode (Section 3.2.3). Because of these multiple use cases for a browser, an abstract class which provides a base with often needed methods has been implemented in *org.lifesocial.ui.browser.views.InternalBrowserView*.

3.2.1 Communication between Java and Javascript

When using a browser for the UI, it is obvious that it has to be possible to trigger Java methods by DOM events like clicking links or buttons.

For this, a *org.eclipse.swt.browser.BrowserFunction.BrowserFunction* object is created in the *InternalBrowserView*. It provides a method which is triggered by the JavaScript function *lslink(key,arg1,arg2,..)*. The Java method invokes a callback method which corresponds to the key and can be registered using *registerLink(key,callback)*. Thus, calling *lslink* with a key can trigger any functionality in LifeSocial.

3.2.2 jQuery

Since many implementations in JavaScript require similar basic functionalities like document traversing, modifications and event handling, many libraries to ease these tasks came up in the last years. One very popular library is *jQuery* [2]. It has been included in the AbstractBrowser.

3.2.3 BBCode

One option for text formatting in user-generated content is to allow the usage of HTML itself. This would however create a potential security risk as HTML allows to embed any scripts or third party objects (like *Flash*). It can also be designed very freely, so that *phishing attacks* could come up.

A common solution for this is to use a markup language like Markdown or BBCode. Since BBCode is used in most web forums, we decided to also use it for the forum in LifeSocial. An existing JavaScript library for parsing BBCode was created by Patrick Gillespie and released under MIT license [3]. It has been added to the AbstractBrowser, so it can be reused in other plugins.

3.2.4 StringTemplate

Generating HTML in plain Java is quite uncomfortable because the focus of the source code lies on the program code, rather than on the content of the document. Therefore, templating languages have been developed. A very wide known templating language is *StringTemplate* [4]. Though no project group members had experiences with StringTemplate or any other Java templating library, the decision to use StringTemplate has been made because it appeared to be very mature. Eventually when using it for generating the HTML, it turned out as very stable and easy to use.

3.3 Forum

The above described *groups plugin* provides an extensible mechanism for implementing features that can be used by groups of users. As an example we implemented forums. They can be created and opened in the group manager and can therefore also be arranged in folders. These folder correspond to the categories which most modern software for bulletin boards support. The forum plugin itself currently only provides the basic functionality that is needed for discussions: Users can create threads which contain all the posts related to topic. When a thread is created, the author also creates a first post; afterwards users can respond to this post.

For the graphical interface of the forum, a browser has been implemented (Section 3.2). In comparison to a SWT-based solution, the main advantage lies in the flexibility of HTML in terms of layout and design.

3.4 MultiView and Topic Chat

insert some text here (structure of subsections)

3.4.1 Multiview

Lifesocial is equipped with many social interaction features which involves single user, In this PG taking into consideration for the need of multi-user environment features, Multi-View a single interface for the multi-featured functions was introduced. In this Multi-View consist of three part views, first part consists of the friendlist and second part lists the multi-features that can be applied on this friendlist. Third part specially depicts the active topic chat functionality. Currently in this PG there is only one multi-feature which is developed (Topic Based chat).

3.4.2 Topic Chat

In the previous PG one-to-one chat was implemented. Logged-in user could chat with any of his friends, this chat was based on one-to-one message-dispatching concept. Now *Topic Chat*, a new feature has been plugged to the *Live-Chat* plugin. Topic based chat uses the concept of TopicChannel i.e., one TopicChannel per chatroom. TopicChannel needs a channel name when it is created. These TopicChannels receives all the messages published from other peer's TopicChannel instances with the same name. Therefore every chatroom has to be uniquely identified with the topic name. The initiator of the topic-chat creates an instance of the topic channel with an unique name and sends out an invitation to join the topic chat for the selected users. On receiving the invitation, every user has a choice to accept it. On accepting the invitation, an instance of the topic channel will be created at the user's machine with the same name as of the initiator and he will be subscribed to that channel. Further this user receives all the messages sent to this channel and also all the subscribers to this channel will receive any message sent by this user.

Subscribtion to Topicchannel

Once TopicChannel is created with the chatroom name the next thing is to subscribe to this chatroom. All the users who are willing to participate in this chat must subscribe to this chat room to start receiving messages.

To subscribe below method is used: *public void subscribe(ReceiveCallback¡String, Message¿ callback) throws SubscriptionFailedException;*

The callback must then implement a method (specified by ReceiveCallback interface) to receive Messages. With this subscription method and a class that implements the ReceiveCallback method for catching the Messages the user gets all Messages anyone publishes to that chat room.

Publishing messages to channel

For publishing messages to a TopicChannel the channel must be subscribed like described above.Then use the below method in the TopicChannel class.

public void publish(Message msg);

Every message published gets automatically received by any user subscribed to the same chat room/channel name, the receiving users gets the published message delivered to the class/method specified as callback while subscribing.

Active Topic Chat Features

Active-Action viewpart of MultiView GUI shows the current active topic channels. Each active TopicChannel is shown as a tuple of three buttons(addMembers, unsubscribe and listmembers). Selecting any user from the friendslist shown in the first view part of MultiView interface and clicking addMembers will add the selected users to this channel if they are not already subscribed. To unsubscribe from a topicchannel just click corresponding unsubscribe button, this will unsubcribe you from that channel and further will not receive or will not be able to send any messages to that channel. At any point of time during the chat it is possible to view the chat members by clicking listMember button.

The current TopicChat creates a private chatroom, which can be joined only on imvitation. This enables the discussions to be private to certain invited people. As a future extension to this Topicchat feature it is required to create public chat rooms.

Users must be able to search some public chat rooms and join those discussions. These kind of public discussions would emphasize the need of topicchannel. Also it is possible to create private chat rooms same as this topic-chat but this must be public-ly available and must be allowed to join on request. In this PG we discovered the need of identifing the people based on their online-offline status. It would also be great addon to Lifesocial if it is possible to have this feature. In return this online mechanism could be used by topic chat to discover if the members are still online, if not automatically unsubscribe them from the channel.

3.5 FileHosting

In p2p-networks all the data is distributed in the network itself. This fact allows us to store any kind of files by using the storage provided by the p2p-network. Until now, only distributed data

structures can be stored in the network, but we would like to allow storing all kind of data in the network. For that, new functionality has to be implemented next to the distributed data structures that already exist. By implementing the new functionality, some aspects like the size of the chunks have to be considered, since big chunks can flood the entire network.
The FileHosting plugin allows the users to store different files in the p2p-network. By uploading a file, the file will be splitted into chunks of the size of 256 kb, which is the default size of chunks in torrent. By that every user holds a list with the references to the files he uploaded to the network. This list is encrypted with the users private key, since only the user should be able to see his own list. Additionally each file is encrypted by a new generated key, which is also stored in the list. Each file object itself holds a list of the references to it's chunks where each chunk is holding a byte array of the maximal size of 256 kb. The chunks also use the encryption key of the related file, since there is no need to differenciate between the file object and its chunks.
In order to download a file from the network, the user can use the storage id in combination with the encryption key of the specific file. This information can be exchanged as links between users. While downloading a file, the chunks will be downloaded and put together. For the moment the files are stored in the path of the Eclipse application.
As the GUI part of the plugin, two dialogs are provided: one for the download and the other for the upload of files. Furthermore a view is provided, which lists all the files that have been uploaded by the user himself. This view is also providing the ID's of these files, which could be send to some friends. By opening the context menu of any item of the table, the download key can be copied to the clipboard.
For the future one can think of adding the upcoming functions:

- The plugin may allow the user to choose the size of the chunks. It has to be discussed, if there is a fair way, since too big chunks can flood the network.
- The contribution to the Groups Plugin is relevant, since groups can provide group related files, that could be only accessed by the group participants, if the group is private.
- Different ways of providing the ID's of the files should be discussed. E.g. the ID could be send to any friend in the friend's list.

3.6 Voting

Voting is a somehow essential functionality of social networks. Since the networks provide different kinds of social media like pictures or simple short messages on personal walls, users have the opportunity to publish their opinion about that media. Furthermore the idea of votings for serious questions is quite interesting in combination with the provided security solutions of the p2p-framework. By that the data security issues can be solved.
The voting plugin deals with the functionalities provided by the p2p-framework and offers a poll alike function. By that each user is able to create polls and share them with ones friends. For the usability, the plugin consists of several views that guide the user to create proper polls (see Figure 3.3).
The view for the creation of the polls provides several options to configure it. Possibilities can be defined and the number of allowed votes can be set, but the most important option is the privacy

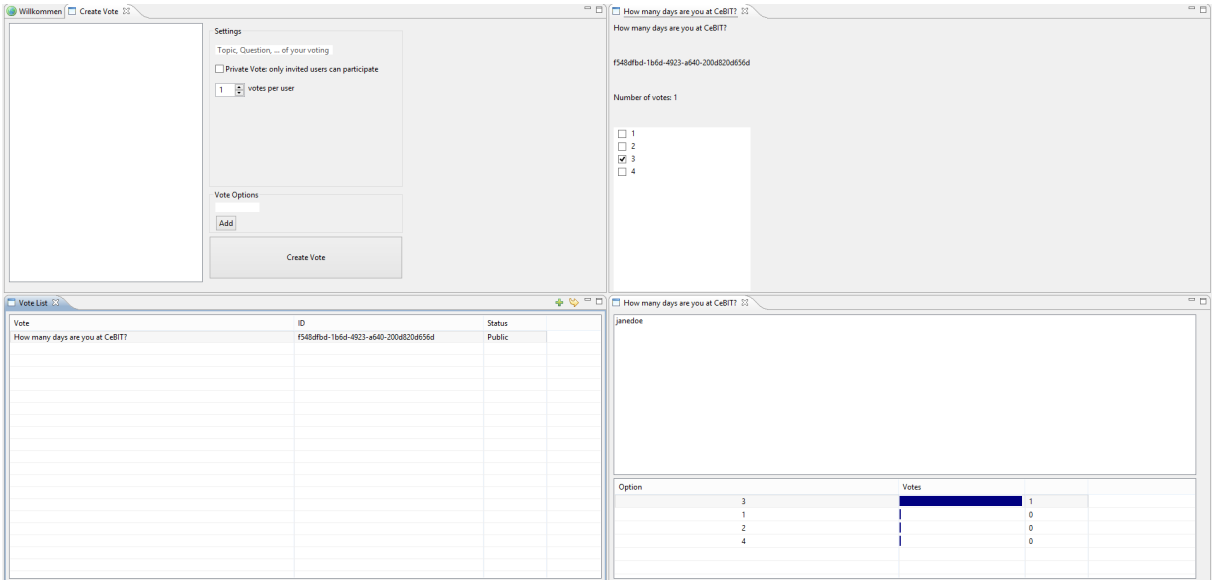


Figure 3.3: Views, that are provided by the voting plugin.

option. Therewith the vote can be configured as public or private poll. So if the poll was created as a private poll, only authorized user can view and participate in the poll.
In order to invite other users to ones polls, the ID of the poll can be shared. The interested users only have to use this ID in order to participate. But if the poll is declared as private, the administrator of the poll have to invite the users. These users will get a notification and can accept or decline the invitation.
Furthermore the user can see the current results of each vote in a seperate view. For each possible option the users are shown, that have voted for this option. By that the results can be analysed.
Further improvements could be:

- The framework functionalities could be used for the "'like"'-button for several StoredItem-Objects
- The contribution to the Groups Plugin is relevant, since groups can provide polls

3.7 FileTransfer

- Adding widget to FriendsPlugin

3.8 Appstore

Social network grows very fast since last decade and has now billions of users all over the world. Users want to personalize their social networking services to suit their own demands. We have therefor decided to realize the Appstore for Lifesocial, which can be used for expansion of its base functionalities as well as addition of various new features with plugins.

3.8.1 Requirements

One important characteristic of the peer-to-peer network is there are no notion of client and server, all the nodes are equal in the network and act as both ”server” and ”client” for the other nodes on the network. The nodes can contribute to the network with their capability such as their storages. At this time, they act as a ”server”. The user should can act as two roles in Appstore, the plugin provider and the plugin user. As plugin provider user can do some contributions to the network such as publish new plugins into the network, release new versions for their published plugins, edit the meta data or even delete the plugins that they have published. As a normal plugin user, they can do the search for the plugins which were published into the network, download the target plugins and install them. They can also manage the installed plugins after the installation, such as start, stop, update or uninstall the plugins. In order to provide these functions, we also need the related the GUI for the different use cases. For the provider, we need a *publisher view*. And for normal user, we need a *search view* and a *view* for the management of the installed plugins.

3.8.2 Basic idea

In this section there will be an introduction about the basic idea and the concept of the Appstore.

Roles

There are two different roles which are defined in Appstore: plugin provider and plugin user. Plugin provider means the author of the plugin who has published the plugin and will be permitted to publish new releases of the plugin, to remove any version of it or even the whole plugin later. Different from plugin provider, a plugin user has only read rights, e.g. search for plugins, read the meta-data and also download and install the plugins. As normal user, they can also do some further management after the installation of the plugin.

Models

Each plugin which is stored in the network consists of 3 parts:

- plugin meta-data `PluginMeta`
- version meta-data `VersionMeta`
- plugin object file `PluginObjectFile`

The `PluginObjectFile` is an object which stores the plugin *.jar* file in the network. The `PluginMeta` contains the basic information about the plugins such as plugin name, provider name, current version, old versions, dependencies and the icon, while the `VersionMeta` contains the information of the released version of a plugin: plugin name, plugin version, release notes and dependencies. The `PluginMeta` will be generated by plugin publishment and will be adjusted by releasing new versions, but the `VersionMeta` will be generated for each versions by every new releasing, it will not be adjusted by new releasing.

The idea of private repository is also applied by Appstore. Corresponding to the two roles in Appstore, there are two private repositories realized:

- `PrivatePublischerRepository`
- `PrivateUserRepository`

The `PrivatePublischerRepository` contains the list of plugins, which published from the user while he is acting as provider. The `PrivateUserRepository` contains the list of plugins, which the user has installed while he is acting as normal user. Both of the repositories will be loaded by the starting of the Appstore, if any plugin in the `PrivateUserRepository` is not installed locally, Appstore will download and install it automatically. The private repositories will be generated by the first start of Appstore from the user and will be adjusted during the publishing and the installation of the plugins.

Plugin Publisher

By the plugin publishing, the `PluginMeta`, `VersionMeta`, and `PluginObjectFile` will be created and stored into the network. At the same time, the `PrivatePublischer-Repository` will be adjusted, a new entry for the new published plugin will be store into the repository. By publishment or deletion of the plugins the private repository will be.

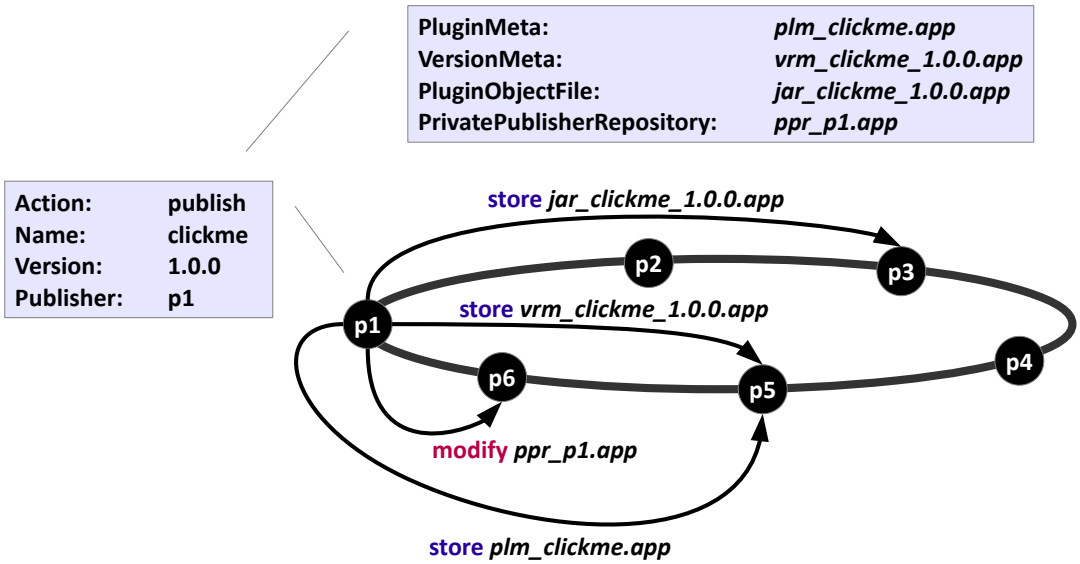


Figure 3.4: Plugin publishment

The Fugure 3.4 shows an example of publishing a plugin. The peer *P1* acts as a provider and wants to publish a plugin with name ”clickme”, and version 1.0.0 into the network. The place where the meta data and the plugin object will be store, will be decided by Pastry. As an existing user, the private repository of *P1* was stored on certain nodes in the network. *P1* sends request to the related nodes. As example here, the PluginMeta, VersionMeta and the PluginObjectFile will

be stored by nodes *P5* und *P3*. A request for modify the `PrivatePublisherRepository` will be sent to the node *P6*, where the `PrivatePublisherRepository` is stored.

After the publishment of a plugin, the user becomes the provider of the plugin. So he can release a new version for the plugin. By this new version releasing, the existing plugin meta will be modified: the new inputted version will be stored as current version and the old current version will be stored into old versions. At the same time, the new plugin object and the version meta will be stored in the network.

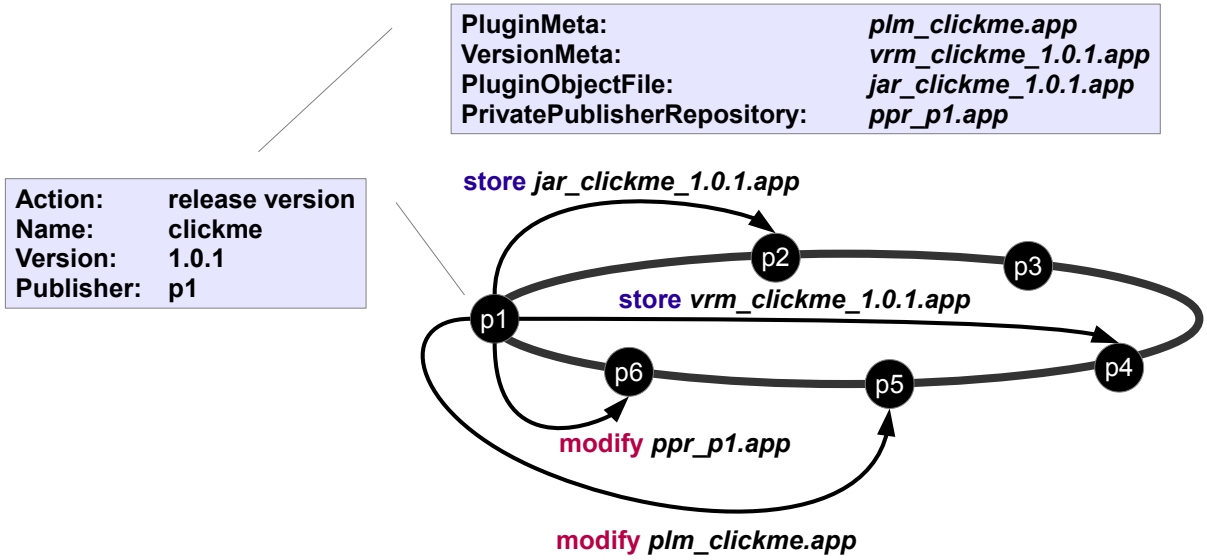


Figure 3.5: New version releasing

The Fugure 3.5 shows an example of releasing a new plugin version. The peer *P1* acts as provider and wants to release a new version with version 1.0.1 for the plugin with name ”clickme” into the network. Because *P1* is an existing user and the plugin was published into the network from *P1* earlier, the `PluginMeta` and `PrivatePublisherRepository` of *P1* was also stored on certain nodes. As example here, the `PluginMeta` of plugin ”clickme” and `PrivatePublisherRepository` of *P1* is stored on nodes *P5* and *P6*. *P1* needs to send a request for modifying them. The version meta and the `PluginObjectFile` of the new released version will be stored on nodes *P4* and *P2* by sending a store request.

As provider, user can edit the meta data of their published plugins. The description and the release notes can be modified. The `PluginMeta` will be adjusted by changing the description of the plugin, and the `PluginVersionMeta` will be adjusted if the release note is changed. For the deletion of plugins there are two different variants: completely deletion of the plugin or only remove some plugin versions. For the second issue if the target is the current version, the `PluginMeta` should be modified as followed: the current version will be replaced by the last version in old versions while the related plugin object and `VersionMeta` is deleted. If the target is in the old versions, the entry in old versions will be deleted while the related plugin object and VersionMeta is deleted. For both issues, the `PrivatePublischerRepository` will be adjusted and reloaded.

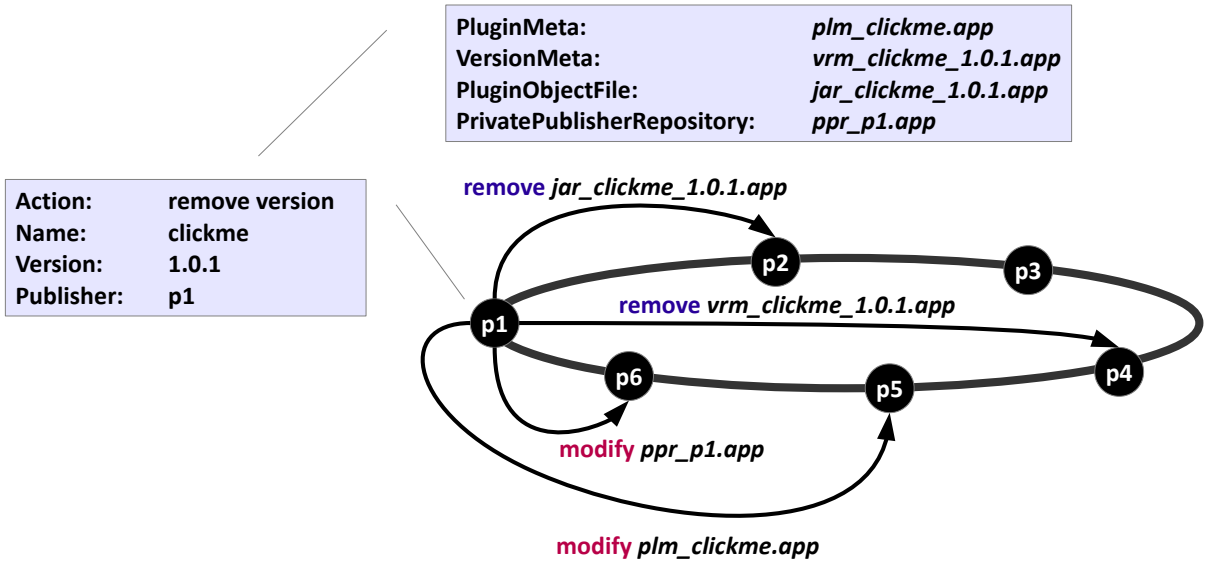


Figure 3.6: Plugin version deletion

The Fugure 3.6 shows an example of removing the plugin. The peer *P1* acts as provider and wants to remove the version 1.0.0 of his published plugin click me from the network. Because *P1* is an existing user and the plugin version was published into the network from *P1* earlier, the `PluginMeta` and `PrivatePublisherRepository` of *P1* was also stored on certain nodes. As example here, the `PluginMeta` of plugin "clickme" and `PrivatePublisherRepository` of *P1* is stored on nodes *P5* and *P6*. *P1* needs to send a request for modifying them. A remove request will be sent to nodes *P4* and *P2* so the `VersionMeta` and the `PluginObjectFile` of the new released version will be removed.

Plugin user

The most simple search approach is to search the plugin by means of a precise plugin name. This precise search mechanism will be used in the appstore. At first, the related `PluginMeta` will be found by searching with of the exact plugin name. The list of available versions, which can be download and installed, will be created from the latest version and a list of old versions.

3.8.3 Implementation

The source code of Appstore is divided into 4 layers: framework, plugin, GUI and example plugins for Appstore.

Framework

There are 3 Appstore packages in the framework part under `com.p2pframework`:

- `com.p2pframework.appstore.model`

- com.p2pframework.appstore.server
- com.p2pframework.appstore.utils

The model package contains the interface AppstoreData which is the general interface for all data that can be stored by the Appstore and the class SignedAppstoreData. All data stored in the network by Appstore must be serializable and implement the interface StorageItem which is used by StorageDispatcher. The function generateKeyValue(), which must be able to generate the unique key for the stored data, must be realized in all classes, that implement the AppstoreData interface. SignedAppstoreData represents the signed data, which will be used by remote operation approach for security reasons.

The server package contains the classes AppstoreServer and AppstoreDataServer which represent the server concept of appstore. The functions in both of these packages are the server routines and must be started by all the nodes in the network, which will store or provide any appstore data such as metadata or plugins in the network. The classes AppstoreServer contains the general low-level functions such as retrieveData and storeData, which interact with StorageDispatcher and manage the AppstoreData instances in the network. AppstoreServer is realized also with the signature and security mechanisms. The class AppstoreDataServer inherits the AppstoreServer and extends it with the client message processing and realizes 4 general functions: storeMeta, removeMeta, retrieveMeta and modifyMeta.

The utils package contains the AppstoreServerReturnCode enumeration, which defines some return codes to indicate the result of the current server-operation on the client side.

