

Benchmarking of a Distributed Framework for Online Social Networks

Bachelor's Thesis

by

Liat Khitman

born in
Haifa, Israel

submitted to

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kálmán Graffi
Heinrich-Heine-University Düsseldorf

March 2020

Supervisor:
Newton Masinde, M. Sc.

Abstract

This thesis focuses on the evaluation and benchmarking of LibreSocial, a fully distributed and secure online social network. LibreSocial is built on top of FreePastry to enable secure routing between peers in the network. Furthermore, plugins could be developed and added to the system thanks to the utilization of the OSGi modular system development to increase LibreSocial's quality service as a social media application.

Prior to this thesis some work regarding the testing of LibreSocial was conducted, however little was done in terms of large scale testing to find out the functionality of the application in a large environment.

In order to benchmark LibreSocial we utilize the integrated testing plugin. To gather the results we use a distributed monitoring tool, SkyEye.KOM, to gather statistics about different metrics in the application. We design three unique test scenarios with the aim to test several quality aspects of LibreSocial, namely stability, scalability, performance and cost. The tests are deployed to the high performance cluster at the university that provides capability to support up to 4000 LibreSocial nodes.

We successfully show that LibreSocial is scalable, up to 2000 nodes, while remaining stable and during other test scenarios. We also show that LibreSocial maintains high performance under low as well as high workloads. Moreover, by varying the replication factor values to find an ideal value we show that LibreSocial performs well while maintaining low computational costs. We also indirectly show, as a result of some failed test attempts, that LibreSocial is relatively robust when internal failures occur.

Acknowledgments

The past three years have been a great adventure both academically and personally which came now to an end and I would like to take the chance to thank all the people who helped me get to this point. First of all I would like to thank Prof. Graffi, who gave me the opportunity to write my thesis in this field and work on developing LibreSocial further and Mr. Newton Masinde, who gave me support and guidance in the last months and always steered me in the right the direction whenever he thought I needed it. I owe thanks to a very special person, my partner, Alon for his continued and unfailing love, support and understanding during this time. You were always around at times I thought that it is impossible to continue, you helped me to keep things in perspective. I wish to acknowledge the support and great love of my family, my parents, Vadim and Olga; my grandparents, Bella and Shmil; and my brother, Sean. I wish to express my sincere appreciation to Irmgard, Trudy, Laura and Philipp for welcoming me to the family with open arms and giving me such warmth, love and affection and always rooting for me over the years. Last but not least, I am extremely grateful for my best friend Ilana, for her never-ending support and love.

Table of Content

List of Figures xi

List of Tables xv

1 Introduction 1
- 1.1 Motivation . . . 3
- 1.2 Contribution . . . 3
- 1.3 Structure of the Thesis . . . 4

2 Distributed Architectures 5
- 2.1 Definition . . . 5
- 2.2 Centralized Architecture . . . 7
- 2.3 Distributed Architectures . . . 8
  - 2.3.1 Partially Distributed . . . 8
  - 2.3.2 Fully Distributed Architecture . . . 10
- 2.4 Summary . . . 13

3 LibreSocial: A Peer-to-Peer Online Social Network 15
- 3.1 The Overlay Layer . . . 16
- 3.2 The Framework Layer . . . 17
  - 3.2.1 Storage Solutions . . . 17
  - 3.2.2 Security Features . . . 19
  - 3.2.3 Communication . . . 20
- 3.3 Plugins and Apps . . . 21
  - 3.3.1 Login Plugin . . . 21
  - 3.3.2 Wall Plugin . . . 22
  - 3.3.3 App-Store Plugin . . . 23
  - 3.3.4 Calendar Plugin . . . 23

3.3.5 Groups Plugin . . . 23
3.3.6 Forum Plugin . . . 24
3.3.7 File-Storage Plugin . . . 25
3.3.8 Friends Plugin . . . 25
3.3.9 Live-Chat Plugin . . . 26
3.3.10 Messaging Plugin . . . 27
3.3.11 Notifications Plugin . . . 27
3.3.12 Photos Plugin . . . 28
3.3.13 Commands Plugin . . . 28
3.3.14 Search Plugin . . . 29
3.3.15 Voting Plugin . . . 29
3.3.16 Monitoring Plugin . . . 30
3.3.17 Tester Plugin . . . 30
3.3.18 Web-Provider Plugin . . . 30
3.4 Summary . . . 32

4 LibreSocial Test Framework **33**
4.1 Monitoring . . . 33
4.1.1 SkyEye.KOM - Information Management Over-Overlay for Structured P2P Systems . . . 34
4.1.2 Metrics . . . 36
4.2 Testing . . . 37
4.2.1 Test Plugin . . . 37
4.2.2 Test Cases in LibreSocial . . . 39
4.3 Summary . . . 43

5 System Evaluation **45**
5.1 Quality Aspects and Relevant Metrics . . . 45
5.2 Pseudo-random Behavior . . . 47
5.2.1 Experimental Parameters and Scenarios . . . 48
5.2.2 Experimental Results . . . 49
5.2.3 Result Summary . . . 53
5.3 Stability Test . . . 53
5.3.1 Experimental Parameters and Scenarios . . . 54
5.3.2 Experimental Results . . . 54
5.3.3 Result Summary . . . 56

5.4 Replication factor 56
5.4.1 Experimental Parameters and Scenarios 57
5.4.2 Experimental Results 58
5.4.3 Result Summary 67
5.5 Summary 68

6 Conclusions **71**
6.1 Challenges 72
6.2 Future Work 72

Bibliography **75**

List of Figures

2.1 Types of distributed architectures 6

3.1 Overview of the LibreSocial structure 16
3.2 The LibreSocial graphical user interface 31

4.1 Metric aggregation strategy 34
4.2 Flowchart of testing and monitoring processes in LibreSocial 35

5.1 LibreSocial plugin transition probability [Kal13] 48
5.2 Total number of nodes in the network 50
5.3 Rate of sending messages in the network 50
5.4 Rate of sending messages for a single node 50
5.5 Data sent rate for a single node 50
5.6 Memory usage for a single node 50
5.7 Retrieved DDS for a single node 50
5.8 Number of nodes 52
5.9 Leafset size 52
5.10 Routing table size 52
5.11 Amount of sent messages in the network 52
5.12 Maximum rate of sent messages in the network 52
5.13 Amount of lost messages in the network 52
5.14 Total network memory usage 52
5.15 Data sent rate (mean) 52
5.16 Retrieved DDS rate (mean) 53
5.17 Number of nodes 55
5.18 Network messages 55
5.19 Message-plugin messages 55
5.20 Lost messages 55

5.21 Sent data 56
5.22 Sent data rate 56
5.23 Number of nodes for 10 nodes test 61
5.24 Number of nodes for 100 nodes test 61
5.25 Number of nodes for 1000 nodes test 61
5.26 Number of nodes for 200 nodes test 61
5.27 Number of nodes for 2000 nodes test 61
5.28 Hop count for 10 nodes 61
5.29 Hop count for 100 61
5.30 Hop count for 1000 61
5.31 Hop count for 200 62
5.32 Hop count for 2000 62
5.33 Total replicas in the network for 10 nodes 62
5.34 Total replicas in the network for 100 nodes 62
5.35 Total replicas in the network for 1000 nodes 62
5.36 Total replicas in the network for 200 nodes 62
5.37 Total replicas in the network for 2000 nodes 62
5.38 Maximum data storage time for 10 nodes 62
5.39 Maximum data storage time for 100 nodes 63
5.40 Maximum data storage time for 1000 nodes 63
5.41 Maximum data storage time for 200 nodes 63
5.42 Maximum data storage time for 2000 nodes 63
5.43 Maximum data retrieval time for 10 nodes 63
5.44 Maximum data retrieval time for 100 nodes 63
5.45 Maximum data retrieval time for 1000 nodes 64
5.46 Maximum data retrieval time for 200 nodes 64
5.47 Maximum data retrieval time for 2000 nodes 64
5.48 Message sending rate for 10 nodes 64
5.49 Message sending rate for 100 nodes 64
5.50 Message sending rate for 1000 nodes 64
5.51 Message sending rate for 200 nodes 64
5.52 Message sending rate for 2000 nodes 64
5.53 Message received rate for 10 nodes 65
5.54 Message received rate for 100 nodes 65
5.55 Message received rate for 1000 nodes 65

5.56 Message received rate for 200 nodes 65
5.57 Message received rate for 2000 nodes 65
5.58 Used space for 10 nodes 65
5.59 Used space for 100 nodes 65
5.60 Used space for 1000 nodes 65
5.61 Used space for 200 nodes 66
5.62 Used space for 2000 nodes 66
5.63 Memory usage for 10 nodes 66
5.64 Memory usage for 100 nodes 66
5.65 Memory usage for 1000 nodes 66
5.66 Memory usage for 200 nodes 66
5.67 Memory usage for 2000 nodes 66

List of Tables

3.1 OSGi commands for the Login Plugin 22
3.2 OSGi commands for the Wall Plugin 22
3.3 OSGi commands for the calendar Plugin 23
3.4 OSGi commands for the Group Plugin 24
3.5 OSGi commands for the Forum Plugin 24
3.6 OSGi commands for the Friends Plugin 26
3.7 OSGi commands for the Chat Plugin 26
3.8 OSGi commands for the Messaging Plugin 27
3.9 OSGi commands for the Photo Plugin 28
3.10 OSGi commands for the Search Plugin 29
3.11 OSGi commands for the Voting Plugin 29
3.12 OSGi commands for the Voting Plugin 30
3.13 OSGi commands for the Voting Plugin 31

4.1 Operational status of all test cases in LibreSocial 40
4.2 List of all available Preconditions in LibreSocial 42

5.1 List of used metrics for each quality aspect 47
5.2 System parameters 49
5.3 System parameters 54
5.4 Workload parameters 54
5.5 System parameters 57
5.6 Workload parameters 58
5.7 Results summary 67

Chapter 1

Introduction

Since 2009 the number of internet users has increased significantly from 1.7 billion users to 4.1 billion users in 2019[1]. Following this increase, the number of social media users between October 2018 and October 2019 increased by 328 million users[2]. The craving to check the latest tweet in the morning, watch videos on the commute to work and the urge to upvote the most trending photo caused social media to become an imperative part of the day-to-day lives of many people. We are also no longer bound to interact only with the people in our immediate living area, we can rather connect with 3.7 billion[3] other active social media users around the world.

The most common social networks such as Facebook[4] and Instagram[5] implement a centralized model as their underlying technology to handle the saved data. In this type of model the server is the sole entity that is responsible for data storage, manipulation, organization and accessibility, which means that the users could sometimes not be able to access their data when required or their files could be manipulated by the server's owner without the clients' permission. This structure is very popular because the information is stored in one place, on the server. This enables the owner easy access for maintenance work, central management, security and easy scalability. On the other hand, this rises challenges such as breaches in data privacy for the user and expensive scalability for the maintainer.

[1] https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/
[2] https://wearesocial.com/blog/2019/10/the-global-state-of-digital-in-october-2019
[3] https://www.statista.com/statistics/617136/digital-population-worldwide/
[4] https://www.facebook.com/
[5] https://www.instagram.com/

Contrary to the centralized model are the distributed technologies that could be broken down into two categories, namely fully distributed and hybrid. An example for a fully distributed model is Peer-to-Peer (P2P), in which the peers communicate directly with each other. In this type of network, no single entity holds the power to send instructions to other peers, but rather each peer makes decisions on its own. A hybrid technology holds aspects of centralization alongside distribution, in other words, the network consists of peers and a central server that helps peers communicate with each other and find information in the network. The distributed technology directly contrasts the common centralized model because every peer is responsible for saving data on its local storage, therefore data is not accumulated in one particular place. Furthermore data is usually replicated inside the network to ensure interminable access to it.

P2P technologies ignited people's interest in the 1990s around the world due to the sensational popularity of Napster, a platform that allowed its users to share audio files with each other [Bar17]. Short after, more and more P2P based application started emerging, one of them is Gnutella [AH00]. Unlike Napster, which had a centralized server with information about the whole network, Gnutella was the first fully decentralized P2P network for file sharing and was dominating its field with more than 40% of total P2P networks for a long time [Ban08]. More than two decades later, P2P technologies are used not only for file sharing, but they are also very widespread with large open software communities, such as Skype[6], an application specializes in video and voice calls, and many enterprise organization that offer services such as TV-channels for on-demand streaming [RD10]. The rush towards P2P applications can be further explained because of the advantages that this technology has to offer over the centralized model: anonymity, privacy and encryption, cooperation and resource sharing, data integrity, a high degree of decentralization and more flexibility against errors and attacks just to name a few.

In light of the challenges of existing centralized social media platforms and the strong advantages of the distributed architecture, LibreSocial [GM20] (previously called LifeSocial.KOM [Gra+11]) was developed in the Darmstadt University of Technology, in the University of Paderborn and in Heinrich Heine University in Düsseldorf as an alternative to popular social networks. It provides all the functionalities of a traditional social network like Facebook or VKontakte[7] and also takes advantage of a fully distributed P2P model that it is based on.

[6] https://www.skype.com
[7] https://vk.com/

The rising popularity of centralized and decentralized application alike has necessitated the execution of an extensive analysis for each system. The reasons for testing are to find any problems with the product before its release, to increase code stability, to find room for improvements, to increase efficiency etc. Benchmarking is a comprehensive and broad evaluation of a system which can be carried out by smaller and modular tests. The results of a benchmark reflect on the application's performance and capabilities. In this thesis we focus on the design of test models that will enable us to perform a large scale benchmarking on LibreSocial.

1.1 Motivation

The literature surrounding LibreSocial such as [Gra+10a] and [Gra+10b] was surveyed and it has been noticed that most of the testing that was done on LibreSocial give insightful information about the quality of the system, but not much has been done in terms of large scale testing to find its functionality in a large environment. In light of the challenges previously discussed and noted related to centralized and distributed social networks, we want to show that LibreSocial is a stable and scalable social network application with a good performance degree that has low computational costs. In order to show that, we intend to benchmark LibreSocial on a larger scale than ever before.

1.2 Contribution

Observing the challenges of social networks and realizing that one of the gaps in decentralized applications, in particular decentralized online social networks, is that no concrete data outside simulations exists, made the goal of this thesis to design test-models that will enable us to perform large scale benchmarking on LibreSocial. We expect to scale the system horizontally and we increase the churn and the workload. The performed tests will generate metrics, measurement scale that represents a specific attribute in the system under test, that in turn will help us to evaluate the quality aspects of the system and to determine its overall performance at a large scale.

1.3 Structure of the Thesis

The rest of the thesis is organized as follows

Chapter 2 presents an overview of centralized and distributed architectures and presents different architectural sub-models and their advantages and disadvantages.

Chapter 3 focuses on LibreSocial and the structure of its different layers. An overview of OSGi, modular system development standard, is given and the different plugins already integrated into LibreSocial are discussed in-depth.

Chapter 4 focuses on the testing framework of LibreSocial. We present SkyEye.KOM as a monitoring solution that is already integrated into LibreSocial and talk about different metrics that SkyEye can collect. Then the testing plugin is discussed in length and the structure of each test case is discussed in detail.

Chapter 5 first presents the quality aspects of distributed systems and then describes the designed test scenarios and discusses the obtained results.

In Chapter 6 a conclusion regarding the work that has been done on benchmarking of LibreSocial is given, some challenges that arose during the work on the thesis are presented and subjects for future work are suggested.

Chapter 2

Distributed Architectures

This chapter will give a brief overview of a distributed system architecture and its advantages over a traditional centralized design. Then some specific approaches of distributed systems such as peer-to-peer will be explained.

2.1 Definition

With the increasing demand in computational power, time spent online and an ever-growing amount of data generated worldwide, distributed systems become increasingly popular over traditional single-server designs.

A distributed system is defined by Tanenbaum as a "... *collection of autonomous computing elements that appear to its users as a single coherent system*" [VST17].

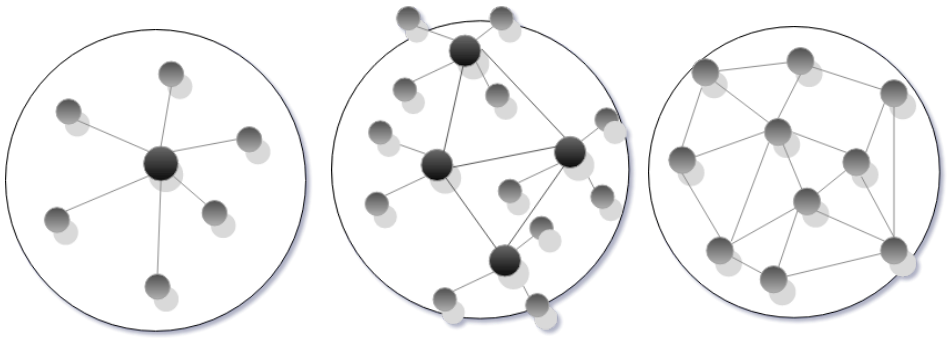
As the definition suggests, a distributed system is a collection of individual machines. Those machines could be computers, servers or virtual machines that may be located far from each other. The geographical location of the machines presents no issue because as long as they are all connected to the same network (local area connection or internet) communication can be established, messages can be exchanged and actions could be coordinated. This communication effort is important on the one hand because the machines work with each other to complete a common goal and on the other hand, the system should appear to the user as one singular machine. The complex nature of the underlying system is abstracted by the

software to enable a more pleasant and convenient user experience. Beside message exchange, individual machines in a distributed system also share computational power and access to local memory.

