

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

A Peer-to-Peer Framework for Social Networks
Project Group 2013

Final Report

by
Christy Jovita Bibiana Arockiaraj, Kailaash Balachandran, Navendu Barua, Bing Gao, Eshant Tulsidas Gour, Christopher Holomek, Thomas Jacobs, Siddharth Kumar Bhagvanjibhai Kaneriya, Maitri Modasia, Weifeng Peng, Chander Prakash, Manasa Veena Rapolu, Florian Schulte and Rohan Sukumaran

Supervisor:
Jun.-Prof. Dr.-Ing. Kalman Graffi
Paderborn, April 11, 2014

1. Motivation 5
2. Status of Framework before the Project Group 6
2.1. Architecture 6
2.1.1. Pastry Overlay 7
2.1.2. Framework and Testing 8
2.1.3. Plugins/ GUI 11
2.2. Open Problems and Weaknesses 13
3. Proposal 14
3.1. Proposed Improvements 14
3.1.1. Major Problems and Limitations 14
3.1.2. Further Improvements to Existing Features 16
3.2. Proposed Extensions 17
3.2.1 Network Address Translation (NAT) 17
3.2.2 Video and Audio Communication 18
3.2.3 Collaborative Work 20
3.2.4. Notifications and Live Feeds 22
3.2.5. Secure Monitoring 23
3.2.6. Tags, Likes and Ratings 24
3.2.7. Secure Replication 26
3.3. Not implemented Extensions 27
3.3.1 E-Learning 27
3.3.2. Video and Audio Streaming 28
3.3.3. BitTorrent 29
3.3.4. App Store 30
3.3.5. Map Reduce 31
3.3.6. Mini Games 33
3.4. Proposed Milestones and Organization 35
4. Outcome of the Project Group 36
4.1. New/ Modified Architecture 36
4.2. Overview of Improvements and Extensions Implemented 37
4.2.1. Plugins 37
4.2.2. Framework 41

4.2.3. Overlay ……. 42
5. Achievements in the Plugins/ GUI Layer ……. 44
5.1. Technologies used for new GUI ……. 44
5.2. Parts of Plugin/ GUI ……. 45
5.2.1. Plugin Template ……. 45
5.2.2. Plugin Logic ……. 45
5.2.3. Web Provider ……. 46
5.3. REST Handler ……. 48
5.4. HTML Architecture ……. 49
5.5. New Plugins ……. 57
5.5.1. Notification Plugin ……. 57
5.5.2. Messaging Plugin ……. 59
5.5.3. Wall Plugin ……. 62
5.5.4. Chat Plugin ……. 65
5.5.5. Audio/ Video Chat Plugin ……. 67
5.5.6. Photos Plugin ……. 70
5.5.7. Groups Plugin ……. 74
5.5.8. Files Plugin ……. 82
5.5.9. Friends Plugin ……. 85
5.5.10. Calendar Plugin ……. 89
5.5.11. Search Plugin ……. 95
5.5.12. Profile Plugin ……. 98
5.5.13. Error Notification and Console Plugin ……. 100
5.5.14. Voting Plugin ……. 107
5.5.15. Login Plugin and Auto Login ……. 111
6. Achievements in the Pastry Overlay ……. 114
6.1. Rings and Ring Identification ……. 115
6.2. NAT - Network Address Translation ……. 118
6.3. Secure Replication and Modification of Items ……. 125
6.4. Signing and Java Serialization ……. 128
7. Open Topics and Future Work ……. 132
7.1. Proposed Improvements ……. 132
7.2. Proposed New Features ……. 133

7.3. Further Topics to be addressed ……..……..…….. 138
8. Testing ……..……..……..……..…….. 139
8.1. Unit Test ……..……..……..…… 139
8.2. Test Plugin ……..……..……. 139
8.3. Alpha Test ……..……..……..…….. 143
8.4. Monitoring ……..……..……..…….. 143
9. Conclusion ……..……..……..……. 145

Attachments:

Screenshots of LifeSocial ……..……..…… 146

1. Motivation

Social networking has grown from simple chat rooms to full scale web services that almost anyone can use; with features such as personal profiles and ways to share things like photographs and video clips. The rapid development of social applications has grown tremendously with almost twice as many internet users having an online profile than two years ago with more virtual friends from all over the world[1]. Individuals add new friends or extend their business connections by connecting and interacting with others, thus the geography of a person is no longer an obstacle. There is no easier or faster way to get connected to social world than via the social network.

Online centralized social networks certainly have limitations over security concerns. Particularly privacy and protection from massive data-mining and big-brotherly treatment of the users by the social networking service providers are problems. Since all data is accessible by the provider on a central place, users have to trust this third party not to steal his data. Centralized servers require a lot of maintenance and are expensive, as well. There is always a high risk of single points of failures which results into a complete breakdown of the whole network.

Peer-to-peer (P2P) networking is on the boom because of properties like fault-tolerance, scalability and the self organizing nature. It allows participants in a network to connect and share their resources without any centralized component like a server. As participating nodes in P2P (Peer-to-Peer) networks act as both, client and server, they are also known as servants. Peers share the load of processing power for computations, storage capacity and cost among each other, which improves the utilization of unused resources. The P2P network technique can be useful in oppressed countries like Syria, Egypt etc.. Since there is no central server, it is hard to shut down the entire network. When extending this idea of a P2P network with end-to-end encryption and privacy options, this enables a secure, decentralized communication. Although your data possibly is stored at some other nodes in the network, it is encrypted and protected against unauthorized access. These ideas form the building blocks of our project group.

Of course, we are not the first ones building a peer-to-peer network. Examples of P2P networks are Kademlia, BitTorrent and Gnutella which are mostly used for file sharing. We extended the basic principles to implement a fully decentralized social network: LifeSocial.

[1] http://www.guardian.co.uk/business/2009/oct/16/social-netwrkign-facebook-internet (last checked: 12.07.13)

2. Status of the Framework before the Project Group

To overcome the weaknesses of a centralized, server-based social network, LifeSocial was developed since 2007 at the University of Paderborn and the University of Darmstadt. It is entirely based on P2P technology without the need for central instances. Responsibilities are spread throughout the entire network on the participating peers. A major design decision was to support secure communication. LifeSocial provides authentication, confidentiality, encryption and access control by using symmetric and asymmetric cryptography. This way, censorship and collection of data by third parties are prohibited. The following subsection introduces the general architecture of LifeSocial as it was before starting with our project group in April 2013.

2.1 Architecture

LifeSocial is based on a layered architecture. Before our project group, it was organized as shown in Figure 2.1.1:

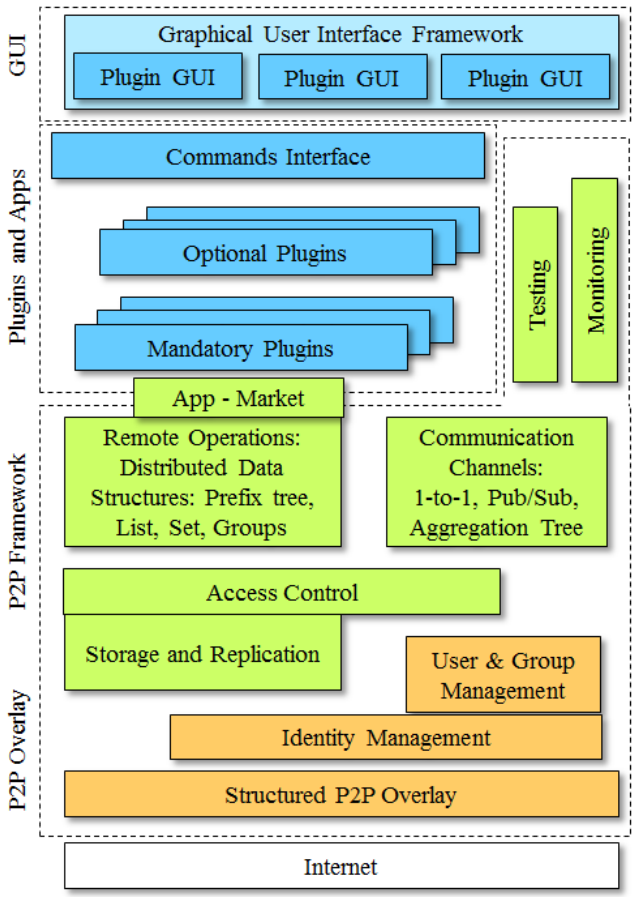


Figure 2.1.1: Layered architecture of LifeSocial in April 2013.
(from Project Group 2012 - Final Presentation)

2.1.1 Pastry Overlay

The basis of LifeSocial is the overlay providing the basic peer-to-peer functionalities. It is entirely based on FreePastry, an open-source implementation of the structured Pastry overlay. Objects and peers are identified by a global unique ID and each peer stores the objects with IDs in its responsibility range. Since Pastry is a structured overlay, efficient ID-based routing in time O(log(N)), with N being the number of peers, is supported.

During the previous project groups, FreePastry has been extended to overcome security issues and to take care of heterogeneous peer capacities:

Secure nodeIDs:

Asymmetric cryptography is used for encryption and decryption of data, as well as for digital signing. Keys are generated based on elliptic curve cryptography, taking the username and password of a user as the input. Each key has a length of 160-bit. The public keys are used as nodeIDs. Since the user may change his login information, the nodeID does not suffice for a unique and immutable identification of a user. Therefore, an immutable userID is generated when creating a new user profile. To map the nodeID to the userID, a mapping-item is created, the key of which is equal to the nodeID. Since a node knows his nodeID, the Pastry routing mechanism can be used to retrieve the corresponding mapping-item. Another point is that plugins often want to communicate with a user having a certain userID. Because they in general are unaware of the nodeID which currently belongs to that userID, another mapping-item is stored in the network to map a userID to the current nodeID.

Parallel, iterative routing:

Each routing entry is a bucket of multiple peer-addresses. A requesting node delivers messages to K different peers, which in turn respond with their best routing entries, the message should be forwarded to. The requesting node waits for the answers and then, iteratively in the same way, forwards the message to the K most promising peers, which in turn respond with their best routing entries. This strategy is iterated over and over again, until the message is delivered to the target peer. This way, the requesting node has more control on the routing process and the impact of malicious nodes during routing is reduced.

Heterogeneous peers and peer capacities:

A responsible node for a data item keeps track of peers which requested this item. These peers are treated as caches. Whenever the responsible node is asked to provide the data item, it can redirect that request to a caching node. To decide, to which cache to forward a request, the responsible node keeps track of different metrics, like bandwidth, storage and memory.

Mobile nodes:

Mobile nodes currently are identified by even TCP port numbers, which are statically assigned. These nodes do not store data items and do not take part in routing. Instead, nearby strong nodes forward messages and store the data.

2.1.2 Framework and Testing

The Framework layer provides the core functionalities of LifeSocial. These are data structures, like distributed lists, access control and secure storage, as well as different communication options. All these functionalities are provided to the upper-layer by a well defined interface, such that any plugin can make use of them.

Replication:

The data is replicated in the network to make the data available even when the peer responsible for the data goes offline. LifeSocial uses PAST working on top of FreePastry to take care of replication. Currently, PAST uses a *(k,r)* replication scheme. In such a scheme for a given data object with a key *k*, PAST ensures *r* replicas are made available of the data item with key *k*. PAST replicates the data item to the leafset nodes of the responsible peer and thereby maintains *r* replicas for a data item of key *k* in the network.

Data Structures:

Data structures are important to handle data in an appropriate way since they offer specific solutions how to organize and access data efficiently. Presently, a distributed list, a distributed set, a prefix hash tree and a distributed tree are implemented in the application. Since all these data structures are distributed in nature, they provide interfaces to manage and access the data in a distributed network. Search is implemented as a class in the PG, which is a collection of all search methods supported by the framework. Using this class, it is easier to perform searches as to use the distributed data structures themselves.

The distributed set and the distributed linked list splits a list of data items into single buckets of a specified size. The operations for an item of the list are just sent to remote nodes storing the affected buckets which then operate on local data structures. Since the data structures are distributed in nature, it manages the data and distributes the data over the nodes in the network. The access controls are enforced by encrypting the buckets with cryptographic keys. The distributed tree uses the distributed link lists. In this data structure, each node is stored in a distributed linked list which is split up into several buckets. Each buckets refers to the distributed linked list where its child nodes are stored. The tree structure can be seen in Figure 2.1.2.

Access Control:

Access control mechanisms allows only authorized users to read and write from/to data items present in the network.

Presently, access control mechanisms are implemented by using cryptographic keys. For read access in the framework, the data is encrypted with a symmetric cryptographic key, which is then made available to the authorized users, to a user group or a hierarchical user group. The symmetric key is encrypted with the group's public key or the user's public key. To access the data item for reading, the symmetric key is decrypted and is then used to decrypt the encrypted data item. The symmetric key can be decrypted by using the user's private key or the group's private key. In a hierarchical group, the symmetric key is retrieved by a depth-first traversal through the group hierarchy.

The write access to data is also implemented by using signatures. The public key of the owner or group is stored together with the item when it is first inserted. The data 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 data should be stored and whenever it is not valid, the storing node can simply deny the insertion of the data into the network.

App Store:

The App Store provides the functionality to manage the plugins in LifeSocial. These plugins are realized as OSGi bundles, which allow loading, unloading, starting and stopping the plugins at runtime. The present App Store mechanism allows two roles - the plugin provider and the plugin user. The provider is the owner of a plugin and has rights to modify or remove the plugin. The user has only read access to available plugins in the App Store. He is also able to install and use these plugins.

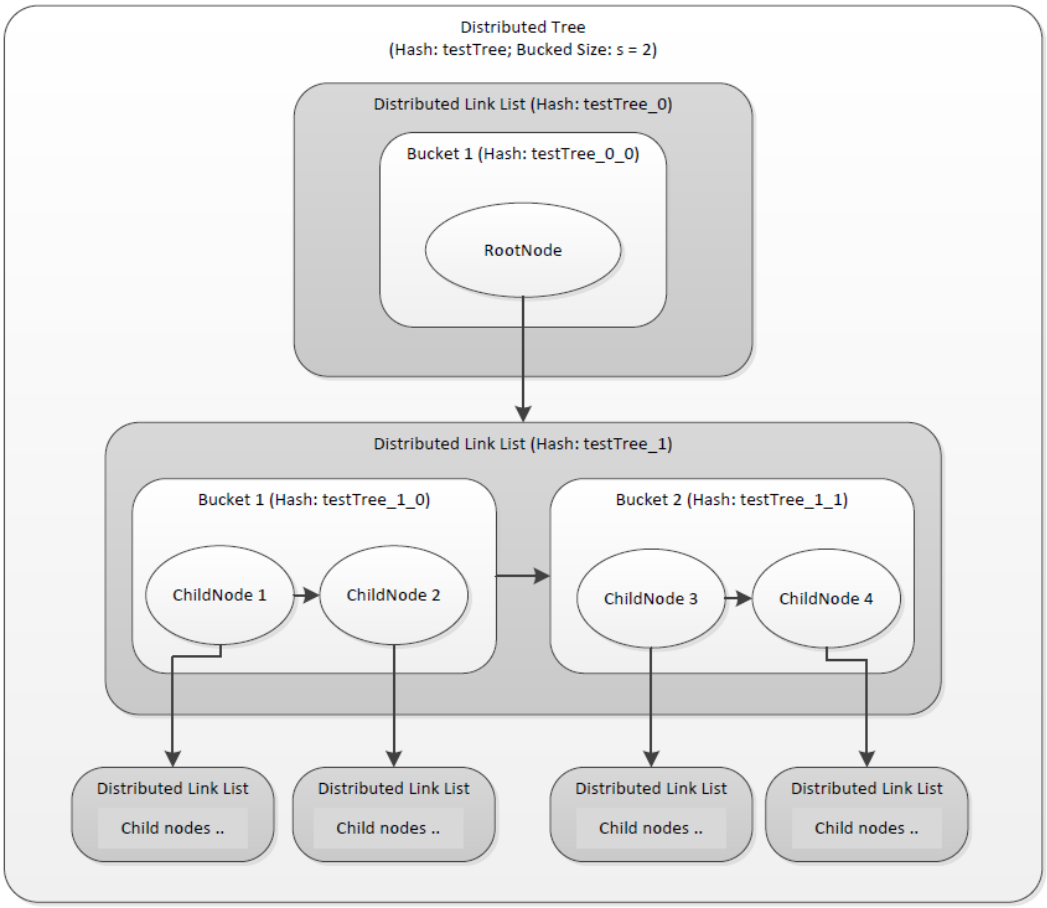


Figure 2.1.2: Distributed tree for a bucket with size = 2
(from Project Group 2012 - Final Documentation)

Message Dispatcher:

The MessageDispatcher is used to create instances of different types of channels: MessageChannel, TopicChannel and AggregateChannel.

The MessageChannel interface provides bi-directional 1-to-1 communication.It is used to send message to its defined address and allows to register a listener that gets notified, whenever a new message arrives at this MessageChannel. Each MessageChannel is identified using a unique name. It additionally provides 1-to-N communication by defining a list of receivers to whom the message should be sent.

The TopicChannel interface uses a publish/subscribe mechanism. It does not only send a message to a single receiver, but to every node that has registered to this TopicChannel.

The Message interface defines the minimal requirements that are needed to communicate in the network. Every message needs a defined sender and a receiver, to whom the message will be

delivered. While the receiver can be set by the user, the sender of a message is always enforced by the different channels to be the current ID of the user.

2.1.3 Plugins/GUI

The plugin/GUI layer is the top layer LifeSocial's architecture. A plugin is based on OSGi and extends the functionality of our application. The plugins/GUI layer uses the functionality provided by the framework layer. For each plugin it is possible to define a graphical user interface which is displayed as part of a Eclipse RCP window. Figure 2.1.3 presents an overview of all implemented plugins in the version of LifeSocial before our project group. Some of them are explained more in detail, now:

Groups:

The group plugin allows users with common interest to meet and get in contact with each other. The members of a group are able to create forums to exchange information. To provide a better overview, it is possible to structure the forums into folders. Each folder in a group can be seen as a sub-group and each sub-group can have its own forums and sub-groups. LifeSocial provides a GroupManager plugin to manage and to create groups.

Abstract Browser:

The abstract browser provides the basic functionalities to render HTML pages and websites in LifeSocial. The abstract browser is also used for text formatting in forums and wikis. This text formatting is handled via BBCode, which is a markup language and offers protection against phishing attacks. For the generation of HTML content the abstract browser uses StringTemplate. StringTemplate is an easy to use templating language and is usable in Java.

Voting:

The voting functionality provides a polling system in LifeSocial. Presently, it allows to create polls being either public or private to a user. The maximum number of votes that a user can add is defined by the owner of the poll. The owner is allowed to vote, as well. Each user is allowed to revoke their own votes and the mechanism allows the users to get the results of voting. In case of private poll only authorized users can participate and see the results.

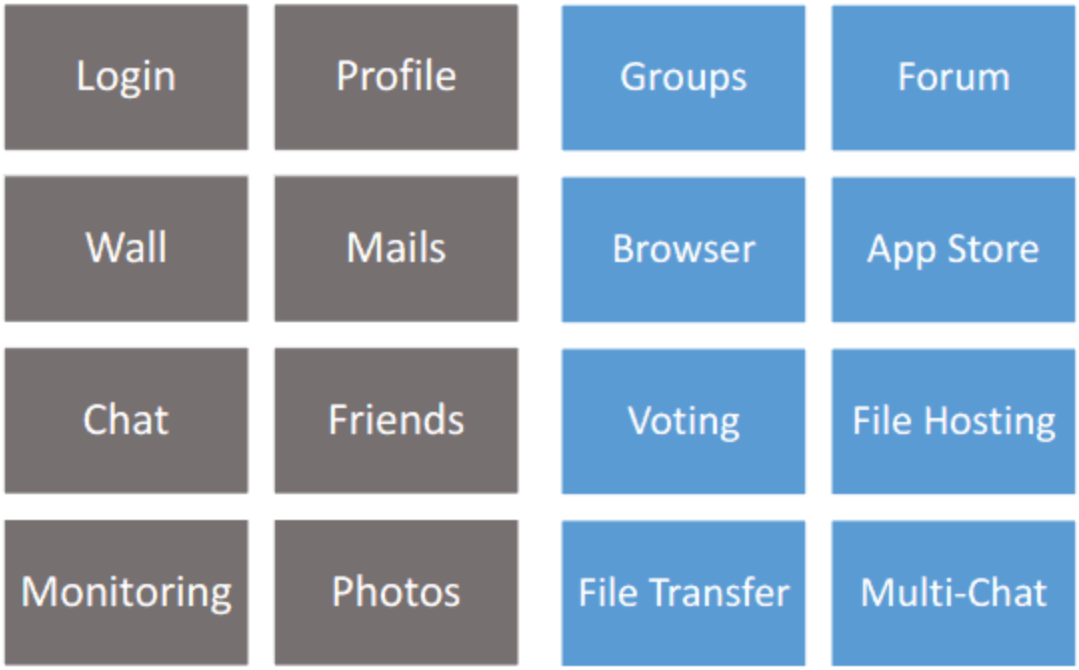


Figure 2.1.3: Overview of the currently implemented plugins.
(From PG 2012)

Chat:

The current version of LifeSocial also provides a chat plugin. With this extension, every user is able to chat with multiple other users. The graphical user interface to achieve that is called MultiView. In this view the user is able to invite friends to a so called Topic Chat, which needs a unique name. Besides simple text messages, the Topic Chat also supports a list of emoticons.

File transfer:

File transfer allows the users to send files to others in the network. Presently, the file transfer allows one-to-one communication i.e. the user can send the file only to another single user, but not to multiple users in the network. File transfer is implemented in LifeSocial using the MessageDispatcher plugin, the same plugin which is used for chat plugin.

Perspectives:

Perspectives are components of the Eclipse RCP. The perspective is a template that is holding the arrangement of different views or tabs. The user is able to rearrange these perspectives just as create new ones. Furthermore, each plugin itself can provide perspectives by arranging views provided by itself and the other plugins. With the startup, LifeSocial opens the last perspective and all views which were open at the last termination of the application.

2.2 Open Problems and Weaknesses

Before starting the project group, LifeSocial was rather a prototype showing the possibilities and opportunities of peer-to-peer technology. There has been done lots of research in the prior project groups which has been integrated into LifeSocial and successfully been tested. However, it was still unlikely that LifeSocial would productively been used in practice. It was tightly integrated into the Eclipse IDE which does not offer the usability, users are expecting. The following list gives a short insight into the problems we faced when collecting the requirements of our project group in April 2013:

- The initial setup is difficult and error-prone.
- Network Address Translation (NAT) currently is not working
- It takes several minutes to start-up Eclipse.
- The concept of views and perspectives is different from what the user knows from other social networks based on HTML.
- Navigation currently is provided by pull-down menus, which is rather inefficient and not what the user expects.
- The GUI looks old-fashioned and in most areas it is inconsistent.
- LifeSocial has been extended with a lot of features, but these features are not working in a stable way or are inconvenient to use (refer to Section 3.1).
- Existing features often are not integrated with each other (ex.: no possibility to directly start a chat with a group member, search plugin not integrated in other plugin GUIs).
- Because of the strong coupling to Eclipse, mobile nodes are currently not supported and it is unlikely that they can be supported in the future.

3. Proposal

The aim of PG was to enhance the LifeSocial functionality to provide a better user experience and to make it ready for the alpha test. The plan was as follows:

As LifeSocial was not running in a stable way, the **first phase** aimed to go through the problems listed in Section 3.1 and to fix them. Furthermore, we needed to fix the NAT issues as a higher priority in order to make the application more real and usable on a broad range of computer devices. Since the aim was to use LifeSocial as an E-Learning platform during the winter term of 2013, we planed at least basic E-Learing functionality to be provided. Dependant on the time left during the first phase, we also considered audio & video extensions of higher priority.

Whereas the first phase should concentrate on bug fixing and making LifeSocial to be reliable and usable in practice, the **second phase** aimed to implement further extensions. The focus is on such features that are related to E-Learning. Again, dependant on the time left, we planed to implement the other extensions as mentioned in Section 3.2. The aim especially was to have a stable version at Hannover Messe in April 2014.

Overall, at the end of the PG, the aims were as follows:

- LifeSocial works in a stable and usable way.
- LifeSocial works in non-lab environments.
- E-Learning features are provided.
- The high priority extensions mentioned for Phase 2 are working.

3.1. Proposed Improvements

The previous chapter gave some insights to LifeSocial. This chapter now gives an overview on the problems which we faced before starting the project group and provides the list of requirements we collected. In order to improve LifeSocial and to give a better experience for user, we identified existing problems and shortcomings which should be handled by our project group:

3.1.1 Major Problems and Limitations:

During our initial tests in our lab-environment, we discovered the following major problems:

Local Storage:
- When the peer joins the network, it uses the temporary local storag Now, when another user (without restarting LifeSocial) logs in on th machine, he works on the same data as the old user (ex.: sam friendlist)

Setup:
- Often, one cannot connect to LifeSocial, or one can connect, but nothir

happens at all.

Profile:

- Email address cannot be changed.
- When forgetting the password, there is no way to get the password fo the user.

Live Chat:

- The plugin frequently crashes.
- Sometimes, the "send" button has to be double clicked.
- No possibility to join an active chat session again, after closing th window.
- When unsubscribing, a new chat window opens.
- Chat invitations sometimes are lost.
- After an initial phase, messages are lost frequently.

Friend Graph:

- Very often, the Friend-Graph plugin crashes and LifeSocial hangs.

Friends:

- Friend List does not show which friends are online or offline.
- “Path to Friend” freezes LifeSocial.

Photos:

“Create Album” window does not show the lowest two buttons. The semanti of the elements on the “Create Album” window is completely unclear:

- What is the semantic of the drop-down box?
- Do the buttons “add” and “remove” refer to the drop-down list?
- Button “remove” does not do anything -> privileged users not removabl
- Lower two buttons partwise hidden.
- Unclear to the user that the lower two images are buttons.

Access control is not working at all for photo plugin. Buttons to create a album stay gray and often the GUI does not refresh. Creating an album a opening an album has long delays - no visual feedback. Unprivileged frien can see photos. A user can access the Friend-List of a friend and then ca access the photos of the friends of that friend.

Wall:

- There are many problems in this part, like cloned messages, message delayed, view not refreshing and lost messages. In summary, this plug currently is unusable (same problems in Comments)

Messaging:

- There is no notification when new messages arrive. The “New Messag window disappears after sending a message - How to send a secon message? There is no manual refreshing button. There is automati refreshing but it is huge delayed.

Groups:

- User cannot invite friends in the GroupManager window.
- One can join a public group through the “join” button (My friends → Sho Profile → View Groups), but there is no notification after it succeeds This is completely unusable.
- Renaming of Folders and Forums throws exceptions and does not work
- Not able to figure out how to enter a forum and to create topic (double-click on forum name does not work).
- When no group is selected it is possible to leave that none-existin group, which then throws a Java exception.

Voting:

- User cannot edit voting options after he created it.
- It is possible to press the vote button without any selection
- It is possible to vote many times for one voting. Only the last choice remembered. This behavior is correct but not intuitive.
- The voting is nowhere after accepting a voting invitation from a friend.
- Votes created by a user, sometimes do not appear in his vote-list.

Search:

- User cannot search groups.
- User cannot search votings.
- It seems only to support search for users.
- It is unclear what exactly is searched for.

Games:

Tic Tac Toe itself is not complete. We had the situation, where we had a player game, but actually the interface of player 2 was additionally shown another peer, which did not took part playing the game.

- In some environments, starting a game leads to crash of the plugin.
- Users cannot reject the invitation from a friend.
- The game cannot work correctly when the “New Game” button pressed several times, no matter on initiator side or receiver side.
- User cannot start a new game after he closes the previous one.

App Store:

- After choosing to add a new plugin, that dialogue cannot be aborted.

Localization:

- Parts of the GUI are in english, other parts in german.

3.1.2 Further Improvements to Existing Features

Apart from the more severe problems and shortcomings listed above, we thought about the following improvements:

Login / Registration:

- Error messages are generic and do not describe what error exactl happened (ex.: User forgot typing an ”@” as part of his email address).

- Time needed by the Login-Process often takes very long.

User Profile: The current user profile is very limited. The following features would be nice have:

- **privacy:** For each profile-element set access rights
- **custom elements:** Allow the user to add further profile elements
- **custom layout:** Allow the user to change the profile layout.

Currently there are two views - one for editing and one for watching a profile combine these views to one.

General GUI:

- Improve the look-and-feel.
- Make the user interface look consistent.
- Improve interactivity.
- Possible layout:
    - one centered view, friends on the side, navigation on the other.
    - contents of centered view is replaced when opening a new plugin.

Groups:

- Improve the usability.
- Features only can be selected by a right-click.
- no menu is provided.
- One cannot see the members of a group.
- There is no possibility to directly interact with group members (e.g. by chat)
- The whole GUI-concept has to be redesigned.
- View does not refresh, when group member creates folders and forums.

Email / Chat:

- Combine live chat and messaging as one plugin.
- Provide message history and possibility to archive messages.
- Provide a possibility to search for messages.

Friends:

- No possibility to block people. Even, when user B denies friend request some user A, user A can spam B with further friend requests.

Search:

- Provide filters and specify in which items to search.

3.2. Proposed Extensions

3.2.1 Network Address Translation (NAT)

Priority:
High

Motivation and Use Cases:

In P2P networks, nodes connect and communicate with each other directly without any server. However, NAT makes it difficult, because NAT usually changes several internal IP addresses into one public IP address. Therefore, external nodes cannot connect to the internal nodes behind a NAT since the external nodes do not know the exact IP address of internal nodes. Considering most peers in networks are behind NATs, the NAT issue causes huge damages to P2P networks. Moreover, different NAT types perform this IP modification process in different ways. To realize P2P communication behind NATs, a suitable NAT traversal solution should be implemented.

Challenges:

We need to select the best (relatively good at least) NAT traversal solution among many possible solutions for our P2P framework. And most solutions need some peers to play the role of rendezvous servers or STUN servers. What kind of peers should be chosen to act as these servers? And how? Should we implement several solutions for different NAT types, because most solutions are not able to work with all NAT types, or maybe we implement some powerful but complex solutions?

Required Adaptations in Life Social:

GUI	-
Plugins	-
Framework	-
Overlay	add NAT traversal functionality to Pastry

To implement NAT traversal solutions into the Pastry Overlay, we need to:
1. select a suitable NAT traversal solution
2. if server peers are needed, choose servers carefully (maybe in Leaf Set)
3. re-define the current P2P communication method adopting NAT traversal technique

3.2.2 Video and Audio Communication

Priority:

High

Motivation and Use Cases:

For a social network application, it is important to provide suitable techniques to support the

communication between the users. Besides live chats and messaging, another often used approach is a video chat. Therefore, the users are able to communicate via a microphone and a video camera. This provides an overall new experience and allows to a way of communication, which is more powerful. There exist some technologies which can help us realizing this extension. Two suitable technologies are OpenIMAJ and OpenCV which are usable directly in Java. The other approach would be to use WebRTC, which uses HTML5 and JavaScript.

The following use cases can be considered:

1. One to one or many to many video chat. Similar to applications like Skype.
2. One to many, where one person is a presenter and the others are able to see and hear this person.
3. Additionally to the functionality of point 2, the video is stored as well. This is, people are able to start the presentation from beginning and move in the video back- or forward until it is live again or the presentation is finished.

Challenges:

One challenge of this extension is to get access to the video camera and microphone. Moreover, it will be necessary to reduce the captured data to a streamable size. Therefore, a special encoding technique for video data is required. After the compression it is also important, that it is possible to integrate and display this data in the user interface.

Required Adaptations in Life Social:

The adaptions required depend strongly on the used technology.

OpenIMAJ or OpenCV:

GUI	Display encoded audio and video data, control interface
Plugins	Signaling and encoding
Framework	Streaming audio and video data
Overlay	-

This extension need work in several layers of our application. To use the video chat, it is necessary to exchange some signaling messages like start or cancel the video chat or adjust the data compression depending on the available bandwidth. This will be done on the plugin layer. Besides that it is necessary to compress the video data to streamable size. When the data is small enough the data needs to be transferred to the communication partners. Therefore, we can use a protocol like Real Time Media Flow Protocol (RTMFP) to exchange this kind of data. After

the data is exchanged we need to display it in the Eclipse SWT user interface. It needs to be figured out how this can be done. OpenIMAJ uses Xuggler and OpenCV uses FFmpeg for the encoding and displaying of media files. This will be done on the GUI layer of our application.

WebRTC:

GUI	Design HTML page
Plugins	Signaling and create server for HTML generation
Framework	-
Overlay	-

To create a plugin that uses WebRTC for the video communication has the benefit that it is not necessary to encode the data, because WebRTC already takes over this part. WebRTC also provides a signaling functionality, but it is necessary to forward this messages to the communication partner. Because this messages are simple text messages, they can be forwarded with already existing communication techniques of our framework. Besides that it is also necessary to generate the HTML pages, which are readable by a browser. Therefore, a server is required which supports that. Therefore, we can create this server manually or we can use a technology like JavaServer Faces, which separates the user interface from the application logic and other valuable features. Another point is to define if the server is only created for this plugin or if we want to implement the server in the framework. The last approach can help us to get away from Eclipse SWT and allows the use of HTML pages for the graphical user interface in other plugins.

3.2.3 Collaborative Work

Priority:
Medium

Motivation and Use Cases:
Collaborative work seems essential due to its growing demand in academia and corporate world where people from different places across the globe come under the same platform to create, modify or share the workspace synchronously or asynchronously. Google docs enables different users to collaboratively work on the same file at the same time i.e in a synchronous manner, unlike wikis which allows asynchronous communication. But these systems rely on centralized servers which cause many problems. It is a good idea to implement the feature of collaborative work on a P2P network, which has proven to be successful in a variety of applications.

The following use cases can be considered:

1. Collaborative text editor interface that enables users in the group to create, edit and share a common document. Work can be stored in different folders.
2. A chat feature can also be implemented that enables users to chat with their group members while working together.
3. Implementing audio and video support i.e users must be able to work together with an audio and video chat which would be very efficient.
4. Users can create groups with access control and can communicate with other groups by inviting them.
5. Users must be able to modify the data both synchronously and asynchronously depending upon the availability of the user. When users are online they work in real time else async. communication takes place.
6. Implementing calendar, to mark important dates and deadlines related to the workspace. And, whiteboard to enable the users to draw collaboratively which improves the understandability by pictorial representation.

Challenges:

In P2P network, as the data is replicated among all the peers, inconsistencies arise when multiple users work concurrently on the same document. Thus, overcoming this problem would be a challenge. To overcome this problem several approaches have been discussed, among which the suitable and reliable approach should be selected and implemented for our P2P framework.

Required Adaptations in Life Social:

GUI	Place for working collaboratively, i.e a collaborative text editor interface
Plugins	All the appropriate plugins required for the mentioned use cases
Framework	-
Overlay	-

To include the functionalities listed in the proposal, appropriate plugins need to be included like the chat plugin, which enables the users to communicate while working. The audio & video plugin enables to have a live face-to-face communication which would be an efficient and fast means of working together. But implementing this would be a challenge, as this needs to include live streaming. Additional features like calender, needed to mark the dates related to the workspace and white board, to use the diagrams while working together to make things clear when needed.

Graphical representation for all the features mentioned like the main text editor where users can open a document, start working on it and save their works in folders. Chat, calendar, whiteboard

representations which shows the name and details of the user who is modifying or currently using those functionalities. All these features need to have a nice and user-friendly interface to be developed, which enables the users to work comfortably in a collaborative platform.

3.2.4. Notifications and Live Feeds

Priority	**Task**
High	Notification Center for various notifications like invitations, chat requests (to support both online and offline notifications)
Medium	Live Feeds: gather wall entries of friends, present them to user -> nice to have, but not that crucial
Low	Implicit Notification Center: whenever a person is being mentioned in some photo, post or some comment

Motivation and Use Cases:

Notifications and live feeds are vital ingredients for social networking. In social networking, the people are often interested about what a particular user is upto or what are the latest update for a particular person of interest. Notification may also include when a new friend request is arrived, when someone commented or liked the post that we have shared or it may also arise at the time someone mention us in some post or in some photo. The notifications are also handy when someone invites us for some meetings or for some events.

In social networking, it also may possible that a person is interested in the other things too, like what is hot and happening in a particular area of interest. This may be provided by Live feeds. This can be think as a person has been subscribed to some communities or has been subscribed to some channel, so he/she must be able to see all the events about that community/ channel in real time. It might differ from person to person so live feeds has to be highly customized and highly dependent on various things like region, point of interest etc. It may also be used to provide some current news from around the globe.

Challenges:

A challenge for these extensions will be to keep track of all the things that a person is subscribed to and generate notifications as soon as a new update is available from that particular person or community. The notifications may be some pop-ups when the user is still online but it must be stored at some place in case user is not online or if he/she wants to check the notifications later on. For live feeds to work we must have those all channels/ communities that user has been subscribed to. Another challenge is to provide user the ability to customize these notification settings, as the user may wants to put things according to his/her own way. It is better to provide the functionality of customization. Another good thing can be to send a mail to the user (in case

user is agreed upon this) to notify user about the new things whenever they occur.

Required Adaptations in Life Social:

GUI	Place to put live feeds and notifications
Plugins	Appropriate plugings for these functionalities
Framework	Data structures has to be modified or maybe new data structures has to be created to keep the track of all the notifications and provide a mechanism t track it back whenever required.
Overlay	-

The plugins may be live-feed and notifications plugins. Optional plugins can be customized plugin for these two mandatory plugins. The above mentioned modifications/ addition of functionalities require some changes. We need to store the information about the notifications so that it can be retrieved. For example in the case of notifications, the structure may contain *notification_id, from_user_id, to_user_id, type*(friend request/ comment/ chat request/ event), *date* and *time* of the notification etc. Same can be applied (with small modifications) for Live feeds too.

3.2.5. Secure Monitoring

Priority:
Medium

Motivation and Use Cases:
In the framework we have a solution called SkyEye.KOM that is responsible for monitoring of the whole network. SkyEye.KOM forms a tree structure on top of the Pastry overlay to gather information about the whole network. To achieve this, every node sends its own data and the data collected from the child nodes to the parent node. This is done either in aggregated and non-aggregated form, depending on the type of information. The overall data for the network is computed in the root node of the tree and sent to all descendants step-by-step. This is a very convenient way because it prevents overloading of single nodes in the network. Also statistics show that this solution comes with very low overhead.
The problem with the current implementation is that it has no protection against manipulation of the collected information. Attackers could use modified versions of the client and send manipulated aggregation data to the upper nodes. This leads to wrong statistics in the whole network and wrong decisions.

Challenges:

The main task here is to make SkyEye.KOM capable to detect manipulation at least to a certain extend. To achieve this, it would be good to use multiple parent nodes for a range of nodes in one tree level. This way changes can be detected and even corrected very easily. Additionally it makes sense to use the public/private key mechanisms that are already used in the framework to sign the data messages. The combination of these two techniques should prevent most of the possible attacks. This way the weaknesses of the solutions are mutually eliminated. To make these protections effective, it must be ensured that local information of a single node that no other node is aware of, cannot be manipulated as well.

Required Adaptations in Life Social:

GUI	-
Plugins	-
Framework	● Multi parent support in SkyEYE.KOM overlay tree ● Use the public/private keys to create signatures for data messages ● Establish protection against local value changes
Overlay	-

First the way the tree is constructed must be adapted to support multiple parents. For this a few things have to be changed. This includes the responsibility function of the nodes that determine the ID of the current parent node. This has to be changed so that multiple hosts are selected for this purpose. Additionally uniqueness of this decision for has to be preserved for all nodes that belong to a certain tree-domain in the tree. Furthermore the aggregating nodes must keep track of the messages they receive. Because not all messages they receive are for different domains anymore they must distinguish here and have to select a suitable value that is then aggregated with the chosen values from other domains.

To make sure that the multi-parent method cannot be outsmarted, the messages sent to from nodes to the parent have to be unforgeable. So the information of a node that is aggregated in a parent node in the current implementation has to be signed and passed through a few hops towards the root node of the tree. For the signing the already existing private/public keys can be used.

3.2.6. Tags, Likes and Ratings

Priority:

Medium

Motivation and Use Cases:

The most social network applications try to promote the communication between the users. Therefore, a common approach is to support tags of users on different contents like pictures, walls, maps, etc. Our intention is to provide this and similar features to improve the user experience. Another feature of this kind is to allow users to rate content. Therefore, a common approach to achieve that is to use a suitable scale. An example for that would be to rate content with *good* or *bad*. This is also known as the so called *like* feature. Another feature is to represent hyperlinks to web pages with additional information. This can give the user a first impression of the content of a web page.