Plugin

The plugin part includes 4 packages:

- org.lifesocial.plugin.appstore
- org.lifesocial.plugin.appstore.client
- org.lifesocial.plugin.model
- org.lifesocial.plugin.utils

The root package org.lifesocial.plugin.appstore contains Activator (see Appendix B), main class AppstorePlugin and two classes UserToolKit and PublisherToolKit that gather all the useful functions, that will be used by GUI. Both ToolKits inherit the abstract class AppstoreToolKit and use the client classes for sending the requests to the corresponding nodes and operate the Appstoredata stored in the network.

In the package org.lifesocial.plugin.appstore.client there are two classes: abstract class AppstoreClient and child class AppstoreDataClient, that describe the general operations of managing the objects in the network: storeMeta(), retrieveMeta(), modifyMeta() and removeMeta(). These functions send the requests to the corresponding servers and collect the servers answers. The AppstoreDataClient can be used for each type of AppstoreData.

The model package includes all appstore classes which instances can be stored in the network:

- `PluginMeta`
- `VersionMeta`
- `PluginObjectFile`
- `PrivateUserRepository`
- `PrivatePublisherRepository`

These classes have corresponding parent StorageItem classes (e.g. `PluginMetaStorageItem`), that describe the general function `generateKeyValue()` which will be used for generation of the unique string key under which the object will be stored.

The last package `org.lifesocial.plugin.appstore.utils` contains:

- `PrivateRepositoryInstalledEntry`
- `PrivateRepositoryPublishedEntry`
- `SeriVersion`
- some exception classes

`PrivateRepositoryInstalledEntry` and `PrivateRepositoryPublishedEntry` are the properties containers for the `PrivatePublisherRepository` and `PrivateUserRepository`. SeriVersion is the serializable version of the `org.osgi.framework.Version`.

OSGi

Open Services Gateway Initiative (**OSGi**) is an open platform for distributing and executing service-oriented or component-based applications. **OSGi** is provided by **OSGi** Alliance. LifeSocial uses the **OSGi** framework for managing plugins.

The components in **OSGi** are called bundles which are Java Archives (JARs) containing applications and libraries. The bundles are described by a special manifest file, that contains bundle metadata (e.g. bundle name, version). Also the plugins used by Appstore contains the bundles and specific Appstore meta-data. The bundles can be installed, uninstalled, started, stopped or updated. **OSGi** manages this bundle life cycle. The changes in the bundle state can be made at runtime.

The Appstore contains lots of functions, that realize the locale management operations of installed plugins (bundles) and use the **OSGi** functionality therefor. All these functions are included in the `UserToolKit` class from `org.lifesocial.plugin.appstore` package. `UserToolKit` uses two necessary classes from `org.osgi.framework`: `Bundle` and `BundleContext`. The first one represents the installed bundle in the **OSGi** framework and provides a lot of useful functions. Most interesting are the function:

- `start()`
- `stop()`

- `uninstall()`
- `getSymbolicName()`
- `getVersion()`

The `BundleContext` instance was used for installation of a bundle. By means of BundleContext the bundle can interact with the **OSGi** Framework. The function `installPlugin()` in the `UserToolKit` uses the `BundleContext` function `installBundle()` in order to install the corresponding bundle from a *.jar* file. `installBundle()` obtains a string, which represents the path to a file, as parameter. This string must begin with the substring "file:///" which is followed with path to the file. The BundeContext grants the access to the installed bundles with the function `getBundles()`. In `UserToolKit` are also other functions to manage the plugins in LifeSocial, that use the **OSGi** functionality.

3.8.4 Evaluation

In this section there will be discussion about the results of our tests on Appstore with monitoring. The interest is on 3 different metrics that characterize the properties of Appstore: the storage consumption of storagedispatcher, the data sent rate and also the data receive rate of each node.

The Test was done with 20 computers. Within 4 minutes 1 plugin with the size of 1.5 MB was published into the network from each node, and for every node there were 5 rounds. It was in summary in 20 minutes 100 Plugins for all the 20 nodes. Then we did again 20 minutes for searching and downloading the plugins. Also 1 plugin to search and download within 4 minute each node. So in all it was in 20 minutes 100 plugins for all the 20 nodes. There was a pause for 5 minutes between the publishing and downloading of the plugins. The whole test process should be around 45 minutes and several minutes for start and stop lifesocial and monitoring function.

From Figure 3.7, we can see the storage size was growing when the plugins were published. And on other time, the storage stays while there was no upload in the network. That accords with our expectation. There are also some points which do not accord with our expectation entirely. First the value on the y axis is measured with Byte / Sec. Thats a bit too small for a long time test. And the value what we got is not clear for us. e.g. 5e+007, which we assume as nearly 50 MB large and maybe larger than it should be. A second example is on the X axis. Also, all test nodes were run at nearly same time, but from the diagram we can see some nodes started to have upload from the 30th minutes on. Secondly, some of the nodes are clearly more popular than other, the storage which is used on these nodes are double or triple larger than other. We assume that maybe they are so popular that other nodes use their storage by publishing also.

On the diagram of data sent rate, which we expected, should be with 10 peaks by the time when the publishment or downloading because the 1.5 MB plugin object would be uploaded or downloaded. From the Figure 3.8(a), we can see there was nearly no sent data at the first half of time, only some peaks at the end of time. Seems that by publishing there was no upload happened and by downloading only several peaks which means onlytestfew nodes did the download. That is much different than the test result that we got which the upload and download finished within 5 seconds.

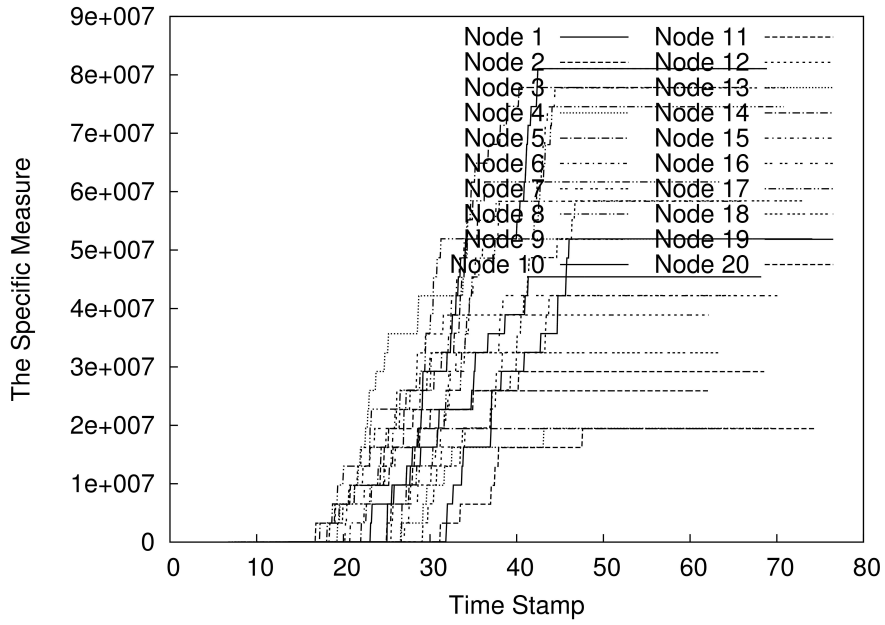
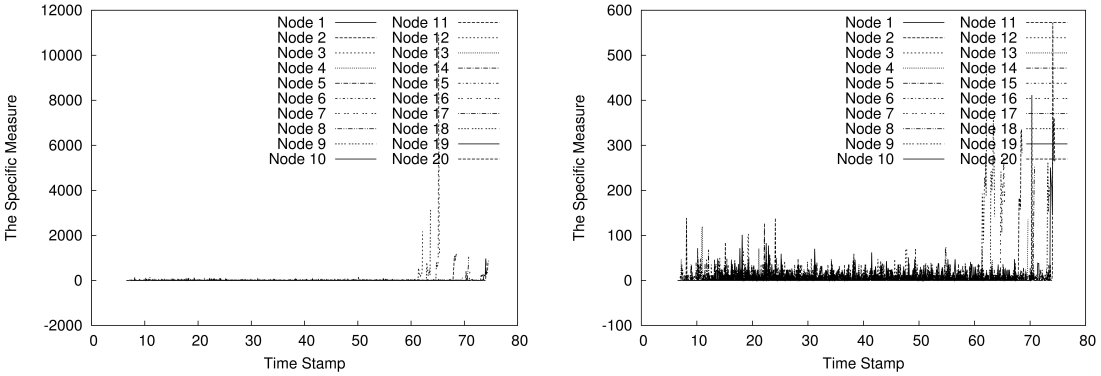


Figure 3.7: Data total size used by Storagedispatcher (bytes)



(a) Network-data sent rate (Byte/sec)

(b) Network-data received rate (Byte/sec)

Figure 3.8: Network-data exchange diagrams

From the Figure 3.8(b) we can not get the expected result also. There are peaks from the beginning. Although the peaks are not so high as they should be. And even the higher peaks at the end of the time stamp are also not high enough. The plugins that we used were 1.5 MB large and both the publishing and downloading finished in $2 - 5$ sec. So the data rate should be several kB/sec, but the y axis shows only in Byte/sec.

From both of the Figure 3.8(a) and Figure 3.8(b), we cannot get result which accords our expectation. We assume that there is maybe problem by measuring the metrics in monitoring of Lifesocial.

3.8.5 Appstore Guide

In order to provide the users an user interface for all the functionalities of Appstore, we have developed 3 views which are *publisher view*, *precise search view* and *installed plugin view*. On the *publisher view*, the users can act as a plugin provider to publish a plugin, release new versions , edit the meta data or remove their published plugins. With the search function on the *precise search view*, the user can search for a certain plugin with its exact plugin name and decide after getting the detailed information of the plugins if they download and install the plugin or not. After the installation of a plugin, user can do some further managements on it on the *installed plugin view*. He can start, stop or install the plugins or get the update if there is a new version available.

Publisher View

The publisher view consists of 2 areas: published plugins area and the input area. In published plugin area the user will find a list of published plugins which are published from himself. Each published plugin is an item of the list, and the user can apply some actions such as new release, edit or remove on the plugin.

In the input area, there are different pages. As initial pages the user will get a short help, a list of icons, which are used on the *publisher view* with the explanation of them. If the user choose the button of ”Publish a new plugin” , the input area of plugin publishing will be appeared (see Figure 3.9).

User can make input of the information about the plugin such as plugin name, version, description, dependencies. Also the path to the plugin object and the icon need to be chosen. Then user can publish the plugin into the network by clicking the button of ”Upload” . After the publishment, the plugin will appear as an item in the list of published plugin (see Figure 3.10).

If the user wants to release a new version for his published plugin, he need to click the ”Update”  button on the plugin in the published plugin area. The page of new release will be appeared after the click in the input area (see Figure 3.9).

The plugin name and icon will be no more editable because this is the update of a certain plugin. But the user will get a suggested version, although he can enter a new version on his own demand. Furthermore the user needs to enter the dependencies and the release notes for the new version and choose the path to the new plugin object. Then the user can click the ”Upload”  button to release the new version of the plugin into the network. After the new release, the new current version will appear on the published plugins and the old one will be moved to list of old versions.

As the provider the user has the rights to edit or remove the plugin, which is published from them. The user needs to click on the ”Edit”  button on the plugin in the published plugin area to get the page of edit appeared (see Figure 3.12).

Then he can edit the description of the plugin or delete the entire plugin. If the user clicks of the version on the current version or the list of old versions, he can edit then the release notes of the target version or delete it (see Figure 3.13). If there is only one version, it means there is only current version and the user wants to delete this version, then the whole plugin will be deleted.

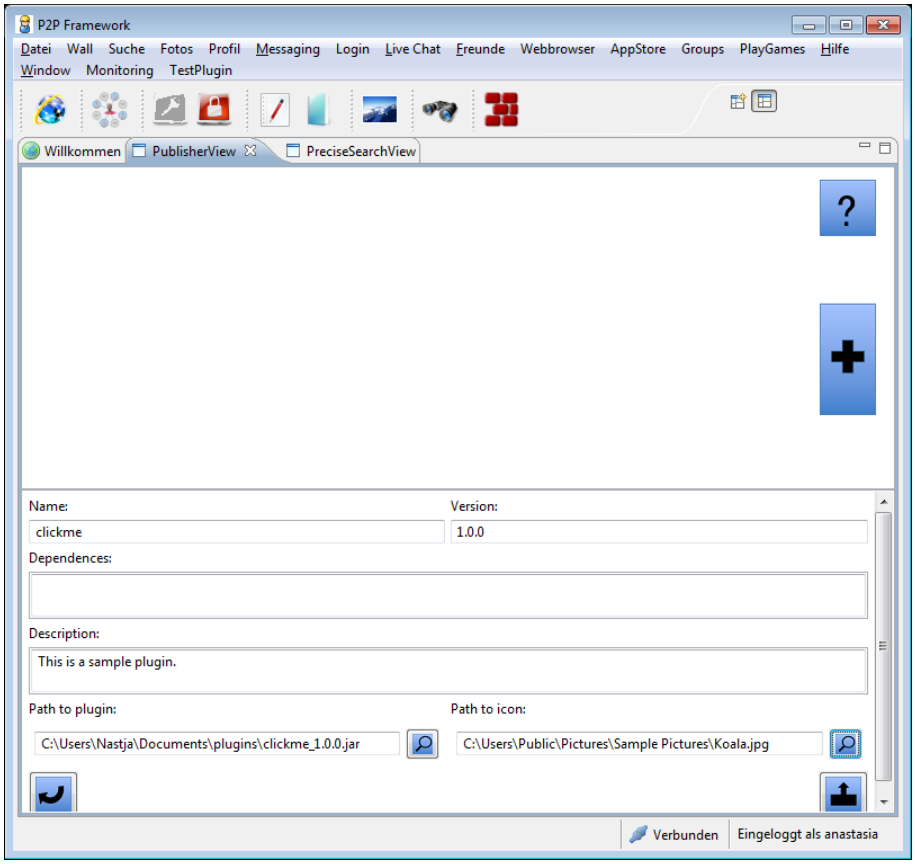


Figure 3.9: Publisher View: Publish plugin

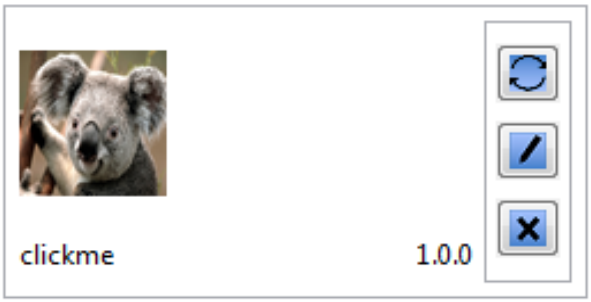


Figure 3.10: Publisher View: Published plugin

Precise Search View

With precise search function, user can find the already published plugins directly from the *precise search view* (see Figure 3.14). In order to use this function, the user needs to know the plugin name exactly. If the user enters the plugin name, he can apply the search by clicking the ”Search” button. After that the user will get a list of version as the result. He can choose one of the versions to check the detailed information under the drop down list of the versions. The target version of plugin can be downloaded and installed when the user clicks the ”Download

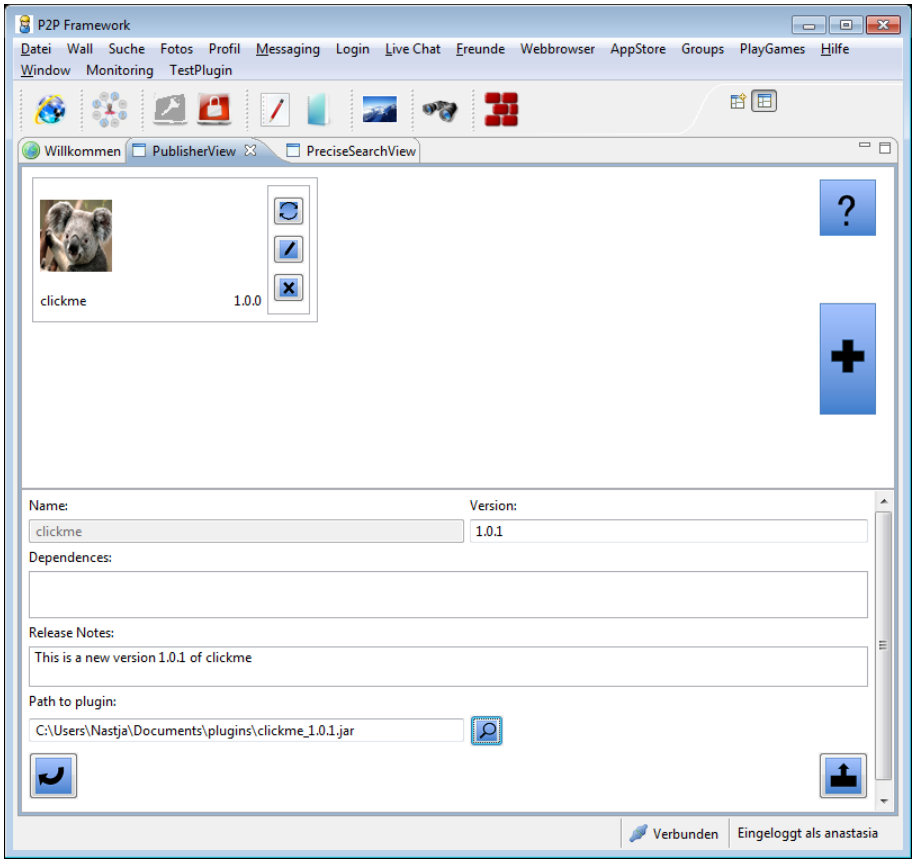


Figure 3.11: Publisher View: Release new version

and install the Plugin" button on the detailed information of the plugin.

Installed Plugins View

Each plugin will be shown as an item in a list on the *installed plugin view* (see Figure 3.15). After the download and installation of a plugin, user can find it and make some further management on this view. If the *installed plugin view* is opened before the installation, user needs to click the button "Refresh" to refresh the list and the detailed information of new installed plugin will be shown. The plugin user has the right to start /stop /uninstall the installed plugin, or update it if there is a new version available in the network.

Plugin Export

The plugins, which can be used in Appstore should be development in accordance with the Plugin Development Guide, see Appendix B. There are further rules which the plugin provider should pay more attention on.

The plugin ID should be identical with the plugin name which the provider enters later by plugin publishing. After the development of the plugin, it should be exported as a *.jar* file to suit the demand of plugin publishment in Appstore.

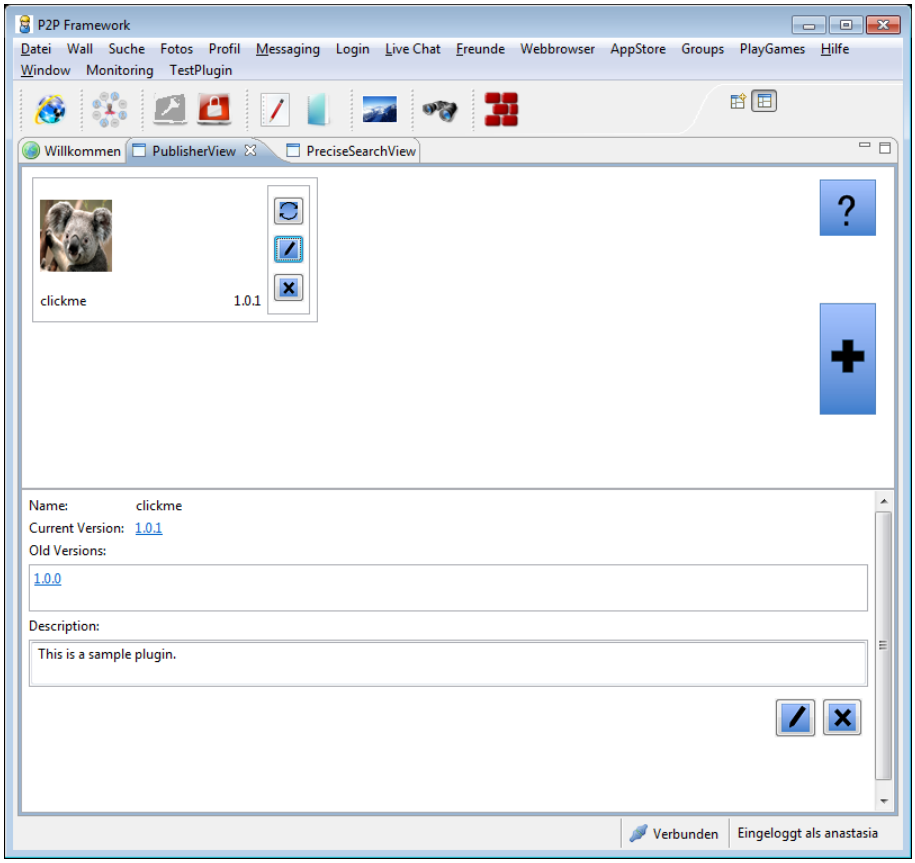


Figure 3.12: Publisher View: Edit whole plugin

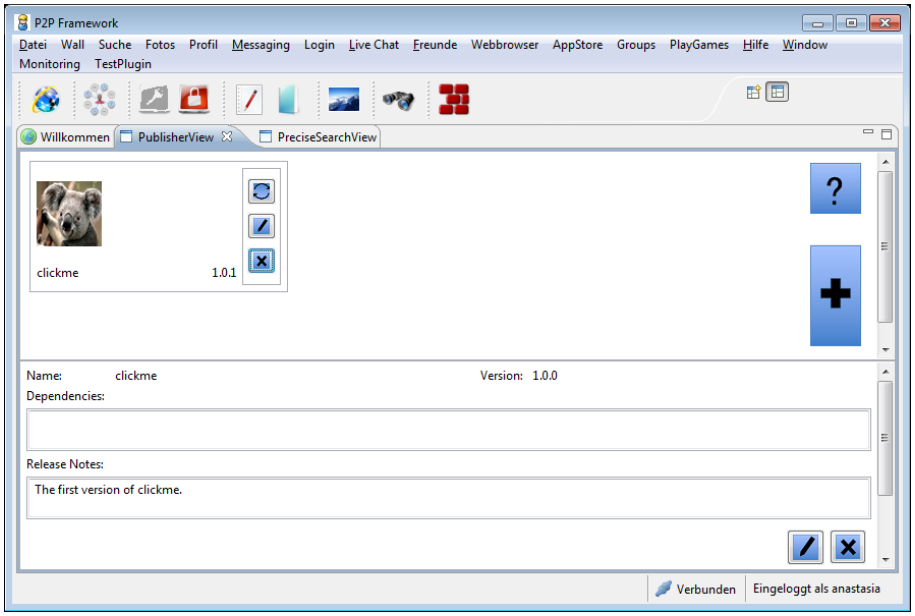


Figure 3.13: Publisher View: Edit single plugin version

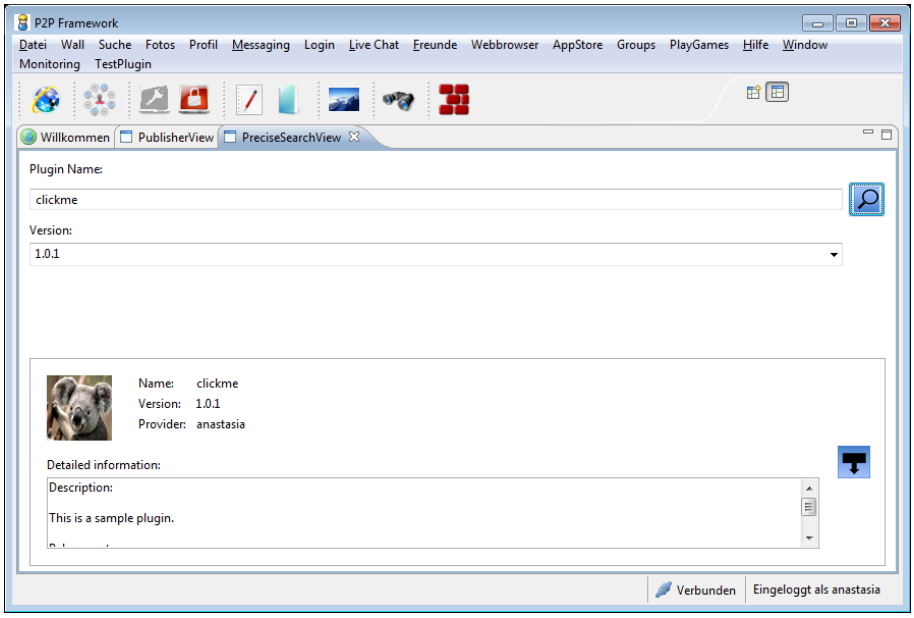


Figure 3.14: Precise Search View: Search the plugin

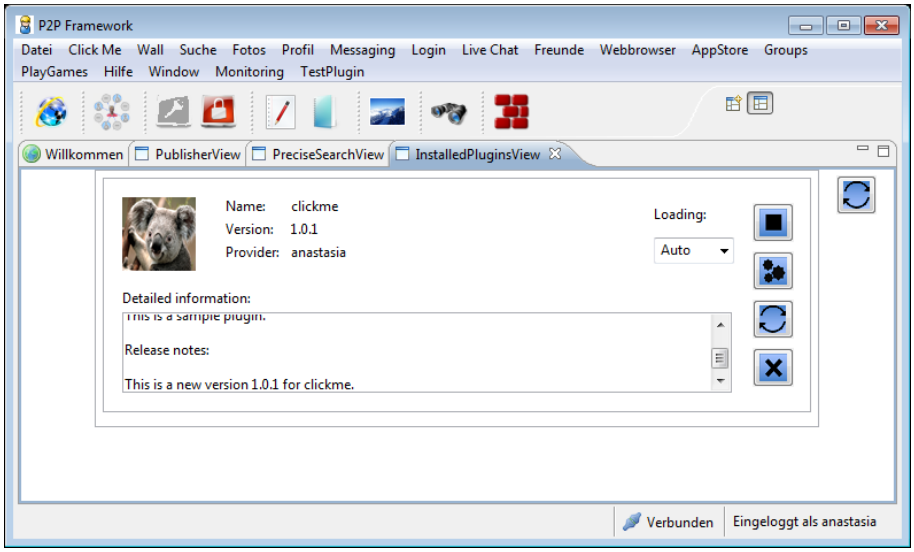


Figure 3.15: Installed Plugins View: List of installed plugins

To export a plugin as a *.jar* file, user should choose ”Export” from the context menu (see Figure 3.16). Then the option ”Deployable plug-ins and fragments” under ”Plug -in Developments” should be chosen (see Figure 3.17).