Even though there are many possible ways to implement such a system, a distributed architecture will usually have the following design objectives and characteristics [VST17] [BH03]:

- Collection of autonomous computing elements- no assumption about the type of the machine are made.
- All components run concurrently.
- All components fail independently of each other.
- Lack of a global clock due to the lack of a common interface.
- An event will take effect in one part of the system later than another part due to communication delays.
- The above-mentioned points may lead to an inconsistent view of the system.

Distributed systems are complex architectures and therefore the organizations of the underlying components are extremely important and could be categorized into *centralized architecture* and different *distributed architectures*. The next sections will give a brief overview of the different architectural models, will state the advantages and disadvantages of each one of them and give several examples of applications that use them.



(a) Centralized (b) Partially distributed (c) Fully distributed

Figure 2.1: Types of distributed architectures

2.2 Centralized Architecture

This architecture logic, also sometimes referred to as a *Client-Server Model* (see Figure 2.1a), is very important and remains the most widely spread among all other models. In this basic centralized model, the services are divided into two categories that show unique cooperation between two types of entities.

A *server* has a passive role of responding to clients' requests by providing resources and services to one or more clients. A *client* has an active role that initiates a connection to a server, which the server may, in turn, accept or reject. If the connection was accepted, the client can send requests for information to the server and wait for a reply. Moreover, only the server is responsible for the storage and manipulation of the saved data and only the server can grant access rights to a client. Therefore, a user might not have access to all the content or have access to modified content [VST17].

Examples for servers are web servers, file servers, mail servers etc. Depending on the type of server, the connection and communication are established by using a specific protocol (for example, a mail server uses a *SMTP protocol* that stands for "Simple Mail Transfer Protocol"). Most social networks such as Facebook and Instagram use a centralized model.

Even though the centralized model is the most widespread, it is not without flaws, the following section suggests some advantages and disadvantages of this architecture:

Advantages

- Scalability- the capacity of the server could be increased at any time with ease because the server is based at a specific location.
- Central management- easy management of a server that is placed in one location.
- Data integrity- access and control over the data is done by a single dedicated entity. The centralization also eases data backup.
- Fast data sharing and integration- services on a single server can share resources and data swiftly.

Disadvantages

- Overload potential- many concurrent client requests can lead to server overload, which in turn may lead to traffic congestion or server downtime.
- High expenses- high costs for hardware, monitoring, management, electricity etc.
- Single point of failure- due to the server being the single responsible entity for data storage and client communication, in the scenario that the server is offline, no client requests will be answered and the system will come to a halt.
- Data security- users do not own their data, but rather it is maintained and saved by the servers owner and could be potentially manipulated by them.

It is clear why the centralized model is so favorable. It gives the owner quick access to the server for bug fixes, general management is easy to carry out and the physical site can stay secure. On the other hand, the saved data is at risk of manipulation, it can be sold to a third party etc. Therefore other technologies exist to prevent data accumulation in one single place. Those architectures are discussed in the next section.

2.3 Distributed Architectures

Contrary to the centralized model, in distributed systems, many machines are connected in different ways (will be discussed in the following sections), the machines communicate through message exchange and operate together to achieve a common goal. The distributed architectural model can be split into two main categories, partially and fully distributed while a fully distributed system could be further split into different network models. In this section, those architectures and sub-models are discussed and for each of them short lists of pros and cons are presented.

2.3.1 Partially Distributed

In a *partially distributed* design (can be also called *decentralized*), the system is composed of many entities that work together in a common effort to accomplish a specific task. These

entities are called *nodes* and they are clustered together around higher lever nodes that are usually referred to as *super nodes* or *control nodes* (see Figure 2.1b). The communication is done through the super nodes that in turn communicate with other control nodes. Unlike the centralized model, higher-level nodes do not make decisions on behalf of other nodes, therefore the decision making lies with each peer that comprises the network, and the behavior of the system is aggregated from all nodes.

Augur[1], a prediction market platform, is an example for an application that uses a decentralized architecture[PK15]. Other use cases for a decentralized model are blockchain applications, private and anonymous networks such as Tor[2] and decentralized databases, in which the information is split and saved on many different, geographically distant nodes (each node acts as a small database) that are not directly linked together, for example BigchainDB[3].

The following are summarized lists of pros and cons of the partially distributed model:

Advantages

- More autonomy than in the centralized model due to the lack of a central server.
- Scalability- the load on the network is distributed, hence new nodes could be added to help with the load and traffic .
- Efficiency- high-level nodes allow more efficient delegation of tasks etc.
- Reliability- due to the setup of the network, it is more reliable and robust than a centralized architecture. It can handle high bandwidth throughput as well as hardware failure (for example,when a node leaves the network).

Disadvantages

- Security- if data is replicated onto multiple nodes to assure its availability, it is more prone to be accessed by a malicious entity.

[1] `https://www.augur.net/`
[2] `https://www.torproject.org/`
[3] `https://www.bigchaindb.com/`

- Dependence on super nodes- the network is not completely autonomous because of the dependency on the control nodes.
- Risk of lost messages- a single node is not directly connected to all other available nodes in the network, but rather a connection between two distant nodes can be established through a chain of nodes in between. A long chain can cause a loss of messages.
- Software- the underlying architecture involves many participating nodes and the design of such a system is complicated.

The partially distributed model frees the user from data security concerns that were risen in the discussion about the centralized architecture (section 2.2) and is much cheaper to maintain over time, but the system is not yet fully autonomous. This issue is solved by the fully distributed architecture that is discussed in the following section.

2.3.2 Fully Distributed Architecture

The most common fully distributed architecture and probably the only one is a *Peer-to-Peer* (P2P) model, in which many machines (peers) are linked together via the internet to divide the workload among all the peers and to share the computational power and storage space. Therefore, in contrast to the client-server model, each peer in the network takes on both the role of the client and the server, hence, all peers are equal. Because the lack of an existing server, there is no single entity that holds the power to send instructions to other machines, but rather, each node makes its own decision and the behavior of the system is decided by the aggregation of all individual decisions. Each individual in the system is equally responsible for contributing to the global mission. This model is very similar to the decentralized architecture but different because the peers are not clustered together and no super peers are present in a P2P system (see Figure 2.1c).

This type of system has gained more acknowledgment in recent years, mainly due to the popularity of Bitcoin and other cryptocurrencies[4]. The use case of a distributed systems is not limited to cryptocurrencies and the banking sphere, it is also applicable in other fields, for example, cluster computing, a set of computers that are coupled together and work in collaboration to achieve a common goal. Those systems possess large computational power

[4] https://www.statista.com/statistics/412255/europe-market-size-alternative-finance/

and the user sees this system as if it was a single computer.

The following are summarized lists of pros and cons of the fully distributed model:

Advantages

- In addition to the advantages that apply to the partially distributed model (subsection 2.3.1), the following are also true.
- Content distribution- generated data can be duplicated and distributed to other peers to make that content available in case the peer that is responsible for the content leaves the network.
- No single point of failure and high fault tolerance in comparison to a centralized system; each peer can leave on any occasion and the network will stay intact.
- Low latency- as a result of high geographical node spread, high traffic or network problems in one geographical location bears no consequences to nodes in other locations.
- High scalability- each new peer in the network brings along more computational power to the whole system.

Disadvantages

- Difficult deployment because a large number of components compose the network.
- Difficult to achieve unanimity because each peer makes decisions on its own without a central entity that sends instructions.
- Difficult to troubleshoot and administrate due to a large number of peers in the network and the geographical distribution.
- No central backup for the generated and saved content.
- Security risks- the security of the network is as strong as its weakest link. If one peer has compromised security, malicious parties can access the whole network through that compromised peer.

The P2P underlying structure evolved into three main types of architectures, namely structured, unstructured and hybrid.

Types of P2P architectures

P2P networks generally implement some kind of an *overlay network*, a virtual network that is built on top of another network, in this case, it is built on top of the physical network, the internet. Data is being sent through the physical layer, but communication between peers can be done directly along the overlay network [SW05].

This section presents three types of overlay networks and briefly describes each of them.

a. *Unstructured Network*: This type of overlay network imposes no specific order. It is formed when nodes join the network and randomly connect to other nodes. The lack of global structure makes the network very robust against high churn, when nodes frequently join and leave the network, and are easier to build. The fundamental drawback is that due to the absence of structure when looking for a specific piece of data the network could be over-flooded with queries. One example of an application that uses an unstructured network is Gnutella, that was described in chapter 1.

b. *Structured Network*: In contrast to the unstructured model, the structured overlay network is organized into a specific topology to ensure that nodes can efficiently search the entire network. The most common topology is a distributed hash table (DHT) that uses a hash function to expedite content lookup. Structured networks are more efficient than unstructured ones but they suggest more centralization as well, therefore they are also less robust. Pastry is an example for an algorithm that implements a DHT structured overlay network. LibreSocial uses the open source implementation of Pastry, called FreePastry[5].

c. *Hybrid Network*: A hybrid model is a combination between the distributed P2P architecture and the centralized client-server model. In this network, a central server saves information about the nodes and can also save information about the data that each node possesses, the server does not maintain the data itself, but rather it helps nodes find each other and the desired data items. Many models implement a hybrid architecture,

[5] `https://freepastry.org/`

each makes trade-offs between the advantages of centralization and node equality. The main advantage of this approach is attaining a significant degree of efficiency and decentralization at once. The known media services provider Spotify[6] was until 2014 an example for a hybrid model.

2.4 Summary

In this chapter, we looked at centralized and distributed systems and differentiated between partially and fully distributed architectures and showed that when we consider a centralized architecture it has the drawbacks such as security despite the profit and gain to the servers owner.

After giving this brief overview of the available architectures and presenting some of the advantages and drawbacks for each one of them, one question was not yet answered- which model is the best? There is no one definite answer. When weighing the possibilities for an underlying system design, each use case has a unique set of requirements, goals, business model, budget etc. therefore the pros and cons for each of the models should be precisely examined in order to determine which design suits better. Alongside these considerations, every solution should aim to alleviate the security and privacy concerns of users and support a dynamic and easy system growth [MG20].

In view of the P2P architecture providing strong reasons in favor of its implementation because of security and privacy concerns, the next chapter will present in detail one P2P based online social network.

[6] `https://www.spotify.com`

Chapter 3

LibreSocial: A Peer-to-Peer Online Social Network

LibreSocial [GM20] (also known as LifeSocial.KOM [Gra+11]) is an online social network based on a P2P technology. It is an ongoing project that was developed in Darmstadt University of Technology, in University of Paderborn and in Heinrich Heine University in Düsseldorf under the supervision of Jun.-Prof. Dr.-Ing. Kálmán Graffi.

LibreSocial provides all the basic functionalities of a traditional social network with the benefits that a fully distributed technology provides. The application consists primarily of three main components that are layered on top of each other. The layers, from bottom to top, are the overlay layer, the framework and plugins layer and the application layer. The forth topmost layer is the graphical user interface (GUI) layer and it will not be discussed in the scope of this thesis. A detailed structure of LibreSocial is shown in Figure 3.1.

This chapter will give a brief overview of the three main layers of LibreSocial and the components inside each one of them. All the available plugins and their functionality will be presented in detail. At the end of this chapter, the reader will have a strong sense of how LibreSocial is built and a general understanding about the different protocols that are used.

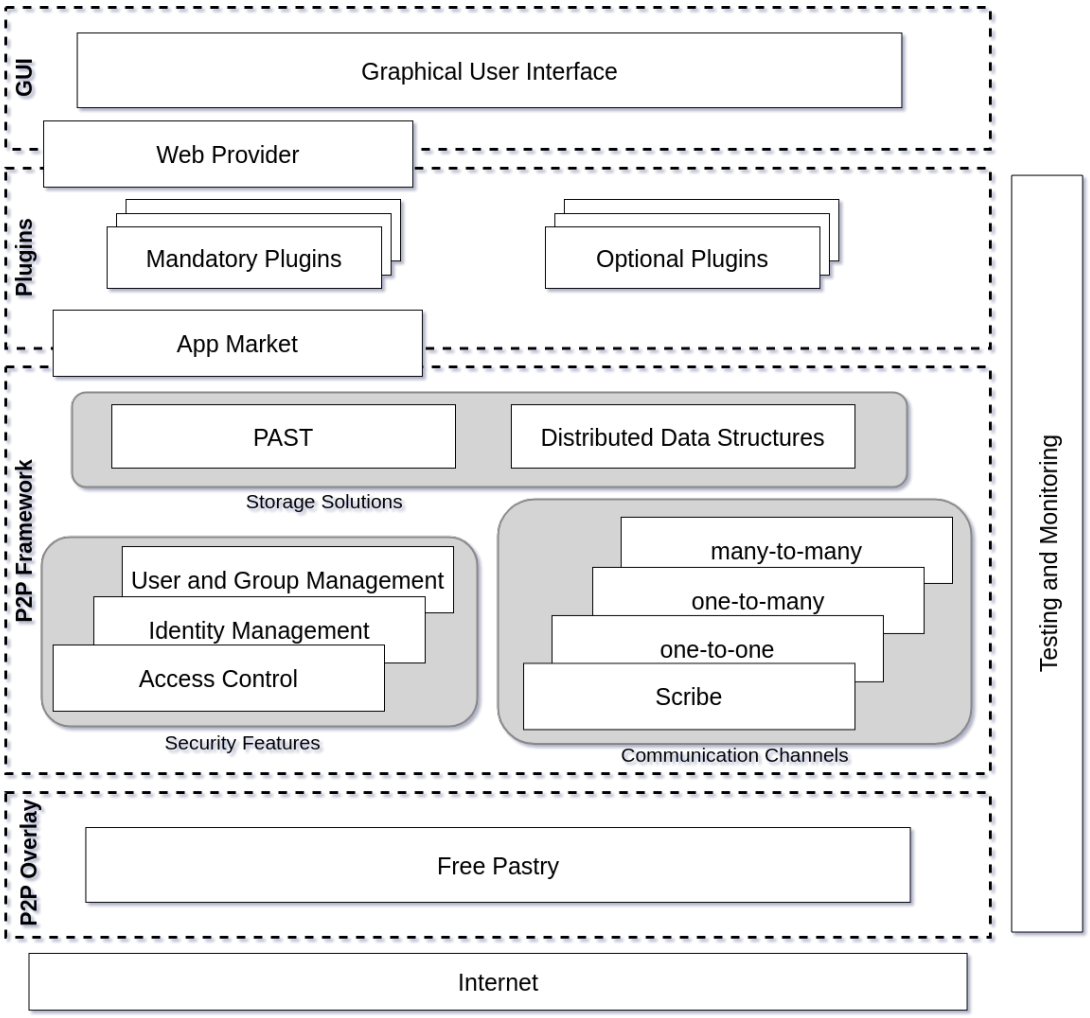


Figure 3.1: Overview of the LibreSocial structure

3.1 The Overlay Layer

The overlay layer is the base for the whole application as it hides the complexity of the underlying physical internet connection and builds a non-hierarchical structured P2P-network that enables secure routing between the peers in the network. LibreSocial uses FreePastry[1] which is a Java open-source implementation of Pastry that was developed in collaboration between Rice University, USA and Max Plank Institute for Software Systems, Germany.

The layer consists of pastry nodes that form a decentralized, self-organizing and fault-tolerant overlay network that provides an efficient request routing, deterministic object location and

[1] `https://www.freepastry.org/`

load balancing in an application-independent manner [Hoy09]. It implements a distributed hash table (DHT) to save the 128-Bit unique numeric identifiers (nodeID) that FreePastry issues to each new node. Such a structure is essential as it is impractical that each node will save information about all other existing nodes in the network. Instead, with this approach, each node has its own *routing table* with a small but well defined subset of nodeIDs, so that it can forward a message to the node that is numerically closest to the nodeID of the destination node. By this routing philosophy, the message is delivered step by step further to the destination with a maximum of $\mathcal{O}(\log N)$ hops while keeping the maximum length of the routing table to $\log N$ entries, with N being the current number of nodes in the network.

Pastry is considered self-organized because each node keeps track of its topologically immediate neighbors in a special Neighborhood Set. Information related to node activity is being tracked, so if a node has gone offline, other nodes are aware of it and failure of the network could be prevented and in a case that new nodes join the network, they will be added to the correct neighborhood sets and routing tables.

3.2 The Framework Layer

The second layer in the LibreSocial application architecture is the framework layer that includes several key components for the functionality of the system such as identity management, storage and replication, access control etc. as could be seen in Figure 3.1. In the next section those components are presented.

3.2.1 Storage Solutions

Two kinds of storage features are offered in LibreSocial, simple and complex. For simple file storage PAST is used and for complex file storage, different distributed data structures are used. PAST will be described in the next section followed by a short discussion of distributed data structures.

a. *PAST*- a data storage and replication layer that is built on top of Pastry (in case of LibreSocial, on top of FreePastry). The main purpose of PAST is to ensure data

availability and persistence storage. Stored data (for example, a picture that was uploaded to a gallery or a text file sent in a group conversation) must be available for other users even if the owner has left the network and is currently offline. In order to avoid the prerequisite that each node will always stay online, data replication is needed.