Challenges:

A challenge for this extension will be to define an interface, which allows the use of the named features in other plugins. Therefore, it would be good to keep the needed changes in these plugins low.

Required Adaptations in Life Social:

GUI	Visual representation of tags, ratings, likes and hyperlinks
Plugins	Modify plugins which should provide this extension
Framework	New data structure, which allows to add pointers to users and contains additional information like Tags, etc.
Overlay	-

The use of techniques like tags, ratings, etc. need extensions in several layers of our LifeSocial application. One part is, that we need a suitable data structure to save the arising information. To link content to special users, we need a pointer in the stored data items to these users. Moreover, we need to save additional information, like tags, ratings, etc. This allows adding tags on pictures at a specified location or tags at specified coordinates on a map. Maybe a more flexible solution would be to provide a key-value functionality to store an arbitrary amount of additional information. These are changes which need to be done on the Framework layer.

Another part is that we need graphical representation for the different features. For example a rating can be visualized as an amount of stars, numbers, thumbs up, thumbs down, etc. For tags of users it would be nice if it is possible to see short information about the corresponding user, like his name, etc. Similar to that hyperlinks could be identified and represented by a small description and picture of the targeting web page. Therefore, it is obviously worthwhile, that a consistent user interface should be used for the individual features.

This is, that all plugins need to get access to these features and their representation, which leads

to modifications in already existing and new plugins. Therefore, a suitable interface need to be invented.

3.2.7. Secure Replication

Priority:

Low

Motivation and Use Cases:

Replication in the network is done to maintain the availability of data when the responsible peer for the file leaves the network. PAST takes care of replication in the network by replicating the data to the peers in the leaf-set of a peer's routing table. The replication of data has to be secured to prevent attacks on data such as storage, retrieval or poisoning. To prevent these security mechanisms, it has to incorporated into the overlay network.

Challenges:

The replication algorithm must determine the broader address range based on the prefixes of the peers to replicate the data. Identification of the compromised peer and the address space is necessary to prevent this.

Required Adaptations in Life Social:

GUI	-
Plugins	-
Framework	-
Overlay	PAST, Replication algorithm to determine the safe possible id range to replicate data

The problem remains that the data replicated is stored under a specific address space in the network. Any attacker having access to these address space controls the peers in the address space and the data in that address space. To overcome this, data should be replicated so that it is distributed over some specific address range rather than the clustered address space, presently implemented in PAST. This can also help to minimize the effect of eclipse attack on the data present in the network.

The overlay layer modifications have to be made to maintain *(k,r)* replication strategy used in PAST. Additional replicas should be maintained outside the clustered id space. The present prefix based replication, with a prefix index of 4 should be changed to a viable prefix, so that even if

there exists an compromised id space, the data can still be recovered from the other replicas available outside the id space.

To maintain data integrity, cryptographic private-key signing and self-certifying on data can be implemented. Data signing will help to identify the peer, which has modified the data. Additionally, it is important to take the access rights and storage overhead into account. To prevent unauthorized modification of data, the private key signing of data can be used. The modification to the data is accepted only if the peers private key belongs to set of acceptable private keys, for which the public key used to secure the data. If the data update is valid and accepted, cryptographic hashes (self-certification) must be updated and the existing replicas should be invalidated and new replicas must be created using the *(k,r)* replication scheme.

3.3. Not Implemented Extensions

3.3.1. E-Learning

Priority:
High

Motivation and Use Cases:
Technology offers many innovative features that can be used to make instruction more appealing to learners. However, many of such features are more technology dependent and need a basic knowledge of technology to operate it or to use it (administration by tutor and usage by students). E-learning offers a huge number of possibilities beyond a one to one or one to many communication between student and lecturer. In our project group the implementation of E-learning can be realized by considering the following use cases.
The following use cases can be considered:

1. We can use live streaming and video on demand facility to provide the communication between group of learners and lecturers.
2. Feedback tools are also necessary, as course materials should be regularly updated.
3. On the spot voting facility.
4. Functions that take care of all the submissions, course materials, online examination and reminders for deadline and particular events.
5. Repository for online learning material which can also act as online library.
6. Forming of different groups, group communication and discussion forums.
7. Access to group calendars.

Challenges:
Supporting a large number of users having different technological experiences, a major requirement of the E-learning platform is good usability and an attractive look and feel. The

challenge is to catch users being unfamiliar to E-learning and make them use this platform. We should provide features, like video streaming and video on demand, pointing out the advantages of our platform.

Required Adaptations in Life Social:

GUI	Extension tools for E-learning and User friendly GUI
Plugins	-Online voting, Reminder alerts, Feedback, Video streaming and Video on demand, Library management, calendar plugin.
Framework	-
Overlay	-

Reminders for deadline may be a subject to think about because there are possibility of the responsible peer to be offline on particular time, in such case it is really hard to reach to that peer. Keeping a track of all the course materials is also a crucial task.

Most probably, we can extend the existing group plugin to provide the features necessary for E-learning. We could implement E-Learning as a special kind of group, offering special functionalities. The group plugin and GUI would represent a portal on top of which we could, one after another, implement the required extensions. However, as Section 3.1 has revealed, the group plugin first has to be redesigned.

3.3.2 Video and Audio Streaming

Priority:
High

Motivation and Use Cases:
One of the most popular applications in today's internet world is streaming. Therefore, implementing streaming in LifeSocial is a good idea. Nowadays, the streaming applications have a great deal of interest in research area. By utilizing the resources among the peers,streaming can be done effectively. Streaming can be done with slight alteration in BitTorrent concept especially in piece selection mechanism.The peers in LifeSocial form a mesh overlay and starts streaming.

The following use cases can be considered:

1. Groups can be created by the users like different genres of movies, music, etc. In these

groups users can share information about the files they have. So that it is easy to search the files.
2. Though, the friends does not have similar tastes in videos, users joined in a particular group have similar taste and can form a set of taste buddies.
3. Using search mechanism,users can search for particular video and start streaming.
4. List of torrents can be shared and comments can be incorporated.
5. While streaming the content among the peers the priority should be given to friends first, then friend of friends and in the least cases the chance can be given to all other social networking members.

Challenges:

The streaming applications are very time sensitive. If the peer responsible for streaming is offline, then it is a failure. It is hard to keep the playback latency low. The quality of video should be good otherwise the peers may leave the system as soon as they find the content with poor quality. The important challenge is to keep track of all the upload and download amount of each peers. The ratio of 1:1 should be maintained for uploading and downloading. Peers do not maintain such ratio are prevented from downloading further.

Required Adaptations in Life Social:

GUI	Place for searching the videos, streaming videos and comments for each each video torrents.
Plugins	Additional plugin for streaming has to be included
Framework	For many to many BitTorrent protocol with slight change in piece selection mechanism For 1 to 1 communication - RTMFP protocol can be used
Overlay	-

In GUI, it is better to have a place for searching video and for each videos the space for comments is included. Peers watching the video later can read the comments and get some ideas regarding the video quality. Moreover, the peer should have permanent Ids to identify action of each users. Therefore, a new plugin and some changes in the framework have to be done.

3.3.3. BitTorrent

Priority:
High

Motivation and Use Cases:
One way to share large files in a P2P network is to go the BitTorrent way. BitTorrent is a widely used protocol for file sharing over the network in an efficient way where a single file is effectively accessed by a large number of users. BitTorrent is based on peer-to-peer networking and this technique shows effective resource utilization among hundreds of connected peers accessing the same file by not compensating on their download speed. Once implemented in LifeSocial, users will be able to search, share and download large files such as movie, music, softwares etc. that has been shared in the network.

Challenges:

- Decentralization of the tracker is needed to make BitTorrent completely decentralized. This can be realized by using the DHT and communication protocols.
- Decentralization of the index search.
- Users should be able to share a file publicly, within a group or give access to selected list of friends.

Required Adaptations in Life Social:

GUI	An easy to use client interface
Plugins	Share, Statistics report, Auto Seeder, File search
Framework	
Overlay	

In addition, provide various incentives to those peers that seed the most ie. social recognition, top contributor badges . Recommendation of files a user may be interested in can be implemented as additional feature.

3.3.4. App Store

Priority:
Medium

Motivation and Use Cases:

Having an App Store allows users to browse through the catalogue, search, download and publish different plugins and applications. This way, LifeSocial can be enriched in a configurable way. The main idea is to give the opportunity to the users to create their own plugins or apps based on the peer-to-peer framework. These new plugins or apps can be shared via the App Store. Every user has two roles i.e. plugin user and plugin provider. Plugin providers can publish new plugins, release new versions, edit or delete them and users can search for plugins, download, install them and can manage the installed plugins. The current version of the App Store is only a closed or private repository. Moreover, we want to implement an open repository for our appstore. Therefore we will have two repository types in our framework i.e. managed repository and open repository.

Challenges:

- A suitable visual representation of the App Store.
- To implement an open repository in our framework (i.e. accessible by every user).
- Currently, the search mechanism in appstore is restricted to plugin names. To search a plugin, the exact plugin name is required. We can use a keyword-based search mechanism to overcome this problem.

Required Adaptations in Life Social:

GUI	A suitable visual representation of the plugins
Plugins	keyword-based search
Framework	possibly need to extend the existing Appstore
Overlay	-

We can implement an open repository so that the App Store is available for all the users. This also enables the users to list different plugins for specific content. The keyword-based search mechanism can be used for different types of plugins. Such a search is reasonable to find plugins that provide a certain functionality or belong to a certain category.

3.3.5. Map Reduce

Priority:

Low

Motivation and Use Cases:

In our p2p framework, data is distributed throughout the network. Furthermore, bandwidth often is limited. The basic idea of MapReduce is, to process data directly at the nodes it is stored on. Only the aggregated data is shifted back to the requesting peer. This is similar to the monitoring approach in the sense of data aggregation. However, the difference is that MapReduce is reactive, rather than proactive. MapReduce tasks are executed on request, whereas in the monitoring approach, the tasks are executed constantly.
Since MapReduce is a generic framework, lots of algorithms can be expressed using MapReduce notions. There are several possible use cases:

1. Collect and filter information of log-files residing on the peers.
2. Collect capacity information of the peers.
3. Use MapReduce to support keyword search in LifeSocial data items (for example user profiles)

In general, using MapReduce is promising, when large data collections shall be processed which are spread over the entire network, or when network bandwidth is limited.

Challenges:

The MapReduce framework has to be implemented such that it allows to execute generic tasks. When MapReduce is executed on data items of LifeSocial, especially bottleneck nodes have to be taken into account. A node storing an item to be processed might be overloaded, or even might not have the capacities to process the task. Furthermore, data has to be decrypted at remote peers, emphasizing the need to think about security issues. Finally, the implementation must work correctly and yet efficient, even under frequent churn and node failures.

Required Adaptations in Life Social:

GUI	-
Plugins	-
Framework	Implement MapReduce module providing a generic interface that can be accessed by the plugins
Overlay	Possibly extend the caching mechanism responsible for data items (see below)

Clearly, the MapReduce framework must be part of the framework-layer of LifeSocial, since it should provide generic functionalities to the plugins. The plugins should have the possibility to hand-over tasks with well-defined interfaces to the MapReduce module.

According to the current state of LifeSocial, a caching mechanism has already been implemented. Possibly, it can be used for MapReduce to avoid bottleneck nodes. Whenever a peer responsible for a data item is overloaded, it could redirect the request to a caching peer. This approach could be further extended, such that a responsible peer is aware of the caching peers having read access to the data item. This way, we could redirect processing requests to nodes having appropriate access rights to the data item.

3.3.6. Mini Games

Priority:
Low

Motivation and Use Cases:
Nowadays, entertainment is an important feature users look for in social networks. Games are one of the main features in that list. Initially, mini-games with low resource requirement will be a better choice. Later more sophisticated games can also be added.

The following list gives an impression on the requirements and nice-to-have properties:

1. AppStore is required in order to install/activate games. It will act as a repository of games available in LifeSocial.
2. Users will have 2 types of games that they can opt to play: PvE (player versus environment) and PvP (player versus player).
3. If a user is playing PvP, he should be able to chat with the other player simultaneously with the game.
4. Whenever a user wants to share his win/score on his wall, he/she should be able to do so. The visibility of the post can be managed by the user himself, as well.
5. Games should be further characterized like strategic, racing, sport etc.
6. User should be able to add games to its favorite list
7. Games played recently should be managed in another list where user can select if he wants to.

SinglePlayer:

- If user clicks on “Restart” the game has to restart from the beginning.
- He/she should be able to select difficulty level.

Multiplayer:

- While playing multi-player games users should be able to interact with each other.
- If a user wants to tease the opponent with some smiley he should be able to do so.
- If a player leaves the game without finishing it he/she should lose the game automatically.
- Among the friend list the User can see who all are online on the game portal.

- User should be able to send the game request to any of the friends online.

Challenges:

- Combining flash plug-in into the LifeSocial could be a challenge for the developer.
- In single-player games how to distinguish between levels of difficulty. Also for puzzle games the difficulty needs to be increased by every level or two.
- In multi-player games, on which device the game will run and how the resources will be stored/ shared. What happens if the two devices differ in resources by a large degree? The game could start lagging for the device with low resources, how can that be handled?

Required Adaptations in Life Social:

GUI	A central portal to manage all the games.
Plugins	Score Update, Winning Post to the wall, search, poking, and restart.
Framework	Data structure for storing scores or wins may be required, to store the favorite ones of the users and the recently played games.
Overlay	

As we propose to extend the App Store by an open repository, we could use a separate repository, or a special filter, for games. The game logic itself needs to be implemented on the plugin layer. Additionally, we need to consider suitable data structure to store scores or wins.

3.4. Proposed Milestones and Organization

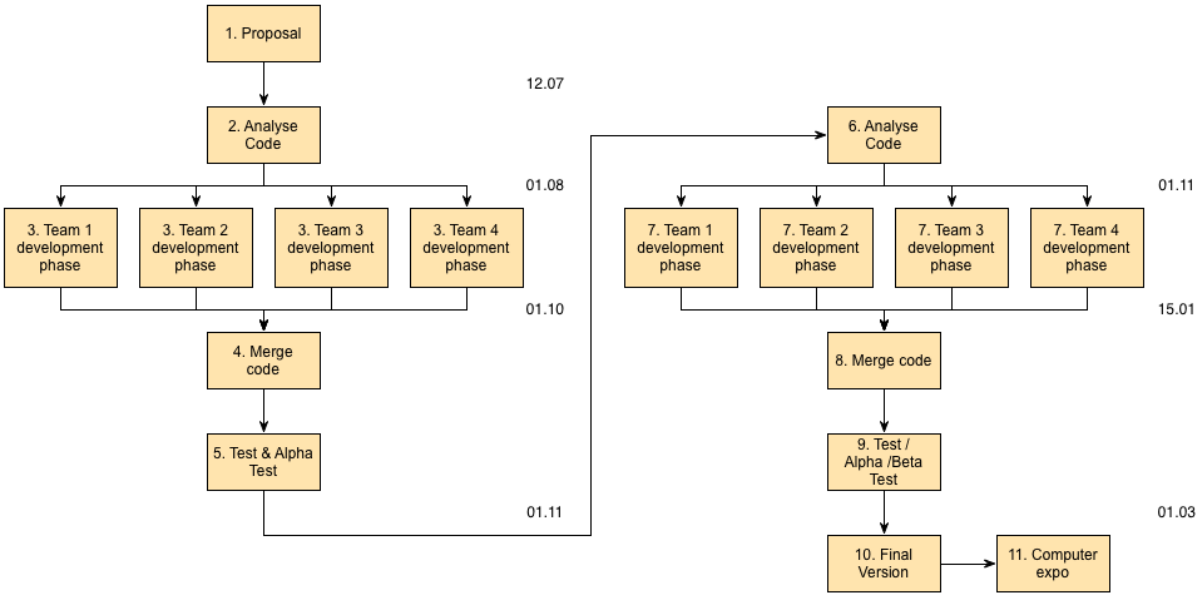


Figure 3.4.1: Timeline

Figure 3.4.1 shows the proposed timeline for the project. In general, we were working in small teams. Each development phase took about two months. We trusted in an agile development method. After each phase, the goal was to have a working and tested system.

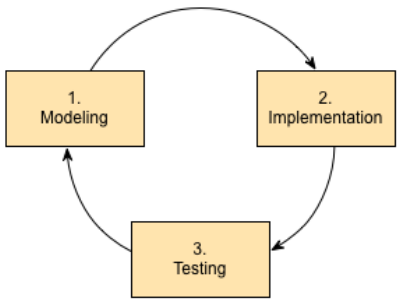


Figure 3.4.2: Agile Development

4. Outcome of the project group

As the chapters before gave an overview of the requirements and plans made before starting the project group, this chapter now focuses on the things which were achieved at the end in April 2014.

4.1. New/modified Architecture (EG, CP)

Figure 4.1.1 shows the current modified architecture of the LifeSocial. The old architecture has been modified at the top levels to successfully implement all the proposed features discussed in the previous sections.

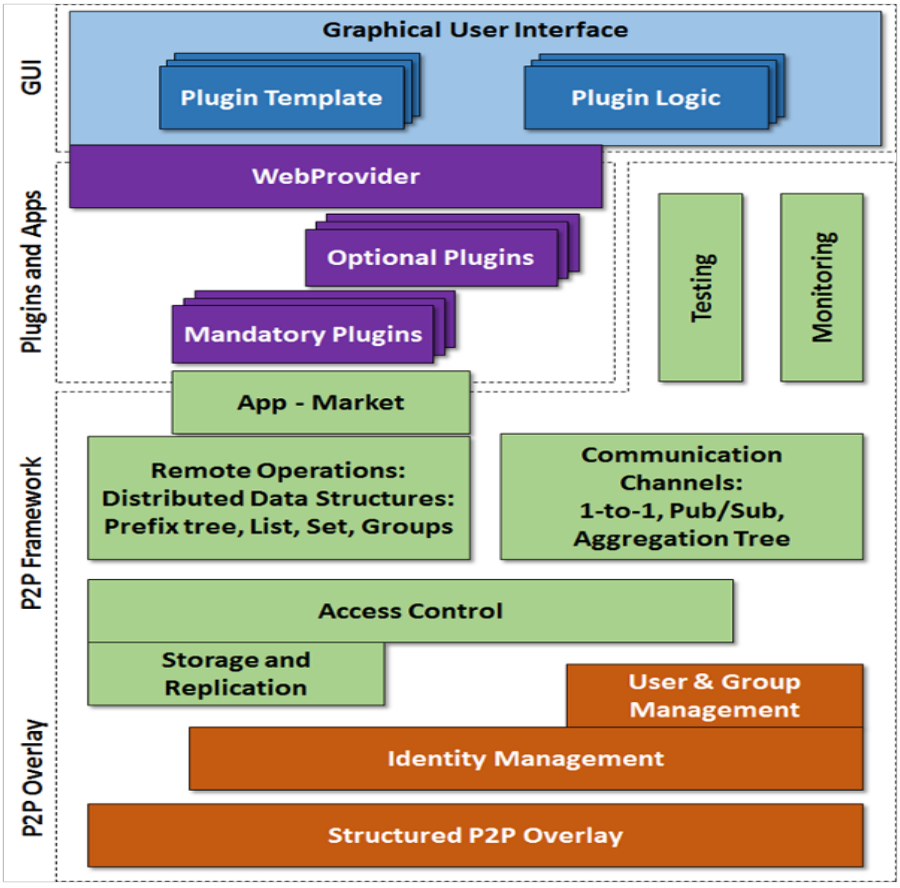


Figure 4.1.1: New modified architecture of LifeSocial (PG-2013/14)

In the new architecture, most of the modifications have been made on the top most layer, the GUI layer. As the old GUI was purely plugin based so the old GUI was not consistent and it was lacking in the interactivity, thus the usability of the LifeSocial was not upto the standards. To overcome this problem and providing a nice and consistent user interface, the LifeSocial has

adopted the new intuitive HTML5 GUI which makes it clean and elegant and more professional. The main components of the modified architecture of the LifeSocial are described in the following sections. The next few sections are going to cover what a plugin template does, how the plugin logic works and the WebProvider, which acts like a bridge and connects the GUI with the Plugins and application.
One another significant change is made in the Overlay Framework to make the user of LifeSocial aware about the right that he has been connected within and to distinguish between the rings. In last PG there was also no way a user can know a node to boot when he/she wanted to log in. To provide these functionalities, few changes into the Free Pastry has been made. Those changes will also be discussed in the subsequent sections.

4.2. Overview of Improvements and Extensions

4.2.1 Plugins:

Notifications

The notification plugin allows LifeSocial to inform the user about events, which need the intention of the user. One example for that is the friendship request. The initiator can send this request as an notification to the target and the target will be informed in a non disturbing way.

Therefore, the notification plugin supports the following features:

- encrypted via public key of the target
- handles messaging to contact users directly
- handles storing even when the target user is offline
- Other plugins can register, which allows:
    - display and sending of notifications
    - drop, display or replace arriving notifications

Error console

The error console plugin enables the user to see what is happening inside LifeSocial in a very user friendly manner and language. The idea of this plugin is to have a central place to log all the application level messages (informations, error, warning, debug). These messages can be read/seen by the user through accessing the logging console from the GUI of LifeSocial. On seeing, the user can take countermeasure for correcting particular problem (in case if it is an error message, for example).

This plugin functionality can be accessed once the user is successfully logged in to LifeSocial. For better readability and ease of access, all types of messages are displayed using different colors (can be seen in front of a message generated on error console).
Depending on the type of messages generated by the framework, these can be classified as

follows (details will be followed in next subsequent sections):-

- Warning (warn)
- Debug
- Information (info)
- Error

Groups

The groups plugin models a community environment, where a group of users are provided with a shared or common drive and a forum for discussions among the set of users.

The groups plugin provides the following functionalities

- Create groups
- Add friends to groups and notifies friends that they have been added to the group
- Shared drive to create folders, up and download files
- Forum for discussion, allows users to create threads and posts
- Rich text support for the forum posts

Voting

The vote plugin provides basic polling functions for users. Two types of voting are possible: public vote (accessible to any user) and private vote (only accessible to the invited users).

Functionalities have been implemented in voting:

- Create vote
- Get a public vote
- Remove vote from user's vote list
- Invite users to vote
- Participate in voting
- See vote result

Chat

The chat plugin allows the user to stay in contact with his friends in a direct way of communication. Therefore, our application supports a one-to-one and a many-to-many communication. For the one-to-one chat applies:

- encrypted via public key of the target
- history stored encrypted by public key of logged in user
- emoticon and url support
- attachments
- add user to create a many-to-many chat

Similar to this the many-to-many chat supports the same features as above but regarding the encryption we changed it like the following:

- symmetric key encryption implemented in TopicChannels

- key exchanged via notification

Audio / Video chat

To provide the user a more intense way of communication we implemented a audio and video chat. It is a one-to-one chat which provides the following features:

- Open standard WebRTC used
    - 128bit AES encryption
    - NAT traversal
    - Handles audio and video codecs and the access to the videocamera
- Signalling messages
    - Send over our network
    - encrypted via public key of the target
    - uses notification plugin to establish the call

Files

In the files plugin, the user can upload and download all types of files. The user can also share files via messages, chat or email among the users. The current plugin supports upload, download and deletion functionalities for the files. Thus, the users over a distributed network can share files with the help of this plugin.

Messages

In the Messaging plugin the user is able to send messages to his friends or other users. The behavior is similar to an email program. Therefore the complete plugin was rewritten. The following list shows the implemented features:

- Inbox and Outbox
- Labels:
    - create new labels with name and color
    - hierarchical structure of labels
    - assign or remove labels from messages
- Message
    - write a new message with target users, subject, attachments and text
    - replay and forward messages
    - mark as seen or unseen
    - remove messages
    - Notification when a new message arrives.

Profile

The Profile plugin has the information of every user and also allows the owner to decide which part of the profile data should be made to public or kept private by changing the privacy settings of

every fields. The new profile view is enhanced with multiple fields added thereby enabling user to provide more information about themselves. Editing of the profile data can be made in the same view compared to two views that was present earlier. The newly implemented features are briefly discussed below.

- Privacy
    - Allows users to control privacy over their profile data.
    - Privacy settings can be made on every field in the profile page.
        - Only you: Make the selected field visible only to the owner.
        - Your Friends: Friends of the owner can view the field.
        - Public: The field is not hidden to anyone.

Friends

In Friends plugin, the user is able to send, receive, accept, reject, block or unblock friendship requests. Fetching the friend list and block list is also a part of this plugin. Since all the friendship request can be seen in the Notification, user is able to accept/ block/ decline requests even if he/ she is offline at the time of receiving the request. Also, the current Project group handled improvements of blocking and unblocking another user from sending friendship request again to the owner.

- Blocking
    - Block further request from a particular profile. This will also add the blocked profile to User's Blocked List.
    - Unblocking from Blocked List.
- Online/ Offline
    - This functionality allows user to set 4 kinds of mode - Online, Away, DnD and Invisible. This mode will be shown to his name in all of his friends instance.
    - After successful login by user the mode is set as Online by default.

Calendar

Calendar plugin offers a private calendar for each user. In the Calendar plugin the user is able to create, delete and modify events. The implemented features are listed below:

- Calendar
    - Calendar view - Currently, the calendar provides a month view.The month view provides days of the month along with their corresponding dates.
    - Previous month - navigates to previous month.
    - Next month - navigates to next month.
    - Month navigation - user can directly select the desired month and year.
- Event
    - Create Event - creates a new event with event name, event description, event start/end date, and event start/end time.
    - Delete Event - deletes an already existing event.

- Modify Event - modifies an already existing event.
- Event Reminder - user is notified about the event.

Photos

The photos plugin enables the users to share their photo among their friends on the LifeSocial platform. Moreover, users can upload new albums, add photos to existing albums and delete photos from albums. The user is able to open photos in a so called photobox, which allows her/him to download these photos or to start a slideshow. The photo plugin is rewritten to provide access control to the users and other features like comment on the photos, display thumbnails of photos etc.

Privacy settings and GUI

- To protect the users data s/he is able to set different privacy settings for an album. All the privacy settings are described in details in the later sections.
- Albums abre being displayed with their name and description. In case of more than one album it displays the name of the album in a rectangular box with the thumbnails of the included photos in that particular album.
- Inside the album all the photos are visible with their small thumbnails. Clicking on the photo will display the photo in full screen with possible navigation and other options like download/delete photo.
- Slideshow of the photos and Commenting on the photos is possible too.

Wall

- Privacy
- Removed duplicated posts

Search

The search plugin enables the user to search for their friends and also their own profile. The search GUI has an advanced search form with many fields. Some of the search criteria used in the GUI are username, full-name, address, postcode, education, occupation, skills, interest, city, country and gender. User can enter any fields and search their friends who are registered in the network.

4.2.2 Framework

Storage items

In the previous project group they implemented a mechanism to store encrypted items. It also

supports an access control so that only the users with the corresponding rights are able to read that items. However, this created ***SecureStorageItem*** was not used in LifeSocial before and when we tried to use it to protect the data of the users the data get lost. After some debugging we figured out that the reason for that was that the data was signed incorrectly. So we wrote a fix to solve this problem but we also tried to change as less as possible to keep the overall architecture there. After that the related Junit test passes and we could start encrypting the data of the users. Later we figured out that the same mistake was done in two other implementations. So we fixed the ***PastryStorageItem*** and the ***DistributedDataItemContainer*** as well.

We also rewrote the caching of items. At least of ***SharedItems***. Therefore, we implemented a simple LRU cache in the ***SignPastryStorageDispatcher***, which is used as an middleware interface between LifeSocial and the Framework. We also implemented a mechanism to remember the time an item is stored in the cache so the user always gets the updated data. Whenever a plugin uses this storage dispatcher it can specify the maximum time an item is stored in the cache. But because of the LRU strategy it cannot be guaranteed that the item is stored for the defined maximum time. Another improvement here is that the cache uses ***SoftReferences*** of Java. This way the garbage collector can remove the cached items when the memory is running low.

Secure replication and modification of items

Replication ensures data availability in the network when the responsible node for the data becomes offline. Each data item is associated with a key. The replication scheme ensured availability of data, however, it does not not consider the content. Replication ensures there are enough copies of the key K in the network. To ensure that data was correctly updated, the workflow of storing the data in persistent storage was modified with security checks and key validations, so that updated data is available in the network and consistency of data is maintained. To ensure secure replication, before any replicas accept the data, the signature of the data is verified.

4.2.3 Overlay

NAT

Network Address Transalation (NAT) is a technique implemented in routers, to support computers with private IP addresses to connect to the internet using one (or even more) external IP addresses. To support this, the router creates temporary mappings of the form X:A <> Y:B, where X is the internal IP address and A is the internal port, and Y is the external address and B is the external port.

The basic problem with NAT is that external hosts, which were not contacted before, are not able to contact a node behind a router. Two solutions have been developed during the project group.

First, we support port forwarding. After the user has enabled a static mapping of the form X:A <> Y:B inside his router, pastry can be configured to use this mapping and to advertise Y:B as the address of the node. The router will ensure that all traffic for Y:B is forwarded to X:A. The second solution is relaying. Basically speaking, we use non-NATed nodes to relay traffic between NATed ones. The latter strategy has the advantage that the user does not have to configure his router and pastry. However, public nodes may become bottlenecks.

To support further NAT strategies, STUN has been implemented to detect different types of NATs. Currently it is unused, but in the future it can be used to implemented NAT-type dependant circumventing strategies.

Differentiating between Rings

A drawback so far was that each time the user wants to log into a ring, he had to know at least one bootstrap node. Furthermore, the user had no control on which pasty ring he joined. This was determined by the bootstrap node the user had to choose. To overcome these issues, we first assign ring identifiers to logical pastry rings. Furthermore, once logged in to a ring, a cache of online bootstrap nodes in that ring is built up. When the user wants to log into a ring, this cache is used to determine a bootstrap node. When the chosen node of the cache is offline, or moved to another ring, it is skipped and the next node in the cache is used. In the optimal case, the user does not have to know a bootstrap node anymore. Furthermore, the user now has control on which ring he joins and can create new ones in an easy way.

Others:

- GUI console
- Language support
- Extension of GUI support

5. Achievements in the Plugins/GUI Layer

The LifeSocial GUI from the last project group had several drawbacks both in terms of consistency and tight coupling with Eclipse. The current project group has done significant improvements over these issues. The final result of the PG has decoupled the GUI from the Eclipse, means no more old GUI but without sacrificing the functionality. Actually the application still runs but it is allowed to run in background only and the new modified GUI is opened in the browser.
In this section we will discuss what are the new things implemented in Plugins/GUI layer. These all technologies are used to develop a new GUI for the LifeSocial and to give it a brand new look.

5.1. Technologies used for new GUI

Current PG (2013/14) took the full advantage of some of the latest standard web technologies to achieve this milestone. Some of these technologies are:-

- HTML5
- AJAX
- JQuery
- Bootstrap
- Knockback

The combination of all these technologies gave LifeSocial a brand new cutting edge look which is throughout consistent, provides multi-language support (currently only English and German) and support for mobile devices (atleast GUI can be accessed). Some other benefits also includes a single page design and only parts of this page is getting changes as per the data requested from the user. Moreover the floating windows are also persistent even after page is getting refreshed.
For all these functionalities to integrate in LifeSocial, the GUI layer has been modified. Figure 5.1 shows the modified GUI layer (main parts are discussed later in this section).

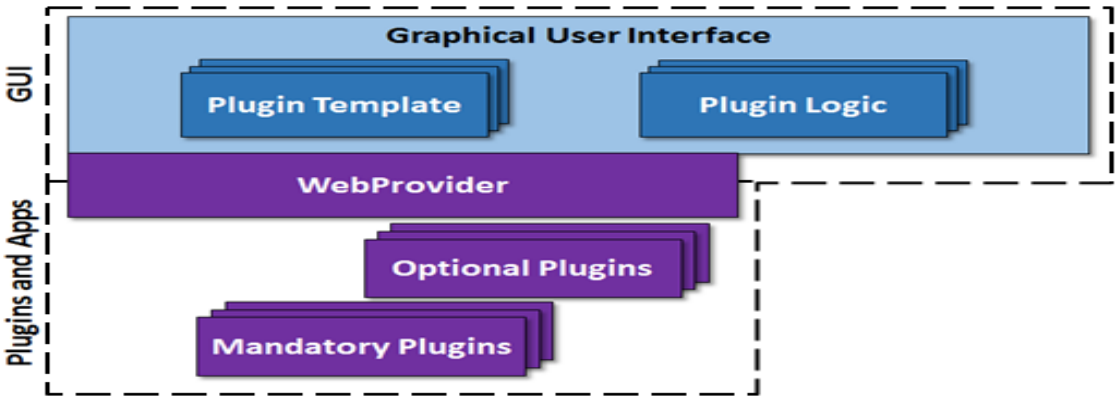


Figure 5.1: New modified GUI Layer (PG-2013/14)

This part provides all the look and feel of the current LifeSocial. Each and every function in LifeSocial is implemented through plugins only which user expects from LifeSocial. Whether it is

related to the functions of Voting or the functions related to a friend, each and every functionality inside the LifeSocial has been realized through plugin. For each majority of group of functions, we have a plugin.

5.2. Parts of Plugin/GUI

Mainly we will discuss these three topics which are essential part of the new GUI.

- Plugin template
- Plugin logic
- WebProvider

5.2.1. Plugin Template

The plugin templates are typical html file but they use knockout.js. KnockoutJS is a standalone JavaScript implementation of the MVVM (Model-View-View-Model) paradigm. Some of the advantages that KnockoutJS provides are:-

- Declarative binding can be used to bind each element to a model
- Dependency tracking reflects the value whenever the dependency changes
- Automatic UI refresh when data model's state changes
- Small and very lightweight

All the templates that LifeSocial uses can be found inside templates/ directory inside webapp (see WebProvider for file structure used in LifeSocial for GUI).
A typical example of binding used by the KnockoutJS is given below;-

```
<p>
        First name: <input data-bind="value: firstName"/>
</p>
<p>
        Last name: <input data-bind="value: lastName"/>
</p>
```

Code block 5.1: Example of KnockoutJs binding

This piece of code is a part of html file. Here the text values of two text boxes are bounded using data-bind.

5.2.2. Plugin Logic

Plugin logic is all about taking user events from the frontend sending it to the REST Handler using WebProvider and when the data is returned from the REST, the plugin logic takes that data and render it to the desired template. For making this all work, various other things have to be created like Models, Collections and Views (remember that we are using MVVM paradigm) which are the

backbone of this all logic. The Views and Models have to created into .js files of the corresponding plugins. For Models and Collections we have used Backbone.js and for Views we have used Knockback.js.

Backbone.js gives structure to web applications by providing models with key-value binding and custom events. With Backbone, the data is represented as Model. It gives the flexibility to create, validate and destroy the Models as per the programmers need. When UI changes some attribute of the Model, it fires back a change event and it can be used by the View to re-render itself using modifications made in the model.

A Model provides a basic set of functionality and contains interactive data. An example for the Model for Profile is given below:-

```
LS.Property = Backbone.Model.extend({
        defaults : {
                fieldName: '',
                fieldType: '',
                fieldValue: '',
                fieldPrivacy: 'private',
                privacyChangeable: true
        }
});
```

Code block 5.2: Example of Profile model used in LifeSocial

A Collection on the other hand is an ordered set of models. A typical example from the Profile.js is given below:-

```
LS.PropertyList = Backbone.Collection.extend({
   model: LS.Property
});
```

Code block 5.3: Example of Profile.js binding

5.2.3. Web Provider

Web Provider is a new plugin written by this project group which has an integrated web server in it. The main advantage of this new plugin is that it allows to access the API of other plugins using a HTTP/REST interface (more information about REST will follow in next sections). The embedded web server does not need to send dynamic pages and all the dynamic requests are served inside the browser only.
WebProvider acts as an intermediary between other plugins and the static files which the browser requests at times as in figure 5.2.1. All the calls from LifeSocial are served through the WebProvider.

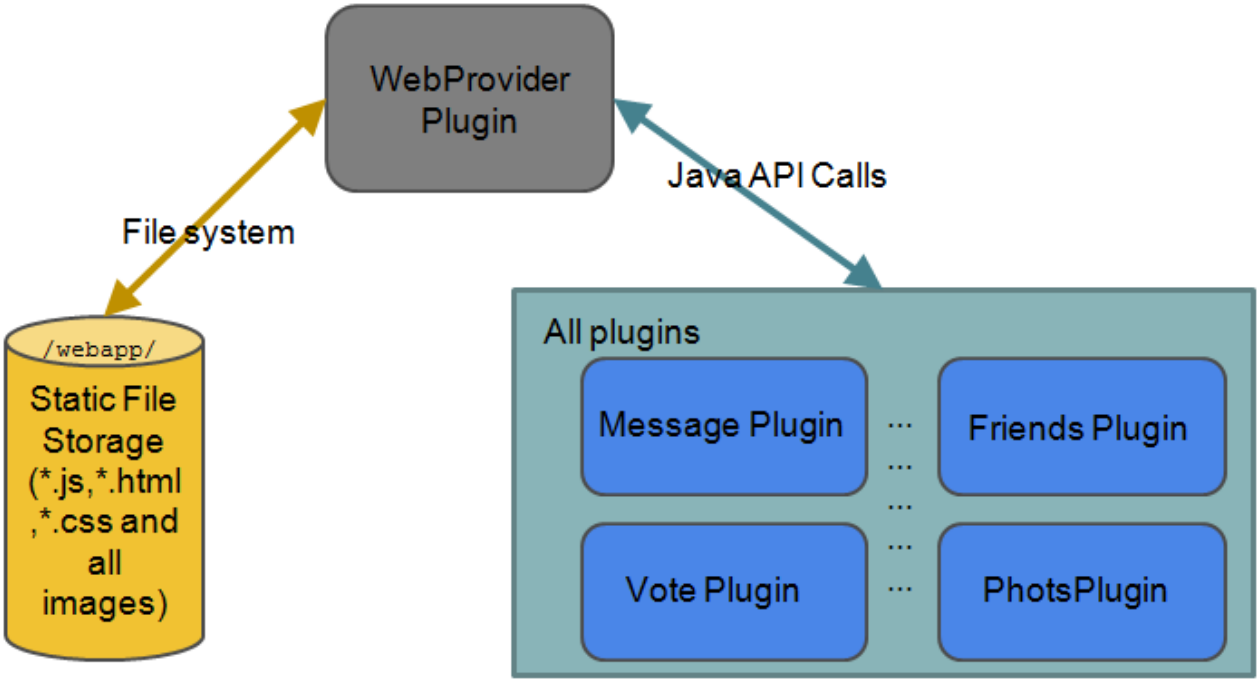


Figure 5.2.1: WebProvider plugin works in between File system and Java API

All the static files that are being used by LifeSocial they are inside a file system, called “webapp” which acts as the file system for the web server. Webapp is the root directory for the server and all the files resides in it, as shown in figure 5.2.2.

