

Project Group 2011

A Distributed Framework for Social Networks

Final Report

Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Supervisor: Dr.-Ing. Kalman Graffi
April 29, 2012

Contents

1 Introduction **1**

2 Goals **3**
2.1 Approach . . . 3
2.2 Non-Functional Requirements . . . 3

3 Fundamentals of P2P Networks **5**
3.1 FreePastry . . . 5
3.2 PAST . . . 7
3.3 Conclusion . . . 10

4 Concepts **11**
4.1 Communication Principles . . . 11
4.1.1 Communication Strategies . . . 11
4.1.2 Message Passing Strategies . . . 13
4.2 Publish/Subscribe . . . 15
4.2.1 Scribe Publish/Subscribe Solution . . . 15
4.2.2 Problems and Counter Measures . . . 16
4.2.3 Conclusion . . . 17
4.3 Streaming . . . 18
4.3.1 Splitstream Streaming Solution . . . 18
4.3.2 Problems and Counter Measures . . . 19
4.3.3 Example . . . 19
4.3.4 Conclusion . . . 20
4.4 Security . . . 21
4.4.1 Existing Security Concepts and Modifications . . . 21
4.4.2 Secure Messages . . . 22
4.4.3 Secure Storage . . . 23
4.4.4 Conclusion . . . 23
4.5 User Management . . . 24
4.5.1 Users . . . 24
4.5.2 Groups . . . 27
4.6 Distributed Data Structures . . . 28
4.6.1 Providing Linkage . . . 29
4.6.2 Remote Operations and the Backend . . . 30
4.6.3 Distributed Set and Linked List . . . 31
4.6.4 Prefix Hash Tree . . . 33

4.6.5 Conclusion . . . 36
4.7 Global Aggregation and Monitoring . . . 36
4.7.1 Node Internal Structure . . . 37
4.7.2 Aggregation Strategies . . . 38
4.7.3 Network Topology . . . 38
4.7.4 Conclusion . . . 41
4.8 Test Plugin . . . 42
4.8.1 Motivation . . . 42
4.8.2 Implementation . . . 42
4.8.3 Generation of Content . . . 43
4.8.4 Conclusion . . . 44

5 Framework Architecture **45**
5.1 Discussion . . . 45
5.2 Components . . . 46
5.3 Interfaces . . . 48
5.3.1 Framework . . . 48
5.3.2 Network . . . 49
5.3.3 Messaging . . . 50
5.3.4 Storage . . . 52
5.3.5 Security . . . 52
5.3.6 Streaming . . . 53
5.3.7 Users . . . 53
5.3.8 Distributed Data Structures . . . 54
5.4 Conclusion . . . 55

6 Integration **57**

7 Test Documentation **63**
7.1 Goal of Testing . . . 63
7.2 Tests . . . 63
7.3 Conclusion . . . 66

8 Organization **67**
8.1 Seminar . . . 67
8.2 Implementation . . . 69
8.3 Planning . . . 70

9 Conclusion **75**
9.1 Achievements . . . 75
9.2 Future Work . . . 76
9.2.1 Graphical User Interface . . . 76
9.2.2 LifeSocial . . . 76
9.2.3 P2P Framework . . . 76
9.3 Summary . . . 77

Appendices **79**

Appendix A How-to Manuals **81**

Appendix B Test Plugin **91**

Appendix C Test Plots **99**

Appendix D Remote Operation Protocols **101**

Appendix E Aggregation Strategies **113**

Bibliography **117**

List of Figures

3.1 Routing a message with key E46A1E from node 75A1FD in Pastry. 7
3.2 Inserting a data object with file-ID E46A1E in PAST. 9

4.1 Streaming to 7 nodes over 2 stripes 20
4.2 The structure of a secure storage item 24
4.3 FreePastry ring with user-ID and node-ID. 26
4.4 Positioning scheme of a Distributed Linked List 32
4.5 Prefix Hash Tree (Bucket size: $s = 5$) 35
4.6 Aggregation Tree [10] 40

5.1 Components of the framework 47
5.2 Content of package: `com.p2pframework` 48
5.3 Content of package: `com.p2pframework.network` 49
5.4 Content of package: `com.p2pframework.message` 50
5.5 Content of package: `com.p2pframework.message.aggregate` . . 51
5.6 Content of package: `com.p2pframework.storage` 52
5.7 Content of package: `com.p2pframework.security` 53
5.8 Content of package: `com.p2pframework.stream` 54
5.9 Content of package: `com.p2pframework.users` 54
5.10 Content of package: `com.p2pframework.dds` 55

6.1 Architecture of LifeSocial [11] 57
6.2 Components of the social network after the integration 59
6.3 Screenshot of LifeSocial 60

7.1 The pool area PCs used for remote testing 64
7.2 Network traffic during a successful test 66

8.1 All members and the supervisor of the PG 67
8.2 Members of the PG at CeBIT 2012 71

C.1 Total number of elements in stored data 99
C.2 Total size of stored data 99
C.3 Messages processed by the Message Dispatcher 100
C.4 Messages processed rate by the Message Dispatcher 100
C.5 Data transfer over Network 100
C.6 Data transfer rate over Network 100
C.7 Message Transfer rate over Network 100

C.8 Message received over Network 100

1 Introduction

Online social networks (OSN) account for nearly 10% of the time people spend on the Internet [3]. Their purpose is to allow users to publish personal information, link it with friends, share different kind of data and communicate with each other. These users can present themselves and provide insights into their friendships, interests and likings. In almost the same manner peer-to-peer (P2P) networks are getting more and more accepted. These networks provide a data sharing experience. However, the important factor is that they leverage the benefits of high collaborate bandwidth and large distributed storage.

So, at the moment two trends are becoming apparent: Firstly, online social networks are getting ubiquitous and enjoy a steady growing user base; secondly, distributed P2P systems are becoming widely accepted. Known social networks like Facebook or Google+ are built and realized with large server farms. This provides them with tight control, but carries the burden to finance this architecture. After all, a single company provided with this vast control and massive power over all the personal information is a big privacy issue. The company owns all the user data and makes money by selling it to other companies, e.g. adverisement companies. Lately more and more users become aware of this situation and many of them disagree with this practice.

Consequentially, some ideas arise to overcome this architecture. One of them is to use P2P technology to store the personal data of all users distributed in the network. To constrain access rights the data obviously has to be encrypted and a solid access control system has to be put into place. This results in a situation where only the user who owns some data and all users which were permitted to interact with the data are able to really interact with it. This report presents a P2P framework that allows to easily create P2P applications.

2 Goals

The project group (PG) created a framework for a distributed system based on a P2P network. The capabilities of the framework are demonstrated by a distributed online social network as an alternative approach to existing social networks.

2.1 Approach

In a P2P network all participating users act as a server and as a client at the same time. Due to utilizing the whole collective bandwidth and memory, these systems avoid the server as a bottleneck. They also generate fewer costs for the operating company by lowering the server requirements. Furthermore, using the P2P approach is a good alternative to client-server architectures because it enhances the privacy and the control over the provided user data. This is done by encrypting and distributing all data across all peers. Therefore all data is saved decentralized, so that nobody can misuse or sell any private data of the users.

Because of these promising features of distributed P2P systems, the PG was highly motivated to develop a framework on top of a P2P overlay. This framework should be usable for all kind of P2P application, such as chat applications and games. However, the main goal was to construct the basis for a distributed online social network with distributed control mechanisms and all common features known from existing centralized solutions.

Therefore, the PG combined the two trends mentioned in Section 1 and built a distributed social network based on our P2P framework.

2.2 Non-Functional Requirements

The approach of a distributed online social network based on a P2P network does not only have advantages. There are also some challenges the PG faced. Most of them are common to distributed applications.

Availability All data in a P2P network is stored at the participating peers. Because the online behavior of these peers is unpredictable, the availability of the data has to be ensured. Replication stores the same data on different peers to make it available, even if some of these peers go offline. The modification of a replicated file and consistency between all distributed replicas is not trivial.

Performance The additional routing layer that is introduced by the P2P network puts additional latency to every message sent through the network. This decreases the performance of the system built on top of the P2P framework. Especially the response time of interactions in the social network is suffering from this fact. In the same way the available bandwidth may be low and e.g. slow down a file transfer. A flexible caching mechanism could reduce the effect and increase the performance.

Heterogeneity Another problem in P2P networks are heterogeneous peers. The framework has to ensure that devices with different capabilities can be integrated in the same system. It cannot assume the availability of a certain amount of storage, processing power, or bandwidth, but has to adapt its need to the actual resources. The problem is to make the distributed social network work on all of these devices in the same manner, without disturbing the user experience.

Security Security is the last big challenge to face. In comparison to a centralized system, a distributed system does not have a central authority that can be trusted, hence all nodes in the network have the same rights and access to all data. Therefore, the use of verification and encryption techniques is necessary to achieve confidentiality, integrity, authenticity, authorization and non-repudiation. Furthermore, the P2P system should still be functioning in presence of adversaries and especially it should be robust against routing and storage attacks.

For some of these challenges, there were already existing solutions; the PG adapted and implemented these solutions. For other problems without an existing solution, the PG had to develop one from scratch.

3 Fundamentals of P2P Networks

This chapter elaborates on the fundamentals of P2P overlays. Here the groundwork is shown in order to discuss solutions in Chapter 4 that solve the identified challenges of Chapter 2.

Section 3.1 discusses the P2P overlay FreePastry including routing mechanisms and the provided distributed hash table (DHT). Following, Section 3.2 goes into detail about solutions how to keep data stored in the DHT persistent and resilient to node failures.

3.1 FreePastry

This section will give an introduction to Pastry, a network overlay that offers a routing mechanism for sending messages in a P2P network. Such a mechanism is needed, because in a large P2P network it is not feasible, that each peer knows how to address all the other participating peers of the network directly. A routing mechanism forwards a message step by step further to the destination, by using other peers as intermediate goals. With this approach, it suffices that each participating peer knows only a small but well defined subset of the peers in the whole network. The PG decided for Pastry, because it is a mature and scientific investigated routing overlay. In addition, the initial plan of the PG was to extend an existing prototype for a P2P-based, distributed social network, which already used FreePastry, a free implementation of Pastry. The description of FreePastry is based on the publications of Druschel and Rowstron [8, 13, 14].

Pastry is able to route messages to peers within the network by only needing $\lceil log_{2^b} N \rceil$ hops, where N is the number of peers in the network and b is a configuration parameter[1]. The core idea of Pastry is, that every peer — in the following called node — stores shortcuts to other nodes in the network. Every node can be addressed by its so called node-ID, which is a unique identifier. The node-ID is a value between 0 and $2^{128} - 1$ and is assigned randomly when the node enters the network. Since the nodes with node-ID 0 and $2^{128} - 1$ are defined to be adjacent to each other, the node-ID space can be seen as a closed circuit. The nodes are only adjacent within the Pastry overlay, so they may have a different location, since the node-IDs are assigned randomly. This has some benefits, which will be described later on in Section 3.2. Below is a description of Pastry's routing algorithm and the used data structures.

Assuming a message m has a numerical key k, then m is routed towards the

[1] b depends on the used notion, i.e. $b = 4$ in a hexadecimal system

node t with the node-ID that has the closest matching prefix of k. It is not feasible that every node stores routing information about all the other nodes in the P2P network. It is also not efficient to delegate the message step by step through adjacent nodes until t is reached. Therefore, Pastry's routing algorithm takes advantage of two data structures, namely the *leaf set* and a *routing table* with shortcuts. The shortcuts are different for every node.

In the leaf set, each node stores the node-IDs of the $\frac{|L|}{2}$ adjacent nodes with a larger node-ID than its own node-ID and the $\frac{|L|}{2}$ adjacent nodes with a smaller node-ID[2], with L as the size of the leaf set[3].

In the routing table, each node stores shortcuts to distant nodes in the node-ID space. Each routing table has $\lceil log_{2^b} N \rceil$ rows. Each row can be seen as a *level* that stores routing information to other nodes. The higher the level l is, the more the node-IDs of the nodes in this level match to the node-ID of the respective node, namely the l most significant digits of the node-ID are equal[4].

With these data structures, an efficient routing algorithm is possible. If the message m can directly be routed from node s to the target node t, k must be within the range of the node-IDs in the leaf set of s. Otherwise, m is routed towards a node within the routing table of s, which most closely matches with k and the operation is repeated recursively until m is delivered to t.

Figure 3.1 illustrates the routing algorithm of Pastry. In this example, the value of the configuration parameter b is 4. This figure represents only the boundary nodes 0 and $FFFFF...F$ of the node-ID space and the nodes involved by the routing algorithm, so imagine there are several more nodes between the represented nodes. The node s_1 with node-ID $75A1FD$ wants to route the message with key $E46A1E$, but since it is not in range of its leave set, it is routed to the node s_2 with node-ID $E13DA4$, because this is the node in the routing table of s_1, which most closely matches the key. Assuming a high populated network and randomly generated node-IDs, the amount of nodes between s_1 and the target node t could nearly be halved with the help of this shortcut. This procedure repeats, until the message is delivered.

To signal that a node is still alive, it sends repeatedly messages to the other nodes in its leaf set. Thus, when a node notices that a neighboring node has not sent a live signal for a certain time, it is assumed to be dead. In this case, the affected nodes update their leaf sets. For detailed information of Pastry, it is recommended to take a closer look at the work of Druschel and Rowstron [13].

Distributed Hash Table

The concept that Pastry uses for routing data in the network is called distributed hash table (DHT).

In general, a DHT uses a predefined ID space for all nodes, typical sizes for this

[2] imagine the node-ID space as a closed circuit
[3] L is a parameter that is defined a priori
[4] assuming $l = 0$ in the first row

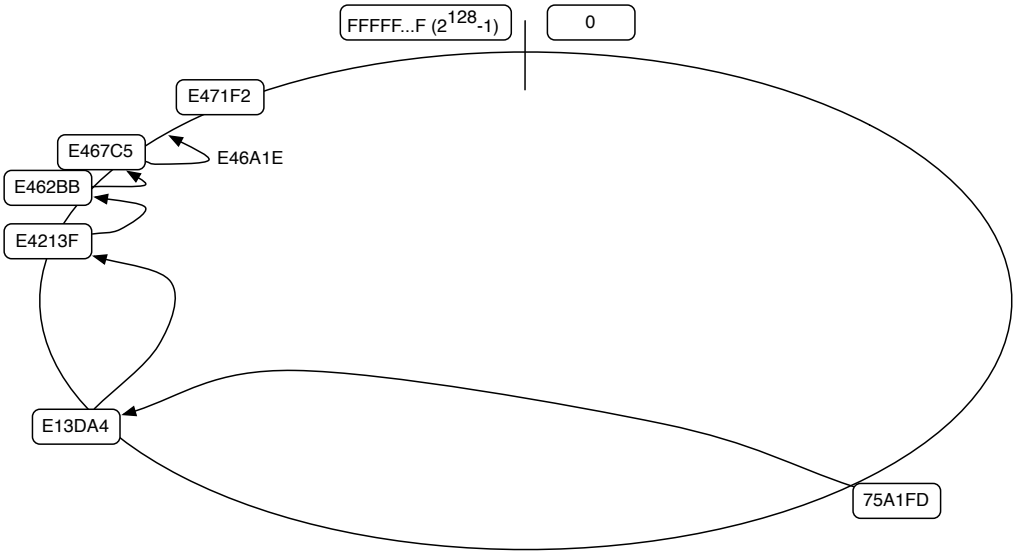


Figure 3.1: Routing a message with key E46A1E from node 75A1FD in Pastry.

ID space are 2^{128} or 2^{160}. All data items have an assigned node-ID from the ID space at which they are stored. The mapping from data to an ID is done using a hash function. Participating nodes also have an assigned ID from the same ID space. The ID space can be seen as a circular structure in which the successor of the highest ID is the lowest ID, i.e. 0. This results in a ring with active and passive nodes, where the active nodes are participating peers and the passive nodes represent data items. An active node is responsible for a passive node if it is the *closest* active node to this passive node. Depending on the used DHT implementation the *closest* node can be defined differently. Pastry, for example, defines the closest node as the active node with the node-ID that has the longest matching prefix with the node-ID of the passive node.

Every active node has a routing table with routing entries to other active nodes. The routing table can be constructed such that it is always possible to find a node which is closer to another passive node. Eventually, this routing process will approach a responsible node for the requested passive node. This allows the DHT routing protocol to find a route to any node-ID, independent of the node-ID of the requesting node. Typically, DHT routing protocols have an expected routing time of $O(\log N)$, where N is the number of nodes in the network.

3.2 PAST

This section describes PAST, a data replication layer based on Pastry. Data replication is needed, since the data in a peer-to-peer network is stored on different peers. Without data replication, each peer that stores data has to be online to

ensure access to every data object[5]. As this condition is not wanted and possibly not feasible, data replication is used to avoid it. Therefore, the original object is replicated on several other peers. There are four basic operations within data replication, namely *read, write, update* and *delete*, which all take as input the identifier of the relevant data object. The following presentation of PAST is also based on the work of Druschel and Rowstron [8, 14, 13].

Since PAST is a replication layer based on top of Pastry, each node in PAST is also identified by a 128 bit node-ID. In addition to this, each file inserted to the network can be addressed by a 160 bit identifier, called file-ID. This identifier can be calculated by hashing the "file's name, the owner's public key, and a randomly chosen salt" [14].

An inserted file is stored to that node, whose node-ID most closely matches to the prefix of the inserted file's file-ID. This node r can be found via Pastry's routing algorithm, with the file-ID as the key of the routed message. Because PAST is a replication layer, this file is not only stored once at node r, but is replicated to $k-1$ additional nodes. The remaining nodes are the $k-1$ other closest nodes regarding to the file-ID and can possibly be found in r's leaf set. This is where the in Section 3.1 mentioned benefit of Pastry's random node-ID generation asserts. Due to this, the replication nodes for the same file may be distributed to different locations, so the access latency can be decreased. Additionally these nodes are not correlated to local disasters, like earthquakes. During this insert operation, it is ensured that the file has not been corrupted or manipulated during transfer. This is achieved by creating an additional *file certificate*, which includes the "file-ID, a SHA-1 of the file's content, the replication factor k, the used salt" [14] and a timestamp, and is sent along with the file. With the hash of the file's content, the receiving node r can calculate the hash of the received file itself and check if it is identically to the hash in the file certificate, to ensure the data integrity. To even be sure that actually k replicas were created, each successful replicating node sends an acknowledgement message—called *store receipt*—back to the origin file insertion node. In the case of any errors, the client is notified.

Figure 3.2 illustrates replicating in PAST, with replication amount parameter $k=5$. Like in Figure 3.1, this figure also represents only the boundary nodes 0 and $FFFFF...F$ of the node-ID space and the nodes involved by PAST's replication, so imagine there are several more nodes between the represented nodes. The node i with node-ID $75A1FD$ wants to insert a file with file-ID $E46A1E$. The responsible node r with node-ID $E467C5$ is found via Pastry's routing algorithm. r checks the integrity of the file and after successful validation, it distributes the file to the four remaining *closest* nodes.

Another operation in PAST is the lookup operation. The purpose of this operation is to access the stored file. Therefore, it takes the file-ID of the respective file as input parameter. With this parameter, the responsible node r can easily be found via Pastry's routing algorithm. Additionally to the file, r also sends the file certificate back to the origin lookup initiation client, so that it can

[5]depending on the granularity, this could be e.g. a file or a data block

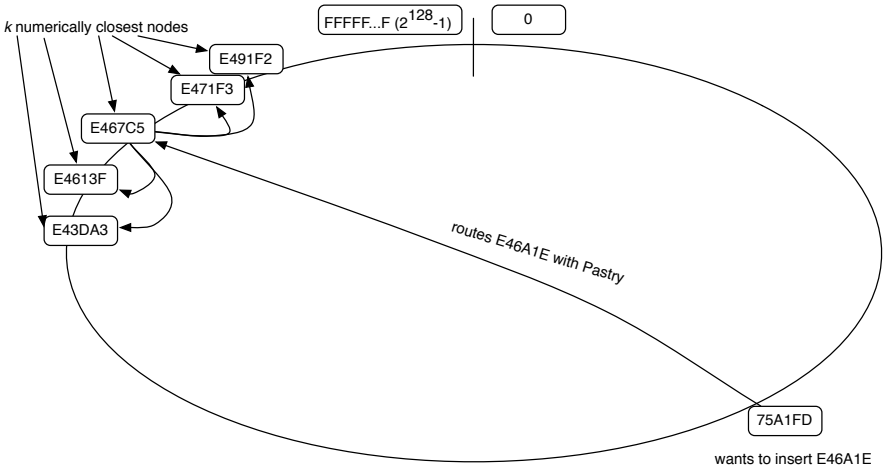


Figure 3.2: Inserting a data object with file-ID E46A1E in PAST.

validate the data integrity.

A further operation in PAST is the reclaim operation. It takes as input parameter the file-ID and the file's owner-credentials. As a result of this operation, the availability of the respective file is no longer ensured by PAST, but the file itself is not actually deleted. Before this operation is executed, it is ensured that the reclaim initiator is the file owner.

As empirical studies show, these operations itself are not suitable to ensure successful file insertion [14]. Therefore, PAST has two mechanisms to handle this, namely *replica diversion* and *file diversion*.

The first mechanism—*replica diversion*—aims to balance the load within a leaf set L. When a node r runs low on storage, it may delegate an incoming replicate request to another node t within its leaf set that has more storage available and was not previously selected for replicating the respective file. If t accepts the request, it stores the file and r stores a pointer to t. To ensure that the copy in t is still accessible, even if r fails, there is additionally added a pointer to t in the $k+1$-th closest node u. If r fails, u becomes responsible for the replica. Though it already contains a pointer to t and so the invariant of k replicated files is still fulfilled.

The second mechanism—*file diversion*—is initiated when a node's complete leaf set L is reaching its storage capacity. In this case, the file is distributed to another part of the nodeID space in PAST. This is achieved by choosing a different salt value, which generates a different file-ID.

To increase the effect of these two mechanisms, Druschel and Rowstron present three policies, which define,

1. when a node accepts an incoming replication request

2. which node is selected for replica diversion

3. when to initiate file diversion.

A node r accepts an incoming replication request of the file f, if the size of f divided by the remaining storage space of r is smaller than a threshold $tres$. The threshold $tres$ is a parameter that is defined separately for the primary replica nodes ($tres_{pri}$) and the diverted replica nodes ($tres_{div}$), with $tres_{pri} > t_{div}$.

The selected node t for a diverted node fulfills the following condition. As previously mentioned, t must be within the leaf set of the initiating node r. From this nodes, t is the node with the most remaining free storage space and must not have been selected previously for replication of the respective file.

The policy for initiating file diversion implies that file diversion is initiated after a node failed two times when trying to divert the replica to another node within its leaf set.

3.3 Conclusion

The services provided by Pastry and PAST suffice as a basis for routing and data replication in our distributed online social network. The presented results in a simulation of PAST show, that a reliable data availability can be provided [14]. Additional features like mutability and secure deletion of files could be achieved by modifying some of the PAST operations. Since FreePastry is an open-source implementation this should be possible. The support for a heterogeneous network, still remains a task for future work.

4 Concepts

This chapter elaborates how to solve the identified challenges of Chapter 2. In the following sections, a variety of problems in P2P networks are discussed and solutions introduced. These problems are split according to their area of application. Therefore, each section is mostly independent and comprises a problem description, solution or adaptation to the P2P Framework, some examples how to use the provided solution and a conclusion.

4.1 Communication Principles

The presumably most important problem to tackle in P2P systems is communication. It is the most basic operation that a network node must support. The PG decided to build a P2P framework that hides most of the reoccurring low level communication code and instead offers a high level interface. Therefore the problem to reach these goals consisted in creating a powerful and fast communication platform and designing an interface that supports easy message passing. The PG identified very different use cases for communication which could not be handled through one communication method only. That is because every communication method has its niche where it is very efficient and better suited than others. Therefore the diversity of communication modes was additionally considered as a goal.

The following Section 4.1.1 shows the overall communication possibilities of the P2P framework. Some examples are presented and their application discussed. Afterwards Section 4.1.2 elaborates on options how to pass arriving messages internally from layer to layer and what consequences rise from that alternatives. In the end every section concludes separately what was done by the PG.

4.1.1 Communication Strategies

The developed P2P framework supports many different communication modes. These modes can be classified by the criteria: waiting for response, kind of transmission and number of participating nodes. Every one of the following subsections explains one of these different communication manners.

Synchronous vs Asynchronous

If the framework user wants to wait with the execution of a follow up operation until the sent message is received and acknowledged this would be synchronous

messaging. Synchronous operation mode is equally supported as asynchronous ones, where the sending node continues executing while the transmission is in progress.

Overlay Hop-Based vs IP Based

Normally, communication is done via hops through the overlay P2P network. Sometimes it is important to communicate directly with the other node, considering low latency or high throughput. For the later kind it is possible to retrieve the IP information of a node once and using this to communicate directly without additional hops to other nodes in the P2P overlay. This is indeed used for streaming and file transfers.

Participating Nodes

Here the most differentiation of communication modes happens. There are very different modes to choose depending on the number of participating nodes. The list below shows all the supported configurations.

- **Unicast (1-to-1)** is the most basic kind of communication. A unicast communication has only two participants, a sender and receiver. The sender has to know the address of the receiver. It is also possible to chose from the above mentioned communication types. Besides messaging, another kind of 1-to-1 communication rises through file transfers.

- **Multicast (1-to-N)** builds on top of unicast. A multicast consists of N unicasts. Therefore it needs a list of receivers. 1-to-N communication is also used with streaming where one node distributes content live to some receivers. Streaming is technically based on publish/subscribe of topics and implemented very efficient through distribution trees (Section 4.3).

- **Publish/Subscribe (N-to-M)** removes the need to know the receiver addresses. Receivers register themselves as subscribers which get new messages concerning the registered topic. Senders can post new messages associated with a topic not knowing any receiver address. The topic acts as a kind of category for the messages. Thus subscribers get only messages published by other nodes on the same topic. Details can be found in Section 4.2. As mentioned earlier, streaming is also a kind of publish/subscribe mechanism, because it operates also on specific topics.

- **Aggregation (N-to-All)** is accessible to everyone in the network. This mechanism permits to distribute and aggregate information in the whole network. It can be used to monitor and aggregate performance measures or other small content. On the one hand one can provide local content through registering a sensor that collects the content. Since the content gets aggregated the sensor also has to provide an aggregation function that can

combine this kind of content. On the other hand it is possible to collect the aggregated global content anywhere in the network through registration of a callback. For more information on this, see Section 4.7.