This persistent storage mechanism is built around the features of the existing Pastry network. Because of that, each PAST-node, similar to Pastry nodes, also has a 128-bit identifier (nodeID) and every data-file has a unique 160-bit fileID that is generated as a secure hash (SHA-1) of the file-name, the public key of the file owner and a random salt (i.e. an arbitrary data item that is used in the process of hashing, in order to make the hash more unique). The file is stored at the node whose nodeID most closely matches the prefix of the inserted file's fileID, and is replicated onto other nodes (mostly to the nodes that are spread out from the leafset of the file's owner node). In case that the owner of the file left the network, PAST ensures that `k` replicas of the original file are always accessible in the overlay layer. The number of replicas is also called the replication factor. All the files are easily found via the Pastry routing mechanism[DR01] [RD01].

PAST provides different operations in order to access files from different nodes: `INSERT` (saves the original file and its `k` replicas across the network), `LOOKUP` (searches for a file using its unique fileID) and `RECLAIM` (a quasi-deletion operation, space occupied by a file and its replicas could be reclaimed and repurposed for future insertion operations).

For the development of LibreSocial PAST was modified as to contain two more unique operations, namely `UPDATE` (replaces the original file and its replicas with the new file) and `DELETE` (deletes the original file and all its replicas from the network). The new `DELETE` operation replaces the original `RECLAIM` operation.

b. *Distributed Data Structures*- in the aforementioned discussion about PAST, it was shown that access to single files for users with appropriate credentials is allowed. In the use case of a distributed social network like LibreSocial, where potentially thousands of users will initiate PAST operations, a single file access is deemed to be too slow. Also, single files could not show a relation (if existing) to other data objects which is sometimes necessary (for example, a relationship between a photo album and the photos inside it).

The Framework Layer solves these problems by introducing different forms of distributed data structures such as distributed linked lists, distributed sets, prefix hash trees, buckets etc. that are similar to their non-distributed counterparts, but take advantage of the decentralized structure of the network to store files on different nodes. Tree-like data structure can show a relation between files, but the search time is relatively slow [Laf17]. To avoid slow search times, a bucket data structure could be used as it can contain multiple files and a reference to the next bucket so that the storage process could be improved.

As shown above, a trade-off has to be found between different metrics, which heavily depends on the individual use case, therefore the Framework Layer supports the usage of different data structures. In LibreSocial, distributed sets are used to manage friend lists, uploaded files and albums and distributed lists are used in forums and to mange the inbox. Prefix hash trees are used to search within a range.

3.2.2 Security Features

LibreSocial implements user identification through a unique userID that grants the user access to only permitted data items to ensure security of user's data. In this section different security features are presented.

a. *Access Control*- as discussed above, insertion and manipulation of stored data is possible in PAST. Access control permits only authorized nodes to access certain parts of the network. In LibreSocial, the combination of public and private keys along with symmetric encryption is used to encrypt and sign the data files to guarantee accessibility, privacy and integrity.

b. *Identity Management*- the distributed social network supports the creation of user profiles for each peer that wants to join the network. During the registration process, a unique userID is created that does not change even if the user changes their credentials (name, password etc.). The identity management components associates the unique userID with the previously mentioned Pastry-nodeID and PAST-nodeID to enable communication between users in the network and to facilitate data storage within the system.

c. *User and Group Management*- the unique userID is also used in the friends list as well as for communication between LibreSocial users. A group in LibreSocial is a user on its own that comprises of different users that share communication and data access permissions. It is also possible to be a part of many groups and to grant different users different types of access rights [GM20].

3.2.3 Communication

A communication channel is where messages between users can be exchanged. Such a channel can either send messages or receive them. A channel is open to all users that are registered to it. LibreSocial offers both synchronous as well as asynchronous messaging. The following are characteristics of the communication channels that are made available in LibreSocial [GM20].

a. *Unicast, multicast and aggregation*- LibreSocial allows three types of transmissions: unicast messaging (one-to-one) can be found in direct messaging between two users, multicast messaging (one-to-many) such as group messaging and multi-chat and aggregation (many-to-many) to spread information throughout the network. Messages can be saved and read at a later time thanks to asynchronous messaging that allow messages to be encrypted and saved in an inbox.

b. *Scribe*- is a publish/subscribe system that operates on a multicast tree structure and allows users to communicate without the knowledge of the address or userID of each other [Cas+02].

c. *Streaming Channels*- LibreSocial utilizes two different streaming solutions to provide to its users live video and audio communication. Those solutions are WebRTC and SplitStream.

   i *WebRTC*- is not a P2P solution but rather it is a service that relays on the web-browser that provides access to sound and video streaming between users.

   ii *SplitStream*- a pure P2P sound and video streaming solution. In this streaming option, the communication is directly between two nodes in the P2P network.

3.3 Plugins and Apps

The topmost layer of LibreSocial is the Plugins and Apps Layer which provides the product an application specific functionality and was developed on top of the Framework Layer. This layer consists of independent software packages called *OSGi*[2] (formally known as Open Service Gateway Initiative) which is a standard for modular system development. A single component in OSGi is called a *bundle* (in Eclipse it is called a *plugin*), the smallest unit of modularization that has an independent lifecycle, i.e., it could be started, stopped and removed independently of the whole system [All18].

The fundamental idea behind those plugins, or bundles, is to provide a set of *services*, via a Java API, to the main *service registry*. Each package can search for the services of other packages in the service registry. This architecture opens the option for services to be activated at any point as they are needed, and deactivated as plugins are not being used any more without affecting the overall system functionality. By using this methodology, application launch time decreases because not all plugins must be loaded at once but rather dynamically loaded during runtime. Interaction between OSGi bundles as well as activation and deactivation etc. of bundles is done with the help of *OSGi commands*. Many useful commands are already built in[3] and new commands can be easily added if needed (see LibreSocial specific commands in the next sections).

All plugins in LibreSocial are managed by the App Store and could be extended and developed independently, thus they remain open for changes and improvements. The following sections give an overview of the currently available plugins in LibreSocial and the corresponding OSGi commands for each one of them.

3.3.1 Login Plugin

The Login Plugin is the entry point to LibreSocial. Its main functionality is to log in or register a user to the network. The Login Plugin works in combination with the Profile Plugin because when a new user registers in the network, a new profile is created using the credentials provided by the user at the login phase. This plugin is also responsible for logging out of

[2] `https://www.osgi.org/`
[3] `https://enroute.osgi.org/FAQ/500-gogo.html`

LibreSocial.

The available OSGi commands of the login plugin are shown in Table 3.1.

Command	Description
Login	Logs the user **in**
Logout	Logs the user **out**
ChangePass	Changes the user password
IsLoggedIn	Indicates whether the user is currently logged in
LoginStatus	If user is logged in- returns the username, if user is logged out- return appropriate message

Table 3.1: OSGi commands for the Login Plugin

3.3.2 Wall Plugin

Every user in the LibreSocial network has their own wall and it's the first thing that users see after they log in the network. The wall provides basic functionality of social networking through a one-to-many communication method. Each user can post text and photos on their wall and decide who can view their posts, the posts could be set to private (only seen by the owner), public (everyone can see the post) or friends (only the user's friends can see the post). Users can also comment on those wall-posts. The wall plugin uses a distributed linked list to save wall posts.

The available OSGi commands of the wall plugin are shown in Table 3.2.

Command	Description
AddComment	Adds a comment to the wall post of another user
AddPost	Adds a new wall post to own wall
DeleteComment	Deletes a comment from another user's wall post
DeletePost	Deletes a post from another user's wall
RetrievePosts	Returns a list with all posts of a user with a specified ID

Table 3.2: OSGi commands for the Wall Plugin

3.3.3 App-Store Plugin

As discussed above, the app store is responsible for all available plugins in LibreSocial. Another functionality of it is to give users the ability to add their own plugins or to load them from an external source (only if the external plugin supports an OSGi Framework).

No external commands for the app store plugin exist at this point as most of the plugin interactions are done internally.

3.3.4 Calendar Plugin

The calendar plugin provides a basic functionality of a paper calendar or agenda. This plugin allows the user to privately schedule and keep track of important events and appointments. The user can create, modify and delete events. A notification function is also implemented in the plugin.

The available OSGi commands of the calendar plugin are shown in Table 3.3.

Command	Description
CreateEvent	Creates an event for a specific user with the given parameters
DeleteEvent	Deletes an event with the given ID
EditEvent	Edits an event with the given ID a new parameters
GetEvents	Returns all events for a specific month and year

Table 3.3: OSGi commands for the calendar Plugin

3.3.5 Groups Plugin

This plugin enables users to create, join, delete and invite other users to a group. Groups help users to connect with other users with the same hobbies, common interests or place of birth etc. It provides an essential platform in terms of collaboration and E-learning. After a user joins a group, they can post messages to the group's wall (similar to the wall plugin). The

wall entries of each group are stored in a distributed linked list.

The available OSGi commands of the group plugin are shown in Table 3.4.

Command	Description
AddFolder	Creates a folder with specific parameters
AddFiles	Adds files into a specific group's folder
AddFriendstoGroup	Adds friends into the group
AddGroup	Creates a group with the given parameters
DeleteContent	Deletes a file from the group's folder
DeleteGroup	Deletes the group with the given ID and all its content
GetContents	Returns all the content of the group
GetGroupsList	Returns a list of the current logged-in users
LeaveGroup	User leaves the group
RenameContent	Renames a file

Table 3.4: OSGi commands for the Group Plugin

3.3.6 Forum Plugin

This plugin enables users with common interests to exchange messages, i.e, threads, about different topics. Each LibreSocial group has its own forum that the members can use to chat, upload files, upload photos etc. Only registered group members can access the stored files.

The available OSGi commands of the forum plugin are shown in Table 3.5.

Command	Description
CreateThread	Creates a new thread inside a forum
CreatePost	Creates a post (comment) under a certain thread
GetForum	Returns all information about a specific forum, including threads and posts

Table 3.5: OSGi commands for the Forum Plugin

3.3.7 File-Storage Plugin

This plugin enables the storage of data (files, photos etc.) and it does not have any direct OSGi commands, but it receives commands from the groups and the forum plugin.

3.3.8 Friends Plugin

The friends plugin is used to store, view and manage all the friends that a user has inside LibreSocial network. It shows the social interconnection facet of the network which can be different from the underlying technical network structure. The plugin provides several user management functions like sending, accepting and declining a friend request. User can also block and un-block other users.

The available OSGi commands of the forum plugin are shown in Table 3.6.

Command	Description
AcceptFriend	Accepts a friendship request that was sent to the user and adds the username of the initiator to the friends-list
BlockFriend	Blocks the given friendship request
BlockFriendFromUserFriendList	Blocks a friend from user's friends-list
BlockListLoaded	Indicates whether the requested blocked-friends-list is loaded
CheckMessagesCommand	Checks for undelivered messages for the friends plugin
DeclineFriend	Declines the given friendship request
DeleteFriend	Removes a user from the friends-list
FriendsMsgAvailable	Indicates whether new friends messages are available
GetBlockFriends	Returns a list of all users that are blocked from the currently loaded friends-list
GetFriends	Returns a list of all users from the currently loaded friends-list
GetFriendsBlockListHost	Shows the username of the **blocked** friends-list owner, which is currently loaded
GetFriendsListHost	Shows the username of the friends-list owner, which is currently loaded
GetFriendsMsgs	Returns all messages and friend requests
GetMyFriendsBlockListAndWait	Requests user's own **blocked** friends-list and waits for its arrival if not available

Command	Description
GetMyFriendsListAndWait	Requests user's own friends-list and waits for its arrival if not available
ListLoaded	Indicates whether requested friends-list is loaded
LoadFriendsBlockList	Returns the list of all blocked users for the given user
LoadFriendsList	Returns the user's friend-list
RequestFriend	Sends an friendship invitation to the specific user

Table 3.6: OSGi commands for the Friends Plugin

3.3.9 Live-Chat Plugin

The live chat plugin allows a user to interact with other users via private instant messages. The plugin uses an IP-based messaging that the underlying framework layer provides, hence the message transmission happens in real time and without noticeable delays. The user interface provides a message history for previous conversations. The plugin also provides the option of a chat channel between multiple users, i.e, multi-chat.

The available OSGi commands of the chat plugin are shown in Table 3.7.

Command	Description
CheckForUndeliveredMsgs	Checks for new chat messages
GetChatMsgs	Returns all chat messages that are available
LiveChatHistoryForUser	Returns chat history with another user
LiveChatHistoryUsers	Returns list of **usernames** that the user had a chat with
LiveChatMsgsAvail	Checks whether new chat messages are available
LiveChatSendMsg	Sends a chat message to a specific user
MultiChatSendInv	Sends an invitation to multiple users
AcceptInvitation	Accepts a Multi-Chat request and adds the username of the initiator to one's own friends-list
DeclineInvitation	Declines a Multi-Chat request
SendTopicMsg	Sends a Multi-Chat message
UnsubscribeTopic	Unsubscribe from a channel, i.e. user leaves a Multi-Chat
GetTopicName	Requests the name of the chat

Table 3.7: OSGi commands for the Chat Plugin

3.3.10 Messaging Plugin

The messaging plugin offers email-like functions. Users can write messages and send an attachment along the message. The files that are attached are not sent along with the message but rather they are uploaded to the users file storage and a link to that file is sent along the message in order to save storage space and bandwidth. In contrast to the live chat Plugin, the messages are arranged individually and not by recipients.

The available OSGi commands of the messaging plugin are shown in Table 3.8.

Command	Description
MessagesAvailable	Indicates whether new messages are available
CheckMsgs	Loads undelivered messages for the messaging plugin
DeleteInboxMsg	Deletes an inbox message
DeleteOutboxMsg	Deletes a message that was sent
GetInbox	Returns a list of all message IDs available in inbox
GetOutbox	Returns a list of all sent message IDs available
SendMsg	Sends a message (messaging plugin) to a particular user
ShowInboxMsg	Returns the content of an inbox message
ShowOutboxMsg	Returns the content of a sent message

Table 3.8: OSGi commands for the Messaging Plugin

3.3.11 Notifications Plugin

The purpose of the notifications plugin is to alert the user about new events inside the LibreSocial network such as incoming messages, upcoming calender events, friend requests etc. The notifications are stored in a distributed linked list as an encrypted object so they are stored for the user to access them if they are offline at the original time of arrival.

This plugin currently does not offer any OSGi commands.

3.3.12 Photos Plugin

The photo plugin provides a functionality to upload and delete photos, it is also possible to group photos in different albums. The user can set a visibility option to either private or public, the public visibility options allows the user's friends to see and comment on the album and photos.

The available OSGi commands of the photo plugin are shown in Table 3.9.

Command	Description
AddPhoto	Adds a photo to an album
AlbumsLoaded	Indicates whether the requested list of albums is already loaded
CreateNewAlbum	Creates a new album with given title, comments and privacy settings
DeleteAlbum	Deletes an album
DeletePhoto	Deletes a photo
EditAlbum	Changes information about an album
EditPhoto	Changes information about a photo
GetAlbumHost	Returns the host of the currently loaded album
LoadMyAlbums	Loads all user's albums
LoadPhoto	Loads a photo
LoadUserAlbums	Loads all albums for a certain user
PhotoLoaded	Indicates whether a requested photo is already loaded

Table 3.9: OSGi commands for the Photo Plugin

3.3.13 Commands Plugin

The commands plugin defines an interface for all the OSGi commands. When an OSGi command is executed in LibreSocial it first arrives at this plugin before it begins the execution cycle.

3.3.14 Search Plugin

This plugin provides basic search capabilities from within the LibreSocial network. A user can search for other users by a certain criteria like country, username etc.

The available OSGi commands of the search plugin are shown in Table 3.10.

Command	Description
Search	Searches for usernames based on given criteria
IsCompleteCommand	Indicates whether previously executed search is complete
GetSearchResults	Returns a list of users for the previous search

Table 3.10: OSGi commands for the Search Plugin

3.3.15 Voting Plugin

The purpose of the voting plugin is to provide basic polling options. A user can post a question and provide different answer possibilities, their friends can participate in the vote and choose their preferred answer(s). It is also possible to create private votes.

The available OSGi commands of the voting plugin are shown in Table 3.11.

Command	Description
CreateVote	Creates new vote
DeleteVote	Deletes a vote
AddPublicVote	Makes a vote public
InviteUser	Invites users to participate in a vote
Vote	Votes in a voting
GetVotingResults	Returns results of a vote
GetMyVotings	Returns all the votes a user created, added or received

Table 3.11: OSGi commands for the Voting Plugin

3.3.16 Monitoring Plugin

The monitoring plugin is in charge of gathering information about the system and converting that information into a user friendly format such as a sqlite3 format. The plugin will be discussed in depth in chapter 4.

The available OSGi commands of the voting plugin are shown in Table 3.12.

Command	Description
monitorStart	Starts the monitoring plugin
monitorStop	Stops the monitoring plugin
monitorStatus	Prints the current status of the monitoring (i.e if the plugin is running or not)
monitorGui	Opens a graphical user interface with the gathered statistics

Table 3.12: OSGi commands for the Voting Plugin

3.3.17 Tester Plugin

This plugin forwards the received command from the user's console (and not from within the network) to all registered slave nodes. The commands initiate different testing modes and different test cases. It will be discussed in more detail in chapter 4.

The available OSGi commands of the voting plugin are shown in Table 3.13.