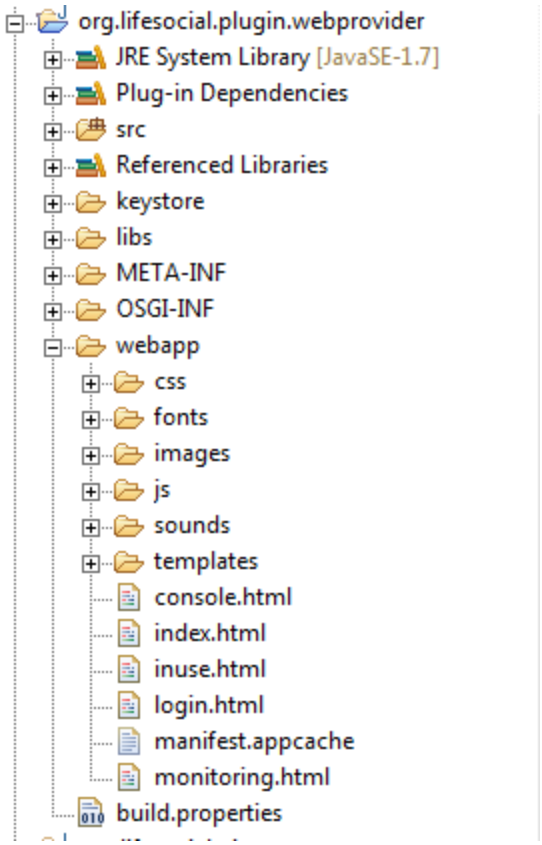


Figure 5.2.2: Webapp files structure and types

Webapp contains the files of type:-

- CSS
- Fonts
- Images
- Javascript
- Sounds
- HTML Templates

5.3. REST Handler

REST Handler is a class which is used by the WebProvider plugin to access API using AJAX call. These may include GetMessage, GetVote, UploadFile or getting some data from the network (profile pictures etc). REST Handler class consists of multiple if-else-elseif blocks and the requests are made through baseRequest/target, e.g., photos/getalbumlist, where baseRequest is photo and target is getalbumlist. In other words it can also be viewed as the REST Handler is calling getalbumlist method from the photos plugin. The result of the this call is transferred to the browser using response.

```
Response.getWriter().println(responseJSON);
```

Code block 5.4: Response of REST Handler call fed to browser

This line sends the output to the browser and after that we have to tell browser explicitly that the request was successful using following line of code:-

```
Response.setStatus(HttpServletResponse.SC_OK);
```

Code block 5.5: Telling browser that request was successful

REST Handler handles requests sent by the browser and executes the corresponding method from the plugin defined inside the else-if block. The typical example is shown here:-

```
String response= gson.toJSON (PluingProvider .getPlugins().
getMessagesPlugin().getOutbox());
```

Code block 5.6: Executing a method and getting the output in a variable

5.4. HTML Architecture

Short overview:

This chapter discusses the concept of the surrounding HTML architecture used in LifeSocial. HTML 5 is introduced to overcome all the difficulties like inconsistent design structure, which were faced due to the integration of the previous version of LifeSocial, which used an Eclipse RCP graphical user interface. Now the LifeSocial GUI is totally decoupled from Eclipse RCP and uses a web browser as an activity platform while the actual application serves as a backend.

Adaptations to LifeSocial:

GUI	HTML 5, knockback.js, bootstrap, websockets, jquery, AJAX
Plugins	Webprovider was added, Plugin contains the GUIs for the other plugins
Framework	
Overlay	

Detailed description:

In the original version of the software a graphical user interface was used that was based on Eclipse RCP. The problem with this user interface was that it was not so well accepted by the users, because it had a somehow old-fashioned design with many inconsistencies.

For the new version we decided to implement a new GUI, which is implemented as a web app relying on latest HTML5 and Javascript technology, because it handles most of the evaluated problems. As any other functionality that is related to the social network part of the software, the webprovider is implemented as an osgi plugin and uses this technique to communicate with the other plugins. Currently the user interface source code for all plugins is part of the webprovider plugin for simplicity. Because all plugins that are currently implemented are more or less mandatory, this is not an issue. The previously implemented app store is not yet integrated into the new web based gui, but the interface for loading optional plugins is provided.

For the presentation of the web user interface a normal web browser is used instead of a Eclipse webview component to allow the usage of the latest HTML5 and Javascript technology and to ensure that the application is completely decoupled from the graphical user interface. So the actual application does not have an own user interface anymore and uses an integrated http server to communicate with the browser. Therefore, it can also be used as a background service. This also allows to use a web browser on a different machine than the actual p2p application and mobile devices can be used as well to interact with the network.

Browser-Backend-Communication

Between the backend application and the web browser there exist three different forms of communication, as it can be seen in the figure 5.4.1:

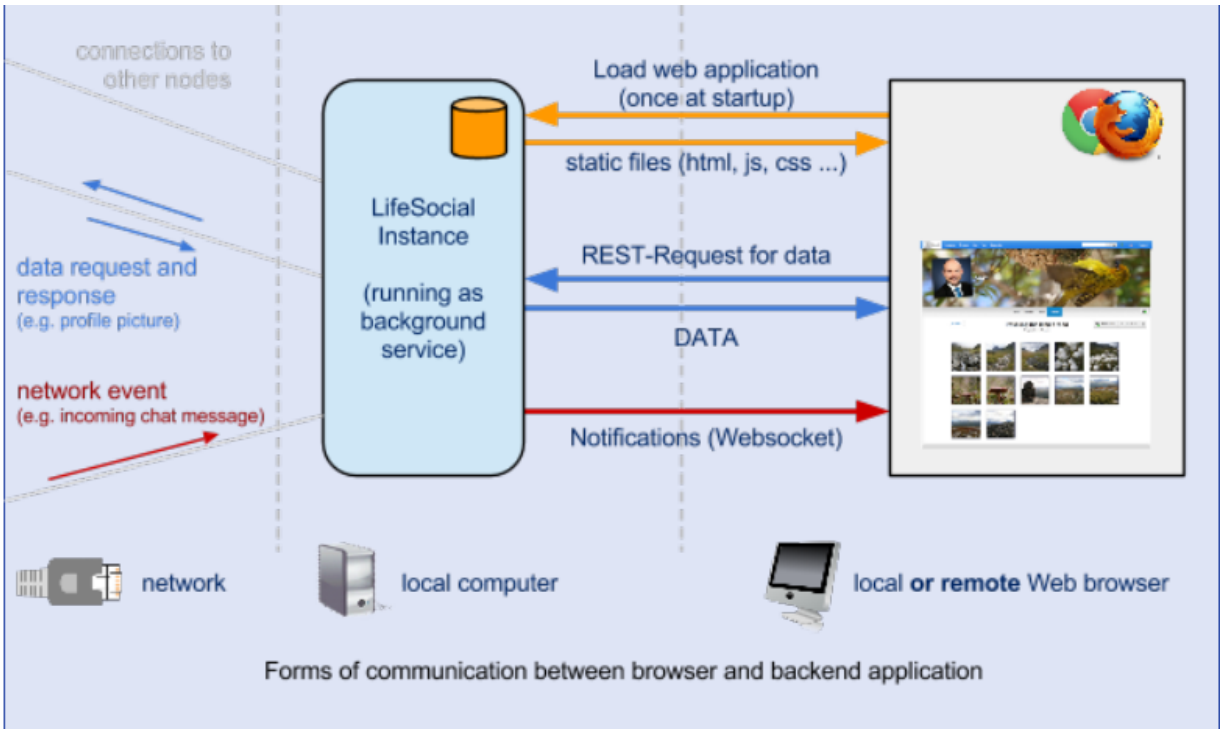


Figure 5.4.1: Forms of communication between browser and backend application

The first kind of connection is used to transmit the HTML, Javascript and CSS code from the application to the browser. Because all this elements are static in our case, this only happens in the beginning, when the user opens the GUI.
The remaining two connection types basically represent a mapping of the requests and messages that are sent to and received from other participants of the network. They are processed and transferred as JSON-serialized data which is very usual for javascript applications as it can be parsed efficiently and has relatively low overhead. For all requests a REST Interface is used, whereby especially the graphical interfaces of the plugins make use of it to interact with the plugin backend implementation. For communication that is initiated by the backend and subsequently from the network (e.g. status messages, incoming messages from other users), websockets are used. They provide a simple way to send data from the HTTP server implementation to the browser without the need to poll for new data in certain intervals.

Structure of WebProvider Plugin

The Webprovider plugin contains both, the Java classes used for the webserver and request translation to the plugins as well as the static html, javascript and css files for the web application. It uses the webserver library jetty to handle the connections from the browser. As shown in the following illustration, the requests from the connections are then handed over to a handler class that is suitable for the current type of request. Depending on the actual request command this class yields the response.
For static files of the web application the WebAppFilesHandler class looks up the resources folder of the web provider project and delivers the files to the browser.

The actual data for the plugins that is transmitted via REST, is handled in the RESTHandler class. This class only acts as a delegator. For each plugin a handler class exists that receives and interprets the request. The request is then forwarded and executed by the actual plugin implementation. If there is data expected as a response to the request, the data from the plugin is serialized to JSON in the PluginHandler and transmitted to the browser.
For events that are sent from the backend application to the browser web app, websockets are used as a unidirectional channel.

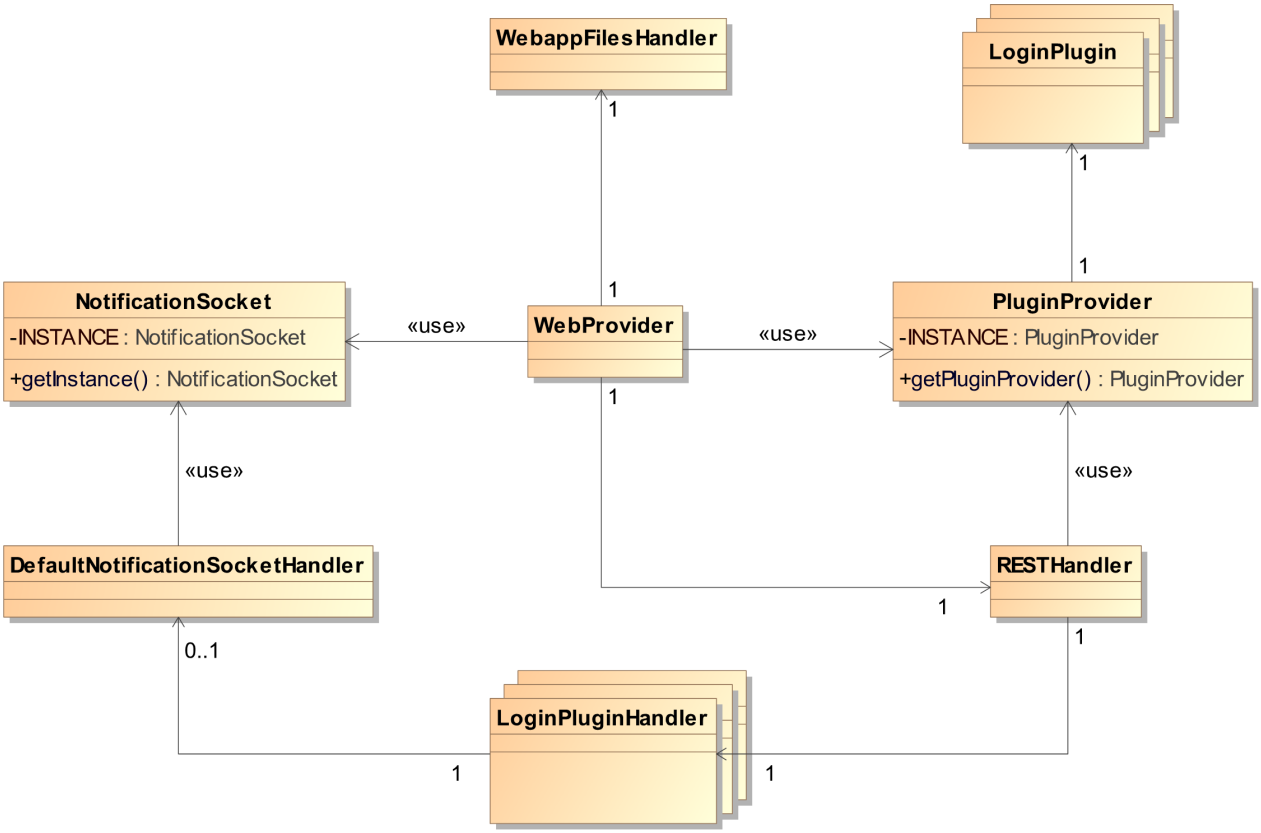


Figure 5.4.2: WebProvider Plugin Structure

Adaptions at other plugins

While creating the web provider the need arose to change the way how the other plugins are doing their network requests. In the previous version, the requests were sent asynchronously so that the calls are sent and then returned. The plugin then informes about the arrival of the requested data. This was necessary because waiting for the requests would have blocked the whole RCP user interface. In the new graphical user interface although the javascript code in the browser runs in a single thread, the requests are sent asynchronously anyway. So the requests were changed to synchronous calls because it makes the data handling easier.

Beside this, the RCP interface for the plugins was completely removed, only a tray icon was added to give the user some control over the backend application.

Web Application

While the web provider plugin is an osgi plugin programmed in java, the actual gui is based on HTML5 and Javascript. This web application is implemented as a so-called single page application. This means that it basically consists of a single html page which is kept during the whole usage session. Only parts of the page are exchanged dynamically when the user interacts with the page. This approach has some advantages over backend generated pages that are exchanged with each request.

First, only the data that is actually requested or has actually changed is being loaded from the backend and subsequently from the network. Second, it gives the user a more fluid way of interaction as there is always at least the scaffold of the page visible and the loading elements are animated. Furthermore this technique allows to use elements like windows (e.g. chat window) and keep them open all the time.

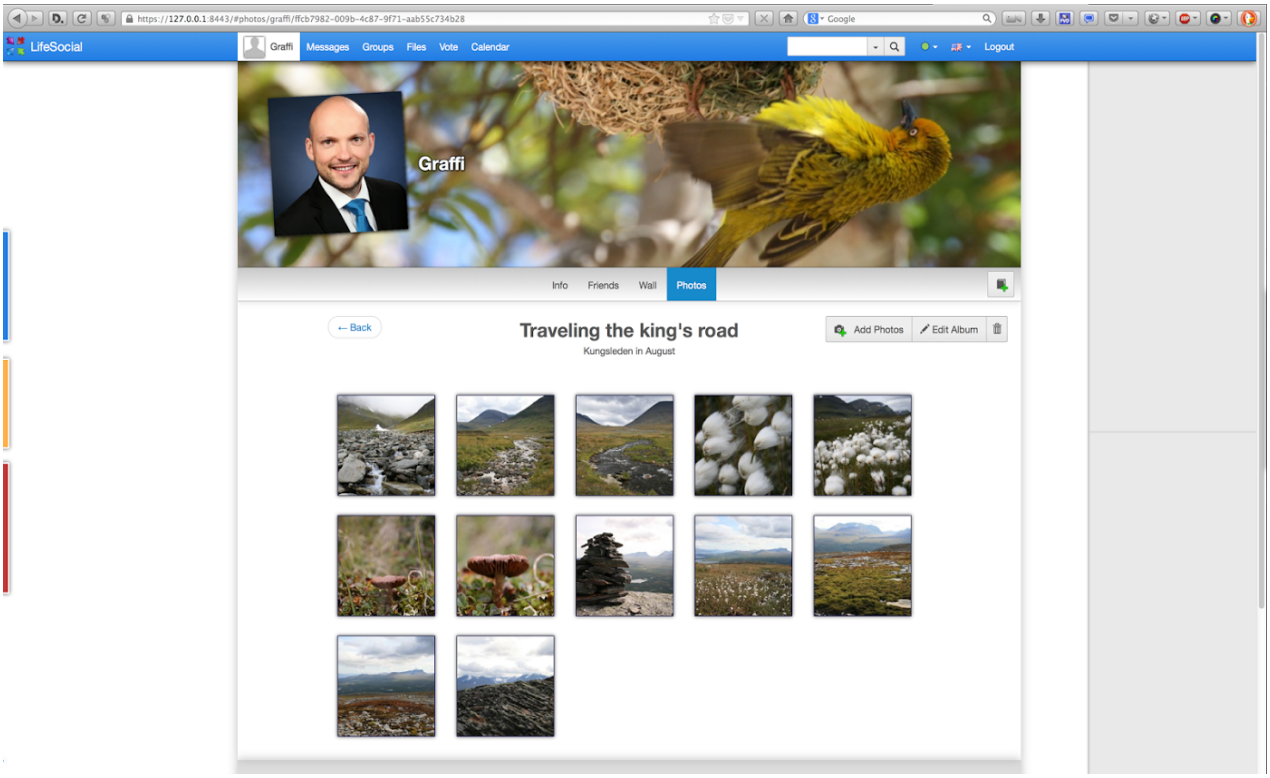


Figure 5.4.3: Snapshot of a User's Photo page

The used approach has a high demand of structural organisation because there are many components that have to be rendered in the single page. Because a simple stringing together of ajax calls is not expedient here, we decided to utilize some libraries on the client side that can handle the needs efficiently. We choose a stack of libraries which is called knockback.js, that basically brings together the best parts of two libraries, backbone.js and knockout.js. While backbone.js is used to request, retrieve, process and store the data that is coming from the

backend application, knockout.js is used to bring this data together via so called templates, which are presented to the user.

Basic steps to create a plugin

To create a plugin with an interface for the new system, some things are necessary that substantially differ from the steps of creation of plugins in the old GUI.

Backend part:

All plugins, which are used via OSGI should have a definition file that contains the provided services and the used references so that the system can resolve the dependencies between the plugins. Typically an interface class is used for this. While this was also the case for the old RCP GUI, the code for the interfaces is stored in the webprovider. So the provided interface of the plugin must be referenced from there.
In the webprovider itself a so-called PluginHandler must be implemented for the plugin, which translates the browser requests to the API calls of the plugin and transforms the provided data to the correct serialized form. This PluginHandler class also has to be registered with the RESTHandler.java which maintains two lists of PluginHandlers. Only one of this lists is processed depending on if the user is logged in or not. So typically the list for case “logged in” will to be used if the plugin is not related to logging in or out.
In the PluginHandler itself it must be distinguished what REST call the user used. This can be achieved by interpreting the target parameter of the handle method.
To use the websocket communication channel from the backend to the frontend, a DefaultNotificationSocketHandler object should be instantiated which provides a method *notifyClient* that allows to send data to the frontend.

Frontend part:

For the part of the code that runs in the browser normally two files have to be created which also have to be referenced from some other place. On the one hand this is the plugin javascript file. This file should contain at least the necessary backbone models and collections alongside with the knockback.js/knockout.js ViewModel definition that uses these models. On the other hand a template file is used to render the contents of the viewmodel. For further information about how to use these libraries, please refer to their documentation.[2]

Internationalization

To provide multi language support, a file called lang.js is used. It contains the translations for all texts in the application. To add a translation, just add an identifier and the corresponding translations (currently german and english are supported) to this file. The translations can be used in the template files by inserting:

```
LANG.getText('translationIdentifier', [insertions]);
```

The second parameter of this call is an optional array and provides the ability to replace placeholders in the translations, which are marked using *%1%*, *%2%* and so on.

Registration in index.html

To use the plugin, the javascript file has to be registered in the index.html file. To do so, a script tag with a src attribute pointing to the file has to be added.

Defining the location in the website where the plugin is rendered

Normal plugins typically render its contents in the center area of the web page and only show up in response to user actions. For this it is necessary to register the plugin at the page router manager with some identifier. All page requests of the form http://address:port/#identifier are then intercepted by the page router and the plugin is shown. The registration is done by using the following call in the javascript file:

```
PageRouter.route('identifier', null, function(user) {
    PluginView = new LS.PluginView();
    PageRouter.loadPage(kb.renderTemplate('templname',    PluginView));
});
```

Furthermore it is possible to render the plugin's ui at some static place of the web page so that the plugin persists when the user navigates. For this some kind of HTML Element with an id has to be added to the index.html file.

Navigation

For non-persistent plugins it may be desirable to add them to the menu. There currently exist two levels of menus. The first level menu is located in the bar at the top of the page. Use the following code snippet in the javascript file to register the plugin.

```
Navigation.registerMenuItem({
    name: LANG.getText('translationIdentifier'),
    link: 'identifier',
    stackable: true
});
```

Worth mentioning are the parameters of this call. While the name parameter just specifies the label of the menu item, the link defines where the user is redirected to, when he clicks the link. In this case the user would be routed to http://address:port/#identifier which would be handled by the previously defined plugin. The third parameter, stackable is used to mark whether the item can be moved to a drop down menu in case a small screen device is used and there is not enough space to display the whole menu.

Apart from the main menu, several sub menus can be specified where items can be added. Currently this is only used for some user centric plugins like wall, photos, friends list and user information. To add an item to such a sub menu, a similar function can be used:

```
MySubNavigationMenu.registerMenuItem({
    name: LANG.getText('menuPhotos'),
    link: 'photos'
});
```

Backend to Browser communication using websockets

To be able to receive messages from the backend, a websocket callback has to be registerd at the websocket manager. This is as easy as defining the an identifier, which can be the plugin name again and a function that is called when data arrives:

```
WebsocketsManager.addMessageEvent("myplugin", function(data){
    //Some action here
});
```

Useful development tools and advises:

During plugin development a few options in the WebProvider.java whould be changed which are used to configure the some optimizations so that efficient debugging is possible.

Option	Debug	Release
STATIC_FILES_PREVENT_CACHING	true	false
COMBINE_JS	false	true
COMBINE_CSS	false	true
USE_APPCACHE	false	true

Furthermore it turned out that the debugging console of google chrome is very helpful for debugging and as an extension to this the knockout.js debugger should be installed to the be able to observe the allocation of the variables.

Future work

One important aspect to treat for the further development should be the completion of the plugin interface for the webprovider. It was intended to have some interface for the webprovider that is

exported via OSGI and allows to register PluginHandlers and (web-) resource files from external projects. This also reenables the option to integrate the currently disabled app store again.

5.5. New Plugins

5.5.1. Notification Plugin (CH)

Short overview:

The idea of this plugin is to inform the user about what is going on in the network. An example for a notification would be a friendship request or an invitation to a chat. These notification are displayed at the right upper side of LifeSocial. A sound indicates that a new notification is available, so the user is informed even if s/he has left the LifeSocial browser tab.

Adaptations to LifeSocial:

GUI	Several notification templates
Plugins	**NotificationPlugin:** ● Send or display a notification ● Remove notification ● Plugins can register to this plugin **NotificationPluginHandler:** ● registered to RESTHandler ● request current notifications ● User action: remove notification ● inform frontend when notification arrives
Framework	
Overlay	

Detailed description:

The Notification plugin allows to send, display or remove notifications. A notification consists of the target username, the responsible plugin id, a map of data, an integer which indicates the kind of a notification (This is currently not used but could be used to highlight notifications), and the message type which is also used to define the template name for the GUI. Notifications are sent via the ***MessageDispatcher,*** which means that they are sent as a ***SecureMessage***. This messages are encrypted via the public key of the target. Notifications are stored in a **DistributedLinkedList** to allow that other users can store notifications for me. This is essential to prevent the loss of notifications. Figure 5.5.1 illustrates how this is guaranteed:

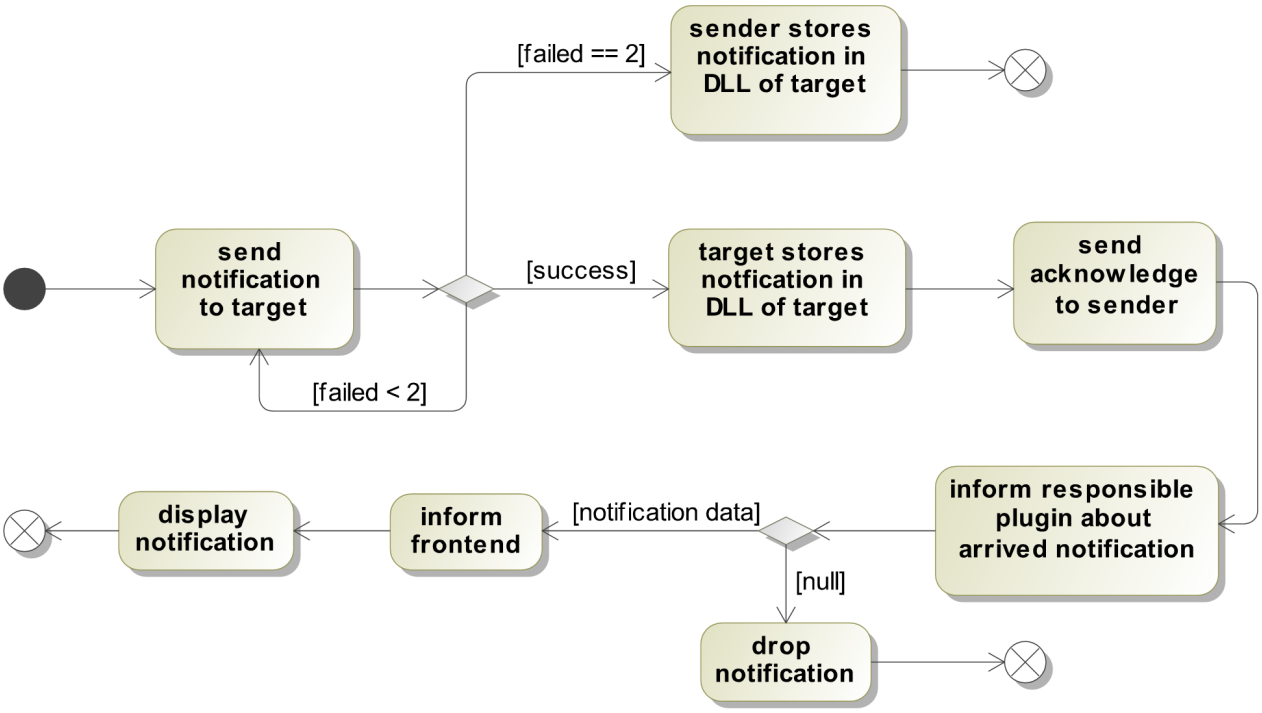


Figure 5.5.1: Send, store and display Notifications

The sender first tries to contact the target directly. When this works the target processes the notification, sends an acknowledge to the sender and stores the notification in her/his ***DistributedLinkedList***. The sender assumes that the sending failed when s/he does not receive an acknowledgment after a certain time. The sender tries to contact the target again until it failed for the second time. Then the sender stores the notification in the ***DistributedLinkedList*** of the target. Notifications are stored as ***SecureMessages*** to prevent that other users in the network can read the notifications. Due to the problems with the ***DistributedLinkedList*** we also serialize the Message to a String as a workaround. The reason for that is that there is still a problem in the ***DistributedLinkedList***. If another type instead of a String is defined as the generic type of the ***DistributedLinkedList*** the list behaves weird. For example it often returns the wrong entries from the list.

To send notifications the other plugins can register to the Notification plugin. Therefore, the plugin needs to implement the interface ***INotificationPluginListener***. This interface has only one method which is called ***confirmArrivedNotification(...)***. This method is called by the notification plugin when a notification arrives. It is only called for the responsible plugin. If not responsible plugin exists the notification is directly shown. Via this method the responsible plugin can decide whether it want to display, drop or replace the notification.

The method ***displayNotification(...)*** of the Notification plugin allows LifeSocial to display a notification on the current instance. This is for example used in the Calendar plugin to indicate an appointment.

The frontend uses the method ***getNotifications()*** to fetch the current notifications for the logged in user. The message type of the notification defines the template used for rendering this notification. Each template is stored in the ***webapp/templates/notifications*** folder of the Webprovider. Via the websocket connection the notification plugin directly forward arrived notifications to the frontend.

Implications and problems:

Now we have a solid way to inform the loggedin user or even other users about events, which needs the attention of the user. Moreover, it is done in a not disturbing way, like popups, as it was implemented before. Furthermore, we have a central way to inform the user, which also handles the messaging and storing of this information.

5.5.2. Messaging Plugin (CH)

Short overview:

The messaging plugin is comparable with an email application. It consist out of an in- and outbox. It is possible to send and receive messages and it is also possible to organize messages by assigning labels to them.

Adaptations to LifeSocial:

GUI	**Messages:** own view in frontend. **Notification:** displays notification when a new message arrives
Plugins	**MessagingPlugin:** ● load and stores in- and outbox ● handle messages, labels and badges ● registered to NotifcationPlugin **MessagingPluginHandler:** ● registered to RESTHandler
Framework	
Overlay	

Detailed description:

With the messaging plugin it is possible to be in touch with users inside the network. It is possible to send and receive messages. As it can be seen in the figure 5.5.2, the plugin consists of an in- and an outbox. Both are positioned at the left side of the plugin. Additionally, it is possible to

create new Labels with a hierarchical structure. This helps the user to organize her/his messages by assigning these labels to a message. It is also possible to assign multiple labels to a message. Moreover, we added a badge behind the labels or the inbox, which indicates how many messages are not seen yet.

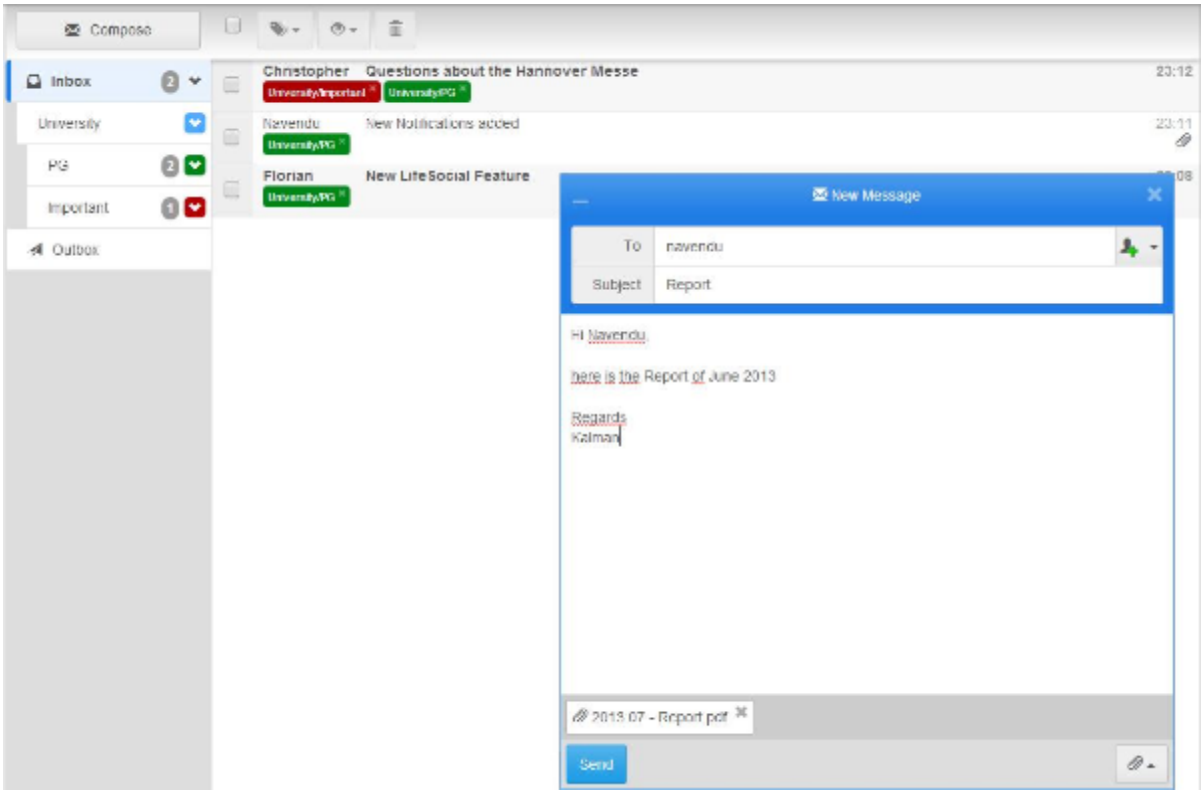


Figure 5.5.2: Messaging Plugin GUI

When the user wants to write a message s/he needs to click on the compose button and a new modal window opens. This window is integrated in the browser tab and is always on top. It is possible to move this window around and even when the user switch between the plugin the window stays open. Moreover, the window can be minimized or closed to reduce the disturbance. Inside of this window the user needs to select one or multiple users as a target. These users needs to be separated by a comma. Additionally, the message requires a subject and an optional text. Moreover, it is possible to share files via messages. These attachments are shown at the bottom of the window. It is possible to pick files from the already uploaded list or to upload a new file. After that the user can send the message by clicking on the send button.

For sending a message we use a ***SecureMessage*** to protect the sended data. To prevent the loss of messages we use the same strategy as in the notification plugin. So the sender first tries to contact the target directly and only when it fails for the second time the sender stores the message in the ***DistributedLinkedList*** of the target. Otherwise the target is responsible for the storing. The only difference here is, that the target reads out the ***DistributedLinkedList*** after logging in and remove the messages from this list. After that the target stores new messages in

an inbox item. The outbox is stored in the same way but in a separate item. Both items are encrypted with the public key of the target so s/he is the only user who can read them. To achieve that we use the ***SecureSharedItem*** which is able to encrypt every ***SharedItem*** with public key encryption. Although the terminus “shared” does not fit anymore it was very helpful and we did not need to change much of the previous implementation.

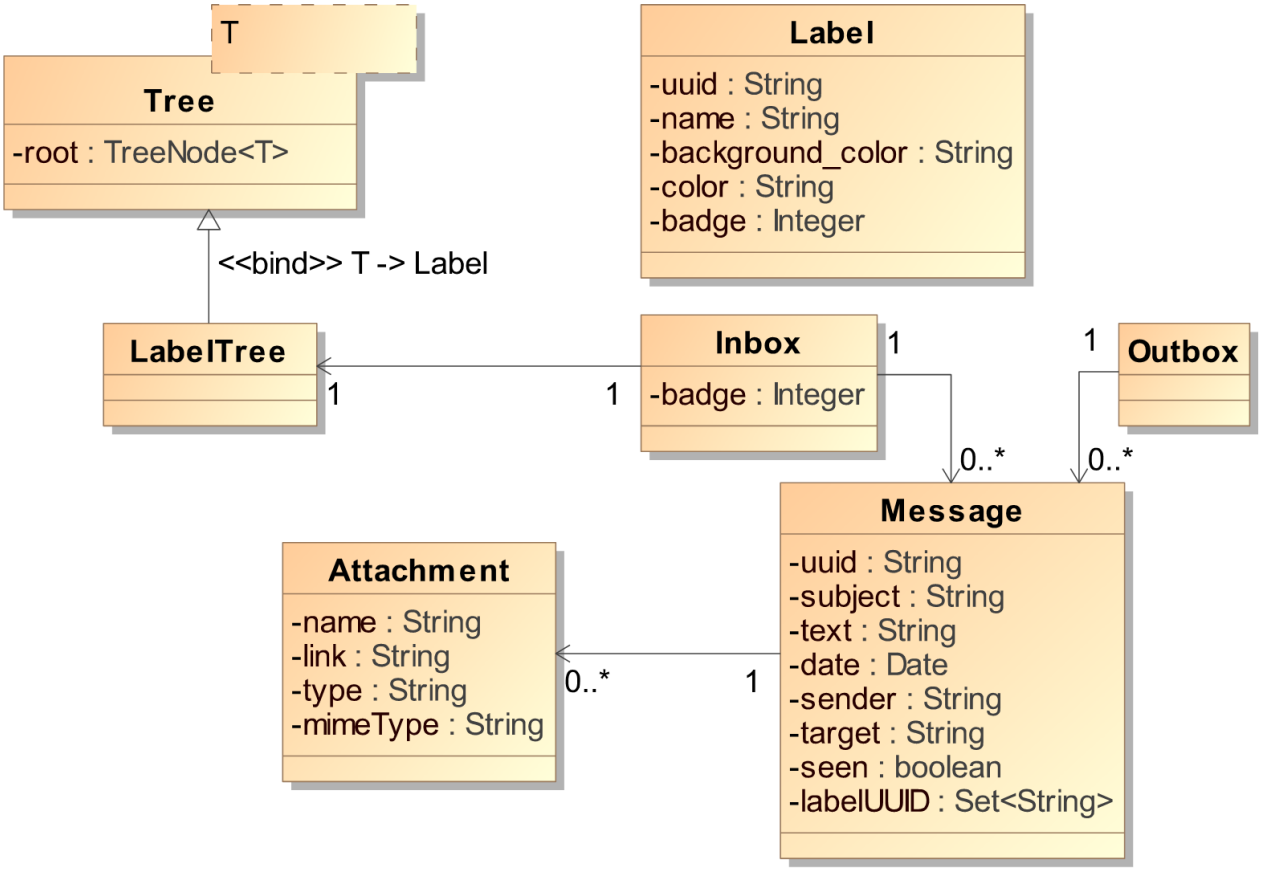


Figure 5.5.3: Messaging Plugin data structure

As already told it is possible to organize messages via labels. Therefore, we use a Tree to allow a hierarchical structure of labels as it can be seen in the figure 5.5.3. The root node is a ***TreeNode*** with content null. This represents the inbox itself. This way every message has by default no label and is automatically in the inbox. To assign labels to a message the message contains a set of UUIDs of the assigned labels. A user is also able to define a name, a font color and the background color for the label. Each property can be changed at any time.

To help the user to distinguish between messages s/he has already read, we mark this messages as already seen and remove the highlighting in the frontend. For the attachments we do not send the whole file but instead we send a reference to this file. This reference contains the necessary information to download the file. Besides that, we send some other information like the filename, the type and the mime type which is helpful for the graphical representation. We also implemented a notification, which is shown when a new message arrives. This notification can

be used to open the arrived message directly.

Implications and problems:
Via this plugin we provide the user the possibility to send messages to other users without the need that the communication participant needs to be online at the same time.

5.5.3. Wall Plugin (FS)

Short overview:
The wall plugin allows the user to post entries on her or his personal page, that others can comment. Thus it provides a simple form of one-to-many communication. In this year it was extended so works more similar to other social networks.

Adaptations to LifeSocial:

GUI	Wall
Plugins	**WallPlugin:** ● handles sending and storing of wall entries ● supports privacy via ACL encrypted items ● wall entries can be pushed to the wall of the friends **WallPluginHandler:** ● registered to RESTHandler
Framework	
Overlay	

Detailed description:
Using the wall plugin users can publish status messages to the network. These posts can be commented by the author or other users. Initially, every user had an own wall with entries that all the other members of the network could see when they were accessing the users wall.

In this years project group, the behaviour of the wall was changed in two respects.

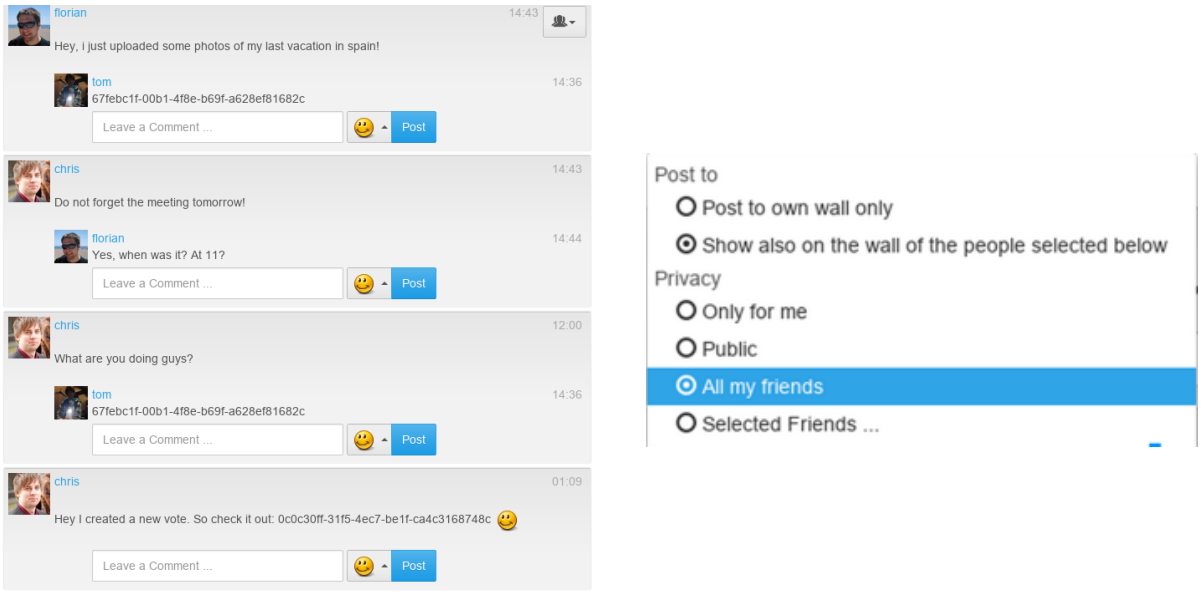


Figure 5.5.4: Privacy settings for wall posts

First, we introduced the ability to encrypt the posts in the wall so the user has the control who can see the posts on his/her wall, as in figure 5.5.4. This way the user can specify for each individual post, who is able to see the post. There exist 4 different types of visibility for this privacy setting. The user can either specify that s/he is the only one who can see post. This is useful if s/he wants to write something before s/he actually makes it visible to other users. Additionally it is possible to either make the post visible to all or only a subset of his or her friends. Finally it is still possible to make the wall post public by not encrypting the wall post at all.