Conclusion

Starting from only one possible communication mode in the previous system the PG widened the options for future framework users considerably. Publish/Subscribe and Streaming are two examples for very advanced communication modes. And now there are not only more participating nodes possible but also modes like IP based routing with IP information retrieval. But also some work is left. With some additional effort multicast can be implemented with distribution trees just like Publish/Subscribe. This was not done because this PG concentrated on giving the framework a wide scope of supported use cases. All in all the supported diversity of communication modes is more important than the optimization of a few of them.

4.1.2 Message Passing Strategies

Here it is described how information is passed through application layers of one node. The general problem is how to decide what to do with incoming data. So the overall use case would be that a framework user issues an operation which sends some data to another node. After some time the other node responds with a result. Now the framework user needs the result of the operation in a follow up operation. Two basic approaches exist to handle the message passing to higher layers.

Poll Based The framework layer must support to be asked wether the message is already there. To stay in the scenario above, the framework user would send a message over the network and then ask repeatedly the framework until an answer has arrived. Then he continues with the next operation that needed the answer from the remote node.

Push Based The framework layer must support registration of callbacks. These callbacks get executed from within the framework if an answer got received or error encountered. For the above scenario, this means the framework user sends a message and also registers a callback with the follow up operations handled there. Thus the sending thread can continue to work on other stuff and does not block. If the answer gets received the framework automatically calls the provided callbacks and the follow up operation gets executed without a waiting thread. This approach removes much of the latency imposed through the polling interval while also removing resource requirements due to extra threads.

The designed P2P framework at hand uses the push approach through out the framework. To clarify on the use of different names for the callback function/object,

there follows a list that explains them. Callbacks can be semantically divided into the following categories.

- **Callback** is the general term for a function given as an argument for later calling.
- **Continuation** is used for asynchronous operations where the callback gets called to continue some work after completion. For example a storing operation where you register a continuation with code that gets continued after the store operation.
- **Receiver** gets used as a callback that receives content. For example a message receiving function which gets called every time a message has been received.
- **Listener** is a callback registered to a process in progress. The listener gets informed over the status of the process. An example would be a running file transfer where a listener displays the progress on GUI level.

Communication Channels

The developed P2P framework supports different communication channels. Such a channel represents the point where messages can be sent and received. Therefore a channel pushes incoming messages to its receiver callbacks and provides functions to send over this channel. They can be created in relatively large numbers and function like categories. Messages sent with a certain channel can only be received at nodes that also have this channel. This simplifies communication for different framework users who only want to get certain messages. One immediate use case is a plugin architecture on top of the framework. Here every plugin simply creates the channels it wants to communicate over. Then the incoming message is automatically received at the right plugins. It is not needed to sort through incoming messages and then redirect them. They can simply use different channels.

Legacy Adapters

An adapter can be written if some system on top needs to poll the framework for incoming new messages because itself uses polling. Such a legacy adapter registers a callback on a channel and provides a function for polling by the legacy system. If new messages arrive the adapter gets them over the callback and saves them. The polling function returns newly saved messages when asked. Thus the system can poll the adapter which saved the messages for it.

Conclusion

Many of the aforementioned communication strategies result in different communication channels. There are message channels for unicast and multicast, topic

channels for publish/subscribe and streaming channels. They all act as communication end points and incorporate the push approach for message passing to framework users. These new communication channels paired with a faster internal message passing results in a versatile and fast communication framework. It also hides many low level tasks like message sorting and delivery from the user. Hence we got a very powerful communication system to build on top.

4.2 Publish/Subscribe

This section is about the publish/subscribe communication method and its realization. Publish/subscribe allows users to communicate without knowing their addresses. Messages are published to a given topic channel (cf. Section 4.1.1) and all users that have subscribed to this topic get the message. One additional use case to support is the anycast. An anycast will be sent to any registered subscriber and if it does not accepts the message it will be sent to another subscriber until anyone accepts. These topic based communication methods should be efficient to be used in news and notification systems where messages are delivered to a large subscriber base. Therefore an implementation of it must support large user bases while also scaling well with the number of topics used through out the system. Considering the P2P nature of the system, a general requirement is to balance the work load and traffic over the whole network.

The lower level techniques are implemented with Scribe [6] while on top of that the PG organizes caching of created topics and a push based interface. The next section goes into detail how Scribe realizes the publish/subscribe mechanisms. Then the section after that discusses failure recovery techniques in place and some possibilities to enhance them. At the end a conclusion is drawn.

4.2.1 Scribe Publish/Subscribe Solution

Scribe builds multicast distribution trees. These trees consist of real nodes and not just hash addresses. So they have to manage node failures, joins and leaves. In the following, a description of the Scribe protocol is presented. The description is split into sections that handle one protocol message respectively.

Creation Message If a topic channel gets created for the first time a *create* message gets sent to the node responsible for the hash of the topic name. This node is the only one that is defined through hashing and responsibility ranges. All other nodes are not known beforehand and have to be found through connections of real nodes. Thus the node receiving the *create* message forms a rendezvous point for all operations concerning this topic. It is also the root node of the distribution tree.

Join Message The join operation of subscribers is described here. Publisher do not have to be registered besides for the rights management done while

multicasting, described below. Any node that wants to join or subscribe sends a *join* message to the rendezvous point or root node mentioned above. Due to Pastry routing, routes with the same destination converge early. This means that join messages sent to the distribution tree root node take the same route through the p2p network at an early point of their travel. Also, a node can be member of the tree without been subscriber. This would be a forwarder node which only distributes the messages. Now if a *join* message gets routed along a node not in the distribution tree this node saves the requester as its child and sends an own *join* message itself. Therefore becoming a forwarder. Important though, loops can not be formed because Pastry routes only to nearer nodes. On the other side, if a *join* message comes along any node that is already a forwarder for the topic in question, this node will also add the requester as child. But no own *join* message will be sent. The root node does only get some join requests in small trees. Afterwards its children will catch the requests and since these message routes converge early it is probable that they will come along some forwarders of the distribution tree. This message snooping saves the root node from message overload while all requesting nodes have an address to send to.

Leave Message Leaving the distribution tree is easy. *Leave* messages are sent directly to its parent therefore snooping is not needed. The parent removes the sender from its children collection of the given topic and if this collection gets empty the node leaves also the tree. Thus forwarders do not remain in the network if the topic loses many subscribers. A node should only send a *leave* message if it is a leaf in the distribution tree.

Multicast A multicast is done through a *multicast* message sent to the root node. Remember the address of this node has not to be known since the root node is identified as responsible for the hash of the topic. The root node can also implement some rights management where it stores all nodeIDs of allowed publishers. Now the root node simply forwards the message to all children of the specified topic. The children can be forwarders which only relay or subscribers that also want the content for themselves.

4.2.2 Problems and Counter Measures

The most important property of Scribe is its resilience to node failures. In order to achieve this the following techniques are employed. This list describes problems and counter measures of distribution trees.

Root Node Failure The root node is the single point where all messages are sent to. Luckily it is not represented through one real network node but a hash. If the node responsible for this hash leaves or has a failure Pastry assigns another node that gets responsible for the hash. Additional use of PAST ensures that the new responsible node has also the same data. Thus root

node failures can be tolerated as long as the data gets replicated with PAST. Here it is also beneficial that children do not need to know their parents.

Parents Lost Children get multicast messages through their parents. But they do not know them. Hence the problem, how to detect parents that have left the network. To solve this, forwarders send heartbeats to their children. If the forwarder has some of its children in its Pastry leaf set or routing table which is very probably, it does not have to send heartbeats since Pastry does this already in order to maintain its routing and leaf sets. So children can suspects lost parents when no heartbeat was received in a while. Then the tree has to be repaired. This is done by sending a new *join* message again. In that case Pastry sends this message to a new existing parent, thus repairing the tree.

Capacity Bottleneck Here the problem is that the distribution trees described above assume equal capacity of all nodes. Thus low capacity nodes form bottlenecks in real worlds scenarios if they have to act as a forwarder. Hence Scribe bounds the amount of multicast forwarding and offloads children to other nodes if necessary. This can be done without signaling the child because the child only reacts if no heartbeat arrives.

Supporting Additional Guaranties All the supported communication modes of scribe are best effort practices. Since Scribe only deals with node failures and capacity bottlenecks it is sometimes important to address additional requirements. Scribe is very robust but if a guarantied delivery is needed more failure recovery measures need to be in place. At the moment it does not guaranty the order of messages been preserved and if TCP connections fail messages could be lost. Luckily Scribe supports callbacks that can be used to get informed about joins, leaves, forwards and faults. These callbacks are enough in order to implement resending of lost messages. Also important is the preserving of order of messages in some use cases. For that, it is advised to use sequence numbers in those messages.

4.2.3 Conclusion

This approach scales very well with the number of subscribers. In fact it works more efficient with a high number of subscribers. Also the number of created topics scales well. Due to having many forwarders in the P2P network an additional topic just means few additional child collections. Forwarder and children are not much separated thus P2P hops are minimized. The root node, despite being rendezvous point for all messages, gets released of much message handling through message snooping of forwarders. Thus there is also no single point of failure. The only limitation of this architecture [6] are many very small groups of subscribers. This rises from the fact that in these situations the distribution trees only consist of long non branching forwarders. Therefore in situations with many small subscriber groups more efficient solutions with better branching can be applied.

The shown counter measures for node failures improve the network stability considerably. All messages involved in these repair strategies are local to the failing node and only a small number of nodes is involved. Thus providing good failure resistance. Additionally implementing resending of lost messages is easy to build on top. And sequence numbering, although on a higher level, is implemented for our streaming capabilities that build on top of Scribe.

4.3 Streaming

Streaming in this framework is thought as an high bandwidth 1-to-N communication. Thus it is indeed very different compared to simple file transfers. To achieve high bandwidth streaming it has to deal with low bandwidth nodes and since all this is happening in a P2P manner there are hard upload constrains for many endusers. These endusers can not contribute many upload bandwidth due to their asymmetric internet connection. Therefore simply using distribution trees which have a multiple upload than download bandwidth requirement are not feasible. A solution needs to distribute the content while also imposing only low upload bandwidth needs on all user nodes. This equals the general P2P problem to distribute resource requirements evenly over the participating nodes.

One solution gets described here in Section 4.3.1. Afterwards some features to recover from failures are presented and in Section 4.3.3 an example is given. In the end Section 4.3.4 draws a conclusion.

4.3.1 Splitstream Streaming Solution

The implementation used here utilizes Splitstream [5] which is based on Scribe distribution trees described above in Section 4.2. But Splitstream uses these trees in a special manner that circumvents the higher upload than download bandwidth requirement. This means that every nodes outgoing bandwidth is never higher that the incoming bandwidth requirement although the incoming data gets duplicated and sent to children. This is done as follows.

These distribution trees are named stripes. Because the content gets divided into stripes and delivered through different trees. But these trees have to span over the same nodes due to all nodes have to receive all stripes. Considering that in one distribution tree only inner nodes have to contribute more outgoing than incoming bandwidth and leaf nodes do not have to contribute any outgoing bandwidth the inequality gets obvious. To overcome this inequality one node gets different roles in different distribution trees. Most important now is the property that every node is only an inner node in one tree and a leaf node in all others. Then every node has never more outgoing edges than incoming ones. This is always possible since the number of leafs in a balanced tree is greater or equal than the number of inner nodes. Suppose the tree has depth d and arity a, Equation 4.1 compares the number of internal nodes to leafs. This means that

the fraction of leaf nodes increases with higher arity. Hence every node only has to be an inner node once.

$$|\text{inner nodes}| = \frac{a^d - 1}{a - 1} \leq a^d = |\text{leaf nodes}| \tag{4.1}$$

4.3.2 Problems and Counter Measures

Sadly Splitstream delivers also just best effort routing. But since Splitstream is built on top of Scribe it is also resilient to node failures. These node failures can cause lost messages if the timing is bad. Also the ordering of the messages can be disturbed. So, at first the PG implemented an order preserving mechanism through sequence numbering. Then the focus went to implementing methods to recover from lost messages. The first example was the possibility of a retransmit request from the receiver. But this method has its drawbacks. If many packages get lost the sender gets many requests. A better way to recover from lost messages is to send replicas which is also possible at the moment. These redundant messages get sent through different stripes such that the lose of one does not effect the other. But here the bandwidth overhead is relatively large. Jet another way is to use forward error correction (FEC). FEC is very powerful. If the average message lost rate is known the FEC can be adjusted to just get messages through without more overhead. At the moment the PG only uses checksums to detect errors. Another approach is to use content that can be split into many stripes where the lose of one stripe is not dramatic for short time. This independent but combinable content encoding can be found in movie encodings where every stripe just improves the quality of the movie. Java does not support FEC easily so the second approach moves more into focus.

4.3.3 Example

In Figure 4.1 it is shown how such a forest of distribution trees looks if seven nodes receive a stream over two stripes. There it is also apparent that every node has no more outgoing edges than incoming ones. Thus having lower upload bandwidth consumption.

In the top of Figure 4.1 is the source of the stream shown. It is the node that provides the content sliced into multiple stripes. Since it only provides the stripes to one node respectively it just has to upload the whole content once. Therefore the upload bandwidth is already lower than sending it to two children. These children (*1* and *7*) start distributing their stripe which is only half the size of the overall content. In this example they distribute through a binary tree. Every node in Figure 4.1 has two incoming and two outgoing edges only the node *6* is speared of outgoing edges because there are only seven participating nodes. Thus every node gets the two stripes and can reconstruct the whole content. Despite the need to duplicate the content, every node only needs to send two copies of one stripe which corresponds just to the incoming whole content.

Now lets apply Equation 4.1 to the first stripe of the example. Here we have four leaf nodes (*4*, *5*, *6*, *7*) and three internal nodes (*1*, *2*, *3*). Therefore the fraction of leaf nodes is $\frac{4}{7} \approx 57\%$. For an arity of two this fraction is always higher than 50% and with higher arity the fraction of leaf nodes just rises. Thus more than 90% of a 16-ari tree are leaf nodes.

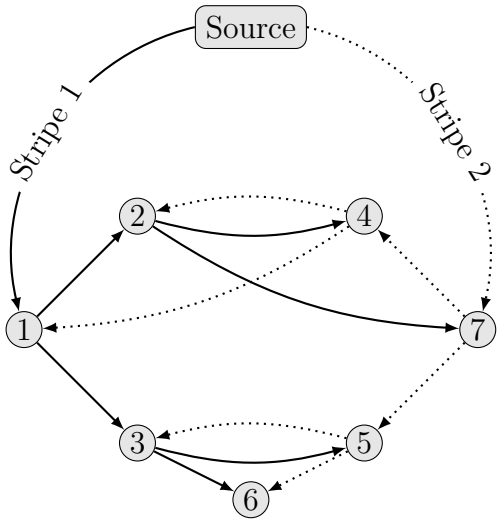


Figure 4.1: Streaming to 7 nodes over 2 stripes

4.3.4 Conclusion

Splitstream provides an elegant solution to minimize the upload bandwidth requirement through equally distributing the workload. There are no more pure leaf nodes which consume without participation in routing. Thus fulfilling most of the posed requirements.

The support for order preserving was integrated and can be enabled and disabled due to some applications just not needing these guaranties and therefore avoiding the overhead. Thus the P2P framework supports low overhead streaming for data that is already formatted appropriately. Additionally there are also options to resend lost packets and use checksums to verify received data. So the framework user can decide what is most appropriate for the application on top. At the moment the P2P framework also supports adapting the bandwidth utilization and registration of listeners to get status updates for bandwidth changes and progress. These use cases and some code examples get listed in the how-to section in Appendix A.3.2.

4.4 Security

In a P2P network all stored data is generally accessible by everyone. In the same way, many messages are handed through other nodes of the network until they reach their destination and can thus be easily intercepted. This makes the application of appropriate security mechanisms both challenging and very important. The goal of the applied security mechanisms is that all data and messages are only readable by authorized persons.

This section introduces the concepts to realize secure messaging and secure storing of data. Because of the structure of P2P systems, this is only possible by using cryptography. Therefore, at first some general security concepts are introduced in Section 4.4.1. After that, Section 4.4.2 introduces the used concepts for secure messages and Section 4.4.3 explains how the secure storing of data can be achieved.

4.4.1 Existing Security Concepts and Modifications

This section introduces some general security concepts that are used to achieve a security-aware P2P framework. These are symmetric and asymmetric crytopgraphy and the concept of access control lists. Some of the used security concepts required some minor modifications to be usable for our scenario. Therefore, this section furthermore explains the required modifications to the introduced concepts.

Symmetric Cryptography

The framework requires symmetric encryption at several places. In symmetric cryptography the same key is used for encryption and decryption. The used symmetric key algorithm is AES with a key size of 128-bit. This algorithm was chosen because it is an open, standardized and fast cipher. The official recommendation of the National Institute of Standards and Technology is that the use of the 128-bit version of AES is sufficient for at least another 20 years [2]. However, increasing the key size to 192-bit or even 256-bit is easily doable.

Asymmetric Cryptography

Asymmetric cryptography is used for the encryption and decryption of data as well as for the signing and verification of data. The PG decided to use the elliptic curve cryptography (ECC) approach because it works with very small key length. Thus, instead of using a 1024-bit RSA key only a 160-bit ECC key is needed to achieve similar security. Using a 160-bit ECC key is considered to be secure for at least another 10 years [4].

With a small asymmetric key length it is possible to use the public key as a node ID which leads to fundamental security benefits. The node ID length of the used overlay—FreePastry—is 160-bit, therefore, using a 160-bit ECC key results

in a one-to-one mapping between the public key part of the asymmetric key and the node ID.

Eventually the key size will have to be increased to maintain efficient security in the future. There are two basic ways to achieve this:

1. In addition to the key size also increase the node ID length of the used overlay. At least in FreePastry this is a non trivial task.

2. Split the used public key into a 160-bit part and a part containing the remaining bits. Then use the 160-bit part as the node ID and attach the remaining bits in an additional field to it. This way the full length public key can be retrieved from these two parts. As a result the important attribute that the public key belongs to a certain node ID persists.

The PG had to apply a slight modification to the used FlexiProvider ECC algorithm. The problem is that the actual public key of the FlexiProvider ECC algorithm uses 161-bit. To ensure a 160-bit key multiple hashing of the key seed is used until the first bit of the key always has the same predefined value and thus representing a 160-bit key.

Access Control List

An access control list (ACL) is used to define user permissions for a given object. This can be a simple list that maps user-IDs to the access permission (e.g. read permisson). Using this list, the owner of the object can verify if a requester has the necessary access permission. The ACL is stored together with the object.

This technique can also be used if an object is accessible by everyone. In this case the object obviously has to be encrypted to hinder non-authorized persons from reading the content. The required information to decrypt the object is then put in an ACL. However, as the ACL is also readable by everyone, this step is not yet sufficient. To solve this, the required information to decrypt the storage item—typically a key—is encrypted with the public key of each authorized user. This guarantees that only the authorized users can decrypt the key required for the decryption of the storage item because the decryption involves the private key of an authorized user.

4.4.2 Secure Messages

The first goal of having secure messages is to transfer messages through message channels that are only readable by other authorized users. The authorized users can be either only a single user or a group of users. Furthermore, the second goal of secure messages is to prove the authenticity of the origin of a message.

To achieve these goals, a secure message represents all kind of security enhanced messages. For this it wraps an existing message and applies the requested security mechanism. The two basic security mechanisms that are supported are:

Encryption This mechanism is used to achieve the first goal. The message can be encrypted with either a symmetric key or with the public part of an asymmetric key. In case of symmetric encryption the symmetric key has to be distributed to all authorized receivers, so that they can properly decrypt the message. If the message is encrypted with the public key of the receiver only the receiver can decrypt the message because no one else knows the private key of the receiver.

Signature A signature is used to prove the authenticity of the origin of a message. The message has to be signed with the private part of the asymmetric key of the sender. The public part of the same key is publicly available because it is also the node ID. Therefore, every receiver can verify if a received signed message was sent by the stated receiver.

With these two basic security mechanisms all goals for secure messages can be achieved. Depending on the requirements for a message, a secure message can be only encrypted, only signed or both encrypted and signed.

4.4.3 Secure Storage

The goals of secure storage items are the same as the goals of secure messages: only authorized users should be able to read the content and it should be verifiable who wrote the content.

To achieve these goals the PG used a fixed structure for secure storage items. The fixed structure implies that every secure storage item contains a normal storage item, which is in an encrypted form. For the encryption a random symmetric key is used. This symmetric key is required for decryption, thus, it has to be shared with all authorized users. This is done using an access control list (ACL), which stores the symmetric key encrypted with the public key of each authorized user (as described in 4.4.1). The encryption ensures that only authorized users can read the content. To verify if the content has been written by the owner of the content a signature is used. The signature is also required to detect if a secure storage item has been changed improperly. The encrypted storage item,the ACL and a signature are the components of a secure storage item.

The structure of a secure storage item can be seen in Figure 4.2. $\texttt{Enc}_{\texttt{Pub}}(\textit{Symm})$ describes the asymmetric encryption with the public key `Pub` of the given symmetric key `Symm`. The shown secure storage item is stored at a given ID in the P2P overlay.

4.4.4 Conclusion

By consequently applying common security concepts to the basic functionality of the P2P framework, it is possible to achieve security for messaging and storing of data. This is possible even though everything—messages and storage items—is

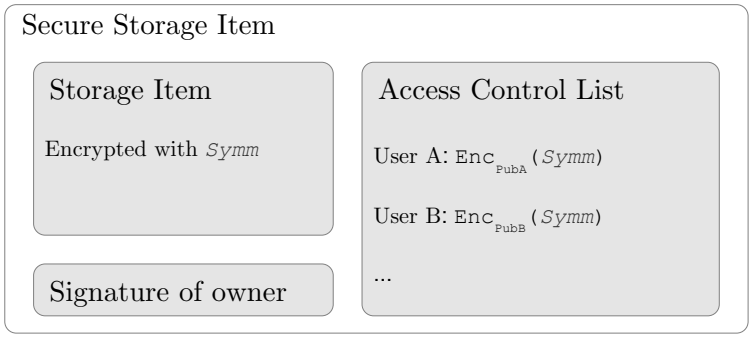


Figure 4.2: The structure of a secure storage item

accessible by everybody. In regard to secure messaging, all plausible use cases are covered with the introduced technique.

The presented solution for secure storage items covers the most needed use case. This use case is that there is one owner of the storage item who has write access to the data item, furthermore, the owner of the item can assign read rights to as many users as possible. There already exists some preliminary solutions for other use cases. To allow write access for multiple users *groups* are required, which are introduced in Section 4.5.2. In addition to the security enhanced traditional storage items, there are also security enhanced distributed data structues. With these it is also possible to implement *append* rights, where users can add items to a list and modify their own items, but they cannot modify foreign items. Distributed data structures are described in detail in Section 4.6.

4.5 User Management

This section explains how users in a P2P framework can be handled. Because of the nature of a P2P framework, identity theft is a common problem. How this problem can be solved is explained in Section 4.5.1. For a couple of situations it is necessary to arrange several users together so that they form a group. This concept is explained in the Section 4.5.2.

4.5.1 Users

Users represent the basic form of an identity. A user can register with a username and a password (the credentials). The P2P framework should support users in a natural way. Therefore, this section firstly identifies the goals of a proper user management. Secondly, the used solution is presented, which is rounded up with an example. At the end, an evaluation assesses the used solution.

Goals

As mentioned earlier, the most important goal is to prevent identity theft. For this, the sender of a message has to be known and verifiable. In general, this can be done with the secure message introduced in Section 4.4.2. However, it is impossible to hinder an intermediate node to modify the sender's ID. So there has to be at least a way to detect if the sender's ID has been modified. To achieve this it is necessary to bind the sender's ID to the cryptographic signature of the message. If the sender's ID is used to successfully verify the message, it is easily detectable if the sender's ID has been changed.

Another important requirement for a proper user management is that every user has a unique identification. On the one hand, it is required that users can store permanent links to other users, e.g. in a friend list. On the other hand, users should have the possibility to change their credentials. Because of this the credentials have to be independent of the user-ID.

Solution

As a solution for the first goal, the PG decided to use the public key of each user also for the node ID. This characteristic (public key = node ID) is the most essential part of the user management. The benefits are as follows:

- The receiver of a message automatically receives the public key by using the sender's node ID as a public key.
- Adversary users cannot modify the sender's ID without being noticeable because the receiver uses the sender's ID to verify the signature of the message
- To successfully modify a message both the content and the sender's node ID would have to be changed. This equals a complete new message by another person.
- By sending an asymmetric encrypted message to any node ID, it is ensured that only the real receiver can decrypt this message because only the user at node ID (= public key) owns the corresponding private key.

The public key is created from the username and the password, thus the public key may change occasionally. Because of this mutability the node ID cannot be used as a unique identification for a user. A unique identification is required if a user references another user somewhere in his storage, e.g. in a friend list. This identification has to be immutable because if a user changes his public key he cannot also change the storage of each user, which references him. The user might not even know where he is referenced. Therefore, the PG introduced a user ID, which permanently identifies a user of the framework. This user ID is only created once and remains unchanged even if the credentials (= public key = node ID) of a user are changed.

As a consequence there are two different IDs for each user: the immutable user ID and the mutable node ID. Having different IDs requires to have an efficient way to convert one to another. The mapping from the user ID to the node ID is very important, therefore, it is stored multiple times at different positions in the network. In case of a potential forgery all mappings could be checked to tremendously increase the probability of achieving a correct answer.

To achieve a correct mapping from the node ID to the user ID is less important. This is because this mapping is normally only required if a user is logged in. The item that maps the node ID to the user ID is stored at the node ID. Therefore, the user itself is responsible for the storage item used to map the node ID to the user ID. Because of this, it is sufficient to only have one item mapping the node ID to the user ID.

Example

Figure 4.3 shows the node ID at which a certain user is logged in and the corresponding items that are used for the mapping between the node ID and the user ID: `ID2UserIDStorageItem` and `UserID2IDStorageItem`.