In the dialog window after that the user should choose a store destination for the exported plugins (see Figure 3.18). A new folder with the name ”plugins” will be generated and the new exported plugin will be stored there.

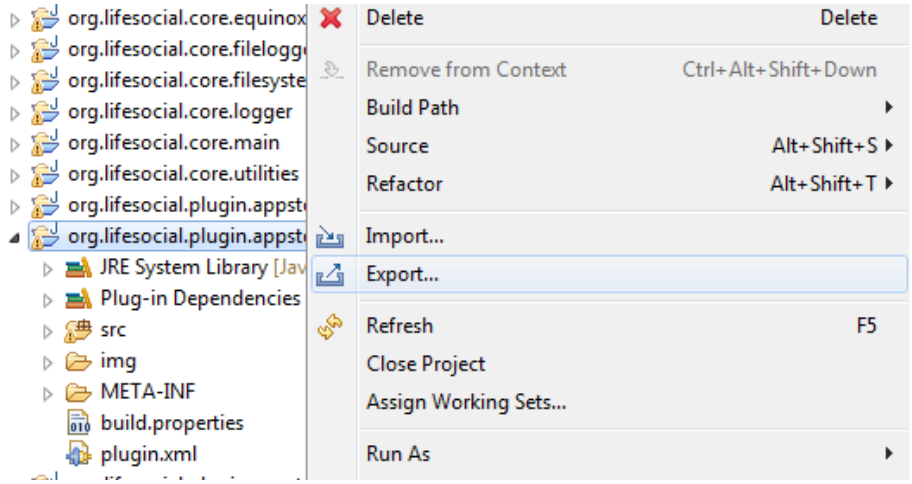


Figure 3.16: Plugin Export: Context menu

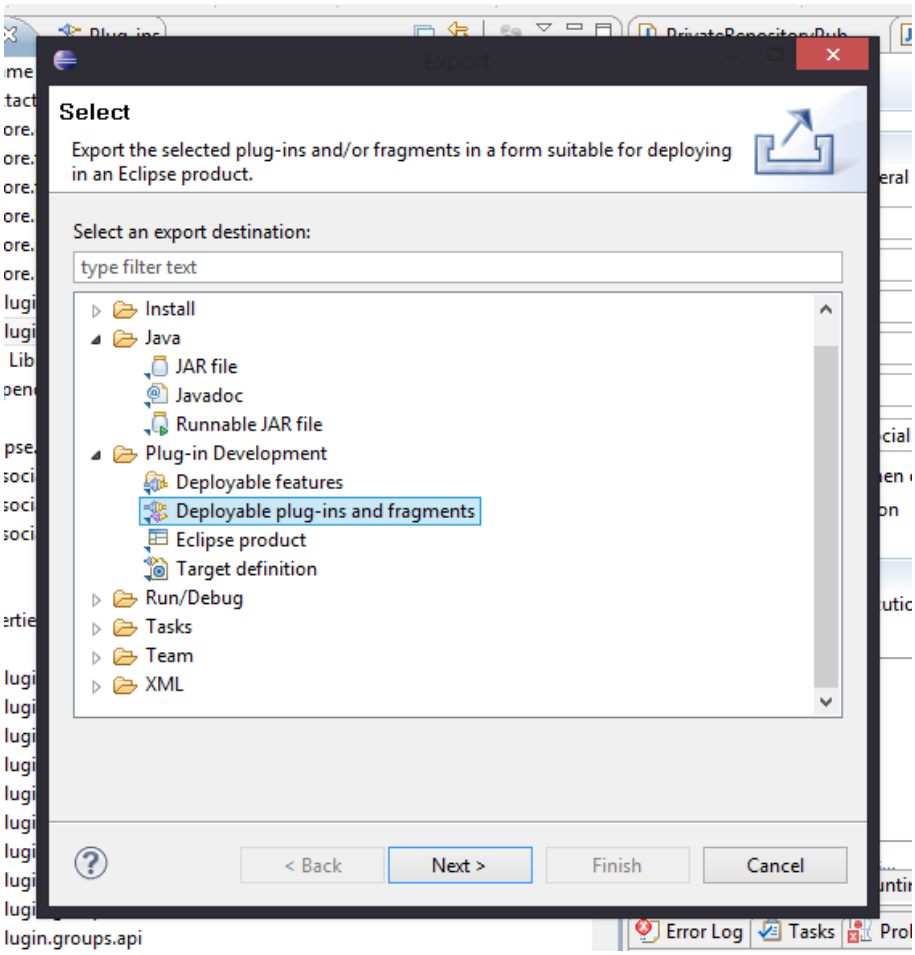


Figure 3.17: Plugin Export: Export deployable plug-ins and fragments

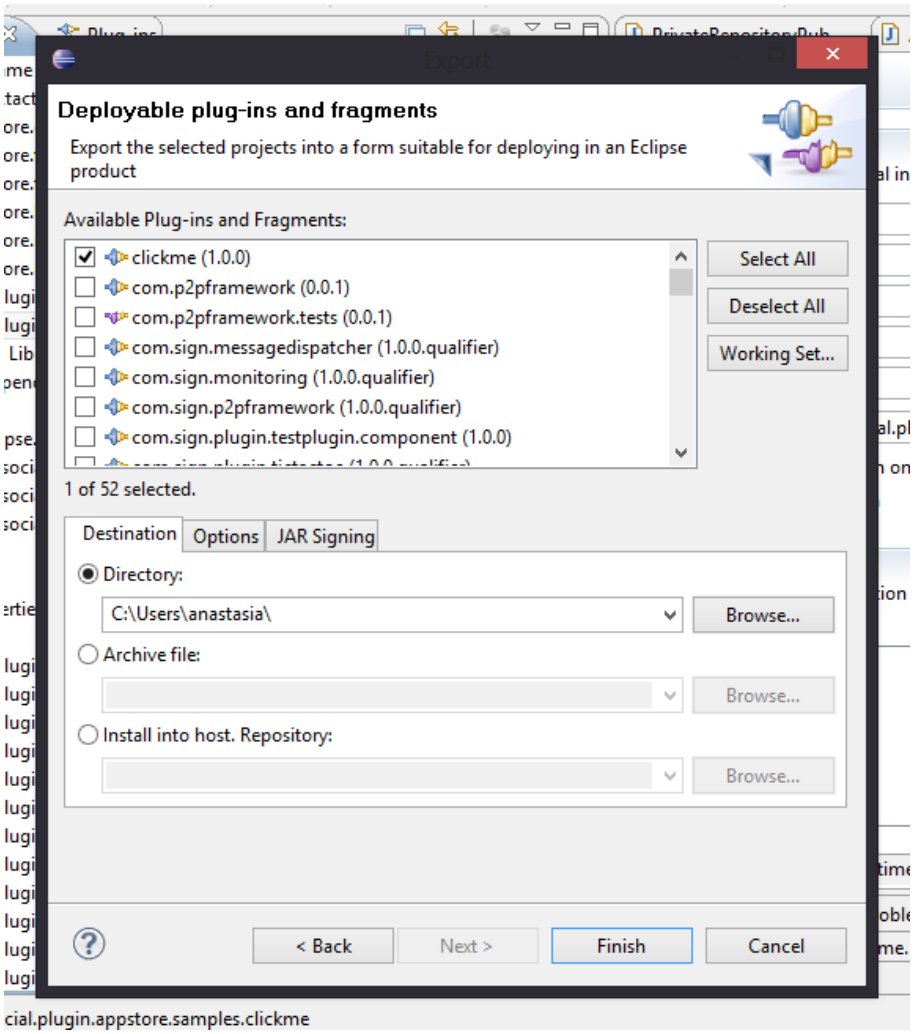


Figure 3.18: Plugin Export: Output directory

4 Graphical User Interface

As criticized at the last CEBIT presentation, the GUI of LifeSocial needed to be improved. Therefore the decission was made to use the perspectives of Eclipse RCP. The perspectives can be seen as templates for the arrangement of the views, that are provided by the plugins. With these perspectives the users of LifeSocial have the opportunity to create, save and switch their own arrangements of views, so that the graphical user interface can be fitted to the requirements of each user.

4.1 Perspectives

Perspectives are an essential component of the Eclipse RCP. In fact the graphical user interface of the RCP application becomes more flexible by using perspectives. Each perspective can be seen as a template that is holding the arrangement of different views. The user of the application is able to rearrange these pespectives just as create new ones. Furthermore, each plugin itself can provide perspectives that are arranging views provided by itself and the other plugins.
By including perspectives, the last state of the workbench will be stored and restored after start up of the application. But if at least one view of the perspective is required being logged in, it is throwing exceptions, since the view tries to get some content after opening it without any id of a user. That means, a propper use of the perspectives is only possible, if the user is already logged in. To solve this problem, a new structure of the views has to be implemented. The new views should distinguish whether the user is logged in and is able to see the content of the view or not. This problem was not solved during the project group so that it has to be done in the future by the next project groups.

4.2 Futher GUI improvements

Most time only one user will be using a single installation of the LifeSocial application. To improve the usability for that scenario, the AutoLogin is introduced. By using this option, the user can be logged in automatically by each start of LifeSocial.
For the setup, the login name and the password have to be set into the preferences just as the option for the auto login has to be checked. The preference page can be seen in Figure 4.1.
Unfortunately the password cannot be encrypted in a secure way because it must be readable by the local application. Thus, it is stored in plain text.

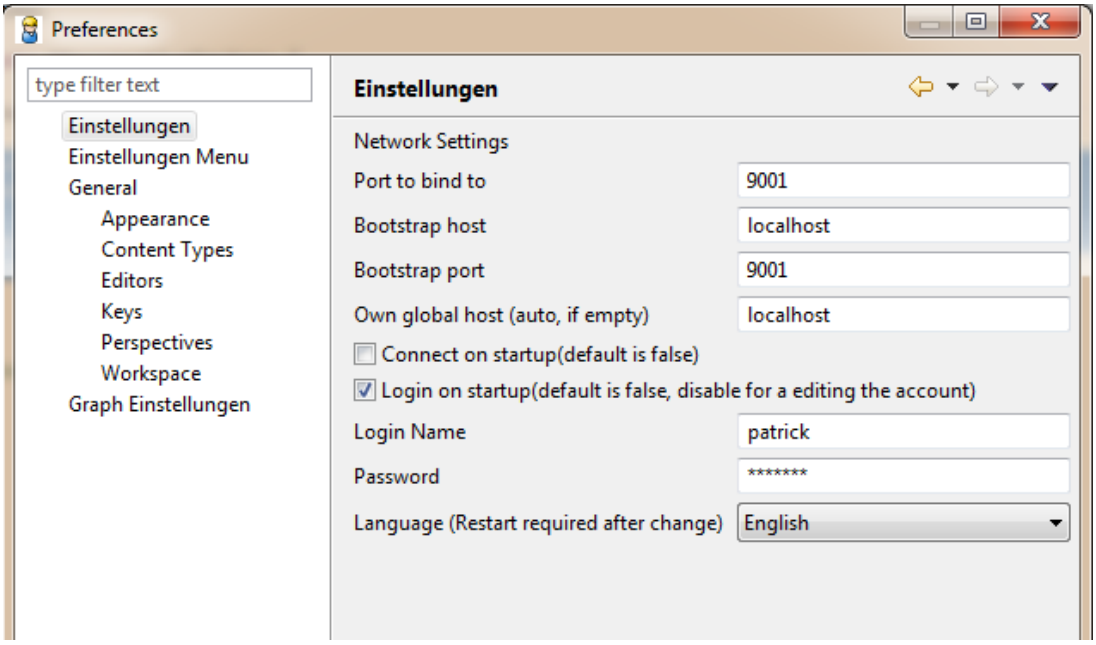


Figure 4.1: Preference page with the auto login option.

5 Framework

In this chapter we elaborate on the work of our project group concerning the framework layer on top of the overlay. The use of this layer is to provide a wide range of basic functionalities that can be used and combined by plug-ins of LifeSocial. One important aspect we focussed on are distributed data structures and access control to them. Distributed data structures are an important element since they provide a lot of functionalities which go far beyond the basic put and get operations of the DHT. Such data structures can be used, e.g., for walls, chats or forums. In Section 5.1 we present our solution for the implementation of a distributed tree that can be used, e.g., for hierarchical structures. Section 5.2 addresses the question how we can distribute keys for groups of users and how we can allow such groups to form arbitrary complex hierarchies. Based on this key management we developed a read and write access control for single items in the DHT as well as for our distributed data structures as described in Section 5.3. In Section 5.4 we present a basic approach for a voting functionality which can be used, e.g., for plug-ins like polls, doodles or a like-button. Finally, Section 3.8

what is presented regarding the appstore?

5.1 Distributed Tree

A distributed tree structure can be useful for several applications. It is, for instance, needed for hierarchical structures like file systems, where a lot of data has to be stored. If all the data was stored on one single node, this node would have to store huge amount of data and have to deal with a lot of requests, the single point of failure rises. With the help of the Distributed Tree this data is distributed over multiple nodes within the network. The data structure has to make sure that the requests are balanced among these nodes. The framework already contained distributed data structures like the Distributed Set and the derived Distributed Linked List. The Distributed Tree makes use of the Distributed Linked List by storing the childnodes of single nodes in particular Distributed Linked Lists. The Distributed Linked List splits a list of data items into single buckets of a specified size s. Each bucket is located at the hash "ListHash_BucketID" where BucketID $\in \mathbb{N}$. All operations for an item of the list has to be performed remotely on the backend containing the item. Therefore the operations are just sent to remote nodes storing the affected buckets which then operate on local data structures. The structure described above is shown in Figure 5.1, where each node is stored in a Distributed Linked List which is split up into several buckets and refers to the Distributed Linked List where its childnodes are stored.

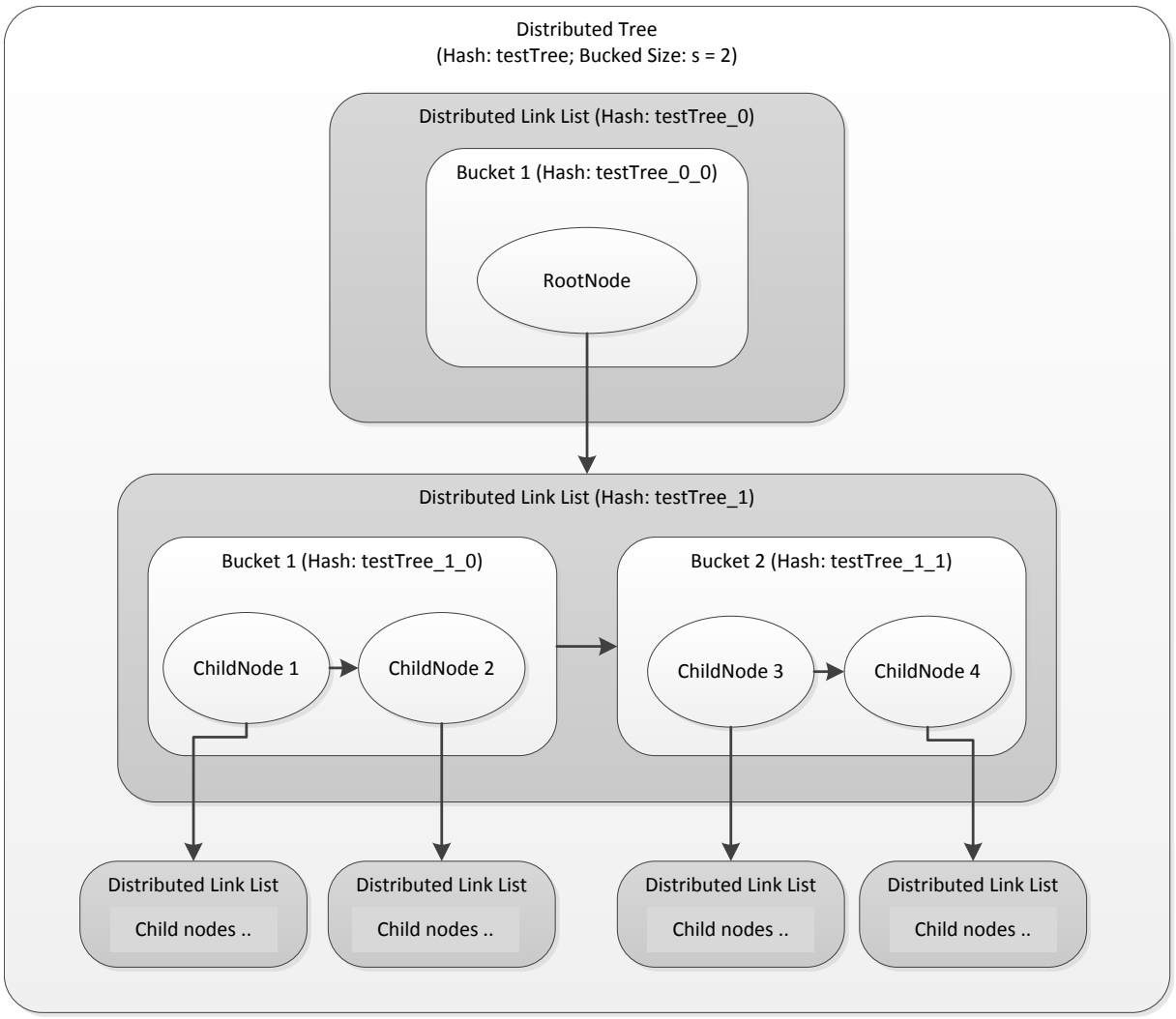


Figure 5.1: Example structure of a Distributed Tree for bucket size 2

5.1.1 Mechanism Description

Each node in the Distributed Tree is represented by the `TreeElement` class. The class stores the nodes' content, the hash for the Distributed Linked List in which it is contained, and the hash for the Distributed Linked List of its childnodes. A new Distributed Tree is stored at a specific hash. The Distributed Tree class only contains metadata and provides the basic tree functionalities like `get/setRoot()`, `get/setChild()`, etc. If a new node is created by calling for example `setRoot()` a new `TreeElement` with the given content and a new ChildList Identifier is generated by increasing the current ID count. The hash will look like "DistributedTreeHash_IDCounter".

5.1.2 Example

A new Distributed Tree containing Strings can be created by calling `new DistributedTree<String>(Identifier)`. This is also used to get access to an existing tree. You can also use the group decryption, where the content is encrypted and only decryptable to users in the specified group. It is now possible to set a root for the tree by calling `TreeElement<String> root = tree.setRoot("root")`. Childnodes can be added very similar by calling `TreeElement<String> rootChild1 = tree.addChild("1", root)`. The Distributed Tree class offers much more basic tree functions like `getTreeSize`, `removeChildElements`, `get/setChild()`, `removeTreeElement`, `updateContent`, `isEmpty`, etc. A Junit Test of the complete functionalities can be found in `com.p2pframework.tests`

5.2 Role Based Access Rights

In this section we describe our approach for the dissemination of keys as it is required for read and write access control in the network.

5.2.1 Existing Approach

The previous implementation of access rights to single files in lifesocial was managed by the *Group* class in the package `com.p2pframework/users`. A group can be created by every user and is accessible under a hash defined by the user. Each group consists of a public/ private key pair and a symmetric key. Its basic idea is to store the private and the symmetric key encrypted for each member with its public key. By this only members of the group are able to decrypt the current private and symmetric key of the group with the their own private key.

To make a file accessible only for members of a group, the content is stored as the basic `GenericStorageItem`. This item is now encrypted with the group's symmetric key and stored as a `SecureGroupDataItem`. To automatize this process, we added this functionality to the basic storage dispatcher functions `storeItem`/ `retrieveItem`. These functions can now be called with a group-hash and the dispatcher automatically decrypts/ encrypts the file with the groups symmetric key if the current user is member of the group.

Deleting a user has to guarantee that this user has no longer acces to any updated files of this group, therefore deleting a user requires a new private and symmetric group key, otherwise the

deleted user could have stored the previous keys and is still able do decrypt new items of the group. These new keys are created and stored in the group as soon as a user is removed. The old ones are kept in order to be able to decrypt files that have been created before the generation of the new keys.

5.2.2 Basic Enhancements

The basic goal we are focussing at in this section is to develop a kind of role based access control. The underlying idea is to use groups such that each role corresponds to a group. Because we want to allow hierarchical roles and combination of roles it is necessary to extend the groups in such a way that it is possible to define other groups to be members of a group. This is simply possible, because we can use the id and keys of a group instead of the user's id and key.

Hierarchical Roles

To create a hierarchy as shown in Figure 5.2 where a professor has the most rights at the top of the hierarchy, the assistants of the professors have less rights and the students have the lowest rights, we can create the different roles and add the members and roles to create the hierarchy.

If for example *Prof1* wants to access the time plan that is encrypted with the key of the role *Students*, he has to decrypt the key of the role *Assistants*. He is able to do so, because he is a member of the role *Professors* that is a member of the role *Assistants*. The exact functionality of actually retrieving the key is described in Section 5.2.3

Share an Item with Multiple Roles

To share a file with multiple roles as for example the Timeplan in Figure 5.2 with the roles *Students* and *Management* its necessary to have one key which is only accessible by members of these two roles. Therefore we have to create a new group containing *Students* and *Management* as members that provides the new key. The file can now be encrypted using the key of this new group.

Share an Item with the Intersection of Roles

To share a file with the intersection of roles as for example the Timeplan in Figure 5.2 with users that are members of the role *Assistants* and also members of the role *Management* we can add a list of roles into a group which is handled as a single member. To achieve this goal we have to encrypt the keys of the group twice. Once with the key of the role *Assistants* and second with the key of the role *Management*. In order to decrypt the key the user must have access to the keys of both roles.

Definition of Admin Rights

Another modification we made concerning the groups is the definition of admins for groups. Basically a group is managed by an administrator. The creator of the group automatically becomes an admin and can add or remove other users. We extended this concept in such a way

that now it is possible to define a set of administrators instead of restricting the number of admins to one. Admins are represented in a special admin-group which has its own keys. These keys are used to protect the group such that only admins are able to add and remove users from the group (for a detailed description of write access control refer to Section 5.3). Furthermore, the decision to manage admins in a special admin-group allows to reflect a hierarchy of groups also in the corresponding admin definitions.

5.2.3 Retrieval of Keys

In order to decrypt an item that is secured with a group key, it is necessary to know which group was used to encrypt the file. For that purpose we store the hash of the corresponding group together with the encrypted item inside the `SecureGroupDataItem`.

If the requesting user is member of this group, he can simply retrieve this key using its own private key. In case of hierarchical or combined roles we have to follow the hierarchy until we reach a group where we are member. From that point we can go backwards and decrypt the keys we need in order to finally decrypt the file. To find this group we execute a DFS (depth-first search) starting at the group associated with the file. Consider the example in Figure 5.2 and assume the file is shared with the groups *Students* and *Management* and *Prof1* who is not a direct member of *Students* and not a member of *Management*. The DFS first tries to retrieve the key of the group *Management* in order to decrypt the key the file is encrypted with. Because *Prof1* is not a member of this group, DFS goes backwards. In the next step the search tries to get the key of the group *Students* where *Prof1* is also not a direct member. Therefore we go deeper until we reach the group *Professors*, where *Prof1* is a member and able to decrypt the role key. From this point we can go backwards and use the key of *Professors* to decrypt all keys we need on the visited path until we are able to decrypt the file. This is efficient if we assume a hierarchy forming a tree with n roles, because DFS has a runtime of $\mathcal{O}(n)$.

5.2.4 Update of Keys

If we have to remove a member of a group or update the group keys because of other reasons it is necessary to update all keys among the groups below the affected group within the hierarchy. This process is managed by the user who initiated the update. All users of a group are able to initiate a key update, because a user possibly can loose or accidentally publish a secret group key. Consider the previous example in Figure 5.2 and assume that *Prof1* initiates a key update on his group *Professors*. Because he is an indirect member of *Assistants* and *Students* he can also initiate the key update on these groups. In some cases a right hierarchy might contain loops. For example a RoleA contains a role RoleB and RoleB contains RoleA. In this case RoleA first updates its keys and afterwards the keys of RoleB. Now RoleA holds its key encrypted with the old key of RoleB. To avoid endless loops, RoleB only updates its entry in RoleA (encrypt RoleA's key with its own new key) instead of initiating a complete key update.