To achieve this flexibility, the items from the wall are stored using an access control list. This functionality is provided by the framework and uses a combination of symmetric and asymmetric encryption. The actual wall post is encrypted using a randomly generated symmetric key. This key is then encrypted for each user that is allowed to see the contents of the wall entry with his/her individual asymmetric key. If a user accesses the encrypted wall item, s/he tries to retrieve the symmetric key using his/her private key. If this succeeds, the actual data can be decrypted. If some user comments on a wall entry the symmetric key of the post is used to encrypt the comment data. So the comments always have the same visibility as the wall post.

One problem that had to be considered was that the list of friends is variable. So it is possible that new friends are added or that friendships are withdrawn. In this cases the post that are made visible to all or specific friends are re-encrypted. For wall entries that are made visible to all friends this is true for every change in the list of friends. When a new friend is added he/she is just added to access control list and the symmetric key is also encrypted using his/her public key. If a user is removed from the friends list that had access to a wall post before, the whole item is encrypted with a new symmetric key, so that the user can not access the data anymore. Of course the user could have made a copy of the wall entry before but this cannot be prevented. The drawback of this is that all wall entries that are made visible to friends have to be processed, which leads to additional cpu activity and network load.

Second, we changed the form of the wall. Wall entries are not necessarily visible on the wall of the corresponding user only but optionally also on the walls of the persons that were selected in the privacy settings. This way it is not needed to visit every friends wall anymore to see what happens in the circle of friends. Instead it is sufficient for a user to have a look at the own wall and see all the entries, the friends made visible to him/her. So the wall can be considered as some kind of news stream. Of course it is still possible to directly access the wall of another user. In this case only the wall entries are shown to the user where s/he has access to, because he is not able to decrypt any other entries which are therefore skipped.

Technically the wall items are stored in a distributed link list. This is an entity which holds multiple items and is splitted every n entries and distributed in the network. In the original version a 2-dimensional list for each user was maintained. In the first dimension of this list the wall posts were stored and for each of this items a second list was used to store the comments.

For the new implementation this concept was adapted a bit as it is presented in the figure 5.5.5. The wall entries are stored as individually encrypted items in the network using a storage id. The first list was changed so that it does not store the actual wall posts anymore but the storage id as a reference to the entries. If a post is published for all or selected friends of the author, the reference is also added to their list of wall entries. This ensures that the wall post can be changed afterwards without the need to change multiple copies of the wall entries in the friends walls. Although this means that the users who were removed from the list of privileged users, of a wall entry, still have the reference entries in their wall list. When a user is not able to read an item s/he skips that item. As this is not very likely to happen the list should not be bloated up too much. Possibly some kind of garbage collection can be implemented here.

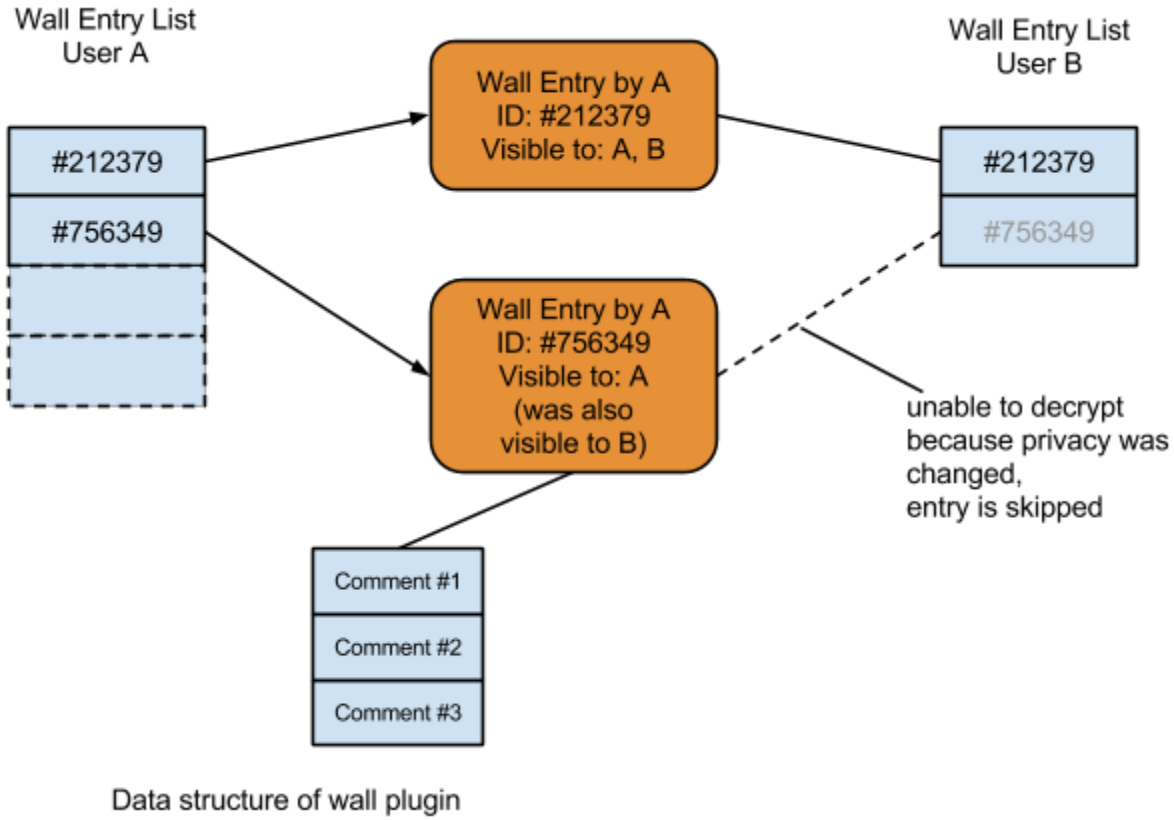


Figure 5.5.5: Data Structure of Wall Plugin

One further thing to mention is how the graphical user interface displays the entries. When the wall plugin is loaded, only the 10 latest entries are retrieved from the network to reduce the loading time. If the user wants to see older entries, he has to scroll down and click the corresponding button. For this to work the backend always has to send an offset value which holds the id of the oldest id that is currently displayed. This is then used when the user clicks the button to retrieve the next wall entries. We also considered to modify the behaviour for the user so that the browser notices that the user scrolled to the end of the list and automatically initiates the retrieval of further entries. The disadvantage with this approach is that the user is losing control and cannot reach the lower parts of the website, especially the footer. This is why we did not used that approach.

5.5.4. Chat Plugin (CH)

Short overview:

With the chat plugin LifeSocial provides the functionality that users can communicate with each other. The use of emoticons, url parser and attachments make the chat more powerful. Besides

a simple one-to-one communication, we also included a many-to-many communication.

Adaptations to LifeSocial:

GUI	Chat
Plugins	**LiveChatPlugin:** ● handles sending and storing of chat messages ● supports one-to-one and many-to-many chats ● registered to NotificationPlugin **MultiChatPluginHandler:** ● registered to RESTHandler
Framework	**TopicChannel:** ● added symmetric encryption support
Overlay	

Detailed description:

The logged in user can contact a friend by clicking on the corresponding username in the sidebar. There exist a button to open the chat window. This window provides the same functionality as the window from the chat plugin. So it is possible to move, minimize or close the window. The sanpshot of chat window can be seen in figure 5.5.6.

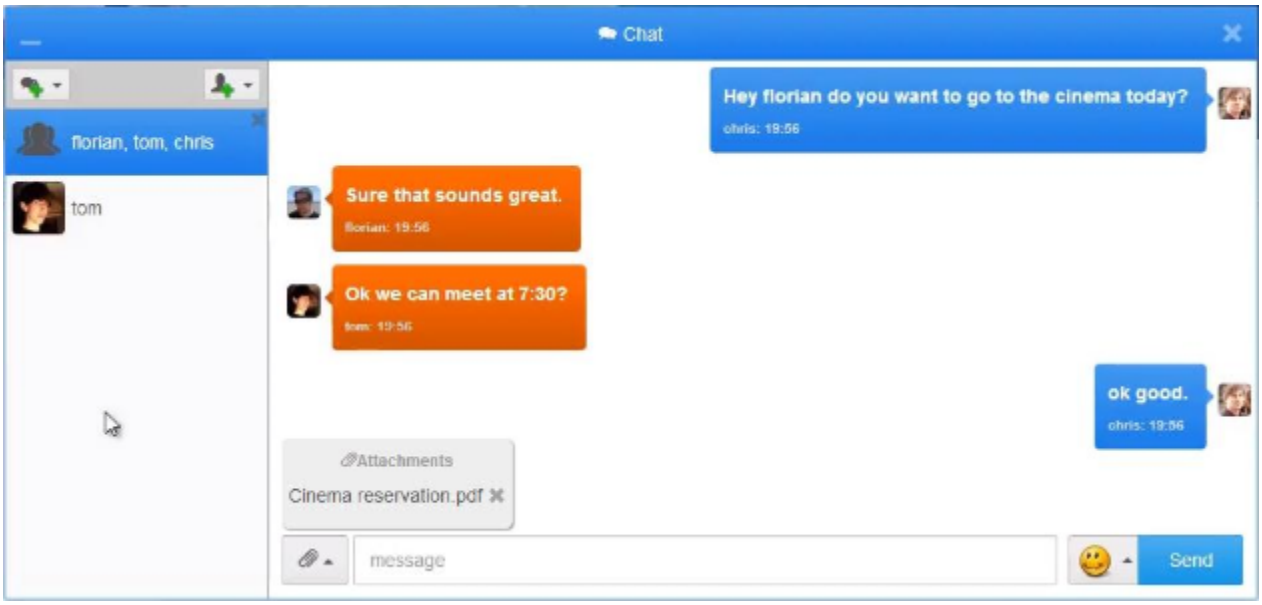


Figure 5.5.6: Chat Plugin GUI

On the left side of this window is an overview of active chats. There are two different kinds of chats. A one-to-one chat which is represented by the username and profile picture of the communication partner and a many-to-many chat, which is represented by a group icon and the names of the participants. The chat messages are shown on the right side of the chat window. Additionally, all previous messages are shown as well. To make the loading much faster only twenty messages are shown and only loaded when the user scrolls to the top of the window. Messages from the logged in user are always at the right sides of the section. Messages from other users are at the left side. The controls to type and send messages are at the bottom. It is also possible to add attachments or emoticons to messages. At the top left corner of the window it is possible to create a new many-to-many chat or to add user to the selected chat.

To protect the data of the users we encrypted both chat types. For the one-to-one communication we decided to use public key encryption by using the ***SecureMessage*** which is used when we send messages over the ***MessageDisptacher***. For the many-to-many chat we use a ***TopicChannel*** from our Framework. Each channel has its own unique identifier. In our case we use a random UUID to pick such an id. Because the ***TopicChannel*** was not encrypted before, we extended the functionality to define a symmetric key for such a channel. This means that every message sent over this channel is automatically encrypted. To exchange this symmetric key we use a notification, so that a user needs an invitation from members of the chat s/he wants to join.

The history of both chat types is stored in an ***SecureSharedItem***, which uses the public key of the logged in user to encrypt a ***SharedItem***. The symmetric key is also stored in this history so it is possible to subscribe the chat direct after the login. If a user leaves a many-to-many chat, the history is deleted so s/he cannot subscribe this channel anymore.

Implications and problems:

Because of the new implementation LifeSocial now provides a secure way for one-to-one and many-to-many communication. Also the possibility to share files over the chat makes this plugin a powerful communication tool.

5.5.5. Audio / Video Chat Plugin (CH)

Short overview:

With the Audio / Video Chat Plugin we implemented a completely new plugin. It provides a one-to-one audio and video communication. Therefore, we use the open standard WebRTC.

Adaptations to LifeSocial:

GUI	**Chat:** HTML5 Video Element inside a movable and resizable window
Plugins	**VideoChatHandler:** ● registered to RESTHandler ● registered to NotificationPlugin **WebRTCNotificationSocketHandler:** ● handles signalling messages between sender and target
Framework	
Overlay	

Detailed description:

In the new GUI of LifeSocial it is now possible to call a friend by selecting her/him from the sidebar and use the quick access button in the opened popover. After that the target user gets a notification where s/he can decide if s/he want to accept or decline the call. When the call is accepted the communication is initialized and the audio / video chat begins. Therefore, a movable and resizable window opens as shown in the figure 5.5.7. In the bottom right corner of the window the logged in user can see her-/himself. The video is also mirrored which makes it a lot easier for the user to position her-/himself in front of the camera. The video of the communication partner is shown in the middle and take the most space of the window. This time the video is not mirrored. Via some control elements at the top of the window it is possible to toggle the fullscreen mode, the audio and the video. Via the hangup icon or the close button of the window it is possible to cancel the call.

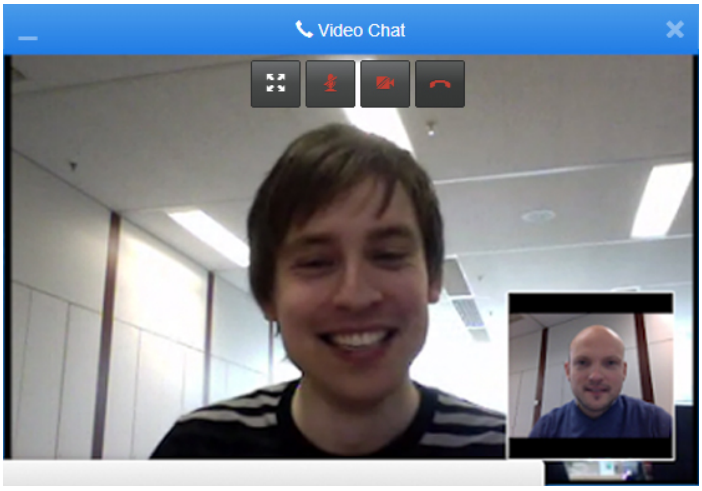


Figure 5.5.7: Video Chat Plugin GUI

Instead of handling with different video and audio codecs and the support for different cameras we decided to use the open standard WebRTC. WebRTC allows to access the camera inside the browser. But it provides even more. It is able to traverse different NAT types and establish its

own 128-bit AES encrypted connection. But before this is possible we need to provide two other functionalities. One is a STUN server, which is required to identify the NAT type and the external address. Sadly it is not possible to tell WebRTC the address and the type. We started to implement the STUN protocol but due to the lack of time we decided to use a free Google STUN server instead. It is not that bad, because the only transmitted data here is the external ip itself after that WebRTC established its own connection to transmit the video and audio data as it can be seen in figure 5.5.8. The other functionality we need to provide is a server which handle the signalling messages of WebRTC. This messages can contain information like the supported codecs of each communication partner. In our case we use our Websocket connection to forward the WebRTC messages to our backend. Then the ***WebRTCNotificationSocketHandler*** is responsible for forwarding the messages to the target user. Therefore, we use the ***SecureMessage*** to protect these signalling messages. When the target receives such a message the handler forward this message via Websocket to the frontend.

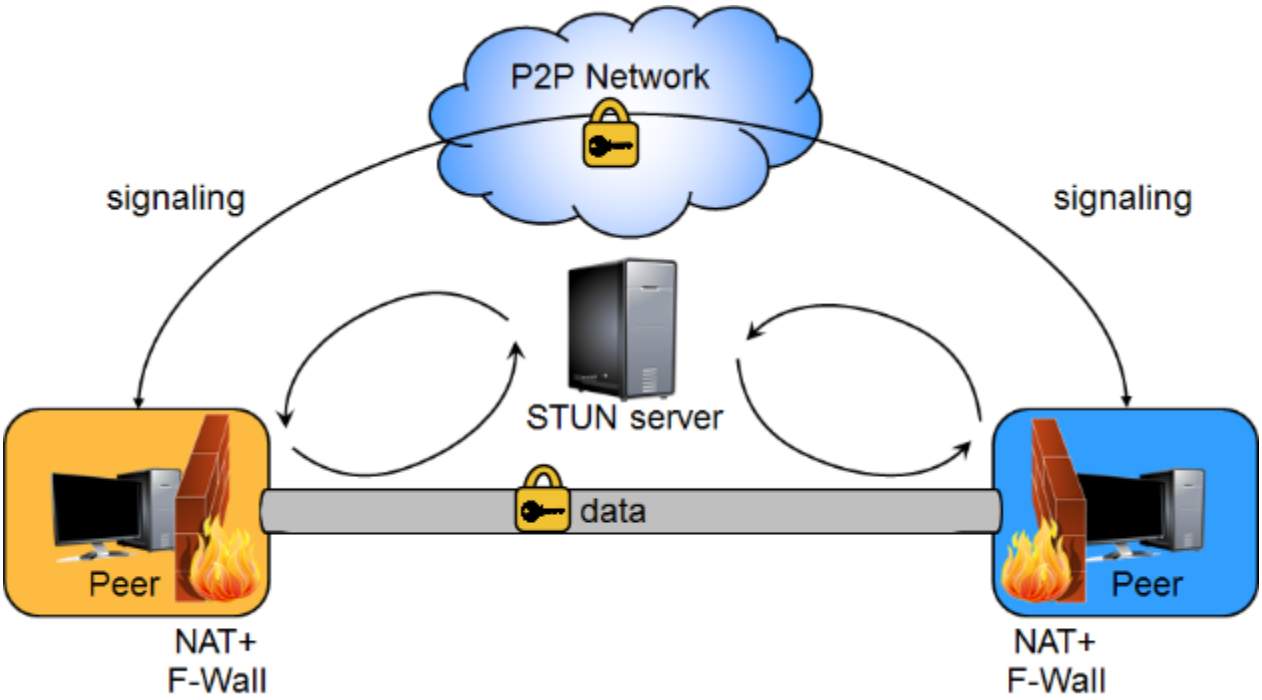


Figure 5.5.8: Video Chat Plugin - WebRTC integration

Besides that we use the ***WebRTCNotificationSocketHandler*** for our own signalling messages. An example for this is when a user hangup. Before the a user leaves s/he sends a signalling message to the other participant and informs her/him that s/he has left the chat. The target then can react on such events. Other events are for example the enabling or disabling of the audio or video stream. The ***VideoChatHandler*** is used to handle the requests like a call someone and accept or decline a call. Therefore, the handler uses the notification plugin to inform the other participant.

Implications and problems:

WebRTC made is quite easy to implement a audio and video chat. The only thing we need to provide is to forward the signalling messages correctly. To be fully independent it would be necessary to implement our own STUN server. It would be possible that such a server is started at every public node in the network.

5.5.6 Photos (SK)

Short overview:

The photos plugin provides the sharing of photos and albums with other users. Users can comment on photos of other users as well as on their own photos. The owner of an album can chose the audience with whom s/he wants to share their albums, by selecting the corresponding privacy settings.

Adaptations to LifeSocial:

GUI	Photos, Photobox, Friends Selector HTML GUI
Plugins	**PhotosPlugin** ● Create Album ● Load Album ● Edit Album ● Add Photo ● Edit Photo ● Delete Album ● Remove Photo ● Store Photos **PhotPluginHandler** ● Registered to RESTHandler
Framework	
Overlay	

Detailed description:

Albums are identified by the unique album key assigned to them. The same key would be used while adding a photograph to the existing album or deleting a photograph from the same. The StorageKey assigned to an album is stored in a list, which is public accessible.

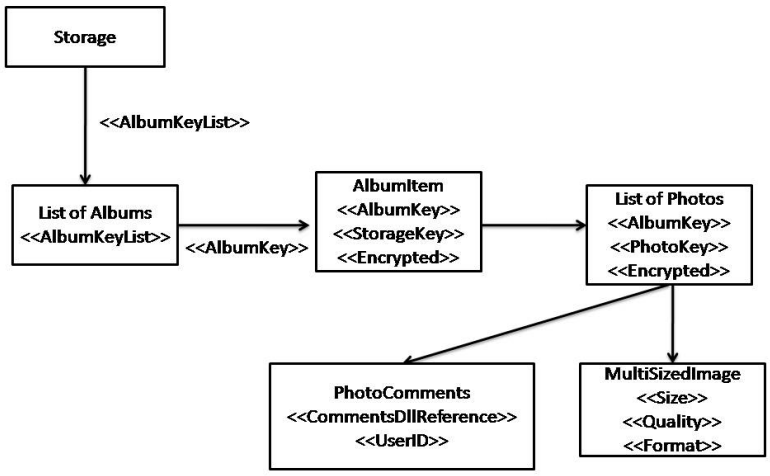


Figure 5.5.9: Storage of albums and photos

Albums are identified uniquely by the album key and can be retrieved by the storage key. Album Items which are basically generic items converted into the secure storage item. An album also contains a list of sorted photos. These photos contains a unique photokey, an albumkey the photo is assigned to, a caption, comments and a ***MultiSizeLSImage***.

Photo comments:

Are stored as a reference inside of an photo. All comments of a photo can be retrieved by using these reference. A ***DistributedLinkedList*** is used to store the comments.

MultiSizeLSImage:

Used to store the images in different formats, qualities and sizes. The ***MultiSizeLSImage*** contains a list of references to the real photo data of the different versions. When an image is requested it automatically decides which version of the photo fits best to the requested one. Therefore, the request can contains the format and size of a photo. As a result the MultiSizeLSImage returns a StorageKey under which the photo version is accessible. The different versions of a photo can be defined at the beginning and each version is then automatically created when a photo is stored. Therefore, the MultiSizeLSImage does not store the versions but returns them as a SharedItem or a StorageItem plus the StorageKey. The storage needs to be handled by the plugin, which uses the MultiSizeLSImage.

Privacy settings:

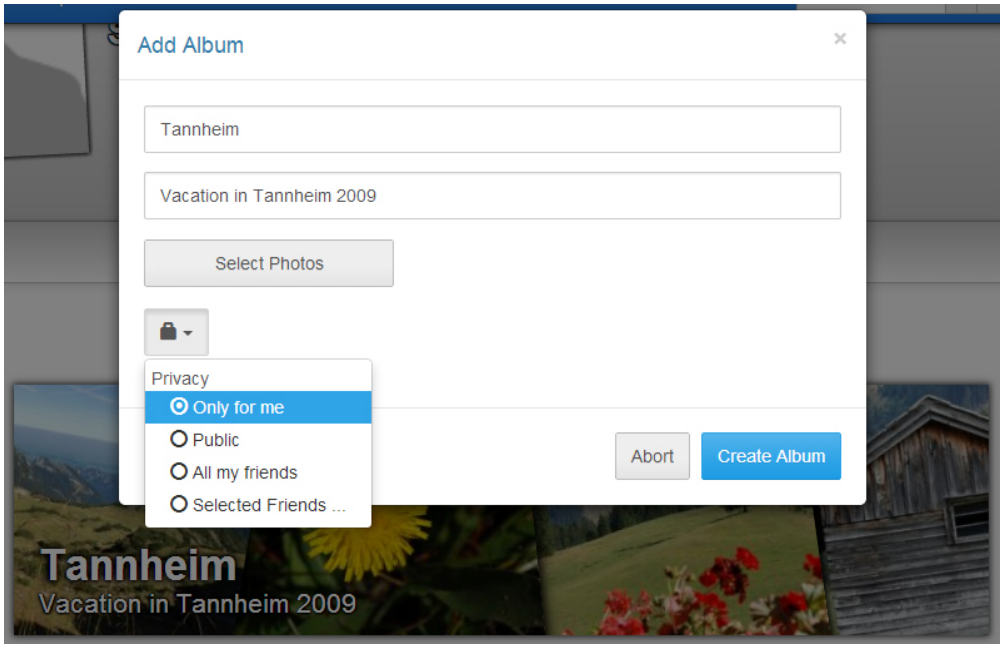


Figure 5.5.10: Create album GUI & Privacy settings

All photos are encrypted by a symmetric key, which is stored in the corresponding album. An album is encrypted with a symmetric key which is accessible via an Access Control List, which is included in the SecureStorageItem. As you can see in the figure 5.5.10, users can choose different privacy setting while uploading a new album and they can also edit the same afterwards. Below is the detailed description of each privacy setting.

Only for me:

In this privacy setting the album is stored as a SecureStorageItem and the randomly generated symmetric key for this album is not shared with other users. Therefore, the owner is the only person which is able to decrypt the album. This is, only the owner can access the symmetric key for the photos.

Public:

The album is stored as a GenericStorageItem. Therefore, every person in the network is able to read that item and access the symmetric key for the photos.

All my friends:

The album is stored as a SecureStorageItem. The symmetric key of the album is encrypted with the public key of her/his friends. The user her/himself and her/his friends are the only people in the network, who have access to the symmetric key for the album and are therefore the only people in the network, who can decrypt this album, by using their private key.

Selected friends:
This setting works same as above but the only difference is that only selected friends are added to the access control list.

Different functions have been used in order to perform different activities for this plugin. Below, you will find the detailed description of all the functions.

1. createNewAlbum(String title, String description, String[] privilegedUsers, int privacyType):

This function basically creates an album based on the input parameters given by the user as you can see in the figure 5.5.10. This function will generate a random album key and create a an album item(***addAlbum(String albumKey)***), which is a ***GenericStorageItem***. In the case of applying privacy settings, the album is encrypted with a symmetric cryptographic key, which can be made available to the authorized users(basically the users in the access control list). The symmetric key is encrypted with the user's public key and can be decrypted by using the user's private key. To encrypt an item we use the ***SecureStorageItem***, which contains the ***GenericStorageItem*** and an ***AccessControlList***. After, we store the ***GenericStorageItem*** or the ***SecureStorageItem*** in the framework, we also add the album to the Album list.

2. editAlbum(String albumKey, String title, String description, String[] privilegedUsers, int privacyType):

This function is used to edit the album privacy settings, album description, title and retrieve the targeted album by using the method ***getAlbumFromStorageItem(StorageItem retrieveItem, String albumKey, String username)*** and updates all the input parameters.

3. addPhotos(String albumKey, List<File> photofiles, String title, String viewId):
This function will add the photos to the targeted album on basis of the specified album key. The photos are added to the album and existing the privacy settings of the album are considered while the photos are stored.

4. deleteAlbum(String albumKey):
This function is used to delete the album. First it will look for the album key from the ***AlbumListPayload*** of a target album which is going to be deleted. Then, it will remove the album key from the ***AlbumListPayload*** by using the method ***removeAlbum(String albumKey)*** and delete all the photos which are belong to that album by using ***deletePhoto(Album album, String photoKey)*.**

5. deletePhoto(Album album, String photoKey):

This function will delete the targeted photo belong to the same key as given in the input parameters by using the method ***removePhoto(String photoKey)*** And it will overwrite the items with an EmptySharedItem.

Implications and problems:

Eventhough photo plugin works correctly, it is not consistent. The problems which has been identified till now are described as follows.

- No visual feedback from the back end side in case of delays during loading and displaying an album.
- Slide show does not work properly due to problem with buffering and loading the images. Photos are retrieved from the network and it is usually a little bit slow procedure and it takes some time to load the photos while slide show has 2-3 secs of clock in order to move to the next picture so it would not be a great idea to increase timer of slide show instead we could have found some technique to store the photos into the cash once they are loaded completely. So that they won't take lots of time while being loaded for the second time. Cash also has to be deleted after some time.
- Move photo from one album to another would be one of the needed missing feature.

5.5.7. Groups (RS, MV)

Short overview:

The groups plugin models a community environment, where a group of users are provided with a shared or common drive and a forum for discussions among the set of users.

The members are allowed to add their friends to any group that they are part of. Whenever any friends of the user is added to the group, the friends are notified that they are added to the group. Any user who is a part of the group can leave the group any time the user wants.

The shared drive provided can be used by the group members to create folders and upload files, so that they can be shared among the members in the group. The forum allows the members of the group to create threads and post their comments, answers or questions in the threads created.

Adaptations to LifeSocial:

GUI	**Groups** ● Group list and groups management

	Group drive ● To display the shared drive contents **Group forum** ● The forum for the group
Plugins	**Groups** ● For group management and shared drive **Forum** ● For discussions on topics in the group **Files** ● For file upload and download in the group **GroupsPluginHandler** ● registered with RESTHandler **ForumPluginHandler** ● registered with RESTHandler
Framework	
Overlay	

Detailed description:

Group:

Every group is identified uniquely with the group id, which is a 128-bit universally unique identifier (UUID). When user creates a group, group id is generated and group gets added to the Groups list of the user. Each member of the group is identified by a unique userID, provided by the framework. When a user adds his friends to the group, the userIDs of the friends are added to the groups user list. Every group members can add their friends to the group. The users who are added, gets notification indicating, they are added to the group. Any user in the group can leave the group when the user wishes to. The figure 5.5.11 shows the data structures used and how the group is used as reference to store the shared drive, group forum and the list of users in the group.

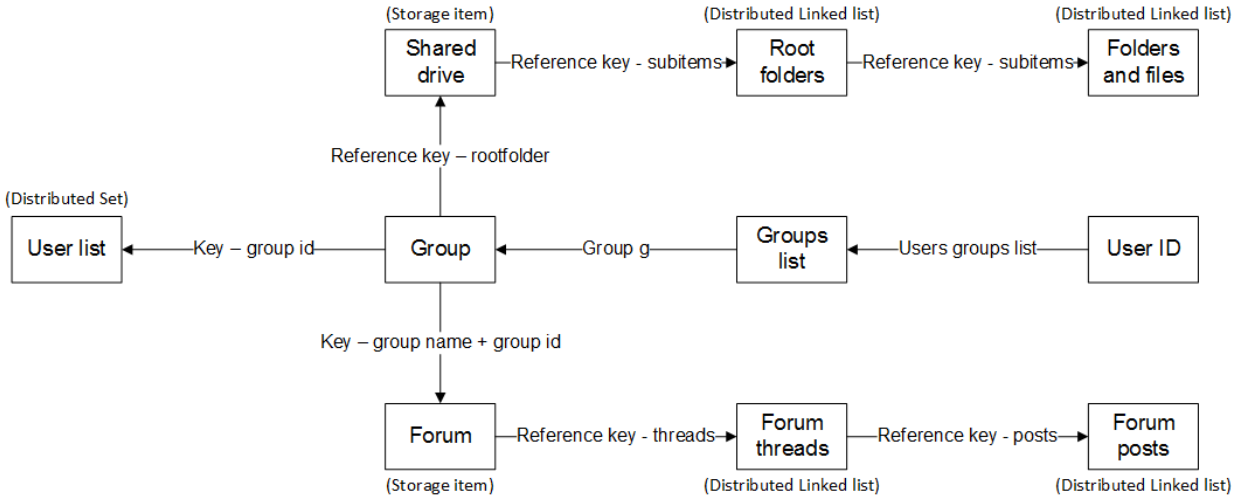


Figure 5.5.11 The data structures used and mapping of storage items in groups

The members in a group (User list) are stored using a ***DistributedSet*** as shown in the Figure 5.5.11. In order to retrieve the user list of any group, group id of the particular group is used as the key to fetch the list. This is achieved by instantiating the ***DistributedSet*** with the corresponding identifier for the group. To get the identifier for any group, the method ***getGroupUserListStorageKey(Group group)*** returns the identifier, which is used to retrieve the list of members in the corresponding group. When any user leaves the group, the user ID of that particular user is removed from the user list. Similarly when any user in the group leaves, the group list of the user is updated by removing the group from the user to group mapping and the users group list.

Groups provide a shared drive, organized in the form of folders and files where all the members can view the contents of the folders, create folders and upload/download files. The structure of the shared drive is a tree hierarchy with root folders in the top level, files and folders in the levels below. Figure 5.5.12 shows the graphical structure of the shared drive.

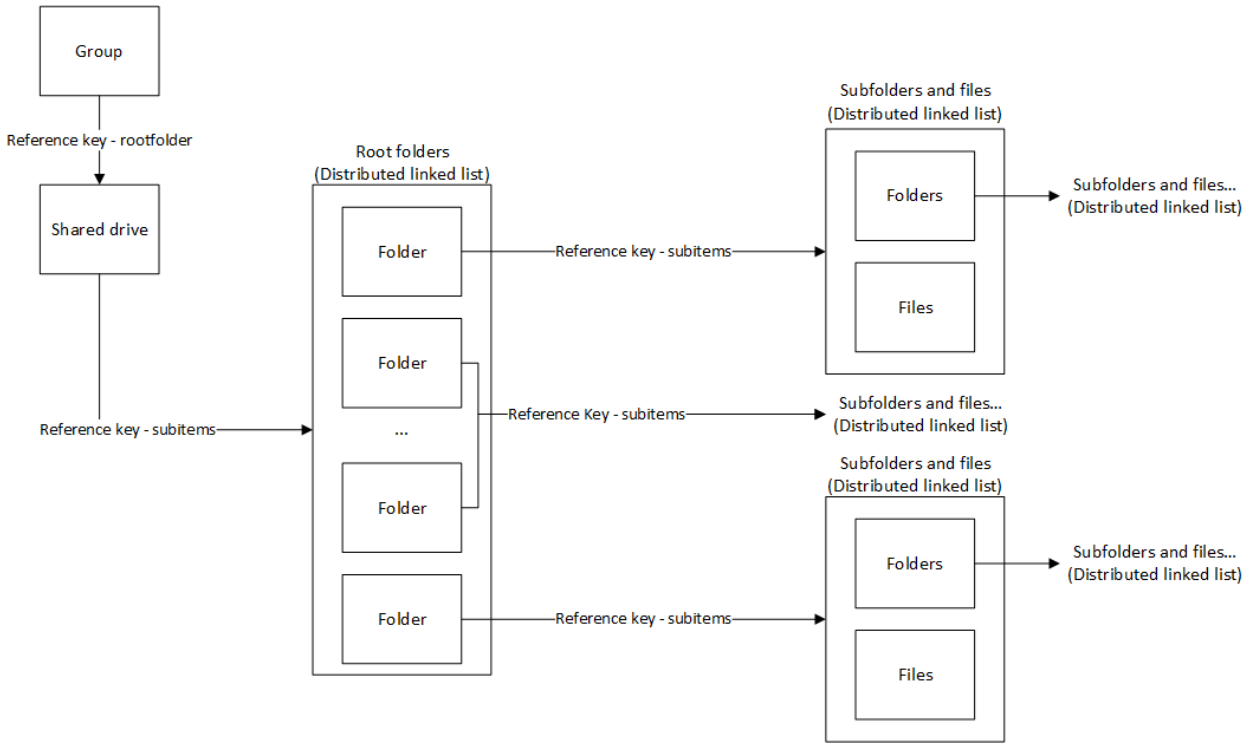


Figure 5.5.12: The file system layout used in groups shared drive

The shared drive is accessed using reference key “rootfolder” from the group. The reference key ensures that, the shared drive cannot be accessed by any node or users without having access to the corresponding group. The shared drive is created when the group is created. The shared drive is a ***Folder,*** that is stored using the reference key “rootfolder”. The parent of this folder is set to ***null.*** To retrieve the shared drive for the group, asynchronous function ***getRootFolder(final AbstractValueCallback<Folder> callback)*** is used, which loads the item using the method ***getOneReference(String key)*** provided by ***AbstractStorageItem*** class, with key as “rootfolder”.

The files are stored using the file plugin and reference of the file is added to the ***DistributedLinkedList***, which stores the contents for the corresponding folder. The contents of each folder is accessed using the reference key “subitems” as shown in the figure 5.5.12. The shared drive provides functionalities for adding folders, files, renaming folder, deleting files and folders. ***DistributedLinkedList*** implements item signing, which is used to enforce basic access controls on the files and folders in the shared drive. This only allows the owner of the file or the folder to delete or rename the item in the shared drive.

Each folder or file is associated with unique identifier, the UUID. To retrieve a folder or a file from the shared drive, a backtracking method is implemented. In this method, each folder or file stores a reference to its parent. The parent folder is added to a list, path list, which is populated until we reach the shared drive, which is stored using the reference key “rootfolder”. Using this, a path list is created with the starting element as the shared drive. Then, using the reference key

“rootfolder”, the shared drive is accessed and using the path list we traverse through ***DistributedLinkedList,*** stored using the reference key “subitems” as shown in the figure 5.5.12. For efficient traversal, BFS (Breadth First Search) technique is used. The function ***searchFolder(final Folder folder)*** implements the BFS technique to retrieve the folder from the shared drive. Once the target folder is retrieved, the shared drive operations such as renaming folders, deleting and adding folders or files can be performed.

To rename a folder, delete folders or files we set the target folder to be the parent folder. For adding files or folder, we use the current folder as the target folder. Using the function ***searchFolder(final Folder folder),*** we find the target folder in the shared drive. Using the target folder method ***getSubitems(),*** which internally calls the method ***loadManyDDSReference(String refName)*** provided by ***AbstractStorageItem*** class, with ***refName*** as “subitems”, the contents of the folder is retrieved. This method returns the contents of the folder as ***DistributedLinkedList.*** The standard list interface operations such as ***add(Object object), remove(Object object)*** are used on the ***DistributedLinkedList*** to perform rename, addition or delete operations of folders and files in the drive. An explicit save command is not issued from the plugins to the operations provided by any distributed data structures, since these operations save and synchronize the data in the network automatically.

Shared drive operations:

1. Add folder:

We load the target folder using ***searchFolder(final Folder folder).*** To add folder we then use target folder method ***getSubitems()*** and use the list interface method ***add(Object object)*** to add the folder into the ***DistributedLinkedList***.

2. Add files:

The Files plugin provides the method ***storeFile(File file, String storageFileName)*** which returns ***FileInfo*** for the file uploaded. ***FileInfo*** contains the id and the key, which are used to access and download the file. We store this ***FileInfo*** in the ***File*** class. The ***File*** class is then added to the ***DistributedLinkedList*** using the list Interface method ***add(Object object).***

3. Rename Folder:

To rename a folder, we remove the item using the list interface method ***remove(Object object)*** and add the item back into the ***DistributedLinkedList*** using the list Interface method ***add(Object object).*** The name of the item is set using ***setName(String name).***

4. Delete File and Folder:

To delete a file or folder, we use the method ***deleteItem(final Item i).*** We load the item from the shared drive using the method ***searchItem(final Folder parentFolder,Item current),*** which internally calls ***searchFolder(final Folder folder),*** with folder as ***parentFolder.*** Once the item is

retrieved, it is deleted from the drive using the asynchronous method ***deleteCascaded(AbstractVoidCallback cb)***, provided by ***AbstractStorageItem*** class.

In case of files, there does not exists any dependencies. So the method only removes the file from the storage and from the ***DistributedLinkedList.*** Deleting the file from the shared drive does not remove the file from the file storage plugin. However, in case of a folder, a cascaded delete operation is executed, wherein all the dependencies of the folder are removed from the storage.

HTML GUI for group:

Group lists and management:

This GUI implementation allows the user to manage the groups that user is a member of. The GUI allows user to create a public group, to add his friends to any group, where the user is a member and also to leave any group, the user is a member of. The common friends selector template is used to display the list of friends of the user, when the user wishes to add his or her friends to any group. Default view loads the users group list, retrieved using the function ***List<Group> getGroupsList(),*** which uses the userID to fetch the group lists as shown in the figure 5.5.11.