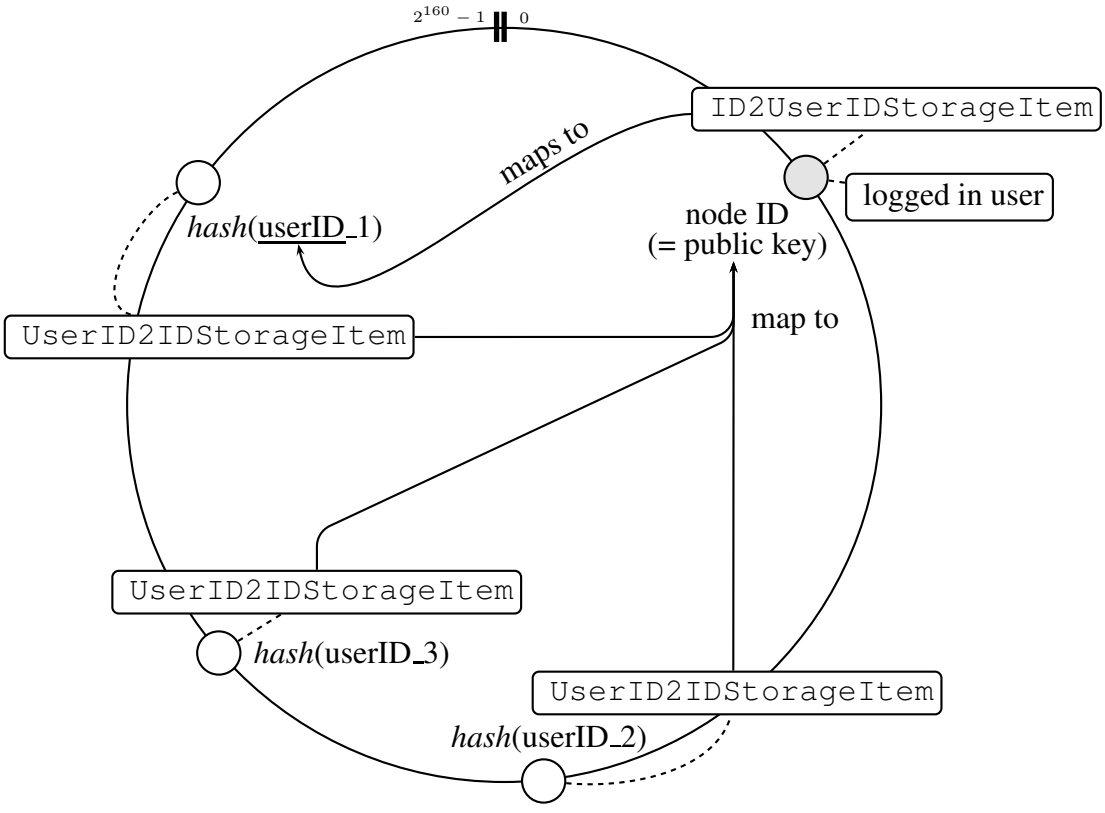


Figure 4.3: FreePastry ring with user-ID and node-ID.

If the public key has to be changed the node ID of that user also has to move to the new public key. Furthermore, all mappings have to be updated and resigned.

However, the items at *hash*(userID_i) remain at the same position because the user ID stays unchanged.

Evaluation

The developed user management solves all identified goals. Currently, there have not yet been deeper thoughts about how many `UserID2IDStorageItem` should be queried. There also have not been any considerations concerning a forgery detection which would switch the ID mapping to safe mode.

The direct link between node ID and public key is a very handy solution. However, if in the near future a 160-bit ECC key should prove to not be secure enough a longer key could also be used although the node ID is shorter. This can be achieved by still using 160 bits as the node ID and attaching the additional bits in an additional field next to the sender's ID to the message.

4.5.2 Groups

A group is the union of several users. The characteristics of a group are similar to a user and in fact a group can often be used in place of a user.

Functionality

Groups are stored at a user defined, unique group ID, which never changes. A group always has a current asymmetric and a current symmetric key. Similar to the ACL (see 4.4.1), the private part of the asymmetric key and the symmetric key are encrypted with the public key of each group member. The public key is accessible by everyone.

There are a couple of considerations that have to be taken care of if a user leaves a group. Firstly, a new current asymmetric key and a new current symmetric key have to be created, so that the user, which is not part of the group anymore, does not have access to the upcoming new data of the group. However, the old keys still have to remain valid because there are for example cases where data encrypted with a group key is signed with the private key of a user. Because of this, it is not possible to reencrypt and resign everything once a user has left the group. As a consequence a list of all previous keys exists. Therefore, if a user in the group is confronted with some encrypted group data, he also needs to know which version of the key was used to encrypt it. Thus, the key version is also stored with the encrypted item.

Fortunately, the joining of a new user involves much less work because no keys have to be recreated. Only all existing keys have to be unveiled to the new user. Afterwards the new user has access to all data of the group.

Managing Write Access With Groups

As already mentioned in Section 4.4.4, groups are required to manage write access for muliple users for storage items. This is achieved by setting the owner of a storage item not to a user, but to a group. Every user of the group can then modify the storage item and resign it with the private group key.

Groups can also be used to support *append* rights. To enforce these the node responsible for the item has to verify the rights. So the responsible remote node is enhanced to also check the signatures of the items before saving. These items have an encrypted content, a signature created with the users private key and a signature created with the group private key. The group signature certifies that the user has the necessary write rights for this group. But a user should only be allowed to override own items. Hence, a check is required if the user who signed the old item is the same as for the new one.

Furthermore, groups also simplify the handling of rights with access control lists, because instead of listing every user of a group, the group itself can be added to an access control list.

Evaluation

Groups are a simple but very powerful concept. At the moment, the most prominent use of groups is to extend write access to multiple people. This concept is currently already used by the distributed data structures, which are going to be introduced in Section 4.6. However, there are a lot more areas that could benefit from using groups. Examples for this are: sending messages directly to groups (which could be used for a group chat) or simplifying read access in access control lists by substituting all users of the same group with the group itself.

4.6 Distributed Data Structures

This section is dedicated to the problem how to manage large data sets. At the moment data storage is only available through a distributed hash table with some replication possibilities. All the user access rights and security measures have to be used explicitly. Additionally, write accesses have to be checked by the saving node which is not done in the DHT. Most important for managing large data sets is the possibility to search. Especially search mechanisms that go beyond simple key-value lookups. Therefore the goals are integrating access rights management including write access on remote nodes and supporting powerful search mechanisms. A secondary non functional requirement is easing the use of the interface between the data sets and the framework user. Thus it should mostly be similar to their local pendant data structures. All in all the distributed data structures have to reduce the burden to handle these topics explicitly.

Now three distributed data structures will be shown. The first two data structures, the distributed linked list and distributed set are shown in Section 4.6.3.

Afterwards the Prefix Hash Tree is shown in Section 4.6.4. But in order to deal with these, it is important to start with a basic example in Section 4.6.1 how to build a distributed data structure on top of a DHT. After that it is also needed to discuss the low level architecture of remote operations in Section 4.6.2. This chapter concludes afterwards in Section 4.6.5. Additional help on basic usage and code examples for these data structures are listed in Appendix A.3.1.

4.6.1 Providing Linkage

This is a first example how a DHT can be used to build a data structure on top. It should provide an easy start since most of the techniques later are based on the idea presented here.

The general idea is to use the unique hash of an object, as a pointer [11]. This way an object can be retrieved from the DHT if the hash is known. Then this hash can also be viewed as content and stored in the DHT under a different hash. So this mechanism provides means to establish a link from one data item to another through saving the hash. With this in mind there are two possible ways to design a distributed data structure.

1. Store the item under a hash that is hard-coded and known to the nodes in the network. These items form hard-coded entry points to start traversing the structure.

2. Store the object under its hash and then store the actual hash in another item. If someone has the item with the hash, it is possible to retrieve the item with the object because the hash reveals the position. Eventually this method ends with an item stored using method number one. Then, to retrieve an item, the application could brachiate from item to item.

Due to the graph structure, on the fly retrieval of an object is only possible if it is stored as an entry point (method 1). All in all, this is similar to the well known object oriented programming (OOP) paradigm. However, many abstraction layers accompanied by a high level of indirection, common in OOP, cause a high level of latency produced through traversing the pointer or hashes in the P2P overlay. In this distributed scenario every level of indirection results in a full 1-to-1 communication over the P2P overlay including multiple hops. But this holds another advantage. Now there is also the possibility to split some large objects like videos into smaller parts. Then we only need one additional object that holds a list of all the part hashes. Again, this means adding another level of indirection. But here it is essential in order to retrieve the parts of the large file in parallel from different nodes. The drawback raises if one of the parts is missing or offline the whole file could not be assembled and used.

This eases the handling of the persistent data and provides a clean abstraction over the P2P overlay. This distributed scheme can be used to store and retrieve whole graphs connected through these hash pointers. But be aware of the latency introduced through indirection and additional layers.

4.6.2 Remote Operations and the Backend

Distributed data structures store data on many different nodes in order to balance storage, bandwidth and other requirements. Therefore a node that wants to operate on that data has to communicate with many other nodes. Thus the lower levels of most distributed data structures are similar; they have to transmit messages. These messages contain a command the other end has to carry out. Despite being in a P2P environment there are these two roles; a server and a client. But here these roles have to be fulfilled by every node. Hence the name *Servent* which is a node that can serve some requests and issue others itself. In the architecture this gets captured through a remote operation client also called the frontend and a backend server that carries out the commands. Both are built on top of a message channel. They provide an interface to perform operations remotely since this is the basic building block for any P2P protocol.

Backend The Backend is the server that receives requests. In this remote operation system the request is an action the server should execute on some data it is responsible for. Thus the data to operate on is already there and no further networking is necessary to get it. So, to use it you have to create an operation message with the name the operation and some parameters of the particular operation. Then the operation message is sent through the message channel to the specified remote node that is responsible for the data we want to operate on. This information will be received by the backend of that node. After that, the backend executes the operation requested. For all operations the backend has to send results and error messages back.

Frontend At the moment, the remote operation frontend works synchronous, which means the execution waits until the remote operation is done. As mentioned, the other node has to send back results. Thus the frontend knows that the operations is completed and it can also hand back results to the user of the data structure. One remote operation can also be sent to multiple nodes. These nodes execute the operation then in parallel. The frontend waits until all remote nodes has answered with a result. If there is a timeout the operation gets repeated. After receiving all the incoming results the frontend has to evaluate them. Due to serving as a general system for all data structures defined later all these operations happen on a level that has no idea about the meaning of the results. Therefore a data structure that is using this remote operation system has to provide an aggregation function that can combine all the results into one. This aggregated result is then returned as result of the operation.

Example

Consider an operation like *contains*(x) as defined in a Java Set class. This operation has to determine if the object x exists in the data structure and returns a boolean that signals so. For a distributed data structure with many items

located at other nodes this is a problem. In order to answer the request it has to contact every one of them in the worst case. But with this remote operation system it can issue a *contains* operation also equipped with the parameter object x. The operation can be sent to many responsible remote nodes at once. The receiving nodes backend executes the command through searching locally for the object x in the appropriate structure. Finishing the local search operation, it sends back a boolean result if x is contained in the local structure. Arriving at the frontend these results can be aggregated with a boolean *or* operation. Thus providing a speedup while also hiding the lower workings for the asking distributed data structure.

Conclusion

This system is the basis for the following distributed data structures which of course have to define meaningful remote operations themselves. As mentioned before it is needed to define matching aggregation functions too. A list of all remote operations for every data structure is given in Appendix D and equally all used aggregation functions are listed in Appendix E.

4.6.3 Distributed Set and Linked List

As mentioned in the beginning of the section the main goals are improved search capabilities and automatic rights management within the data structures. Considering tag search where links to certain content are associated with a tag and the content should be retrievable providing this tag. If all items belonging to one tag are just stored in the DHT with that tag as hash then all these data will be stored at one node responsible for that hash. Thus a single point of failure rises on a node that also gets unevenly more requests than others. Distributed sets can be used to store these tag collections. Now the distributed data structure has to make sure that the data gets distributed among many nodes such that no single point of failure exists and requests are balanced among the nodes. So the main motivation to find a solution to get efficient distributed sets is the keyword search that comes along with it.

Moreover the distributed set also implements the Java collection interface `Set`. Thus it acts completely like a local set and it can be used as one too. The distributed linked list inherits from it and extends the functionality. Technically they operate analogous. So the mechanism description below does not differentiate much between them.

Mechanism Description

The basic idea here is to build small buckets out of the whole data structure and distribute these to other nodes. A bucket has a given size and holds a collection of items. Figure 4.4 shows these buckets and their positioning. The bucket size (s) is a parameter of the data structure and should depend on the size of the

items the user wants to store. Then the data structure is thought as an array and all items from the interval $[s(m-1), sm-1]$ for $m \in \mathbb{N}$ get stored in a bucket located at the hash 'structurename_' concatenated with the interval ID m. If the index of an item is known, the responsible remote node where the bucket resides is known and it can be contacted directly. Therefore the additional latency is just one indirection. This also means that all the operations defined for lists and sets have to be performed remotely on the buckets stored at 'structurename_m' $m \in \mathbb{N}$. That is where the aforementioned remote operations from Section 4.6.2 come in handy. So there is only the need to design a backend that gets the operation message with parameters and performs the operation locally. Thus most of the operations were just sent to remote nodes which then operate on local data structures.

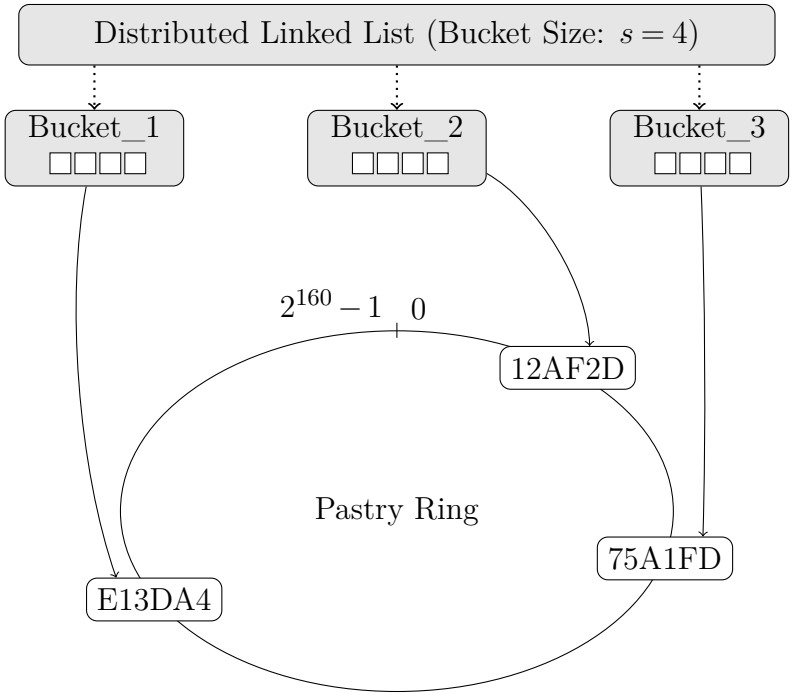


Figure 4.4: Positioning scheme of a Distributed Linked List

Rights Management If the data structure employes rights management all the items are encrypted and then sent and stored. When receiving such an item as the result of an operation it must be decrypted and then handed over to the user. This scheme guarantees that the responsible remote node only handles encrypted items since it should not know the content. That fact leads to problems for operations like the *contains*(x) operation from Section 4.6.2. Since the object in question x gets encrypted before sent to the remote node and all the stored items located on that node are also encrypted, the remote node could run into trouble deciding if x is in the local set not knowing the content. In this case the remote node compares

the encrypted items for equality. Encrypting an item twice should result in the same result. This way the remote node can answer the contains operation without knowing the content.

Initialization Only in some cases the user has to be aware that it is in fact a distributed structure she is dealing with. One case would be some options that determine how often and when the structure synchronizes with the network. Another situation is the initialization of the data structure. If the user creates a specific data structure, be it a set or linked list, that already exists in the P2P network, the created structure acts automatically as it were the existing one. But if you want to find out if the structure already exists there is also a constructor function to test that.

Example

This example considers the operation *removeAll*(c) where the parameter c is a collection of items that should be removed from the data structure. At the beginning the distributed set at the users node sends an operation message to every node responsible for a bucket of this distributed set utilizing the remote operation system described in Section 4.6.2. This operation message contains the function parameter which is the collection of all items due to remove. On the remote nodes the arriving message gets processed by the backend. It applies the removal of all items of collection c to its local bucket. The backend then returns the number of removed items to the requesting node. The frontend of the requesting node, the node that issued the remote command, waits until all results have been returned. Getting all these numbers the frontend starts aggregating them when they arrive. In this case it uses the aggregation function *sum* to get the number of overall deleted items which it returns to the user. All parameter and return values can be looked up in Appendix D.

Conclusion

The distributed sets and linked lists provide a compact yet powerful interface for big data collections. The framework user is freed from explicit rights management and networking. Thus the user does not have to bother about the network nature of the framework. Instead keyword searches with large value collections per keyword are within reach. Additionally Section 4.6.5 provides some lookout for future work concerning distributed data structures.

4.6.4 Prefix Hash Tree

As mentioned in the beginning of the section the main goals are improved search capabilities and automatic rights management within the data structures. An advanced search method not supported by DHTs is the range search. Range searches are needed in order to find content with attributes in some range. For

example a social network could use this to search for profiles with a given age. Also searches for events in time can be seen as range searches. If all items belonging to one age are just stored in the DHT with that age as hash then all these data will be stored at one node responsible for that hash. Thus a single point of failure rises on a node that also gets unevenly more requests than others. The solution below tries to overcome these problems while providing balanced resource requirements for all nodes. Noteworthy it also has to scale well in order to support the large data sets expected to be used.

Mechanism Description

The described solution the PG implemented is the prefix hash tree (PHT) [12]. The tree structure stores key-value pairs whereby keys can be inserted multiple times. But the main difference over the DHT consists in the possibility to query range searches over the keys efficiently. Just to simplify the following explanation, the key alphabet will consists of binary digits. Thus all keys are binary numbers.

A PHT starts as one node stored at 'structurename'. Every key-value pair inserted into the PHT is stored there. If the size exceeds a given split size, that was given when the structure was created, two new children are defined that have to store the data in their buckets. One has to store every item with keys starting with '0' and the other stores all items with keys starting with '1'. These buckets are positioned under the hash 'structurename_0' and 'structurename_1' and the responsible remote nodes manage them. In general there would be $|Alphabet|$ new nodes which are also called leaf nodes (should not be confused with Pastry's leaf set). Now the original root node does not store any items, but if many items get deleted the structure can collapse back to this single node.

Lookup To illustrate a lookup operation consider a well filled PHT like in Figure 4.5, then the items are only stored at leaf nodes in this tree. The tree node labels only show the prefix they are responsible for; the structure name which gets concatenated is omitted. If we want to lookup a key we know it can only be stored at nodes responsible for prefixes of that key. A value with key '010' for example can only be stored at 'structurename', 'structurename_0', 'structurename_01' or 'structurename_010'. There are three methods to perform a lookup.

1. Iterate the possible hashes and ask for the stored items. This approach has high latency but low traffic overhead.

2. Contact all nodes responsible for these hashes in parallel. But remember only one will have the stored items. This method has low latency but imposes high bandwidth usage.

3. Try a binary search by starting to ask for items at a node in the middle of the possible tree. If the node is a leaf node which stores the items it can answer the query. If the node is an inner node we learn that the prefix that

identifies the sought leaf node should be longer. And otherwise, if the node does not belong to this data structure we can conclude that the sought leaf node has a shorter prefix.

Concerning performance, it is important to mention that the first variant is not useful. Due to inner nodes near the root being asked way to often and thus distributing load very unevenly. The P2P framework at hand supports the second and third lookup method. Hence there are options to toggle the optimization of lookup and information retrieval between latency and throughput.

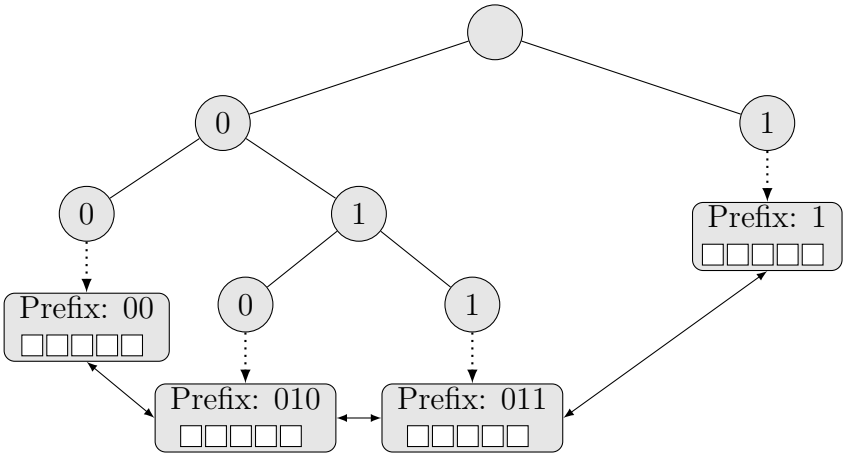


Figure 4.5: Prefix Hash Tree (Bucket size: $s = 5$)

Key Space The PHT supports arbitrary key alphabets. For decimal number keys this would be a character set of all digits from zero to nine. The size of this alphabet and the split size determine the spreading factor or number of children when one node has to split and become an inner node. In order to perform a range search the keys must be comparable. Thus the user has to provide an ordered character set that is sorted from smallest to largest character.

Needles to say the length of the keys should be given since the lookup goes from the most significant bit to the least one. If the key space is to small and one leaf node has split due to size while no additional key digit exists to expand to, the node choses one digit for every value. These digits can be seen as digits after the decimal point. They have no semantical meaning. They only provide the option to further spread the data.

Range Retrieval This is the most relevant use the PHT has to support. For range queries it is important to traverse just the leaf nodes from a starting node to a finishing node determined by the key range. In order to retrieve key-value pairs efficiently from neighboring leaf nodes these leaf nodes maintain the hash of their two neighbors. Figure 4.5 depicts this through links between the buckets.

Thus only two leaf node lookups are required. This is done by first performing a lookup of both borders of the range in question and then traversing the leaf nodes from the low to the high border leaf node. Since this could be a large data set, it is wise to just request an iterator and take as many objects as you need to display at a time. In this case the iterator loads only one bucket and knows the place to get the next bucket if the user needs more; thus also saving leaf lookups.

Example Application

Here it is shown how this distributed data structure can be useful in the context of a social network application. Consider storing the date a user joins the network in Unix time as key and the hash/place of her profile as value. Now you can query the prefix hash tree for all profiles from users that joined between *1.1.2012* and *3.1.2012*. Or since this could be a large data set, just request an iterator and take as many objects as you need to display at a time.

4.6.5 Conclusion

The PHT supports range searches over keys which is definitely a feature needed for some applications on top of this framework. Most impressive for this data structure is the logarithmic lookup time while supporting spreading widely over the P2P network. Concerning robustness, a PHT survives node failures as long as they do not happen in leaf nodes. But this is also covered if the P2P overlay supports replication. Thus it is an advisable choice if range queries are needed. One note though, multi dimension range searches are not supported and much more difficult.

Further Work Some search mechanisms could not be implemented. One such search mechanism is the multiple keyword search. Multiple keyword searches are handy to find a profile which meets more than one criteria. The most promising direction is to include bloom filters to keep the bandwidth down when combining keyword searches. Also substring searches are needed in order to find user names and groups easier when misspelled or incomplete. The PG investigated some methods to do this. The method presented in the seminar phase consisted basically of a keyword search with clever paired substrings.

4.7 Global Aggregation and Monitoring

For most computer systems exist the need for monitoring the overall status. This is needed in order to measure the performance or encounter problems early. In classical centralized systems this is not a big problem since all the used servers are maintained locally. However in distributed systems there are no easy means to supervise a whole network. Caused by the distributed nature of the system, a bandwidth and latency preserving aggregation mechanism is needed. Simply

sending all relevant data to one node is not feasible due to the network traffic overhead imposed on the receiver. Thus the P2P framework needs some global aggregation mechanism that aggregates locally collected attributes over the whole network. Such a system has to combine multiple attributes into one through some aggregation function. One example would be the aggregation of the local download bandwidth as an attribute. And it could be aggregated through the sum function to get the overall bandwidth of the P2P network. In order to provide transparency what is monitored, in the end all the global monitoring data is available on all nodes of the network. So on the one hand this mechanism should be usable by framework users for their own measures and tools. Hence a game plugin could monitor and measure attributes it is interested in. On the other hand there is an additional requirement, posed by the use of this component as measuring tool in hard times. Therefore it has to be robust and should die last, due to the use as failure notification system.

In the following section it is shown how one node handles the attribute passing internally and after that in Section 4.7.2 the mechanism and strategies to combine attributes into one are discussed. Section 4.7.3 depicts how the backend actual manages the aggregation at a node to node base.

4.7.1 Node Internal Structure

This section shows how attributes get passed from layer to layer in one node. There are two directions for attributes to be passed. The first direction is the collection of local attributes and the second one is the transport of the global attributes to the receiver functions of the framework user. Whereby in between is the backend that is responsible for the actual aggregation at a node to node base. The backend is also used in Section 4.6.2 to act as the server part in distributed data structures.

Local Attribute from Sensor to Backend In order to use aggregation, the user has to create an aggregation channel. Fore more information on channels see Section 4.1 on communication principles. These channels allow the user to register sensors that have to provide the local attributes the user wants to be aggregated. These sensors will be asked repeatedly for new attributes. These attributes hold the measurement, its name, unit, aggregation strategy, number of already aggregated participants and the sensor it was measured with. Besides that an attribute could hold any data and only the size would be a restricting factor for this aggregation system. So, at a given time interval, all sensors are repeatedly contacted to hand in all there local attributes. Now the backend aggregates its requested local attributes with the ones from all other nodes which is described in detail in Section 4.7.3.

Global Attribute from Backend to Receiver At a fixed interval the backend receives the global aggregated attributes. Now it distributes them back to all the

created aggregation channels and they filter through which attributes came from their registered sensors. After that, the channel hands them over to the receiver callback the framework user has registered in order to get the globally aggregated data. This filtering ensures that users only have to care about sensor data they disposed themselves. This also means that an aggregation channel only hand out a global attribute if it is based on an original measure taken from one of its sensors. For a framework user it appears that the created aggregation channel has only aggregated its own sensor data and nothing from other aggregation channels of the same node. Whereby all the attribute aggregations were handled all together by the backend.

4.7.2 Aggregation Strategies

In this section the combining of two attributes into one is described. As mentioned before an attribute holds the measurement, its name, unit, aggregation strategy, number of already aggregated participants and the sensor it was measured with. In order to combine two attributes every one of them needs to have an aggregation strategy which specifies the operation to combine them. An aggregation function can also use all of the extra information like unit and aggregated participants. But it will only aggregate with attributes that have the same name, aggregation strategy and sensor they were measured with.

The set of possible aggregation strategies and data types is not fixed and easily expandable. Thus providing a general aggregation framework for all kinds of data. Standard aggregation strategies like *sum*, *minimum*, *maximum* and *mean* are present. Most of them only have to use the measured values of the attribute. Only *mean* uses also the number of already aggregated attributes in order to weight the values which could already be aggregates. For a more detailed description of those standard strategies and additional other strategies have a lock at Appendix E.