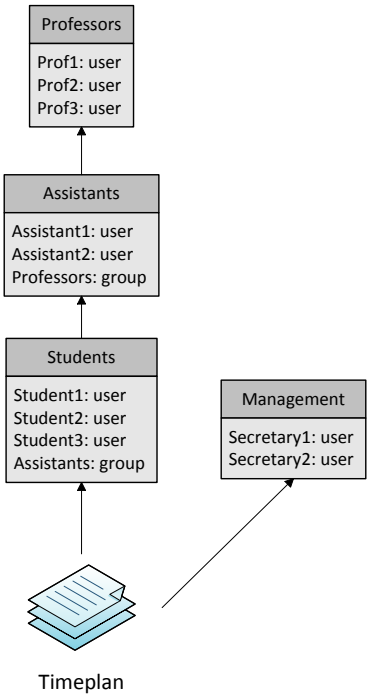


Figure 5.2: Example of a role hierarchy

5.2.5 Rolemanagement

All functions regarding the role management are provided in the `UserRoleManager`. The function `createUserRole` is used to create a new empty user role or a user role with members. A user role with members can be created by calling the function with an $ArrayList < ArrayList < Object >>$, where the inner ArrayLists represent intersections of roles (Section 5.2.2). For example a list of (RoleA, RoleB) would lead to allow access to all users which are member in RoleA and also in RoleB. A list of a single user or role would allow access to all users of this role. The outer ArrayList defines all roles that should have access to the role keys (Section 5.2.2). Therefore it is possible to define arbitrary combinations of roles. To add a member after the creation of the role it is possible to use the functions `addSimpleRoleRight` and `addCombinedRoleRight`. In order to remove a member of the role or update the role keys, the `UserRoleManager` provides the functions `removeUserRoleUser` and `updateGroupKeys`. These functions manage the complete key updates of the underlying hierarchy as described in Section 5.2.4.

5.3 Write Protection

The goal here is to develop and implement a write protection based on chryptographic keys. It means that an item in the network can only be (over-)written by its owner. To do so the public key of the owner (either the key of a single user or the key of a group/role whose members should be considered as owners) is stored together with the item when it is first inserted. Afterwards such an item can only be overwritten if it is signed with the corresponding private key belonging

to the specified public key. This signature is verified by the node at which the item should be stored and if it is not valid the storing node can simply deny the insertion of the file into the network and thus, the overwriting is not possible. To prevent replay attacks, not only the item is signed but also a random nonce stored with the item which should be overwritten and has been created during its creation.

5.3.1 Remote Operations

The write protection as described above only works if the node that issues the store request is the actual owner of the item. Because several functionalities in the framework are implemented by remote operations, further changes are necessary. In a remote operation the node that wants to change a file (subsequently called client) only sends a request containing these changes to a remote node (subsequently called server, the node that is responsible for the corresponding identifier of the item to be changed) which in turn performs the changes. In this setting the node that issues the remote operation usually is the owner of the file but the server is not. Thus, the server cannot sign the changed file appropriately in order to overwrite the existing file. To overcome this problem, we developed and implemented a protocol for remote operations whose basic concept consists of the following steps:

1. The client requests the current version of the file that should be changed from the server. If the client already has a version of the file, it also sends a hash of it to the server.

2. The server compares the received hash with the hash of its current version of the file. If the hashes differ, the server replies with the current version.

3. The client performs its changes locally and afterwards computes the signature of the modified file and the nonce of the file that should be overwritten. Afterwards, it sends a request containing the desired changes, the signature and the hash of the file on which the client locally performed the changes to the server.

4. The server checks whether its current version and the version on which the client has performed its changes are the same (compare the hashes). If they are not equal, go back to step 2. Otherwise, the server performs the desired changes and then inserts the file into the network using the signature received from the client.

Note that the goto in step 4 is necessary because in this case the changes performed by the client and the changes performed by the server are done on different versions of the file. On the one hand this is obviously not reasonable. On the other hand the insertion process would fail at the server's side because the signature provided by the client is based on a wrong nonce since different versions of a file usually have different nonces as it should serve as a replay protection.

5.3.2 Distributed Set and Distributed List

Beside the enhancement achieved by the modifications described above, we decided to implement further write protection mechanisms for distributed sets and lists. So far the situation for these data structures is as follows. If the data structure is secured with a group/ role the items are

encrypted with the corresponding group key and by the basic write protection only users of this group are able to add elements to it. Nonetheless, single elements of a set or list are not stored directly in the network, but encapsulated in a bucket (which contains several elements based on the split size). This implies that all member of the corresponding group are able to delete or overwrite single elements of the list, since only the corresponding bucket have to be signed. This behavior often does not reflect the intention. For example, if the wall of a user is implemented by a distributed list, then all friends (managed in a corresponding group) should be able to add entries. But additionally only the author of an entry should be able to delete or change its entry. Therefore, a second write protection based on the single elements of a set or list is required. This protection is basically implemented in the same way as described above. The only difference is that if a bucket should be inserted into the network not only the signature of the bucket itself is verified, but also the signature of the contained single elements.

Consolidation of Distributed List

Whenever an element of the list is deleted, it is overwritten by an artificial `DeletedItem`. By this we can ensure that an element can only be deleted by an authorized user because deleting an element exactly corresponds to overwriting an element. However, this approach causes another drawback: if an element is deleted the size of the list stays the same because a `DeletedItem` is inserted at the corresponding position. To handle this situation, we implemented two mechanisms. First of all each position at which a `DeletedItem` is located is stored in the metadata of the list. By that it is still possible to access an element at a certain position i because it is simple to calculate the i^{th} element of the list based on the knowledge about `DeletedItems`. If we, for example, want to access the i^{th} element and we know that there are j `DeletedItems` in front of it, we simply have to access element $i + j$.

Nevertheless, this approach has the drawback that we can never decrease the size of a list and it can contain a lot of `DeletedItems` that actually do not provide any information. The idea to overcome this drawback is that if an element of the list is deleted, the structure has to be contracted again, i.e. if the element at position i is deleted then each element at a position $j > i$ in the list has to be shifted one position to the left. Because such a shift can not be done by arbitrary nodes because a shift corresponds to a change in a bucket and buckets are signed it is necessary that a node which has a write access to the list performs such a consolidation. However, performing a complete consolidation would on the one hand put a huge burden on such a node since it had to compute the complete list after shifting the elements and the corresponding $\mathcal{O}\left(\frac{n}{splitsize}\right)$ many bucket signatures. On the other hand a lot of communication between the storing nodes would be necessary in order to avoid inconsistent states of the list, i.e. avoid loosing elements because a bucket could not be stored successfully. Therefore, it was necessary to develop a new protocol for the consolidate function which overcomes this problem.

Consolidation Protocol The basic idea of this protocol is to shift a `DeletedItem` one bucket to the right for each time the protocol is applied. By this a `DeletedItem` is shifted again and again until it eventually reaches the end of the list and we get rid of it. The advantage of this protocol is that the initiating node only has to compute the signature of two buckets and there are only two storing nodes that have to communicate in order to coordinate their actions.

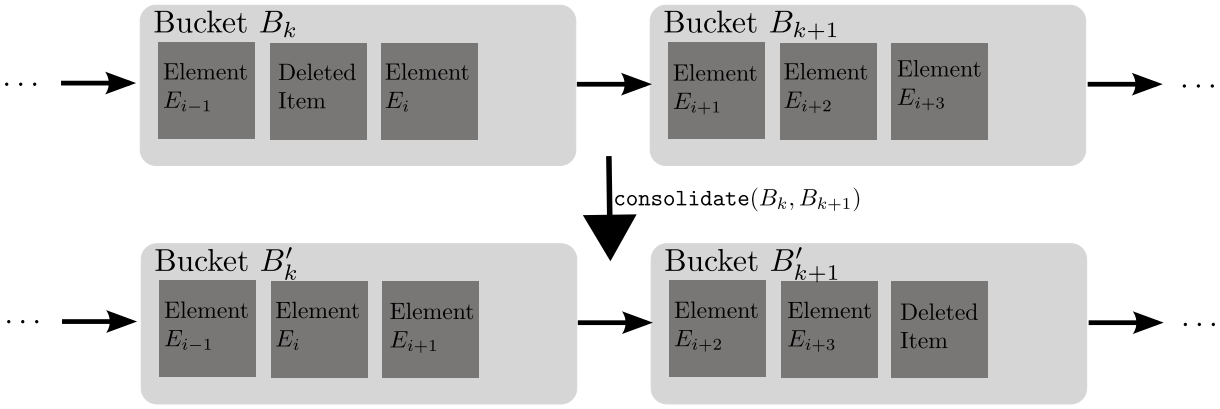


Figure 5.3: Buckets B_k and B_{k+1} before and after consolidation

For the following description of the protocol refer to Figure 5.3. Assume that bucket B_k contains a `DeletedItem` and `consolidate`(B_k, B_{k+1}) is called. These two buckets are consolidated so that afterwards the `DeletedItem` is located at the end of bucket B_{k+1} and the remaining elements E_j with $i < j \leq E_{i+3}$ are shifted one position to the left. The main steps of our protocol to perform this consolidation are as follows:

1. The node initiating the consolidation computes the buckets B'_k and B'_{k+1} and its corresponding signatures $Sig(B'_k)$ and $Sig(B'_{k+1})$. Both signatures are sent to nodes S_k and S_{k+1} which are storing B_k and B_{k+1}, respectively.

2. S_k and S_{k+1} compute B'_k and B'_{k+1} themselves and afterwards verify the corresponding signatures received beforehand. If a node encounters an invalid signature, it cancels its consolidation process.

3. If the signatures are valid, S_{k+1} notifies S_k.

4. If S_k receives the notification of S_{k+1}, it stores B'_k in the network. Afterwards it notifies S_{k+1}.

5. If S_{k+1} receives the notification of S_k, it stores B'_{k+1} after checking that B'_k is stored in the network as expected.

Note that because in step 4. and 5. the corresponding node waits for a notification of the other node and such a notification possibly will never arrive (e.g. some verification failed), the waiting time is bounded by a variable. If the notification does not arrive within this time span the consolidation process is canceled. The main idea of the communication between S_k and S_{k+1} is to avoid that elements can get lost. At first S_k has to wait for the notification of S_{k+1} to ensure that S_{k+1} already has done its verification which is based on B_k. Furthermore, because S_{k+1} should only store the new bucket B'_{k+1} (which does not contain element E_{i+1} anymore) if E_{i+1} is now contained in bucket B'_k, it waits for the notification of S_k and afterwards also checks bucket B'_k as stored in the network. By that we can ensure that a list element cannot get lost. The only unwished situation that can happen is that bucket B'_k is stored in the network but B'_{k+1} is

not (e.g. S_{k+1} leaves the system directly after step $4.$), which is, however, not as bad as the loss of an element.

The protocol as described above is called whenever a node performs a write access to the list. In this case a bucket which contains a `DeletedItem` is chosen at random to perform the consolidation. This randomness ensures that each affected bucket is consolidated over time. Other selection rules for consolidation like always choosing the first affected bucket might lead to unsufficient consolidation. This can happen, for example, in a situation where we alternately have a removal and a consolidation for several times in the first bucket. In this case the `DeletedItem` is shifted to the second bucket each time and thus, a lot of `DeletedItems` are gathered in the buckets directly after the first one but none can reach the end of the list.

5.3.3 Groups and Roles

A group (or a role) has the goal to provide common keys for all its members. To do so a group objects stores the corresponding keys encrypted for each member in an object in the network. Obviously, such a group object has to be protected against unauthorized overwriting and only the admins of a group should be able to modify the group object. Therefore, it is necessary that all admins of a group have a common public/ private keypair to sign the group object appropriately. To do so each group has a reference to a corresponding admin group which contains all admins as members. This admin group provides the keys to store the group securely. Because the admin group in turn has to be stored in a secure way the keys of the admin group itself are used for storage purposes, i.e. to sign the admin group when inserting or changing it. To conlcude, these mechanisms ensure that on the one hand only admins of a group can change a group and on the other hand that only admins can add or remove other admins.

Another modification that had to be done regards the references (referring roles) between groups. Because of the role based access rights we implemented the possibility to define hierarchical roles. To find keys in such a hierarchy it is necessary that a role A which has role B as a member creates a corresponding entry in role B. Due to the write protection, this is not possible anymore if these entries are stored in the group object B since the user who wants to add this entry is not necessarily allowed to overwrite the group object B. Therefore, the referring roles are now stored in a distributed list (outside of the object B) so that group objet B need not be overwritten. This distributed list is protected in such a way that any user can add an entry but only the owner of an entry can overwrite or remove it. In this setting the owner is the admin group of group A. Since everybody can add entries to the referring roles list it could possibly happen that unvalid entries are added. To overcome this problem the authenticity of an entry is verified whenever the corresponding part of the role hierarchy is traversed, i.e. before taking the edge from B to A it is verified whether the entry A in the referring roles list of B was signed with the key of the admin group of A.

5.3.4 Evaluation of Distributed List

In this section we want to review some results gained in a thorough evaluation of our approach. In all experiments we are basically interested in three metrics in order to assess the quality of our protocols: the traffic in the network caused by operations on our list, the number of messages

Operation	Distribution Setting A	Distribution Setting B
`get` (i)	$1/2$	0.93
`add` ($item$)	$3/8$	0.053
`remove` (i)	$1/8$	0.017

Table 5.1: Distribution of list operations in our test settings

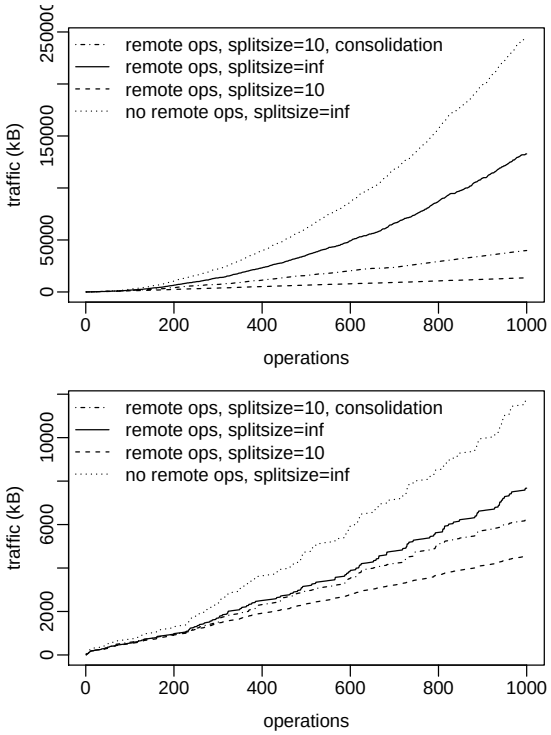


Figure 5.4: Traffic caused by list operations

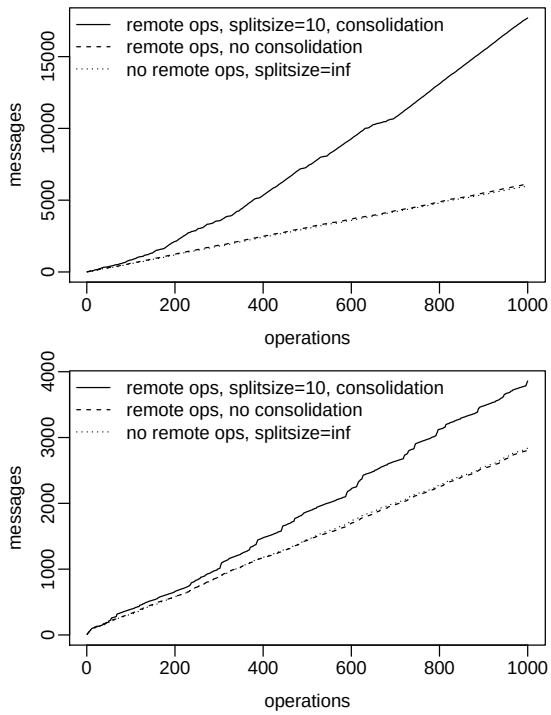


Figure 5.5: Messages caused list operations

exchanged and the ratio of the number of `deletedItems` in the list and the overall number of items in the list during the experiments (which coincides with the fraction of buckets that we could save if the list was consolidated (without `deletedItems`) all the time).

In order to get meaningful results we consider two different settings: in setting A we have performed 1000 randomly chosen operations according to the distribution in Table 5.1 and in setting B the same amount of operations according to a distribution based on an analysis of realistic user behavior in social networks as described in [5]. In both settings all list elements that are inserted and thus, all elements that are accessed later on are of size 1 kB. Note that the main difference between setting A and B is the number of write accesses. While in setting B just 7% of all access requests are remove or add operations, in setting A 50% are of this kind. Thus, in setting A the number of elements as well as the number of `deletedItems` will basically be higher than in setting B. In all of the following figures showing our evaluation results the graph at the top presents the results for setting A and that at the bottom the results for setting B.

In our first experiments we are interested in the basic gain of using remote operations and of splitting the list into buckets. In Fig. 5.4 it is shown that in both test settings a non-distributed list

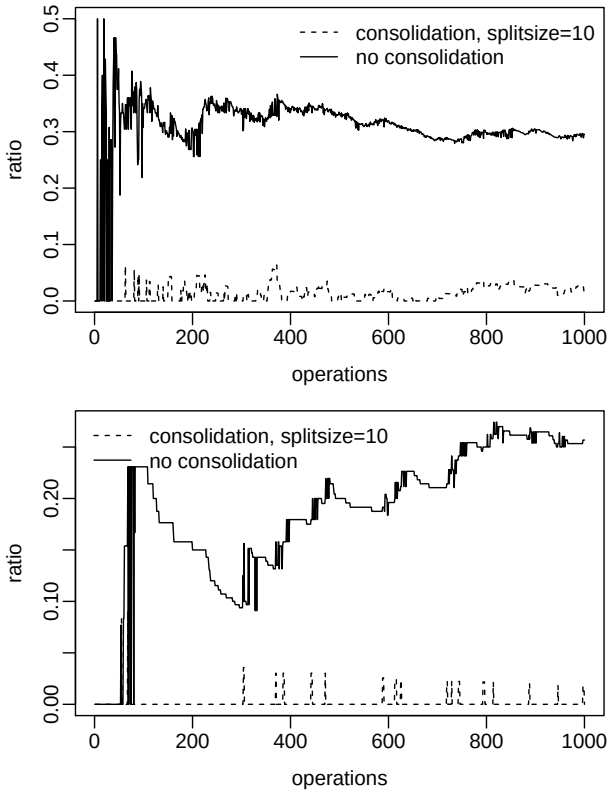


Figure 5.6: Fraction of `deletedItems` over time

without remote operations clearly causes the highest amount of traffic since for each operation we have to retrieve the entire list from the network. The usage of remote operations can almost halve the traffic and both approaches are definitely beaten by the combination of remote operations and the usage of buckets. If we also take the number of messages exchanged for the applied list operations into account, one can observe that this number does not differ for remote or non-remote operations (cf. Fig. 5.5).

Also note that the number of messages is independent of $splitsize$ since for all operations in our setting the affected bucket is known in advance. However, in both metrics considered so far an increase in the traffic is caused by the consolidation protocol. In fact, this is not a surprise since we apply this protocol after each access and thus, an additional exchange of messages and buckets becomes necessary as described in Subsection 5.3.2. Anyway this slight increase might be acceptable because of the considerable improvement regarding the `deletedItems` in the list as shown in Fig. 5.6. In both settings the fraction of `deletedItems` can be decreased from values up to 0.3 to fractions mainly not exceeding 0.05 while increasing the traffic not too much in comparison to a list not employing the consolidation protocol.

So far we have seen the actual advantage of using remote operations and buckets. However, we stuck to a $splitsize = 10$ and the suspicion obviously is that a higher $splitsize$ leads to a higher traffic since larger buckets have to be transfered. Because of that we also want to consider our metrics dependent on the chosen $splitsize$ of the list. In Fig. 5.7 and Fig. 5.8 the traffic caused by list operations is shown against the $splitsize$ for a list with and without the consolidation

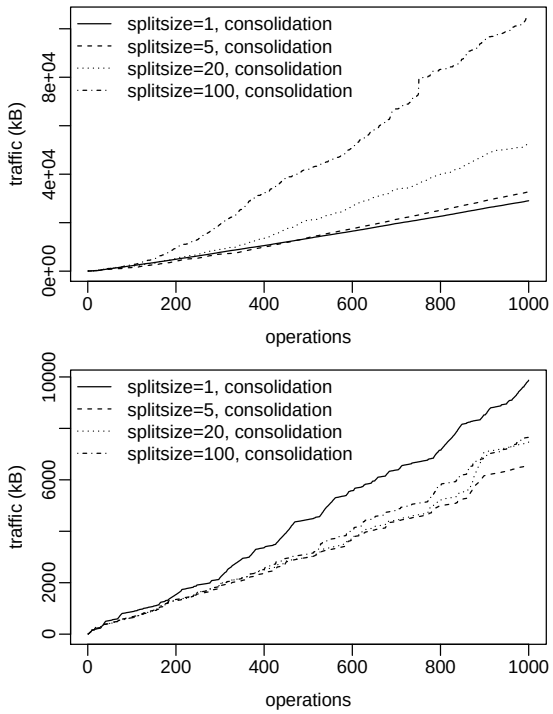


Figure 5.7: Traffic caused by list operations (with consolidation)

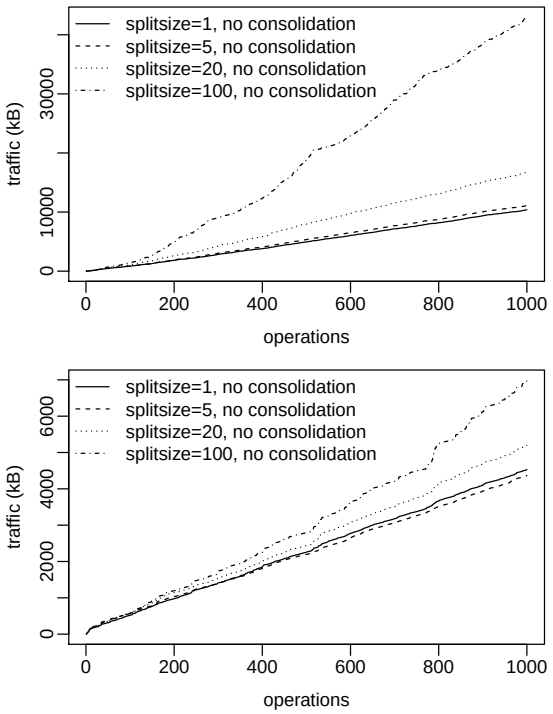


Figure 5.8: Traffic caused by list operations (without consolidation)

protocol, respectively. In general one can observe that a larger $splitsize$ usually leads to a higher traffic. An explanation for this might be the fact that whenever new elements should be added to the list (or two buckets are consolidated) larger buckets have to be transfered in order to compute the required signatures.

On the other hand larger buckets have another major advantage. Because elements are distributed among less buckets than for a small bucket size, it is easier to consolidate such a list. This is true because a given list is shorter in terms of the number of buckets if the $splitsize$ is larger. Thus, less consolidation steps are necessary. This results in a lower number of messages and `deletedItem`s in the list as shown in Fig. 5.9 and Fig. 5.10. To conclude, we observe that the size of buckets directly influences our three metrics under consideration. The choice always serves as a tradeoff between traffic, messages and how well our consolidation protocol works. For our setting, for instance, a $splitsize$ of 20 might be a good choice. On the one hand the traffic is not much higher than for smaller values (particularly in contrast to a $splitsize$ of 100), but on the other hand the number of messages is reduced and especially the consolidation works considerably better than for smaller buckets.

5.4 Voting

In online social networks like Facebook or Google+ different kinds of voting functionalities are widely used. There are, for example, those simple ”like” or ”+1” buttons but also other votings

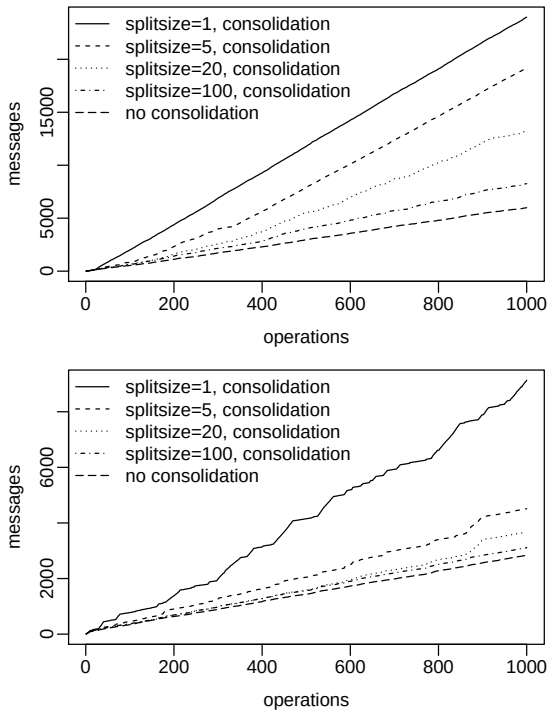


Figure 5.9: Messages exchanged due to list operations

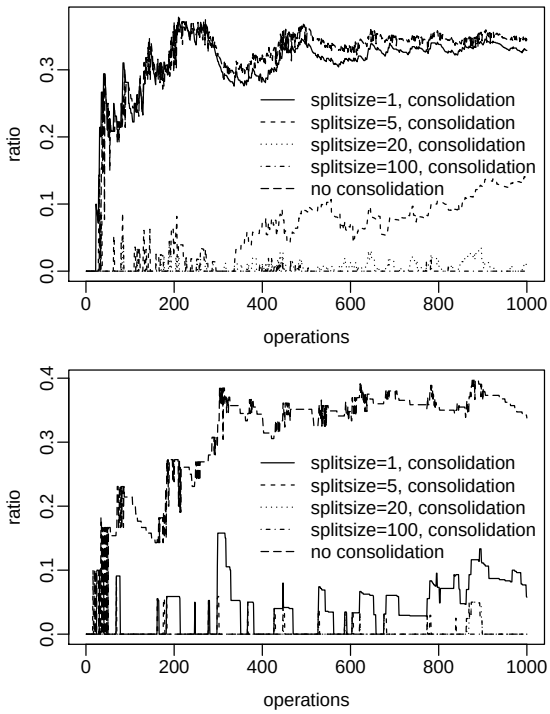


Figure 5.10: Fraction of `deletedItems` over time

where the user has the possibility to choose between several options, like in a poll, are popular. The goal was to develop the basic functionalities that are needed on the framework layer in order to allow plug-ins to provide different kinds of votings. In the following the basic approaches are described in more detail.