3.3.18 Web-Provider Plugin

The Web-Provider Plugin serves as the interface between the plugins and backend part of the network and the graphical user interface (GUI) which is the forth layer in the LibreSocial stack. An example screenshot of LibreSocial GUI can be seen in Figure 3.2.

Command	Description
Login	Initiates one node login
Test	Runs a test case that creates a photo album
ToggleDelegaterView	Opens GUI window for the test plugin- slave mode
toggleMasterView	Opens GUI window for the test plugin- master mode
LogRandomSlavesOutX	Logs out X nodes from the network. Amount of nodes available for this command are:2, 25, 50, 100, 200 and 250 nodes
executeTimedTestPlan	Starts executing a timed test plan
RandomTest	Starts executing a random test plan
StopRandomMode	Stops random test execution
sendFriendRequest	Sends one friend request to a random node in the network
sendFriendRequestInOrder	Initiates a friend request test case for all nodes
Memory	Prints the total, free and used memory
Logout	Initiates a log out
Quit	Quit tester plugin

Table 3.13: OSGi commands for the Voting Plugin

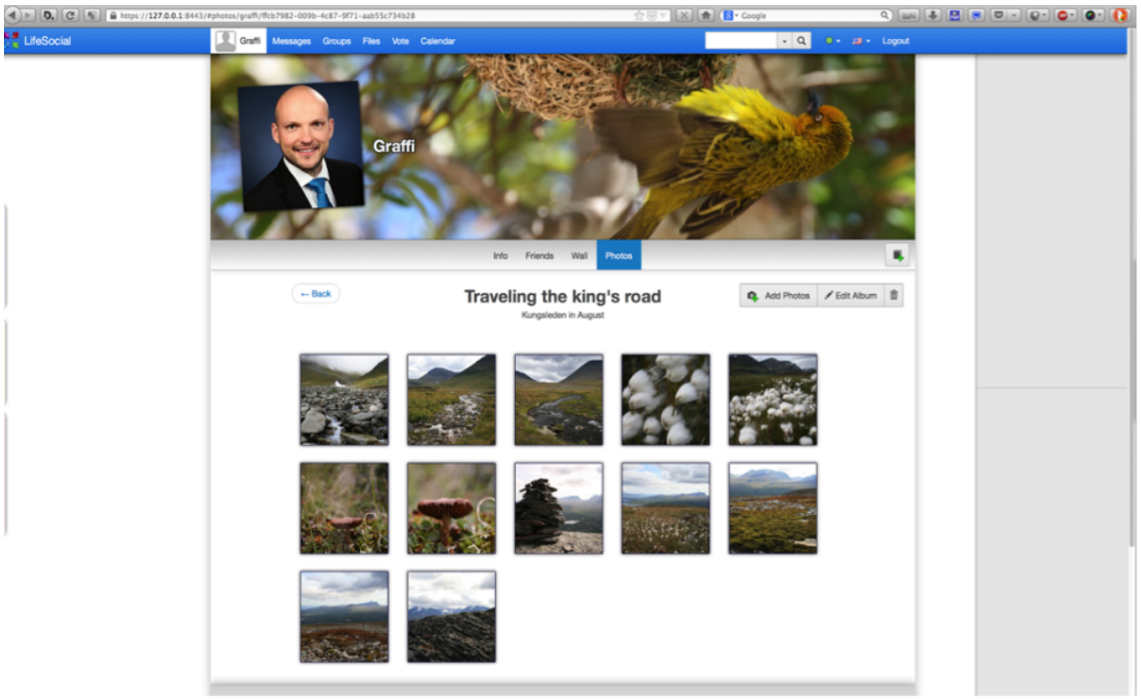


Figure 3.2: The LibreSocial graphical user interface

3.4 Summary

In this chapter we introduced LibreSocial, a fully distributed online social network. We have detailed on the different layers that comprise LibreSocial, namely the overlay layer, the framework layer and the plugins and apps layer. We have discussed about the FreePastry protocol which comprises the overlay layer, next we have detailed the different key components that compose the framework layer, namely storage solutions, security features and communication channels. Furthermore, we have introduced the plugin layer and talked about all the currently implemented plugins that LibreSocial offers.

In the next chapter we will go into detail about the LibreSocial testing framework which consists of the testing and monitoring plugins.

Chapter 4

LibreSocial Test Framework

In addition to what was discussed in the previous chapter LibreSocial also includes a testing framework as shown in Figure 3.1, which incorporates a monitoring aspect and testing feature, both are incorporated as plugins. We will forecast on the monitoring and discuss the testing aspect next.

4.1 Monitoring

Slawek Ligus defines monitoring as "... *the process of maintaining surveillance over the existence and magnitude of state change and data flow in a system. Monitoring aims to identify faults and assist in their subsequent elimination. ... A set of software components used for data collection, their processing, and presentation is called a monitoring system*" [Lig12]

As the definition suggests, the goal of monitoring is to collect quality metrics, in order to have a constant view of the system's state and performance. A distributed system usually consists of a large number of processes running on many different machines. Monitoring such a system requires keeping track of all the processes on all the machines to ensure that they are performing as expected.

4.1.1 SkyEye.KOM - Information Management Over-Overlay for Structured P2P Systems

SkyEye.KOM is a monitoring mechanism developed by Kalman Graffi et al. in the Darmstadt University of Technology [Gra+08]. SkyEye.KOM has a special information management approach as it acts as a new layer above the overlay layer, hence its name information management over-overlay. It forms a tree topology and gives each node that joins the network a unique domain-ID in regards to its position in the tree, SkyEye.KOM calculates metrics based on that.

In SkyEye, each node collects metrics about itself (for example, metrics about storage space or network activity) and sends them to the parent node, which in turn calculates its own metrics, aggregates the metric-information sent by its children nodes and then finally sends everything to its parent node. This process repeats itself until the root node of the tree topology receives information from all the nodes. The root node aggregates the received information with its own metrics and then it sends the global metrics back to all the nodes down the tree as seen in Figure 4.1. Moreover, as a result of constant node position calculation and regular communication with other nodes, SkyEye.KOM offers an easy detection mechanism for faulty or offline nodes. This monitoring tool is depicted in Figure 4.2 as the monitoring process.

SkyEye.KOM is a powerful tool that allows us to gather data on storage information, message sending rates, download rate etc. in order to calculate the distributed system's availability, scalability and other quality aspects.

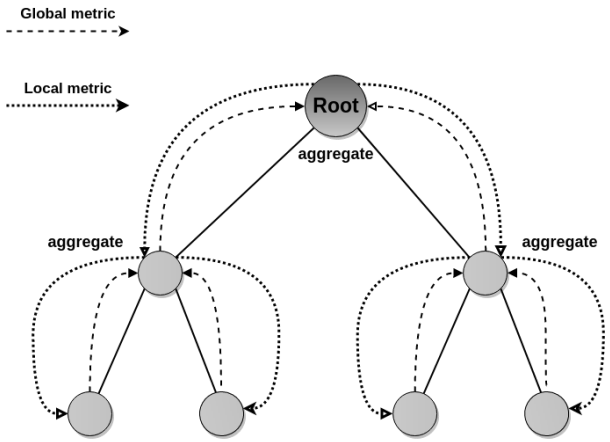


Figure 4.1: Metric aggregation strategy

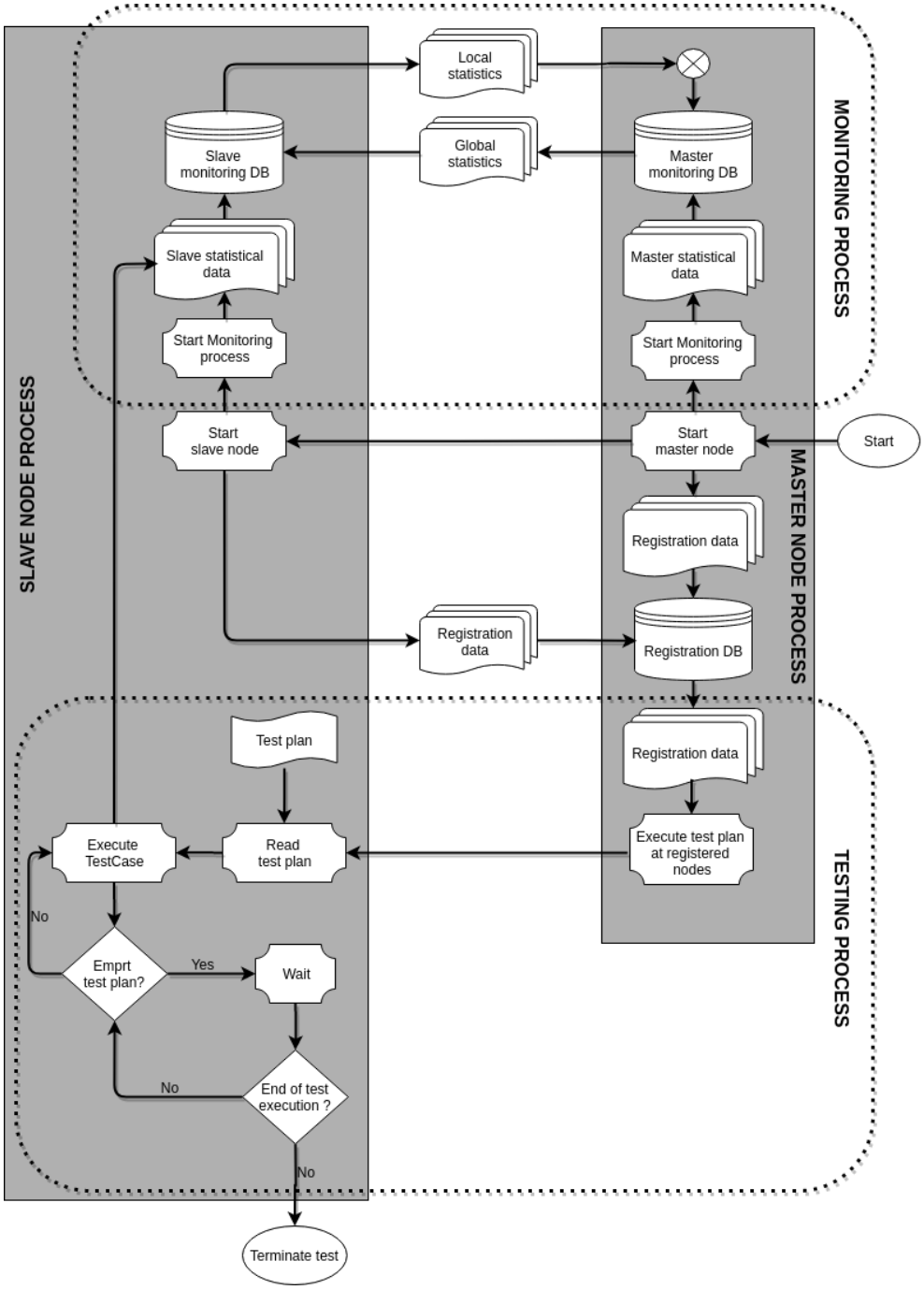


Figure 4.2: Flowchart of testing and monitoring processes in LibreSocial

4.1.2 Metrics

Metric is a standard measurement scale that represents a specific attribute. Metrics could be computed for the whole system under test or for an individual node, e.g., storage usage or data-rate of a single node. With the utilization of SkyEye.KOM monitoring tool, the generation of global metrics at runtime that are based on individual node metrics is possible. Attributes of a metric contain the following basic information: value, aggregation strategy, source, unit and name [Liu13].

- *Unit-* is highly customizable and depends of the measured attribute. Examples of units when measuring data are bytes or megabytes and when considering a time specific action a unit can be milliseconds or seconds.
- *Value-* describes what is being measured, for example, memory usage size, messages received and routing table size etc.
- *Aggregation strategy-* indicates which metric is being aggregated. Strategy choices are: summation, mean, maximum, minimum and count.
- *Source-* specifies where in framework the data is collected from. For example, when considering the value "messages sent", the source point could be network, message dispatcher or message plugin.
- *Name-* consists of the source, value, aggregation strategy and unit. An example name is `NETWORK_MEMORY_USAGE_SIZE_MAX_MB`. `NETWORK` is the source, `MEMORY_USAGE_SIZE` is the value, `MAX` is the aggregation strategy and `MB` is the unit.
- The prefix `GLOBAL` indicated that the metric data is aggregated from the whole network. If the prefix is not written then the metric data is accurate for a specific local node.

SkyEye.KOM collects more than 100 unique metrics for the network, storage dispatcher, all the plugins etc., for example,
`PLUGIN_LOGIN_NUMBEROFLOGIN_COUNT_elements`
`GLOBAL_PLUGIN_GROUPS_JOINGROUP_COUNT_elements`
`GLOBAL_NETWORK_MESSAGES_RECEIVED_COUNT_messages`
`GLOBAL_NETWORK_PASTRY_DATA_SENT_RATE_MAX_COUNT_kbytes_per_s`
In order to evaluating LibreSocial even better, two new metrics were added during the work on this thesis, namely `NETWORK_LEAFSET_SIZE` to monitor the size of the leafset of a

LibreSocial network and `NETWORK_ROUTING_TABLE_SIZE` to monitor the routing table size for the nodes that are connected to LibreSocial's network. For both new metrics all the defined aggregation strategies are available as well and the metric can be gathered for a single node or it could be aggregated for the whole network. The importance of the two values will be addressed in the next sections.

4.2 Testing

The goal for testing LibreSocial is to put the application under different sets of conditions, vary the amount of load on the system and vary the stress that is put on it, in order to observe the performance and functionality of the nodes in the network, the P2P framework and the FreePastry overlay. Furthermore, a discussion about the benchmarking of LibreSocial will be held. Benchmarking is a process of measuring the performance and the capabilities of a service or a product in order to identify areas for improvement and help gain necessary insights about the product. In this chapter, the LibreSocial test plugin and the structure of test cases will be introduced. The testing plugin is depicted in Figure 4.2 as the testing process.

4.2.1 Test Plugin

The test plugin was developed in order to test individual functionalities of LibreSocial. It can be further used by combining individual tests and simulating a real-user behavior to carry out a more comprehensive test of the whole system. The test plugin can be run in two different modes, a *Master* mode and *Slave* mode as shown in Figure 4.2. It is important to differentiate at this point between the two. In every test run there could be only one master node, but many slave nodes. The master node is responsible for reading all necessary configuration files, receive the test plan and delegate it to all registered slave nodes and therefore it must be the first nodes that enters the network.

The master node does not participate in the execution of the test itself, but rather it keeps track of the LibreSocial network, for example, it creates and updates lists of all registered and online users, creates and updates lists of available bootstrap addresses and used ports etc. Each node reads a special LibreSocial configuration file (see next section for more information) and

compares its own IP-address to the IP-address in that file. If they match, the node is set to be a master node. If they don't match the node is a slave node.

In contrast to the master node, once connecting to the LibreSocial network and registering at the master node, a slave node awaits for further instruction. When it receives a test plan from the master node the slave node starts executing tests. There should be at least four slave nodes in the network in order to execute tests as this is the smallest network size in LibreSocial.

All essential files for test execution such as configuration files, test plan, files will name pool etc. must be included in a `LibreSocial/resources` folder.

Configuration Files

In LibreSocial, several configuration files exist in order to change key parameters and settings for the application and network. In the following segment, a couple of these files will be shortly presented and described.

- *LibreSocial configuration file for testing purposes* - this file is reviewed in every instance start of LibreSocial, hence it must be included in the resources folder under `LibreSocial/resources/config.txt`. This configuration file provides the test plugin important information about the test environment, in particular the IP-address of the master node (line 2), whether the test-plugin should be enabled or not (line 5), path to important files such as the test plan (lines 18, 21 and 24), important waiting times (lines 30, 33, 36) and other parameters. Every line that starts with '#' is a comment and is ignored by the source code.
- *LibreSocial configuration file* - this file can be found under `LibreSocial/plugins/org.lifesocial.utilities_1.0.0/config/LibreSocialConfig.txt` and it provides additional important information about the LibreSocial network such as size of heterogeneity cache (line 8), whether the monitoring plugin should start automatically (line 33) etc. One important parameter that can be changed in this file is the replication factor (line 18) (previously discussed in 3.2.1). This parameter will be changed several times in the process of benchmarking LibreSocial and will be further addressed in Chapter 5.
- *FreePastry parameters file* - this file is accessible only from inside the eclipse IDE

under the `rice.freepastry` bundle. It holds default values for pastry and the application such as different buffer sizes, socket parameters etc. One crucial parameter is `pastry_lSetSize` (line 14) that lets us modify how big the list of closest peers by nodeID each node has. The leafset size is directly connected to the replication factor as it must not exceed half the size of the leafset. The importance of this parameter will become more apparent in Chapter 5.

After configuring the aforementioned files test cases can be then deployed. A test case is a set of operations that are executed in a sequence to review a certain feature of the software. A test case usually contains input data, preconditions and post-conditions for a specific test scenario that needs to be verified. The output of the test and system logs are then compared to the expected results to determine if the system passed or failed that specific test case. Development and execution of test cases help find problems with the software that contrast with the requirements. The specific LibreSocial test case structure is addressed in the next segment.

4.2.2 Test Cases in LibreSocial

There are 56 software tests for LibreSocial in total (also called 'test cases'), however, some of the test cases did not work properly or completely failed. Table 4.1 lists all of the test cases in alphabetical order and indicates for each one whether it was faulty in any way or if it worked properly before this bachelor thesis. As it is seen in Table 4.1, 28 out of 56 which is 50% of the tests that were not operational and had to be examined and fixed.