4.7.3 Network Topology

As mentioned above, the main requirement for an aggregation system is balanced low traffic overhead, low latency and resilience of node failures. This section describes such an architecture and defines an additional P2P overlay that aggregates attributes provided by nodes. In the end, all the aggregated global attributes are present at every node of the network. This solution imposes only two requirements on the P2P underlay. First is a function to check the own responsibility range and second is a function to route to a specified ID.

The additional P2P overlay consists of a tree structure that aggregates the attributes at every node in parallel. For the following description of the used technique, some reoccurring terms will be defined first.

ID space The mentioned new overlay consists of an ID space of $[0,1)$. Every peers nodeID is normalized over this ID space in order to get the position of the peer which is denoted by p.

Responsibility Function FreePastry is used as the P2P overlay to build upon. There, every node is responsible for a range in the ID space. This function is also used here to determine the responsibility for the additional overlay. It is named: (*ID range*) responsibleFor(*peer*). Every ID of the whole ID space has an associated responsible node at all times.

Aggregation Tree This is the main structure for the network topology. It is a tree structure of node addresses whereby every address in this tree is associated with one real node. One example mapping between these addresses and real nodes is depicted in Figure 4.6.

Splitting Factor k Even Figure 4.6 only exemplifies on a binary tree, the algorithm is parametrized with the trees splitting factor k. Thus every tree node has exactly k children.

Level l The variable l denotes a certain level of the tree. The level start with 1 at the root node level. This means on every level l of the tree structure the whole ID space is split into k^l equidistant domains. Figure 4.6 shows the domains that overlap with the responsibility range of peer p as thick black lines and all other domains with dotted greyed out lines up to a tree level of five.

Domain D_p^l The Domain is a range in the ID space. The domain D_p^l is calculated dependent on the level l and the position of the peer p in question. For calculating the domain of node p at level l, D_p^{l-1} is cut into k equidistant pieces and the piece that contains the ID of the node p is D_p^l. Equation 4.2 shows a non recursive calculation for the borders of the domain.

Domain Key K_p^l In the middle of a domain D_p^l resides its domain key K_p^l. These domain keys are used as nodes in the tree structure. Foremost these domain keys are just addresses and no real nodes. Equation 4.4 shows the calculation function.

Coordinator Coordinators are real nodes that are responsible for a domain key address. They act as coordinator nodes that are located at the domain keys position in the tree structure. Since an address has only one responsible node at a time the mapping from tree node to coordinator is injective. But one real node can be responsible for many domain keys therefore the mapping is not bijective. These coordinators manage to aggregate attributes received from its subtree.

$$min(D_p^l) = \left\lfloor \frac{ID_p}{(\frac{1}{k})^l} \right\rfloor * \left(\frac{1}{k}\right)^l \tag{4.2}$$

$$max(D_p^l) = \left(\left\lfloor \frac{ID_p}{(\frac{1}{k})^l} \right\rfloor + 1 \right) * \left(\frac{1}{k}\right)^l \tag{4.3}$$

$$K_p^l = \frac{min(D_p^l) + max(D_p^l)}{2} \tag{4.4}$$

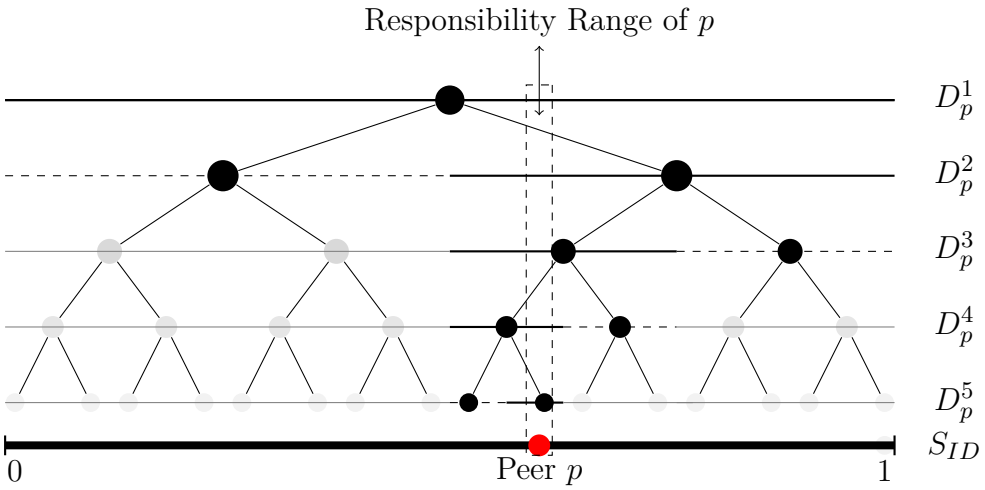


Figure 4.6: Aggregation Tree [10]

Coordinator Mapping Every domain key in this aggregation tree is only an address. Thus a mapping to real nodes also called coordinators is needed. This mapping should possibly be done by any node itself. This way the network traffic for maintenance is kept low. In Figure 4.6 an example is shown how one node can find its own position and role in the aggregation tree. Now, to determine the own responsibilities in this aggregation tree, the node p iteratively calculates the domain starting at $l = 0$ and checks if it is responsible for the address of the corresponding domain key. If $K_p^l \in \text{ResponsibleFor}(p)$ is valid the node p knows it is the coordinator in that tree position and has to manage the associated subtree. With this method every node in the network can calculate its coordinator nodes in the aggregation tree starting from its position and responsible range.

Noteworthy, the functioning of this tree is not coupled to individual nodes. If node failures occur the responsible range changes and other nodes fill in to do

the job. Thus the assignment to a coordinator role of the aggregation tree is not permanent and due to its fast calculation, takes place before every send operation.

Aggregation Method With all this at hand the aggregation is easy. In given time intervals every node sends its measured attributes to its parent coordinator. A message can be sent to the parent coordinator because knowing the own position in the aggregation tree enables to know the parents domain key. And since the P2P overlay can route to IDs, the parent coordinator gets the message if the child routes to the domain key. Internal nodes which receive these messages also aggregate all fitting incoming attributes with their own before sending them to the parent coordinator. At the same time every node has to respond to such incoming messages with an ACK message packed with the global attributes. At the beginning the parent coordinators do not have any global attributes and therefore just send ACK messages. Only after the root coordinator got messages from its children it can return the just created global attributes. This way the root coordinator has valid global attributes in $log_k(n)$ steps. After this, the global attributes get propagated down the tree and every node has them with only $2 * log_k(n)$ time steps latency.

4.7.4 Conclusion

There can be an advantage for solutions deeply integrated into the P2P network but the PG decided to use an additional overlay because it is more maintainable and does not need to apply changes to FreePastry. The presented solution is very scaleable due to the small delay $= 2 * log_k(n) *$ interval between local measure and global measure. It is also resilient to node failure since the tree coordinator roles are discovered and reassigned dynamically without communication outside of FreePastry. There is also an advantage in the modular extendable design because this aggregation system can be used for many different things. Monitoring the network is just one use case of global aggregation and the framework user can implement own information schemes where data gets aggregated. Also the sensor is just a small interface that provides the local attributes. It is easy to implement. In fact most of the P2P framework implements the sensor interface, too. Thus systems built on top of the framework can always measure framework components for error detection and performance analysis.

Improvements One performance improvement is possible through better synchronization of the time interval. Then the overall delay can be reduced. Another improvement rises from the fact that at the moment there is no access control for the aggregation system. This is important because denial of service attacks are possible with a big attribute which gets then relayed through the whole network. Access control can avoid such miss uses.

4.8 Test Plugin

This section describes the purpose and the implementation of the Test Plugin. The usage of the Test Plugin is described in detail in Appendix B.

4.8.1 Motivation

The purpose of the Test Plugin is to have the possibility to test the correctness of the features of our framework and of the social network easily in a real P2P network. It is also used to generate content and traffic in the network to analyze the performance when monitoring these network activities with the monitoring plugin . The main idea of the TestPlugin is, to have one master node, that can remotely control several slaves. After its setup, the master node can delegate commands to multiple slaves. These commands are executed by each slave and generate content in the network. Typical examples commands are, for example joining the P2P network or to change a user's profile picture.

4.8.2 Implementation

The following section describes the implementation of the Test Plugin. It explains how the connections between the master and the slaves are established and what kind of commands are available.

Connection

Since the Test Plugin is used to let the slaves create or join the P2P ring, the connection between the master and the slaves is established in an early stage and therefore needs to be independent of the tested P2P network. This is achieved by using Java Server Sockets. In a normal setup, the master node starts first and listens on a server socket for incoming slave registrations. After the setup, the slaves may join and register to the master. The slaves themselves open a server socket and listen for incoming commands, sent by the master. At the end of the setup, the master has a *slave list*, which contains IP address and port number of each slave. To send a command to a slave, the master opens the socket using the slaves' IP address and port and sends the command as a string to the slave. This string is then interpreted by the slave. Since the master only has to maintain the list of registered slaves and sends only messages of low data size, this approach scales well because the slaves execute the commands all by themselves. A simple configuration file is used by the Test Plugin. It determines whether the executing client application is the master or a slave and contains the IP address of the master, so the slaves can register.

Social network commands

The social network the PG used, offered so called OSGi commands, that can be executed in an underlying console. The purpose of a command is to execute a certain functionality, like registering to the network or sending a friend request. The command *help* lists all available commands that can be executed. In order to execute a social network command, the prefix *execute* is necessary or the usage of the Test Plugin command *doXTimes*. Most of these commands need parameters, for example to specify the user who should receive a friend request. The following command demonstrates, how a client application can logout from the network.

execute LoginPlugin.Logout

An other command sends 10 group messages:

doXTimes 10 5 GroupsPlugin.SendGroupMsg groupName message

The *doXTimes* command repeats 10 times, with a wait interval of 5 seconds. Each time, the command calls another command $SendGroupMsg$, which gets a $groupName$ as well as a $message$ as parameter.

Test Plugin commands

In addition to the existing social network commands, the Test Plugin introduces also several new commands. Most of these commands are executed on the master machine, which manage the master's slave list or delegate commands to the registered slaves. For example the *slavesDo <command>* command delegates the respective *<command>* to all registered slaves, while the *slaveDo <ipAndPort> <command>* command delegates the command only to one specified slave. The previously mentioned *doXTimes* command can be used to repeat a social network command certain times. How this and the other available commands of the TestPlugin work, is explained in detail in Appendix B.1.

Since the social network was developed before the Test Plugin was planned, some social network commands may not work properly in combination with the Test Plugin commands or need some previous configuration. For example, to let a slave create a new photo album for it's user, it first has to execute the *PhotosPlugin.LoadMyAlbums* command. In another example, the slave has to also execute the *ProfilePlugin.StoreMyProfile* command, after it has changed it's user's profile picture with the *ProfilePlugin.EditProfileField ProfileImage picture* command. These two commands were the only commands with this unexpected behavior recognized by the PG. The other used commands — as listed in Appendix B.3 — work proper.

4.8.3 Generation of Content

To enhance the power of the *doXTimes* command, the Test Plugin introduces variables. To avoid executing the same command multiple times with exactly the

same parameters, it is possible to replace certain words with randomly chosen – but predefined – values. These values are defined in text files that are accessed by the Test Plugin. The existent variables and the usage of these variables to generate content in the network are described in Appendix B.2.

4.8.4 Conclusion

The TestPlugin is a powerful tool to easily create a P2P network that can be controlled by one master. On Unix based systems the setup is easy and not too time-consuming, since the slaves just have to start the application and then they register automatically with their IP at the master node. On windows machines the PG observed problems of the slaves to obtain the correct IP to register with at the master. This is due to an inconsistent implementation of the Java method `InetAddress.getLocalHost()` on windows.

Since the resources like group names, messages and pictures for the content generation is outsourced in external text files, they can be adapted without needing to change the code. In future developed plugins can be tested by the TestPlugin, if they offer OSGi commands.

5 Framework Architecture

During the PG, a lot of effort has been put into the development of the P2P Framework. The following chapter describes the approach, the PG has chosen for the implementation and gives an overview about the external components and interfaces of the framework.

5.1 Discussion

The PG started with an existing prototype for a P2P-based, distributed social network, called LifeSocial [11]. It was developed by students in different projects by the supervisor of the PG, Kalman Graffi, and gave the idea for this PG. LifeSocial provided the functionality that is needed to create a P2P-based social network. It was possible to establish a network, register an account, create a profile, add friends, share pictures and send messages. But due to its initial design decisions, it suffered from a slow response time for all network-related interactions, e.g. sending messages or requesting friendships, and a complex internal architecture.

Because the focus of this PG was more on an independent framework for generic P2P-based applications, the PG had to decide which approach to use for the implementation of the framework. Not only should there be a plain framework, but also a social network as a comprehensive showcase. At best, it should even reuse some already existing parts of LifeSocial. After some discussion, the PG considered the following options for the implementation process:

1. Extracting the P2P-related parts of LifeSocial and adapting it to the needs of the PG. This approach would have the advantage of reusing a lot of existing code. But due to the complicated architecture of LifeSocial, introduced by using the OSGi Service Platform [16] (OSGi) as well as Declarative Services, it could be much work to extract some modules and to break or resolve their implicit dependencies to parts of the code that were not necessary for a framework. Also, the performance of LifeSocial is still improvable, because of the used poll-based approach for the message passing instead of the PG favored push-based approach.

2. Creating a new framework from scratch using OSGi as a technology to build independent modular units. This approach would lead to a new, clean and simple architecture. By constantly using push-based message passing, it could also provide a better performance than the simple reuse of existing code. But using OSGi in the framework was also considered to be too much

overhead. Most of its functionality is not needed for the framework. Also, an integration of this new framework into the existing LifeSocial could be a complex task.

3. Creating a new framework from scratch using plain Java only. As written above, a newly designed framework would also lead to a new and clean architecture. Sticking to a push-based approach for message passing and omitting the overhead of additional technologies such as OSGi and Declarative Services could improve the performance, compared to existing code, and also ease debugging of the framework. But the disadvantage of this approach is that this new framework needs to be integrated into LifeSocial. Also, a mechanism to extend or replace certain modules of the framework needs to be developed. Especially the integration into the existing LifeSocial could be a complex task.

After some initial testing, it turned out that the effort for an extraction of the framework from the existing P2P-related parts of LifeSocial was very high. In any case, much of the legacy code needed to be rewritten, so that the complexity of the system could be reduced. As a result, the PG decided to write a completely new framework from scratch, without reusing any existing code. The development should be feature driven through simple example applications, but also allow an early integration of the framework into the existing LifeSocial. The goal was to again have a running social network on top of the framework.

Also, the option to use OSGi as a module system in the framework was rejected. The experience with LifeSocial showed some problems with the usage of OSGi, especially related to the usage of Declarative Services. By using commonly known Design Patterns [9], it is also possible to create an extensible and modular framework, even without OSGi. It even leads to a higher degree of freedom for potential users of the framework. They are not forced to use a certain technology on top of Java.

The implementation of the framework is described in detail in the following sections. The integration with LifeSocial follows in Chapter 6.

5.2 Components

After deciding on a course for the implementation, the PG designed a modular architecture for the framework. The architecture is made up of seven mostly independent components that strongly focus on the individual features of the framework. An overview of the components is given in Figure 5.1.

The component `network` provides an essential part of the functionality of the framework. It is needed to establish a connection to a P2P network or to create a new network if it does not exist yet. The `network` component also contains the functionality to uniquely identify each peer in the network. The implementation of the component is realized using the FreePastry Java library (see Section 3.1).

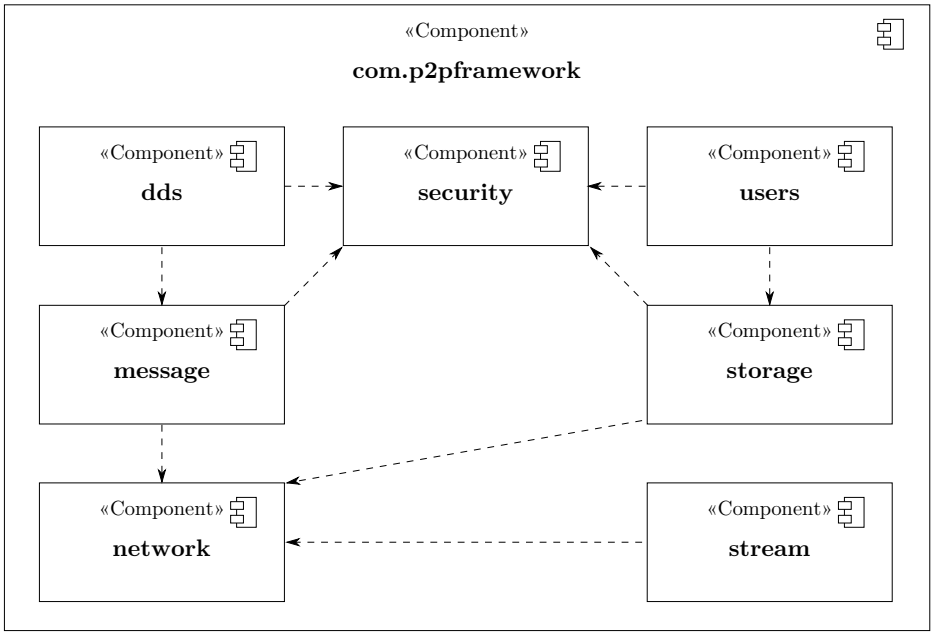


Figure 5.1: Components of the framework

The `message` component realizes another central aspect of the framework. It provides the functionality to send messages in the network. Either directly to a peer or indirectly routed via the P2P network. The component contains also a Publish/Subscribe mechanism and the functionality to aggregate messages in the network (see Section 4.7). The implementation of this component is realized using the Scribe library that is built on top of FreePastry (see Section 4.2).

The `storage` component contains a third essential aspect of the framework. It provides the possibility to store and retrieve arbitrary data that is distributed in the network. Both functionality can be used in a synchronous as well as an asynchronous manner. The implementation of this component is realized using the Past storage system that is built on top of FreePastry (see Section 3.2).

The `security` component can be used to manage keys that are needed for signing end encryption of data. All messages that are sent in the network and all data that is stored can be signed as well as encrypted. For the data items stored in the network, the access can even be controlled via access control lists. The `security` component provides elliptic curves as an implementation for the security algorithms. Details about this implementation are described in Section 4.4.

To manage users in the network, the component `users` can be used. It provides the functionality to register users in the network and to abstract from the peers of the network, such that a user may login from different computers and can change its password or username, but never looses its unique identity. Details about this implementation are described in Section 4.5.

The `stream` component provides the functionality to send streams of arbitrary

data over the P2P network. That could be a simple file transfer between two peers, but also a video and audio stream that is broadcasted to the peers in the network. The implementation of this component is realized using the Splitstream extension for the FreePastry library (see Section 4.3).

And finally, the PG developed the `distributed data structures` (`dds`) component. It provides a set of commonly used data structures, that are handled like their local equivalents, but are automatically distributed over the network. The distributed data structures are described in detail in Section 4.6.

5.3 Interfaces

This section presents a detailed descriptions of the classes and interfaces available for users of the P2P framework. It is divided in each of the components listed above and represents a short documentation of the public interface.

5.3.1 Framework

The `com.p2pframework` package is the central access point of the P2P Framework. Figure 5.2 gives an overview of the classes of the package.

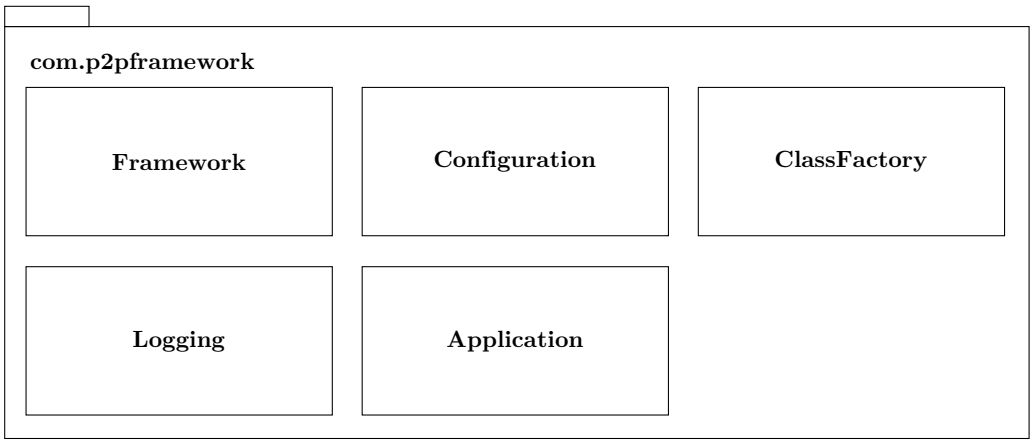


Figure 5.2: Content of package: `com.p2pframework`

The `Framework` class is the main management class of the framework. To use the framework, an instance of this class is needed. It provides access to all separate modules of the framework, but also maintenances the configuration needed to run the framework.

The `Configuration` class is used to hold all necessary configuration values for the framework. It is the only argument of the constructor of the `Framework` class and has to be instantiated before. It contains the address and port of a bootstrap host in the network, the local port and address the framework should bind to,

but also the path to a local storage folder and the name of the currently used `ClassFactory` class. The configuration can be set programmatically, but also be read or written from or to a XML file.

The abstract class `ClassFactory` can be used to exchange individual modules of the framework. It is used from the framework to create instances of the different management classes of the modules. The `ClassFactory` uses the Abstract Factory Pattern [9] and can be configured using the `Configuration` class. To exchange a module in the framework, an own `ClassFactory` class needs to be implemented, or from the existing default implementation inherited and the adequate factory method overridden.

The `Logging` class provides and manages the logging functionality of the framework. It uses the default Java logging mechanism and is used inside the framework to write log messages. Users of the framework are able to set the log level in this class, but can also define an different log file writer than the default console logger.

The `Application` class is an abstract helper class for users of the framework. It provides a quick and easy way to use the framework by defining an initial structure to hook into. The `Application` class is only useful for simple application. If more flexibility is needed, the `Framework` class should directly be used.

5.3.2 Network

An overview of the interfaces of the package `com.p2pframework.network` is presented in Figure 5.3.

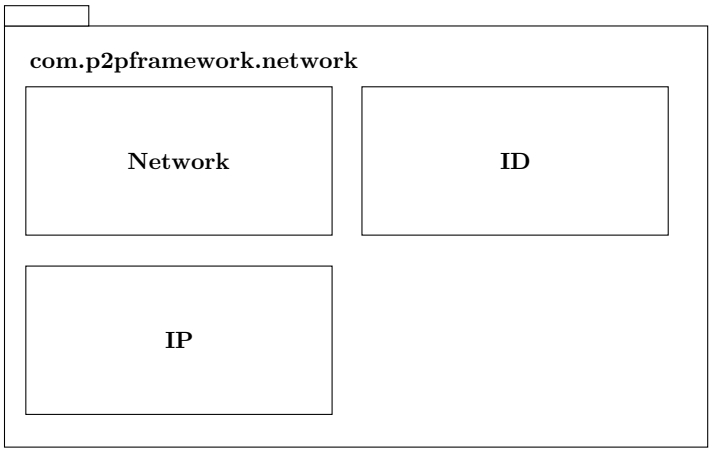


Figure 5.3: Content of package: `com.p2pframework.network`

The `Network` interface provides the central aspect of the package. Its implementation is used to log into an existing network, or to create a new one, if it does not exist yet. The technical parameters for this connection are defined in the `Configuration` class, so that the user only has to give a username and a

password to connect to the P2P network. But, it is also possible to log in as an anonymous user.

The `Network` interface provides the current `ID` in the network. The `ID` is a unique identifier of a peer in the network. It is the target for a message that is sent in the network. In the current implementation, the user is not able to pick his `ID` by itself. It is computed using his username and password.

An `IP` represents the necessary information to establish a direct communication. This means to use the underlaying Internet instead of the P2P overlay. This is useful if message latency is important. In that case a user may want to save the `IP` in order to circumvent refetching it slowly over the P2P overlay.

5.3.3 Messaging

In Figure 5.4 an overview of the interfaces of the `com.p2pframework.message` package is shown.

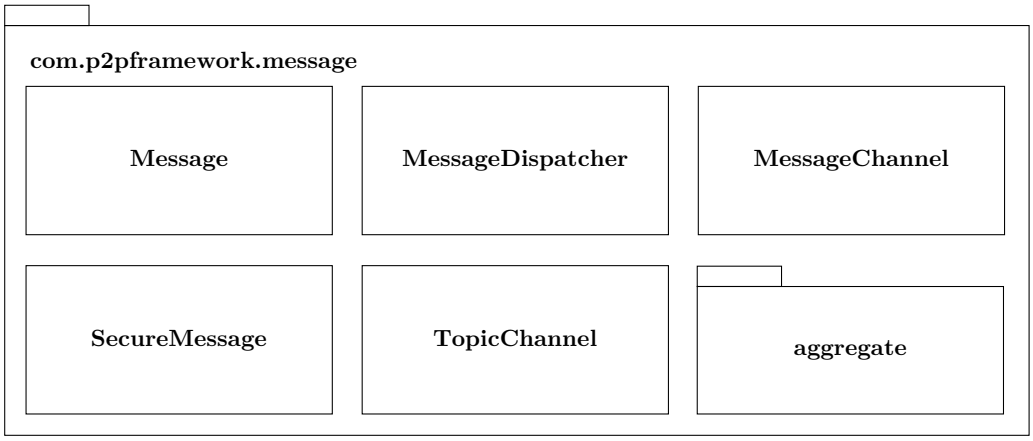


Figure 5.4: Content of package: `com.p2pframework.message`

The `MessageDispatcher` is the main management interface of the package. Its implementation is used to create instances of different types of channels: `MessageChannel`, `TopicChannel` and `AggregateChannel`.

The `MessageChannel` interfaces provides bidirectional 1-to-1 communication. On the one hand it is used to send a `Message` to its defined address, on the other hand it allows to register a listener that gets notified, whenever a new `Message` arrived at this `MessageChannel`. Each `MessageChannel` is identified using a unique name. To communicate through a `MessageChannel`, both partners need to create an instance of a `MessageChannel` implementation as an endpoint using the same name for the channel. Next to the 1-to-1 communication, the `MessageChannel` also provides 1-to-N communication. In addition to a `Message`, the user needs to define a list of receivers to whom the `Message` should be sent.

A third communication pattern is provided by the `TopicChannel` interface. Using a publish/subscribe mechanism, the `TopicChannel` does not only send a `Message` to a single receiver, but to every node that has registered to this `TopicChannel`. To achieve this, every `TopicChannel` is identified using a unique name. To communicate through a `TopicChannel`, all participants have to create an instance of a `TopicChannel` implementation using the same name and subscribe to it. Afterwards every subscriber is able to publish messages to the `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. To send own messages in the network, it is needed to implement this interface. The payload of the `Messages` is automatically defined by the member variables of the class implementing this interface.