5.4.1 Public Voting

In a public voting every user should be able to vote and to see the results of the voting. The main challenge here is to ensure that no fake votes are given, that a user cannot vote under another user's name and that a user only can vote once.

The basic idea how to implement such a system is based on the remote operation approach that already was used in the last PG. If a user wants to create a new voting he first defines the possible `VoteOptions` that the other users should be able to choose as well as an identifier for this voting. Afterwards a `VoteList` that stores all information of the voting, like e.g. what options a voter can choose or which option was chosen by a user, is created and stored at the node that is responsible for the corresponding object. If a user wants to have access to an already existent voting in order to vote or see the results he just has to know the corresponding identifier.

In principle there are basically two approaches that can be used to add or get a vote. One possibility is that the user retrieves the whole `VoteList` modifies it and stores it again in the network. However, we decided to use remote operations because then less data has to be transferred and the node that stores the voting data and processes the remote operations can

perform several necessary verifications. To avoid votes under another user's name all votes are signed with the voter's private key so that the responsible node can verify the authenticity of the vote. Furthermore, the node can prevent multiple votes.

5.4.2 Private Voting

In a private voting only a restricted set of users should be able to vote and see the results. The mechanisms described for public voting are applied here as well. Furthermore, a private voting has a reference to a `Group` that defines which users should be able to participate in the voting. Such a group creates and distributes a symmetric group key as well as an asymmetric key pair. The symmetric key and the private key are only accessible by users that belong to the group. Now these keys are used to implement the private voting. Each vote is encrypted with the symmetric group key so that only authorized users can read them. Additionally, each vote is signed with the private group key so that only authorized users can give a vote since the responsible node verifies the signature when processing the corresponding message.

5.4.3 Conclusion

The voting functionality provides basic operations that are needed by a plug-in to implement polls or other elements that allow the user to assess content stored in the network. The decision to implement the functionalities as remote operations allows to verify votes and create private votings that can be only used by a predefined group of authorized users.

It is very easy to relate such a voting to any item like e.g. a photo album or a wall entry since it is only necessary to store its identifier defined during the creation of the voting. If the voting should be related to such an item it is very easy to restrict the access to it and the item to the same group of users since it is only necessary to set the corresponding identifier of the group in the voting.

All classes regarding the new voting functionality in the packet `com.p2pframework.Voting` have been tested using JUnit tests and a code coverage of about 90% has been achieved.

5.5 AbstractStorageItem - a basic model class

When creating plugins, it is obvious that many different kinds of data must be stored in the network. Therefore, one has to implement a *model* class which implements the `Serializable` interface. In addition these model classes often have more common requirements:

- Each object should be identified by a unique identifier which is used to create the pastry storage key. For this identifier the *Universally Unique Identifier* (UUID) standard is appropriate which is implemented in the `java.util` package [6]. When the UUID is randomly chosen, it is very unlikely that collisions happen. A UUID is a 128-bit number in hexadecimal form.

- Often, objects have associations to each other. These associations can be divided into to-many relations and to-one relations. For to-many relations a list of linked objects must

be available - the multiplicity is *0..**. For achieving this, the existing distributed data structures *DistributedLinkedList* and *DistributedSet* can be used. As the storage key for the list we can use the UUID of the current object concatenated with a static identifier for the reference (i.e. a reference name). For to-one relations just one link is stored, the multiplicity is *0..1*. To store these links, the UUID of the linked objects must be stored in the current object.

- Objects can depend on the existence of other objects. For example, a thread in a forum cannot exist without the forum. Referring to UML, we call these types of associations *containments*. When a child object is contained in a parent object, and the parent object is deleted, the child object has also to be deleted.

For these common requirements we introduce an abstract model class which can be extended by the concrete model classes. Concerning the requirement for a unique identifier, it creates a randomly chosen UUID in the constructor. The constructor also takes a *Framework* object as parameter which is used for retrieval, storage and deletion operations. When loading an object, one needs to call `setFramework` afterwards, because this reference cannot be serialized as it depends on the client.

The handling of the references is also done by the abstract class, though some additional settings can be made by the inheriting classes. To-one references can be simply set by calling `setOneReferenceID(String refName, UUID refID)`, which stores the ID of a reference under the specified name. The loading should be done using `getOneReference(String refName)`, which loads the object when it is called the first time, stores it in an internal cache and returns it. This concept is also called *lazy*. For using to-many references the method `loadManyReference(String refName)` must be overridden. It must return a *DistributedSet* or *DistributedLinkedList*, which inherits from *DistributedSet*. The storage key of the DDS should be created using the method call `getStorageKey(getID(),refName)`. To access the to-many references, the method `getManyReference(String refName)` should be used. Similar to to-one references, it returns the lazy-loaded DDS.

For specifying which of the references are containments, the methods `getOneContainments()` and `getManyContainments()` must be overridden, which return string arrays containing the reference names of the containments. When calling `deleteCascaded()` the object and all its containments are recursively deleted.

6 Overlay

LifeSocial is built on the top of FreePastry [7] overlay . Previous PG used the bundled version of **FreePastry 2.1** and hence was completely dependent on the traditional FreePastry strategies. The task of the overlay team was to provide enhancement to this overlay by modifying part of the source codes wheresoever needed. We worked on two open issues as a part of this team, namely, security and capacity awareness. Security deals with a safe and secure routing in the system. While on the other hand, capacity awareness takes capacity of various users into account so as to provide support for heterogeneous environments.

In this chapter we firstly present approaches which we came across during the research phase. Further, we present brief overview on ideas and strategies used to solved the corresponding problems.

6.1 Security

Security is one of the major concerns when it comes to social networks. We basically considered three open topics, namely, Sybil Attacks, Eclipse Attack and Routing Attacks. In this section we first explain the possible types of attacks and then provide a brief overview of the present approaches and argue their feasibility with LifeSocial. Later, we discuss the strategies we used to solve these problems.

6.1.1 Security Threats

In a real peer-to-peer overlay, adversaries are part of the network. The adversary can be a malicious node that follows a protocol which is harmful to the network. The malicious node can generate arbitrary packets with forged source IP or receive only the packets that are addressed to itself. The erroneous nodes which do not follow the protocol correctly are also considered as an adversary in peer-to-peer networks.

Attacks can occur at the application layer or at the network layer. At the application layer, an adversary can provide false information. For example, it can deny the existence of a data, or a node may report less bandwidth than its real bandwidth and in this way receives less traffic. Free-riding is also a problem that the selfish nodes benefit from the overlay by using the others files but do not share anything.

There are some other types of attacks such as returning wrong data or deny the existence of a certain data in the network. Distributed denial of service (DDOS) is also another type of attack, which is not related to the topic of this paper. In DDOS attack, the malicious nodes try to poison the routing tables of the other nodes and address a popular file to a wrong node. Consequently, a large number of requests will be forwarded to the target node, and this node reaches its threshold and stops responding to the other nodes. Undoubtedly two important types of attacks in the DHT

overlays are the Sybil attack and the Eclipse attack which can be used also for routing attacks. We are going to explain more about these two types of attacks in the following sections.

Sybil attack

In the Sybil attack the adversary creates many fake nodes and joins the network with numerous forged identities. As a result, the adversary gets a better view of all the network and can control other nodes more easily. In addition when all the fake nodes join the network the bootstrapping action cause a lot of stabilization traffic in the network. The problem will get worse if the overlay design, needs to transfer a lot of data in the bootstrapping phase. On the other hand, when all the forged nodes fail suddenly it causes to decrease the robustness of the network dramatically. These fake nodes potentially have the power to partition the network and create a parallel network to hijack the newly arrived nodes.

Eclipse attack

If the adversary keeps the control of the neighbors of a good node, then this valid node can be Eclipsed Which means it cannot communicate with the rest of the network any more because it is surrounded by the malicious nodes. The attack can be done by sending inaccurate updates to the valid nodes, consequently the routing table of the good nodes will be filled by totally wrong information after a while. That is why the Eclipse attack is also called routing table poisoning in some documents [8, 9]. The adversary can stop forwarding of the good node's messages to the rest of the network and also can hide the good node from the other nodes of the overlay.

The Sybil attack can be used to perform the Eclipse attack and the Eclipse attack can be used to smooth the way of the routing attack.

Routing attack

As a network layer attack, a node can forward a message to a non-existent address or a wrong destination. This leads to increase in network traffic and decrease in bandwidth usage efficiency . An adversary can also provide false information for the other nodes and pollute their routing tables with incorrect information. Another problem can be occurred in bootstrap phase.The action of sending request to join the overlay is called bootstrapping. If a new node which wants to join the network sends the join request to a malicious node, then the malicious node can lead the newly arrived node to a virtual overlay which consists of only malicious nodes. This situation is called network partitioning. If a new node bootstrap to such a partitioned network all its actions can be observed, and it can be controlled by the adversary easily.

6.1.2 Related Work

We came across many papers providing a variety of solutions to solve these problems. However, after considerable research we believed that concentrating on Sybil Attacks would also restrict the probability of all the other routing attacks. In this section we present the solutions we came across to solve such attacks.

Social network based solutions

SybilGuard [10] and SybilLimit [11] are two decentralized approaches for limiting Sybil attack. The basic property/requirement of each approach here was to have a social network which in a way became appealing for us. SybilLimit was basically an enhancement provided by the same authors of SybilGuard. Here we just discuss the core idea which was nearly similar to the both approaches and further we argue its feasibility with LifeSocial. In SybilGuard (SybilLimit) the edge of the graph represents a friendship between users. Each user creates its own partition of the network using random route and creates a route to each of its edges. When a node wants to find out whether the other node is an honest node or not, it checks for intersections between their routes. The verifier accepts the suspect node only if at least half of its routes intersected with the suspect's routes (SybilGuard and SybilLimit implement this in a different fashion). If this happened it means that the suspect node most likely belongs to the same region as the verifier. To do this each node further maintains two tables: registry table and witness table. These data structures are used to prevent nodes from lying about their routes. Communication between any two nodes proceeds with the help of symmetric key which can be distributed offline. However, this can be considered as one of the disadvantages as it is really difficult to have an off-desk key. Also, further we realized that the message distribution results in a lot of overhead as compared to other approaches. SybilLimit however provides a considerable guarantee by restricting the effect of malicious node to 10% compared to SybilGuard. This approach in conclusion was really appealing due to its applicability in social network however besides offline sharing of symmetric key the major point of concern was high messages overhead.

Centralized/Distributed Server

LifeSocial being completely distributed, having centralized server to verify identity is not suggested. However, this was one of the major fields which solved Sybil Attack effectively. The idea here could be later on transformed to distributed platform if found useful. We came across two useful solutions here:

Payment Gateway Here we can transfer to different amount of money to the new nodes and they can join the network only if they confirm the right amount of these two transactions. We can also ask them to send back these amounts of money if it is necessary. This kind of verification is already used by Paypal. However a completely decentralized implementation is not an easy task.

Computational Puzzles Computational Puzzles allowed to restrict the Sybil attack by a considerable amount. SybilControl [12] was one of the most interesting solution targeting Sybil Attack with the help of computational puzzles. It was demonstrated on Chord overlay and could be implemented on most of the DHT based overlays. The idea was to integrate puzzles with the periodic $keepalive$ messages which every node sends to each other to keep the routing table updated. It basically accomplishes two things:

1. A requestor can verify that the provider has solved the puzzle

2. A requestor can also verify that a provider has included requestors input in the computation.

Without going into much detail we discuss advantages and disadvantages of this approach. Firstly, it asks every node to solve the puzzle periodically. This can be viewed as an advantage if we consider just malicious node, as if it decides not to solve the puzzle then it is easy to find that out. However, if we consider honest nodes then they are forced to prove their honesty in every cycle. However, the major concern of this approach was its feasibility with heterogeneity. For example, it is really difficult to find a balance between the puzzle strength such that even the slowest node could solve it and also the fastest node has to wait for some period before proceeding.

6.1.3 Solution and Implementation Strategies

In this section we present you the overview of all the implemented solutions for the problems which were discussed earlier. There were basically three solutions which we think could lead to a stable and secure network. Here, by secure we mean that we can reduce the security threat posed by earlier system (the threats still exist but comparatively less). In the further subsection we propose our solutions.

K-Bucket

The default size of the routing table in Freepastry is $O(log_{2^b}N)$ rows and $2^b - 1$ columns and there exists only one address at each cell of routing table. To decrease the probability of being victim of adversary, we need to increase the number of node ids in the routing table. The idea here was to use k-bucket. In k-bucket each node in the routing table contains k entries in comparison to just one in the earlier strategy. The parameter of k is already defined in Freepastry but was not used initially (the value was 1). So, now we have a dynamic setup where we could change the parameter based on requirement. Currently, the parameter is set to 5, and hence each node contains at-most 5 entries in the cell. It is important to note that k-bucket is the base of our strategies and it is necessary to have a value more than 1 for the security solution to work. If it is set to 1 then we have only iterative routing. Parallel routing would be missing in such a case.

Parallel lookup

In a traditional/old system, a node would just query one node hoping that it would find the answers. However, in a system with malicious node it is really difficult to guarantee the success. We propose to use parallel routing, which means that at every lookup, the initiator node contacts multiple node instead of just one. All, the multiple lookup set are taken from the node's leaf set or routing table. Hence, due to providing multiple the contacted set it is really difficult to prone a threat. However, such a system would lead to a lot of network congestion. If we set the parallel routing parameter to k then at every step we increase it equally by k. This could result in a lot of redundant messages and hence network could fail. Here, we propose a short optimization technique (this technique is related to the next section as we would see).

Iterative Routing

This is one of the core parts of the solution. In a traditional/old system, a source/querier node is never aware of the message path. The role of the querier is just to start to lookup and nothing

more. However, if the querier becomes aware of the entire path during the lookup then it could keep the track of the solution and could identify whether it is reaching closer or farther to the solution. This is where iterative routing comes into the picture.

Source of Lookup Here, we propose the role of source node explicitly. Previously, there was no such a role in function of system, so to provide implementation of LifeSocial with the theory of iterative lookup, the role of source node was really an integral part of development. Our solution was firstly to store the node id of initiator of lookup in the message hence every node in the path is aware of the source node id. This the source node id must be serialized to avoid missing the content. Secondly, it is defined in code that only source node can forward the lookup ahead and no other node in the path.

Functionality The idea of iterative routing is that always the next hops are contacted by the source node and not by the intermediate nodes. So, the process would be as follows:

1. querierNode asks intermediate nodes for hints
2. intermediate nodes replies querierNodes with a list of possible next hops
3. querierNode contacts the hint nodes (intermediate nodes) and repeats the process till the answer is reached

Here, we see that at every step the source node could see a list of possible hops and further could select the ones it feels as the best. This basically reduces the probability of getting eclipsed and also further reduces any kind of routing attacks.

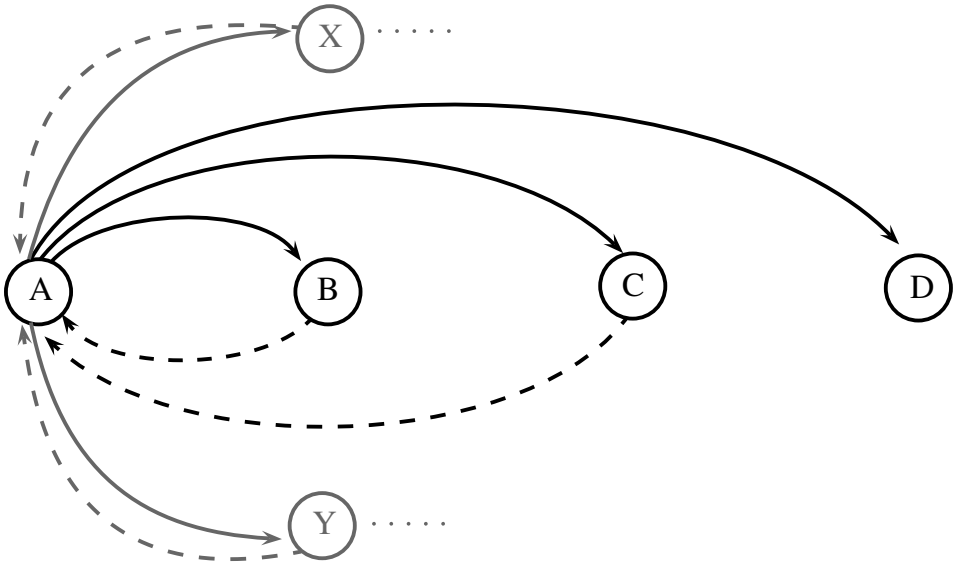


Figure 6.1: Iterative-Parallel Routing

Figure 6.1 demonstrates the idea of parallel-iterative routing. *Node A* here initiates the query by sending request to multiple nodes (this demonstrates parallel routing) and then each contacted node replies back with the corresponding next hops (this demonstrates the iterative routing). It is important to observe that the source is always fully aware of the path to the solution since only source node contacts the next hops. The task of all the other nodes here is to just provide a set of possible next hops. Source node could then send the message to the next hop based on algorithmic strategy. Since source node is aware of all the contacted nodes this gives us an opportunity for optimization which we see in the following section.

Optimization

Both the approaches discussed above could lead to network congestion as we are basically increasing the hops required to reach to the solution. And also further, there are many redundant messages generated during the entire lookup process and even after the lookup process. There are basically two kinds of optimizations required here:

Termination of redundant messages after successful lookup As we saw that parallel routing would keep sending messages with an exponential increase in the number of contacted nodes at every point. Here, we propose to use a unique number for every query. This basically helps to keep the track for each query. Since, we use iterative routing, the source could keep a track of all the open queries and closed queries. A query is closed basically when the solution is found and otherwise it is open by default. As soon as the result is received, source node could just close the query. So, any new message with the same query number could be rejected thus terminating the redundant lookups. However, this approach introduced the problem of **message lost during the node join** and hence we had to disable this approach for the further use. Here, we firstly, tried to initialize the open queries at the bottom layer, however, somehow one file was missing and hence we ocassionaly had exceptions that the query was not added to the open queries and hence was throw away while routing resulting in message lost. Further, we tried to add the query to the list at the source node itself which we though would definitely remove the earlier problem. But here too, we faced the same problem of message lost during the join process. **It is important for the future team to lookup over this approach in deep to find the solution to this error.**

Rejecting farther nodes Again, this solution is only possible when we have iterative routing. Here a source node keeps a track of all the contacted the nodes for that uniqueId which we introduced. Having a list of such nodes firstly helps in avoiding sending messages the same node. Also, we propose to use a list for the closest contacted nodes. Here, when a new hint is arrived then it follows two processes. Firstly, if it is already contacted it is thrown away and if not then the distance to the destination/target is found. When the distance is better compared to the closest k distance, only then the message is forwarded to this node. In all the other cases we just dump the nodes removing any question of redundancy.

Evaluation

Figure 6.2 shows the comparison between messages sent in the presence and absence of the iterative-parallel routing. As we can see in the Figure 6.2, security approach sends approximately double the messages compared to the earlier routing. The hike in the messages sent here is because of two independent approaches. Firstly, since messages are sent to many nodes (in comparison to one in the normal routing) the message sent count is bound to increase. However, the other major reason is that the use of iterative routing approximately doubles the message count, since every node here replies back to the source node (querier node).

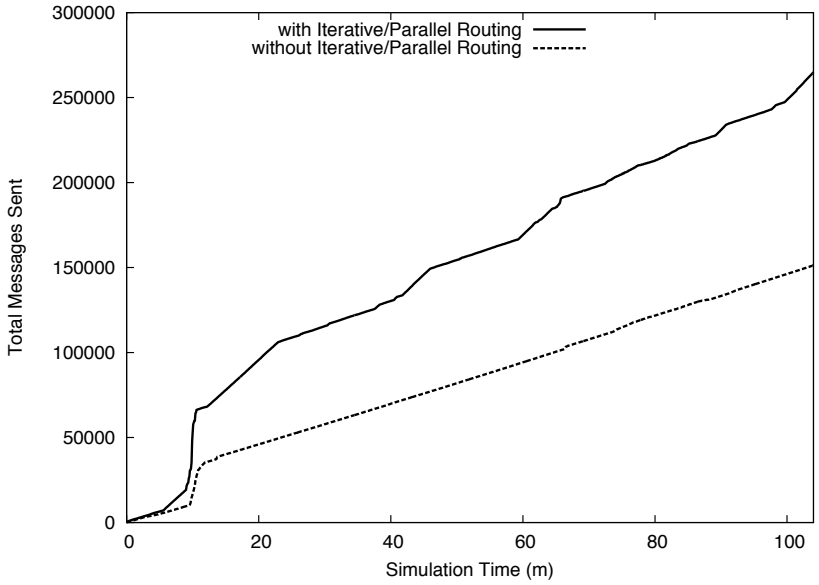


Figure 6.2: Messages Sent Comparison

6.2 Capacity Awareness

Mobile user devices like smart-phones, tablet computers, and notebooks allow to use Internet services at any time. This freedom in usage is a big gain for the user but it introduces challenges for application developers. Especially designers of p2p systems have to be aware of the different capacities of the nodes participating in the system. Nodes in a p2p system are the consumer and provider of services and resources at the same time. FreePastry tries to balance the load equally among all nodes in the system. This is done in a way that each node in the system is assigned the same workload. However, due to the heterogeneity of the node set, such a system is unfair and unbalanced. Nodes with little capacity like smart-phones may be overloaded with the assigned fraction of the workload. On the other hand, stronger machines may still have free capacity available that is unused. This does not only slow down the performance of the low-capacity devices, it also slows down the performance of the whole LifeSocial network because these machines become bottlenecks.

In the following we will present different approaches that we considered to support node heterogeneity and prevent weak nodes from being overloaded. Subsequently we will describe the model that we implemented, introduce the protocol in detail, and provide some test results that show the behavior of our implementation.

6.2.1 Related Work

To find a suitable solution for capacity awareness in LifeSocial we considered three approaches with very different techniques. HiGLOB [13], Adaptive Chord [14], and a third load balancing model proposed in [15].

One way to balance the load in a structured peer-to-peer system is HiGLOB. Each node in HiGLOB maintains a histogram which is used to locate under- or overloaded nodes. When such a node is found, nodes change their location to balance the overlay. Each node in HiGLOB maintains two key components, a histogram manager and a load-balancing manager. The histogram manager collects information of the load distribution in the network. This data is used to determine whether a node is under- or overloaded. The load balancing manager is then used to redistribute load when a node joins or leaves the overlay network (statical load balancing) and at runtime (dynamical load balancing). The major disadvantage of HiGLOB is the overload introduced by gathering the statistical data for the histogram and maintaining it. Even though HiGLOB proposes ways to reduce this overhead it appears that HiGLOB will have a too big impact on the performance of LifeSocial.

The key idea of Adaptive Chord is to introduce a second overlay only responsible for capacity awareness. Nodes and their ids are sorted according to their capacity on this overlay. Since in FreePastry, when a new node joins ring, it will find the node with closet node-ID to connect to. So our idea was to use the node-ID to represent the node's capacity information, then what we would have to build are CapacityNodeIdFactory and CapacityNodeFactory. And the main problem of a new capacity overlay is that it is only one-dimension (only capacity information). Furthermore, this approach doubles the routing and overlay maintaining effort of the nodes due to the second overlay. This second overlay would additionally introduce more points of failures and possibilities for malicious nodes to attack LifeSocial.

The third approach introduced by Graffi et al. in [15] seems to fit our needs more. It does not require a lot of overhead for communication or storage. This approach was implemented by us. The implementation and the model are described in detail in the following sections.

6.2.2 Model

The model uses the idea of indirect storage of data in the peer-to-peer network. Furthermore, more replication is introduced by not storing a data object only at the responsible node for the id of the object. It is additionally cached at nodes that accessed the object before. The responsible node maintains a list of all of such nodes, called providers, and can decide to forward a request to any of these providers. The responsible node decides which provider is chosen depending on their capacity. Providers with a high capacity are more likely to be chosen than providers with a small capacity.

The authors of the paper distinguish three different roles a node can take up. These roles are the following:

- **Requesters**
  A requester of an object is the node that initiates a lookup for that object to retrieve it from the network.

- **Responsible Nodes**
  A responsible node of an object in the protocol is that node that is responsible for the object according to Pastry. This is the node that would answer the request of a requester in the unmodified version of FreePastry.

- **Providers**
  A provider of an object is a node that previously retrieved this data object. It stores the object in a cache so that the node can provide it to other nodes.

In the model, the requester of an object first sends this request to the responsible node just like it would be the case in the original FreePastry. The responsible node will then decide whether to answer the request by itself or to answer the request not with a data object but with the address of a suitable provider. In case the requester receives the object the lookup succeeded. Otherwise, when the requester receives the address to a provider it sends a request to the provider will thereby differ the network traffic away from the responsible node to the provider.

The model can be illustrated best by examining the activity of the requester and of the responsible node after a request for a data object was initiated.

Requester Activity

The activity of a requester after it requested a data object is depicted in Figure 6.3. After it sent the request to the responsible node it waits for an reply. This reply does either contain the requested data object or the IP of a provider. In the former case the process is completed. In the latter, the requester will then request the data object from the provider specified in the IP address.

This provider will then answer the request with the object in which case the request succeeded. It is also possible that the provider does not contain the object anymore because of the limited size of the cache. The cache we use is a LRU (Least Recently Used) cache that will remove those objects that have been least recently used in case the cache is full. Another reason for a unsuccessful object retrieval from the provider is when the provider is failed or left the network. In all of these cases the requester will send a new lookup request to the responsible node.