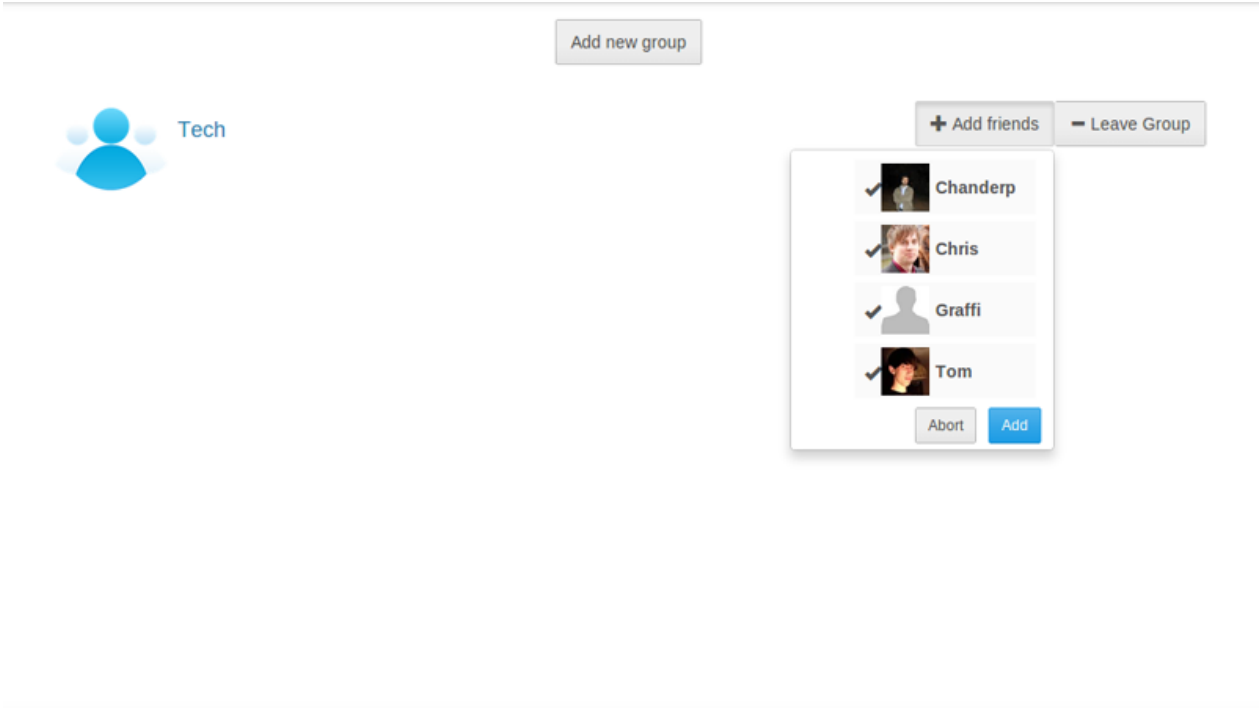


Figure 5.5.13: The HTML GUI for Group lists and groups management

Group shared drive:

The shared drive contents are loaded on demand. The entire shared drive is not loaded, instead only contents of the folders are loaded. The function ***getcontents(String groupID, String folderID)*** returns ***GroupContentDescription,*** which is a class that contains information that is used to display the contents to the browser. The ***GroupContentDescription,*** includes information on the group, the members in the group, the contents of the folder and the path to the folder or the breadcrumb information. This information is parsed and displayed in the browser. The request format is specified as ***#groupcontents/{UUID}/{UUID},*** where the first UUID is used to identify the group and second to identify the folder in the shared drive.

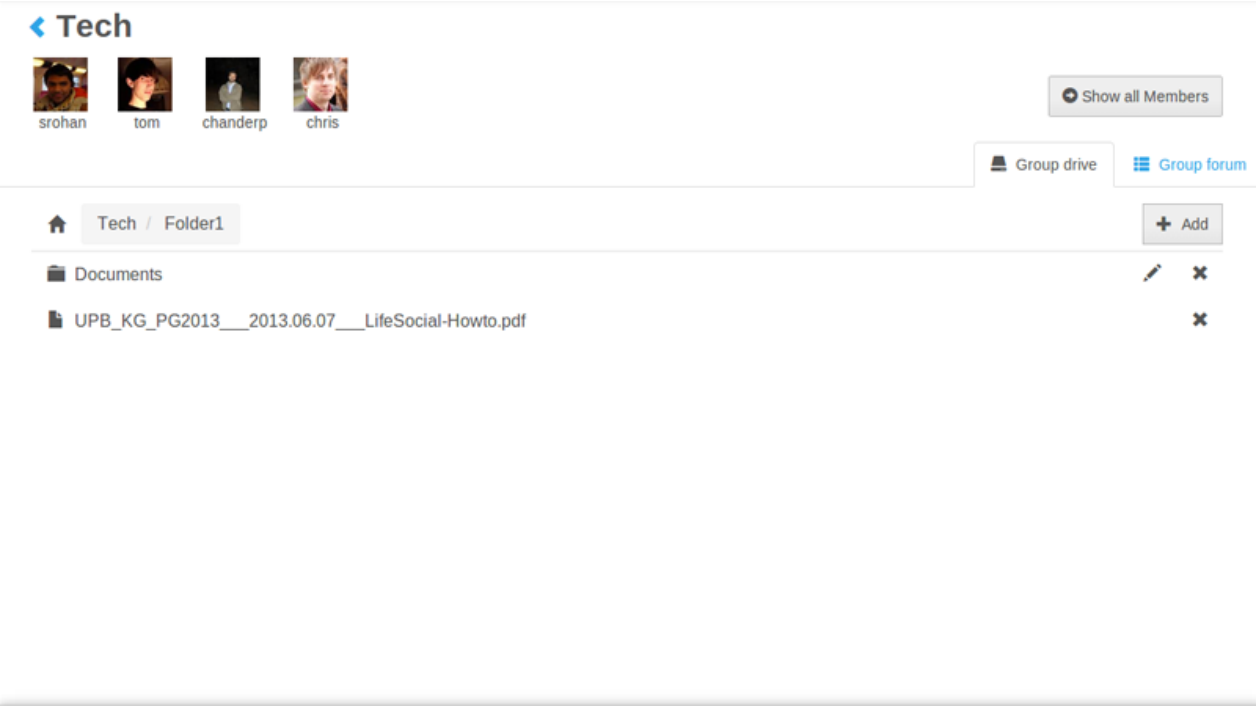


Figure 5.5.14: The HTML GUI for the group shared drive

Forum:

The forum for the group is created upon the first request by any group member. The forum is stored as a ***GenericStorageItem,*** with key combination of the group id and the group name. For each group only one forum is provided, only one-to-one mapping is allowed. The previous implementation of having forums as files inside the folder in the shared drive was removed.

The forum allows the members of the group to have discussions on topics. Forum threads created by the users models the topic on which any member wishes to have a discussion. The threads are stored as ***DistributedLinkedList***, which can only be retrieved from the forum using the reference key “threads” as shown in the figure 5.5.11. Forum posts are comments or answers or questions that members post in reference to the forum threads. Forum posts are also stored using ***DistributedLinkedList*** for each forum thread. The forum post can only be retrieved from the forum using the reference key “threads”, then by using the reference key

“posts” as shown in the figure 5.5.11.

HTML GUI for forum:

Group Forum:

Similar to the shared drive, the forum contents are also loaded on demand. The function ***getForum(String groupID, String forumThreadID)*** returns ***ForumContents,*** which is a class that contains information that is used to display the contents to the browser. The ***ForumContents,*** includes information on the group, the members in the group, the forum, the forum threads or the forum posts. This information is parsed and displayed in the browser. The request format is specified as ***#groupdiscussion/{UUID}/{UUID},*** where the first UUID is used to identify the group and second to identify the forum thread. If the second UUID is set as 0, then the view displays the threads in the forum, if not the view fetches as displays the posts for the corresponding forum thread.

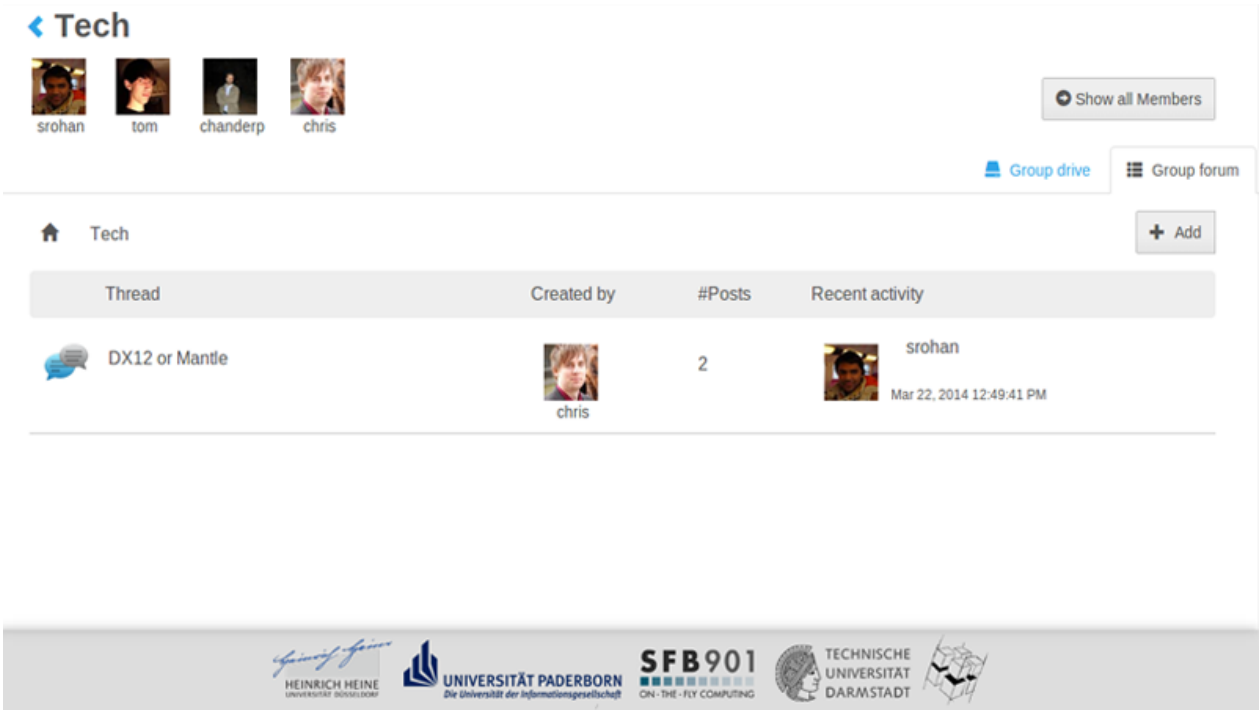


Figure 5.5.15: The HMTL GUI for Forum Thread view

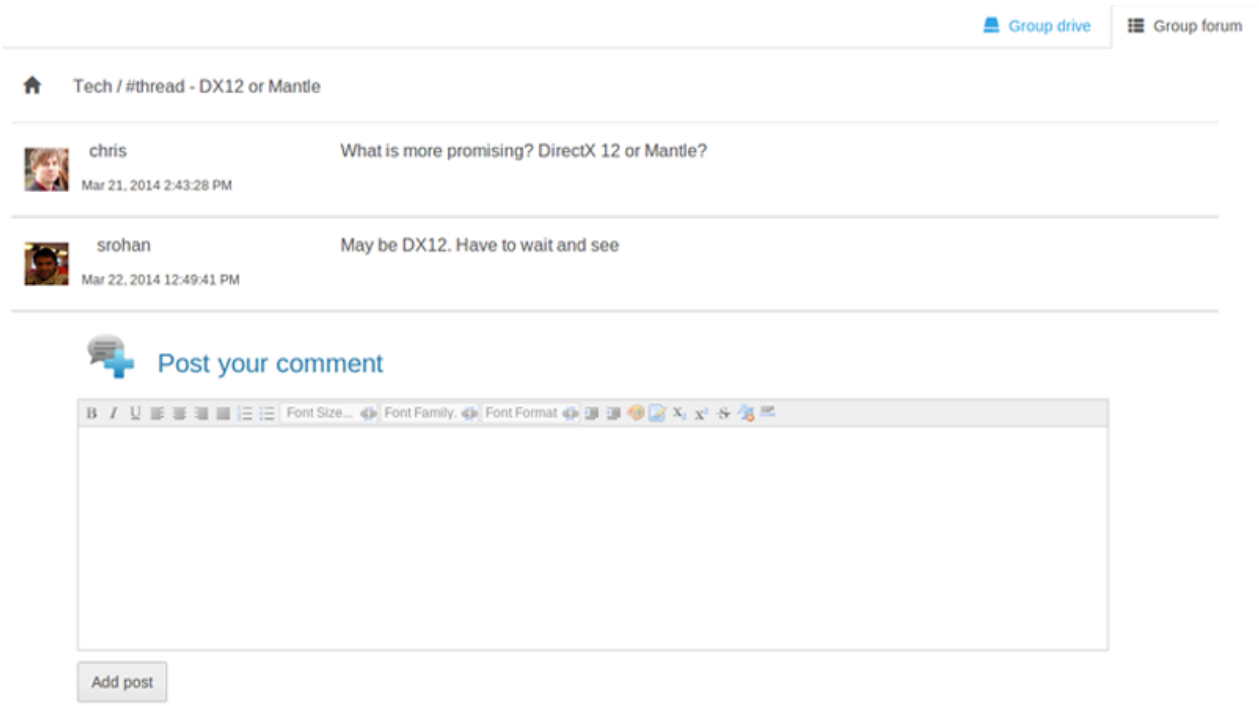


Figure 5.5.16: The HTML GUI for Forum posts

Implications and problems:

The group implemented is not encrypted and stored. This is potential vulnerability in the network. Presently only public groups can be created. Private groups needs to created, which can be encrypted and stored using symmetric keys. The data signing enforces some basic access controls, but only the owner of the item can delete the item. This is access control mechanisms must be modified so that restrictions on folders can be implemented, the group owner should be allowed to delete any items in the drive and administrators can be assigned to the group. Additionally, user invitation should be implemented for private groups instead of the present mechanism to add users to group.

The forum only allows users to create threads and posts. The forum does not allow the posts or threads created to be edited or deleted.

5.5.8. Files(MM)

Short Overview:

The "**FilesPlugin"** is used to store and retrieve files in the network. The users can upload and download different files with different extensions(.pdf, .xls, .ppt, .doc, .png, .jpeg etc.). Also, the files can be sent to other friends in the network via message, chat or email. The delete option is

used to remove the files permanently from the network. Altogether, the main features of files plugin is to upload, download, delete and share files.

Adaptations to LifeSocial:

GUI	Files HTML GUI
Plugins	**FileStoragePlugin:** ● Upload files ● Download files ● Delete files ● Share files via chat or message **FilesPluginHandler:** ● registered to RESTHandler
Framework	
Overlay	

Functionality Overview:

The Files plugin provides the method ***storeFile(File file, String storageFileName)*** which returns *FileInfo* for the file uploaded. *FileInfo* contains a unique id and key, which can be used to access and download the file. This *FileInfo* is stored in the *File* class. The ***File*** class is then added to the ***DistributedLinkedList*** using the list Interface method ***add(Object object).***
Similarly, the file can be deleted using **deleteFile().** The file can be downloaded by calling the function **getFileList().**

Detailed description:

HTML GUI for Files

Whenever a user clicks on the files option in the main GUI page he can view the page as shown in the figure 5.5.17:

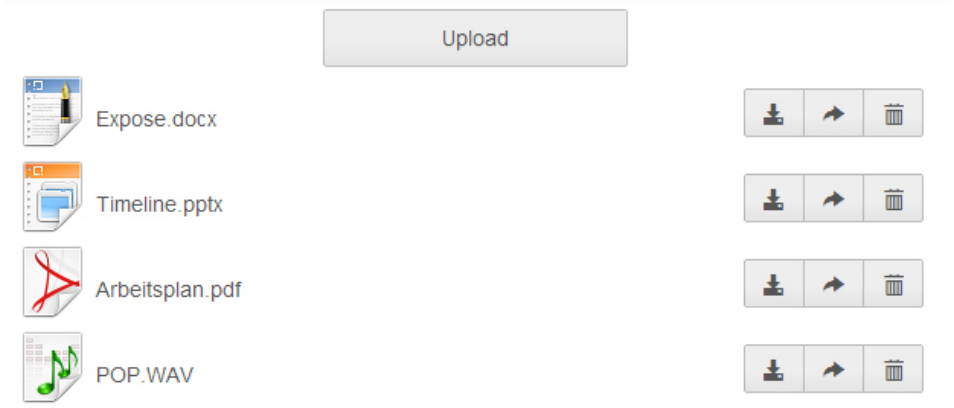


Figure 5.5.17: GUI for Files Plugin

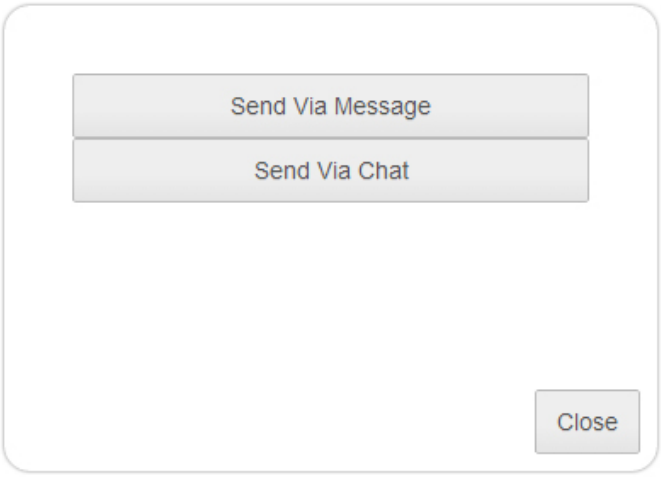


Figure 5.5.18: Ways to send the file

The user can upload files with various kinds of documents such as word documents, excel sheets, pictures, powerpoint slides, audio and video lectures etc.

Technical Description for GUI:

The request for the file upload from the browser is sent to the resthandler interface call such as

```
StoredFile dlFile = PluginProvider.getPluginProvider().getFileStoragePlugin().getFileName(baseRequest.getParameter("filename"))
```

The PluginProvider invokes methods **getPluginProvider(), getFileStoragePlugin()** and **getFileToOutputStream(baseRequest.getParameter("filename"), response.getOutputStream())**

In order to delete the uploaded file, **PluginProvider.getPluginProvider().getFileStoragePlugin().deleteFile(id)** is used.

5.5.9. Friends (NB)

Short overview:

The friends plugin provides functionalities to Users for sending friendship requests to other users, adding, declining, blocking and unblocking friendship requests from the selected user (another). Project Group for P2P Framework focussed on two features for this plugin. The first one is blocking/ unblocking feature which allows User to restrict receiving friendship requests.

The second feature is that the plugin also manages different login statuses set by the User. The login status can be set to Online/ Away / DnD/ Invisible. After login the status of user is set to Online by default. This status is visible to his friends on their chat bar containing friends list.

Adaptations to LifeSocial:

GUI	Friends HTML GUI
Plugins	**FriendsPlugin:** ● Send friendship requests ● Accept friendship requests ● Decline friendship requests ● Blocking Users ○ Friend ○ Not a Friend ● Sending updated login status to all friends. **FriendsPluginHandler:** ● registered to RESTHandler
Framework	
Overlay	

Detailed description:

Blocking another user works exactly like adding another user to friend list. Difference is that blocking stores the username in a different list (***myOwnBlockedList***). For example, User A has been blocked by User B and then A sends the request again. In this scenario when ***acceptFriendshipRequest()*** gets the control for User B, it will be checked if User A exists in B's Blocked List. If A is not present in the blocked list of B, only then B is able to receive notification for friendship request from A, otherwise request is discarded.

Blocking of friendship request is handled by the method ***blockFriendshipRequest(username, notificationId)***. For blocking a user there are two use cases:

1. Blocking an existing friend: In this scenario the friend is removed from ***myOwnList*** of User which contains his friends list using ***dismissListedFriend(username)*** and then ***addFriendToMyBlockList(username)*** adds the friend to ***myOwnBlockedList*** which contains blocked people for the user. The list is then stored using ***storageDisaptcher*** to store the shared item i.e. ***myOwnBlockList***.

2. Blocking *the* person who is not a friend already: Here the username is directly added to the ***myOwnBlockedList*** of the User using ***addFriendToMyBlockList(username)***. And then stored using ***storageDisaptcher***.

Figure 5.5.19 depicts the storing mechanism of Blocking and accepting friendship request.

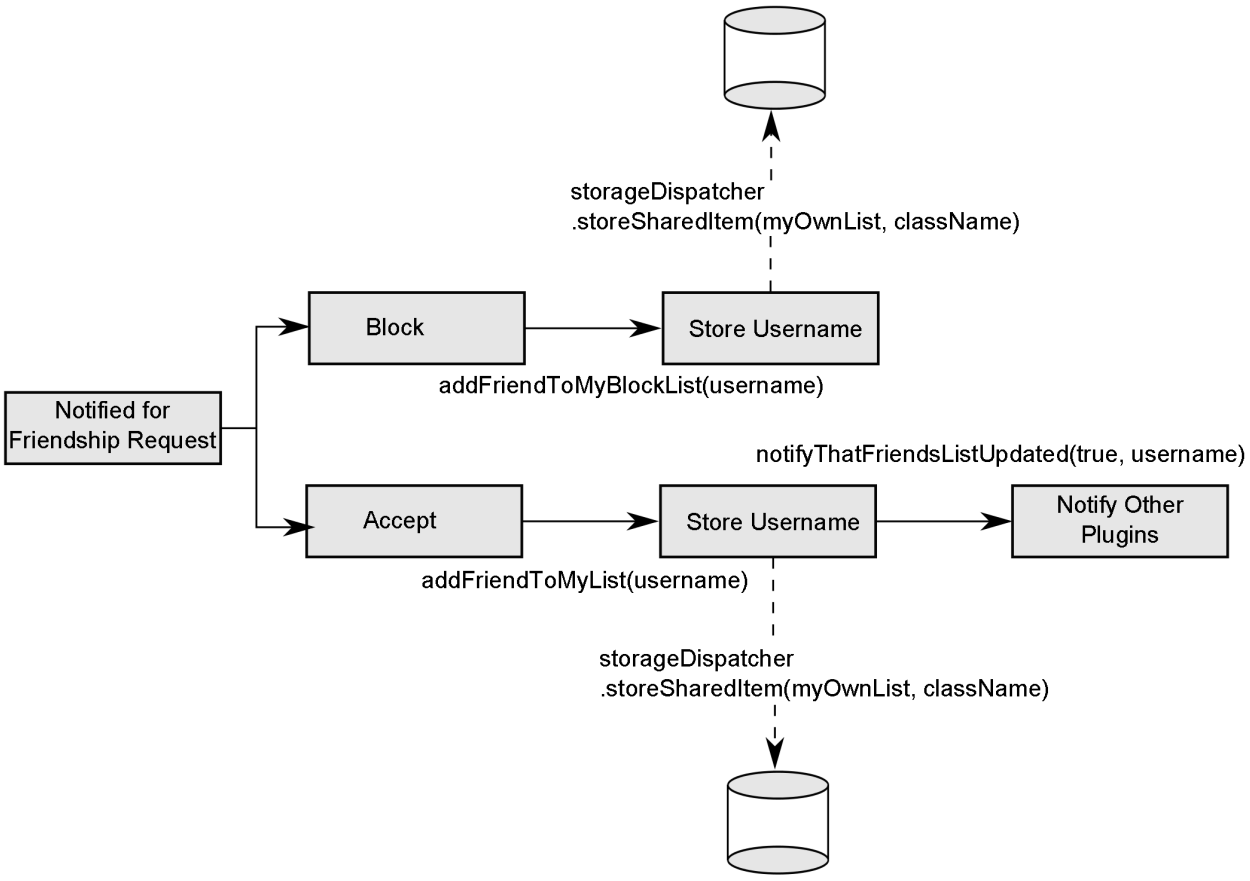


Figure 5.5.19: Storage mechanism for Blocking and Accepting Friendship Request

HTML GUI for Friends Plugin:

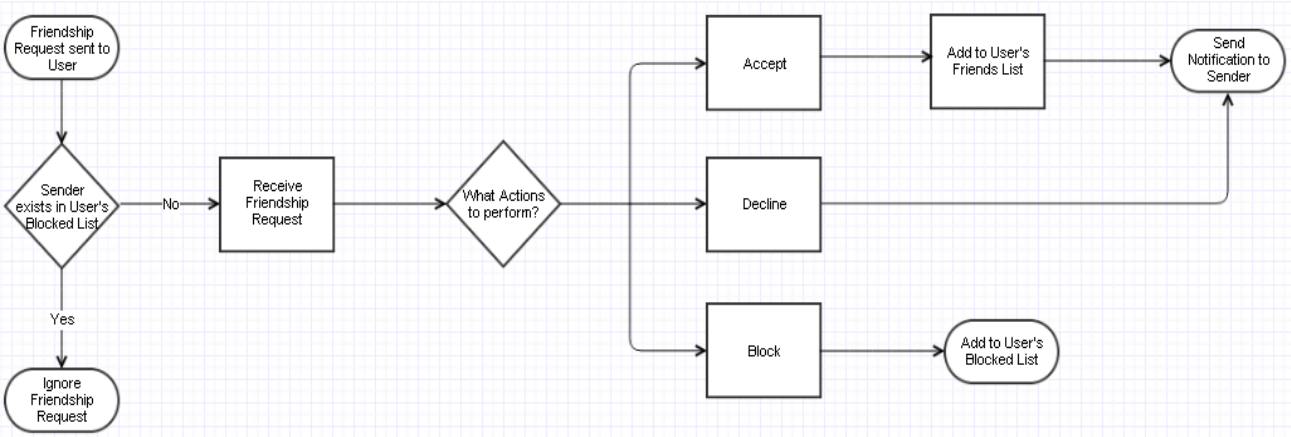


Figure 5.5.20: Receiving Friendship Request

Figure 5.5.20 depicts the scenario of receiving friendship request. It is always checked first, whether sender is in blocked list of the user by the method ***isUserBlocked(username)***. The method returns ***true*** in case the sender is present in ***myOwnBlockedList,*** otherwise ***false***. The user is notified only if the method returns ***false*** otherwise the request is ignored.

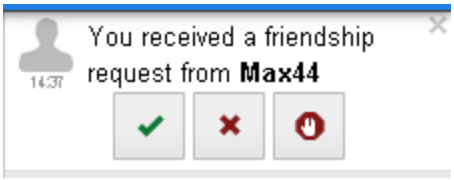


Figure 5.5.21: Notification of Friendship Request

After user is notified about the friendship request he/ she can also accept/ decline it.

1. Accept:

It is handled by the method ***acceptFriendshipRequest (username, notificationId)***. If username of sender is not present in the Blocked list of receiver, sender is added to friend list of the User (receiver) by ***addFriendToMyList()***. The list is then stored using ***storageDispatcher*** to store the shared item i.e. ***myOwnList***. After storage is successful a notification is sent to the initiator informing him/ her about acceptance using ***sendNotification()***. Similar notification is displayed for the acceptor as well. Also all the other plugins are notified about the added friend using the method ***notifyThatFriendsListUpdated(true, username)***.

2. Decline:

Method ***declineFriendshipRequest (username, notificationId)*** handles declining of a friendship request. If username of initiator is not present in ***isUserBlocked(username)***, the request can be declined and notification is sent to the initiator.

The snapshot in figure 5.5.22 is for the user **Max70** showing his friend list and blocked list.

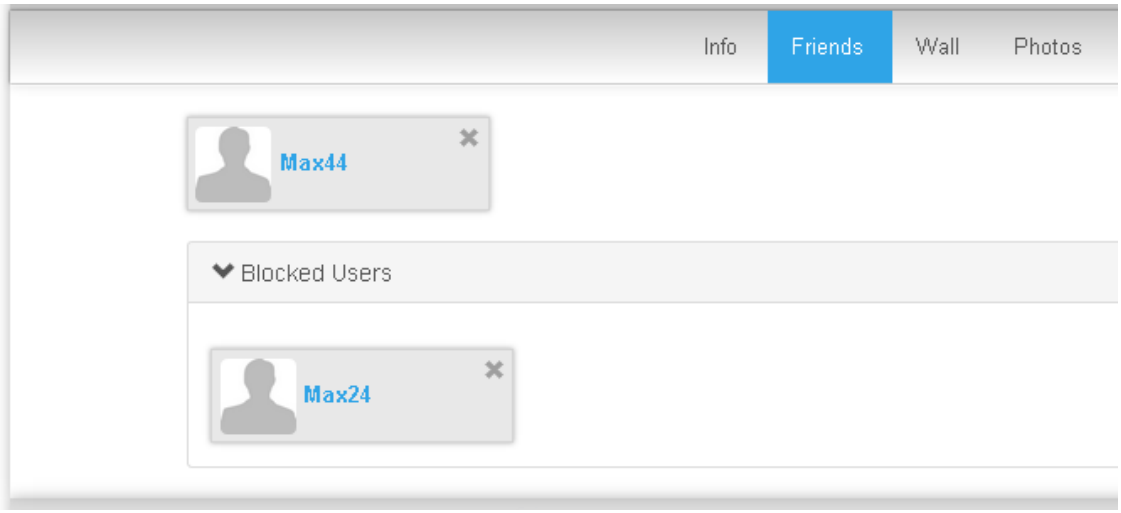


Figure 5.5.22: Snapshot of Friends and Blocked List for the user Max70

HTML GUI for Login Status:

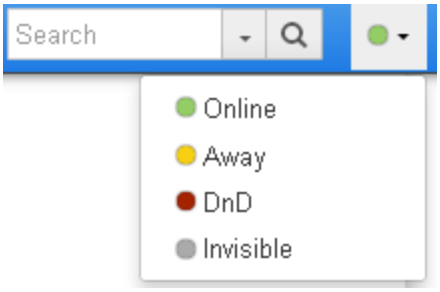


Figure 5.5.23 Login Status

Change of login status is handled by the method ***loginStatusChanged()***. If user logs out this method resets the friends plugin. Otherwise method ***sendLoginStatusUpdateToFriends()*** is called for updating view for all of the user's friends wherever the login status is shown for him. The message is sent to all his friends using ***messageDispatcher*** without storing and without checking their online status. One of the content of the request message is the status of the user which is fetched using ***getLoginStatus()***.

Implications and problems:

Currently, if user blocks another user, blocked user is still able to look for the User's profile using search. This can be managed using privacy settings for the user if he wants to be searched or his profile to be viewed by anyone/ friends of friend/ unblocked users. Another improvement can be done for default value of login status. It is always set to **Online** whenever user logs in. Login status should be set to the last changed one instead of **Online** every time user logs in.

5.5.10. Calendar(EG)

Short overview:

Calendar plugin offers a private calendar for each user. In the Calendar plugin the user is able to create, delete and modify events. Events can also be created over multiple days. A single event stores the following information: event name, event description, event start/ end date and event start/ end time. Furthermore, user also receives a notification at the time of event as a reminder

Adaptations to LifeSocial:

GUI	Calendar HTML GUI
Plugins	Calendar ● create events ● delete events ● modify events ● event reminder CalendarPluginHandler ● Registered to RESTHandler
Framework	
Overlay	

Detailed description:

The working mechanism of calendar plugin is shown in figure 5.5.24. If a new user is registered, a calendar is created for the user. The same calendar is fetched every-time the logs back in. Every calendar is identified by an unique **calendar key**. This key is generated when the calendar is created. A calendar maintains a **list** of events. Each event is identified by a random **UUID**. A calendar only fetches the event for the selected month and not the whole list. Modification and deletion of event is done with event ID.

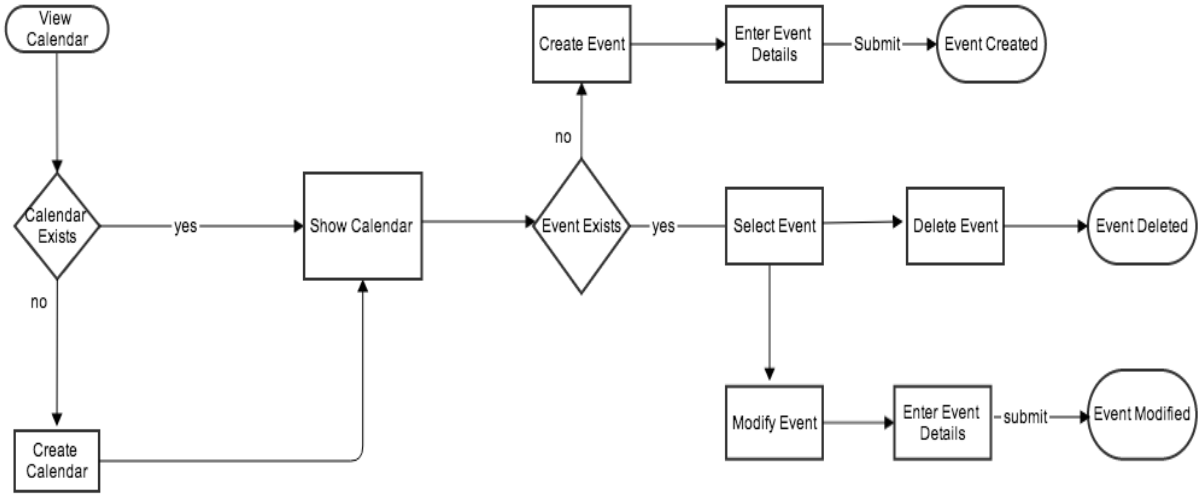


Figure 5.5.25: Working mechanism of Calendar plugin

Creating a calendar:

Every calendar is identified by a unique calendar key. The calendar key is generated by concatenating the username and 'calendar', which is a string. For e.g. If a calendar is created for user 'A', then his calendar key will be 'calendar + A'. Here, the username is provided by the framework. The funtion ***buildCalendarKey(String username)*** is responsible for creating calendar keys. The function ***createCalendar(String userName)*** creates a calendar for a user. So, when ***createCalendar(String userName)*** runs, it calls ***buildCalendarKey(String username)*** that generates an unique calendarkey. This calendar key gets assigned to the created calendar.

Storing a calendar:

The function responsible for storing a calendar is ***restoreCalendar(Calendar cal1, String calendarKey)***. The calendar that has to be stored is of the type ***GenericStorageItem*** that is the generic implementation of the ***storageItem***. A ***storageItem*** is an object that can be stored and retrieved in the network. Keeping the future scenario in mind that a user can share his calendar with other users, read rights over the calendar has been enforced. The data is encrypted with a symmetric cryptographic key, which can be made available to the authorized users. The symmetric key is encrypted with the user's public key. To access the data item for reading, the symmetric key is decrypted and is then used to decrypt the encrypted data item. The symmetric key can be decrypted by using the user's private key. Before a calendar is stored, it is encrypted by using the Asymmetric key of the user that is provided by the framework. Then the ***SecureStorageItem*** is created that contains the genericItem, userKey, userID and acl. ***SecureStorageItem*** allow secure sharing of the stored item. Lastly, ***storageDispatcher*** stores the ***SecureStorageItem*** in the framework.

Creating Events:

Currently, ***createEvent*** functions provides the following fields: event name, event description, event start/ end date and event start/ end time. The function ***createEvent*** creates an object and stores the above details in that object. Before adding the event, the function ***getCalendar()*** retreives the calendar of the logged in user. The created object is then passed to ***addCalendarEvent(CalendarEvent event)*** that adds this object to event list. This list maintains the ordered collection of events. Then the function ***restoreCalendar()*** encrypts and stores the event in the calendar. Every event is identified by an event ID which is a random UUID.

Deleting Events:

The function ***deleteEvent(String event_id)*** takes care of event deletion, deletion of events is done with eventID. Firstly, the calendar of the logged in user is loaded. Secondly, the id of the event to be deleted is passed to ***removeCalendarEventById(String id)*** and the calendar is restored. The function ***removeCalendarEventById(String id)*** searches the list for the event with the same id and removes that event from the list.

Modifying Events:

The function ***modifyEvent(...)*** is responsible for modifying events. Again, the event to be modified is identified by its eventID. Firstly, the calendar of the logged in user is loaded. Secondly, the eventID is passed in ***getEventByID(event_id)*** that searches the list for the event with the same id and loads that event. Now the event details can be modified. Lastly, it restores the calendar.

Fetching Events:

The function ***getCalendarEvents(...)*** fetches only a month of events and not the whole list. For e.g., if the month is January and the year is 2014, then, it will only fetch the events occuring on January 2014.

HTML GUI for Calendar Plugin:

The HTML GUI of the calendar provides a month view. The calendar takes the current month as its default view. The calendar displays the events on their corresponding dates. The events are stored as a list. Multiple events can be created and displayed on a single day, whereas, a single event can also created and displayed across multiple days as shown in figure 5.5.26.

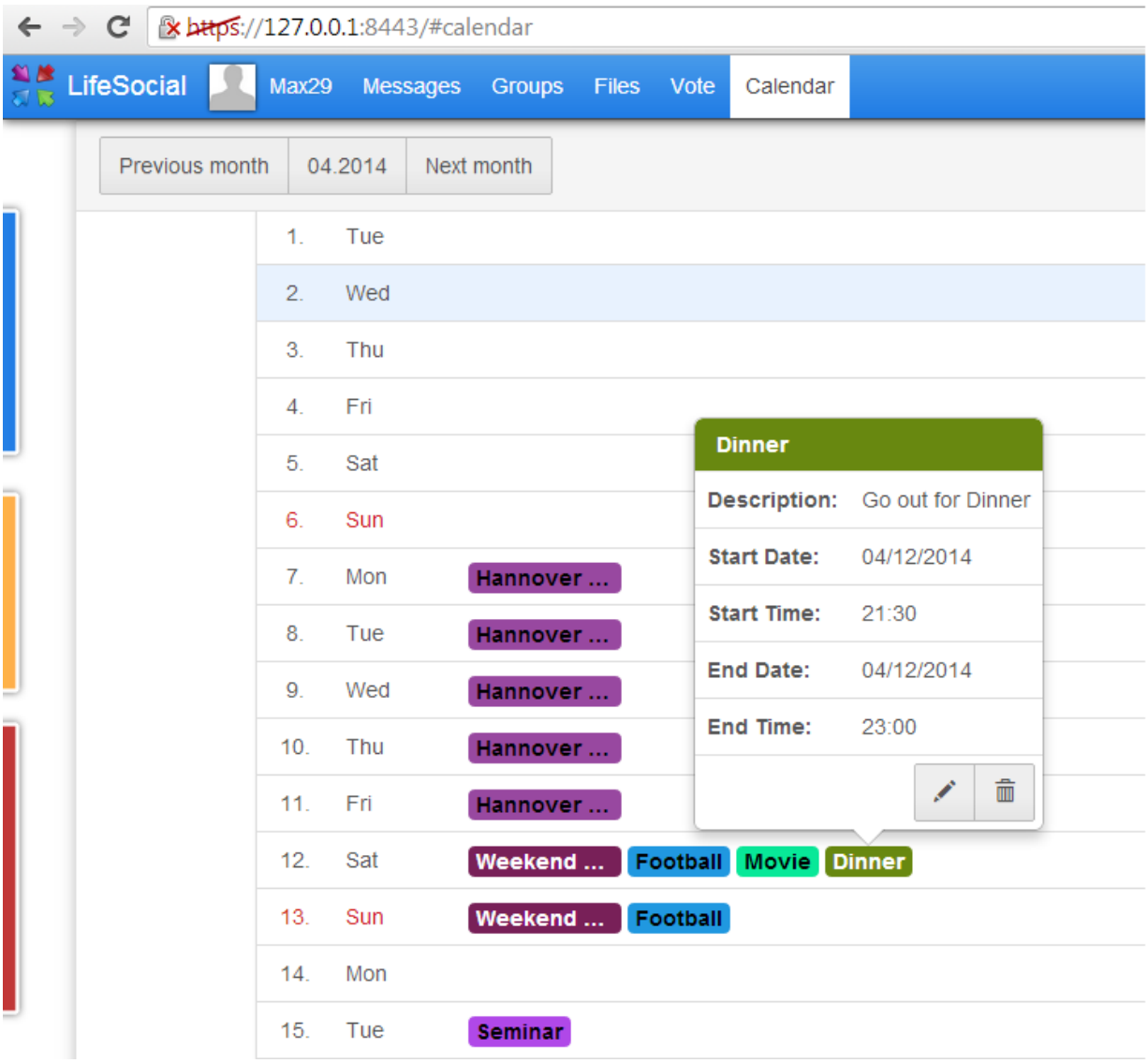


Figure 5.5.26 The HTML GUI for calendar

Currently, there are three options for month navigation:

- Previous month: navigates to previous month
- Next month: navigates to next month
- Date picker: allows users to select any month and year and displays the calendar for that month

Figure 5.5.27 given below shows the available options for month navigation. It shows the datepicker from which we can navigate to any month in just few clicks.

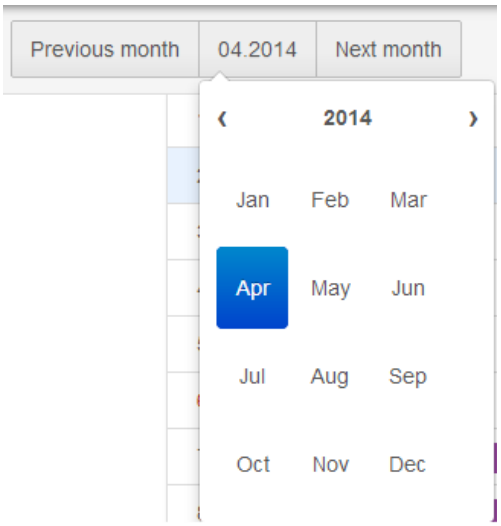


Figure 5.5.27: Month Navigation

Currently, the create event function provides event name, event description, event start time, event end time, event start date and event end date. Therefore, while creating events user can enter the above details. When the event is created, its name is displayed in front of its start date in the calendar. Events can also be stored over multiple days, in this case, the event name is displayed across the event start date and event end date. Figure 5.5.28 shows the modal window used for creating events.