The `SecureMessage` is an implementation of the `Message` interface. It provides the functionality to sign and encrypt a `Message` that is transported as its payload. The details about this class are described in Section 4.4.2.

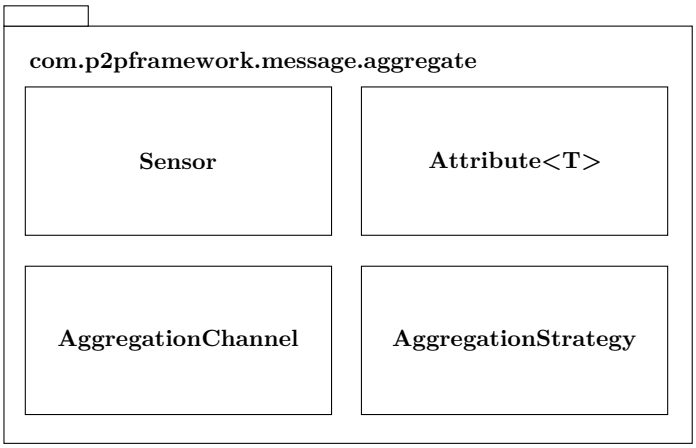


Figure 5.5: Content of package: `com.p2pframework.message.aggregate`

The package `com.p2pframework.message` contains an additional package with the name `com.p2pframework.message.aggregate`. A content overview of this package is given in Figure 5.5. More details can be found in Section 4.7. An `AggregationChannel` can be created with the MessageDispatcher. It can be used to combine data on a global/network scale. To provide data just add a Sensor and to receive combined/aggregated data register a callback. A `Sensor` provides the attributes to aggregate over the whole network. It can be registered with a AggregationChannel. An `Attribute<T>` holds a single measurement, its name, unit, aggregation strategy and the sensor it was measured. These are the elementary items that gets aggregated over the whole network. The parameter T is the type of the attribute data. The `AggregationStrategy` defines the aggregation strategy used to combine to data attributes.

5.3.4 Storage

Figure 5.6 gives an overview of the content of the `com.p2pframework.storage` package. A `StorageItem` is an object that can be stored and retrieved in the

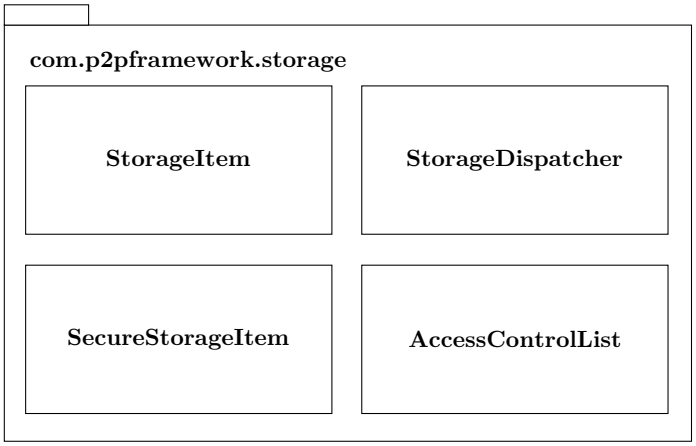


Figure 5.6: Content of package: `com.p2pframework.storage`

network. The `StorageDispatcher` offers several methods to store, retrieve and delete `StorageItem` objects in the network. It supports synchronous and asynchronous execution of the operations. The `AccessControlList` is a list that contains encrypted keys for authorized users. See Section 4.4.1 for further details. The `SecureStorageItem` is an implementation of the `StorageItem` interface. The stored data is encrypted and the item is enhanced with an access control list, which defines the user rights for an item. See Section 4.4.3 for further details.

5.3.5 Security

An overview about the interfaces in the `com.p2pframework.security` package is shown in Figure 5.7.

The `AsymmetricKey` interface represents an asymmetric key. After its initialization, the key can be used to encrypt and decrypt or to sign and verify various bytes or any `Serializable` object. In its current implementation, the `AsymmetricKey` uses an elliptic curve cryptography approach.

Analogously, the `SymmetricKey` interface represents a symmetric key. Once the key is initialized, it can be used to encrypt and decrypt various bytes or any `Serializable` object. In its current implementation, the `SymmetricKey` uses AES as an encryption algorithm.

The `KeyFactory` is a management class that creates instances of implementations of the interfaces `AsymmetricKey` and `SymmetricKey` using the factory pattern [9].

A description of the implementation of the two key interfaces `AsymmetricKey` and `SymmetricKey` is given in Section 4.4.1.

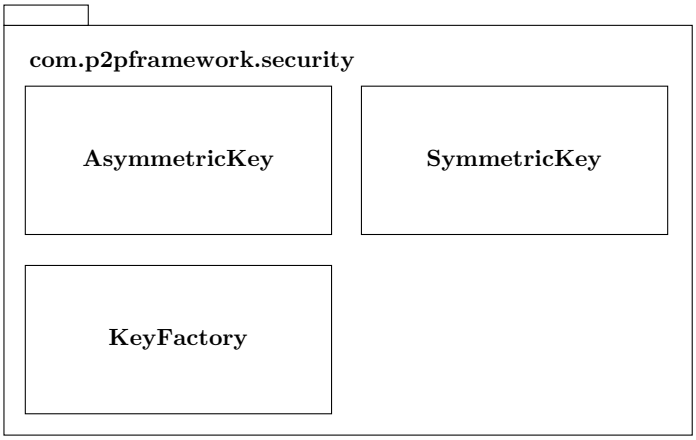


Figure 5.7: Content of package: `com.p2pframework.security`

5.3.6 Streaming

The package `com.p2pframework.stream` enfolds all classes needed to transmit large data. There are basically two kinds of transfer. On the one hand the framework can live stream data presented in ByteArrays or ByteStreams and on the other hand it is possible to transfer whole files. Additionally there are some listeners that can be used to monitor and some callbacks to supervise the process. Figure 5.8 depicts an overview of the classes of this package but for more information take a look at Section 4.3. The `StreamDispatcher` is the main management class to create new StreamChannels or FileTransfers. It also provides a protocol to retrieve files from other nodes. A `StreamChannel` represents a stream tuned to a specific channel name. It can be used to send and receive data to/from this channel. It is a publish/subscribe mechanism with support for high bandwidth. A `RetransmitStreamChannel` subclasses a StreamChannel. So additionally to a StreamChannel it can adapt the bandwidth to the network conditions and retransmit lost packages. A `FileTransfer` represents a file transfer over the network. It manages the transfer and informs listener about the transfer status. A `FileTransferListener` is an interface to listen to file transfers and get informed about status changes, bandwidth, timing etc. It can be used on the sender and receiver side. A `StreamListener` is a listener interface to get informed about the status of a stream. The updates will be send every second if changes happen. A `FileReceivingCallback` is an interface that gets called as the final action of the undergoing file transfer, its node has accepted to receive.

5.3.7 Users

In Figure 5.9 the classes of the user package is summarized. The `UserManager` creates, modifies and looks up user and group related information. It also offers methods to map between the `UserID`, the node ID and the username. For groups

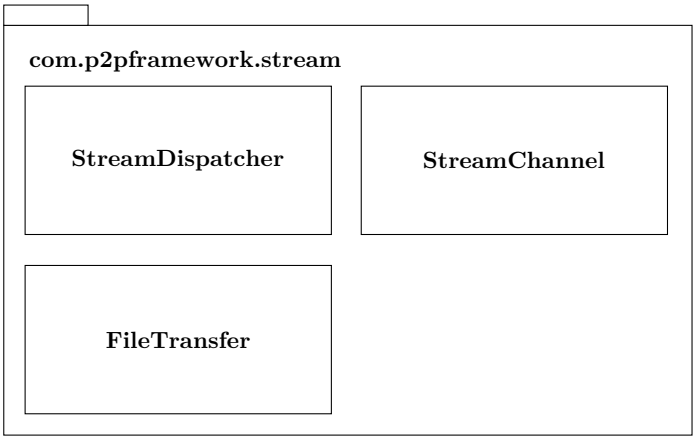


Figure 5.8: Content of package: `com.p2pframework.stream`

the management of user rights is taken care of.

The `UserID` is an immutable ID of a user, which is created by the `UserManager`. It stays the same even if the username or the password of a user is changed. For details regarding the mapping between `UserID` and the node ID, see Section 4.5.

A `Group` represents a number of users that form a group. Every group has an asymmetric key and a symmetric key. To prevent former members of a group to misuse these keys they are updated whenever a person leaves the group.

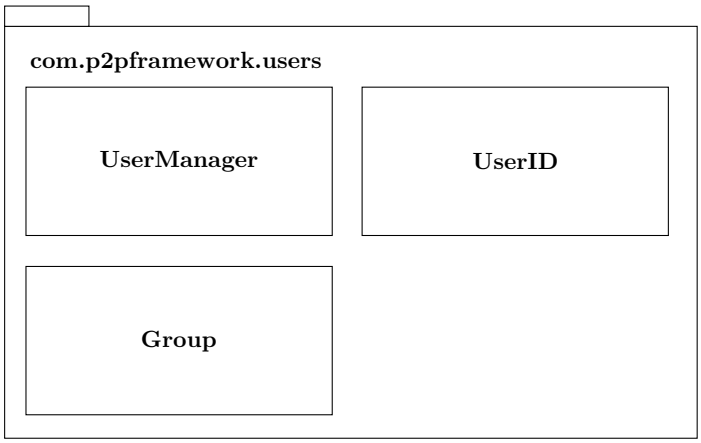


Figure 5.9: Content of package: `com.p2pframework.users`

5.3.8 Distributed Data Structures

The P2P framework provides three distributed data structures. These data structures can be used as if they were local data structures. In fact, they implement

the known interface of their local pendants. But internally these data structures are distributed and spread over many nodes to scale well with large content. Figure 5.10 shows the classes of the enclosing package. Details of the implementation can be see in Section 4.6. The type parameter `E` is the datatype of objects the collection should hold. The `DistributedLinkedList<E>` represents the collection class List in a distributed manner. Nearly all methods of the List interface are implemented. It can be used just like a local List. The `DistributedSet<E>` represents the collection class Set in a distributed manner. Also here, nearly all methods of the Set interface are implemented and it can be used just like a local Set. A `PrefixHashTree<E>` stores and retrieves key-value pairs. Additional it supports efficiently retrieving all values in a given key range. It is also possible to store multiple values under the same key. The `Search` class is a collection of all search methods supported by the framework. Thus, there is an easier accessible class to perform searches as to use the distributed data structures themselves.

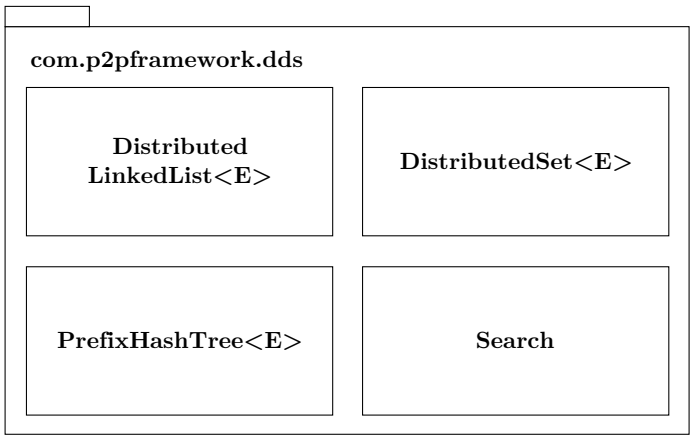


Figure 5.10: Content of package: `com.p2pframework.dds`

5.4 Conclusion

The interfaces presented in this chapter provide a comprehensive framework that allows the rapid development of generic, P2P-based applications. It has been tested using unit-tests and simple example applications for the individual packages. But, it is also used within a distributed online social network, acting as a comprehensive showcase for the framework. The integration of the framework in the existing application LifeSocial is presented in the following Chapter 6.

6 Integration

In the beginning, there already existed a prototype for a P2P-based, distributed online social network called LifeSocial. Because the PG decided to not reuse the P2P-related parts of the core of LifeSocial but to write a completely new framework instead, it was needed to port LifeSocial to the newly created framework to have a social network as an extensive showcase for the features of the framework.

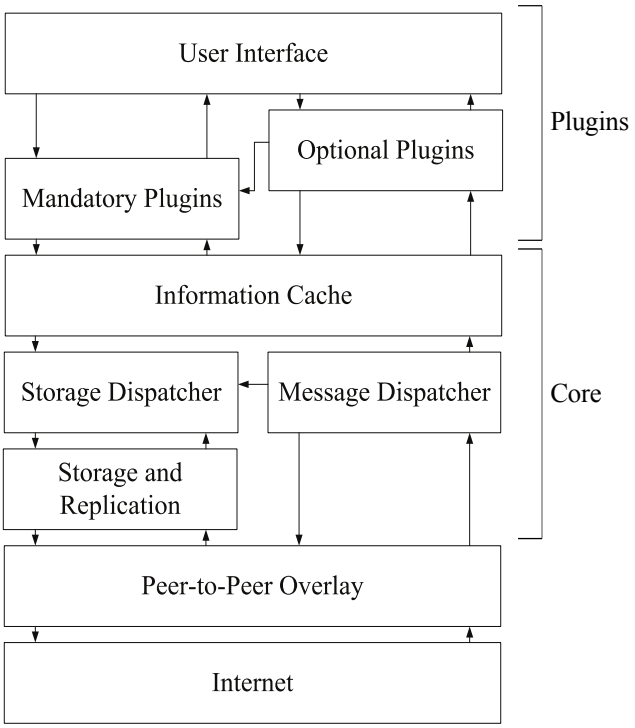


Figure 6.1: Architecture of LifeSocial [11]

The architecture of LifeSocial, as presented in Figure 6.1, helped to complete this rather complex task. The architecture is decomposed into components that are layered on top of each other. Every layer offers a well known service to the layer above and uses the services from the layer below. The *Internet* is the basis for the whole architecture. It provides the physical connectivity for the *Peer-to-Peer Overlay*. The *Storage and Replication* layer uses the P2P network to create a reliable storage for data objects. The *Storage Dispatcher* keeps track of the actual

application data being stored, while the *Message Dispatcher* provides a message exchange between individual peers. *The Information Cache* stores previously requested data and thus makes it instantly available for further requests. The plugin layer, consisting out of *Mandatory Plugins* and *Optional Plugins*, contains the actual functionality of the application. It is split into plugins to allow an easy extension of the application. Every plugin is expected to also have a *User Interface* that is decoupled from the actual functionality of the plugin.

In a first step, it was needed to identify all parts of the code that were directly using the native P2P network library. A manual analyzing process resulted in the components *Message Dispatcher*, *Storage Dispatcher* and *Network* (which is not presented in the abstract architecture shown above). The PG replaced all of these modules with own implemented wrappers, that provide the interfaces needed for LifeSocial to still work as intended, but internally use the P2P Framework as an implementation. After replacing the old components with the wrappers, LifeSocial was still functional but no longer dependent on the native P2P library used before.

In a second step, the usage of the new wrappers in the different LifeSocial plugins were analyzed and the corresponding code rewritten, to directly use the functionality of the framework instead of the wrappers. Especially by porting the login procedure of the social network to the framework, the *Network* component was no longer used and could be completely removed. As a side effect, also two different functionalities of the old version of LifeSocial, connecting to a network and logging in to the network, were merged by the integration to a single functionality. Unfortunately, the usage in and the dependency of the different LifeSocial plugins on the *Message Dispatcher* and the *Storage Dispatcher* was so high that the PG decided to not port all plugins to the new framework, but instead stick on using the developed wrappers. For this PG, there was not enough time to accomplish this comprehensive task.

An overview of the different components after the integration of the framework into LifeSocial is presented in Figure 6.2. The *Message Dispatcher* as well as the *Storage Dispatcher* have been moved into the wrapper package. All wrappers by itself are now using the P2P Framework which provides all functionality related to the P2P network.

In the last step of the integration, the plugins described in the following were ported to the framework, for example to enhance their performance. Furthermore, there have been also new features that were added to the social network.

Since the original chat plugin used the poll approach, users received the chat messages with a big delay. Because this was an unwanted behavior, the chat plugin was the first plugin to be enhanced by using the push approach of the framework. Therefore, the *MessageDispatcher* introduces an interface, where other plugins can register with a *PluginID*, to get notified if there are new messages. The chat plugin implements this interface and therefore gets immediately notified, if there is a new chat message.

In addition, the messaging of the groups plugin was also enhanced. In the

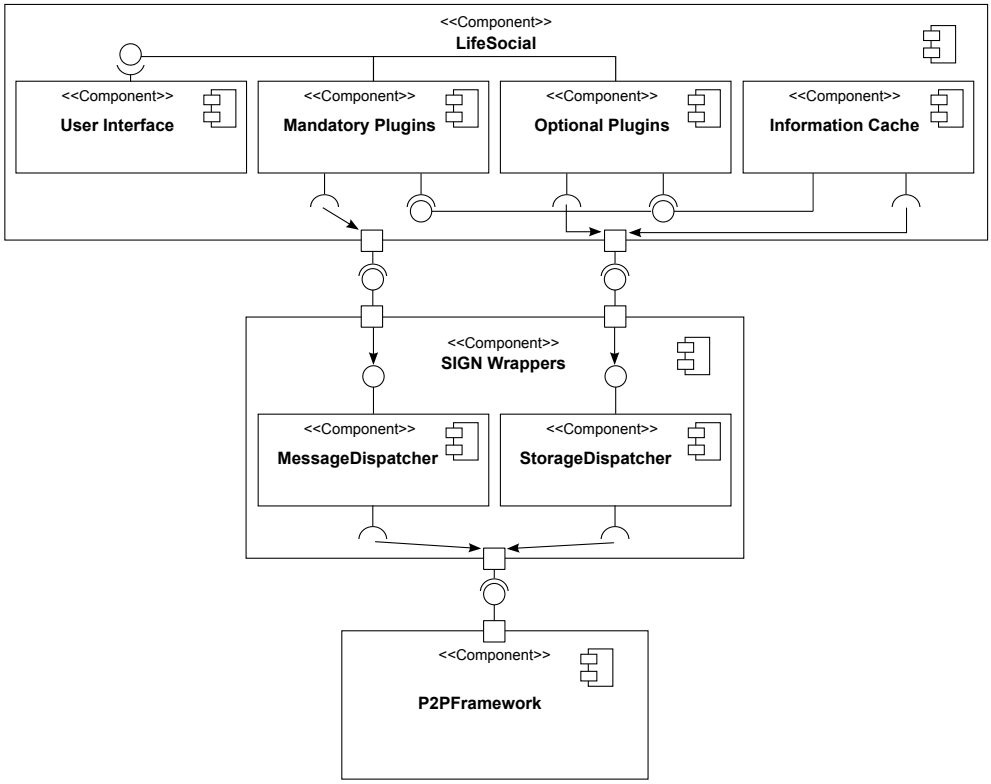


Figure 6.2: Components of the social network after the integration

original groups plugin, when a group message was sent, it was sent to all users that join this group. After this message was received by a user, it was stored on the local file system of the user. This has the following two main disadvantages: First, if the user switches from one machine to another, the group messages are inconsistent and not the same on all machines. The other disadvantage is, that new users that join the regarding group, will miss all previous messages sent to this group. These problems could easily be solved, by using one of the distributed data structure—the distributed linked list. Each group has its own distributed linked list, where all the messages are stored. The identifier of the distributed linked list consists of a constant prefix, which determines it is a groups message list, and the name of the group.

The monitoring code that was used in a previous version of LifeSocial was also adopted to the new structure of the system. The code that aggregates all measured values from all nodes in the network has been newly implemented in the framework (see Section 4.7) and a new back-end that collects the local values from the framework has been developed. Only the graphical user interface as well as a component that writes all measured values to log files has been reused.

One completely new plugin for LifeSocial is the wall. Similarly to a guestbook and in the style of the Facebook wall, the LifeSocial wall can be used to write

status updates on the own wall, or to write posts on friends' walls. In addition, it is possible to comment on all wall posts. The new wall plugin is heavily using the new distributed data structures and thus also serves as a showcase for them. In practice the performance of the wall proves to be quite good.

Figure 6.3 shows a running instance of LifeSocial. The interface can be arranged at one's will. Here, it is split into four areas and each area shows one widget. In the top left the welcome screen can be seen. Below that is a profile, which includes some personal information and a profile picture. In the top right, there is the friend list of the logged in user. Finally, in the bottom right the current user just started a chat with a buddy from his friendlist.

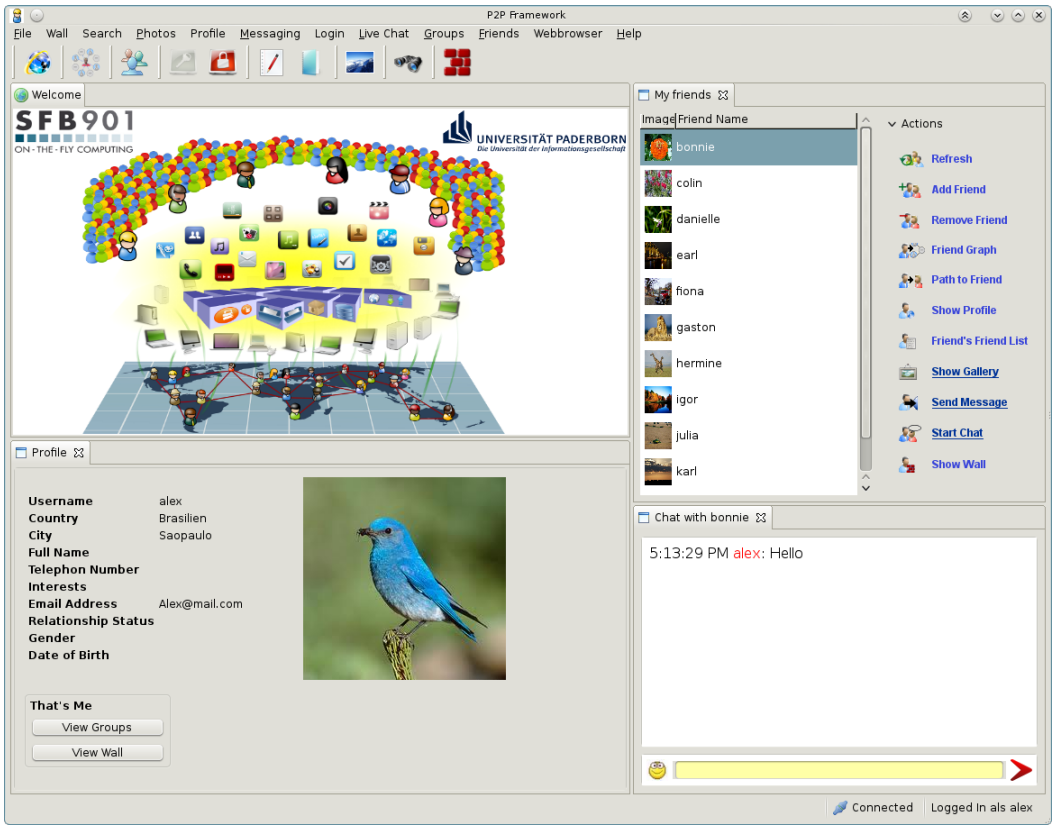


Figure 6.3: Screenshot of LifeSocial

The integration of the framework into the existing online social network LifeSocial was not easy to accomplish. Thorough planning and the development of the framework towards the integration into LifeSocial helped to achieve the goal. All parts of LifeSocial that were previously using the native P2P library were replaced with the components of the framework. The PG first provided wrappers for the framework, but then started to port plugins to directly use the framework. Unfortunately, the porting of individual plugins was only possible for an initial set of plugins, as listed above. The remaining plugins still depend on the wrappers and could be ported in the future.

The online social network that resulted from the integration of the framework into LifeSocial was still functional, but benefited from a major performance improvement for all network-related interactions. This greatly improved the overall usability of the application.

7 Test Documentation

The following chapter describes the different tests that were executed by the PG. It first presents the goals the PG focused on for the testing, then continues to describe the executed tests and ends with a brief analysis of the results.

7.1 Goal of Testing

The overall goal of testing was to test and measure the performance and reliability of the P2P network. One advantage of a P2P systems is the scalability. The more peers contribute storage and bandwidth to the network, the better the reliability and performance should be. A further aspect was to check whether the framework, plugins and the GUI are functioning in the right way as expected.

7.2 Tests

To check the correctness and measure the performance of the framework, the PG performed various kinds of tests on parts of the distributed framework as well as the social network built on top.

Unit Tests In order to verify the correctness and to check the robustness of new features of the framework, the PG used unit tests. Each developer contributed his own unit tests for the features he implemented. Before committing new code to the Git repository, all developers were encouraged to run the complete set of unit tests to ensure that the new code does not interfere with already existing code. To cover most of the features of the framework, the PG developed about 90 different unit tests. All of the unit tests succeeded.

Work Station Tests After new features were implemented to the framework or to the social network, the PG regularly tested these new features in a small P2P network with up to 5 peers, using their own work stations. The main goal of these tests was to ensure the correct functioning of the new features and to find severe bugs. In case of any bugs, the bugs were fixed immediately and after that the work station tests were repeated. This procedure repeated until no more bugs in the new feature were found.

Pool Area Tests Several tests were conducted in order to check the performance of the P2P network. The PG chose a set of computers from the university's pool area and performed the tests.



Figure 7.1: The pool area PCs used for remote testing

The first pool area test was planned to take place on five pool PCs with Windows as operating system. This test failed, because the windows firewall did block all the communication between the peers. Due to security aspects and access restrictions, the firewall could not be deactivated.

For the second test, the PG planned to use up to 40 pool PCs with Linux as operating system and to use the Test Plugin for the first time. The chosen time-slot was the evening, because this guaranteed to not disturb other students. Unfortunately, after setting up the master and the slave peers, the pool PCs automatically shutdown at 8 p.m. due to a shutdown script of the administration group of the university. Rebooting the machines was not possible because of an additional outage of the central file servers at the same time. Thus, the test could not be continued.