Responsible Node Activity

Figure 6.4 shows the process that the responsible node follows to provide the data object. This step is capacity aware and balances the load in the system in a way that strong nodes get more workload assigned than weak nodes.

Each node maintains a list of providers for the objects it is responsible for. When such a responsible node receives a request it identifies the most suitable node from all providers (including itself) for the requested data object. When the responsible node itself is the most

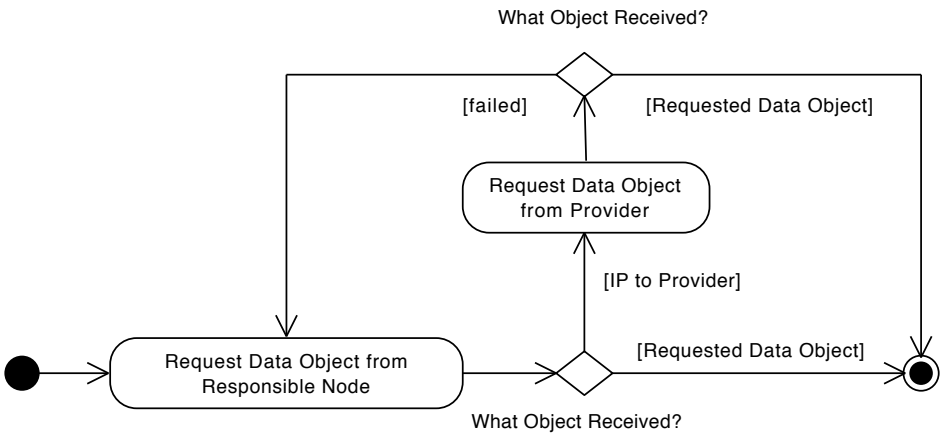


Figure 6.3: The activity diagram of the requester node

suitable node it transfers the data object to the requester. Afterwards the requester is added to the list of providers.

When a provider was chosen as most suitable node then the provider will check whether that provider is alive and can serve the object. When this is not the case the responsible node removes that provider from the list of providers and identifies another suitable node. But when the provider can serve the object its IP is sent to the requester.

6.2.3 Protocol

In this section we will describe the protocol and message flow, used for capacity awareness, in detail. This is done by illustrating the main functionality for the new data object lookup. Additionally the newly introduced function to find the most suitable provider among all providers of a file is described in detail in this section.

Lookup a Data Object

Here we will present the detailed protocol of a data object lookup. This part of the documentation contains all messages that are send during a lookup process. Figure 6.5 describes the whole process in case the responsible node chooses a provider to respond and that provider is available to answer the request.

First, the requester sends the normal FreePastry `LookupMessage` to the responsible node. This node identifies a provider and sends a `CanYouProvideMessage` to it. The responsible node uses this message to check whether the provider is still alive and if the cache of the provider still contains the requested object. In case the provider informs the responsible node that it can provide the object, the responsible node answers the initial lookup message by returning the address of the provider to the requester.

Second, the requester sends a `RedirectedLookupMessage` directly to the provider and starts a timer. Subsequently, the provider answers with the requested object. The requester stops the timer and uses the time the retrieval took and the size of the object to calculate the bandwidth

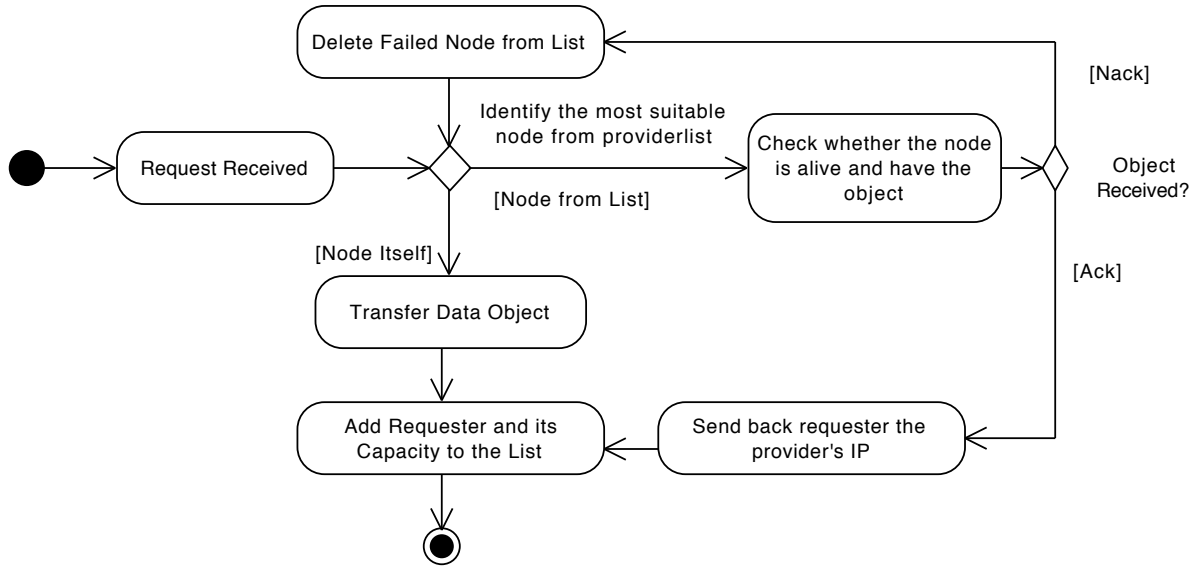


Figure 6.4: The activity diagram of the responsible node

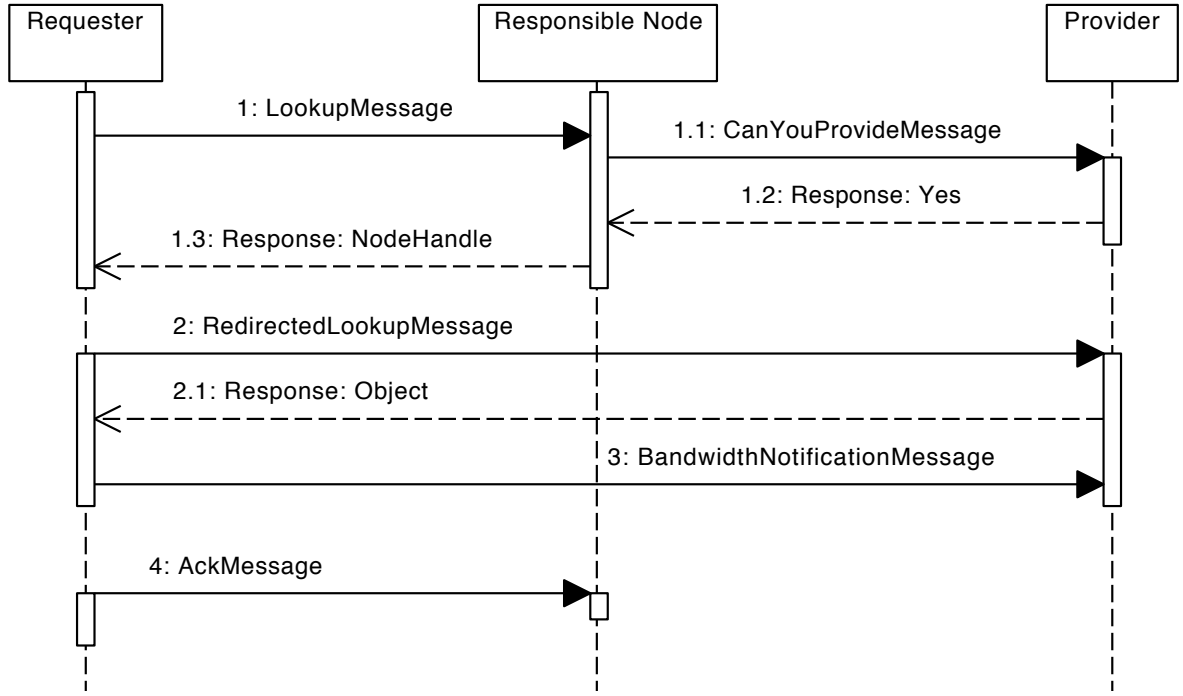


Figure 6.5: The sequence diagram of the lookup process

of the provider. After the calculations it will then send a `BandwidthNotificationMessage` to the provider to inform it about its bandwidth.

Last, the requester sends a `AckMessage` to the responsible node. This allows the responsible node to add the requester to the list of providers for the retrieved object.

Find Suitable Provider

When the responsible node receives a request, it checks its `ProviderList`. If the list is not empty, the responsible node has to decide which node to choose to be the provider. The mechanism of this selection of suitable provider is based on the discrete probability distribution. The advantage of using probabilities is, the node with highest capacity won't be used all the time meanwhile all the other nodes are just kept in idle. Now by probability selection, the workload is distributed among the nodes in the `ProviderList` in a more fairly way. Each node in the provider list gets to be assigned a value of probability P_i, the node with the highest value is selected as suitable provider. This value of probability P_i of node i is calculated based on the capacity of node i. If there are m nodes $Node_1...Node_m$, and each of them has the corresponding capacity $C_0...C_m$, the value of probability of $Node_i$ is calculated as :

$$P_i = \frac{C_i}{C_0 + C_1 + ... + C_m}$$

By using this function, the larger the numerator, which is the capacity of the node to be calculated, the higher the probability that it will be chose as the suitable provider. The denominator is the sum of capacity values of all current nodes in the `ProviderList`.

6.2.4 Capacity

Currently, three metrics for the capacity are considered: storage, memory and bandwidth. In the current implementation only the bandwidth is useful. This is the case because even if the responsible node has several providers in its `ProviderList`, it still has to store the data object. However, on the other side, due to the object may be transferred not only by responsible node, but also the provider, the bandwidth of the responsible node can be improved in this way.

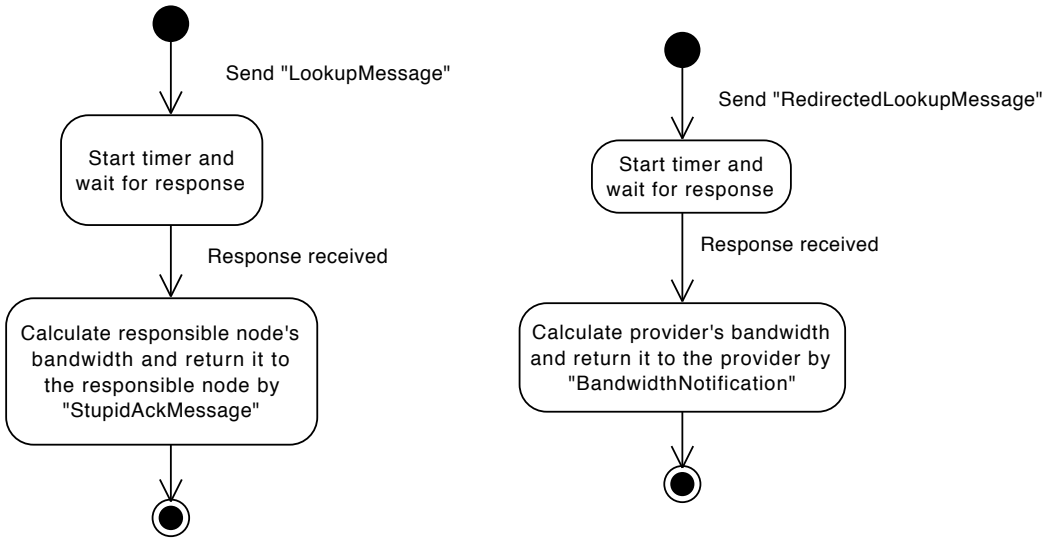
In other possible implementations the responsible node might delete the object from its storage when enough providers are present. In this case also the metric for disk space is useful.

Bandwidth

The bandwidth of a node is measured by calculating the size of message and round trip time between two node. The measurement can happen in two different scenarios depicted in Figure 6.6.

1. Requester sends `LookupMessage` to the responsible node and calculates the bandwidth once getting the response. Figure 6.6(a) demonstrates this process.

2. Requester sends `RedirectedLookupMessage` to the provider and calculates the bandwidth once getting the response. Figure 6.6(b).



(a) MessageLostMessageExceptions (b) NullPointerExceptions

Figure 6.6: Bandwidth Estimations

Assume the `LookupMessage` has the size $S_{message}$, the object has the size S_{object}. The upload bandwidth and download bandwidth of requester and responsible node are $B_{up_{req}}$,$B_{up_{res}}$, $B_{down_{req}}$ and $B_{down_{res}}$. The total time of this request and response process is T. Now we can get the equation:

$$\frac{S_{message}}{B_{up_{req}}} + \frac{S_{message}}{B_{down_{res}}} + \frac{S_{object}}{B_{up_{res}}} + \frac{S_{object}}{B_{down_{req}}} = T$$

Compared to the size of the object in the response, the `LookupMessage` has a relatively small size which means it occupies a negligible part in the total time. Meanwhile, in most cases the upload bandwidth is less than the download bandwidth, therefore the upload bandwidth is the bottleneck during the file transfer. So for our solution, we simply take the following equation to calculate the upload bandwidth of the responsible node.

$$\frac{S_{object}}{B_{up_{res}}} = T$$

Disk Space

The second monitored metric is the hard drive space. This metric can be important in future implementations of capacity awareness in LifeSocial. We measure the free space of the disk where LifeSocial is located using the following lines of code.

```
public String getPath(){
   String path = this.getClass().getClassLoader()
                                        .getResource(".")
                                           .getPath();
   return path;
}
```

6.2.5 Heterogeneity Configuration

We created a `heterogeneityConfig` file to facilitate testing. In this file, a number of parameters that are important for the heterogeneity implementation can be configured. An example configuration of this file is shown in Figure 6.7.

```
1  # enables the heterogeneity awareness (1 = enabled, 0 = disabled)
2  enableHeterogeneityAwareness = 1
3
4  # size of the heterogeneitycache in MB
5  heterogeneityCacheSize = 200
6
7  # timeout for a canyouprovidemessage in ms
8  CanYouProvideTimeOut = 5000
9
10 # maximal number of prodviders for each data object
11 providerListLength = 20
12
13 # number of replications of a data object in the system
14 # NOTE: It is important that #strongNodes >= replicationFactor+1
15 # in case weak nodes are present
16 replicationFactor = 2
17
18 # enables the object size calculation for the monitoring
19 # (1 = enabled, 0 = disabled)
20 objectSizeCalculationEnabled = 1
21
22 # enables the random selection mode (1 = enabled, 0 = disabled)
23 randomSelectionModeEnabled = 1
```

Figure 6.7: Heterogeneity Configuration File

6.2.6 Testing of Heterogeneity

The goal of testing the heterogeneity implementation was to show the correctness our implementation and performance enhancements. With heterogeneity, the load on a node overloaded node should be distributed to other nodes. Moreover, it was important to check whether the framework, plugins and the GUI are functioning with the heterogeneity implementation.

For that reason we tested the system with various kinds of tests with increasing scale. The main idea was to lookup a certain data object in the network multiple times. Our implementation should then choose different providers to lower the load on the responsible node.

To execute such a test in a small scale it is important to set the replication in FreePastry to zero. When the replication remains at the default (3), then each data object is stored on 4 different

machines. This makes testing the functionality of a lookup with enabled capacity awareness impossible when not more than 4 machines are present. Even with 8 machines, half of them already contain any data object. To set the replication to 0 one can use the configuration file described in Section 6.2.5.

In what follows in section we will describe the three different tests we ran to validate the correct behavior of our implementation and state the results.

Unit Tests

We ran this test before the new implemented feature was committed to the GIT repository. There are available unit tests created for different test purposes. The one used for testing the features of heterogeneity is `PastStorageTest` which is located in the following package path `com.p2pframework.tests`. The operations in this tests are creating nodes, storing and retrieving items, destroying nodes, and disconnecting from the network. In our developing phase, all the unit tests were successful.

Work Station Tests

After several features were implemented, we started to run the tests with two or three machines. This test is not only to check whether the implemented features work and to fix bugs. We also tried to simulate users' behavior to find the details we may have missed during the implementation. Once all the new implemented features were working well, we started to proceed with pool area tests.

Pool Area Tests

When the whole mechanism had been implemented in our program, and run through unit tests and work station tests successfully, it's time to run the test in a pool area. We applied 10 accounts in the pool area and installed `SmartGit` and `Eclipse` on each of them.

We used this test setup of 10 machines to conduct tests with heavy load to gather data proving the efficiency of our implemented capacity aware solution. In our test one user is uploading one photo. Afterwards this photo is consecutively requested by all nodes. The test lets each node start a new request for the photo after the previous request was answered. That way we were able to see that in the system with capacity awareness, the photo is retrieved from different nodes whereas in the original version of LifeSocial the file is only retrieved from the responsible node. This would be different when replication is enabled as described above.

Figure 6.8 shows the load on all nodes during the initial phase of this test with capacity awareness enabled. The load on the responsible node is shown with the red line whereas the load of all other nodes is depicted with blue lines. The figure shows that the responsible node answeres all incomming requests at first. In this moment there are no providers present in the system. Only shortly after this, the responsible node redirects many of the requests after this initial stage as the frequency of blue peaks increases.

Our initial goal was to gather statistical data using the monitoring tool that shows a reduced lookup time. This reduction was expected by us because the responsible node does not have to answer all requests anymore. However, we experienced that LifeSocial can not utilize the

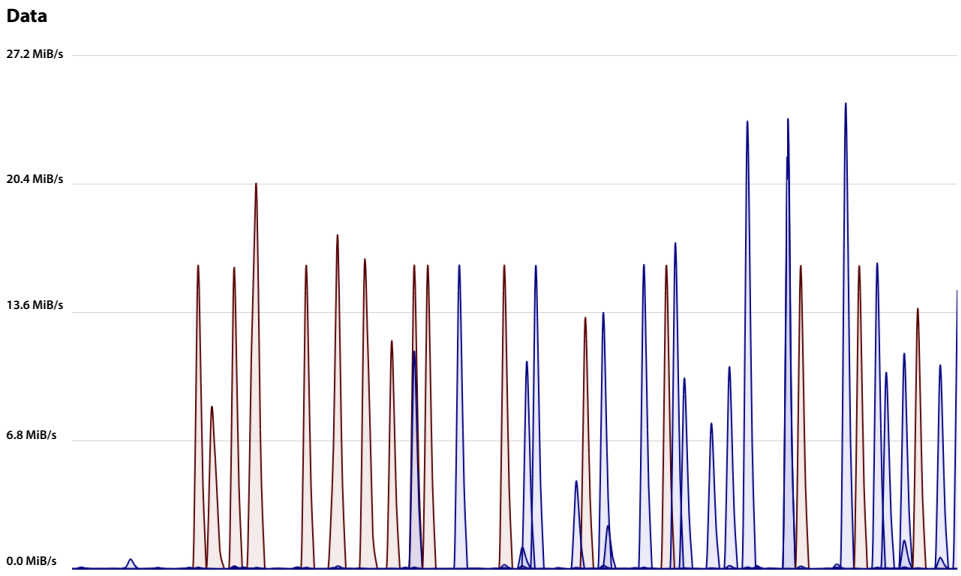


Figure 6.8: Start of the Test

bandwidth of the link between the pool machines fully. This means the bottleneck is not the bandwidth and therefore no lookup-time reduction was observable. In contrast, we were able to see a slight increase in lookup-time. We assume this to be the costs of our solution. We ran four tests to verify these results which are shown in Table 6.1. In each of the tests, the ten machines looked up the data item more than 300 times. Two of the tests were conducted with the heterogeneity approach being disabled and the other two with a enabled heterogeneity approach. We experienced an increased speed of the approach without heterogeneity approach of only 1.48%. This is only very little compared to the assumed gain in an heterogeneous environment in which some nodes have a low bandwidth.

Table 6.1: Successful Photo Downloads

Heterogeneity Awareness	**enabled**	**disabled**
Test Series 1	305	314
Test Series 2	310	310
Sum	615	624
Difference in Percentage	**1.5%**	

Problems

During the tests we encountered several problems and events worth noting.

- Photo Size Increasing
  When a photo was uploaded the data object representing the photo in the network had a significantly higher size than the original photo. The source of this problem was, that the

photos were saved in the Java object type LSImage.
Later during the project group we changed this in a way that photos are now stored in their raw type and not as LSImage anymore.

- Problems When Item Size Too High
  To utilize the bandwidth we experimented with the size of the photo. During this process we experienced that LifeSocial starts to operate very unreliable in the test scenario described above when the size of the photo exceeds a certain threshold. This threshold seems to be between 3,104KB and 4,300KB. In our tests we ran the testplan sucessfully with file sizes of 419KB, 1,161KB, and 3,104KB. However, tests with 4,300KB, 11,200KB and 24,600KB were unsuccessfull.

- Data Flow Recordings in Monitoring Tool Failures
  For analysis the performance of our implementation we intended to use the monitoring tool to gather data flow data of the system. However, we had to realize that this data must be faulty. We were not able to determine the bug in the implementation that produces this problem.

- Throughput of LifeSocial
  LifeSocial seems to be the bottleneck that restricts the speed of data item lookups in our tests. When testing we were only able to maintain an average throughput of about 200KB/s per machine. The capacity of the data link between the nodes lies well behind this and is expected to be at about 75MB/s.

6.3 Handling of Mobile Nodes

The handling of mobile nodes deals with nodes that are either too weak or too unreliable to contribute to the network. These nodes will usually be mobile nodes like smartphones or tablets that cannot provide capacity to the p2p system. There are two reasons why a node should not contribute to the system:

1. The node has not enough resources. These nodes do not have enough free hard disk storage to maintain data from the network, or the network connection or processor capacity is already fully utilized by other processes. Those nodes can simply not store any data, in case of the exceeded hard disk capacity, or would slow the system down in case of the exceeded processor and bandwidth capacities.

2. The node is too unreliable. An unreliable node is a node that leaves the network very frequently because of the user-behaviour or the network connection. Such a user might be a smart-phone user sitting in the train, for instance. Whenever the train enters a metropolitan area the bandwidth will be very high in most cases. However, after the train enters a rural area the bandwidth will reduce and may even be lost entirely.

   Such users that leave the network frequently only slow the system down when they are used for routing and data storage since every time they are used to lookup a data object or to route a message chances are good that the node left the network already and will not reply.

In our implementation we aim at showing the feasibility of ignoring mobile nodes for routing and data storage. To identify these nodes one could implement a capacity-check similar to our implementation for heterogeneity awareness or define instances of LifeSocial running on certain devices to be mobile. An instance running on a phone could then always be assumed to be mobile, independent of its location and availability.

In our implementation we focused on the technical implementation to show the feasibility and assumed each node with a even port number to be a weak node while each node with a odd port number is assumed to be strong. Our implementations aims at ignoring weak nodes for the following tasks.

- Routing messages
- Storing data
- Maintaining the routing connectivity

To achieve this, the implementation of FreePastry has to be changed at two different places. The routing table, and the ”best routing candidate” selection. The changes required in both places are described in the following two sections.

6.3.1 Routing Table

The routing table is used for routing any message. This can be an `InsertMessage` to insert a data object, a `LookupMessage` to lookup a certain object or node id, and other message types.

To ensure weak nodes do not participate in maintaining the routing connectivity and polluting the table (since they are not used for routing anyway) they are never added to the table. In other words, a routing table, independent of belonging to a weak or a strong node, does never contain any weak node.

But it is important that weak nodes are still reachable to deliver a message to. This is ensured by keeping weak nodes in the neighbouring set. Thus, weak nodes will still be reachable as long as there are not too many weak nodes present, or the neighbouring set is too small.

This danger can be illustrated in a small example. Assume the network of LifeSocial consists of 100 nodes. One of which is a strong node while all other 99 nodes are weak. This strong node will now have a empty routing table and relies solely on its neighbouring set for communication. Let the size of this neighbouring set A be 64. When the strong node wants to route a message to a node $n \notin A$ it will not succeed because n is not in the neighbouring set, nor does the strong node have any other node in its routing table that it could send the message to.

Thus, it must be ensured that the fraction of weak nodes in the system does not exceed a certain threshold. Furthermore, the size of the neighbouring set must be chosen wisely.

6.3.2 Best Routing Candidate Selection

The method `getBestRoutingCandidates` in `StandardRouter` returns the node a message with a certain target id should be routed to. This method returns a node from the routing table or the neighbouring set.

This method was changed in a way that it will not return a weak node from the neighbouring set. There exists one exception in which the method should indeed return a weak node. This is the case when the target id of the message equals the node id of the weak node. This is the case, for instance, when a message is sent for communication with a certain node. This behavior leads to the situation that a weak node does in fact allow storing exactly one item. The item with the same id as the id of the node itself. Every single node creates such an object which we assume to be the profile data of the node.

6.3.3 Protocol

This section will describe the protocols for storing a data object, retrieving a data object, and routing a message in an environment with weak nodes being present.

Storing a Data Object

A weak node is not responsible for storing data objects except its own profile data as mentioned above. Now suppose one weak node wants to upload a photo, it needs to find a node which has the closest id to this photo object's id. For that reason, the method `getBestRoutingCandidates` in `StandardRouter` is called and it will not return a weak node, as described above. This photo object will be stored in a certain strong node which is in the `leafset` of the weak node.

When a strong node tries to find a certain node to store its data object, the same happens.

Retrieving a Data Object

When a weak node wants to retrieve a data object, it sends a lookup message, and the message is relayed by a certain strong node in its `leafset`, afterwards the normal data retrieving mechanism is performed in the network. Once the data is found and forwarded back to a strong node which contains this weak node in its `leafset`. This strong node identifies the target of the message, containing the data object, being same as this weak node's id, then it forwards the message to this weak node.