Test Case Structure

All test cases extend the same abstract class TestCase and hence, all of them follow the same structure and logical flow. It was designed by project group of year 2011 in the university of Paderborn [AV12]. Each implementation of the abstract class contains three other implementations, namely the *Precondition*, the *Parameters* and the *OSGi commands*. In order to execute a test case, the algorithm checks first if all the necessary parameters are available, if not, a method that automatically generates them is invoked. Then the preconditions are checked, if they are not fulfilled, a method that checks for the required precondition is

Test Case Name	Status		Test Case Name	Status	
	previous	current		previous	current
TestCaseAddPublicVote	✗	✓	TestCaseSendMessage	✗	✓
TestCaseBlockUser	✗	✓	TestCaseSendMultiChatInvitation	✓	✓
TestCaseChangeProfileInformation	✓	✓	TestCaseSendMultiChatMessage	✓	✓
TestCaseChangeProfilePicture	✗	✓	TestCaseSendWallPost	✓	✓
TestCaseCommentForumThread	✓	✓	TestCaseUploadFileinFolder	✗	✓
TestCaseCommentWallPost	✓	✓	TestCaseUploadPhoto	✓	✓
TestCaseCreateCalendarEvent	✗	✓	TestCaseViewCalendar	✗	✓
TestCaseCreateFolder	✗	✓	TestCaseViewFolder	✗	✓
TestCaseCreateForumThread	✓	✓	TestCaseViewForum	✓	✓
TestCaseCreateGroup	✗	✓	TestCaseViewForumThread	✓	✓
TestCaseCreatePhotoAlbum	✗	✓	TestCaseViewFriendsAlbums	✓	✓
TestCaseCreateVote	✗	✓	TestCaseViewFriendsPhoto	✗	✓
TestCaseDeleteAlbum	✗	✓	TestCaseViewFriendsProfile	✓	✓
TestCaseDeleteCalendarEvent	✗	✓	TestCaseViewFriendsWall	✓	✓
TestCaseDeleteFilefromFolder	✗	✓	TestCaseViewGroup	✓	✓
TestCaseDeleteFolder	✗	✓	TestCaseViewInboxMessage	✓	✓
TestCaseDeleteInboxMessage	✓	✓	TestCaseViewMyGroupsList	✓	✓
TestCaseDeletePhoto	✓	✓	TestCaseViewOutboxMessage	✓	✓
TestCaseDeleteVote	✗	✓	TestCaseViewOwnAlbums	✓	✓
TestCaseEditAlbum	✗	✓	TestCaseViewOwnProfile	✓	✓
TestCaseEditCalendarEvent	✗	✓	TestCaseViewOwnWall	✓	✓
TestCaseGetMyVotings	✗	✓	TestCaseVote	✓	✓
TestCaseGetVotingResults	✓	✓	TestCaseVotingInviteUser	✗	✓
TestCaseInviteFriendtoGroup	✓	✓	TestCaseSendFriendRequest	✗	✓
TestCaseLeaveGroup	✗	✓	TestCaseSearch	✓	✓
TestCaseLeaveMultiChatMessage	✓	✓	TestCaseRenameFolder	✗	✓
TestCaseLogin	✗	✓	TestCaseRegister	✓	✓
TestCaseLogout	✗	✓	TestCaseLogoutLogin	✗	✓

Table 4.1: Operational status of all test cases in LibreSocial

invoked and they are fulfilled. After this checking procedure, a second round of checking for the parameters and preconditions takes place in order to make sure that no errors during the fulfillment of either the parameter generation nor during the precondition fulfillment has occurred. If at this point any errors occur, the test will terminate and display an appropriate message. But if no errors are found, the test will generate the correct OSGi command and send it to the LibreSocial network for execution. At the end of the OSGi command execution, an appropriate notification for the test's result will appear. It is important to mention that any error occurrences during tests should not interrupt the node's functionality or its connection to the LibreSocial network.

Preconditions

The precondition class is an abstract class with the sole purpose of checking whether the defined precondition is fulfilled and if not to fulfills it. Tests in LibreSocial must have preconditions because sometimes, in order to execute a certain function inside the network, some previous actions must have already taken place. For example, in order to execute a test for sending a vote request to a friend, the node that executes the test must be registered and logged into the LibreSocial network, furthermore it must have at least one other user in its friends-list and it must have an already created vote. Only after checking that all of the above stated preconditions are fulfilled, can a voting request be sent to a friend, or else the test fill fail. 27 unique preconditions are available in LibreSocial, a list of all precondition is presented in Table 4.2 below.

Parameter Generation

Different and random parameters are required for the execution of all tests, for example, user's first name, user's city and country of birth, text for sending messages, a name for a new group etc. The test plugin retrieves all the necessary parameters from the Resource Folder. The existing files in this folder are: .txt files for city names, country names, group names, interests, messages, login names, pictures to upload to a photo album and a .pdf-file. These files help the test plugin automate the execution of tests.

During the work on this thesis, the file containing user names was updated. Beforehand,

the list contained 147 names and now the list contains almost 20,000 unique names[1]. This increase in the number of names is important for the execution of a large-scale test, where a lot of nodes participate. Another change was made to the cities and countries lists. In the registration process to LibreSocial, the user must provide a city and a country of birth. Previously, the automated mechanism could choose a city and a country independently of each other, for example, choose Berlin as a city, but Argentina as a country, even though Berlin is not in Argentina. During the work of this thesis, a new text-file with tuples of city and country[2] was introduces in order to solve this inconsistency. The source-code was modified accordingly in both cases.

Precondition Name	Precondition Name
PreconditionAlbumExisits	PreconditionAlbumExisitsNot
PreconditionAlbumsLoaded	PreconditionCalendarLoaded
PreconditionFriendlistLoaded	PreconditionFriendsHavePhoto
PreconditionGroupNotExist	PreconditionGroupsLoaded
PreconditionHasAlbum	PreconditionHasCalendarEvent
PreconditionHasFile	PreconditionHasFolder
PreconditionHasForumThread	PreconditionHasFriendship
PreconditionHasGroup	PreconditionHasInboxMessage
PreconditionHasPhoto	PreconditionHasTopicchannel
PreconditionHasVote	PreconditionLoggedInUser
PreconditionProfileLoaded	PreconditionUserExists
PreconditionUserExistsNot	PreconditionUsernameNotLoggedIn
PreconditionVoteNotExist	PreconditionVotesLoaded
PreconditionWallPostExists	

Table 4.2: List of all available Preconditions in LibreSocial

[1] `https://github.com/smashew/NameDatabases/blob/master/NamesDatabases/first%20names/all.txt`

[2] `https://datahub.io/core/world-cities`. The file was manipulated by the writer on this thesis to fit the requirements of the system.

4.3 Summary

In this chapter, the unique monitoring and testing mechanisms of LibreSocial were presented and discussed in great detail. In the following chapter, we discuss the benchmarking and different tests that were performed on LibreSocial using the the testing plugin, the results that were generated thanks to the monitoring plugin will also be discussed.

Chapter 5

System Evaluation

We aim to evaluate LibreSocial's quality aspects by testing the system's quality. The statistics and metrics generated by the testing and monitoring plugin will help us to evaluate those quality aspects. In this chapter, the test plans and the results that were used in an effort to benchmark LibreSocial are described. In order to test an application, a common set of metrics must be defined so that the evaluation of the results will be made in an objective manner. First of all, the quality aspects that are used for testing as a part of this thesis are described, then the tests and their results are presented and discussed.

5.1 Quality Aspects and Relevant Metrics

Up until this point the theoretical background of distributed systems and LibreSocial was discussed. In this section several quality aspects of distributed systems that will help us analyze and benchmark LibreSocial are defined.

Quality aspects are a common set of metrics that help to evaluate all types of distributed systems. Even though several quality aspects are defined in the literature, we chose to proceed on focusing only on the quality aspects that are in great relevance for benchmarking LibreSocial in particular. For each quality aspect we also present the specific metrics (gathered by the monitoring tool SkyEye.KOM) that are used to evaluate them, a summary of the quality aspects and metrics is shown in Table 5.1. The following definitions of the quality aspects are based on [ESS13].

a. *Performance*- describes the quality level for an operation executed in a P2P system in terms of its responsiveness (how fast the system reacts), throughput (how much load that system can handle in a given time frame) and the extent to which the results match the expectation. The metrics to evaluate the performance of LibreSocial are: *hop-count*, the number of nodes between source and destination, *data storage time*, how fast a data item is saved to the network, *data retrieval time*, how fast data is retrieved from the network, *sent messages rate*, how fast a message is sent, *received messages rate*, how fast a messages arrives at the destination.

b. *Costs*- the amount of resources used in order to fulfill a particular action. In question here is the amount of resources that the application can provide in order to complete a certain task. Relevant cost metrics are *used storage space*, *used memory* and *network bandwidth*.

c. *Stability*- expresses the capability and sensitivity of the system to dynamical and behavioral changes that relate to events occurring in the P2P system. In other words, stability represents the systems ability to perform in the same manner when there is a peer-fluctuation, like churn. The relevant metrics are *number of nodes* in the network at any given time, the *leafset size* and *routing table size*.

d. *Scalability*- describes the ability of the system to function appropriately when the number of peers fluctuate and its capability to adapt to a dynamic number of peers. In distributed systems, there are two common scaling dimensions when talking about benchmarking: horizontal, the number of peers in the system, and vertical, the load generated by each peer. In this thesis we only scaled LibreSocial horizontally.

The system is tested under three different test scenarios: user pseudo-random behavior is presented in section 5.2, replication factor in section 5.4 and stability test in section 5.3. In each of the sections the test scenario will be further described with the test parameters and the obtained results. The test environment for all three test scenarios is the same. The tests are deployed to the high performance cluster at the university of Düsseldorf[1] (the HILBERT cluster) that provides capability to support up to 4000 LibreSocial nodes.

In the below presented graphs, a vertical dashed line is added to indicate the start of the test plan as the time for all nodes to successfully connect to the LibresSocial network vary and is

[1] `https://www.zim.hhu.de/high-performance-computing`

dependent on the total number of nodes in the test.

Quality Aspect	Relevant Metrics
Performance	Hop-count
	Data storage time
	Data retrieval time
	Sent messages rate
	Received messages rate
Cost	Used space
	Used memory
	Network bandwidth
Stability	Number of nodes
	Leafset size
	Routing table size
Scalability	Number of nodes

Table 5.1: List of used metrics for each quality aspect

5.2 Pseudo-random Behavior

The core intent of this scenario is to test whether LibreSocial is stable under random user behavior. For this we rely on an algorithm that mimics user behavior in a social network that was previously presented [Kal13]. This algorithm is based on a study [Ben+09] that aggregated data about the behavior of more than 35,000 users in four different social networks. From that study several key parameters were utilized: the average time that a user spends on a social network website, their login frequency throughout the day, in which activities the user participates and in what order. Based on the statistics of the study, a *total online time distribution* was implemented, in other words a mechanism that selects a period of time that a user spends online and at the end of that time period the user logs out. Furthermore, an algorithm that automatically chooses test cases from a list of all available test cases was implemented as shown in Figure 5.1. The test case choice reflects user behavior that is based on real world statistical data and probabilities from the study.

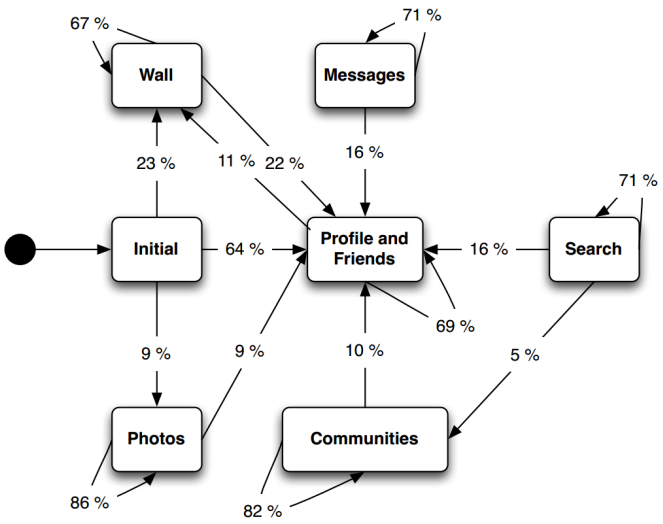


Figure 5.1: LibreSocial plugin transition probability [Kal13]

5.2.1 Experimental Parameters and Scenarios

By using the aforementioned algorithm we can test the scalability and stability of LibreSocial. This can be done by simply running the OSGi command `randomTest`. In this section the system parameters (Table 5.2) and additional important information about this test scenario are presented.

We have considered the following two conditions for testing in this scenario:

a. *Realistic user behavior*- by this we suggest that a user can perform all actions as a normal user does, including logging in and out freely. In this test the setting is `enableAutomaticLogout = true`. However, it was noted that when we ran this type of test the network eventually crashed.

b. *Controlled random behavior*- here users are not allowed to log out once they have logged in to LibreSocial. The logout parameter is set as `enableAutomaticLogout = false`.

Parameters	Value
Leafset size	24
Replication factor	6
Number of nodes	
Test 1	500
Test 2	25, 100, 200
Duration of experiment	
Test 1	
Setup duration	180 minutes
Test duration	24 hours
Test 2	
Setup duration	80 minutes
Test duration	120 minutes
OSGi command	`RandomTest`
`enableAutomaticLogout`	
Test 1	true
Test 2	false

Table 5.2: System parameters

5.2.2 Experimental Results

In this subsection we will discuss the results of both test scenarios. The realistic user behavior results will be presented first and afterwards the controlled random behavior test results will be showcased.

Realistic User Behavior

As observed from Figure 5.2 the test scenario suffered a metric aggregation problem even though the monitoring plugin has worked properly throughout the test. It can also be noticed that the graph is rapidly declining towards the x-axis which means that all nodes have failed during the test plan. All nodes failed approximately 2.5 hours after the start of the test plan. Figure 5.3 shows the global rate for sending messages in the network while Figure 5.4 shows the same metric for a single node. It can be deduced from both figures that communication between nodes was present at first but from Figure 5.4 it is clear that the rate has plateaued at

roughly minute 560 and no further messages were sent for several hours.

Figures 5.5, 5.6 and 5.7 present information about the data rate, the memory usage and data retrieved for distributed data structures (DDS) respectively for a single node which show the same trend as the previous set of graphs. It is evident that the memory and the storage were engaged during the first test cases but later on any further usage had seized.

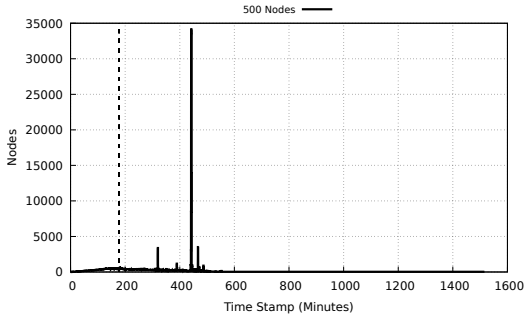


Figure 5.2: Total number of nodes in the network

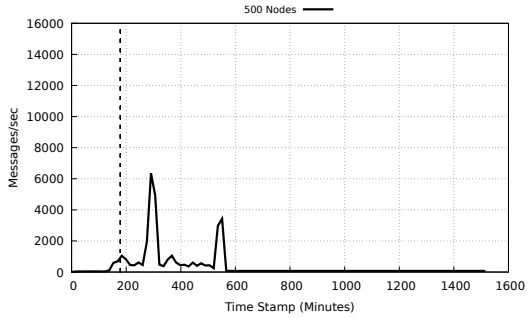


Figure 5.3: Rate of sending messages in the network

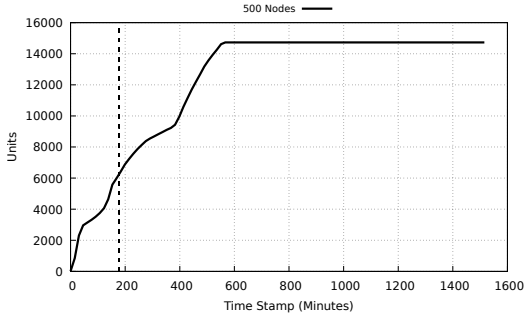


Figure 5.4: Rate of sending messages for a single node

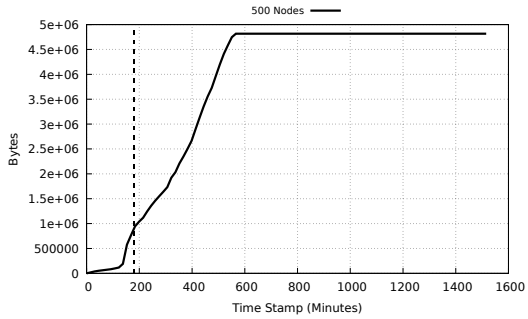


Figure 5.5: Data sent rate for a single node

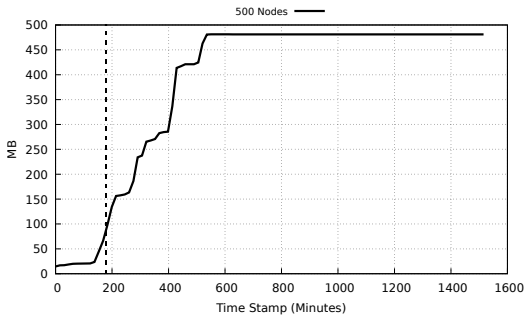


Figure 5.6: Memory usage for a single node

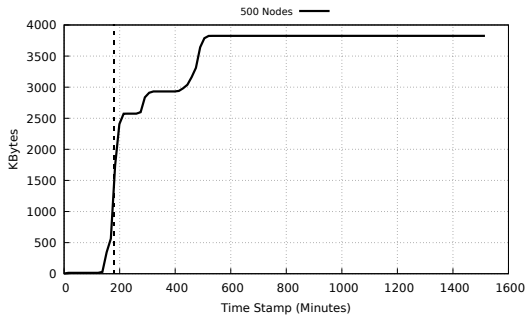


Figure 5.7: Retrieved DDS for a single node

Controlled Random Behavior

As seen in Figure 5.8 the number of nodes for the 25-nodes test scenario is stable throughout the test with only one node failing. The test scenarios with 100 and 200 nodes experienced a more significant amount of node failures. Those failures could be attributed to the random nature of the test plan. In Figure 5.9 the leafset sizes for all three test scenarios are shown and it is clear that the leafset stays stable at the defined size of 25 (24 + 1 master node) for the tests with 100 and 200 nodes. When considering the test that includes only 25 nodes, it is noted that one node fails and the leafset size has to be adjusted. When observing the routing table size in Figure 5.10 it can be clearly recognized that even though nodes failed in all tests, the network recognizes the changes and successfully adjusts to them.