After this, the PG decided to perform a new test with pool PCs on another day in the afternoon. Since there were other students also using these PCs, only 8 pool PCs were used for this test. These PCs had the following hardware and software configuration:

- Processor: Intel i7-2600
- RAM: 8 GB
- Operating System: openSUSE 11.4 (x86.64) with Kernel 2.6.37.6
- Network Interface: Gigabit Ethernet
- Java VM: Java 6, version 7 (build 1.7.0-ea-b104)

The pool PCs served as peers in the P2P network and as slaves for the Test Plugin. In addition, the PG used a university-provided server as a dedicated bootstrap server and one of the work stations of the PG as a dedicated master for the Test Plugin. All participating PCs used a global IP address belonging to the University of Paderborn address range. The PG executed a test unit on February 24th, 2012. The test plan that contained the executed actions on the peers for the test unit was as follows:

- All slaves register to the network and login
- All slaves send friend requests to the users of the other slaves
- All slaves change and store their user's profile picture
- All slaves send 10 group messages
- All slaves create 3 photo albums and add 15 photos
- All slaves write 25 wall entries on friends walls
- All slaves comment on 25 wall entries at friends walls
- All slaves logout

To simulate a real user environment, each slave waited a random time between 0 and 5 seconds before it executes the next command. Therefore, it could happen that most slaves were already posting wall entries, while a single slave was still sending group messages. The PG repeated several test units with the test plan described above. Depending on the result of a test unit, some bugs were fixed and the test plan was, if necessary, slightly adjusted.

Figure 7.2 shows how the network traffic increases during a successful test unit. The graphic shows how the network traffic increases in the beginning, when the slaves change their user's profile picture and reaches a maximum after 140 seconds, when the slaves upload photos to their user's albums. When most slaves have uploaded their photos, the traffic decreases. The next two peaks (at 200 and 270 seconds) are due to wall posts and comments. The signal to logout from the network was sent to all slaves simultaneously, but only after all slaves completed the previous actions. This waiting time explains the decreasing and low point around 300 seconds. After the slaves received the signal to logout, the remaining slaves try to replicate the data which increases the traffic once more. But since they receive the signal all at the same time, there is not enough time to save all the data.

More plots of the monitored data can be found in appendix C. An explanation, why they are not analyzed in detail, can be found there, too.

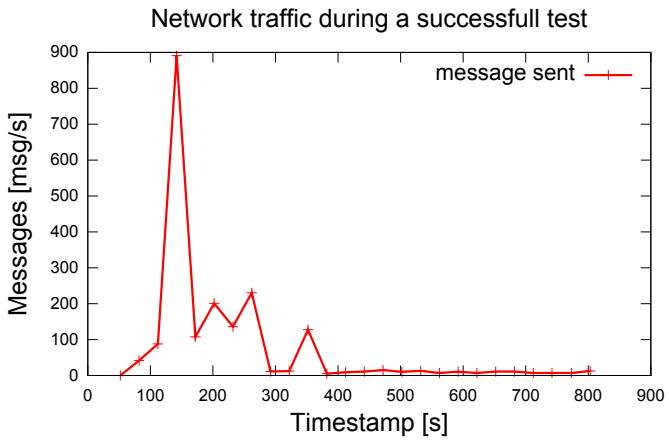


Figure 7.2: Network traffic during a successful test

7.3 Conclusion

The pool area test was helpful to find and fix smaller bugs in the Test Plugin as well as in the application itself. Unfortunately, there were too few peers in the network to make statistically relevant performance analysis. Since the PG did not have the time to do another large dimensioned test, this still remains as an open task for future work. Furthermore, the task to do a big test over the Internet with foreign users to test the availability of the distributed stored data over a long period with alternation of day and night is still outstanding.

8 Organization

The PG was supervised by Dr.-Ing. Kalman Graffi who already worked on a distributed online social network with previous student groups in Darmstadt. After being left by some shaky candidates, the PG only consisted out of five team members: Andreas Volk (team leader), Daniel Breitlauch, Kavitha Jagannath, Pascal Kallien and Jonathan Schlüßler. All team members and the supervisor are shown in Figure 8.1.

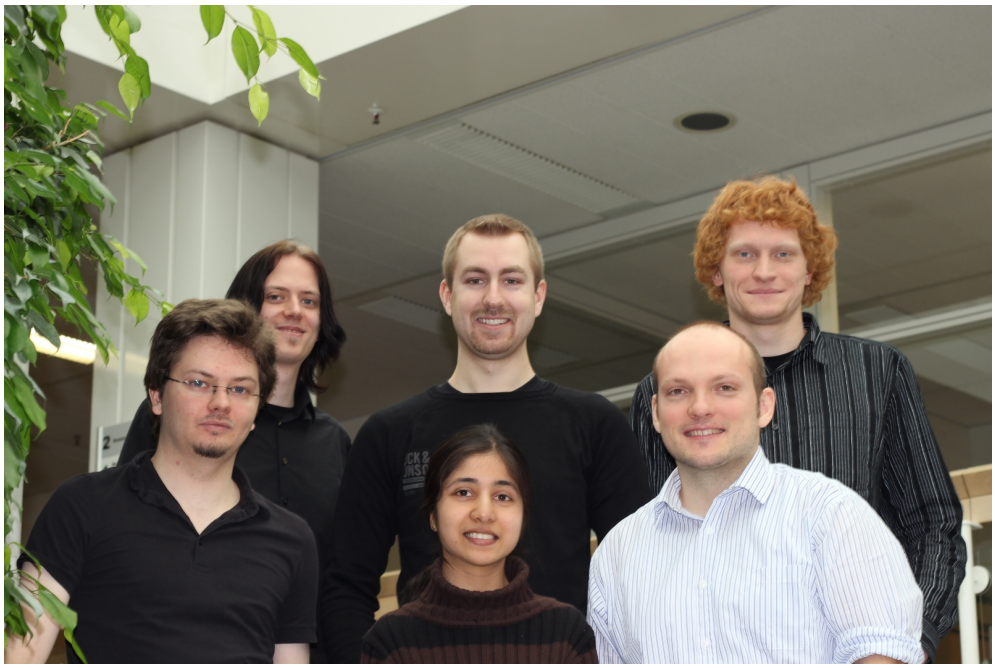


Figure 8.1: All members and the supervisor of the PG

The organization of the PG was split into two phases: A seminar phase and an implementation phase. The two phases as well as some details about the planning and the working behavior of the PG are presented in the following sections.

8.1 Seminar

The seminar phase was the beginning of the PG. To get familiar with the different topics that were needed to work on the project, weekly lectures were scheduled and held by the supervisor of the PG. He talked about the foundations of P2P

technologies, social networks in general, as well as project management. In addition, each member picked a topic that was needed for the project and created a short paper about it. The following presents a list of papers that were created in the seminar phase.

User models and requirements Kavitha wrote about the requirements of a distributed social network with respect to the user and provider point of view. The functionalities of the social network sites, service qualities the users expect, the maximum tolerated load on each users device and some of the non functional requirements that the users expect from the providers. The workload statistics of the usage pattern that helps to model the behaviour of the user was analysed with respect to the view of the provider.

PAST – Replication and Data Storage in FreePastry Pascal explained the task of data replication in a distributed online social network. Data replication is the mechanism of storing the same data on several different peers. This is needed, to ensure that the distributed data is accessible, even if a responsible peer may go offline anytime.

Distributed Data Structures for Online Social Networks Daniel elaborated possible extensions of existing P2P systems, such that they provide persistent distributed data structures. It was shown what possibilities exist to design such distributed data structures. Therefore the paper presented different data structures and their application in order to get better search algorithms in P2P systems. Main topics encompass distributed link lists, sets and prefix hash trees. Additionally addressed were the need for non-probabilistic searches that yield in certainty. At last some insides on attack scenarios and problems when dealing with this kind of data structures were presented.

Essential Security Aspects for Distributed Online Social Networks Jonathan identified important privacy and security issues in P2P overlay networks, with a particular focus on distributed online social networks. Several privacy and security requirements were identified and approaches to realize these requirements in a distributed online social network were pointed out. In addition, several attacks, including the Sybil and Eclipse attacks, and existing countermeasures were examined. The introduced approaches and countermeasures were also evaluated for the practicability to be used in a real distributed online social network.

Services, Apps and OSGi Andreas got familiar with the OSGi Service Platform. It defines a dynamic module system for Java and allows modular units of a system, so called bundles, to be installed, updated and removed at runtime, without restarting the whole system. The paper sketches an approach how to use OSGi for a distributed P2P framework, but also reasons how to efficiently deploy bundles in a P2P network without the need to have a central repository.

The seminar phase ended in a two day workshop in the University. Becoming a sort of specialist, each member of the PG presented his paper to the whole group at the first day of the workshop. At the second day, the PG discussed about features needed for the framework and the social network as well as a possible architecture. After the first initial brainstorming about the goals of the PG, different roles were assigned to the people of the PG.

8.2 Implementation

The seminar phase was followed by an initial preparation phase for the implementation. The PG created a feature list for both, the framework as well as the social network. Each feature was described in detail and was assigned with a priority. It was also needed to get familiar with the current state of LifeSocial and to do some research about required or optional technologies. A heated discussion was about whether to stay with the current Eclipse based GUI or to create a new HTML-based GUI. Due to time constraints, the PG finally decided to stick with the current GUI and as a result of that to focus on the development of the framework instead of new, enhanced features for the social network. Working in this phase of the PG was mostly done individually from home. This was rounded up by weekly meetings for the whole group, which gave a platform for extensive discussions.

With the beginning of the second semester of the PG, the actual implementation phase started. At first, the PG identified the initial required features for the framework that were needed to replace the current core of LifeSocial. As a result, the functionality to connect to a network, send messages and store data in the network were selected as an initial feature set. Driven by these features, the PG developed the required modules in the framework. In addition to that, for every feature of the framework an example application was created to demonstrate the functionality of the feature.

After the initial version of the framework had been finished, the integration of the framework into LifeSocial and a further enhancement of the framework with additional features started in parallel. At that time, the PG was divided into two groups, each of them working on a separate line of development. The first group worked on the integration into LifeSocial (as described in Chapter 6), while the other worked on new features of the framework like security (cf. Section 4.4), distributed data structures (cf. Section 4.6), or streaming (cf. Chapter 4.3). After the initial integration of the framework into LifeSocial had been finished, and LifeSocial was again working as intended, the two separate lines of development were merged again. The additional features of the framework were integrated into LifeSocial and new plugins for LifeSocial were developed. A detailed description of this integration process is given in Chapter 6.

During the coding phase, the whole PG met for two full days a week at a pool area in the University to work on the project. Next to the additional work at home, the PG also had a weekly meeting with its supervisor to discuss current topics,

but also to keep him informed about the progress. Because much of the discussion in the group was already done during the workdays, the meetings turned more and more into simple status reports and were finally abandoned to save time and be able to put more effort in the development. The supervisor still got an update of the progress every time he visited the PG sitting in front of his office, but he also got a status report sent by mail, if he was not available for a longer period of time.

During the workdays, the PG tried to establish a positive team atmosphere. Having lunch together and also a regular coffee break on the afternoon helped a lot. But also the learning phases for an exam or vacations were respected and the missed working hours were caught up at a later time. Unfortunately, the PG was only able to make a few social events, outside the workdays. But also on this few events, the PG had great fun together. And it helped to strengthen the team spirit.

At one time during the implementation phase, the PG was asked to summarize their current progress and the status of the framework in a presentation. The presentation was given on the 10th of November, 2011 and showed a framework that was capable of doing the basic intended functionality of creating a network, sending messages and storing data in the network. Each of these functionalities was presented with an own example application that was developed during the creation of these features.

At the end of the implementation phase, the PG presented the distributed online social network that was ported to use the developed framework on the world's largest computer fair, the CeBIT. The prototype got a lot of positive feedback from both, professional as well as private visitors and was even featured in two newspaper articles and a radio interview. The attendance at CeBIT was a major success and a great experience for every participant. A group picture of some of the members of the PG at the CeBIT 2012 is shown in Figure 8.2.

8.3 Planning

During the implementation phase, an often recurring task was the planning of the PG. An initial project plan has been created right after the seminar phase, but it was only a snapshot at the current time of the planning. The project plan had to be adapted around every two or three months. The reasons for the recurring adaption were versatile: Newly gained experience showed that a planned task will take longer than expected, but also because the PG decided to strengthen the focus on the framework instead of the social network and not to work on the GUI. Also, at some point the requirements changed. The initial project plan did not include the security aspects, because a secure version of LifeSocial was promised and the included security mechanisms could have been reused. Since this version was never delivered, the PG had to develop and implement their own concepts of security in the framework.

To communicate the planning of the PG to the whole group, two different

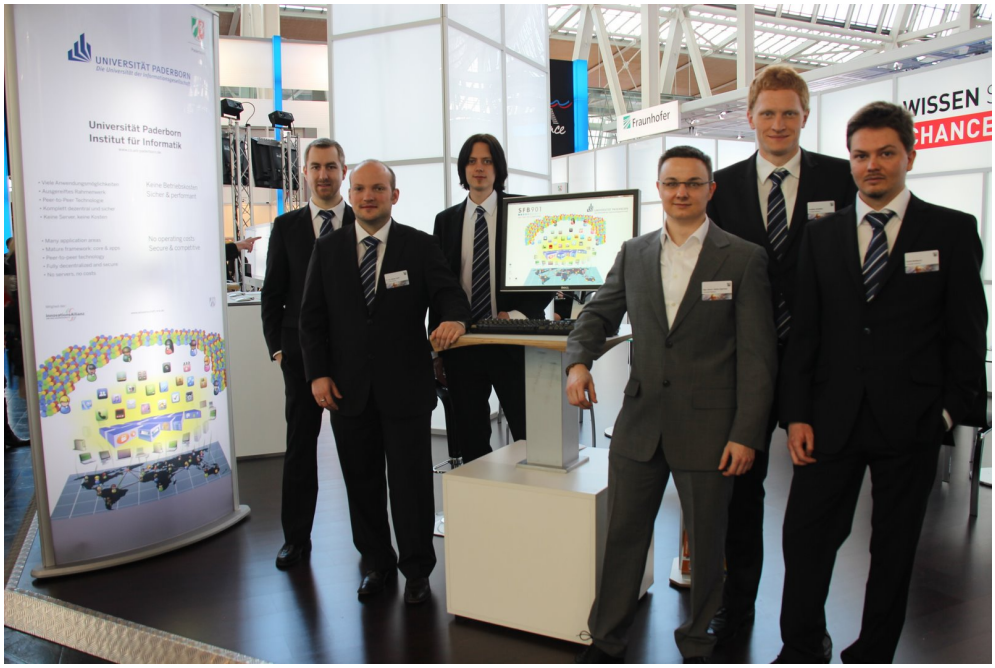


Figure 8.2: Members of the PG at CeBIT 2012

approaches were used. The short-term planning, daily to weekly, was done using a white-board standing next to the work area of the group. All daily tasks were listed there, but also an agenda for the next week. The white-board was the central point of information for anyone who wanted to know, on what the PG was currently working on.

In contrast to that, the long-term planning, of the PG was completely done digital. The PG used a Google Calendar for all of their dates and the university-provided trac system to keep track of their tasks. It also listed the milestones that were set for the PG, as well as their progress. An overview of these planned and achieved milestones of the PG is given in the following.

Basic Prototype

The Basic Prototype milestone was mainly a planning and research milestone. The PG needed to get LifeSocial running on all operating systems and understand the principles of it. For this a list of required features in a social network was created. With this list and the experiences the PG gained in experimenting with LifeSocial, a plan on how to proceed was created. The Basic Prototype milestone has been finished at the 31st of August, 2011.

Framework Functioning

The goal of the Framework Functioning milestone was to create a framework that provides all features that were needed to replace the current core of LifeSocial.

The PG decided to start from scratch because they did not want to use OSGi and Declarative Services in the framework. The task to extract the core from LifeSocial and remove the Declarative Services would have been too complex and time consuming. The PG also wanted to have a clean separation between the framework and the application built on top, which at the moment was not the case in LifeSocial.

The features that were needed to be implemented in the framework were identified as only the functionality to create a network and to store data in the network. In addition, the PG also implemented other basic aspects of the framework. The functionality to send messages in the network, a publish/subscribe mechanism and a solution to provide huge-bandwidth multicast streaming.

To show the capabilities of the framework, the PG created four example applications that showed different features of the framework. A simple *Chat* used PubSub and Messaging to send text messages over the network. An *Address Book* used the Storage capabilities to create a distributed address book that could be edited by multiple users in a network. With the *Tic Tac Toe* game two users could play over a network. It used the PubSub and Messaging capabilities of the framework. The *File Streaming* example used the Streaming feature of the network to send files to multiple nodes in the network. The Framework Functioning milestone has been finished at the 17th of October, 2011.

LifeSocial Integration

The goal of the LifeSocial Integration milestone was to replace the current core of LifeSocial with the framework. This task depended on the functionality of the framework and therefore could start right after the Framework Functioning milestone.

Since LifeSocial was built in a modular manner using OSGi and Declarative Services, the PG replaced the core Services with their own Service implementations that provided the required interface, but internally used the framework. The PG started replacing the `org.lifesocial.core.network` component. This was followed by the replacement of the `org.lifesocial.core.middlestorage` and the `org.lifesocial.core.messagedispatcher` components. In an additional step, the `network` component could be eliminated completely. A detailed description of this process is presented in Chapter 6. The LifeSocial Integration milestone has been finished at the 7th of December, 2011.

Advanced Framework

The goal of the Advanced Framework milestone was to enhance the framework with features that were not provided by the core of LifeSocial, but may be of interests for future applications that are built on top of the framework. The milestone was planned to be in parallel to the LifeSocial Integration as it is independent of it.

The PG added the functionality to do rights management using Access Control Lists (ACL) to the framework, provided the capabilities to encrypt and sign messages and stored items. It also created a set of distributed data structures and included a basic search mechanism. The Advanced Framework milestone has been finished at the 19th of January, 2012.

Beta Testing Candidate

The goal of the Beta Testing Candidate milestone was to create a candidate of the social network that could be used for a beta test. The milestone was planned to start right after the LifeSocial Integration has been done, because it builds on the plugins created there.

With the candidate developed in this milestone, the PG planned to do a distributed test. For this, a test plugin was created that helped to test the functionality of the different plugins of the social network remotely on different nodes. Next to the development of the test plugin, the PG also enhanced the performance and functionality of the existing plugins and created new plugins for the social network. The Beta Testing Candidate milestone has been finished at the 15th of February, 2012.

CeBIT Release Candidate

The goal of the CeBIT Release Candidate milestone was to fix all bugs that were encountered while using the Beta Testing Candidate. The social network should be stable and in a "finished" state to be presented to a broad audience.

The CeBIT Release Candidate milestone has been finished at the 3rd of March, 2012. The prototype of a distributed online social network was presented at CeBIT 2012 without suffering from any major issues.

Documentation

The goal of the Documentation milestone was to finish the documentation of the framework and of the social network, to create a final presentation as well as the final report for the PG. It was the last milestone of the PG and planned to start after the CeBIT Release Candidate milestone. The Documentation milestone was planned to be finished at 31st of March, 2012.

9 Conclusion

This chapter concludes the work that has been done in this PG. The following section shows reached achievements and compares them to the set goals of the PG. Afterwards, Section 9.2 elaborates on open problems, ideas and drafts that could not be implemented, but should be addressed in the future. At the end the final report is summarized in a

9.1 Achievements

The PG achieved its goal to create a generic framework for distributed, P2P-based applications. The framework supports all required basic functionality such as network creation, different kinds of messaging, and storing data; as well as advanced functionality such as streaming, security, distributed data structures, advanced user management, and search. Some of this functionality is realized by completely new concepts, whereas others use modification of existing concepts.

The framework allows for an easy creation of all sorts of applications that may benefit by the use of general P2P concepts. To showcase the abilities of the P2P framework several example applications have been developed: a simple chat application, a tic-tac-toe game, an address book application, and a console based streaming application.

The second goal of the PG, was to create a distributed online social network. This goal has also been achieved using the P2P framework. For this, the prototype of the existing distributed online social network LifeSocial was ported to use the P2P framework of the PG. This resulted in some great performance improvements, which made many of the existing functionality usable after all.

Furthermore, LifeSocial could also benefit from the advanced functionality of the P2P framework. All messages are now sent signed and encrypted and some problems concerning the loss of data could be solved by using the distributed data structures.

Also, some additional plug-ins for LifeSocial have been developed. A wall similar to the Facebook wall, makes heavy use of the new distributed data structures. With a new test-plugin, distributed testing of LifeSocial is made very easy by sending commands to all registered slaves from a single master.

The presentation of the improved distributed social network at the world's largest computer expo, CeBIT 2012, was the final highlight of the PG. The presented prototype received a lot of positive feedback, and thus motivated all team members.

9.2 Future Work

Due to the limited number of members in the PG, it was not possible to do all tasks that the PG actually would have liked to do. On the one hand, there have been tasks that have been known from the beginning, which either had a low priority or could not be solved because of time constraints. On the other hand, new tasks have been identified. This section gives an overview about the open problems and tasks in the areas GUI, LifeSocial and P2P framework.

9.2.1 Graphical User Interface

In the opinion of the PG, the most important task is to improve the GUI. This point has also been mentioned multiple times at the CeBIT. There have been a couple of ideas at the beginning of the PG, however, these were dropped for allowing the PG to focus on the P2P framework. The conclusion of the PG was that a HTML5 GUI with AJAX technology should be used. The communication to the backend server could be done using HTML5 sockets. This has several advantages:

- complete separation of the GUI and the backend
- no need to run a complete webserver
- easy adaptation of the user interface
- mobile clients can connect to a remote backend

9.2.2 LifeSocial

There is, of course, still a lot of functionality that could be added to the distributed online social network. A lot of the functionality is already given in the P2P framework but not not yet used by LifeSocial. Some ideas for new functions of LifeSocial include: a news feed, multi user chat, video chat, advanced search, rating content, friend suggestions, file transfer, or several games. Eventually, it would be nice to support various third-party applications and make them available via an “app market”.

9.2.3 P2P Framework

The P2P framework also still has open tasks, some of them are existent because of the used overlay system—FreePastry. FreePastry was created without paying attention for important security issues, and thus is vulnerable to many attacks. These attacks include the Sybil attack [7], the Eclipse attack [15], and general routing attacks. As soon as a malicious node is in the network, it can dramatically disrupt the whole network. However, the developer of FreePastry state that “subsequent releases will support strong security and allow deployment on insecure

networks where nodes may fail arbitrarily." [1] But as FreePastry has not been actively developed since 2009, it might be a valid consideration to completely switch to another overlay system.

FreePastry is also not supporting heterogeneous nodes, which is especially important for supporting mobile devices. At the moment, all nodes get the same load and they have to share the same amount of storage. This open problem would be partly fixed by using an HTML5 user interface, which enables mobile devices to connect to a remote backend.

Other open tasks for the P2P framework include:

NAT support The P2P framework only works with external IP addresses, thus NATs are not supported.

Block direct access to storage There is no mechanism yet that checks for storage item's node ID collisons. The consequence is that all data can be overwritten without further checks.

Windows firewall There are open problems with the Windows firewall, thus the firewall has to be disabled to properly use the P2P framework.

9.3 Summary

The PG significantly improved the existing distributed online social network by adjusting the existing prototype to a new P2P framework. With these adjustments, the prototype of LifeSocial is finally in a state where it can be showcased to a broader audience. This showcase was successfully accomplished at CeBIT 2012. Eventually, the work of the PG showed that a distributed online social network based on a P2P overlay is a feasible alternative to traditional solutions using the common server-client architecture. Furthermore, the developed generic P2P framework can also be used for several other types of distributed applications.

However, there are still several open problems and tasks that have to be approached. With more work spent in both, the P2P framework and the online social network, a competable solution is quite plausible within the next couple of years.

Appendices

A How-to Manuals

The following chapter contains a collection of small manuals. It includes a step-by-step guide, how to setup a development environment for the social network, teaches some basic fundamentals about how to use the P2P framework, but also describes how to execute a distributed test using the Test Plugin.

A.1 How To Setup LifeSocial

LifeSocial runs on all major operating systems. To get the source code running with Eclipse involves following steps:

1. Download Eclipse RCP Version (Indigo[1] works for sure, the Helios version does not work flawlessly)
2. Start Eclipse
3. In Preferences → Java → Compiler → Errors/Warning? → Deprecated and restricted API → set "Forbidden reference (access rules)" from "Error" to "Warning"
4. In Preferences → Plug-in Development → API Baselines → Options → set "Missing API baseline" from "Error" to "Warning"
5. Import all LifeSocial projects: `org.lifesocial.*`, `com.p2pframework.*`, `com.sign.*` and `quap2p.dims.util.ds.osgi.core`
6. Build all projects
7. Open a compatible product configuration in `org.lifesocial.ui` → ProductConfigurations
8. Select the "Dependencies" tab and click the "Add Required-Plugins" button
9. Save the product configuration file and start LifeSocial by right-clicking on the file → Run As → Eclipse Application.

If there is no compatible product configuration, a new product configuration can be created by adding all required plug-ins in the "Dependencies" tab of the closest fitting existing product configuration.

[1] The newest version at the time of the PG

A.2 Basic Framework Usage

The following section describes the basic usage of the framework. It shows how to connect to a network, send and receive messages, and store and retrieve items from the network.

A.2.1 How To Initialize the Framework

Initializing the framework is quite simple. Only an instance of the Framework class as well as an instance of the Configuration class is needed.

```
1 Configuration config = new Configuration();
2 Framework framework = new Framework(config);
```

Listing A.1: Initializing the framework

Listing A.1 shows the Java source code for this basic initialization of the Framework. Line 1 creates the Configuration and Line 2 creates the Framework. No further parameters are needed at this point.

A.2.2 How To Connect

To connect to an existing network, the framework needs to get initialized as shown above. In addition, the configuration needs to be enhanced by network specific parameters like the address and port of a bootstrap node in the network. To create a new network, the address of the current host needs to be defined as bootstrap address. Establishing a connection to a network needs the username and password of an existing user.

```
1  // [..]
2  config.setBootstrapAddress("localhost");
3  config.setBootPort(9001);
4  // [..]
5  Network network = framework.getNetwork();
6  try {
7      network.connect("MyUser", "MyPassword");
8  }
9  catch (ConnectionFailedException e) {
10     // [..]
11 }
12 catch (UserNotRegisteredException e) {
13     // [..]
14 }
```

Listing A.2: Connecting to a network

Listing A.2 shows the source code to connect to a network. In addition to the code of Listing A.1, the address of the bootstrap host is set in Line 2 and in Line 3 the port to connect to. In Line 5, an instance of the Network is retrieved from the framework. It is used to establish the actual connection in Line 7. The connect method takes a username and a password as arguments and throws two

exceptions, a developer has to deal with. The ConnectionFailedException is thrown, if there is no bootstrap node reachable using the given address and port. The UserNotRegisteredException is thrown, if no user has been registered with the given username and password combination. A user can also be registered automatically, if a third, optional parameter is given.