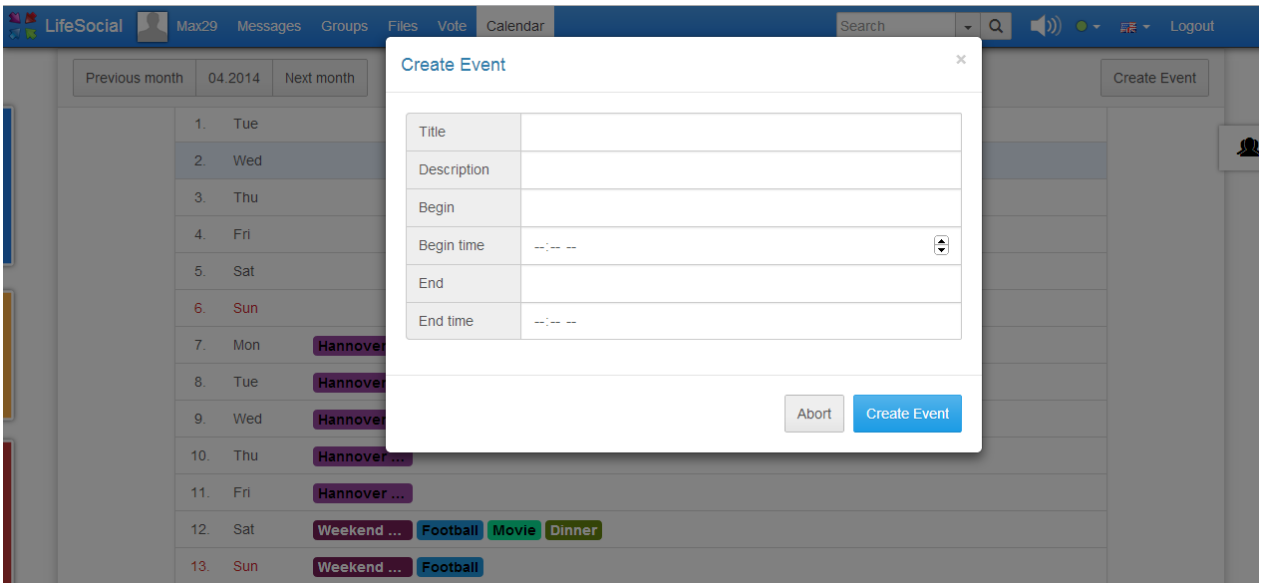


Figure 5.5.28 Modal window for creating events

Once the events are created, they are displayed on their corresponding dates. Event details are displayed via a popover as shown in figure 5.5.29. Deletion and modification of events are provided with this popover.

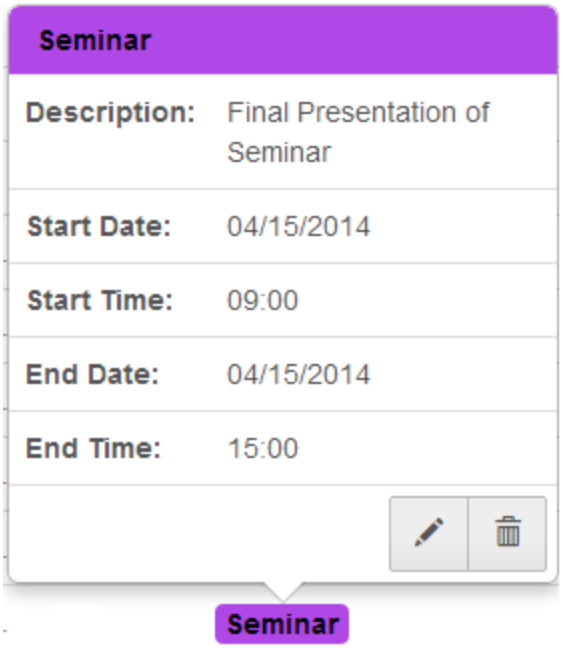


Figure 5.5.29: Pop-over for event details

When modify event is clicked, a modal window with already saved information appears as shown in figure 5.5.30. After we submit the form, the old data is replaced with the newly entered one.

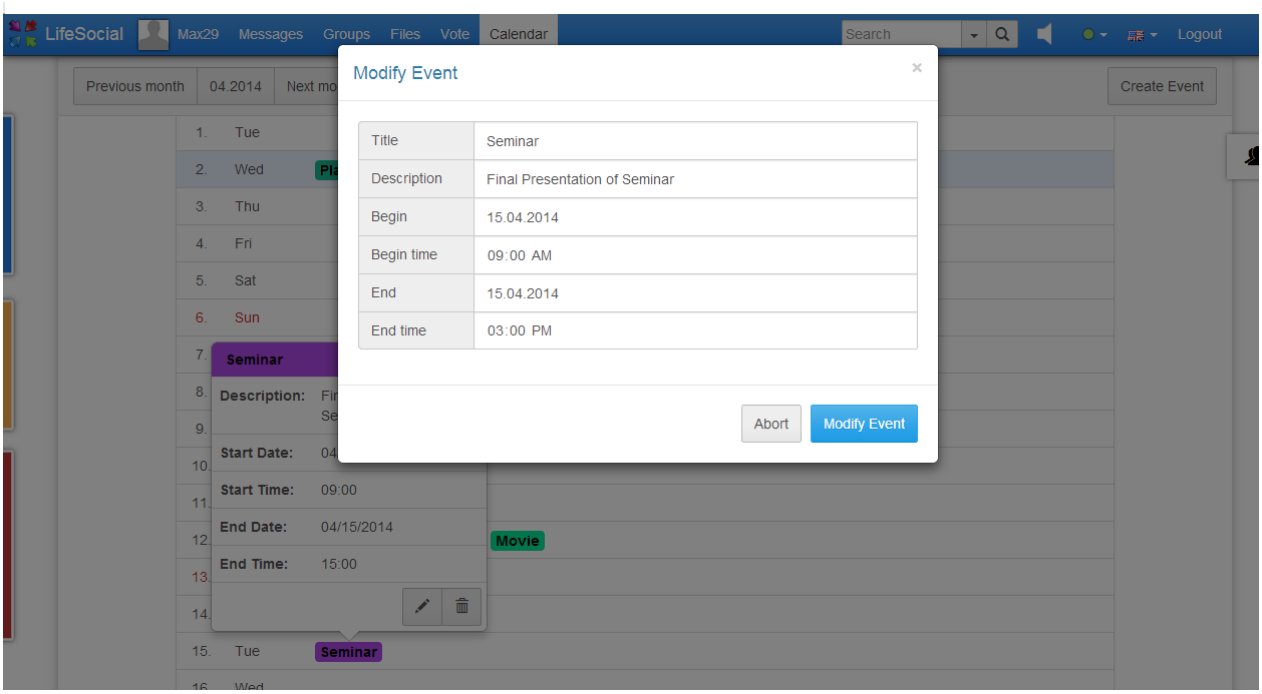


Figure 5.5.30: Modal window for modify event

For every event a user creates, there is a event reminder at the starting time of an event. The user receives a notification that shows that an event has started as shown in figure 5.5.31.



Figure 5.5.31: Event Reminder

Implications and problems:

Currently, there are two open problems with event reminder, they are:

- A user do not have an option to set a date or time for event reminder. The event reminder notifies the user at the starting time of the event. As an improvement, user should be able to set a date or time for event remider for e.g. a day before or 3 hours before the actual start time.
- In order to receive event reminder, user has to log out and log in again. This does not work in the same session.

5.5.11. Search (CJ)

Short Overview

Search plugin is used for searching their friends registered in the network. Users can search their friends with many different criterias based on their name,city,country,gender.etc Users can search for their friends in the drop down menu in the search field GUI with many advanced search fields for every criteria. This enables users to search their friends easily.

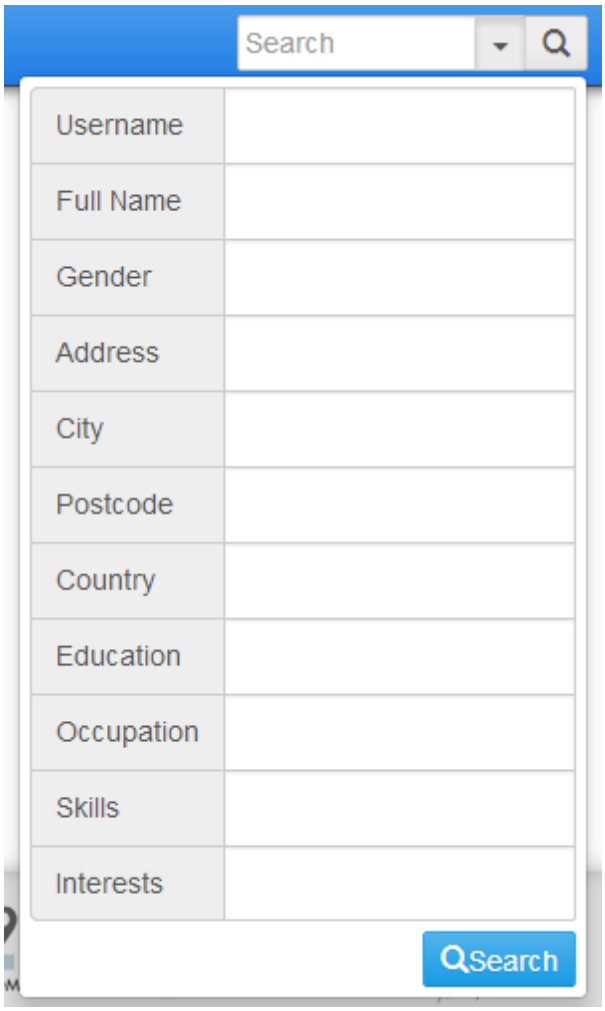


Figure 5.5.32: HTML GUI for search plugin

Adaptations to LifeSocial:

GUI	Generic search field, Advanced search HTML GUI
Plugins	Search plugin, Profile plugin
Framework	
Overlay	

Detailed description:

Indexing:

When user is registered into the network , the user profile data is indexed in the query attribute

based on the **Soundex** value for all the user profile field values. Soundex is an encoding method used to find the word with similar phonemes. The list of query attribute containing the soundex values are then indexed with the SearchContainer containing the real values using Join query engine.

Whenever there is a change of values in the user profile field, the values are updated for the search.

Search :

The user may search the other users by entering values in generic fields or from the advanced search fields. If the value entered is from generic or advanced search fields, the field values are mapped to the respective soundex values. Using the Join query engine the search results are obtained. The search results may contain the usernames which does not sound much similar to other search results. So using **weighted Levenshtein distance** algorithm the values are compared to each other and filtered to get the most similar words for the search results. Levenshtein distance is defined as the minimum number of edits required to transform string A to string B, where an edit is the insertion, or deletion of a single character, or the replacement of a character with another character. It also defined as much the "difference between two strings", for a certain definition of distance. Finally the search results are sorted using levenshtein distance as shown below.

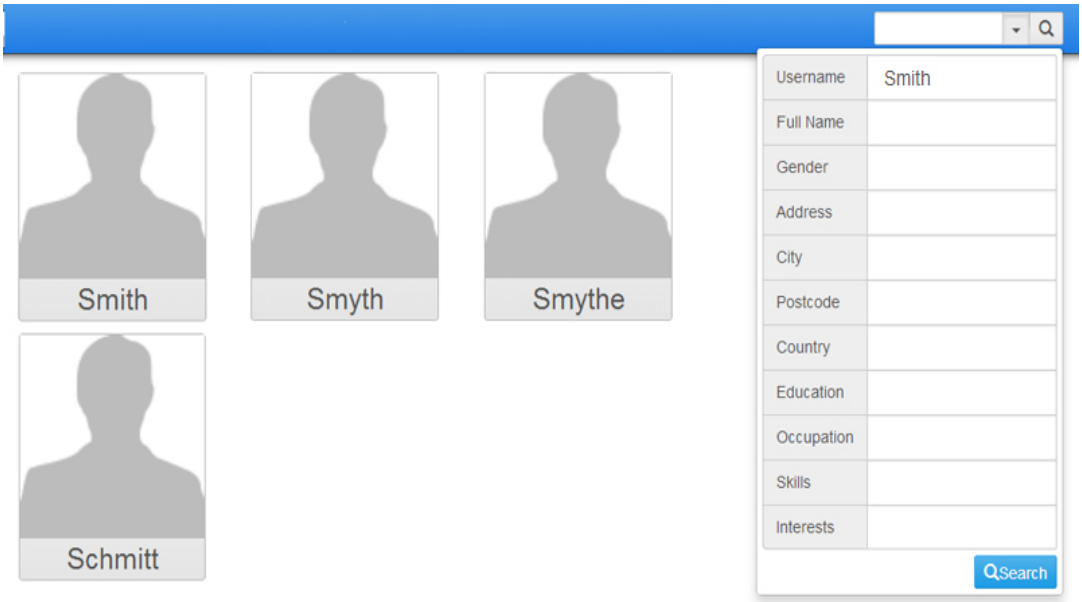


Figure 5.5.33: Search results for username:Smith

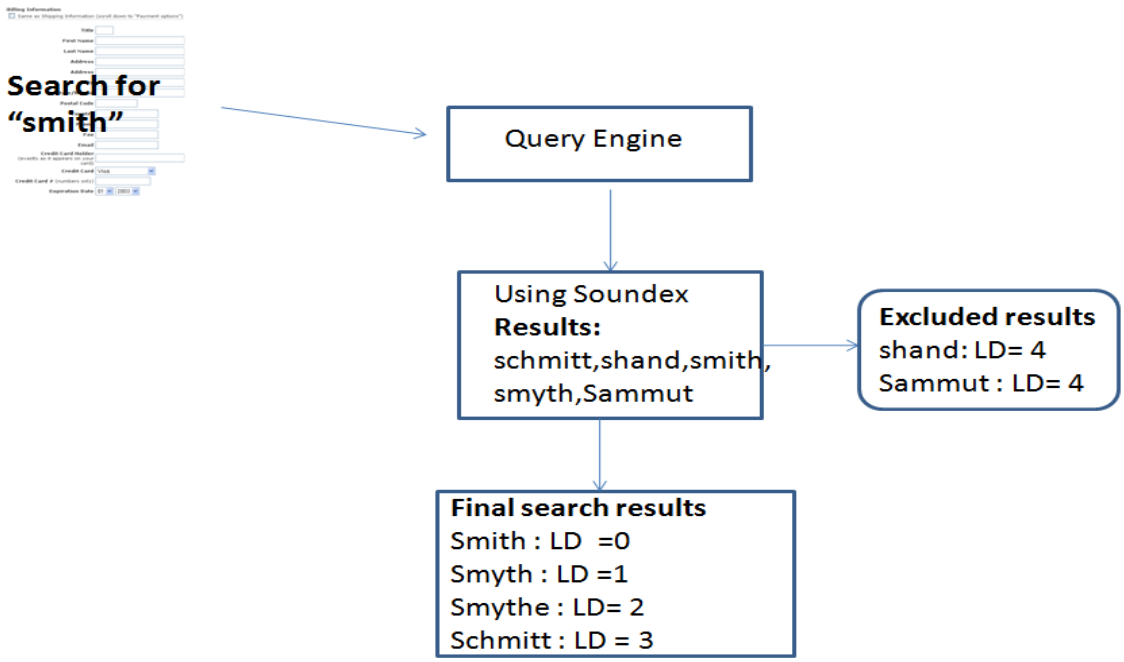


Figure 5.5.34: Working flow of search using Soundex

When a search is performed, the search value is passed to the join query engine. The Join query engine issues a remote parallel operation where multiple clients responsible for the keyword are requested to return the data matching the search criteria. This operation is performed by calling ***issueParallelOperation()*** interface provided by ***DistributedDataStructures.*** The parallel operations returns a collection of strings which are serialized. Using **Levenshtein** distance, this collections are filtered and sorted. We set a timeout of 5 seconds when there aren't any results from the remote operation issued. We set this timeout, since the client issuing the parallel operations waits forever, until any results become available. For better response we set this timeout for the remote operation.

Implications and problems:

Search plugin is only used to search for the users and not for any other informations like messages or groups. Using the remote operation the search takes at most 5s to show the results . Moreover , when there is a search from generic field it takes more time compared to the field specific search as it has to collect the search results for different fields.

5.5.12. Profile(KB)

Short overview:

Profile plugin shows information of the user in a more intuitive HTML GUI. Multiple fields are introduced and are put in four major categories. Editing is made easier for users by using the same view for both edit and show. User can control privacy of profile data by using the privacy

feature found under each fields. Users can also set cover and profile pictures in the new GUI.

Adaptations to LifeSocial:

GUI	HTML GUI
Plugins	profile plugin
Framework	
Overlay	

Detailed description:

In the new profile view, users can set their profile and cover picture that are made visible to public. The info section has the profile view for users with multiple fields and privacy controls associated with each field. Data fields are placed in four categories with relevant fields under each one of them. Edit button enables edit mode of profile within the same view. Privacy settings implemented for profiles are explained below.

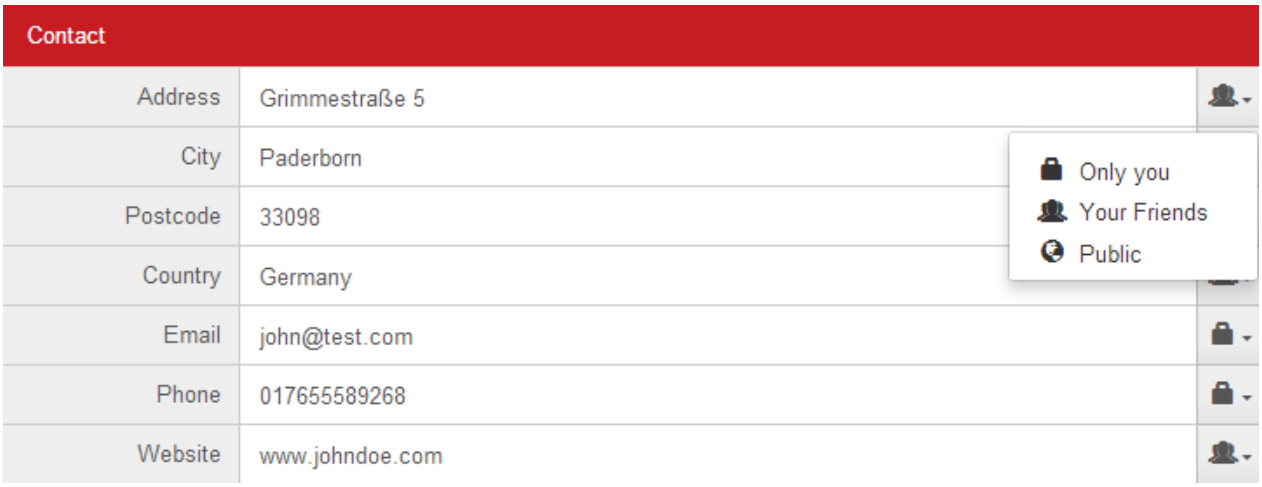


Figure 5.5.35: Profile Privacy Settings View

- Only you: Make the selected field visible only to the owner.
- Your Friends: Friends of the owner can view the field.
- Public: The field is not hidden to anyone.

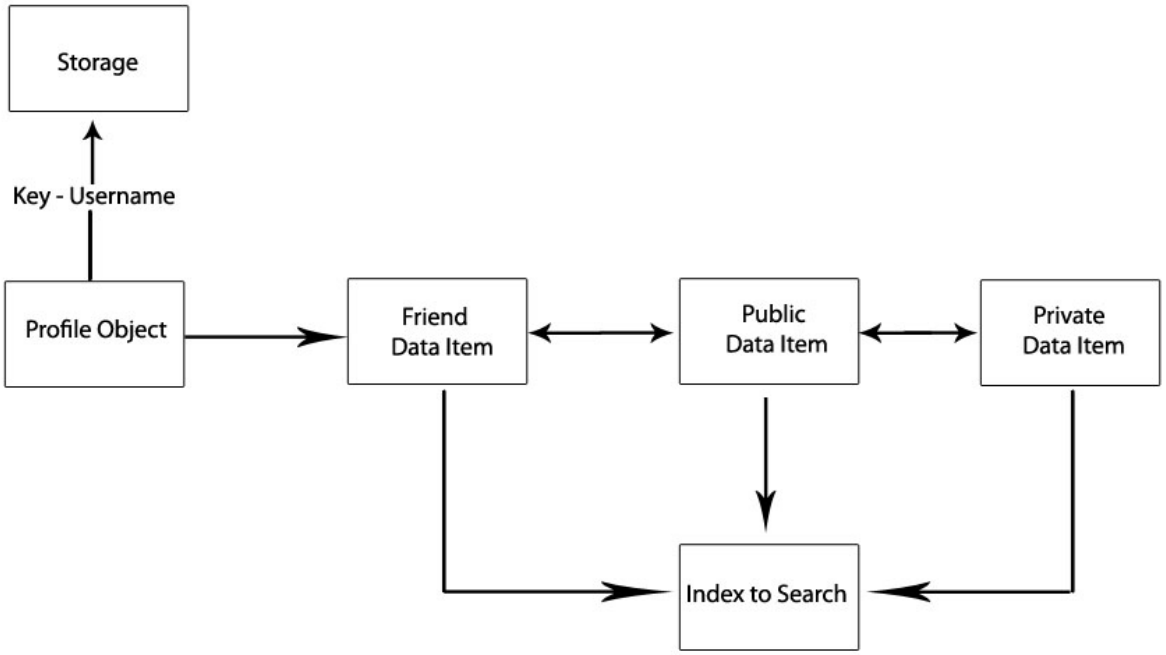


Figure 5.5.36: Handling of Profile Object between Data Items.

When a field is changed to private, the private data item is first checked if it already contains the field, else if friends or public data item returns not equal to null, the field name is removed from the respective data item and set to private. The same operation is performed when a field to set to public or made visible only to friends. The Profile is encrypted and stored in the secure storage item. For any field, intersection of Public, Friend and Private data has to be null which means a field cannot be in two different data items. When the field is moved from private to friend, it is first removed from private and then added to friend data item. Regardless of their privacy setting, every field is indexed to make it searchable using search plugin.

Implications and problems:

Privacy settings can be enhanced by adding master set capabilities. Introducing single privacy handle to all fields or to fields under single category such as contact, work, personal can be done. When a field value is set to private, it shouldn't be indexed to search.

5.5.13. Error Notification and Console (CP)

Short overview:

The aim of this plugin is to make a user capable enough to see and understand what is going inside the LifeSocial at any instant which makes LifeSocial more transparent and easy to understand. In any application, developers are always keep their focus on the correct working of the application they are developing. The LifeSocial differs from other application in the way it

treats its user. The “Error Console” is dedicated to display all the details that are generated from the framework. It includes all the details, like information, error, warning and debug messages.

Adaptations to LifeSocial:

GUI	HTML GUI : Error Console Template used : loggingconsole.html
Plugins	LoggingHandler Plugin : Registered with RESTHandler
Framework	
Overlay	

Detailed description:

This is a new feature introduced in LifeSocial’s new GUI. This was not the part of the previous PG. Till last PG, the LifeSocial was suffering poorly from the way errors were handled and reported. In an application, at times several messages are generated. It may be due to the errors, informations or some other kind of messages that user should know and can respond accordingly. It was seen in the last PG that at times the LifeSocial was hanged due to some problems (errors which were handled incorrectly) and at that time the user has no idea why the whole application is hanged-up e.g., when no bootstrap was provided, the LifeSocial tried to connect in an infinite loop and the user had no idea what is going on behind the screen. The aim for this plugin is to provide a central place to the LifeSocial where all the application related messages can be displayed. This makes easy for a user also to check what is happening inside the application at any time. The user can be notified in real time and based on his/her session.
The error console plugin inside LifeSocial uses easy to understand messages throughout the execution of the LifeSocial. Currently there are four kinds of application level messages to log, they are informations, error, warning, debug. Each one of them explaines the nature of occurrence and distinguishes itself from the other using different colors. Four different colors are used to represent these four types of messages generated in LifeSocial (discussed in subsequent sections in detail).

To access the functionality the user must be successfully logged in to LifeSocial. Once the user is successfully logged in and the session is created, the ErrorConsole can be accessed (see figure 5.5.37) via the “Error Console” link on the left hand side in LifeSocial GUI.

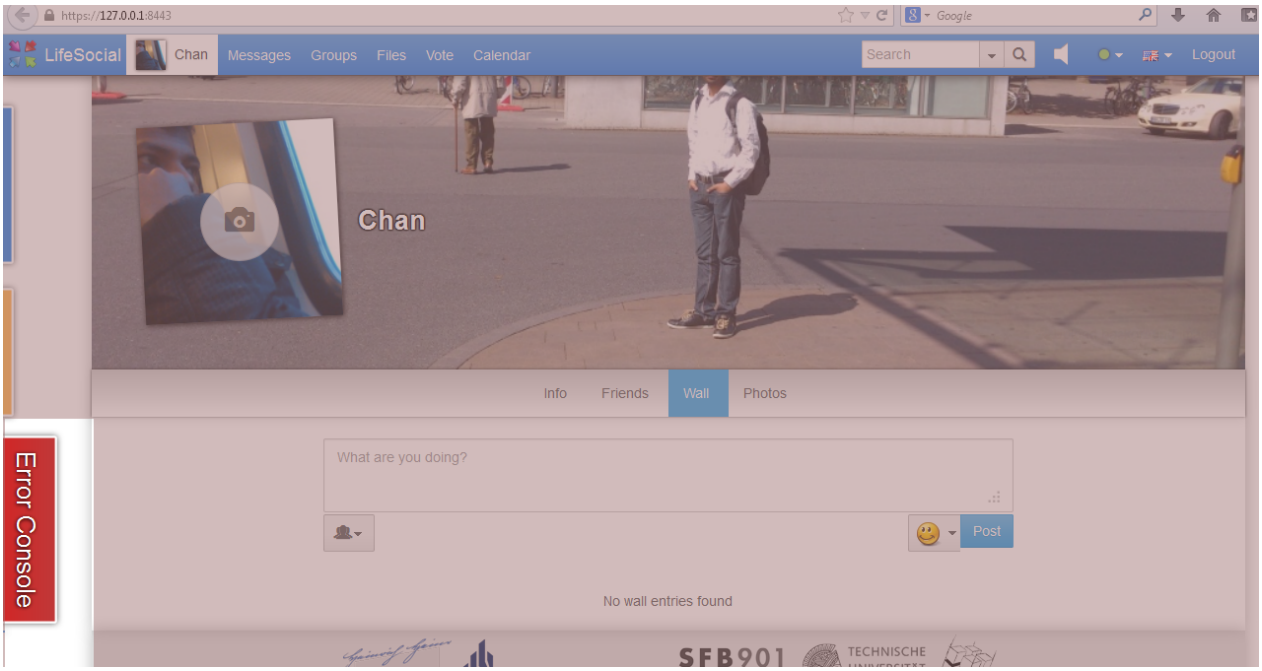


Figure 5.5.37: Accessing Error Console inside LifeSocial GUI

When user clicks on “Error Console” link inside LifeSocial GUI, the following figure pops-out which shows the error console.

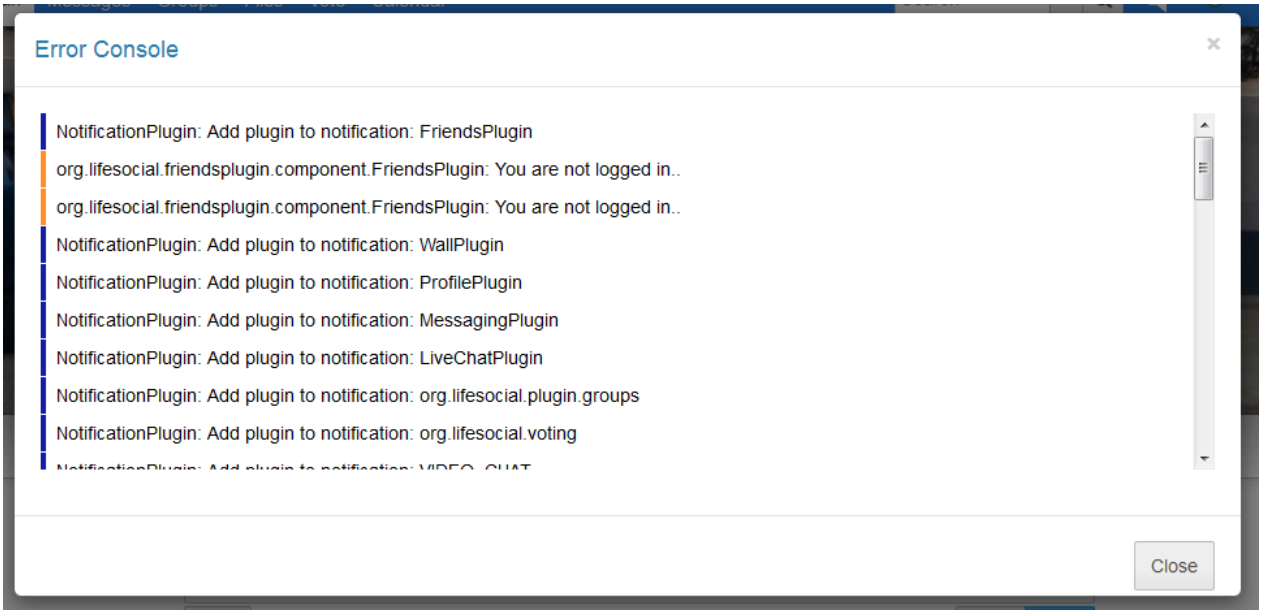


Figure 5.5.38: A typical “Error Console” view

Depending on the type of messages generated by the framework, these can be classified as follows:-

- **Warning (warn)**: These types of messages are generated to inform the user about

something that just happened in the framework for which the user must be warned, like the user is not logged in. These types of messages are displayed with an orange stripe following the message (see figure 5.5.39).

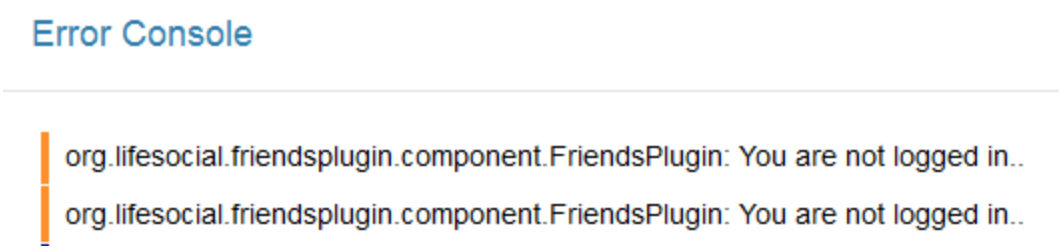


Figure 5.5.39: Error console warn message

- **Debug:** These types of messages are generated to print the debug messages on the error console. Like "building profile item key for:" a particular user. These types of messages are generally have a blue stripe (see figure 5.5.40).

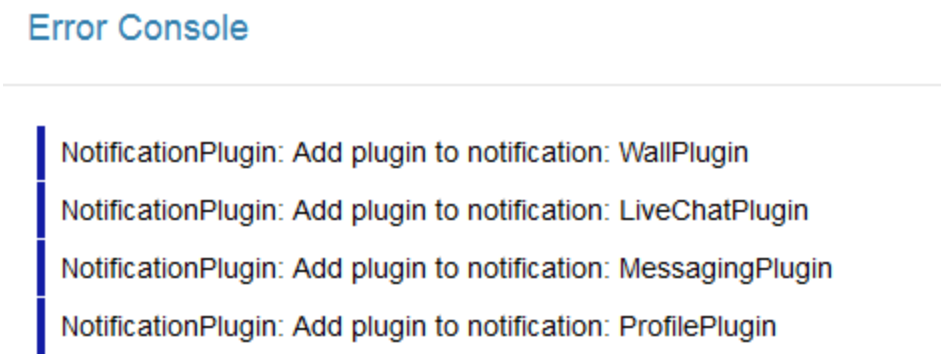


Figure 5.5.40: Error console debug messages

- **Information (info):** These types of messages are generally that contains some information that needs to be displayed like user login succeed, item not found. These messages are followed by a green stripe in the error console (see figure 5.5.41).

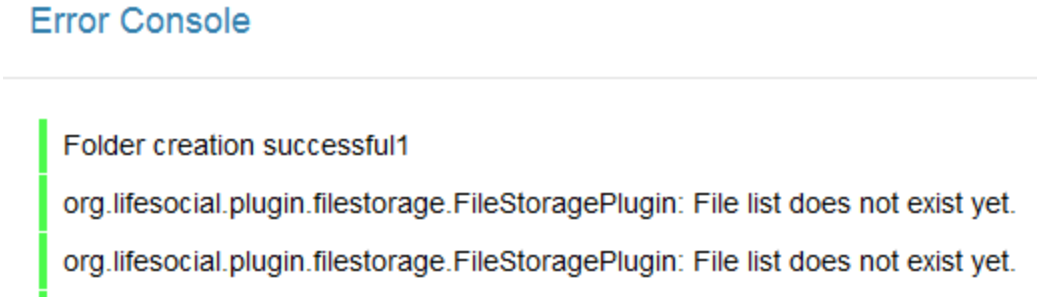


Figure 5.5.41: Error console info messages

- **Error:** Whenever there is something wrong happens in the framework, the message is displayed and the whole stack trace for that particular error is also printed on the error console describing the nature and origin of the occurred error in the framework. These

types of messages have red stripes (see following figure 5.5.42).



Figure 5.5.42: Error console error messages

Working:

The working of this plugin can be divided into two parts, namely, backend (related with the setup of logging services with other plugins) and frontend (using RESTHandler and transforming into HTML template).

The backend works as follows: The ErrorConsole comes into action as soon as the instance of LifeSocial gets started. LifeSocial loads into memory and it starts to call all other plugins from their plugin handlers call. All those plugins which wants to send the messages to HTML GUI ErrorConsole must bind **ILogService** (interface for logging service plugin) for the OSGI to bind it dynamically whenever it gets loaded. This reference should be added in the **XML file** inside the **OSGI-INF** folder of the corresponding plugin. Once the reference is added into the appropriate XML file inside a plugin, we must set binding methods (setService and unsetService). Upto this point the initial setup is complete. Now to forward message to GUI one must use, depending on the type of messages which needs to be forwarded to GUI, the methods defined inside the **iLogService** should be called. For example, if the developer wants to send a message which is an information, one can use method **public void info()** which has two overloads, one with single parameter as **String message** and second with parameters **Class class, String message**. When this method is called with the appropriate parameters, the function executes another method notifyObserver() which contains some additional details for the messages. It contains a level of the message that we have to deal (in this example the level is INFO because this example assumes that the user is using the info method), it also get the real class name for the exception and the message that has been passed to this method. This notifyObserver method adds some extra capabilities to the LifeSocial's ErrorConsole. It makes it dynamic in nature. Whenever there is a change in the anyone of these message types, the notifyObserver reflects it back and the user gets notified for the same. Next the method do is to log this into the LogService which all informations, namely, type of message (INFO in this case), class name and the original message. All other functions that has been used in LifeSocial to provide the functionality of the ErrorConsole have been summed up in the following table. The following table shows the complete list of functions that has been used in the current PG (2013/14). The table has function name with their corresponding use (type of message,e.g. INFO,DEBUG, ERROR and WARN) and a description about these function with their parameters. These all messages are represented inside LifeSocial with different colors so that these can be identified separately.

SlNo	Function name	Used for	Description
1	public void info (String msg)	information messages,	This method is used to send information messages to the GUI. All the information messages are represented by green color in the GUI. This function takes one parameter of string type which is the actual messages that is displayed at the frontend.
2	public void info (Class clazz, String msg)	information messages	This method is an overload for displaying the information messages. It is also represented in the frontend with green color. This method expects two parameters, one is type of class and another is message of the string type. The class parameters is used to get the more details like the class from which this message is generated and the message is the message which is displayed to the user.
3	public void error (String msg, Throwable exception)	error messages,	This method is to send the error messages that are generated in LifeSocial to the ErrorConsole. It is represented in the frontend with red color. The error messages function has four overloads. This method expects two parameters, one is type of string and another is exception of the throwable type. The exception parameters is used to get the more details of the exception occurred inside the application the message is the message which is displayed to the user.
4	public void error(Class clazz, String msg)	error messages	This method is the second overload for the error messages that are generated in LifeSocial. It is represented in the frontend with red color. This method expects two parameters, one is type of string and another is class type. The first parameter is used to get more information about the responsible class for that error message and string msg is the message to be sent to the frontend.
5	public void error (Class clazz, String msg, Throwable exception)	error messages	This method is the third overload for the error messages that are generated in LifeSocial. It is represented in the frontend with red color. This method expects three parameters. First parameter is of class type which is used to gather more information about the class from where the particular error has been generated. Second parameter is of string type which holds the message for that particular cause. The third parameter is of throwable type and is exception which provides the kind of exception occurred.
6	public void error (String msg)	error messages	This method is the fourth overload for the error messages that are generated in LifeSocial. It is represented in the frontend with red color. This method is one of the simplest method for this type and it expects only one parameter which is of string type and holds the message for that particular error.
7	public void debug	debug	All the debug information and related messages are

	(String msg, Throwable exception)	messages,	forwarded from backend to frontend using this method. This method has an overload for displaying the debug messages. It is also represented in the frontend with blue color. This method expects two parameters, one is message of the string type and another is exception of throwable type. The message is the string that contains the original message forwarded to the GUI and exception contains the cause of that failure inside LifeSocial.
8	public void debug (String msg)	debug messages	This method is an overload for displaying the debug messages. It is also represented in the frontend with blue color. This method is the simplest function for sending the debug messages to the GUI and it expects only one parameter, is type of string. This parameter holds the message to be forwarded to the GUI from backend.
9	public void warning (String msg)	warning messages	This method is used to forward all the messages which are of the type warning. This method has one overload, All those messages which are marked as type debug, are represented inside LifeSocial with orange color. This method expects only a single parameter of type string. It contains the original message to be sent to the GUI.
10	public void warning (String msg, Throwable exception)	warning messages	This method is used to forward all the messages which are of the type warning and represented inside LifeSocial with orange color. This method expects two parameters. First parameter is of type String and another is of throwable type exception. The string represents the original message while the exception holds the original exception occurred inside the LifeSocial during its execution.

Table: Showing functions available to be used for sending messages to GUI

To sum-up the working of backend, the following steps can be helpful.

- Add the reference of the logging service interface and its bind, unbind methods in the **XML** file inside **OGI-INF** folder in the corresponding plugin.
- Import the package inside the java class where the logging service has to be used.
- Add the binding methods inside the java class for logging service.
- Replace all the occurrences of **System.out.println(..)** with the appropriate methods from the table above and all the **e.PrintStackTrace()** should be replaced by **public void error(..)** with the appropriate parameters.

The working of frontend can be explained as:- Whenever the ErrorConsole is accessed from the GUI, it calls a function **getLogEntries()** which is of type ArrayList and returns the ArrayList of type **LogEntry**. LogEntry is a public class defined in package **org.lifesocial.logger.api**. It contains a message of type string(used to hold the message to be send to GUI) and a level of type int (for determining the severity of the message,i.e., info, error etc). Whenever anz function from the above table is called, the entry is added to the ArrayList and corresponding changes are propagated through NotifyObserver(...) method inside.

These values from ArrayList are returned to a JSON object and passed to the GUI. At the GUI, inside the js file, these values are fetched and formatted and rendered to the template called **loggingconsole** and displayed inside the div called **loggingConsoleContainer.** When the messages are received by GUI, it uses the severity to differentiate the type of message and uses the different colors for the unambiguous representation to the user.

Implications and problems:

Currently it is not possible for a user to see the ErrorConsole messages from the previous session because once the user logs out from the GUI the session gets destroyed. So when the user logs in next time it sees a clear ErrorConsole window. Though it is unlikely for a user to be interested in the messages from the last time but still in some cases it is worth having to get all the messages at some place from where it can be accessed by the user at any time. For example it can also be used as a self activity information by the user.

There is no option for the user if he/she can sort similar types of messages, for example if he/she can sort all error messages or all information messages. This feature is also not implemented yet.

5.5.14. Voting Plugin (BG)

Short overview:

Voting plugin allows users to create public and private votings, get an existing public voting by the corresponding vote ID, remove vote from user's own vote list, invite other users to vote and see vote result after submitting vote options.