Figures 5.11, 5.12 and 5.13 show some statistical data about messaging across the network. Figure 5.11 depicts the amount of messages that were sent in the network which is as expected. For tests with 200 nodes, the total amount of messages is larger in comparison to 100 and 25 nodes as the number of messages increase when more nodes are present in the network. Moreover, it can be seen in Figure 5.12 that the maximum rate for sending messages is the highest and most stable for 25 nodes and the rate fluctuates for tests with 100 and 200 nodes. This can be easily explained because for the 25-node test, the node count throughout the duration of the test is very stable and hence messages can be sent at the highest rate in a stable manner. For the two other tests the rate is unstable because the nodes constantly fail and leave the network. In addition to this, Figure 5.13 displays the amount of lost messages in the network which is the largest for the 200 node test because many nodes have failed and the lowest for the 25 node test since virtually no nodes failed.

Figure 5.14 addresses the total memory in the network. It can be seen that the test with 200 nodes uses the highest amount of memory due to the fact that the increased amount of nodes produce more data that has to be stored. Figure 5.15 indicates that with an increasing amount of nodes that data rate in the network is higher because more nodes perform actions of data sending and retrieval. Lastly, Figure 5.16 presents the rate of retrieval for distributed data structure (DDS) data. This time the 25-node test has the highest rate of the three graphs which is as expected because with less nodes in the network, each node can send and receive data with a higher network throughput.

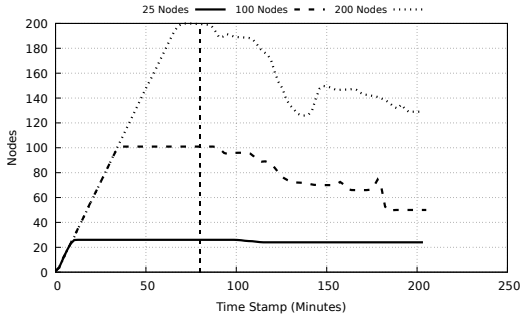


Figure 5.8: Number of nodes

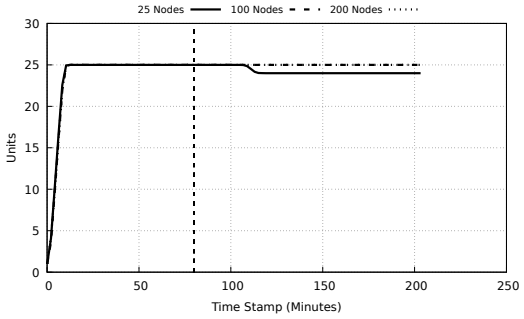


Figure 5.9: Leafset size

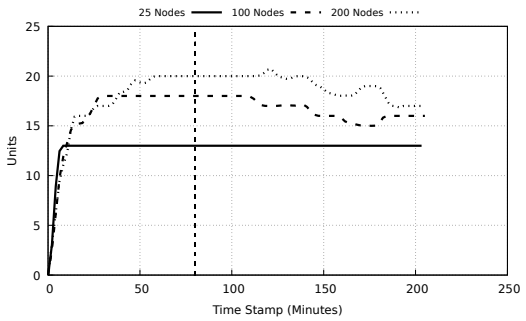


Figure 5.10: Routing table size

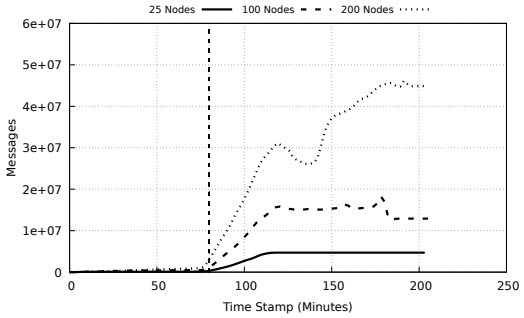


Figure 5.11: Amount of sent messages in the network

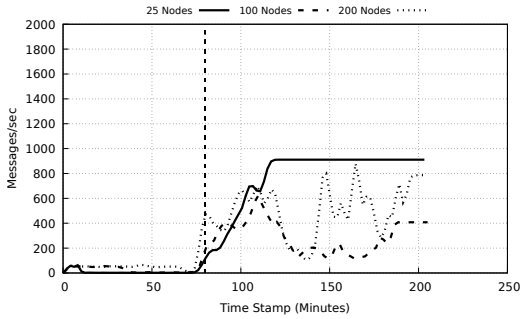


Figure 5.12: Maximum rate of sent messages in the network

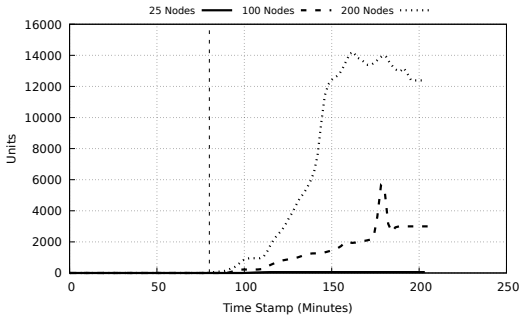


Figure 5.13: Amount of lost messages in the network

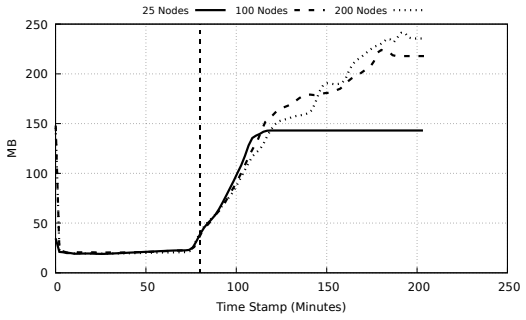


Figure 5.14: Total network memory usage

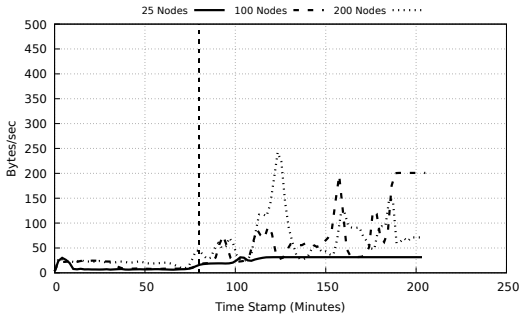


Figure 5.15: Data sent rate (mean)

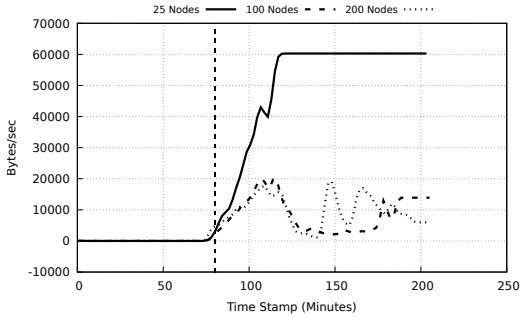


Figure 5.16: Retrieved DDS rate (mean)

5.2.3 Result Summary

From the realistic user behavior test we have discovered an underlying issue with the randomization algorithm. After conducting several tests with the parameter `enableAutomaticLogout` set to false we found that the problem is mitigated but not eliminated. Even though a large amount of nodes have failed during the execution of the controlled random behavior test scenario, we saw from the gathered results and the graphs that the network dynamically adjusts to the dropping number of nodes. The leafset size and the routing table adjusting according to the network behavior. Moreover, the test has ran to the end of its duration without crashing. All of this points to a correct function of LibreSocial and a stable FreePastry network structure.

5.3 Stability Test

In this test scenario, we want to utilize the computational power of the university's high performance cluster in order to add 1000 nodes to the LibreSocial network in order to evaluate the two quality attributes scalability and stability of LibreSocial. In order to do this, we initiate a login of 1000 nodes and test if messages are exchanged properly between nodes as well as some basic functionality of LibreSocial, for example, uploading a photo and commenting on a forum thread and test their underlying distributed data structures (see Table 5.4). Then, we log 250 random nodes out of the network and perform the same set of tests. The logout and testing procedure is repeated three times.

5.3.1 Experimental Parameters and Scenarios

The system parameters are presented in Table 5.3 and the workload parameters which are defined in the test plan are shown in Table 5.4.

Parameters	Value
Leafset size	24
Replication factor	4
Number of nodes	1000
Duration of experiment	48 minutes + set up time

Table 5.3: System parameters

Plugin	Test case	Repetitions	Duration* (minutes)
Messaging	SendMessage	5	2
Photos	UploadPhoto	5	2
Forum	CommentForumThread	5	2
Wall	SendWallPost	5	2
* after each test case a pause of one minute is implemented			

Table 5.4: Workload parameters

5.3.2 Experimental Results

In Figure 5.17 it is evident that LibreSocial can scale up to at least 1000 nodes and remain stable while performing tests with no failures. Also, after logging random nodes out of the network, LibreSocial remains stable and continued on executing tests with no crashes. In Figure 5.18 the total amount of messages exchanged in the network is seen. It is noticed that as the login process is carried out and nodes joined the network, messages between slave nodes and the master nodes were exchanges. As the test plan started at around 320 minutes, more and more messages were exchanged between the nodes to ensure the fulfillment of the tests.

In Figure 5.19 it can be seen that messages initiated in the messaging plugin are sent only from the start of the test plan as the nodes had to send friend request messages, inviting each other to join a group etc. Inevitably, messages can get lost in a big network of 1000 nodes as shown in Figure 5.20, but despite the fact that LibreSocial experienced loss of messages in this test scenario, all test cases ran smoothly to the end.

Regarding the data flow in the LibreSocial network, it is evident from Figure 5.21 that data is sent through the network as nodes are joining and increases significantly as nodes start to perform the test plan. Furthermore, it can be seen from Figure 5.22 that the data rate for sending data items in the network is acceptable at almost 30 KBps.

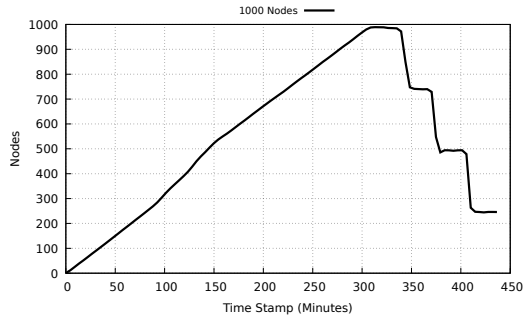


Figure 5.17: Number of nodes

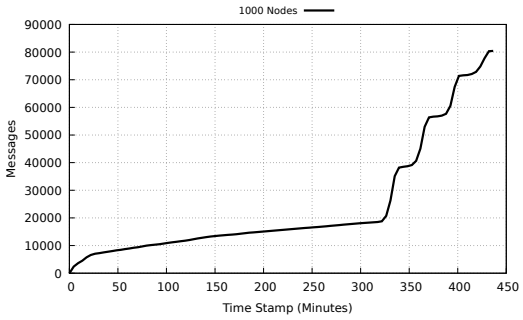


Figure 5.18: Network messages

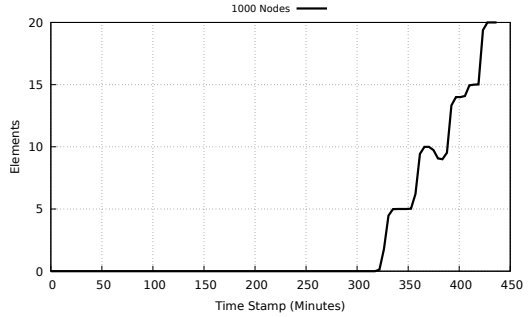


Figure 5.19: Message-plugin messages

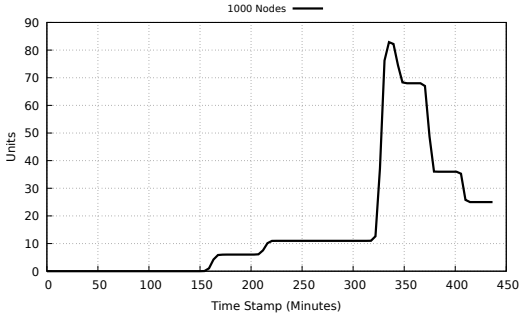


Figure 5.20: Lost messages

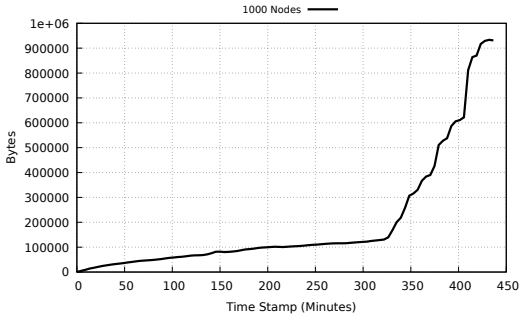


Figure 5.21: Sent data

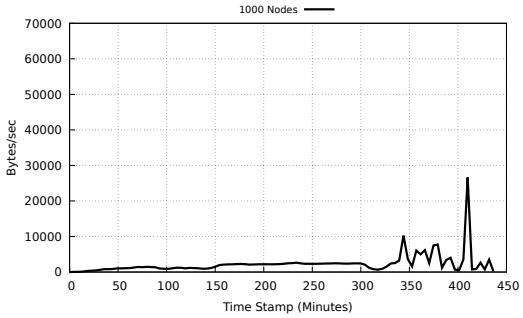


Figure 5.22: Sent data rate

5.3.3 Result Summary

From this test scenario we have gathered vital information about the LibreSocial network and its scalability potential. We have shown that LibreSocial can scale up to 1000 nodes and experienced no crashed and failures. Furthermore, from the consistent results of the underlying network, message and storage dispatchers etc. we are confident that LibreSocial can scale even higher while maintaining its stable operation.

5.4 Replication factor

The replication factor (RF) is responsible for the number of duplicate items in the underlying network so in a case that a peer leaves the network, its previously uploaded files will stay accessible to other peers.

The replication factor variable in LibreSocial is easily accessible through the `LibreSocial/plugins/org.lifesocial.utilities_1.0.0/config/LibreSocialConfig.txt` file. It is important to note that the replication factor is tightly bound to the leafset size of the Pastry network as shown in Equation 5.1. If this requirement is not carried out then the network will not be built and nodes will, therefore, not join the network. The leafset size parameter is found in `rice.freepastry/freepastry.params` which is bundled in the FreePastry library.

In this test scenario the replication factor and its effects are considered.

$$Replication\ Factor = \frac{Leafset\ Size}{2} + 1 \tag{5.1}$$

5.4.1 Experimental Parameters and Scenarios

In order to test the replication factor in LibreSocial three main variables could be changed: the replication factor itself, the number of participating nodes and the test plan. We decided to divide this test scenario further into two smaller test scenarios such that for each number of nodes three different replication factor values will be tested while executing the same test plan. Therefore, the replication factor will vary and the number of nodes as well as the test plan will stay constant.

The system parameters are presented in Table 5.5 and the workload parameters which are defined in the test plan are shown in Table 5.6.