A.2.3 How To Send A Message

To send a message in the network, it is needed to create a MessageChannel using the MessageDispatcher. The message by itself is represented by a class that implements the Message interface. A developer has to define the content as well as the receiver of the message. The message is sent using the MessageChannel.

```
1 // [..]
2 MessageDispatcher msgDisp = framework.getMessageDispatcher();
3 MessageChannel msgChan = msgDisp.createMessageChannel("MyMessages");
4
5 Message msg = new GenericMessage<String>("Hello World");
6 msg.setReceiver(framework.getNetwork().getOwnID());
7
8 msgChan.sendMessage(msg);
```

Listing A.3: Sending a message

Listing A.3 shows the source code to send a message in the network. Line 2 is used to retrieved an instance of the MessageDispatcher from the framework. Line 3 creates a new MessageChannel using the string “MyMessages” as an identifier for the channel. The actual message is created in Line 5. A predefined, generic Message implementation is used to send the string “Hello World”. Line 6 sets the receiver for the message. In this example, it is the id of the current node in the network. The actual sending of the message is accomplished in Line 8.

A.2.4 How To Receive A Message

To receive a message from the network, a MessageChannel is used. Next to the creation as shown in A.2.3, a callback needs to get registered. It gets called every time a message arrives.

```
1 // [..]
2 MessageDispatcher msgDisp = framework.getMessageDispatcher();
3 MessageChannel msgChan = msgDisp.createMessageChannel("MyMessages",
4         new MessageCallback() {
5             @Override
6             public void receiveMessage(Message message)
7             {
8                 System.out.println(message);
9             }
10 } );
```

Listing A.4: Receiving a message

Listing A.4 shows the source code to receive a message from the network. In Line 2, an instance of the MessageDispatcher is retrieved from the framework. It is used to create a new instance of a MessageChannel in Line 3. The identifier is the same as given in Listing A.3. In addition, a second parameter is given that implements the MessageCallback interface. For this, Line 4 defines an anonymous class whose receiveMessage method simply prints all received messages to the standard output.

A.2.5 How To Store Data

To store data in the network, the StorageDispatcher is used. The data by itself is defined by a class that implements the StorageItem interface. The complete class is stored in the network and can be retrieved later again.

```
1  // [..]
2  StorageDispatcher storage = framework.getStorageDispatcher();
3
4  StorageItem item = new GenericStorageItem<String>("Hello World");
5
6  try {
7      storage.storeItem(item, "MyItem");
8  }
9  catch (StoringFailedException e) {
10     // [..]
11 }
```

Listing A.5: Storing data in the network

Listing A.5 shows the source code to store data in the network. In Line 2, an instance of the StorageDispatcher is retrieved from the framework. The data to store in the network is defined in Line 4. A predefined, generic class is used to store the string "Hello World". In Line 7, the data is stored using the string "MyItem" as an identifier in the network. The storeItem method throws a StorageFailedException, if the given data could not be stored, e.g. if the connection has broken down.

A.2.6 How To Retrieve Data

To retrieve previously stored data from the network, the StorageDispatcher is used. To access the data, the previously defined identifier is needed. Listing A.6 shows the source code to retrieve data from the network. The instance of the StorageDispatcher is retrieved in Line 2. This instance is used to retrieve a StorageItem from the network. The identifier given as an argument is the same as the identifier in Listing A.5. If no data is stored at the given identifier, an ItemNotFoundException is thrown. By doing so, it is ensured that there is always a valid instance of a StorageItem returned.

The code presented in Listing A.6 uses a synchronous approach to retrieve the data from the network. The method `retrieveItem` blocks until the data has fully

arrived. In contrast, there also exists a synchronous variant of the `retrieveItem` method that takes a callback as a second argument. This method does not block and calls the callback as soon as the data is available. Also, it does not thrown an exception.

```
1 // [..]
2 StorageDispatcher storage = framework.getStorageDispatcher();
3
4 try {
5     StorageItem item = storage.retrieveItem("MyItem");
6 }
7 catch (ItemNotFoundException e) {
8     // [..]
9 }
```

Listing A.6: Retrieving data from the network

A.3 Advanced Framework Usage

Next to the basic usage of the framework presented above, the following sections will explain the usage of some advanced features of the framework.

A.3.1 How to use Distributed Data Structures

This section depicts how to use distributed data structures. Listing A.7, A.8 show the use of a Distributed Linked List. After that the use of a Prefix Hash Tree gets shown. The Distributed Set can be used analogous.

Distributed Linked List Listing A.7 shows the creation of a DLL with rights management. Therefore a group with the name given in Line 3 is created in Line 5. The creator of the group is automatically admin. When creating the DLL in Line 8 the group name under which the data structure is managed is given as the last parameter. Other parameters are the bucket size defined in Line 2 and the name of the data structure.

```
1 // [..]
2   int bucket = 10;
3   String group = "Groupi";
4   UserManager um = framework.getUserManager();
5   um.createGroup(group);
6
7   DistributedLinkedList<String> list =
8   new DistributedLinkedList<String>(framework, name, bucket, group);
9
10  list.add("2");
11  if (list.contains("2"))
12    System.out.println(list.size());
13 // [..]
```

Listing A.7: Usage of a Distributed Linked List

Creating an DLL with the same name means that they are automatically the same structure. They are also kept consistent between all nodes. In Line 10-12 shows that all methods can be used like on a local Java List.

Now, the DLL supports also more than the simple Java List interface. Listing A.8 shows some advanced use cases. Here, in Line 4, a whole local Collection gets added to the Distributed Linked List. One of the most important method is the iterator. Given that the DLL is filled with large amounts of data retrieving all items is infeasible. In Line 6 the iterator is created and afterwards it is used to retrieve item after item. Internally the items gets loaded in whole buckets. The iterator supports, as shown in Line 10, the removal of the last item retrieved. Line 13 shows the creation of an array out of all elements of the DLL. As mentioned before the iterator is better suited for retrieval because it only requests as many items as needed. All items of a big collection can not be possibly shown in the GUI. Thus it is not advised on big collections.

```
1  DistributedLinkedList<String> list = // [..]
2    Collection<String> localList = // [..]
3
4  list.add(localList);
5
6  Iterator<String> i = list.iterator();
7  while(i.hasNext()) {
8    String lala = i.next();
9    if (lala.equals("hit"))
10     i.remove();
11 }
12
13 String arr[] = list.toArray();
14 for (String item : arr)
15   System.out.printLn(item);
16
17 // [..]
```

Listing A.8: Usage of a Distributed Linked List

Prefix Hash Tree The fundamental difference consists in the stored data. A PHT sores key-value pairs and allows range searches over its keys. First of all the PHT needs an alphabet that defines the possible keys. Line 3 to 5 create this alphabet which is a vector of characters. The order of the characters in the vector is also the order used to determine greater or smaller keys in the key range. In Line 8 is the creation of the PHT with a bucket size of *10* items and a binary alphabet. Here the PHT stores *Integer* values. After that a toggle for the lookup strategy is switched to parallel lookup which is latency friendly but utilizes much throughput.

```
1  // [..]
2    int bucket = 10;
3    Vector<Character> alphabet = new Vector<Character>();
4    for (int i=0; i<2; i++)
5      alphabet.add((""+i).charAt(0));
6
7    PrefixHashTree<Integer> tree =
8      new PrefixHashTree<Integer>(framework, name, bucket, alphabet);
9
10   tree.setOptimizeLookupForLatenzy(true);
11 // [..]
```

Listing A.9: Usage of a Prefix Hash Tree

After the initialization of the PHT it can be used to store key-value pairs as depicted in Listing A.10 Line 3. It is important to use equally long strings to represent the keys because they are read little endian, with the most significant bit left. If starting zeros are omitted the key would get a larger number. But it is no real problem to start with a large key space. Now Line 4 performs a range search providing to keys and retrieving every key-value pair in between. Then the keys are omitted and stored in a collection. A better solution to retrieve large numbers of items is the iterator. Line 7 shows the creation of an iterator for the same key range. Later the iterator is used to retrieve the pairs and check for the item with value *51* which gets deleted afterwards. Noteworthy there could be many values associated with one key.

```
1    PrefixHashTree<Integer> tree = // [..]
2
3    tree.add("00001", 51);
4    Collection<Integer> d = tree.getRange("00000", "00010").values();
5
6    Iterator<Entry<String, Integer>> it =
7      tree.getRangeIterator("00000", "00010");
8
9    while (it.hasNext()) {
10     int value = it.next().getValue();
11     if (value = 51)
12       it.remove();
13   }
14 // [..]
```

Listing A.10: Usage of a Prefix Hash Tree

A.3.2 How to use Streaming

This section shows how to create a streaming channel, sending data and receiving a stream. Additionally it is useful to be informed about working stream channels; thus Listing A.14 depicts how to do this.

The following Listing A.11 shows how to create a stream channel. The StreamDispatcher needs two arguments the name of the channel and the number

of stripes to use. Stripes are independent distribution routes (see Section 4.3). After that the channel accepts some configurational settings where the maximal bandwidth to use, redundancy of stripes and the use of checksums can be set. Redundancy describes the amount of stripes sending the same data. And checksums get sent with the data such that the stream has a small amount of forward correction.

```
1 // [..]
2   StreamDispatcher dispatcher = framework().getStreamDispatcher();
3   StreamChannel channel =
4           dispatcher.createStreamChannel("jackieChan", 4);
5
6   channel.setBandwidth(500); // kByte/s
7   channel.setRedundancy(2);
8   channel.setUseChecksum(false);
9 // [..]
```

Listing A.11: Create stream channel

After creating the stream channel someone might have to send something. Listing A.12 provides some insights how to use the channel for sending data. The most elegant method is the use of BufferedInputStreams. These class can be created from a file. Then the BufferedInputStream just has to be connected to the channel. This connection defines where the channel gets its data to transmit. The only thing left is starting the transmission with the *startPublish()* method. Setting the bandwidth while transmitting is of cause also possible.

```
1   StreamChannel channel = [..];
2
3   BufferedInputStream in =
4     new BufferedInputStream(new FileInputStream(path));
5
6   channel.connectBufferedInputStream(in);
7
8   channel.startPublish();
9   channel.setBandwidth(channel.getBandwidth() + 10);
```

Listing A.12: Stream data from a file

The receiving node also has to create the StreamChannel but connects a BufferedOutputStream. In Listing A.13 it is shown how to save the stream directly to a file. Hence the use of a FileOutputStream in Line 5.

```
1 // [..]
2 StreamChannel channel = [..];
3
4 BufferedOutputStream out =
5   new BufferedOutputStream(new FileOutputStream(path));
6
7 channel.connectBufferedOutputStream(out);
8 // [..]
```

Listing A.13: Receive data from a stream

As mentioned, Listing A.14 depicts a method to get status reports of an ongoing transfer. Here an inner class gets used as a callback. This inner class is an anonymous class and has access to all variables of the outer class. So there is no need to create extra file for this class it can be defined inline. In Line 4, the channel registers the callback inner class while in Line 7 and 12 the callback functions are defined. These functions get called if some status of the channel changes. This snippet saves the number of transferred bytes in Line 9 and reports the status together with the utilized bandwidth in Line 14ff.

```
1  // [..]
2    StreamChannel channel = [..];
3
4    channel.addListener(new StreamListener() {
5      private long kbytes = 0;
6
7      public void transfered(String chan, long bytes, boolean send)
8      {
9        this.kbytes = bytes / 1000;
10     }
11
12     public void bandwidth(String chan, long bandwidth, boolean send)
13     {
14       System.out.println(
15               "Transfered: " + kbytes + "kB" +
16               "Datarate: " + (int) bandwidth + " kB/s"
17                 );
18     }
19   });
20 // [..]
```

Listing A.14: Add StreamListener

There is also another stream channel, namely the RetransmitStreamChannel which supports retransmission requests from the receivers. It also allows to adjust the bandwidth according to complains sent from the receivers.

B Test Plugin

The following chapter describes some technical details about the Test Plugin. It contains a list of commands provided by the plugin and a list of variables to be used in these commands.

B.1 Commands

The following section presents a list of command that can be executed in the OSGi console, if the Test Plugin is loaded.

printSlaveList This command is used to print the IP address and port for each registered slave.

clearSlaveList This command is used to clear the master's slave list. It deletes all IP address and port entries.

addSlave <IP-address:port> / removeSlave <IP-address:port> These commands are used to add (or remove) a certain slaves IP address and port to (from) the master's slave list. The addSlave command should only be used, if for example the master crashed, but all slaves were already registered. Otherwise, the slaves are not listening on the specified IP address and port combination and therefore will not receive any command. Example call: removeSlave 192.168.0.3:10004

slaveDo <IP-address:port> <command> / slavesDo <command> These commands are used to delegate a command to a single slave (by committing its IP and port, or its index in the slave list) or all slaves.

loadTestPlan [pathTo/TestPlan.txt] This command can be combined with slaveDo (or slavesDo). It is used to load a so called *test plan*, a file with a collection of commands. These commands are executed sequentially by a slave. The test plan can be located on a web server or on the slaves machine. The path can be specified in the optional parameter. If no parameter is specified, the default behavior is to load the testplan.txt from the resources folder. Example call: slavesDo loadTestPlan http://www.p2pframework.com/plans/testplan.txt

doXTimes <amount> <sleep> <command> With this command, other commands like the social-network commands can be executed repeatedly for several times. The parameter *amount* specifies the number of executions, the parameter *sleep* declares the waiting time (in ms) between these consecutive executed commands. After the execution of each consecutive command, the slaves waits a random number between 0 and *sleep* before it executes the next consecutive command. A general use case is to simulate a real user, writing several wall posts or group messages. To make this command more powerful, the test plugin is able to generate and choose from some random content, like messages, user names, and even pictures. Details about content generation can be found in Section 4.8.3. Example call:

slavesDo doXTimes 10 5 WallPlugin.AddComment <userID> <postID> <text>

slavesLoginIncremental [bootstrapIP] [loginNameIndex] This is usually the first command sent to the slaves. The master sends each slave the command to connect to the P2P ring and login to the social network. Depending on the amount of slaves and the content of the loginNames.txt in the resources folder, each slave is assigned a user name from the loginNames.txt and the general password *"password"*. The login names are not chosen randomly but incremental, so if there are for example 10 slaves, this command will choose the first 10 login names, when the optional parameter *loginNameIndex* is 0. The optional parameter bootstrapIP determines the IP address of the bootstrap server and the optional parameter loginNameIndex determines the line to start choosing login names. The default values are `pgframework.cs.upb.de` and 0.

slavesLogout [interval] This command sends the logout command to all slaves. The optional parameter *interval* specifies the time before the logout command is send to the next slave. This is useful to slowly shutdown the network and to provide the remaining nodes some time to replicate the data. The default value is $30s$ ($30000ms$).

sendFriendRequests <amount> <friendNameIndex> With this command, the slave sends *amount* friend requests. The usernames for the friendship requests are chosen sequentially from the loginNames.txt file. This command works best in combination with the *slavesLoginIncremental* command. Assume there is a master with 10 slaves. The master first executes the *slavesLoginIncremental* command, which chooses the first 10 names from the loginNames.txt for the slaves. After this, the master can send the *slavesDo sendFriendRequests 10 0* command, which results that each slave sends a friendship request to all the other slaves' users in the P2P network.

B.2 Variables

The following tables describe all variables that can be used in the commands of the Test Plugin. After each table, sample use cases describe the usage of the variables.

$SERVER	Returns the IP address of the server predefined in *config.txt*.
$OWNIP	Returns the current IP address of the machine.
$BINDPORT	Rinds a free port on this machine and returns it.
$LOGINNAME	Chooses randomly one of the predefined names located in the file *loginNames.txt*.
$PASSWORD	Returns the predefined password *password*.
$MAIL	Should be used in combination with $LOGINNAME. Adds *@mail.de* to the previously selected login name.
$COUNTRY	Chooses randomly one of the predefined countries located in the file *countryNames.txt*.
$CITY	Chooses randomly one of the predefined cities located in the file *cityNames.txt*.
$RANDOM_INT	Returns a random integer between 0 and the predefined variable *maxSleepTime*. Is used to randomize the execution time of tasks, for example in the *doXTimes* command.

Table B.1: Variables used for connection and registration.

With the variables described in Table B.1, it is possible to connect to a specified bootstrap server and register or login with a randomly chosen user, with randomly chosen values for city, country and e-mail address. The corresponding command is the following:

```
doXTimes 1 $RANDOM_INT ProfilePlugin.Register $SERVER 9001
$OWNIP $BINDPORT $LOGINNAME $PASSWORD $MAIL $COUNTRY $CITY
```

After the command is interpreted by a slave, it could look like this:

```
doXTimes 1 1482 ProfilePlugin.Register pgframework.cs.upb.de 9001
131.234.61.16 9001 alex password alex@mail.com germany berlin
```

Since the user name is chosen randomly, which makes controlled testing more difficult, it is recommended to use the *slavesLoginIncremental* command describe in B.1.

$MESSAGE	Chooses randomly a message of the predefined messages located in the file *messages.txt*.
$GROUPNAME	Chooses randomly one of the predefined names located in the file *groupNames.txt*.
$FRIENDSHIP_CANDIDATE	Chooses randomly one name from the *loginNames.txt*, but ensures that there has never been sent a friendship request to this username from the actual logged in user.
$USERS_GROUP	Chooses randomly a current group of the actual registered user.
$USERS_FRIEND	Chooses randomly a friend of the actual registered user and returns it's user name.
$USERS_FRIEND_ID	Chooses randomly a friend of the actual registered user and returns it's user ID.
$USERS_FRIEND_ID_ AND_POST_ID	Chooses randomly a friend of the actual registered user and a post on it's wall. The variable returns the friend's user ID and the ID of the post.

Table B.2: User dependent content variables

The following sample commands take advantage of the variables, presented in Table B.2.

```
slavesDo doXTimes 10 $RANDOM_INT MessagingPlugin.SendMsg
                     $USERS_FRIEND $MESSAGE
slavesDo doXtimes 10 $RANDOM_INT GroupsPlugin.CreateGroup
                     $GROUPNAME
slavesDo doXtimes 10 $RANDOM_INT GroupsPlugin.SendGroupMsg
                     $USERS_GROUP $MESSAGE
slavesDo doXtimes 10 $RANDOM_INT FriendsPlugin.RequestFriend
                     $FRIENDSHIP_CANDIDATE $MESSAGE
slavesDo doXtimes 10 $RANDOM_INT WallPlugin.AddPost
                     $USERS_FRIEND_ID $MESSAGE
slavesDo doXtimes 10 $RANDOM_INT WallPlugin.AddComment
                     $USERS_FRIEND_ID_AND_POST_ID $MESSAGE
```

$ALBUMNAME	Chooses randomly one of the predefined names located in the file *groupNames.txt* (since group names may suit as album names as well).
$ALBUMKEY	Chooses randomly one of the user's albums and returns it's key.
$PICTURE	Chooses randomly one of the photo files located in the resources/pictures folder.

Table B.3: PhotosPlugin variables

The following sample commands take advantage of the variables, presented in Table B.3.

```
slavesDo doXTimes 10 $RANDOM_INT PhotosPlugin.CreateNewAlbum
                     $ALBUMNAME#$MESSAGE#users
slavesDo doXtimes 10 $RANDOM_INT PhotosPlugin.AddPhoto
                     $ALBUMKEY#$GROUPNAME#$PICTURE#$MESSAGE
```

B.3 Using the Test Plugin

The following guide describes the setup and execution of a simple test using the Test Plugin.

B.3.1 Prerequisites

The Test Plugin can be started in the Eclipse IDE as well as in a stand alone binary build. This How To supposes to use the Eclipse IDE on the master machine and stand alone binary builds on the slave machines. To use the Test Plugin, it should be verified that the Test Plugin component `com.sign.plugin.testplugin.component` is listed in the dependencies tab of the product configuration. The Test Plugin uses the files described in Table B.4.

config.txt	Contains a flag if this client is master or slave and the IP address of the master
loginNames.txt	Contains a list of possible login names
messages.txt	Contains a list of possible messages
cityNames.txt	Contains a list of possible city names
countryNames.txt	Contains a list of possible country names
groupNames.txt	Contains a list of possible group names
pictures	Folder which contains possible pictures (.jpg)

Table B.4: Files of the Test Plugin

These files must be included in the `resources` folder. The folder must be accessible by the application and therefore be located in the Eclipse installation folder (e.g on Mac OS: `/Applications/Eclipse/Eclipse.app/Contents/MacOS/resources`) — when using the Eclipse IDE for the Test Plugin — or in the `eclipse` folder of the stand alone binary build. If these prerequisites are ensured, the Test Plugin is ready for setup.

B.3.2 Setting up the Master

To setup the master node, the *master=true* flag needs to be set in the `config.txt` file. Once this is ensured, the master node can be started by executing the application in the Eclipse IDE. If the master node is not started in the Eclipse IDE but as a stand alone application, the application should be executed in a terminal to be able to send commands over the console.

B.3.3 Setting up the Slaves

To setup the slave nodes, the *master=false* flag needs to be set in the `config.txt` file and the *server=XXX.XXX.XXX.XXX* line should lists the correct IP address of the master node. Once this is ensured and the master node is running, the slave nodes can be started by executing the stand alone binary build.

The master node prints debug information if a slave has successfully registered. The command *printSlaveList* can be used by the master node to get information about the registered slaves.

After the connection between the master node and the slave nodes is established, the Test Plugin is ready for use.

B.3.4 Executing the test

The executed actions in this sample test are as follows:

- All slaves register to the network and login
- All slaves send friend request to the users of the other slaves
- All slaves change and store their user's profile picture
- All slaves send 10 group messages
- All slaves create 3 photo albums and add 15 photos
- All slaves write 25 wall entries on friends walls
- All slaves comment on 25 wall entries at friends walls
- All slaves logout

To start, the master node sends the *slavesLoginIncremental [bootstrapIP] [loginIndex]* command to the slaves. This can be done by entering *slavesLoginIncremental* `pgframework.cs.upb.de` in the console. With this command, the first slave in the master's slave list registers to the respective bootstrap server with the first login name in the `loginNames.txt`, the second slave registers with the second login name and so on.

The next step is to establish friendship relations between the slaves' users. Therefore, the master node sends the *sendFriendRequests <amount> <friendNameIndex>* command to all slaves by typing *slavesDo sendFriendRequests 10 0* in the console, assuming there are 10 registered users (slaves) in the network.

Since these requests must be accepted by the other slaves, there are several methods to do this. One method is to send the *acceptAllFriendRequests* to all slaves by typing *slavesDo acceptAllFriendRequests* or by accepting these friend requests manually at the slave machines.

The next actions can be executed independently by the slaves, since there is only random content generated and no central control instance needed. Therefore, the Test Plugin introduces *Test Plans*. A *Test Plan* is a file, which contains several lines of social network commands and will be executed sequentially by a slave. An example Test Plan that represents the remaining actions of this test looks as follows:

```
monitor start
doXTimes 1 $RANDOM_INT ProfilePlugin.EditProfileField
           ProfileImage $PICTURE
doXTimes 1 $RANDOM_INT ProfilePlugin.StoreMyProfile
doXTimes 10 $RANDOM_INT GroupsPlugin.SendGroupMsg
            $USERS_GROUP $MESSAGE
doXTimes 3 $RANDOM_INT PhotosPlugin.CreateNewAlbum
           $ALBUMNAME#$MESSAGE#users
doXTimes 1 15000 PhotosPlugin.LoadMyAlbums
doXTimes 15 $RANDOM_INT PhotosPlugin.AddPhoto
            $ALBUMKEY#$GROUPNAME#$PICTURE#$MESSAGE
doXTimes 25 $RANDOM_INT WallPlugin.AddPost
            $USERS_FRIEND_ID $MESSAGE
doXTimes 25 $RANDOM_INT WallPlugin.AddComment
            $USERS_FRIEND_ID_AND_POST_ID $MESSAGE
monitor stop
doXTimes 1 1 LoginPlugin.Logout
```

To let the slaves execute this Test Plan independently but concurrently to the other slaves, the master has to send the *loadTestPlan [pathTo/TestPlan.txt]* command to all slaves. The Test Plan file can be located on a web server or on the slaves machines.

After all slaves finished executing the actions, they can logout from the network. This can be achieved by typing *slavesLogout* in the console.

C Test Plots

The plots in this chapter are based on monitored data, that was captured during the pool area test. Unfortunately, we only have the plots of one test, where the focus was on testing the overall function of the network. It would be more interesting, to analyze and compare the monitored data of different tests. As example, there could be an analyzation of the performance of a large network, compared to the performance of a small network. For this comparison a well evaluated test setup and test plan is needed, that defines a reasonable amount of peers and actions. Another interesting approach would be to monitor the data over a longer period of time, with different users logging in and out.

Without this focus, our monitored data is not as meaningful as expected and therefore not analyzed in detail. Nevertheless, the plots are appended, to give an overview of the possibilities of the monitoring plugin.