Routing Messages

Each node uses its routing table to forward a message close to its destination. Since weak nodes are not present in the routing table of any node they will not be used for routing. Thus, the routing with present weak nodes operates in the exact same way as it does in the original FreePastry with the only exception that weak nodes are not considered.

6.3.4 Testing

We have tested a network with 10 nodes, when the replication is set to 0 and with at most 4 weak nodes, the result was good. We only experienced a very limited number of exceptions that we assume to not be the result of the mobile node implementation.

However, when we have the replication, we did experience some problems. Some nodes seem to become unresponsive. We believe the `FetchNodeHandle` function that is used to fetch the replicas of a certain object does not function well with our implementation of the weak nodes.

6.3.5 Problems

This section provides information of some open questions and problems that we encountered during the implementation and testing of the mobile node aware solution.

- **Weak Node Identification**
  Currently, as mentioned above, weak nodes are identified by port number, it is not a wise method since all nodes now can pretend to be a weak node and escape from contributing to the network. A possible solution can be that, identifying the weak nodes by types of terminal, for instance, all the cell phones users are weak nodes whereas pc and laptop users are strong nodes. Also it can be solved by introducing the reward and penalty mechanism, the more a node contributes, the more it benefits from the network.

- **Replication**
  Weak nodes are currently not working well with replication. Assume a network with 3 nodes which contains 2 strong nodes and one weak node, the replication is set to 2. Now one of these three nodes uploads one photo, and consequently there are two replications, totally there are three files. The replication mechanism knows there are three nodes in the network, so it delivers these three files to these three nodes, however the weak nodes will not accept it.

  Additionally some nodes seem to become unresponsive. In an environment with weak nodes. We believe the `FetchNodeHandle` function that is used to fetch the replicas of a certain object does not function well with our implementation of the weak nodes.

7 Testing

This chapter presents an overview of all the different kinds of tests which were conducted during one year of project group (PG). Testing allows to verify the functionality of a module/modules. Similar to the previous PG, we had two different kinds of test environments. An **unit test** environment in a person/team scope to verify his/their functionality individually. A **Pool area test**, on the other hand, provided an opportunity to verify the functionality of components developed by the entire PG. In the further sections of this chapter we provide a brief overview on the two testing approaches for a clear understanding.

7.1 Unit Testing

Unit testing, as said above, allows to verify the functionality of a certain module. So, we basically have unit tests for mostly all the functionalities developed and it was mandatory for every developer to perform such a test before committing new functionality to the repository. In addition, to 90 JUnit tests developed by previous project group we developed a few more such tests to verify the new functionalities. Figure 7.1 shows a snapshot of the eclipse environment for JUnit tests project.

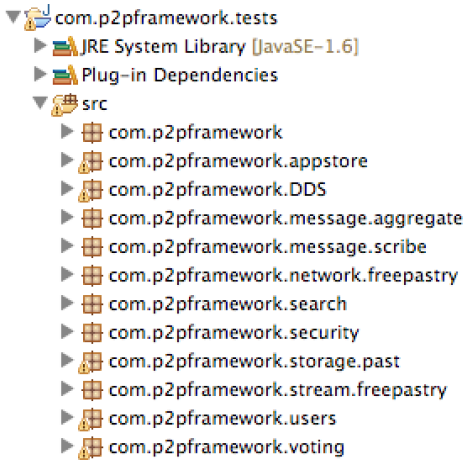


Figure 7.1: Snapshot of JUnit Test

7.2 Pool Area Testing

JUnit tests alone are not sufficient to verify the functionality since a combination of multiple functionalities could still run into problems. After the core implementation phase (until October) we started parallel testing for the combination of functionalities. It was important to have a stable environment all the time and the task of testing team was to find the core of any error and raise the flag of that error to the corresponding developer. For testing there were two different plugins we used during the PG. We started testing with a simple command based scenario for every test case as did the previous PG. Please refer to the the appendix to see how this scenario works. However, in late November we received an enhancement in the test environment by a student of earlier PG who just accomplished his master's thesis in the same area. We just provide a brief overview of the enhancement in test set-up guided by this thesis later in this section.

To start with, we assign nodes one of two roles. One node is the master and all others are the slaves. There must always be exactly one master which is used to initiate the test plan and the rest all machine performs the command given by the master. Commands are given with the help of a common test plan which must be located where it is easily accessible by all machines (currently on dropbox), so basically every node could access it and perform the tasks defined in this test plan. Earlier (before the thesis) to initiate a command or any test plan we had to explicitly type the command in the masters console view so that all nodes could receive it. However now , the master node has two different views. Figure 7.2 shows both the view for one our test run with 20 machines.

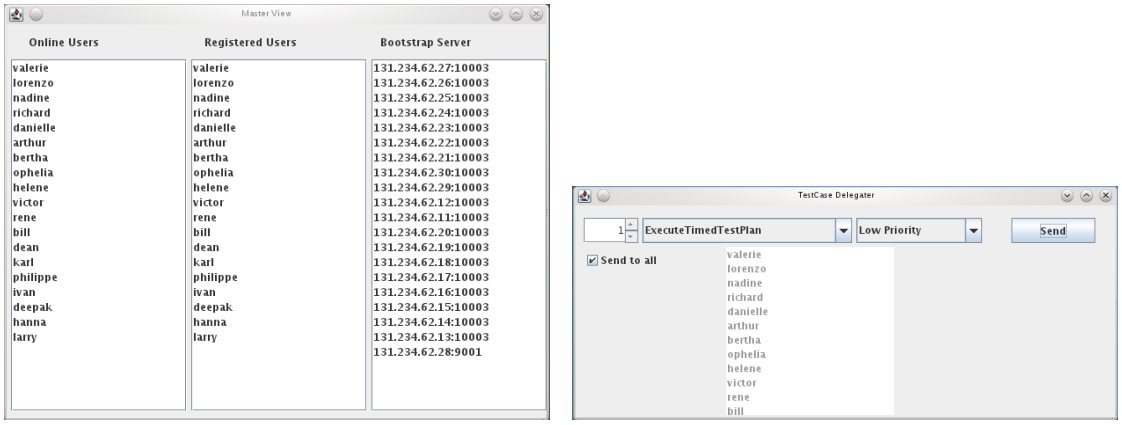
- **Master View**
  The master view shown in Figure 7.2(a) allows to see the online and registered users and also their bootstrap server.

- **Delegater View**
  The delegater view shown in Figure 7.2(b) shows a list of all registered slaves/users and an option to select from a list of multiple test plans which could then be executed on all of them.

7.2.1 Test Setup

Unlike our previous PG we had a setup of 20 machines on the pool area of second floor of Heinz-Nixdorf-Institute (HNI) at the University of Paderborn. However, machines still had hardware similar to last year. But the operating system was updated at the beginning of April

- Processor: Intel i7-2600
- RAM: 8 GB
- Operating System: openSUSE 12.3 (x86.64) with Kernel 2.6.37.6
- Network Interface: Gigabit Ethernet
- Java VM: Java 6, version 7 (build 1.7.0-ea-b104)



(a) Master View (b) Delegater View

Figure 7.2: Master Views

We used the pool area on the second floor with 20 machines and then tried to start multiple instances on each of them to have a large scale test. However, starting 3 and 2 instances on every machine was not that successful. We think that this was because of an unstable network, however, we believe that systems usually performed well in the early morning and then the performance started degrading in the evening (however this is just a belief since we experienced it multiple times). Figure 7.3 shows a sample setup of all the machines in the pool area with the test plan running on each of them (except the master).

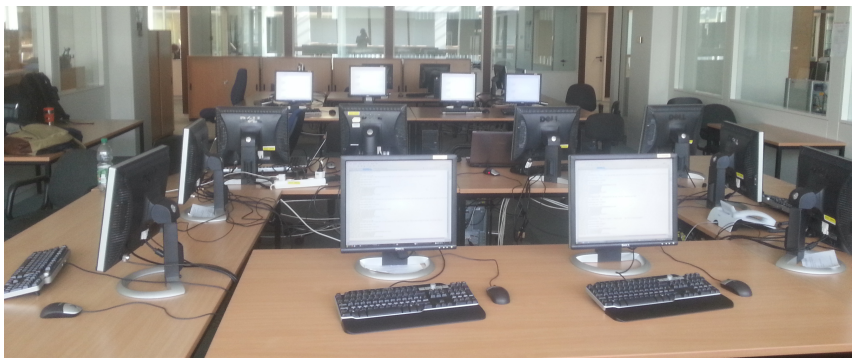


Figure 7.3: Pool Area

We used a **modified testplan**. This new testplan includes some of the newly developed features, like the forum. Figure 7.4 shows the testplan we used for our evaluations. As we can see, every test case has a certain timeout. This timeout strategy is to ensure that all the nodes start and finish at the same time and hence all the slaves synchronized. The test case for the Forum, for instance, is executed 15 times. The timeout happens after 5 minutes. This means, after 5 minutes, every machine continues to the next test case. This happens independent of whether the node finished all 15 iterations or not.



Figure 7.4: Pool Area

Lastly, it is important to understand how your system performs as a whole and this is accomplished with the help of the Monitoring plugin. Every node additionally has an option to view this plugin and to monitor its performance. We took advantage of this feature by enabling monitoring on all the slaves and then plotting the graphs to see how the nodes behave based on certain metrics. In Section 7.2.3 we discuss some of the many interesting results we plotted.

7.2.2 Problems

During testing we encountered a variety of problems that are listed in this section hoping to prevent other users to run into the same difficulties.

- **Friend Request**
  Sending friend requests has been a problem in using the testplan. It seems like the GUI is hanging after the command is executed. This sometimes leads to a unresponsive node resulting in a loss of messages. Furthermore, some nodes seem to not recover from this failure and stay unresponsive.

  Additionally, the following happens sometimes. Node A sends a friend requests to B. B accepts this requests and adds A as a friend. However, A does not add B as a friend. This can lead to problems in the execution of the testplan because in case a node does not have any friends many of the testcases cannot be executed.

- **Overloaded System**
  LifeSocial becomes very unreliable and unpredictable when too much workload and too many messages are executed.

- **Not Joining the Master**
  It happened quite frequently that a node did not succeed joining the master in the initial join of the test plugin. When this happened usually none of the slaves started afterwards are be able to join.

  We suspect the reason for this behavior lies in the unreliable test bed we used during our pool-tests at the University.

7.2.3 Monitoring and Result

Figure 7.5 compares the load between all the nodes. As mentioned earlier, we used 20 machines with one master and 19 slaves and hence in the figure we see a comparison graph between 19 nodes (all slaves). Since we are using a network with only 20 nodes, node with high messages sent could be concluded as the one with responsibility for more data items compared to the others.

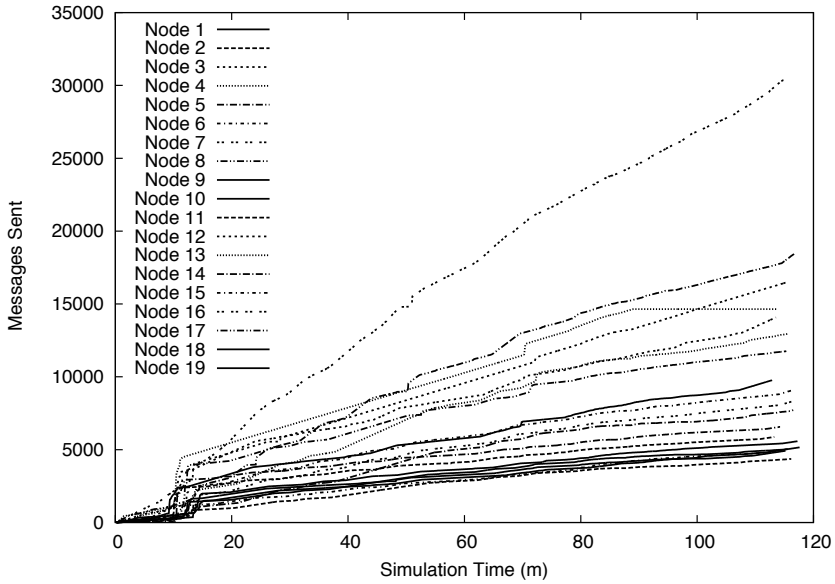
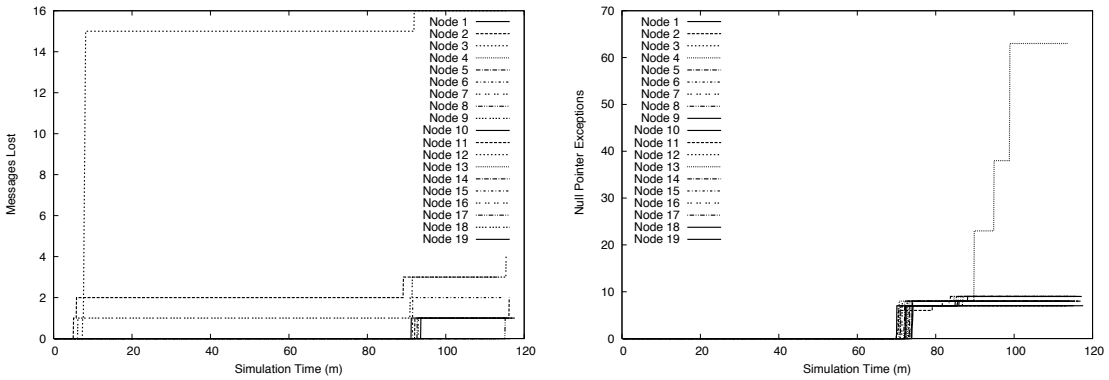


Figure 7.5: Messages Sent

Figure 7.6 shows the `MessageLostMessageExceptions` and `NullPointerException` during the testing. Again, here we see the comparison between all the slaves. Except for the one node every node has approximately two `MessageLostMessageExceptions`. Also, it is important to note that `MessageLostMessageExceptions` is just a temporary failure because of a non-responding node and network recovers most of the times by resending the messages. If we consider here the Figure 7.5, the percentage of messages lost is less than 0.1%. Further, we see that `NullPointerExceptions` only appear after 75 minutes. It was observed that during wall posts all nodes ran into this exceptions. There must be a error in the implementation of the wall that we were not able to debug anymore at the end of the project group. Currently this exceptions are caught, however, it has to be fixed.



(a) MessageLostMessageExceptions (b) NullPointerExceptions

Figure 7.6: Exceptions

Figure 7.7 compares the storage between all the slaves. There are two specific hikes in the graph. Firstly, we see an immediate hike in storage for all the nodes in the beginning. This hike is attributed to the join phase, since all the nodes then would redistribute the data based on the freepastry storage and replication strategy. Secondly, and lastly, we see a rise just before 40 minutes. During this time each slave uploads photos based on the testplan and hence there is a sharp change in storage. However, after this we see a steady graph for all the nodes.

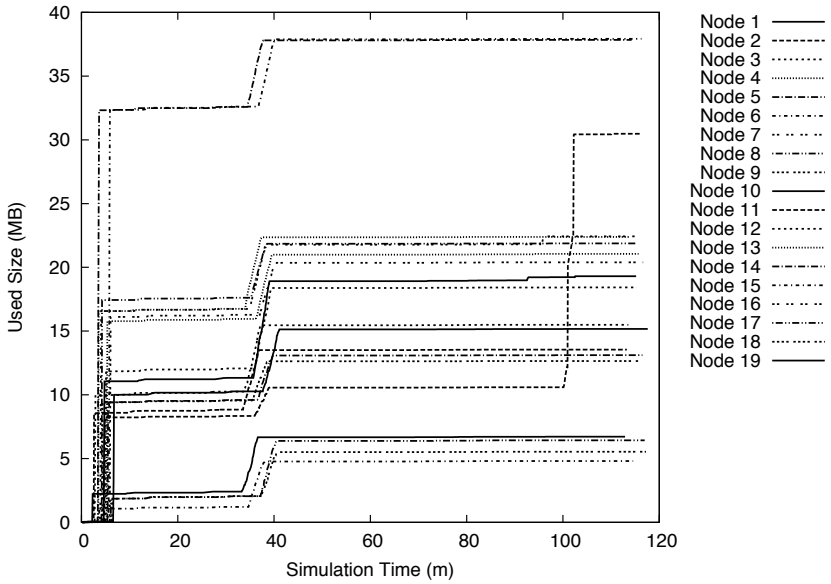


Figure 7.7: Used Size

In addition to the tests with the new system we also ran tests to identify the impact of our changes using the same test environment as described above. One of these results shows the difference in the amount of messages sent when the security approach is disabled compared to

when the security approach is enabled. The result is shown in Figure 7.8. More information on this test and the conclusion can be found in the evaluation in Section 6.1.3.

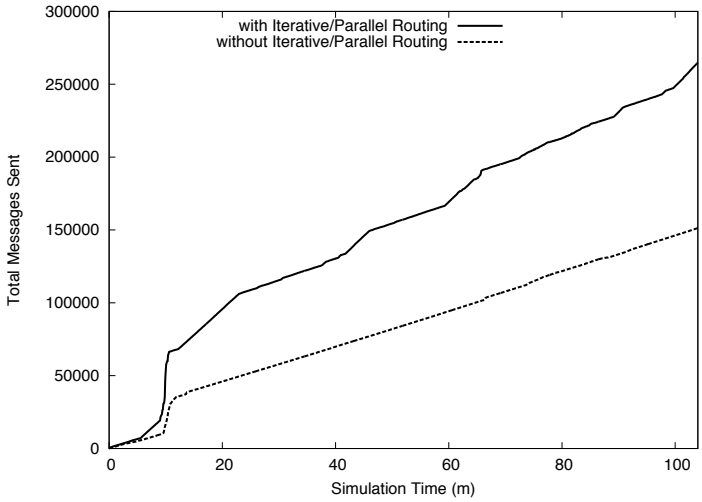


Figure 7.8: Messages Sent Comparison

7.3 Summary

In this chapter we provided an overview on testing. We dealt with two different kinds of testing strategies. Unit testing allowed a developer to the test a specific functionality. Pool area testing on the other hand, provided a large scale testing environment to check the functionality as a whole. We further observed the monitoring results and brief explanation for the corresponding behavior. In total, it was just possible to perform testing and monitoring for 20 machines which could now be taken as a model to be worked upon for a big test (may be for 60-100 machines).

Testing showed that the newly developed components work well alone and as an integrated system. The tests of user behavior using an extensive test plan with a run time of about two hours proved this. However, we ran into some issues with the FriendsPlugin and the WallPlugin that resulted in errors and exceptions. These errors are still open to be fixed.

8 Summary and Outlook

Finally, a summary will be given in this chapter. With it the accomplished features will be summerized and an outlook is given in each section in order to present reasonable improvements, that could be accomplished in the future. These could be open tasks, which haven't been accomplished or ideas that appeared during the work on the PG.

8.1 GUI/Plugins

After working out useful areas for LifeSocial, many ideas for plugins came up. These ideas have been presented in section 2.3.

First of all, the enhanced groups plugin has been developed to introduce functionality for public just as private groups. Hence, user of LifeSocial can create their own groups and invite other users to it. Then, the users of a group can publish content in form of threads and posts in forums, that have been created by the admin of the group. Furthermore a multi-view plugin was implemented to enable multi-user interaction. The topic chat plugin, which was also implemented during the project group, is using the functionality of the multi-view plugin. By that a user can choose user out of his friends list and invite then to a multi chat. Moreover the file hosting plugin offers the possibility to store files in the p2p-network efficiently. Other user can request the download links of the file by the admin in order to download the desired files.

The voting plugin is using the functionality, that is provided by the framework. Thus, a polling function is provided by the voting plugin. Users can create public or private polls and add several options just as the maximum voting number for each participant.

The approach of using the Eclipse RCP is still questionable, even after enabling the perspectives. Compared to well-established social networks, that are based on HTML, the design opportunities with the Eclipse RCP are limited. Here, two possible solutions are possible: On the one hand one can go for HTML, which is easy to use and also well known by many developers. Though it is shown in the browser, a local running p2p client would still be needed. A detailed analysis of the opportunities of using HTML for LifeSocial is given in [16]. The author also proposes a way to handle the security issues that come up when using HTML with third party plugins.

On the other hand, staying with Eclipse would save a lot of development time. For a research project the research concerning p2p networks should be more important than the UI. Though when the project aims for a larger group of users, the usability is very important for the acceptance by the users.

When staying with the Eclipse RCP, an improvement of the perspectives should be considered. At the moment one can create, save and load new perspectives. But the problem is that by closing the application the last state is saved and loaded at the next start automatically. So if the auto login is not activated, the views of the previous session are opened without being logged

in. That leads to different errors of the views, which except you being logged in to see the views. Therefore the views-architecture has to be changed in order to prevent these errors and for example show a short message in each view, that says, that you have to log in first.

Some of the plugins, we developed, can be easily extended by further plugins. Especially the groups plugin provides an easy way for other plugins to provide new types of group content without any modification of the groups plugins itself. The functionality of the file hosting plugin could be added in order to provide group specific files. Also many more plugins for supporting collaborative work could be added. Useful examples would be a wiki, a collaborative text editor, a whiteboard or video conferences.

The multi-view, which is part of the livechat plugin, can also be extended to support more features. In the current architecture this is not possible without changing the code of the livechat plugin. In the future the livechat plugin should provide a similar interface for multi-view functions, as the groups plugin does for content types.

All in all, there are more plugins, which are presented in section 2.3 that could be implemented to complete the full support of the supported areas, that have been worked out during this project group.

8.2 Framework

In the framework layer we developed and implemented different approaches to enhance the basic functionalities on which plug-ins can be based. One problem we mainly focussed on are access control mechanisms. We developed a solution that provides read access control as well as write access control based on cryptographic methods. To disseminate the required keys for encryption and signing of content, we enhanced the already existing approach of `groups` in such a way that now access rights can be defined as complex hierarchical structures. The distributed data structures have been adapted to the new access control. In the future it might be reasonable to modify existing plug-ins so that they make use of this new approach. For instance, the photo album does not store any content encrypted so far and even though the profile declares several attributes as private, currently none of them is stored encrypted. A further nice feature for access rights on the lifesocial layer that can be implemented in the future might be a central concept to define access rights. One possibility one can think of is the definition of circles that can be used in any other plug-in that allows to share content in order to secure the access. Currently the access control implementation for the distributed datastructures supports writecontrol for the owner of a item, it might be useful to extend these writecontrol to the owner of the item and the group admins, in case a admin wants to delete some items from other users.

Another topic we worked on in the framework layer is the voting functionality. So far it provides basic methods to define and run a voting. In the future this functionality could be extended in such a way that, for instance, anonymous, but still authentic, votings are possible. Futhermore, on the lifesocial layer the voting could be used to implement a kind of like button which can be related to any kind of content in the network.

Another issue that should be addressed in the future regards the remote operations. Several mechanisms like the distributed data structures use remote operations in order to, among other things, decrease the traffic. However, if the replication is still done based on transmitting the entire item through the network, we just save a small part of traffic for write operations. Thus,

it might be a useful goal in the future to implement replication in these cases through remote operations as well. Additionally, the verifications that are done for the write access control should also be applied within the replication mechanism so that the corresponding signatures are not only verified at the responsible node but also the replica holders perform a check (e.g. to avoid unauthorized changes by the responsible node itself).

8.3 Overlay

In the Chapter 6 we discussed two open solutions, namely, security and capacity awareness.

In security, we saw that FreePastry is prone to various security threats and it is necessary to overcome these problems to have a better working environment. Hence by explicitly defining the role of source node in routing mechanism, we determined the responsibility of routing directing for the initiator of lookup. Every node in the routing path sends a set of best next hops to the source node, so the opportunity for intermediate nodes to stop the lookup or make a malicious loop, would definitely be less. At the same time source node perform the lookup in parallel manner, here we see why a set of best answers returned by intermediate nodes are useful. Hence this provide the routing mechanism with robustness to be mislead since the source node has this opportunity to try more that one path to the destination. On the other hand, the source node should keep the list of contacted nodes to avoid trying redundant answers and a list of k best answers which guarantee that the routing is getting close to the destination. We saw basically how a combination of parallel and iterative routing could reduce the security threat considerably.

The other major topic we discussed under overlay was capacity awareness. We saw that FreePastry balances the workload evenly in the network, however nodes capacities in the network is not even which means nodes with little capacity may be overloaded. In the Section **??**, we introduce a solution that a node which is responsible to an object, will redirect the request for the object to other nodes, which have requested this object before and cached it, based on the capacity information. Also we tested how well it works and notes test results in this section. Besides, FreePastry does not consider mobile nodes in the network, which have relatively little storage space and an unreliable network connection, entailing churns and degradation of performance. Further, we discussed how we tried to treat mobile nodes correctly in the network. Essentially, we let the strong nodes around mobile work as relay nodes, to help the storage and message transceiver of mobile nodes. Mobile nodes plays no role in the routing and storage in the network. And also we documented the enhancement and possibilities in the future.

Appendix A

StartLifeSocial Guide

A.1 Eclipse configuration

A.2 LifeSocial configuration

How to configure bootstrap node ip and how to connect to each other.... How to self login

Appendix B

Plugin development guide

In this appendix we describe a guideline for developing LifeSocial plugins. This includes the package structure, creation of model classes, testing and monitoring. Additionally, some useful hints regarding existing APIs and using asynchronous programming will be provided.

B.1 Package structure

A functionality, which is in this document often simply referred to as *plugin*, typically consists of two different *OSGI bundles*: One bundle for the domain logic and one for the user interface. For the developer it would be easier to put all code in one OSGI bundle. Though, we experienced that this led to bad code, since one is tempted to mix up UI and domain logic in the same classes. Decoupling is an important principle of software engineering, which means that there should always be as little dependencies as reasonable possible. Since the UI code depends on the domain logic but not vice versa, two different OSGI bundles should be used, which force the developers to respect this principle. The unidirectional dependency has also to be set up in OSGI, so that the OSGI bundles are loaded in the right order. Technically that means to add the domain logic bundle to list of dependencies of the UI bundle in its `MANIFEST.MF`.