Adaptations to LifeSocial:

GUI:	**HTML GUI:** Vote Own view in frontend. **Notification:** Displays notification when a vote invitation arrives. **Templates used:** vote.html, voteresults.html
Plugins:	**VotingPlugin:** ● Load and stores voting objects ● Handles operations: create vote, delete vote, vote invitation, get vote. **VotingPluginHandler:** ● Registered to RESTHandler
Framework	

Overlay	

Detailed description:

The voting plugin has been totally rewritten. Before there was a vote server structure which ran at the node that stores a voting object. It stored and updated the information of vote. But the vote server was redundancy and problematic when there were only a few nodes on the network. Now the vote server has been removed. When user votes, a DistributedLinkedList ***votingChoicesDLL*** is retrieved or created.

The detailed implementation of voting functionalities are described as below:

- Create a vote : when user creates a vote, ***createVoting()*** and ***updateVotingReference()*** are called from background application. A voting object ***Voting*** is created. It is stored as ***GenericStorageItem*** if it is public vote, otherwise it is encrypted with a symmetric key and stored as ***SecureStorageItem***. Voting object ***Voting*** contains vote ID, topic, description of the voting, minimise and maximise numbers of choices users can select, a list of choices, link to the dll where users can insert their vote, and the author of the vote. After creating a vote, the vote list is updated by ***updateVotingReference()***, voting objects are shown on users' vote list. Figure 5.5.43 below is the user interface for creating a new vote.

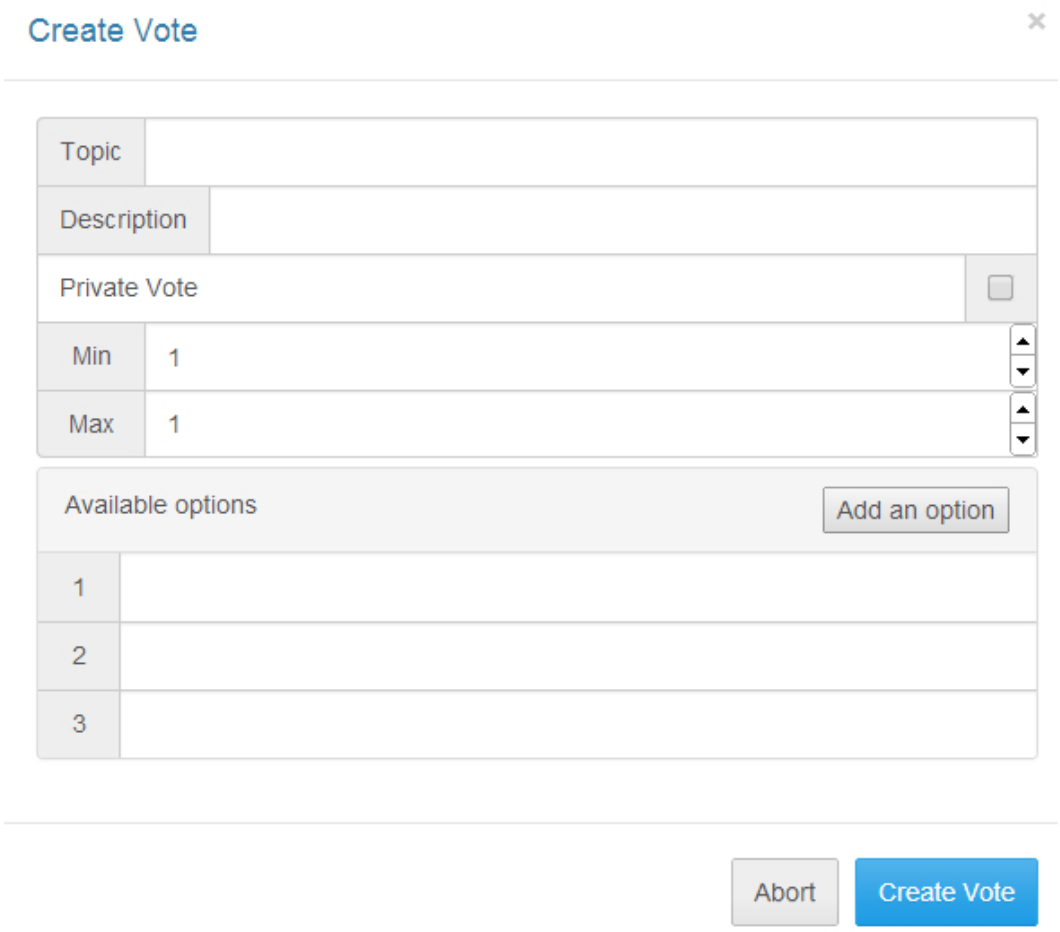


Figure 5.5.43: Create a new vote view

- Get a public vote: when user input a vote ID, ***getVotingsFromReferences()*** is called from background application. If the vote ID exists and the corresponding vote is public vote, the voting object will be shown on the user's vote list. Though the link of dll, user can vote.
- Delete vote: users can delete vote from their vote lists. It will first check if the user is the author of vote. If the author wants to delete the vote, ***deleteVoting()*** is called from background application and the voting object is deleted from storage. At the same time the vote will be removed from the invited users' vote lists. If the user who is not author of the vote wants to delete vote, ***deleteVotingReference()*** is called. Only the link of dll on the vote list is removed, the voting object still exists in storage and it is still accessible to other users.
- Invite users to vote: ***inviteUserToVote()*** is called from background application. If it is a private vote, only the owner can invite users from his friend list, see Figure 5.5.44 below. For a public vote, anyone can invite users to vote. The invitation message is sent via notification plugin.

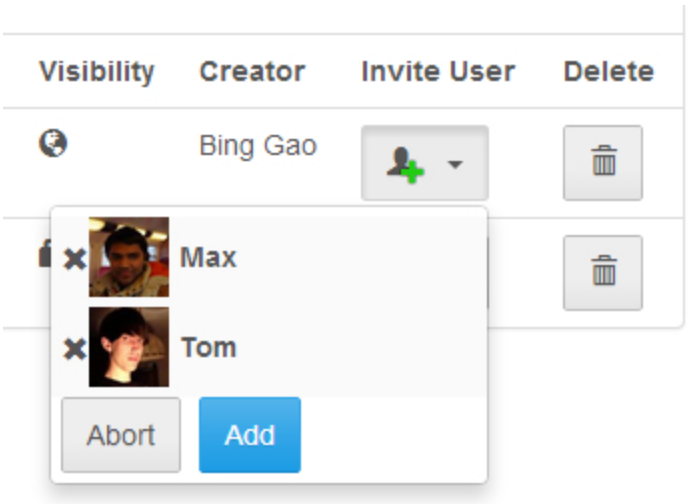


Figure 5.5.44: Inviting users to vote from friend list

- Participate a vote: after accepting vote invitation, user can found the vote on vote list and submit vote options. ***vote()*** is called from backend application and DistributedLinkedList ***votingChoicesDLL*** in storage gets updated after every user votes. After voted, ***setMyVotedId()*** in ***VotingReference()*** is called to update the reference, so user can know that he has already voted and can not vote again. Figure 5.5.45 below shows the participating view for users to vote.

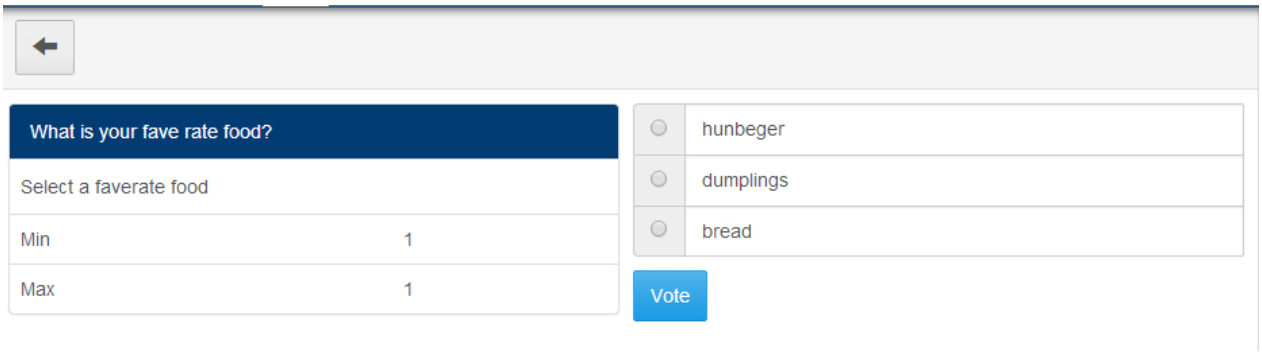


Figure 5.5.45: Participating in a vote

- Vote result: after users insert their votes, the function ***getVotingResults()*** in Voting Plugin is called, the container ***VotingResult*** contains the selected vote options and the proportions they have been selected are displayed on HTML file with Javascript pie chart using jqPlot, as shown in the following Figure 5.5.46.

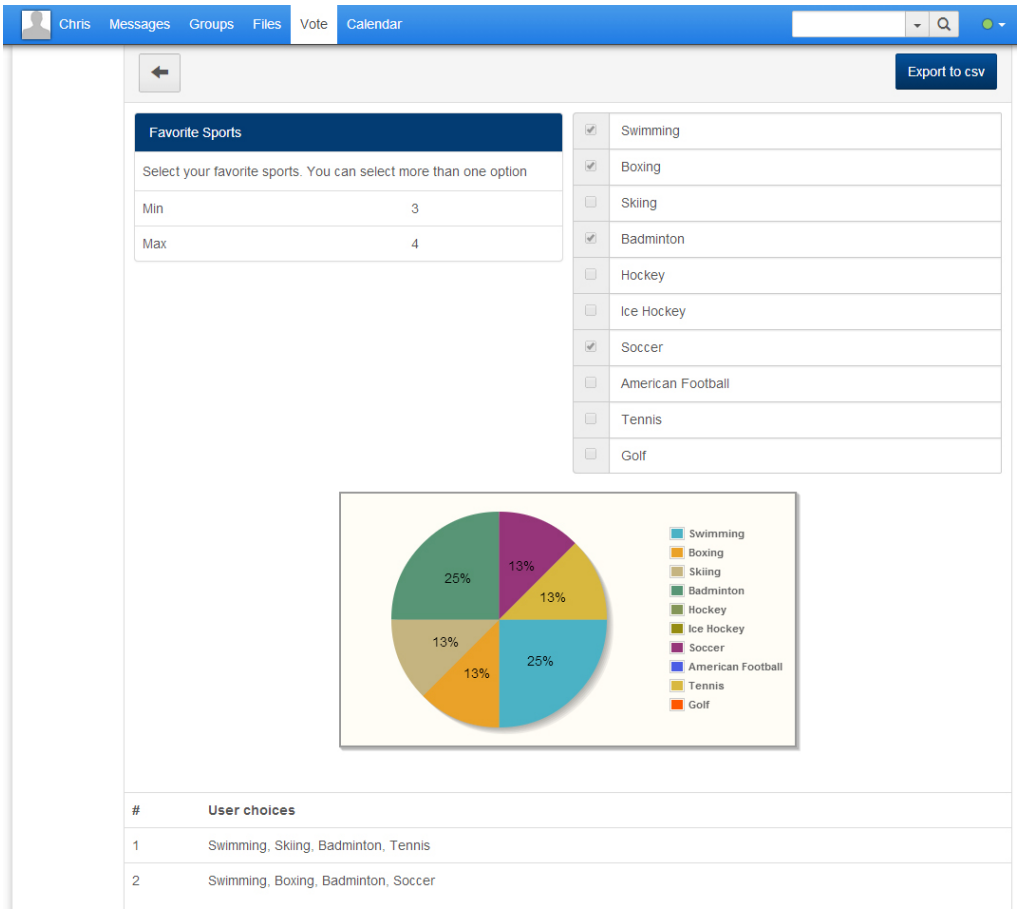


Figure 5.5.46: Vote result shown in a pie chart

Implications and problems:

For the the owner of voting, there is no option to edit a voting again, like changing visibility of the vote, adding more options.etc.

For participants (including the owner), they are not able to change their vote options.

Only users on users' friend list can be invited to public vote, other users can not get access without knowing the vote IDs.

5.5.15. Login Plugin and Auto Login (TJ, FS)

Short overview:

The Login Plugin is extended to support automatic registration of users and to join rings. We use this option on the headless linux instances. Together with a separate shell script, these computers are able to update themselves to new versions of LifeSocial and to re-join the network in a fully automated way.

Adaptations to LifeSocial:

GUI:	**-**
Plugins:	**LoginPlugin:** Added AutoLogin.java
Framework	**-**
Overlay	**-**

Detailed description:

In order to let a node automatically join into a ring, the file autoRegister.conf needs to be placed into the directory <AppDir>/P2PFramework/autoRegister.conf. When LifeSocial is startet, the activate-method of the LoginPlugin will call class AutoLogin to check, whether an automatic login should be used. By default, automatic login is disabled. Only, when the file is present and the corresponding option in that file is set (see below), it is used.

There are basically two mode, a node may be in - Master and Slave:

- **Master:** Is responsible for managing the ring. It is able - when configured - to create a new ring, or to join the last ring it was in at the last logout (by using the bootstrap cache on the local disk). There should only exist one master in the network.
- **Slave:** A slave will always discover the ring, the master is currently logged into and join that ring.

Why do we need a master?

When we update our source code, we distinguish between small changes, which do not affect the storage and big changes. On small changes, the nodes can stay in the ring they are in. However, on a big change, a new ring must be created and all other nodes must join this ring. In order to prevent each node to create a new ring, this is done by the only master node. With a shell script, we make sure that the master is the first node which shuts down and does the update. The other nodes stay online for some more time, which additionally is randomized. This prevents from all nodes joining the network at the same time (which would generate a lot of traffic). Master and slave nodes do not communicate with each other for updating, instead the shell script just checks for updates every 5 minutes by contacting our web-server and randomizes the time when to start the update for the slaves.

How do we propagate new updates?

1. Upload the new deployment of LifeSocial and add a suffix (usually the current date)
2. Update http://p2pframework.de/alpha/version.txt by entering the new file suffix

3. Update http://p2pframework.de/alpha/ringversion.txt to a new timestamp, when you want to do a big update, or leave is as it is when you want to do a small update.

The machines running the shell script will check each 5 minutes for updates. Furthermore, they will monitor the log output and restart LifeSocial on failures of Pastry's Selector Manager. For details, you might want to take a look at the script: http://p2pframework.de/alpha/serverlifesocial.sh

Note that the script needs "dos2unix", "xvfb" and "screen" to run.

An instruction on how to execute the script can be found here: http://p2pframework.de/alpha/info.txt

For the file autoRegister.conf, the following options can be set:

- useAutoRegister - yes or no
- autoRegisterPrefix - prefix of the user name that will be randomly generated
- isMaster - yes or no
- masterCreateNewRing - yes or no
- ringName - the local name of the ring, only for documentation useful
- ringBootstrapMaster - the host from which the salves discover the ring at each start
- localInterface - will be set automatically at the slaves on discovery; only used by the master
- localPort - the network port

A sample file can be found, here: http://p2pframework.de/alpha/autoRegister.conf

Note that when a computer restarts LifeSocial, it will always generate a new username and register that user. Old user accounts are not reused. However, the accounts do not register at the search plugin. This way, the accounts cannot be found when searching for them in LifeSocial.

6. Achievements in the Pastry Overlay

The overlay layer is the bottom layer in the LifeSocial architecture but it is quite important because LifeSocial is entirely based on the Pastry. In other words, this layer can be called as the backbone of LifeSocial. The main function of this layer is to provide peer-to-peer functionalities. During the last project group, this layer has been modified to overcome some issues which were concerned about heterogeneity and security. This time also by our project group, this layer again has been modified to provide/add more functionalities like providing support for NAT (Network Address Translation), providing support for distinguishing different pastry nodes and to make the user aware of the bootstrap nodes.

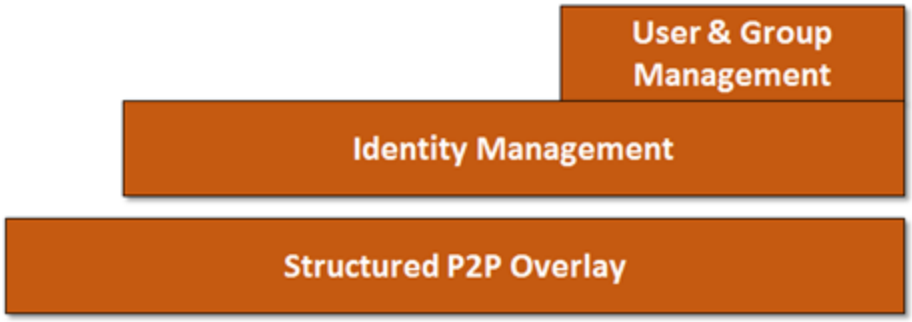


Figure 6.1: Overlay Layer of LifeSocial (PG-2013/14)

Changes to Free Pastry:

In our project group, the Free Pastry has been modified and a new Transport layer has been added to it. This Ring Id Layer contains two methods, **discover()** and **checkMessageHeader()**. The method **discover()** can be used by the user to ask another computers that in which ring they currently are. This helps a user to discover networks. The method, **checkMessageHeader()** is used to check if the incoming message is really intended for that particular ring. It gives an advantage that no two rings can be joined accidently as each message also contains a Ring Id number so it is very easy to verify that which message is for which ring. This diminishes the possibility of sending message intended for one ring to another.

One another major change in the Free Pastry is new Ring Manager which is responsible for storing all the rings that has been discovered and further it is also responsible for caching the addresses of all the online nodes.

These above mentioned changes provide a solid foundation for overcoming from initial bootstraps and ring selection. Till the last project group it was a problem that if, for example, a user U wants to login into a ring R1 in which already there are some nodes available. Further let us suppose that in ring R1, node A is responsible for the initial bootstrap for the user U. Upto this point, it works fine but the problem was when the user logs out and login again into a different ring R2. In that ring, of course user U is not available so that ring R2 rejects that user, saying that user U does not exist. Technically it is correct because user was in ring R1 but at this point user U has no idea what went wrong.

In the current project group, the user knows exactly to which ring he has been connected and

moreover, the user by himself can decide to which ring he wants to get connected. In a nutshell, the user has these options:-

- Create a local ring:- The user has the freedom to create his/ her own private local network and share its details to ask others to join the same, hence creating their own private social networking.
- Discover a ring:- The user can discover a ring based on its IP address and can user that as the initial bootstrap.This ring has to be saved after successful discovery providing a user desired name. After that, the user will be a node of that ring.
- Select/ change a ring:- All the saved rings that the user has discovered or/ and created can be used anytime as long as the user does not delete that manually. Hence the user know exactly to which ring he is connected and moreover, has the complete command on the ring selection.

Another big problem that has been solved by our project group is NAT (Network Address Translation). Till the last project group, this was not implemented and to make a peer-to-peer application run successfully this has to be dealt with. If this problem is not been taken care of, the external nodes will not be able to connect with each other and cannot communicate.
To overcome this problem and to support various NAT strategies, various types of NAT must be identified and then based on the type of NAT detected, the appropriate approach should be followed.
We have implemented two possible ways to overcome NAT problem. They are Port Forwarding and Relaying. Our approaches are not NAT dependent, both of these techniques are independent of the type of NAT. Though we have also implemented STUN to support further NAT strategies. Though we have not used STUN in the current version of LifeSocial but it is always open for the further improvements and adopting it in the LifeSocial when we have to deal with the NAT dependent solutions (for the details, please see the next two sections- Rings and Ring Identification and NAT (Network Address Translation)).

6.1. Rings and Ring Identification (CH, TJ)

Short overview:

Each ring is identified by a unique ring ID. The user is able to discover or create new rings and join them. It is prohibited that two different rings exchange information. Furthermore, while being online in a certain ring, addresses of online nodes are locally cached as bootstrap addresses. This enables the user to login again later, without knowing a bootstrap address.

Adaptations to LifeSocial:

GUI	New login/register GUI

Plugins	
Framework	
Overlay	**Ring-ID Transport layer:** - denies communication with rings not matching the local one - method to retrieve the local ring ID by remote nodes **Ring Manager and Ring Dispatcher:** - manage discovered rings - maintain the bootstrap caches - create new rings - import and export rings configurations (including caches) to disk **RingLifenessChecker:** Abstraction layer to check liveness of nodes and to provide easy to use service methods, like discovering rings.

Detailed description:

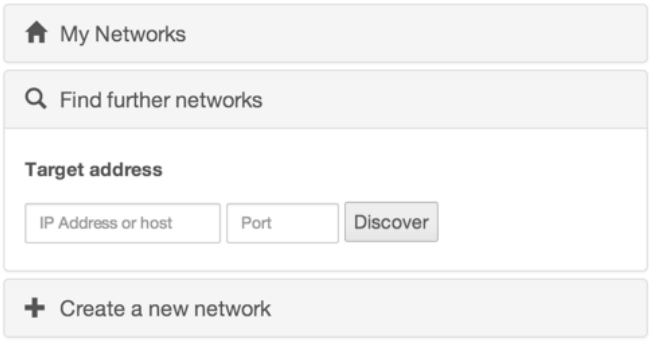


Figure 6.1.1: GUI for Ring Locating

Basically, we support three use cases:

- Create a new network
- Find further networks
- Join a network already in the list of the user

Note that “network” is used as a synonym for pastry ring, here. To support this, first of all the Pastry Network Stack was extended by the ***RingIDTransportLaye****r*. The modified network stack looks as follows, now:



Figure 6.2.2: Modified Network Stack

The two bottom layers and the layers above the new Ring ID layer have not been modified. Traffic not intended for the Ring ID layer is just forwarded. However, each message carries the unique ring identifier (an UUID) the sending node is currently in. When higher layers want to send a message over the network, the Ring ID layer puts this ID to the message header. At the receiving node, the Ring ID layer removes the ID from the header and checks, whether the ID of the message fits to the ID of the ring the receiver is connected to. Messages with non-fitting IDs are thrown away. This avoids a situation, in which two logical different rings exchange information.

Another functionality of the Ring ID layer is to let remote nodes in the internet discover the ring, a local node is connected to. The Ring ID layer provides a special discover-Method. Once called from a remote node, it delivers back the ID of the ring the local node is in. Note that with a “node”, we mean an instance of Free Pastry running on an IP address X and port A. There may be several nodes on a single machine running, which are distinguished by their ports.

The ***Ring Manager*** is responsible of managing all the rings, a user has discovered. Furthermore, it supports to create new rings and manages the export and import to/from the local disc.

A ring consists of the following properties:

- A Ring ID - generated when the ring is created the first time
- A Ring Name - can be chosen by the user and is only locally stored
- A Bootstrap Cache - caches online nodes when the user is connected to that ring
- Local Network settings - outgoing interface, outgoing port, port forwarding settings

Each ring knows how to serialize to XML and how to deserialize back. The ring manager periodically forces a ring to be exported to the local disc. This ensures that updates to the bootstrap cache are persistently stored and that they can be used when the user later logs in, again.

Updates to the Bootstrap Cache work as follows:
When the user is connected to a pastry ring, the corresponding ring object managed by the Ring Manager registers as a callback to Pastrys leaf set and routing table. Whenever Pastry changes the routing table or the leafset, the ring object is informed and can update the cache of online nodes. Note, that in the cache, only public non-firewalled/non-NATed nodes are stored. This is done because we want to make sure that a node in the cache can be contacted directly in order

to use it as a bootstrap node to join the ring.

There are several further building blocks, which support the mentioned features:

- The RingDispatcher keeps track of the currently selected ring.
- The RingLifenessChecker provides an easy to use method to discover remote rings and abstracts away lots of details. When provided with a target address and a target port, it will go through each local network interface and check on which one the target can be reached and then discovers the remote ring ID. Based on this, the local network settings to connect to the discovered ring are configured automatically.

Backwards compatibility:

Before using the ring abstraction, all settings to connect to pastry (basically the local interface and the bootstrap address) were set in Configuration.java. Since the test plugin and the unit tests were not programmed to use our new ring abstraction, but write the settings to Configuration.java, we introduced a debug mode. It is automatically activated and disabled once the graphical LifeSocial interface without the test plugin running starts. This debug mode works quite simple. After the unit tests configured Configuration.java, a debug ring with a fixed, static ring id is created. This ring uses the settings from Configuration.java. The debug ring then is internally used to setup pastry and to connect to the network. It allows us to use the ring abstraction internally without the need to modify our interfaces for backwards compatibility.

Implications and problems:

For joining an existing ring, the user has to discover it. This implies that he knows at least one bootstrap node. Since IP addresses are hard to keep in mind, we support handling DNS names of hosts. However, for later use, having some kind of a public repository with a list of public rings would be nice to have. Furthermore, one could think of a concept allowing private rings. These rings then could only be joined by users having a certain key. Currently we only support public rings, to which everyone can register.

6.2. NAT - Network Address Translation (CH, TJ)

Short overview:

In an overlay ring like pastry, each node takes part with equal responsibilities. It is required that each node is able to communicate with another one in the overlay network. NATs and firewalls prohibit this and we had to implemented strategies to overcome this situation. Basically we support port forwarding on the one hand and relaying on the other. Whereas the first strategy involves manual user interaction, the second one works in a fully automated way.

Adaptations to LifeSocial:

GUI	The new Login GUI provides an interface to enable and setup po forwarding for a selected ring.
Plugins	
Framework	
Overlay	**RingLifenessChecker:** Abstraction layer which can check the lifeness of nodes, which can check whether a node is firewalled (or NATed) and which is able to determine th external IP address of a node. **RendezvousTransportLayer and Rendezvous classes:** Support of the relaying strategy. **InternetPastryNodeFactory (rewritten):** Creates new pastry nodes and does firewall checks before creating a node. Furthermore, checks the port forwarding settings. **STUN (currently unused):** Checks, whether a node is NATed and determines the type of NAT.

Detailed description:

In the following section, we will use the terms “firewall” and “NAT” as synonyms, since a firewall can be seen as a special type of a NAT.

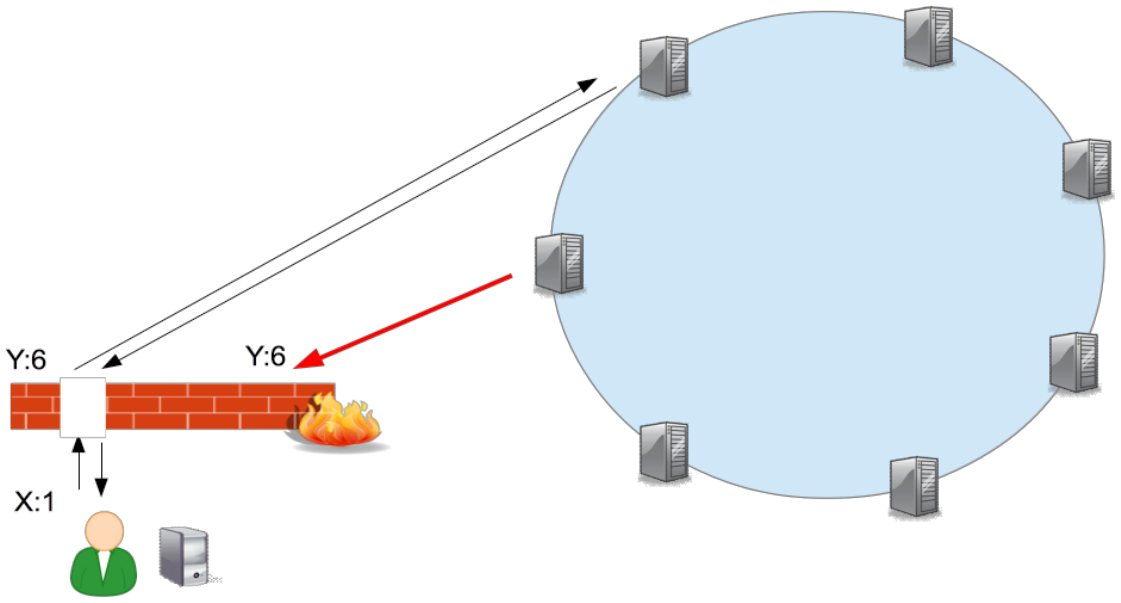


Figure 6.2.1: A typical NAT Environment

The picture shows a typical NAT environment. In such an environment, a router translates internal addresses of the form X:A to external ones of the form Y:B (X and Y IP addresses, A and B port numbers). This is necessary, since internal IP addresses cannot be used within the internet. Now, when an internal computer connects to the outside world, the router creates a temporary mapping. Nodes being contacted can now communicate with the internal node. However, outside nodes not contacted by the internal are not able to do so (note that the exact semantics depend on the type of NAT - you might want to look this up in the web).

Port Forwarding:

The first solution is to let the user manually setup a so-called port forwarding in his router. This is a static mapping forwarding all traffic sent to an external address to an internal one. After the router has been configured by the user to use port forwarding, he has to set up the Pastry ring to use these settings for the local node:

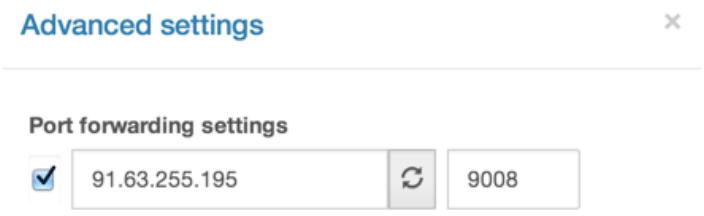


Figure 6.2.2: Port Forwarding GUI

The picture above shows the LifeSocial configuration dialogue for port forwarding. You can find it in the advanced options of the currently selected ring at the login GUI. Basically, the user has to

provide his external IP address and the external port, at which received traffic shall be forwarded to the local address. Since it is quite cumbersome for a user to find out and enter his external IP everytime, we support autoconfiguration at this place. When the field for the external IP is left empty, or when the user presses the refresh button (which can be seen above in the picture), the IP address is retrieved automatically. Figure 6.2.3 shows how this works:

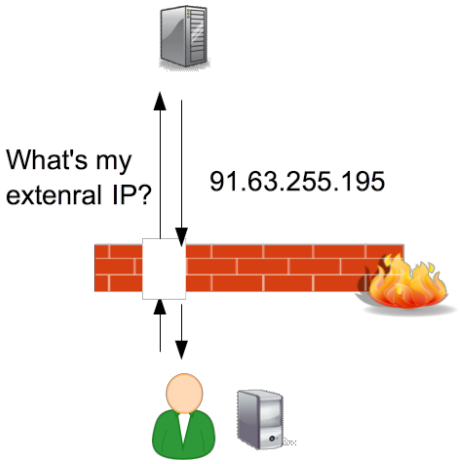


Figure 6.2.3: Port Forwarding Process

The ***RingLifenessChecker*** is used to determine the external IP address. The requirement is that there is at least one public node of the ring, which is online and which is part of the locally stored bootstrap cache (also look at the chapter explaining rings). To this node, a TCP connection is built up by the ***NetworkInfoTransportLayer***. The receiving side handles the incoming socket connection and reads out the address of the node. This address is the external address of the sending node and it is sent back. The functionality to determine the external IP addresses already existed in Pastry before our project group, but it was not working at this time.

After enabling Port Forwarding and when connecting to the ring, the ***InternetPastryNodeFactory*** sets up the Pastry node with both, the internal address and the external one. Other nodes can contact the local node by one of these addresses. This feature was already implemented in FreePastry and we did not have to adapt it.

As a limitation, we currently only support Port Forwarding for nodes behind a single NAT. This could be extended in the future. Pastry already provides interfaces that nodes may have multiple addresses.

UPnP:

A major disadvantage so far is that the user must manually configure his router to use Port Forwarding. When starting the Project Group, we discovered that Pastry already included a module for using the UPnP protocol. When supported by the router, this protocol can be user to set up Port Forwarding automatically. However, we dislike this idea, since UPnP currently does

not provide security mechanisms. Basically every computer in the local area network is able to use UPnP and change the router settings. We believe that the normal user does not want this to happen and we disabled this module of Free Pastry.

Relaying:

To avoid a manual user interaction for using nodes behind a NAT, as a second strategy we make use of relaying. This feature was already implemented in Free Pastry, but was untested and not working so far. We were able to debug it and to get it running during our project group. For most parts, we did not have to change the original code. The central part to support relaying is provided by the ***RendezvousTransportLayer*** and surrounding classes in "*packet org.mpisws.p2p.transport.rendezvous*".

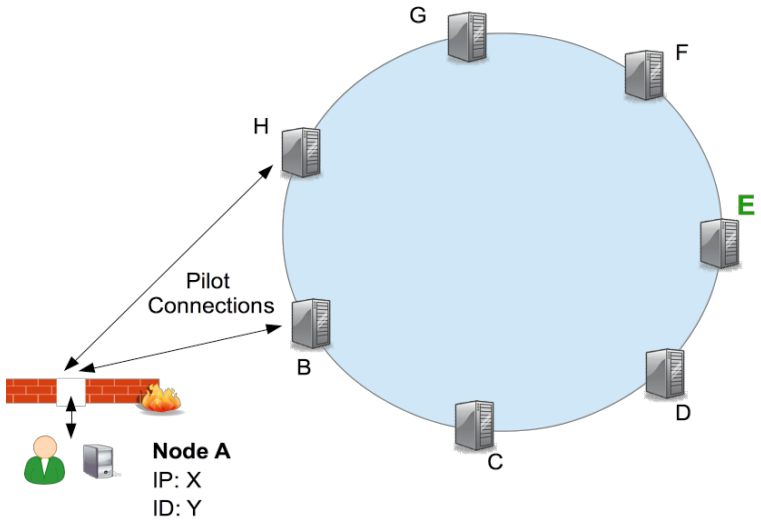


Figure 6.2.4:

When a NATed node joins the Pastry Ring, it builds up TCP connections to non-firewalled leaf nodes. These connections are called "Pilot Connections". The two nodes connected by such a Pilot make sure that the connection stays alive and re-establish it on failures.

Now suppose that in the picture above, node E wants to connect to node A using TCP. Furthermore, suppose for the moment that E is not NATed. Then E does the following checks:

1. Can I contact node A directly? ⇒ If yes, do so.
2. Do I have a pilot connection to A already? ⇒ If yes, use that one.
3. Does a leaf of mine has a Pilot connection to A? ⇒ If yes, use that one.
4. Else, fallback to pastry routing.

In case 1, a direct connection is possible. In cases 2 and 3, a connection request is sent to node A using a Pilot. Node A receives this request and then can establish the a direct connection to E, since E is public. Speaking in other words, the connection establishment order is reversed. Not E

opens the connection (which is impossible), but node A does this.

Case 4 is built on the same idea. We send a request to node A such that it can establish the connection to node E. However, the question now is, how we can deliver the request to node A. The solution is to use Pastry routing. Node E knows the Pastry ID of node A, which is Y. Using this ID, the Pastry Routing mechanism is used to bring the request (closer) to the target. When during the routing process, a next hop cannot be contacted, another next hop is chosen. This is determined by the *RendezvousRouterStrategy.* Finally, the request will be received by a leaf of node of node A, and the pilot is used to deliver it to A. This way, A can establish the connection to E.

A problem occurs, when node E is NATed, as well. In this case both, node A cannot establish a connection to A and A cannot establish a connection to node E. This is solved as follows:

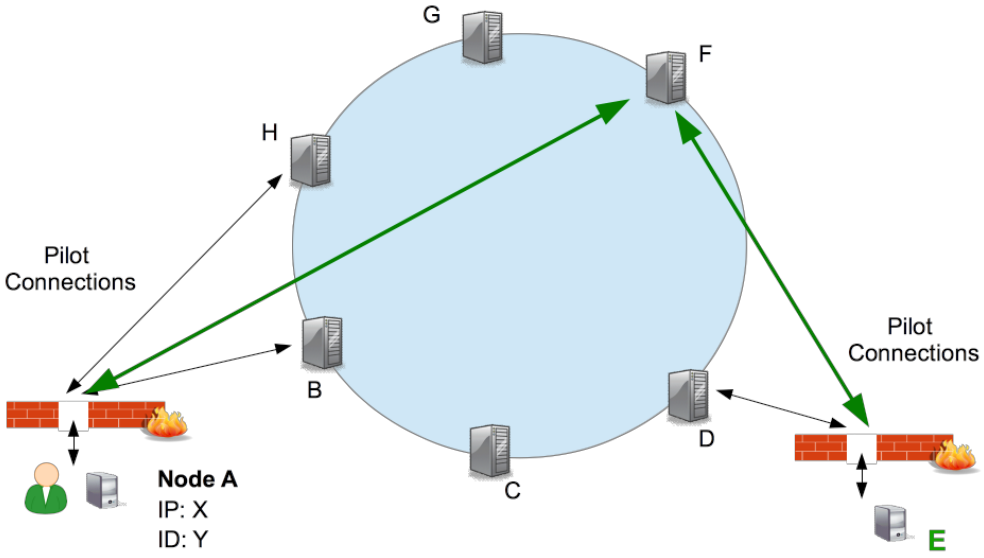


Figure 6.2.5:

As node E has outgoing Pilot connections to public nodes, it picks one at random. In this case, it picks the pilot to F. F then behaves exactly as explained before. All data transmitted between E and A is tunneled over F. As an implication, public nodes may get overloaded. To improve this situation, randomization is used for choosing an outgoing pilot as a relay node.

So far, we only discussed, how TCP connections can be established. Pastry also uses UDP for maintenance and for sending messages from an application, which are explicitly declared to use UDP (take a look at Pastrys ***PriorityTransportLayer***). UDP is a connectionless protocol. This his certain implications. When an internal node sends out an UDP packet to an external node, this external node can not necessarily reply to this message. It depends on how long the router will keep up the dynamic NAT mapping. The *RendezvousTransportLayer* assumes the dynamic mapping being online for 2 hours. When an UDP message is received, the sender is stored to

the ***EphemeralDB***. Within the mentioned timeframe, Pastry assumes that it can deliver messages directly to the origin. This may be false assumption and the message can get lost. Thus, you should use TCP for important messages. Unless explicitly stated to use UDP, this automatically is the case (take a look at the PriorityTransportLayer). But what happens, when a message shall be sent, the receiver of which is not in the EphemeralDB, or the timeframe has passed? Basically, then TCP is used to deliver the message to the target or to a forwarder. Either by using a pilot as a forwarder of the message, or by Pastrys routing mechanism. The details are handled by the ***RendezvousApp***.