Parameters	Value
Leafset size	127
Replication factor	4, 16, 32
Number of nodes	
Test 1	10, 100, 1000
Test 2	200, 2000
Duration of experiment	141 minutes + set up time
OSGi command	`executeTimedTestPlan`

Table 5.5: System parameters

Plugin	Test case	Repetitions	Duration* (minutes)
Messaging	SendMessage	10	1
	DeleteInboxMessage	10	1
Live chat	SendMultiChatInvitation	5	1
	SendMultiChatMessage	10	1
	LeaveMultiChatMessage	5	1
Group	CreateGroup	5	1
	InviteFriendtoGroup	10	1
	ViewGroup	5	2
	ViewMyGroupList	5	2
File storage	CreateFolder	5	1
	UploadFileinFolder	10	1
Forum	CreateForumThread	10	2
	CommentForumThread	10	1
Photos	CreatePhotoAlbum	5	1
	UploadPhoto	10	1
	ViewFriendsAlbums	8	1
Voting	CreateVote	4	1
	AddPublicVote	4	1
	VotingInviteUser	4	1
	Vote	10	1
	GetVotingResults	10	1
Wall	SendMessage	10	1
	DeleteInboxMessage	10	1
* after each test case a pause of 5 minutes is implemented			

Table 5.6: Workload parameters

5.4.2 Experimental Results

In Figures 5.23 to 5.27 the total number of nodes through the duration of the tests is presented. It can be noticed in Figure 5.25 that the test with replication factor of 32 has failed. Furthermore, in 5.27 it can be seen that the number of nodes for replication factor of 32 is stable at 3500 nodes instead of 2000 nodes. We have looked at the log files for this test and all 2000 nodes seem to have finished the testplan successfully. The problem lies at the aggregation of the collected data. Unfortunately, circumstances did not make a re-run of the test possible

It is apparent from Figures 5.28 to 5.32 that with a higher RF more hop-counts will probably be

needed because the network is full of messages that concern the replication factor. Moreover, it seems that the peak hop-counts for the test with 10 nodes is higher in comparison to the 100 and 1000 nodes. When comparing only the tests with 100, 200, 1000 and 2000 nodes, the max hop-count (peak) is roughly the same at 0.7. This indicates that the routing table functionality is efficient. The reason that the hop-count never exceeds one hop is because all the nodes in the network are strong nodes and hence they all are added to the routing table size. Weak nodes (nodes with less computational power and memory storage, for example, mobile phones) usually do not appear in the routing table, but rather, they are present at only one routing table of a strong node. In this way, the routing tables will remain as short as possible to maximize the routing efficiency. This mechanism is not yet implemented in LibreSocial.

As for the total amount of replicas in the network, it can be observed from Figures 5.33 to 5.37 that as the number of nodes increases, from 10 to 1000 nodes and from 200 to 2000 nodes, the total amount of replicated data in the network increases as well. Additionally, it is apparent that as the RF increases the number of replicas increases as well. As far as the number of replicas concerned the network behaves exactly as expected.

When observing Figures 5.38 to 5.42 two important findings are apparent. The first finding is that with increasing number of nodes the storage time increases as well. When comparing the graphs for RF 16 in Figures 5.38, 5.39 and 5.40 the time to store a data item increases from approximately 0.11 seconds for 10 nodes to 2.2 seconds for 100 nodes and finally it reaches 16 seconds for 1000 nodes. The second finding is that with increasing replication factor the storage time increases as well. Even though the biggest spikes in Figures 5.38 to 5.40 are for replication factor of 16, it is noticed that the overall storage time throughout the test plan is higher for replication factor 32. A similar trend is also seen in Figures 5.41 and 5.42, in those figures the replication factor of 32 has the highest peaks because the number of nodes is significantly higher than in the previous set of Figures.

When looking at the retrieval time of data items that is shown in Figures 5.43 to 5.47. We expect the retrieval time to be higher for a lower replication factors because the less replicas of an item there are, the longer it will take to find it in the network. Figures 5.43, 5.45 and 5.47 are showing results as expected. But Figures 5.44 and 5.46 depict higher retrieval times for higher replication factors. The reason for this deviation could be that the replicas were stored in an inefficient manner for those tests, the routing table was not ideal or that for 100 and 200 nodes replication factor of 16 and 32 are not ideal.

In Figures 5.48 to 5.52 the rate for sending messages can be observed. The trend through all of the tests is that with a higher replication factor, more messages are sent through the network and hence the rate is higher. For 5.50 and 5.52 we do not have reliable results for a test with replication factor of 32, but when looking at the results of the tests with replication factor 4 and 16 for both scenarios a similar tendency is noticed, therefore we can safely assume that with replication factor 32 the rate will increase for both tests. Similar results can be observed in Figures 5.53 to 5.57 for the rate of received messages, hence the discussion about sent message rate is true in this case as well.

As we increase the replication factor we expect more space to be occupied by the replicas and as a consequence the total used space by LibreSocial will grow as well. The results that are presented in Figures 5.58 to 5.62 are exactly as expected. In Figure 5.59 the maximum used space for replication factor of 4, 16 and 32 is approximately 57 MB, 95 MB and 191 MB respectively (after conversion). When looking only at replication factor 16 and comparing the used space for an increasing number of nodes, we can see that the amount of used space increases as well. The used space is roughly 53 MB in figure 5.58 , 95 MB in Figure 5.59 and 114 MB in Figure 5.60. For the amount of generated data and the large workload the costs regarding used spaced for executing the test plan are extremely low.

In Figures 5.63 to 5.67 the maximum used memory in the network is shown. When the number of nodes in the network is low there is not much difference in memory usage between the three tested replication factor values. For the tests with 100 and 200 nodes the higher the replication factor, the higher the memory usage gets because many messages as well as save and retrieval commands are generated. When executing a test with a very large number of nodes as seen in Figure 5.65 and 5.67 it is interesting to observe that a lower replication factor of 4 had a drastically higher memory usage than for replication factor 16. The previously discussed explanation about retrieval times applies here as well. As seen in Figures 5.45 and 5.47 the retrieval times for replication factor 4 are higher as not many replicas are stored. Hence, many commands, such as retrieval commands, must be generated and stored in memory until the particular item is found. In this case, we can also witness memory costs of only up to 700 MB which are considered low by the standard of today's hardware.