B.1.1 Domain logic

The domain logic bundle contains all classes and methods which store files, communicate with the network or implement any non-UI-related functionality. It has typically the package prefix `org.lifesocial.plugin`, so its name might be `org.lifesocial.plugin.example`. As subpackages we suggest at least the following ones:

org.lifesocial.plugin.example.api

This package contains all interfaces of the bundle, which can be used by other bundle, e.g. the UI bundle.

org.lifesocial.plugin.example.model

The `model` package contains all data model classes, i.e. the structure of the information that is stored in the P2P network. Creating models is described further in Section B.2. For loading

these models from the network it is very **important that it is exported in the MANIFEST.MF**, because otherwise the instances cannot be recovered during deserialization.

org.lifesocial.plugin.example.utils

The `utils` package contains helper classes, e.g. *PluginPropertiesContainer* containing some constants.

org.lifesocial.plugin.example.commands

The `commands` package contains classes which define methods that can be called using the testing mechanism of LifeSocial. This is described in Section B.3 in more detail.

B.1.2 User interface

The UI bundle is responsible for each and every interaction with the user. It invokes methods of the corresponding domain logic bundle. For getting noticed about events like e.g. chat messages, the *Observer Pattern*[17] should be used.

Observe

The Observer pattern was introduced by the *Gang of Four* in their popular book *Design Patterns: Elements of Reusable Object-Oriented Software* [17]. An arbitrary number of so-called *observers* is interested in events that happen at a specific *subject*. It is used to push event notifications from the subject to the observers. The observers implement a specific interface, so that the subject does not need to be aware of all concrete implementations of the observers.

In the context of OSGI this concept can be used using the *service registry*. The OSGI bundles can register implementations of interfaces (”services”), which can then be queried by other bundles. In terms of OSGI, the registration is referred to as *providing* a service, while the usage of the services is called *binding*. In current Equinox versions the providing and binding is achieved using *declarative services*, i.e. creating XML files in the *OSGI-INF* folder of the specific bundle.

For the use case of notifying the UI bundle of events occured in the domain logic bundle, the UI bundle would implement a specific interface, e.g. *MyPluginObserver*, which is provided by the UI bundle and bound by the domain logic bundle. The latter thus gets to know about all observers and can notify them about events.

Translations

We suggest to put all translations in the `org.lifesocial.plugin.example.ui.translation` package. Eclipse supports the developers in a very comfortable way to extract existing Strings and perform the process of internationalization. This is shown in Figure B.1, Figure B.2 and Figure B.3.

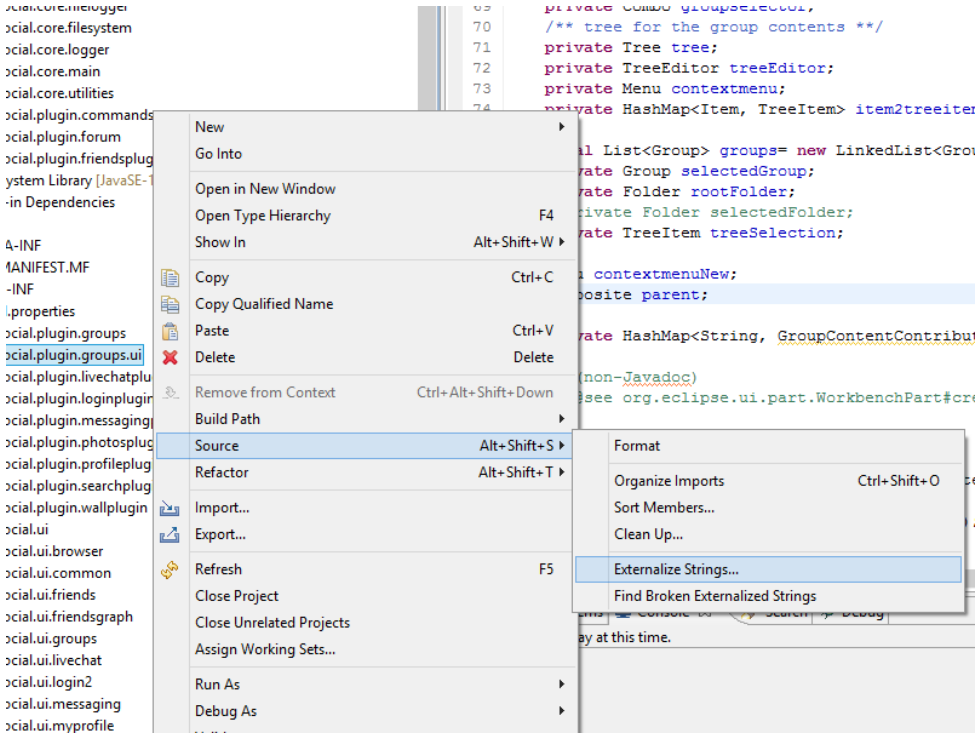


Figure B.1: Select *Externalize Strings* in the context menu

B.2 Models

The models package should contain all classes that contain data which is stored in the network. Most models should inherit from the *AbstractStorageItem*, which contains many features and helpful methods that are required in many models. It has been described in Section 5.5. The following example should give an idea how this could look like.

The forum plugin contains four model classes: `Forum`, `ForumThread`, `ForumThreadListItem` and `ForumPost`.

Each forum has an arbitrary number of threads. One should be able to efficiently access all threads of a specific forum, as well as a specific thread by it's StorageKey. A direct containment of the threads inside a distributed list in the forum would not allow the threads to be accessed directly, because the list items are stored in buckets and not under specific storage keys. Therefor, the `ForumThreadListItem` was introduced, which contains all information of the thread, which is required in thread listings (namely the subject and the date of creation). This creates some amount of redundancy because these information is also contained in the thread objects, though this minimizes the number of required lookups for a thread list very effectively. The `ForumThreadListItem` also contains a reference to the thread. When someone opens a thread from the thread list, `getOneReference("thread")` can be used to load the specific thread with its posts. In contrast to this approach, the posts can be stored directly inside distributed storage items, because they don't need to be accessible by their IDs.

Commented code examples of the `Forum` and the `ForumThreadListItem` can be found in Listing B.1 and Listing B.2.

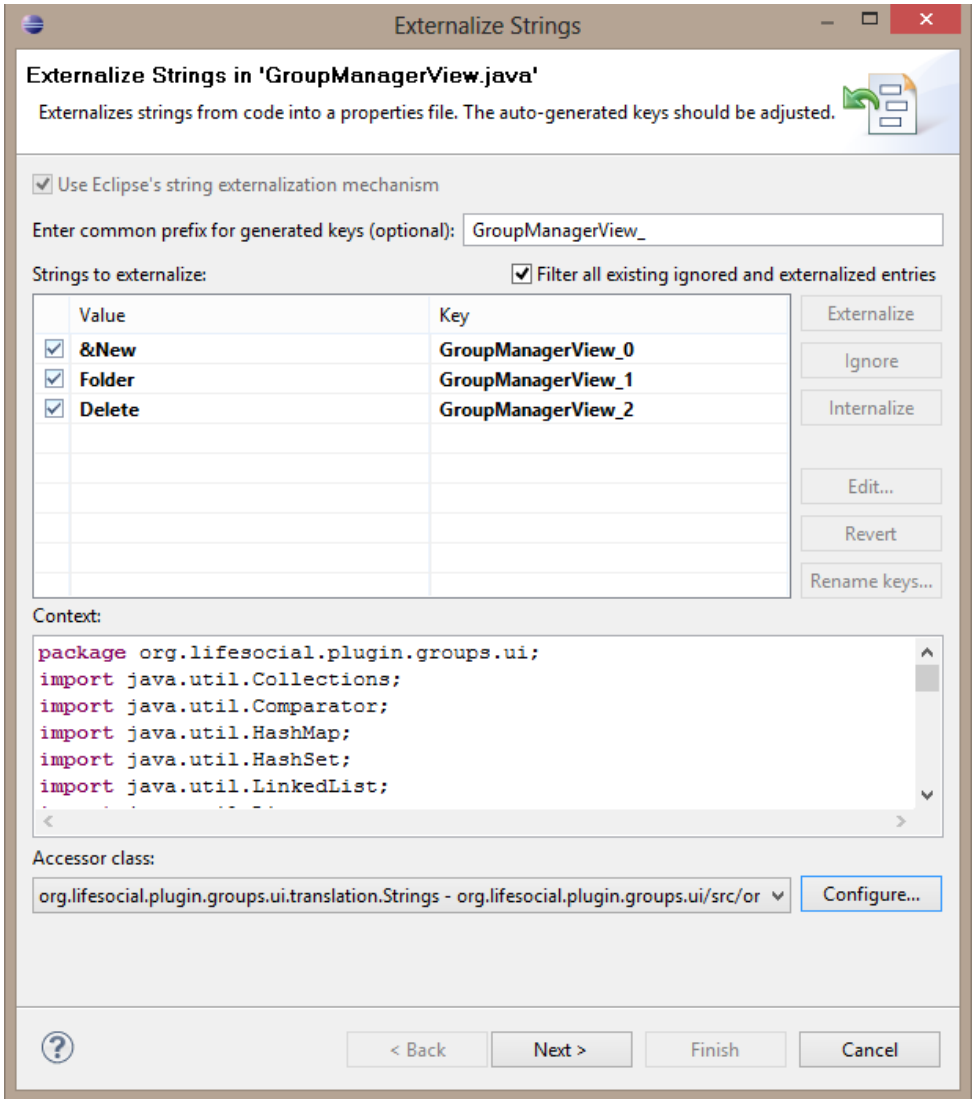


Figure B.2: Choose which Strings you would like to internationalize. Click on Configure to choose the translation package.

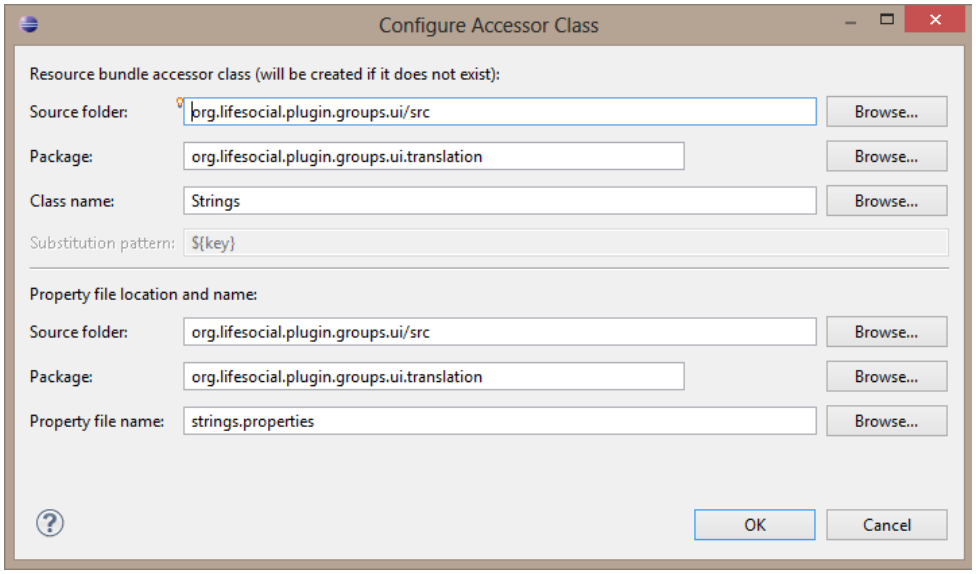


Figure B.3: Example configuration for the translation of an UI bundle

Listing B.1: org.lifesocial.plugin.forum.model.Forum

```
1
2  public class Forum extends AbstractStorageItem{
3
4    // just a random number that has been generated by eclipse to identify
5    // the class during deserialization
6    private static final long serialVersionUID = -4094665104690779226L;
7
8    //any attributes that shall be stored can be declared here
9    String name;
10
11   //the constructor additionally takes the 'name' attribute
12   public Forum(Framework framework, String name) {
13     super(framework);
14     setName(name);
15   }
16
17   public Forum(Framework framework) {
18     super(framework);
19   }
20
21   public void setName(String name) {
22     this.name=name;
23   }
24
25   // here a reference 'threads' is created,
26   // which points to a DistributedLinkedList of ForumThreadListItems
27   @Override
```

```
28    protected DistributedSet<? extends AbstractStorageItem>
29              loadManyDDSReference(String key) {
30      if(key.equals("threads")) {
31        try {
32          return new DistributedLinkedList<ForumThreadListItem>(
33              ForumPlugin.getInstance().getFramework(),
34              ForumPlugin.getForumThreadsStorageKey(getID()));
35        } catch (ItemNotFoundException e) {
36          return new DistributedLinkedList<ForumThreadListItem>(
37              ForumPlugin.getInstance().getFramework(),
38              ForumPlugin.getForumThreadsStorageKey(getID()),100);
39        }
40      }
41      return null;
42    }
43
44    // the previous created 'threads' reference is here declared as containment
45    // i.e. all threads will be deleted, when the forum is deleted
46    @Override
47    protected Set<String> getManyDDSContainments() {
48      Set<String> s = super.getManyDDSContainments();
49      s.add("threads");
50      return s;
51    }
52  }
```

Listing B.2: org.lifesocial.plugin.forum.model.ForumThreadListItem

```
1  public class ForumThreadListItem extends AbstractStorageItem {
2
3    private String subject;
4    private Object created;
5
6    public ForumThreadListItem(Framework framework) {
7      super(framework);
8    }
9
10   //when setting the thread reference, it automatically copies the information
11   //which is needed for thread lists: subject and date of creation
12   public void setThread(ForumThread t) {
13     this.setOneReferenceID("thread", t.getID());
14     this.subject = t.getSubject();
15     this.created = t.getCreated();
16   }
17
18   public ForumThread getThread() throws ItemNotFoundException {
19     return (ForumThread) this.getOneReference("thread");
20   }
```

```
21
22    public UUID getThreadID() {
23      return getOneReferenceID("thread");
24    }
25
26    // this declares that the thread is a containment.
27    // when the ThreadListItem is deleted, the thread is also deleted
28    @Override
29    protected Set<String> getOneContainments() {
30      Set<String> s = super.getOneContainments();
31      s.add("thread");
32      return s;
33    }
34
35    public String getSubject() {
36      return subject;
37    }
38
39    public Object getCreated() {
40      return created;
41    }
42
43  }
```

B.3 Testing / Commands

Adapt for newer testing plugin

This section is about the implementation of performance / scalability test functions as described in Section 7. Functionality tests can be implemented easier using *JUnit*.

As described in Section **??** the sequence of executed functions is provided in the `TestPlan.txt`. This *test plan* calls so-called *commands*, which are provided by the OSGI bundles. The commands can be additionally called from the OSGI console interface as seen in Figure B.4. To provide such commands, a OSGI bundle needs to provide the `de.tud.kom.lifesocial.core.commands.ICommands` service. For that we suggest the following architecture. Note that the commands should *always* be implemented in the domain-logic OSGI bundle, because it is not related to the UI.

An abstract class `MyPuginsAbstractCommandHandler` which inherits from `de.tud.kom.lifesocial.core.commands.AbstractCommandHandler` should be created as the base class for all implemented commands. This inheritance level is required to distinguish the commands from your plugin from the commands from other plugins. Each command, which is typically implemented in the subpackage `org.lifesocial.plugin.myplugin.commands` inherits from your `MyPluginsAbstractCommandHandler`. This fact is also registered in OSGI: For each command there must be an XML file in the `OSGI-INF` folder which claims that the command class provides the `MyPluginsAbstract-`

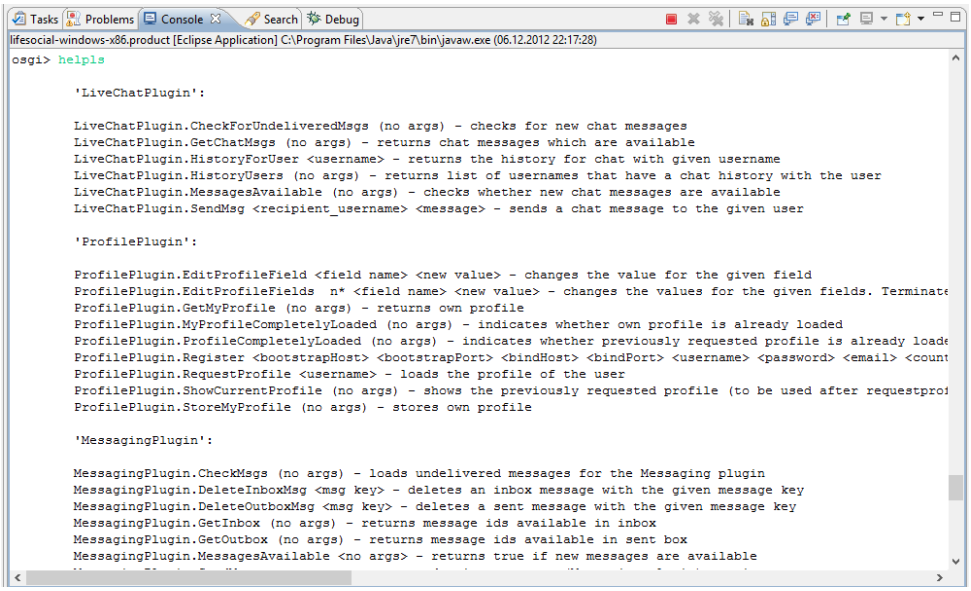


Figure B.4: Listing the available commands in the OSGI console

`CommandHandler` service. Another class `MyPluginsCommands` must be created which provides the OSGI service `ICommands` and references the previously implemented commands, i.e. references the `MyPluginsAbstractCommandHandler` service. This class can inherit all the functionality from `de.tud.kom.lifesocial.core.commands` and just modify the *pluginID* as seen in Listing B.3.

The functionality which the commands execute should not be implemented in the command classes, since most functions are also used from the UI. A better place to put them is, for example, the plugin's main class.

Listing B.3: org.lifesocial.plugin.forum.commands.ForumCommands

```
1 public class ForumCommands extends AbstractCommands {
2
3         public ForumCommands() {
4                 this.pluginId = PluginPropertiesContainer.PluginID;
5         }
6 }
```

B.4 Monitoring

The monitoring mechanism is used to observe the behavior of the whole peer-to-peer network to measure performance. The technical details of the technique has been described in [18, Sect. 4.7]. For implementing further measuring units, the monitoring plugin now supports OSGI services. Thus, OSGI bundles can simply provide the `com.p2pframework.message.aggregate-.Sensor` service to offer their own measurements. In detail this means to implement the `Hashtable<String, Attribute<Serializable>> getAttributes()` function. More information about the settings of the `Attribute` class can also be found in [18].

B.5 Tips

This section contains some generally useful tips for programming LifeSocial plugins.

B.5.1 Asynchronous programming

Concept of callbacks

The plugins that were created before our Project Group were using *polling* to transfer information from the domain-logic bundles to the UI bundles. That means, that the UI bundles were periodically asking the domain-logic bundles whether there is new information available. This technique leads to CPU overhead and a delay of the UI interface, which depends on the length of the period.

To overcome this problem, we suggest to use asynchronous callbacks. The messaging UI bundle should for example be noticed of arriving messages. Though, the domain-logic bundle should not be aware of the existence of the UI bundle. Thus, the UI bundle should register a callback at the domain-logic bundle for getting noticed if new messages arrive. This callback technically a class which implements an interface which requires a specific method. This method is executed whenever the actual event occurs. Abstract classes for different types of events are available in the `com.p2pframework.utils` package.

Network processing thread

The network operations of the framework will block the thread which is executing them. Thus, you should always create a new thread when accessing the network. The framework provides the method `framework.processingThread(Runnable obj)` which can be used for that. For executing code on the completion of the network operations, you can extend the callback classes provided in `com.p2pframework.storage`: e.g. `RetrieveContinuation` and `StoreContinuation`.

SWT thread

One last thing to remember when creating a UI bundle is, that SWT is always using a single thread. To call SWT methods from other threads within view parts, you need to call `parent.getDisplay().syncExec(Runnable obj)`.

B.5.2 Providing extension points

The group plugin can be easily extended by providing the `IGroupContentContributor` service. This is a good example for providing extension interfaces for other plugins. When developing a plugin, you should think of possible extension points, which could be useful for third-party developers. More information about the extensions of the groups can be found in Section 3.1.

Appendix C

Test Guide

This chapter contains a tutorial for how testing can be done and provides some useful tips and hints to keep in mind during testing.

The goal of testing here is to check whether the LifeSocial, with integrated features implemented by each group, is working. And also a long-time running test is set up here to find the robustness of LifeSocial.

C.1 Using the Test Plugin

To use the test plugin follow the following instructions.

Open eclipse, in project explorer find `org.lifesocial.ui/ProductConfigurations`, find the suitable product configuration for your machine and double click it. Then open the dependency tab, add `com.sign.plugin.testplugin` into the dependencies and click the `Add Required Plug-ins` button. Then, the test plugin has been added into the application's dependency, however, to run this test plugin, it needs several resources files that configure the test plugin and provide resources, for instance photos, that are used during the execution of a testplan.

A folder that contains all required resources can be found in the source folder of LifeSocial at the following path: `./Source/com.sign.plugin.testplugin/resources/`. To run the test plugin copy this resources folder to a path that LifeSocial uses to access this data. This location depends on your operating system. We located the right folder for different systems as follows.

Windows	./eclipse/
MacOSX	/Applications/Eclipse/Eclipse.app/Contents/MacOS/
OpenSUSE	/home/username/Documents/

To start the test plugin, make sure it is enabled in the `config.txt` which is located in the resources folder. This config should also contain the IP of the master node.

After this initial setup, start LifeSocial on the master node. Now two windows shown in Figure 7.2 will appear. One is used to show all nodes connected to the master, the other allows to select and execute test-cases on the slaves.

C.2 Tips and Hints

This section provides some tips and hints that will help users working with the test plugin. To get an overview of possible problems during testing please refer to Section 7.2.2.

C.2.1 Storage Folder Removal

The storage folder contains the data which is stored in the overlay. This folder should be removed before a new test is started. Otherwise a node might log in with an already created user account. This can result in a number of problems. For instance, the user might already have friends that are not existing anymore.

C.2.2 Multiple Instances on One Machine

It is possible to run multiple instances on one machine. To do this, just start LifeSocial a second time. The second node should have the same IP as the first. However, the port number will be increased by two automatically. Please note that the port number is not increased by one. This is the case to prevent the user to start a weak node, as described in Section 6.3, by accident.

C.2.3 Send Friend Request Instability

Sending friend requests has been proven to be unreliable during testing. Try to avoid using this testcase.

C.2.4 Overloaded System

LifeSocial becomes very unreliable and unpredictable when too much workload and too many messages are executed.

Bibliography

[1] Alexa, "Top 500 global sites," March 2013. Can be visited at: http://www.alexa.com/topsites/global.

[2] jQuery Foundation, "jQuery: The Write Less, Do More, JavaScript Library." http://www.jquery.com/, 2012.

[3] P. Gillespie, "Extendible-BBCode-Parser." https://github.com/patorjk/Extendible-BBCode-Parser, 2012.

[4] T. Parr, "Stringtemplate template engine." http://www.stringtemplate.org/, 2012.

[5] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, "Characterizing User Behavior in Online Social Networks," in *Proc. of ACM SIGCOMM IMC '09*, 2009.

[6] P. Leach, M. Mealling, and R. Salz, "RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace," 2005.

[7] "Rice University, FreePastry." URL: www.freepastry.org/, Last accessed in April 2013.

[8] H. Lin, R. Ma, L. Guo, P. Zhang, and X. Chen, "Conducting routing table poisoning attack in DHT networks," in *Proceedings of the International Conference on Communications, Circuits and Systems*, 2010.

[9] T. Condie, V. Kacholia, S. Sank, J. Hellerstein, and P. Maniatis, "Induced Churn as Shelter from Routing-Table Poisoning," in *Proceedings of the International Conference on Network and Distributed System Security Symposium*, 2006.

[10] H. Yu, M. Kaminsky, P. Gibbons, and A. Flaxman, "Sybilguard: Defending against sybil attacks via social networks," in *Proceedings of the IEEE/ACM Transactions on Networking*, 2008.

[11] H. Yu, M. Kaminsky, P. Gibbons, and F. Xiao, "Sybillimit: A near-optimal social network defense against sybil attacks," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[12] F. Li, P. Mittal, M. Caesar, and N. Borisov, "Sybilcontrol: Practical sybil defense with computational puzzles," in *Proceedings of the ACM workshop on Scalable trusted computing*, 2012.

[13] Q. H. Vu, B. C. Ooi, M. Rinard, and K.-L. Tan, "Histogram-based global load balancing in structured peer-to-peer systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 4, pp. 595–608, 2009.

[14] unknown, “Adding capacity awareness and load balancing tohomogeneous distributed hash tables.” 2012.

[15] K. Graffi, S. Kaune, K. Pussep, A. Kovacevic, and R. Steinmetz, “Load balancing for multimedia streaming in heterogeneous peer-to-peer systems,” in *NOSSDAV*, pp. 99–104, 2008.

[16] M. Sander, “HTML5 opportunities with servlets.” 2012.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Amsterdam: Addison-Wesley Longman, 1 ed., 1995.

[18] K. Graffi and Project Group 2011, “A Distributed Framework for Social Networks,” tech. rep., University of Paderborn, 2011.