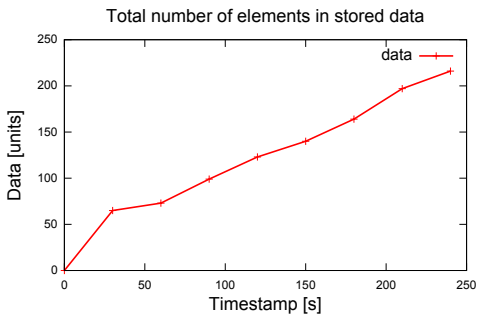


Figure C.1: Total number of elements in stored data

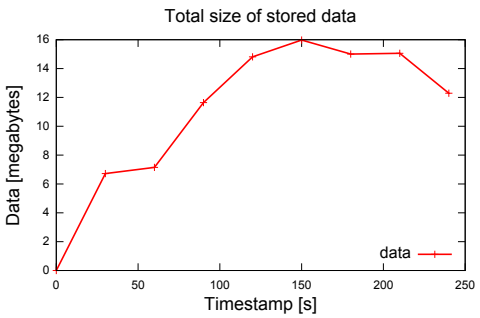


Figure C.2: Total size of stored data

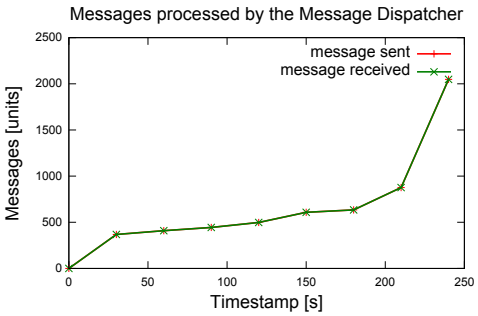


Figure C.3: Messages processed by the Message Dispatcher

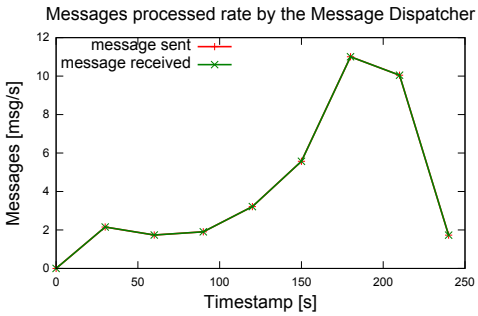


Figure C.4: Messages processed rate by the Message Dispatcher

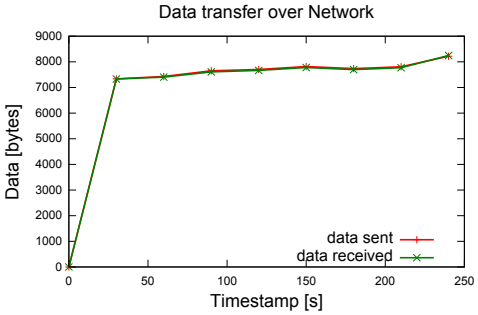


Figure C.5: Data transfer over Network

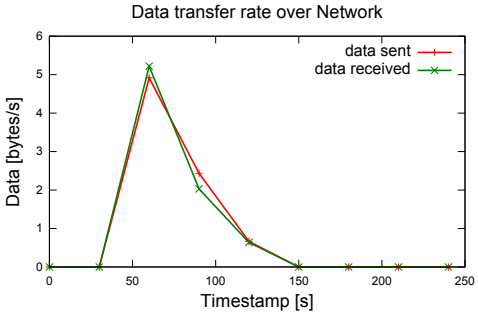


Figure C.6: Data transfer rate over Network

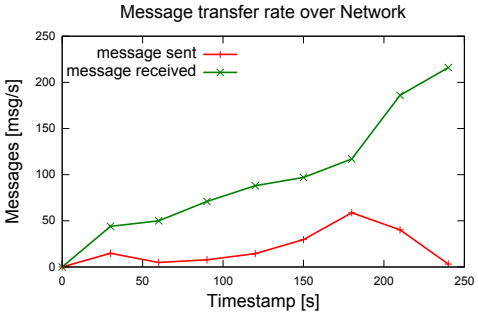


Figure C.7: Message Transfer rate over Network

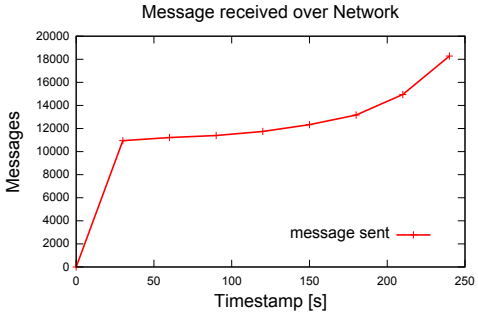


Figure C.8: Message received over Network

D Remote Operation Protocols

The following protocols define commands that can be used to issue work at the backend. This gets used in Section 4.5, 4.6, 4.7.

D.1 Remote Operation Interface

The remote operation call can be done in two manners. The first one issues a function on a single receiver and the second one can issue the command on many nodes in parallel. Therefore the function supports different aggregation strategies to combine the results sent from these issued nodes. Some of these aggregation strategies are just for these situations and get listed in Appendix E.2. These functions return the value produced by the aggregation chosen.

public *result* **issueOperation**(
DestinationHash,
RemoteOperation,
Parameter1, *Parameter2*,
Group Hash,
StartValue, *AggregationStrategy*
)

public *result* **issueParallelOperation**(
Collection<DestinationHashes>,
RemoteOperation,
Parameter1, *Parameter2*,
Group Hash,
StartValue, *AggregationStrategy*
)

Group Hash Parameter:

- Used in a *generate* operation causes the receiver to set the group that manages the access control. The group hash is the address where the group item is stored.
- Used in the same situation and set to *null* causes the data structure not be access controlled.
- if it is an *access* operation, the group hash parameter gets also checked for correctness on the remote end which is necessary to enforce the access control.

D.2 Distributed Set

Generate

< Parameter 1 > :	*null*
< Parameter 2 > :	*null*
< Return Value > :	(`String`) *"done"*
Alternative Error :	(`String`) *"error"*

Signals the receiver to generate a new `DistributedBucket`. If a group hash was given the item is access controlled. To further secure the structure it is good to let the receiver ask a node with a bucket before itself what the right group hash is.

AddAll

< Parameter 1 > :	(`Collection`) Collection of items to add.
< Parameter 2 > :	(`Integer`) SplitSize which is the size of this bucket.
< Return Value > :	(`Integer`) Number of written items.
Alternative Error :	`UserNotMemberOfGroupException`

Tries to store as many items as possible. The return value indicates how many items got stored and how many need to be stored in a later bucket.

Delete

< Parameter 1 > :	*null*
< Parameter 2 > :	*null*
< Return Value > :	(`String`) *"done"*
Alternative Error :	(`String`) *"error"*

This operation removes the whole bucket. It is the counter part to `Generate`.

Contains

< Parameter 1 > :	(`Serializable`) Item in question which is encrypted, if the data structure is access controlled.
< Parameter 2 > :	*null*
< Return Value > :	(`Boolean`) Answer of the query for containment.
Useful Aggregation:	`Bool` agg. strategy to combine the answers of all buckets.

This command lets the receiver go through its bucket and compares the items. In case of an access controlled data structure it compares the byte arrays. If they are encrypted with different key versions, this is a problem. Though important to note, if the data structure is access controlled all data items provided in parameters should be encrypted. The receiver should not know the content of the items.

ContainsAll

< Parameter 1 > :	(`Set`) Set of all items in question. These are encrypted if the data structure is access controlled.
< Parameter 2 > :	*null*
< Return Value > :	(`Set`) Set of all items that are not in the bucket.
Useful Aggregation:	`CollectionRetainAll` agg. strategy initialized with the first parameter given above.

Same procedure as `Contains` above. But the return value is a set of all items not in the bucket. Retaining the returned set from the overall set is the aggregation strategy. If the overall resulting collection is empty this means all initial items where found in at least one bucket.

RemoveAll

< Parameter 1 > :	(`Collection`) Items to remove. These are encrypted if the data structure is access controlled.
< Parameter 2 > :	*null*
< Return Value > :	(`Integer`) Number of removed items.
Alternative Error :	`UserNotMemberOfGroupException`
Useful Aggregation:	`Sum` aggregation strategy to get the overall removed items in parallel execution.

The receiver removes all the given items from its bucket. The return value indicates how many items got removed.

RetainAll

< Parameter 1 > :	(`Collection`) Items to retain. These are encrypted if the data structure is access controlled.
< Parameter 2 > :	*null*
< Return Value > :	(`Integer`) Number of removed items.
Alternative Error :	`UserNotMemberOfGroupException`
Useful Aggregation:	`Sum` aggregation strategy to get the overall removed items in parallel execution.

The receiver removes all items not given in the first parameter from its bucket. The return value indicates how many items got removed.

RemoveObject

< Parameter 1 > :	(`Serializable`) Item to remove. This item is encrypted, if the data structure is access controlled.
< Parameter 2 > :	*null*
< Return Value > :	(`Boolean`) If something got removed.
Alternative Error :	`UserNotMemberOfGroupException`

This operation allows to remove a single item.

GetAll

< Parameter 1 > :	*null*
< Parameter 2 > :	*null*
< Return Value > :	(`Collection`) All items of this bucket.

This command let the sender retrieve all the items of a bucket at once. This is useful for some maintenance of the whole structure.

D.3 Distributed Linked List

IndexOf

< Parameter 1 > :	(`Serializable`) Item in question. This item is encrypted, if the data structure is access controlled.
< Parameter 2 > :	*null*
< Return Value > :	(`Integer`) Global index of the first item.
Alternative Error :	*-1* if it is not in this bucket.
Useful Aggregation:	`SmallestNonNegative` agg. method to get the first index in parallel execution. Therefore the aggregation should be initialized with the error encoding number *-1*.

Retrieves the first index of the given item. These indexes should not be trusted if the data structure has to delete something in the same bucket. It is advised to better use an iterator provided by the Distributed Linked List.

LastIndexOf

< Parameter 1 > :	(`Serializable`) Item in question. This item is encrypted, if the data structure is access controlled.
< Parameter 2 > :	*null*
< Return Value > :	(`Integer`) Global index of the last item or
Alternative Error :	*-1* if it is not in this bucket.
Useful Aggregation:	`Greatest` aggregation method to get the last index in parallel execution. Therefore the aggregation should be initialized with the error encoding number *-1*.

Retrieves the last index of the given item. It is advised to better use an iterator provided by the Distributed Linked List.

Remove

< Parameter 1 > :	(`Integer`) Position of the item.
< Parameter 2 > :	*null*
< Return Value > :	(`Serializable`) The from the given position removed item.
Alternative Error :	`UserNotMemberOfGroupException`

Removes the item at the given index. To check if it is the right one the removed item is also returned. The iterator provided by the Distributed Linked List is better suited to remove.

Get

< Parameter 1 > :	(`Integer`) Position of the item.
< Parameter 2 > :	*null*
< Return Value > :	(`Serializable`) The item at the given position.
Alternative Error :	`UserNotMemberOfGroupException`

Returns an item without altering the data structure. This item is encrypted if the data structure is access controlled.

Set

< Parameter 1 > :	(`Integer`) Position of the item.
< Parameter 2 > :	(`Serializable`) The item to place at the given position.
< Return Value > :	(`Serializable`) The item previously at the position.
Alternative Error :	`UserNotMemberOfGroupException`

This operations exchanges one item at the given index with another from the second parameter. Also here the responsible node checks if the sender belongs to the group and has the rights to do so.

D.4 Prefix Hash Tree

NewPHT

< Parameter 1 > :	(`PrefixHashTree`) one PHT bucket to store.
< Parameter 2 > :	*null*
< Return Value > :	(`String`) *"done"*
Alternative Error :	(`String`) *"error"*

At the moment it is only used to store the PHT if there is only one bucket. A PHT with more buckets operates with the lookupAnd-Do operations.

Lookup

< Parameter 1 > :	(`String`) key of an item.
< Parameter 2 > :	*null*
< Return Value > :	(`String`) hash of the leaf bucket the item has to be in.
Alternative Error :	(`Integer`) *1* if the node is an internal PHT node.
Alternative Error :	(`Integer`) *-1* if the node does not belong to the PHT.
Useful Aggregation:	`nonInteger` agg. method to get the position of the leaf bucket in parallel. In non parallel mode the alternative errors help to do a binary search.

Main operation to find out on which key prefix the leaf node resides. Depending on the optimization setting this happens in parallel or binary search over all possible prefixes of the given key.

LookupAndAdd

< Parameter 1 > :	(`String`) key of the item.
< Parameter 2 > :	(`Serializable`) The item to store in the leaf bucket when the receiver is a leaf bucket.
< Return Value > :	(`String`) hash of the leaf bucket the item is in now.
Alternative Error :	(`Integer`) *1* if the node is an internal PHT node.
Alternative Error :	(`Integer`) *-1* if the node does not belong to the PHT.
Useful Aggregation:	`nonInteger` agg. method to get the position of the leaf bucket in parallel. In non parallel mode the alternative error help to do a binary search.

This operation combines the `Lookup` operation with the `Add` operation. Thus avoiding latency through merging of multiple operations like `Lookup` and item adding.

LookupAndGet

< Parameter 1 > :	(`String`) key of the wanted items.
< Parameter 2 > :	*null*
< Return Value > :	(`Collection`) collection of all items belonging to the key.
Alternative Error :	(`Integer`) *1* if the node is an internal PHT node.
Alternative Error :	(`Integer`) *-1* if the node does not belong to the PHT.
Useful Aggregation:	`nonInteger` agg. method to get the position of the leaf bucket in parallel. In non parallel mode the alternative error help to do a binary search.

This operation combines the `Lookup` operation with the `Get` operation. Thus avoiding latency through merging of multiple operations like `Lookup` and item retrieval. A collection is returned since there could be many items stored under one key.

LookupAndRemove

< Parameter 1 > :	(`String`) key of the item.
< Parameter 2 > :	(`Serializable`) The item to remove from the leaf bucket when the receiver is a leaf bucket.
< Return Value > :	(`String`) hash of the leaf bucket the item was in.
Alternative Error :	(`Integer`) *1* if the node is an internal PHT node.
Alternative Error :	(`Integer`) *-1* if the node does not belong to the PHT.
Useful Aggregation:	`nonInteger` agg. method to get the position of the leaf bucket in parallel. In non parallel mode the alternative error help to do a binary search.

This operation combines the `Lookup` operation with the `Remove` operation. Thus avoiding latency through merging of multiple operations like `Lookup` and item removal.

Clear

< Parameter 1 > :	*null*
< Parameter 2 > :	*null*
< Return Value > :	*null*, but only leaf nodes send an answer back. Internal nodes relay the message to their children.
Useful Aggregation:	`multiMapAddAbortIfLeafsComplete` aggregation strategy initialized with the keys alphabet size. It waits until all leafs has answered. For details see Appendix E.

If the receiver is an internal node of the PHT, it relays the message to all its children. If it is a leaf node it answers to the original sender and returns *null*. In both cases the PHT bucket gets deleted.

RangeSerial

< Parameter 1 > :	(`String`) low bound of the range.
< Parameter 2 > :	(`String`) high bound of the range.
< Return Value > :	(`Multimap<String, E>`) multimap with all key-value pairs fitting in the key-range.
Useful Aggregation:	`multiMapAddAbortIfNull` agg. strategy initialized with an empty `Multimap`.

The receiver should be the leaf node representing the low bound. It will answer and relay the operation message along the leaf nodes until the node representing the high bound is reached. Every node in between will also answer with its key-value pairs and its response number which gets increased by one every time a leaf node gets traversed. The aggregation collects all answers.

RangeParallel

< Parameter 1 > :	(`String`) low bound of the range.
< Parameter 2 > :	(`String`) high bound of the range.
< Return Value > :	(`Multimap<String, E>`) multimap with all key-value pairs fitting in the key-range.
Useful Aggregation:	`multiMapAddAbortIfLeafsComplete` aggregation strategy initialized with the keys alphabet size. It waits until all leafs has answered. For details see Appendix E.

The receiver should be parents to both nodes that represent the low and high border. This node spreads the operation message to all its children and they continue to do so. Leaf nodes answer the query directly to the original sender which is aggregating the answers and continues if all leafs have answered. This algorithm has a drawback if a range like $[01111, 10000]$ is requested, because the common prefix would be the root node. This can be solved if the range is split accordingly and two requests are issued.

RangeSerialIterate

< Parameter 1 > :	(`String`) low bound of the range.
< Parameter 2 > :	(`String`) high bound of the range.
< Return Value > :	(`Multimap<String, E>`) multimap with all key-values of that bucket. It also sets a variable `continuationHash` to the hash of the next leaf node.

Can be send to any leaf node which answers with content that fits in the range and returns also the hash of the next leaf to ask. This hash gets stored in a public variable of the `RemoteOperationsClient`.

NewNext

< Parameter 1 > :	(`String`) prefix of the key the next-link should be set to.
< Parameter 2 > :	*null*
< Return Value > :	(`String`) hash of the leaf node where the next-link was changed.
Alternative Error :	(`Integer`) *1* if the node is an internal PHT node.
Alternative Error :	(`Integer`) *-1* if the node does not belong to the PHT.

This gets used in order to maintain the next-link structure between the leaf nodes while splitting and merging nodes. The receiver changes its next-link to the given one.

NewPrev

< Parameter 1 > :	(`String`) prefix of the key the prev-link should be set to.
< Parameter 2 > :	*null*
< Return Value > :	(`String`) hash of the leaf node where the previous-link was changed.
Alternative Error :	(`Integer`) *1* if the node is an internal PHT node.
Alternative Error :	(`Integer`) *-1* if the node does not belong to the PHT.

This gets used in order to maintain the previous-link structure between the leaf nodes while splitting and merging nodes. The receiver changes its prev-link to the given one.

MergeWithParents

< Parameter 1 > :	*null*
< Parameter 2 > :	*null*
< Return Value > :	*null*

Used to signal the own siblings to merge with the parents. It is important to do this only if all siblings are also leaf nodes. The contacted sibling transfers all its data to its parent calling a `MergeWithYou` operation. The lowest sibling issues a `newNext` and the highest a new `newPrev` operations to their prev-link and next-link respectively. All siblings are deleted after this is done.

MergeWithYou

< Parameter 1 > :	(`Collection`) items that the child wants to transfer.
< Parameter 2 > :	(`String`) key prefix of a neighbor the parent should use. Or *null* if the sender is nether lowest or highest sibling.
< Return Value > :	*null*

Should only sent to an inner node which becomes immediately a leaf node. Depending on the size of the neighbor key prefix it will be set as the new next or previous link.

D.5 Group Managment

CreateGroupItem

< Parameter 1 > :	(`Group`) group item that should be created.
< Parameter 2 > :	*null*
< Return Value > :	(`String`) *"Done"*
Alternative Error :	*null*

The receiver creates the group item on its position. If one already exists the group item gets not overwritten. When overwriting is needed use the `SetGroupItem` command.

GetGroupItem

< Parameter 1 > :	(`UserID`) id of the admin that is asking.
< Parameter 2 > :	*null*
< Return Value > :	(`Group`) group item ready to modify.
Alternative Error :	*null*

This function can only be issued by an admin of the group. The receiver can check the admin rights because they are stored in the group item.

SetGroupItem

< Parameter 1 > :	(`UserID`) id of the admin that is asking.
< Parameter 2 > :	(`Group`) group item that should be stored.
< Return Value > :	(`String`) *"Done"*
Alternative Error :	*null*

This function can only be issued by an admin of the group. The receiver can check the admin rights because they are stored in the old group item that gets overwritten with this one. Between `GetGroupItem` and `SetGroupItem` the admin can change the rights configuration of that group.

GetPublicKey

< Parameter 1 > :	*null*
< Parameter 2 > :	(*Integer*) version of the key. Or *null* to get the current version.
< Return Value > :	(*PublicKey*) requested public key of the group.

This operation is allowed to anyone. Every group can create new public/private keys in order to revoke read rights to banned members. Hence the version parameter. The version numbering starts with zero.

GetEncryptedPrivateKey

< Parameter 1 > :	(`UserID`) id of the member of the group.
< Parameter 2 > :	(*Integer*) version of the key. Or *null* to get the current version.
< Return Value > :	(`KeyContainer`) encrypted private key in a container to transport multiple keys and their version at once.
Alternative Error :	`UserNotMemberOfGroupException`

The private group key is encrypted with the public key of the member such that he is the only one who can decrypt and read it. At the moment a hack gets used because the `PrivateKey` class can not be serialized. So we send here the encrypted seed to set the crypto system that the private key gets reproduced.

GetEncryptedSymmetricKey

< Parameter 1 > :	(`UserID`) id of the member of the group.
< Parameter 2 > :	(*Integer*) version of the key. Or *null* to get the current version.
< Return Value > :	(`KeyContainer`) encrypted symmetric key in a container to transport multiple keys and their version at once.
Alternative Error :	`UserNotMemberOfGroupException`

The symmetric group key is encrypted with the public key of the member such that he is the only one who can decrypt and read it.

GetAllKeys

< Parameter 1 > :	(`UserID`) id of the member of the group.
< Parameter 2 > :	(*Integer*) version of the keys. Or *null* to get the current versions.
< Return Value > :	(`KeyContainer`) encrypted symmetric and private key in a container to transport them with their versions at once.
Alternative Error :	`UserNotMemberOfGroupException`

Used to minimize latency imposed through multiple requests for symmetric and private keys if both are needed which is the case when an access controlled data structure gets used the first time.

E Aggregation Strategies

The aggregation strategies are used to have a general aggregation framework for all kinds of data. The strategies are implemented in the `Aggregator` class. They get mainly used when aggregating attributes for monitoring and the global aggregation method described in Section 4.7. The later use in general aggregation is also accessible by framework users through an `AggregationChannel`.

One other very important use of aggregation strategies is to aggregate results from remote operations that are issued in parallel. These operations are listed in Appendix D and used through out the framework.

E.1 Mainly Used for Aggregation of Attributes

Smallest This strategy takes two values and aggregates them to one, namely the smallest. Thus the values provided have to have the same data type. Different data types do not get aggregated.

$$\text{aggregate}(a, b) = min(a, b)$$

Greatest For this strategy the same procedure happens as above. But the greater of both numbers is taken as the result.

$$\text{aggregate}(a, b) = max(a, b)$$

Sum This kind of aggregation allows to sum up both parts and the result is just that.

$$\text{aggregate}(a, b) = a + b$$

Mean Since whole attributes are aggregated the aggregation strategy can access data stored in every `Attribute` instance. So for example these class stores the number of single attributes that got combined in the attributes. This value of combined attributes follows the aforementioned Sum strategy. So the Mean strategy simply adds both values normalized with their number of already aggregated attributes and divides with the number of overall attributes.

$$\text{aggregate}(a, b) = \frac{a * |a| + b * |b|}{|a| + |b|}$$

$$\text{whereby: } |x| = \text{number of already participating attributes of } x$$

$$\text{and: } |\text{aggregate}(a, b)| = |a| + |b|$$

E.2 Mainly Used for Aggregation of Remote Operation Results

For remote operation aggregation there is always a start value which gets set up before aggregation values come in. This start value gets set as the result and all incoming values which have to be aggregated are combined with the old result to form the new one. The aggregation strategy also defines the number of values that are needed until the aggregation is complete. So some aggregation methods are just counting down until they got as many answers as they have sent and some use sophisticated calculations to determine if they are finished.

Assign This is the default aggregation method. Here the new incoming value gets just assigned to the result and the old value is lost.

$$\text{result} := \text{aggregate}(\text{result}, a) = a;$$

Bool This method is based on logical operations and depend on the start value. If the start value is *true* it operates with logical AND and if it initialized with *false* it operates with OR.

$$\text{result} := \text{aggregate}(\text{result}, a) = \begin{cases} \text{result} \wedge a, & \text{if start value} = true \\ \text{result} \vee a, & \text{if start value} = false \end{cases}$$

TrueToHash This strategy is useful to return the hash of the answer that returned *true* from the remote operation. It allows to reuse the same remote operation `contains(x)` not only for contains(x) but also for whereIs(x). contains(x) uses the aggregation `Bool` while whereIs(x) uses `TrueToHash`.

$$\text{result} := \text{aggregate}(\text{result}, a) = \begin{cases} \text{hash/address of the sender of } a, & \text{if } a = true \\ \text{result}, & \text{else} \end{cases}$$

NonInteger This is nearly the same as above. Here the value gets only stored if it is not from type `Integer`. The `Lookup` operation of a Prefix Hash Tree returns *1* representing an inner tree node, *-1* for a node not in the tree and a hash for the leaf node (Described in Appendix D). If all these nodes gets asked in parallel the only thing to look for is the hash `String`. Hence the `NonInteger` aggregation.

$$\text{result} := \text{aggregate}(\text{result}, a) = \begin{cases} a, & \text{if instanceOf } a \text{ is not Integer} \\ \text{result}, & \text{else} \end{cases}$$

SmallestNonNegative This is the same aggregation as `Smallest` with one additional rule. The value must be greater or equal to zero in order to be considered. This is used for the `indexOf` operation in a Distributed Linked List.

There, the start value gets initialized with *-1*, all positive values get aggregated and if no positive answer are returning the result is the correct error code.

$$\text{result} := \text{aggregate}(\text{result}, a) = \begin{cases} Min(\text{result}, a), & \text{if } a \geq 0 \\ \text{result}, & \text{else} \end{cases}$$

CollectionRetainAll Here the start value and all incoming answers from remote operations should be collections. This is used for the containsAll(x) operation in a Distributed Set. The aggregation function calculates the intersection of both collections.

$$\text{result} := \text{aggregate}(\text{result}, A) = \text{result} \cap A$$

MultiMapAddAbortIfNull This is used for the Prefix Hash Tree `RangeSerial` operation. The answer is a `Multimap` which gets added to the result. Every answering node sends also its response number which represents the order they were traversed. The last answering leaf node sends also a `DoneMessage` with *null* to signal the end of the range. The aggregator finishes if it got the *null* and all DoneMessages with smaller response numbers has arrived.

$$\text{result} := \text{aggregate}(\text{result}, A) = \begin{cases} \text{result} \cup A, & \text{if } A \neq null \\ \text{result}, & \text{else} \end{cases}$$

MultiMapAddAbortIfLeafsComplete This aggregation strategy very useful. It can be used to aggregate parallel queries to internal nodes where the receiver does not answer. The children leaf nodes do answer but the sender does not know how many answers are coming. It aggregates like a `MultiMapAddAbortIfNull` above but has a different waiting criteria. The aggregation method saves all the prefixes of incoming answers. But since all answers come from children under the internal node the prefixes cut the first characters that represent the internal node. Now, if there are as many prefixes with only the last symbol differing as characters in the PHT alphabet then all these prefixes can be shortened one symbol and represented through one of them. This means that all children of one node have answered. This procedure gets applied until the fixpoint is reached. Thus, if all leafs under one internal node are represented in the saved prefixes they would be enough to be reduced to the empty prefix. Therefore signaling that all leafs under the internal node have answered without knowing the tree landscape at start.

Bibliography

[1] Pastry - A scalable, decentralized, self-organizing and fault-tolerant substrate for peer-to-peer applications, 2009 (accessed March 31, 2012). `http://www.freepastry.org`.

[2] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. NIST Special Publication 80057: Recommendation for Key Management – Part 1: General. 2007.

[3] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Proceedings of the Internet Measurement Conference (IMC)*. ACM, 2009.

[4] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. On the Security of 1024-bit RSA and 160-bit Elliptic Curve Cryptography. *IACR Cryptology ePrint Archive*, 2009.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, R. A., and A. Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of the International Workshop on Peer-To-Peer Systems (IPTPS)*. USENIX, 2003.

[6] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20, 2002.

[7] J. R. Douceur. The Sybil Attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*. USENIX, 2002.

[8] P. Druschel and A. I. T. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. IEEE Computer Society, 2001.

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

[10] K. Graffi, A. Kovacevic, S. Xiao, and R. Steinmetz. Skyeye.kom: An information management over-overlay for getting the oracle view on structured p2p systems. In *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society Press, 2008.

[11] K. Graffi, S. Podrajanski, P. Mukherjee, A. Kovacevic, and R. Steinmetz. A Distributed Platform for Multimedia Communities. In *Proceedings of the International Symposium on Multimedia (ISM)*. IEEE Computer Society Press, 2008.

[12] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief announcement: prefix hash tree. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*. ACM, 2004.

[13] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the International Conference on Distributed Systems Platforms (DSP)*. Springer, 2001.

[14] A. I. T. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *In Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2001.

[15] E. Sit and R. Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*. USENIX, 2002.

[16] A. L. C. Tavares and M. T. Valente. A gentle introduction to OSGi. *ACM SIGSOFT Software Engineering Notes*, 33, 2008.