Putting the strategies together:

Responsible for setting up the Pastry Node and for doing the firewall check is the ***InternetPastryNodeFactory***. The set up procedure works as follows:

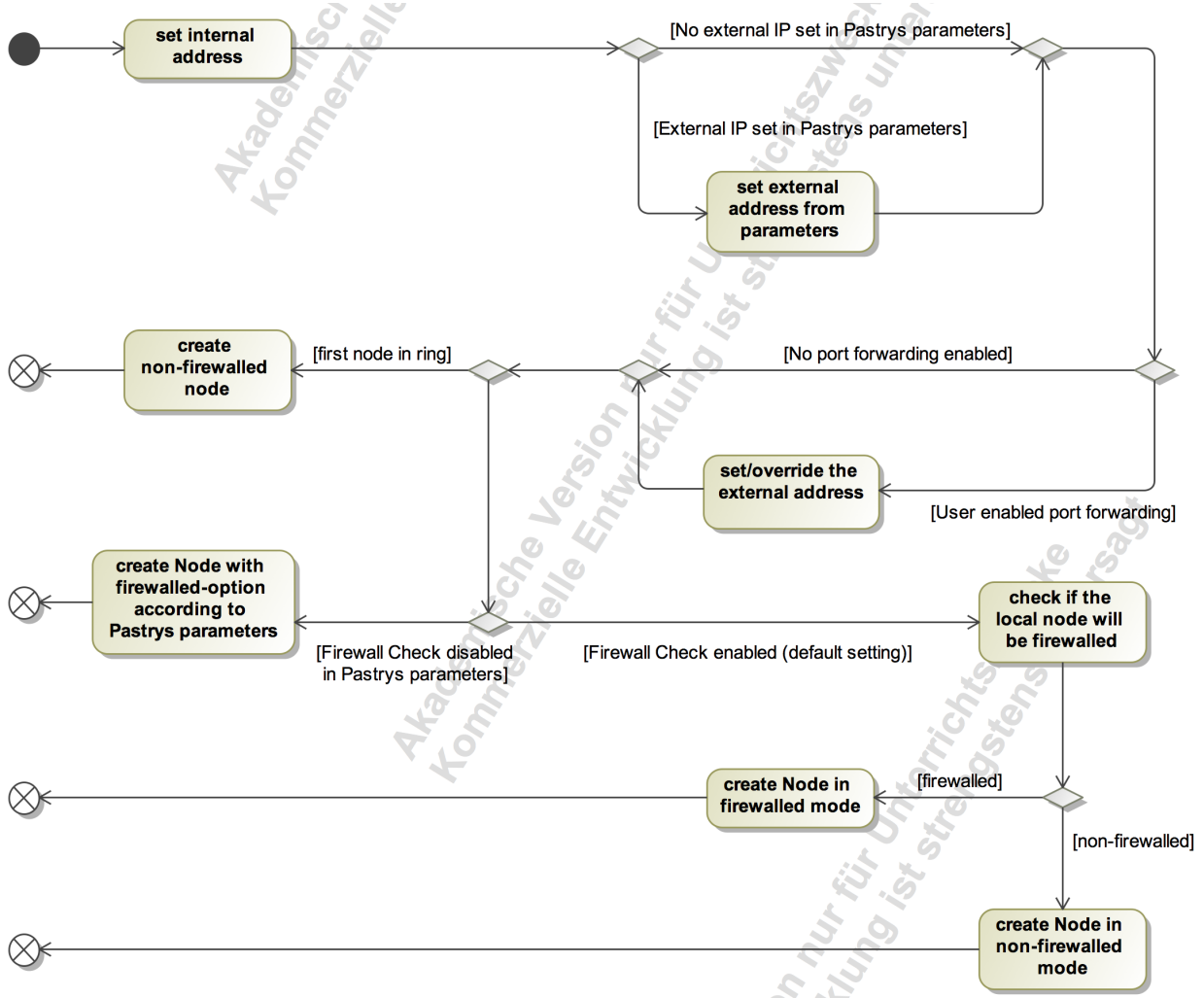


Figure 6.2.6:

The workflow is straightforward. The only question to answer is, how we can determine, whether a node is firewalled. Well, for this, we use the ***RingLifenessChecker***, which in turn makes use of the ***NetworkInfoTransportLayer*** and the ***ProbeApp***. The basic idea is as follows: The factory first creates a temporary node and builds up Pastrys lower transport layers until and including the NetworkInfoTransportLayer. Then, a packet is send to another public node in the ring. This packet is then forwarded to a third public node (which is managed by the ProbeApp). This third node then sends back the packet to the origin. When the packet is received at the local temporary node, we are not firewalled. When it is not received, we are firewalled. This is tested for both, UDP and TCP. Since our tests showed that UDP messages often were lost, we send out several UDP packets. When at least one of these packages is received, and the TCP connection worked as well, we assume a node to be not firewalled. As a special situation, we also considered the case where there is only a signle public node A in the ring. In this case, the packets cannot be forwarded to a third node. In this special situation, A directly replies to the origin.

STUN:

The STUN protocol is used to detect the type of NAT, a node is behind. We implemented it according to the corresponding RFCs. However, we currently don't use it, since we have no NAT-type dependant traversal strategy. We can use it in the future, although.

Implications and problems:

We are now able to use LifeSocial in real environments. However, we need some further strategies, which allow to overcome the NAT problem. The relaying strategy is functional, but it requires that for one firewalled/NATed node, at least one public node is in the leafset to relay the traffic. This may imply bottlenecks. As a prerequisite to support NAT-type dependant strategies, we already implemented STUN. Another problem is that we had trouble in getting the parallel and iterative routing to work (which was implemented in the last project group). Currently, we replaced the Java classes involved with the original ones of Free Pastry. In the future, this has to be reverted again and has to be debug. We could not afford this in this project group since we discovered this problem quite late and it only occured non-deterministic. Another thing to consider is that Pastry currently only supports hosts behind a single NAT. This has to be extended in the future, as well.

6.3. Secure Replication and Modification of Items (RS)

Short overview:

Each data item is associated with a key. Whenever a lookup for the data is requested, the data is found in the networks by requesting the nodes having the data for the key. PAST, the replication layer in pastry ensures that certain number of replicas are maintained for any data key found in

the network. Whenever a request for the data item occurs, the node with the key for the data item replies to the request.

Adaptations to LifeSocial:

GUI	
Plugins	
Framework	**Framework storage** ● com.p2pframework.storage.past
Overlay	**PAST replication layer** ● rice.p2p.past ● rice.p2p.replication ● rice.p2p.replication.manager **Pastry persistent storage** ● rice.persistence

Detailed description:

Secure replication:

The replication manager, provided by PAST runs replication maintenance operation on defined time intervals to ensure that for any data item with key K, R replicas are maintained. To ensure that valid data is replicated in the network, security check was implemented, where the signature of the data replicated is checked before storing the data in the replicas. Each data object stored by the persistent storage is of type ***PastContent.*** For each and every data managed and stored in the persistent storage, interface methods are provided to ensure validity of the data. The ***PastryStorageItem*** implements the interface methods provided by ***PastContent.*** To verify the signature of the data item to be replicated, we use the method ***checkInsert(Id id, PastContent existingContent).*** If the insert is allowed and the data signatures are valid, then the method return the content that can be safely replicated to the replicas in the network. Every replica, when receives a replication request, it first checks the data validity. On successful validation, the data is stored in the replica's persistent storage, otherwise the replication request is discarded temporarily and the data is not stored in the replica's persistent storage.

Modification of items:

When a new data item is inserted into the storage a key or id for the inserted data is generated and mapped using DHT. However, this replication strategy considers maintaining the the data for key K in R replicas, it does not take into account the change in data for the given key K. This replication scheme causes the data to be inconsistent after an update to the data since, the

required amount of replicas for key K is still available in the network. As a result of this, some of the replicas have different data. To overcome this, the workflow used was modified.

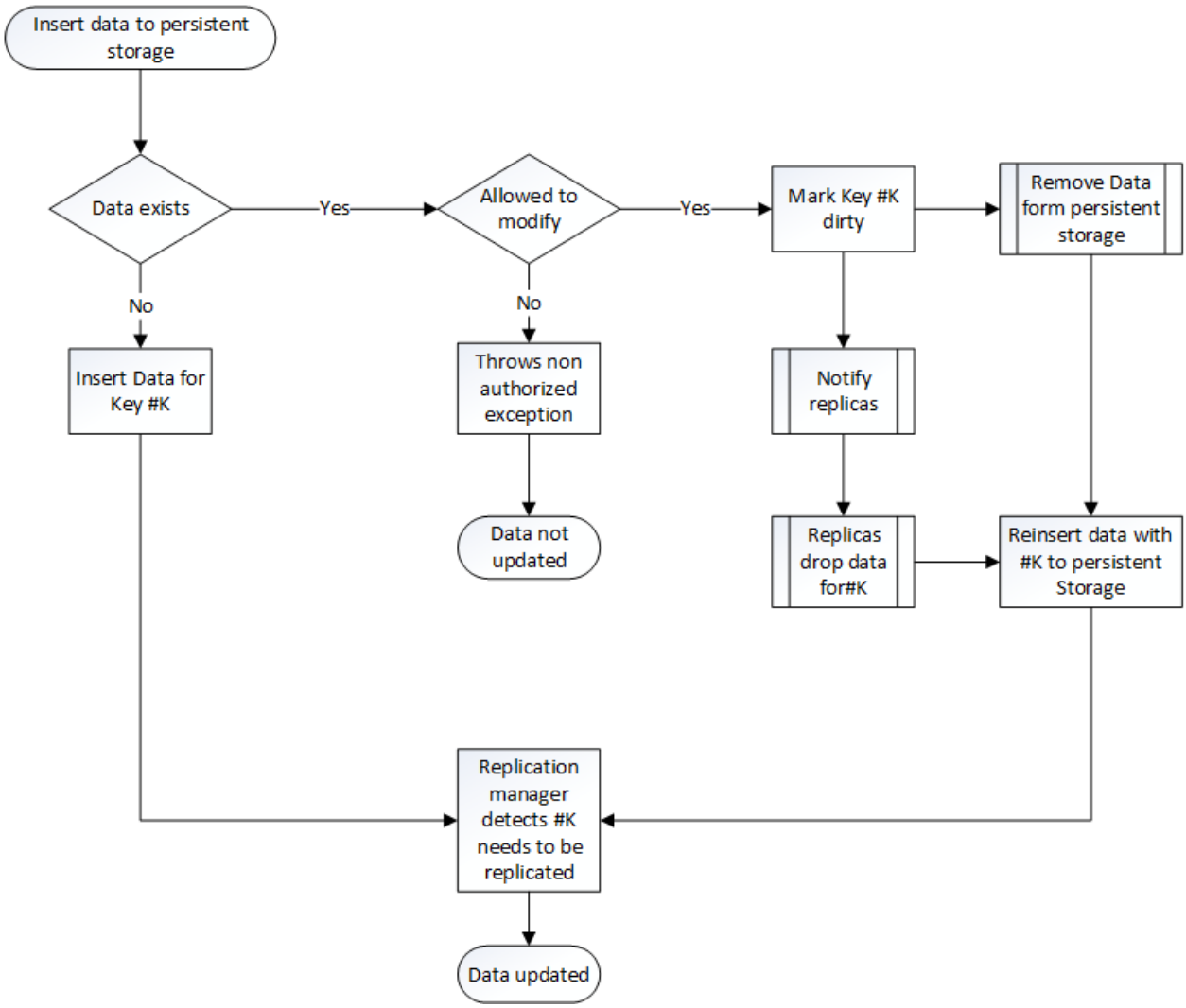


Figure 6.3.1 : The basic flow chart for the replication workflow implemented

In the modified workflow, shown in figure 6.3.1 ensures consistency of the data in the network. Whenever there is change in data for key K, first a check is made to ensure that the node is allowed to change the data. If allowed, the key K is marked as dirty and dropped from the DHT. Once this operation is finished, we initiate replication maintenance operation by calling the method ***replicate(),*** provided by the PAST replication manager. The maintenance operation runs as a parallel thread where, all replicas with key K is notified to drop the data for the key K and instructs the replicas to fetch and store data for the remaining keys in the DHT, so that R replicas are maintained in the network. Parallely, the main thread then removes the data from the pastry persistent storage and performs data validity checks for the modified data to be inserted. Once the replication manager has performed the maintenance operation, the modified data is inserted into the pastry persistent storage and the key K is reinserted into the DHT. Whenever, there is a

request for data with key K, the requesting node will receive the updated data. The replication manager, when performing the maintenance operation next time, will ensure that the reinserted data for key K is replicated to R replicas in the network. Hence, the implemented workflow ensures data is consistent in the network and replicas are updated correctly.

Implications and problems:

The secure replication implementation does not inform the replication manager that the content has failed signature validations. We catch this exception and informs the replication manager that data is stored in the replicas persistent storage. However, some of the replicas might still not have the data stored. So when the lookup for the data occurs, the request is redirected to the responsible node. The replication manager next time performing maintenance operation finds that the data is not stored in R replicas and tries again to replicate the data.

Present implementation for updating data requires the key to be dropped and to force replication maintenance operation for ensuring data consistency. The replication manager does not provide any callback methods or notification that the maintenance operation is completed. The maintenance operation runs as a parallel thread. Hence, an assumption is made that the replication manager can finish the maintenance operation by the time the data is removed from persistent storage and the validity checks on the data is performed before reinserting the data to persistent storage. Better approach would be to handle the data update without key invalidation, dropping and reinserting and without forcing a replication maintenance.

6.4. Signing and Java Serialization (CH)

Short overview:

In the previous versions of LifeSocial the encryption of data was not used because the signing failed. During the work on a fix it became clear that this problem existed in other classes of the framework, as well.

Adaptations to LifeSocial:

The following table shows in which classes the signing bug existed:

GUI	
Plugins	
Framework	● com.p2pframework.storage.SecureStorageItem ● com.p2pframework.storage.past.PastryStorageItem ● com.p2pframework.DistributedDataItemContainer

Overlay	

Detailed description:

The reason why the signing process failed was that the Java Serialization was used in a wrong way. We use serialization to send data over our network.Therefore, we take a short look on how Java Serialization behaves, which is illustrated in the figure 6.4.1:

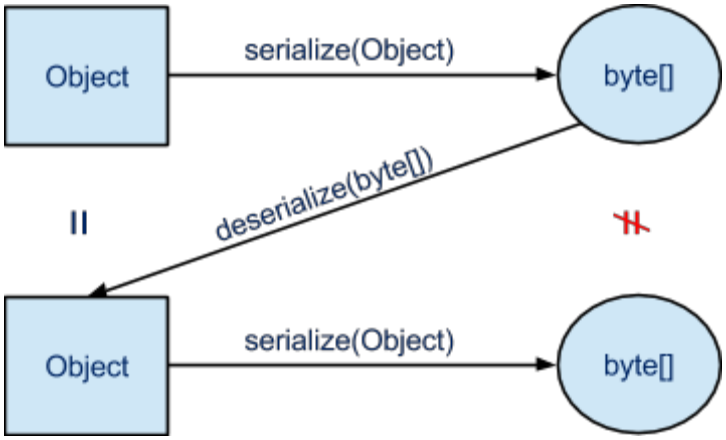


Figure 6.4.1: Java Serialization behavior

The first step is to serialize an object to a byte array. This array is then sent over the network and received by the intended target. The target then deserializes the byte array to an object again, which results in the same object which was sent. The important point is when we now try to serialize the object again the resulting byte array is different from the byte array before.

When we now take a look into the signing and how it was done before, we have exactly this situation. In the figure 6.4.2 the data is serialized twice and is used to create and verify the signature. This cannot work, because the byte array may be different and the signing fails in this case.

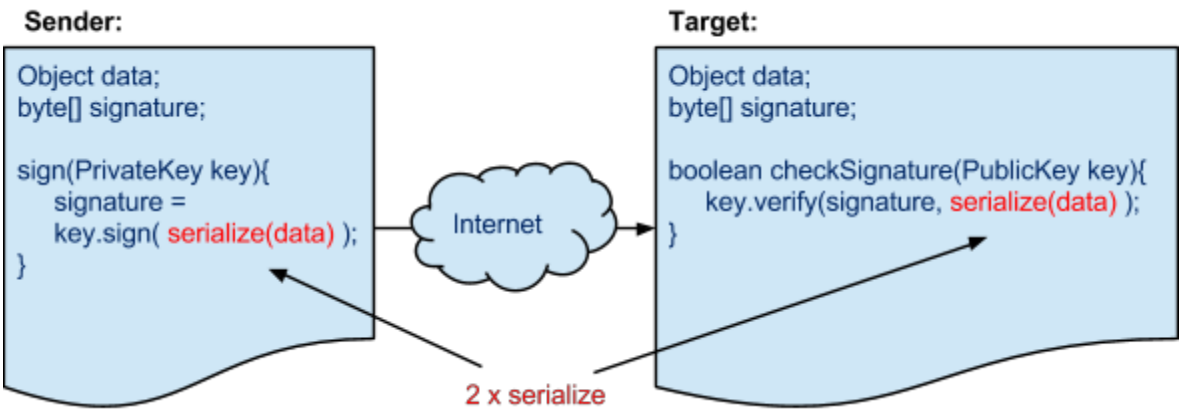


Figure 6.4.2: Signing bug

To remove this problem we wrote a fix which only serializes once. The figure 6.4.3 illustrates how this works. One point is that we do not send the data anymore. This is solved by defining the ***data*** variable as transient. This means that Java Serialization ignores this field. The next step is to serialize the data and assign it to a new variable which we call ***serializedData*** in this example. The rest works as before but we use the new variable to create the signature and on the target side we use this variable to verify the signature. Because of the transient modifier the ***data*** variable is null we now deserialize the variable ***serializedData*** and store the result to the data variable. The result is that we now do not serialize twice, since we use the same byte array. This way the signature check succeeds.

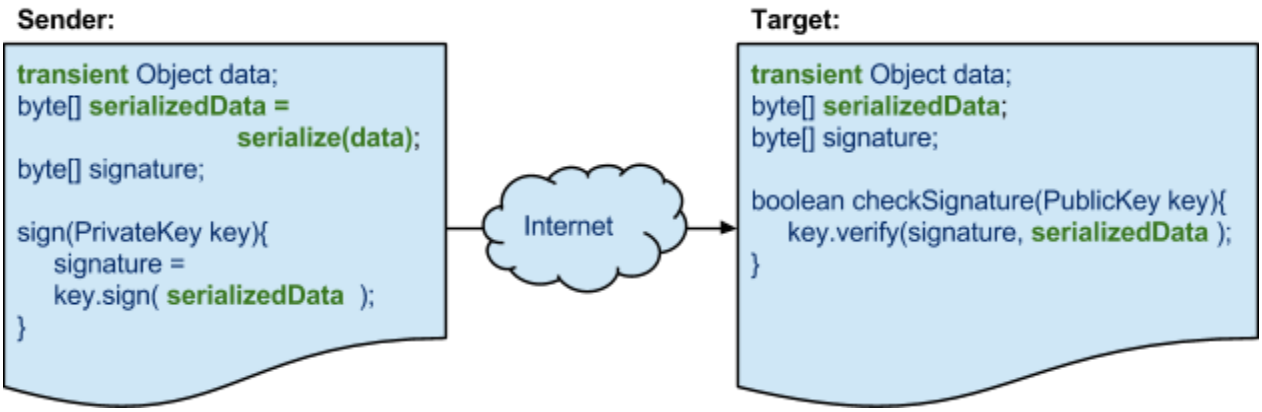


Figure 6.4.3: Signing fix

Implications and problems:

Due to this fix the signing now works and we are able to use encryption in LifeSocial and items do not get lost because of this problem anymore. But there is still another problem in the ***DistributedDataItemContainer***. It seems to be related again to a signing issue but due to the lack of time we were not able to resolve this problem. We could find out that they use different

fields for the signing of this class, which seems to be the reason for that problem. This issue results in a ***rice.p2p.past.PastException*** with the message “not authorized”, which seems to do no harm but should not happen, because in our test cases the involved instances had the access rights to read the items.

7. Open Topics and Future Work

This sub-chapter gives an overview of open topics and goals which were not reached. Since we decided to implement a complete new GUI, most open topics were not reached because of time limitations. Furthermore, often parts of a proposed goal were implemented, such that basic support is already working, while advanced features are still left open.

7.1. Proposed improvements:

Profile:

The profile plugin does not allow any mechanism to reset the password. Currently the profile is encrypted and stored using the password. So we need a mechanism to safely reset the password.

Search:(CJ,BG)

The profile fields are indexed in the query engine for searching when the user is registered in the network. But if the user has blocked another user , the blocked user can still be able to search for the user. The improvement has to be made in such a way, if an user is blocked by another user, the blocked user should not be able to search for the user who blocked . The query engine indexes the private fields of the user which should not be searchable.

The search plugin is currently restricted to search users in the network. It should be extended to search for public votes and public groups.

Votes:(BG)

If there is no option on choices list which is suitable to user, users can click on “input other answer” and input their own choice.

Friend Graph:

Friend Graph is disabled and buggy. It has to be incorporated in HTML GUI.

Groups:(RS, MV)

Access control mechanisms can be implemented which allows only authorized users to read and write data items present in the network. ACL for group owners can be implemented where they can delete or modify the files or folders. This can be implemented using cryptographic keys. For read access in the framework, the data can be encrypted with a symmetric cryptographic key, which can then be made available to the authorized users, to a user group or a hierarchical

user group. The symmetric key can be encrypted with group's public key or the user's public key. To access the data item for reading, the symmetric key can be decrypted and is then used to decrypt the encrypted data item. The symmetric key can be decrypted by using the user's private key or the group's private key. The write access to data can also be implemented using signatures. The public key of the owner or group can be stored together with the item when it is first inserted. The data can only be overwritten if it is signed with the corresponding private key belonging to the specified public key. This signature can be verified by the node at which the data should be stored and whenever it is not valid, the storing node can simply deny the insertion of the data into the network.

Private groups: At present, only public groups are being implemented in LifeSocial. Private groups can be implemented where the users join the community or group by invitation of the group owner or by sending a request to the group owner.

Replication:(RS)

The exception occuring while verifying the data signature is not informed to the replication manager. The exception is caught and we inform the replication manager about the successful replication. One among the main reason why data signature validation fails is that, the data may be bogus or corrupted and this data should not be considered for replication after some pre-defined tries. This implementation is lacking, which can reduce the network traffic and storage overheads.

Calendar:(EG)

A user does not have an option to set a date or time for event reminder. As an improvement, user should be able to set a date or time for event reminder for e.g. a day before or 3 hours before the actual start time. In order to receive event reminder, user has to log out and log in again. This does not work in the same session. In future, a user should also be to share his calendar with other users for e.g., a professor can set lecture dates and shares his calendar with students. In this case, students should only have read rights for which acl should be implemented. Furthermore, a user should also be able to invite other users to his event for e.g., a user creates an event for his birthday party and invites other users. Here, the invited users should have an option to accept and reject the invitation.

7.2 Proposed new features:

NAT:

Need to support NAT-type dependant strategies to avoid using relaying, which impacts the network performance. Furthermore need to consider nodes behind several NATs and need to

test whether the communication works.

Modification of items:(RS)

Present implementation for updating data requires the key to be dropped and to force replication maintenance operation for ensuring data consistency. Additionally an assumption is made that the replication manager can finish the maintenance operation in the by the time the data is removed from persistent storage and the validity checks on the data is performed before reinserting the data to persistent storage. Better approach would be to handle the data update without key invalidation, dropping and reinserting and without forcing a replication maintenance.

E-Learning:(RS, MV)

1. Group calendar - Integrating calendar into groups enables collaboration and sharing of schedules amongst groups. It allows users to send meeting requests and tasks to other users. Users can add, edit or delete events, whenever they want to. A user would also be allowed to share his own calendar with an individual or a specific group. For e.g. an instructor adds exam dates to his calendar and then share this calendar with student group, now the group can view the instructor's calendar. The owner of the calendar can change the event date or can remove the event whenever he wants to. But, the viewer's only have read permissions on the calendar.

2. Group voting - Integrating voting in groups is a useful functionality. The vote or poll should be created and accessible only to the group members. The group members could see the results of the votel. Only the owner of the vote can manage the vote i.e. to remove, stop or add the votes.

3. Live group video - BitTorrent can be used for live group video as the members in the group can start seeding the video and tracker can be used to maintain the peers online.

4. Sub-groups in groups (ACL's) - Implementing access controls in groups is important. Currently LifeSocial offers group management through GroupManager plugin which allows users with common interest to meet and contact with other. The members of a group are able to create forums to exchange information. To provide a better overview, it is possible to structure the forums into folders. Each folder in a group can be seen as a sub-group and each sub-group can have its own forums and sub-groups.

   If we take the scenario of E-learning, the E-learning community can be seen as group for each lecture in a university/college/school. The functionality of such group is to provide functionalities of course management. The lecturer, students can use it for sharing lecture materials, assignments, tutorials etc. Access controls here is significant. The lecturers have read and write controls to the folders provided for uploading lecture materials, tutorials and assignments. Students can only download the uploaded document from these folder or drive. Additional access controls are required for restricting

the read and write permissions to the marks obtained by the student. The lecturers has read and write permission to the students mark lists, but only the student himself can view his/her only marks not any other students marks.

5. Administrators for the groups - For social community, the group created can be public or private. In a private community the users can join the community by invitation of the group owner or by sending a request to the group owner. The functionality of assigning administrators to community would help in the efficient management of the private community. In E-learning community, private group can be created where the students should be added to the community with read only permissions on the lecture materials uploaded to the community. The community owner can be the lecturer or his/her assistant.

Video Streaming:(CJ)

In future, if anyone want to upload a recorded lecture video or live stream a video bittorrent protocol can be used instead of storing in the network. Tracker can be maintained in order to keep track of the peers online and peers can start seeding. As long as one peer is online ,the video is being offered.

Video streaming or Live streaming can be implemented using BitTorrent protocol in HTML 5. The following method uses the swarm player for streaming the video. This method was used by wikipedia to stream the videos in Peer-to-Peer based distribution.

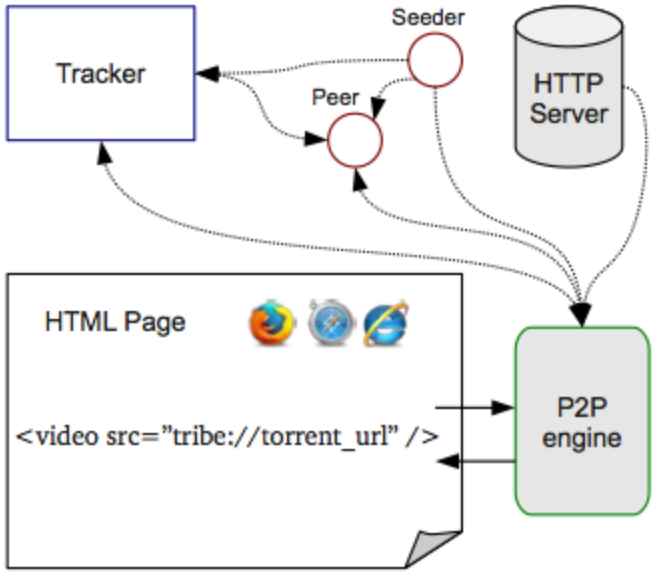


Figure 7.2.1:

The <video> tag in html page can be modified to implement the bittorent in browser. Kaltura HTML5 library adds support for html5 streaming bittorrent via p2p next swarm player.

Bit-Torrent (KB):

Bit-Torrent is a widely used protocol for file sharing over the network in an efficient way where a single file is effectively accessed by large number of users. Bit-Torrent is based on peer to peer networking (P2P) and it improves the transfer speed drastically by splitting the file into segments and collecting the segment needed from the other peers that already have them. The technique shows effective resource utilization among hundreds of connected peers accessing the same file by not compensating on their download speed.

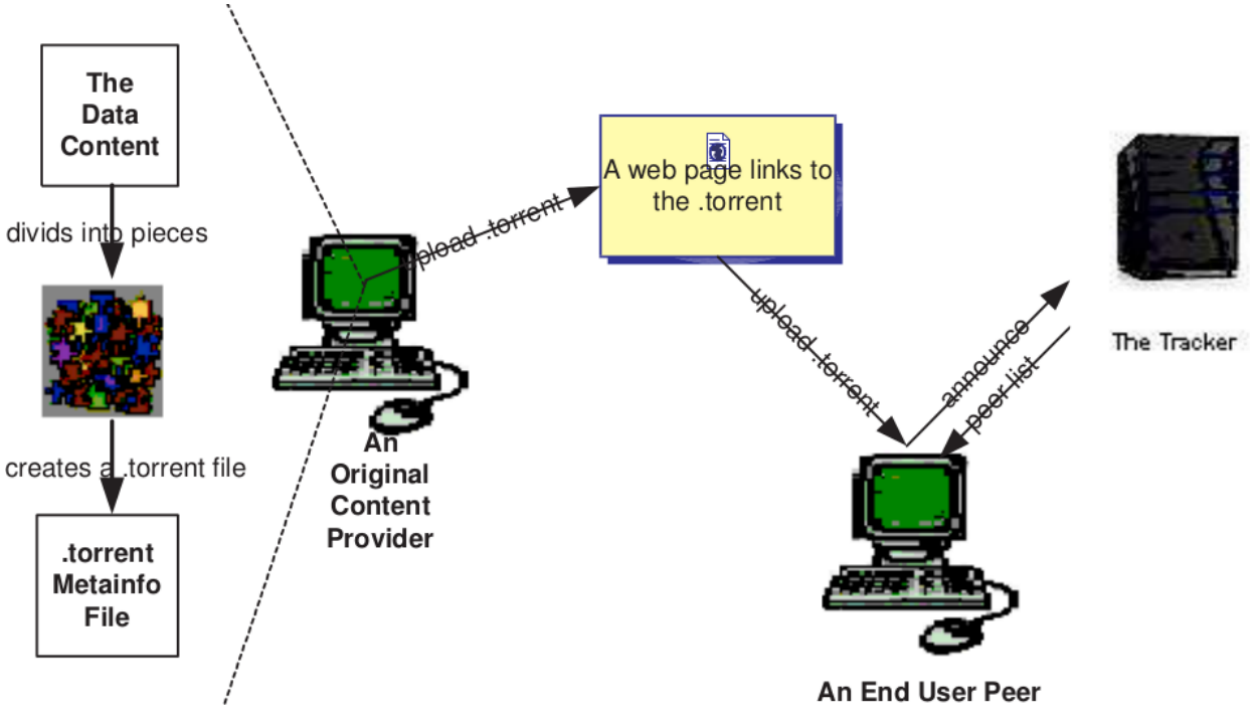


Figure 7.2.2: An Abstract view of Bit-Torrent operation.

When a user decides to share a file using bit-torrent, the peer distributing the file uses torrent client to chop the file into identical sized pieces, creates hash for every single
piece and is recorded in the .torrent file. Peers that has the complete file are called seeders, and the peer providing the first copy, generally the torrent owner, is called the initial seeder. These Torrents have an ”announce” section where URL of the tracker is mentioned followed by an info section containing name of the files, number of pieces, their size and hashes for each piece. These are verified by clients to check integrity of the data that they receive. Torrent files are typically uploaded on websites such as index sites, with at least one tracker. The tracker maintains lists of the peers currently participating in the torrent. The initial seeder who has the complete file should start seeding to make the file available for other connected peers to download.

But the challenge to implement Bit-Torrent in P2PSocial is that we do not need central tracker that acts as a server. We need to have a decentralized server. It can be achieved by using Distributed Hash Tables (DHT) where DHT present in each nodes handles all information about

file segments and its associated swarm of nodes having those segments.

Collaborative Work: (KB)

The groups plugin can be extended to adapt to a collaborative environment with an addition of text editors and white boards to model the E-Learning environment. Users should be able to produce works together through individual contributions on the same view at the same time. This plugin enables multiple users to work simultaneously on single text document or a whiteboard by using a Changeset algorithm.

Files: (BG)

The Groups plugin does not provide any mechanisms to share the existing files using Files plugin. The mechanisms has to be incorporated to make it possible to share the files.

Secure Monitoring

Secure Monitoring is not implemented.

Specifying(@) user: (BG)

When there are multiple comments below a user's post, and the user wants to reply to some comment, it should be possible to specify to which user do you want to reply.
@usernname should be possible with a link to the user's profile. For example:

UserA: Hello world !
B: Hi!
C: Morning!
D: Where are you?
UserA: @D: I am in Hannover Messe :)

This can also work when user want to reply to some comment of his photo. The specified user can be informed with a notification message.

Birthday Reminder: (BG)

Birthdays of friends can be shown somewhere on user's own profile when friend's birthday is coming.

Tagging: (CJ)

Tagging can be done when uploading the photos by selecting the person in the photo and tag the users. The tagged users also should get a notification when they are tagged in any photos.

App Store: (KB)

App store acts as an online repository where users can publish their plugins that can be downloaded. This way we intend to attract users to develop and publish their apps(plugins), thus enriching P2PSocial's app store and its user base.By creating simple OSGi bundles, it can be loaded and unloaded without disturbing any other part of the framework.The App store supports users of two different roles; Plugin Provider and Plugin User.

Plugin Provider:

The author of a plugin can take up the role of the Plugin Provider. The author is permitted to edit, publish new versions or remove the plugin later. We need a different GUI for Plugin providers where they can manage the plugins developed by them and also the plugins that they use as a normal user.

Plugin User:

Users of a plugin become the Plugin User. They have only read rights. As normal user, they can do management of the installed plugin such as remove or update.

Mini Games(CJ):

We can also create mini games like TicTac Toe as a simpe plugin and it can also be added to appstore.

7.3. Further Topics to be addressed:

A requesting node delivers messages to k different peers , which respond with the best routing entries, the messages should be forwarded to. The requesting node waits for the answers and then iteratively do it in the same way, forwarding the messages to the k nodes,which in turn respond with the best routing entries. As of now,the iterative/parallel routing is switched off, as they generate some trouble in high churn environment. This issue has to be identified for any security leaks and against potential attack on the framework.

The issue remote parallel operation takes long time to respond , so it has been modified to timeout after 5s for search plugin. Better mechanism is required for optimizing the searching when there are no search results available.

Some parts of the code contains the old GUI code which has to be refactored and removed from the project.

8. Testing

This chapter describes the features of several test components (Unit Test, Test Plugin, Alpha Test and Monitoring) and the relationships among them. Unlike Unit Test which is designed to verify the functionality of single part of the system individually, Test Plugin is used to test the system as a whole. Test Plugin enables the master node to ask each slave instance to execute certain behaviors based on a test plan. And Alpha Test, usually, is for some other users instead of developers to use the system in order to find out potential bugs. Last but not least, Monitoring enables developer to collect the data generated during the testing and improve the system's quality of service.

8.1. Unit Test

Unit Test, as mentioned above, is used to test the correctness and robustness of a specific component of the system. Usually, a unit test case needs to be well-designed. For a certain functionality, all possible representative input need to be examined, and code coverage should be considered as well. And each unit test case is separate from another, which is good for finding out problems and fixing them.

Ideally, every single functionality should have a corresponding unit test case created by the person or team who implements it. But in the last year, our project group modified the system considerably, therefore a few old unit test cases are abandoned since they are not fit for our new system any more. And the unit test cases for some modified or new feature have yet to create due to time issue. In summary, most functionalities of the system are covered by the existing unit test cases.

8.2. Test Plugin

Different from Unit Test, Test Plugin enables tester to test the entire system so that the performance and reliability of the system can be examined. Basically, there are two kind of roles in Test Plugin. One is Master which is unique in a network. And the rest instances in the network are assigned with the other role, namely Slave. In simple terms, the master initiates a test plan which contains a sequence of test cases to be tested, and sends the test plan to certain or all slaves. Based on the test plan, slaves execute corresponding behaviors under some constraints.

Graphical User Interface (GUI) is created to use Test Plugin relatively easy. As shown in Figure 8.2.1, tester can see online users, registered user, and their bootstrap server clearly through the Master View. More important, the TestCase Delegater in Figure 8.2.2, enables the master to send the test plan to certain or all slaves directly, which is very convenient to use. Besides, master also can send the individual test case to slaves via the TestCase Delegater.

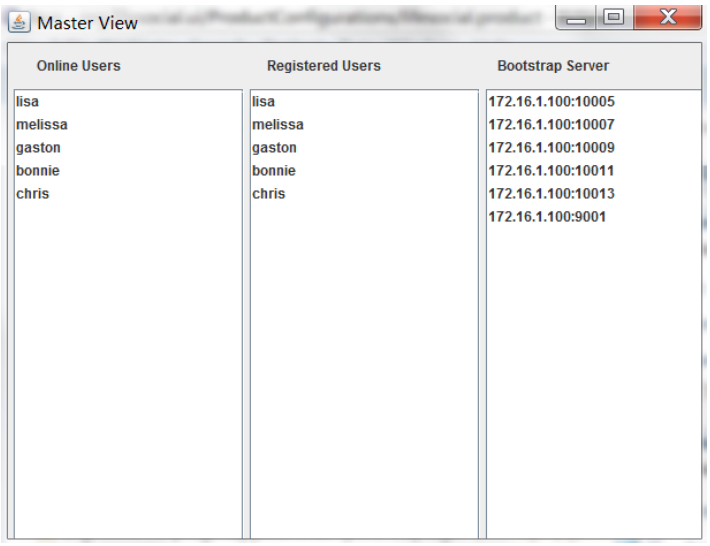


Figure 8.2.1: Master View

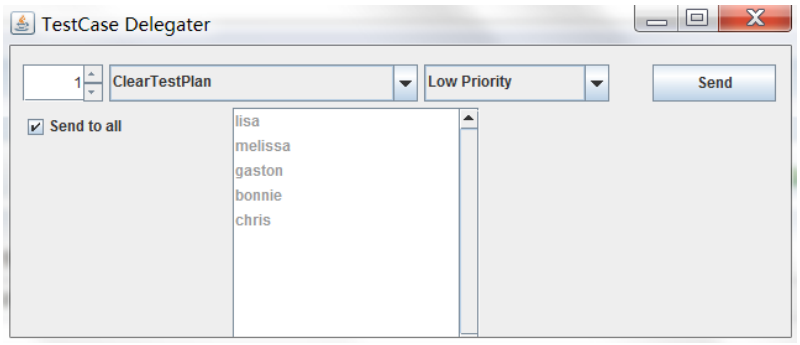


Figure 8.2.2: TestCase Delegater

In case that the GUI would not work in some unexpected situations, there is a second way for the master to send the test plan to slaves. Master can execute the command "executeTimedTestPlan" in console to dispatch the test plan to all slaves. However, for both methods, after all slaves are logged in, the master is required to execute the console command "sendFriendRequestInOrder" ahead of sending test plan since friendship is a precondition for most test cases.

The command "sendFriendRequestInOrder" is newly created in order to solve the potential problem caused by TestCaseSendFriendRequest. As described in [*Kangping's thesis*], during a large scale testing, the execution of TestCaseSendFriendRequest in test plan always leads to some slaves stuck. It is possibly because too many slaves ask the master for an online user name at the same time. Therefore, the new command demands the master to transmit TestCaseSendFriendRequest to each slave in sequence. After the command is executed and every slave has at least one friend, the test plan can be delivered.

In Test Plugin, a test plan named “TimedTestPlan” is used. The TimedTestPlan consists of 3 parameters, as shown in Figure 8.2.3, namely repetitions + test case + time bound. For instance, the first row of the test plan means that slaves need to execute TestCaseSendWallPost 5 times in 5 minutes. If slaves finish the required behaviors less than 5 minutes, they have to wait till the end of the 5 minutes; otherwise, slaves directly jump to next test case when the 5 minutes is up. The time bound ensures that the execution of test plan for every slave is synchronized, which makes it feasible to analyze the performance and stability of LifeSocial with Monitoring.

```
5x TestCaseSendWallPost 5m
1x Wait 3m
5x TestCaseCommentWallPost 5m
1x Wait 3m
5x TestCaseViewFriendsWall 5m
1x Wait 3m
5x TestCaseViewOwnWall 5m
1x Wait 5m
10x TestCaseForum 10m
1x Wait 3m
20x TestCaseCreatePhotoAlbum 10m
1x Wait 1m
50x TestCaseUploadPhoto 10m
1x Wait 1m
20x TestCaseViewFriendsPhoto 10m
1x Wait 1m
```

Figure 8.2.3: TimedTestPlan

Each test case defines a basic behavior of the system, such as uploading a photo or viewing own wall. And as mentioned before, in the past year, our PG changed the system a lot, both functionally and graphically, and we do not have enough time to modify existing test cases for the changed features and create new test cases for the new features. Therefore, it is necessary to verify if the old test cases work on the new LifeSocial.

For this purpose, we did a small scale testing (1 master + 5 slaves) with Test Plugin in the pool area on the second floor of Heinz-Nixdorf-Institute (HNI) at Paderborn University. The environment of the testing is as follow:

- Processor: Intel i7-2600
- RAM: 8 GB
- Operating System: openSUSE 12.3-x86_64
- Network Interface: Gigabit Ethernet
- Java VM: 1.7.0_51 build 24.45-b08

TestCases	TestingResult
TestCaseChangeProfileInformation	working
TestCaseChangeProfilePicture	working
TestCaseViewOwnProfile	working
TestCaseViewFriendsProfile	working
TestCaseRegister	working
TestCaseLogin	working
TestCaseLogout	working
TestCaseLogoutLogin	not working
TestCaseSendWallPost	not working
TestCaseCommentWallPost	not working
TestCaseViewOwnWall	not working
TestCaseViewFriendsWall	not working
TestCaseCreatePhotoAlbum	working
TestCaseLoadPhoto	working
TestCaseUploadPhoto	not working
TestCaseViewFriendsPhoto	not working
TestCaseSendFriendRequest	working
TestCaseSearchCity	working
TestCaseSearchCountry	working
TestCaseSearchUser	working
TestCaseSendChatMessage	working
TestCaseViewChatHistory	working
TestCaseSendMessage	working
TestCaseForum	working
TestCaseStoreFile	working
TestCaseViewFriendsGroups	working

Figure 8.2.4: Testing Result

Currently there are 26 test cases in Test Plugin for LifeSocial. We tested these test cases individually instead of arranging them all into a test plan, in case that invalid test cases cause failure of whole testing. As can be seen from Figure 8.2.4, most test cases are still available for the new system, and 7 of them become invalid for some reason. In the next step, these invalid test cases need to be modified according to the corresponding functionalities. Also, based on the new LifeSocial, some valid test cases' names need to be changed to more proper ones. Additionally, new test cases demand to be created for missing features that are not included in Figure 8.2.4, such as voting and notification.