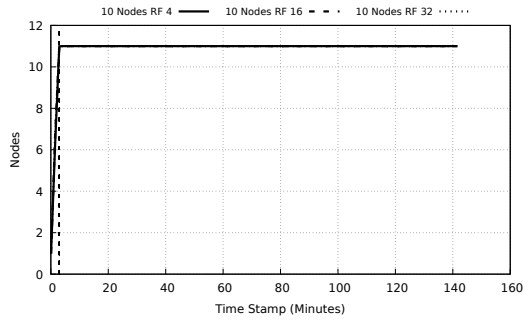


Figure 5.23: Number of nodes for 10 nodes test

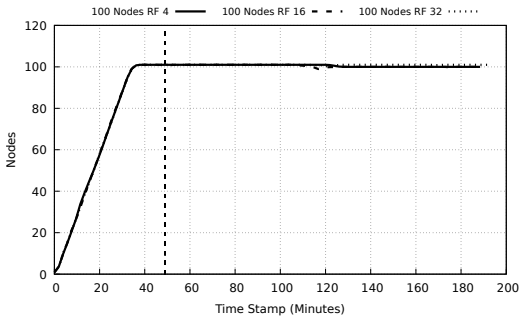


Figure 5.24: Number of nodes for 100 nodes test

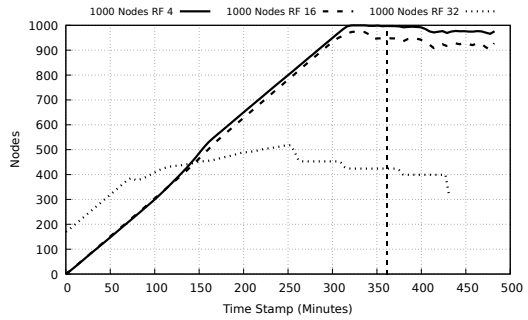


Figure 5.25: Number of nodes for 1000 nodes test

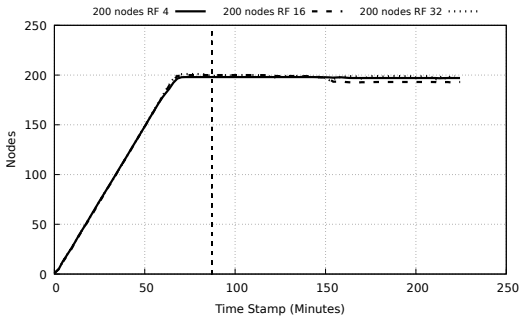


Figure 5.26: Number of nodes for 200 nodes test

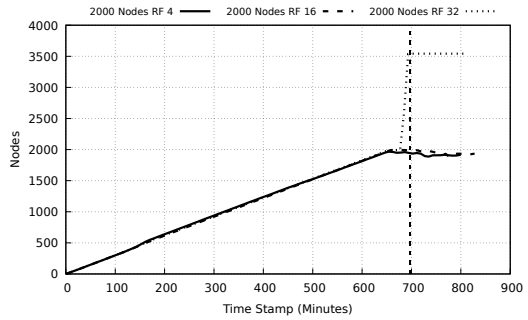


Figure 5.27: Number of nodes for 2000 nodes test

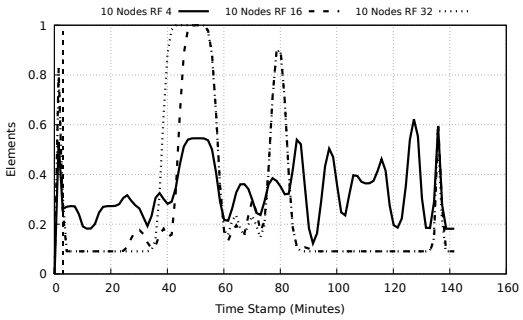


Figure 5.28: Hop count for 10 nodes

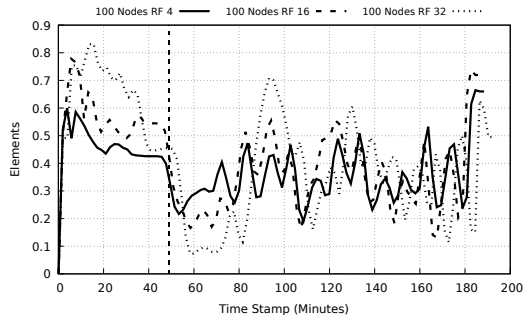


Figure 5.29: Hop count for 100

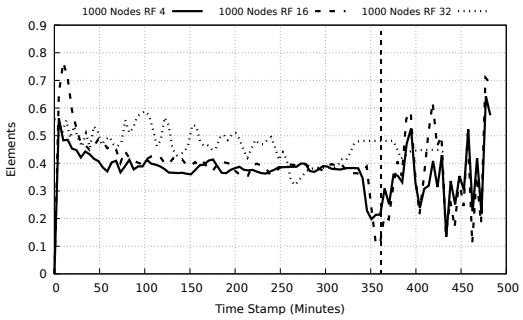


Figure 5.30: Hop count for 1000

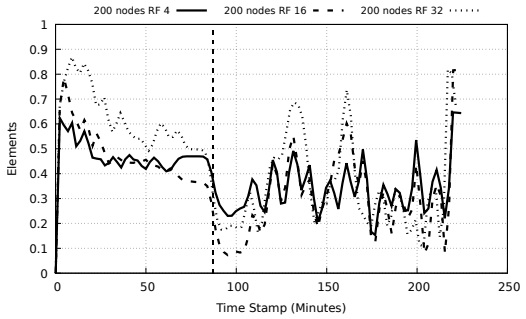


Figure 5.31: Hop count for 200

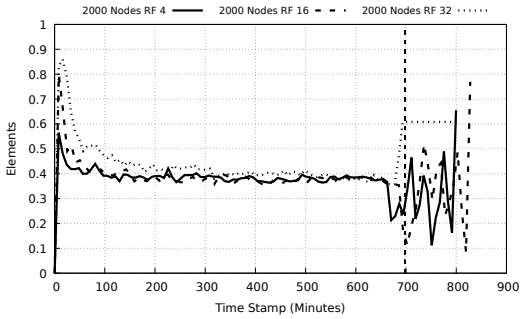


Figure 5.32: Hop count for 2000

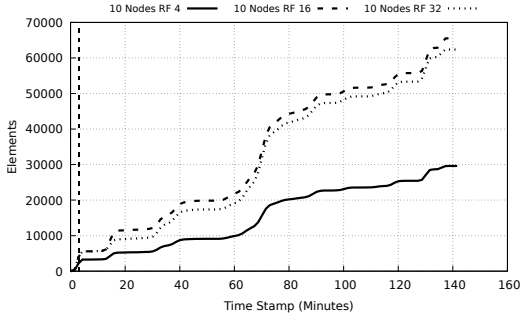


Figure 5.33: Total replicas in the network for 10 nodes

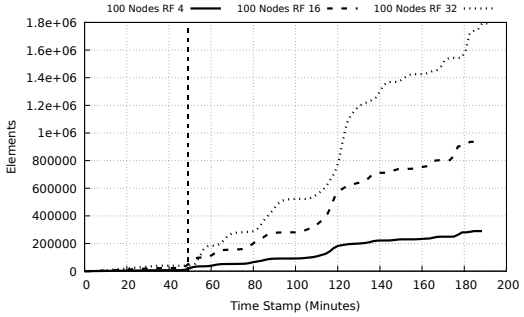


Figure 5.34: Total replicas in the network for 100 nodes

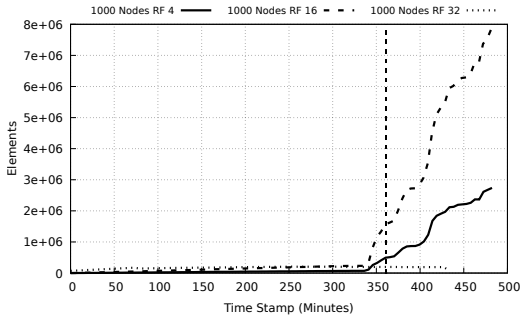


Figure 5.35: Total replicas in the network for 1000 nodes

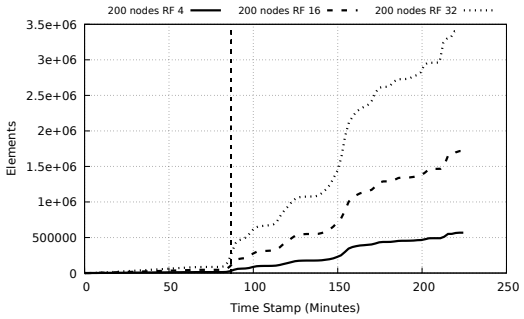


Figure 5.36: Total replicas in the network for 200 nodes

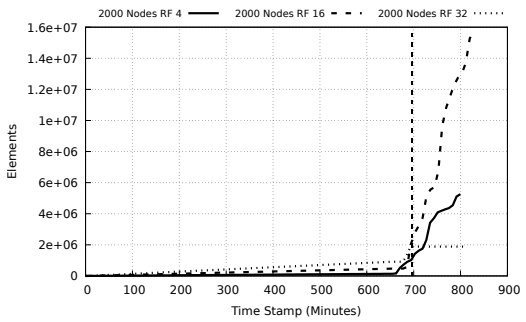


Figure 5.37: Total replicas in the network for 2000 nodes

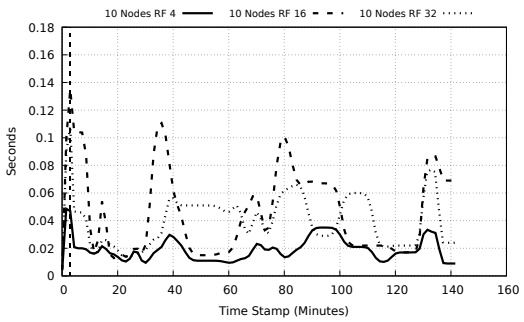


Figure 5.38: Maximum data storage time for 10 nodes

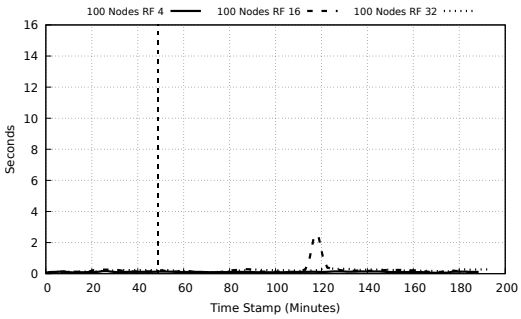


Figure 5.39: Maximum data storage time for 100 nodes

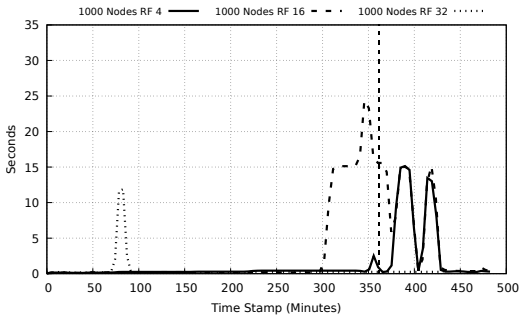


Figure 5.40: Maximum data storage time for 1000 nodes

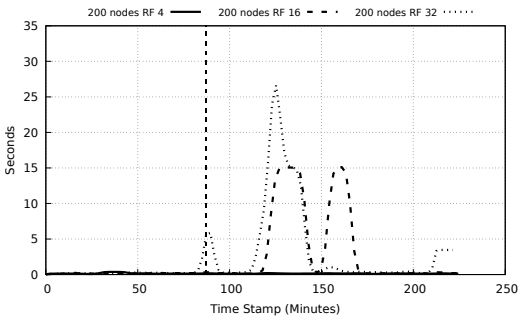


Figure 5.41: Maximum data storage time for 200 nodes

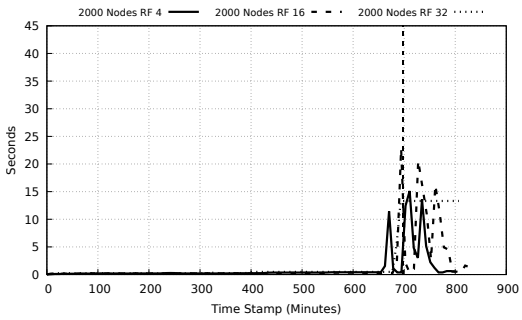


Figure 5.42: Maximum data storage time for 2000 nodes

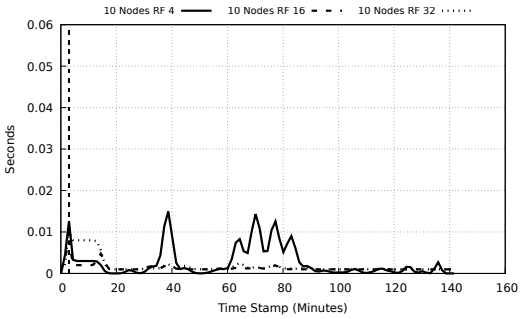


Figure 5.43: Maximum data retrieval time for 10 nodes

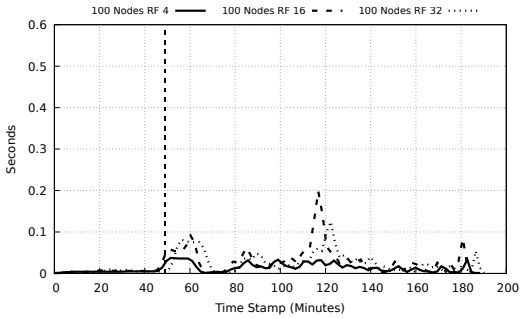


Figure 5.44: Maximum data retrieval time for 100 nodes

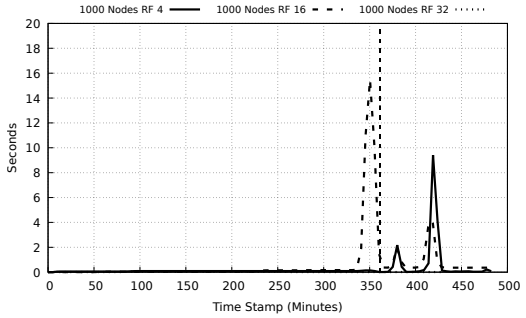


Figure 5.45: Maximum data retrieval time for 1000 nodes

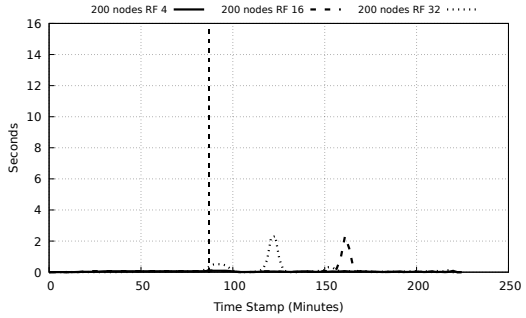


Figure 5.46: Maximum data retrieval time for 200 nodes

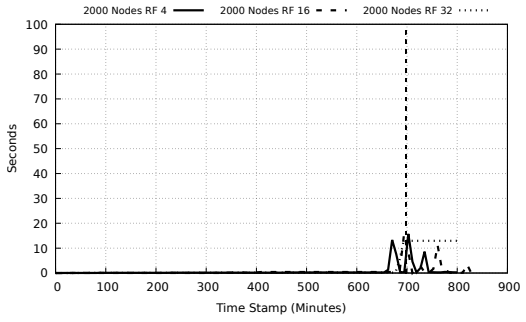


Figure 5.47: Maximum data retrieval time for 2000 nodes

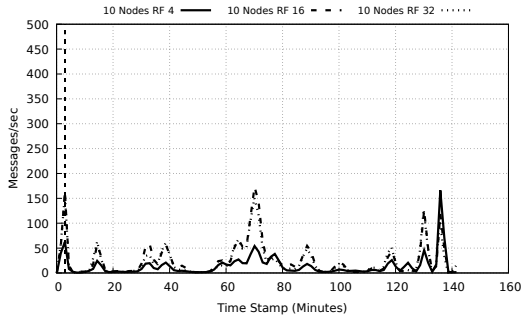


Figure 5.48: Message sending rate for 10 nodes

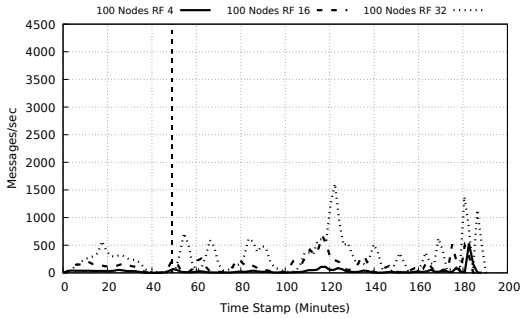


Figure 5.49: Message sending rate for 100 nodes

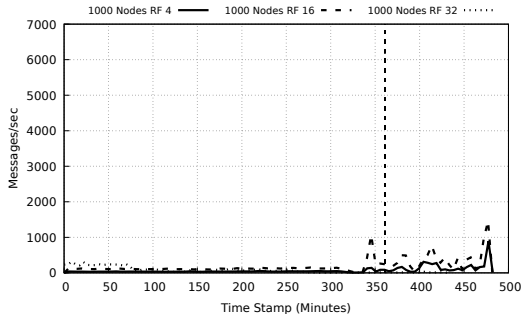


Figure 5.50: Message sending rate for 1000 nodes

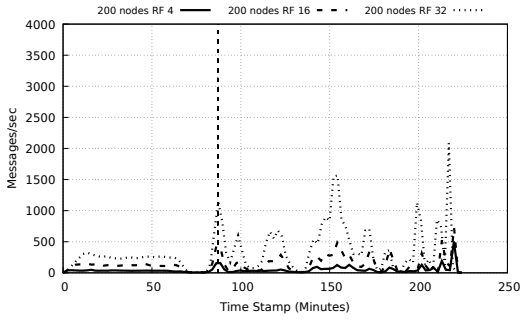


Figure 5.51: Message sending rate for 200 nodes

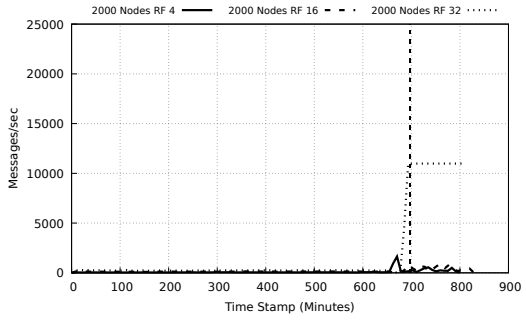


Figure 5.52: Message sending rate for 2000 nodes

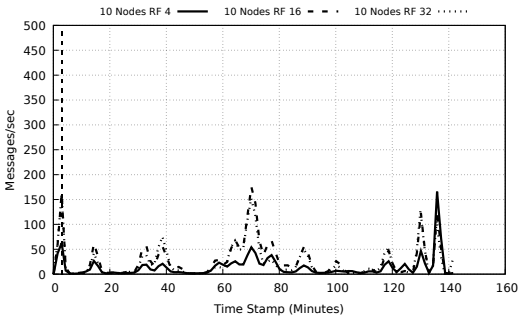


Figure 5.53: Message received rate for 10 nodes

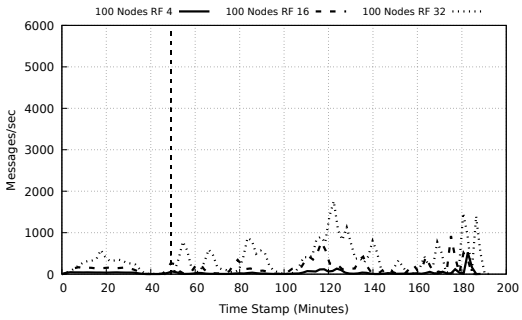


Figure 5.54: Message received rate for 100 nodes

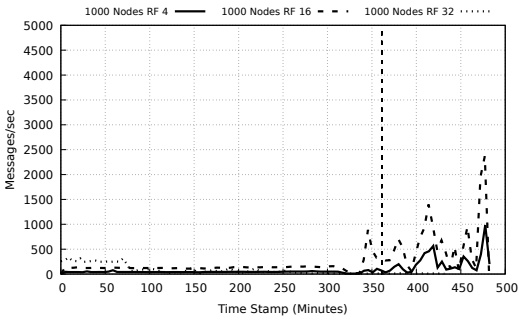


Figure 5.55: Message received rate for 1000 nodes

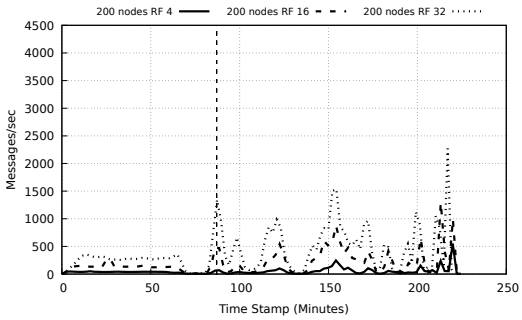


Figure 5.56: Message received rate for 200 nodes

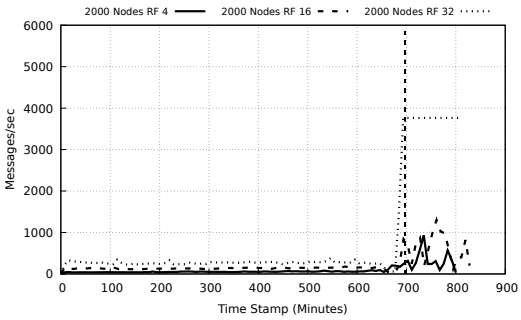


Figure 5.57: Message received rate for 2000 nodes

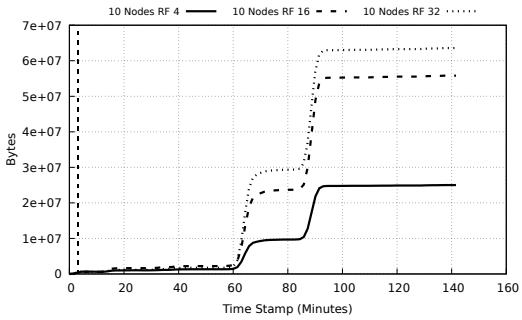


Figure 5.58: Used space for 10 nodes

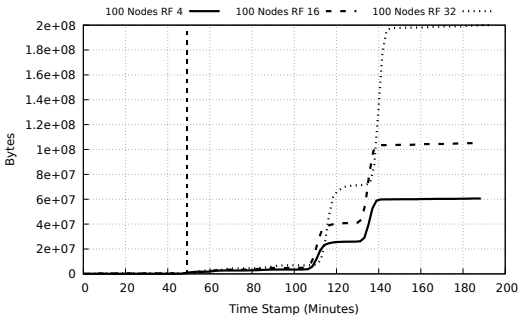


Figure 5.59: Used space for 100 nodes

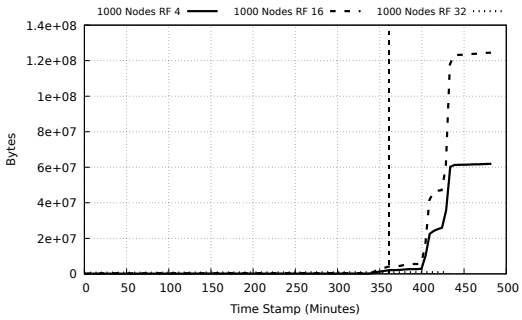


Figure 5.60: Used space for 1000 nodes

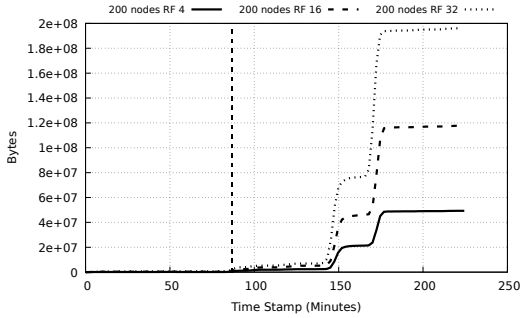


Figure 5.61: Used space for 200 nodes

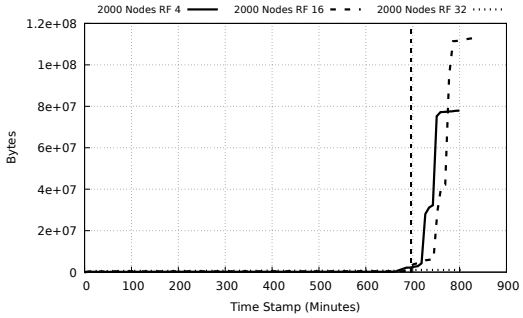


Figure 5.62: Used space for 2000 nodes

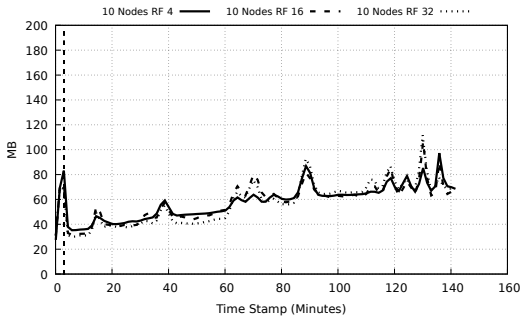


Figure 5.63: Memory usage for 10 nodes

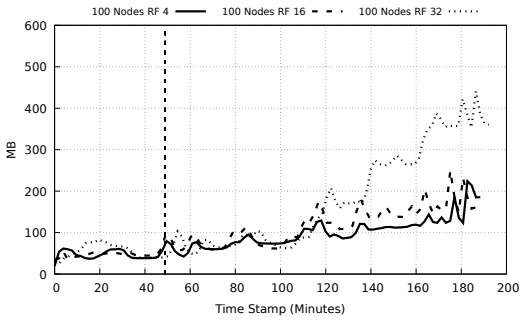


Figure 5.64: Memory usage for 100 nodes

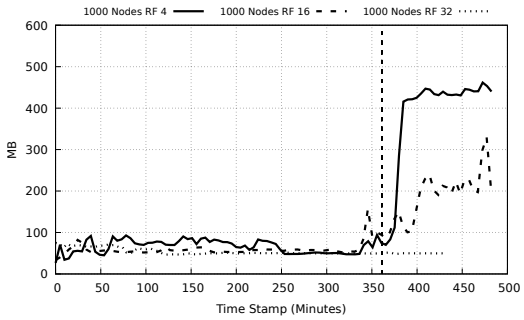


Figure 5.65: Memory usage for 1000 nodes

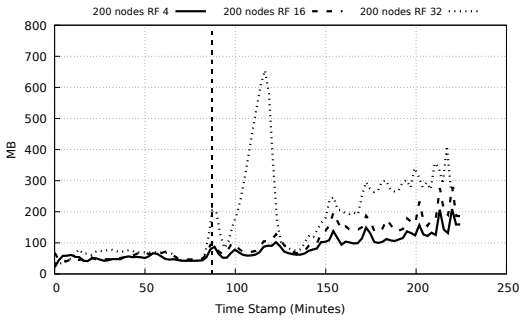


Figure 5.66: Memory usage for 200 nodes

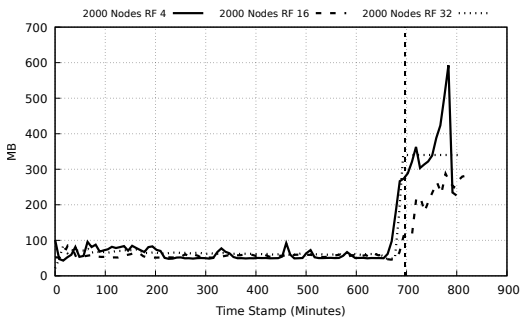


Figure 5.67: Memory usage for 2000 nodes

5.4.3 Result Summary

Table 5.7 summarizes the results for this tests scenario that are specifically relevant to the performance and cost quality aspects.

Nodes	RF	Performance					Cost	
		Hop count (nodes)	Max. storage time (s)	Max. retrieval time (s)	Max sent message rate (msg/s)	Max received message rate (msg/s)	Max. used space (MB)	Max. used memory (MB)
10	4	1	0.03	0.015	155	160	24	100
	16	1	0.11	< 0.01	160	175	52	110
	32	1	0.08	< 0.01	160	175	61	110
100	4	1	<1	0.01	500	500	57	220
	16	1	2	0.2	600	800	95	250
	32	1	<1	0.011	1500	1800	191	420
200	4	1	<1	<1	500	500	48	200
	16	1	15	2	700	1250	114	300
	32	1	26	2	2100	2250	191	650
1000	4	1	15	9	1000	1000	57	450
	16	1	16	4	1400	2500	119	310
2000	4	1	15	15	100	1000	76	600
	16	1	20	11	200	1200	113	300

Table 5.7: Results summary

After reviewing the results we feel that the replication factor has a big impact on the functionality of the network. When the network is relatively small, for example only 10 nodes, a large replication factor will produce too many replicas which overwhelm the network and are not always necessary. However, a very large replication factor has impacts on larger networks as well. We have experienced more inconsistent test runs with larger replication factor in contrast to virtually no problems at all executing tests with smaller replication factors.

From our evaluation of the results replication factor of 16 is an ideal value when considering a network up to 2000 nodes (for larger networks more tests should be conducted) because from the results it is evident that RF 16 is a good middle ground between relatively low costs (physical and computational memory consumption) and reasonably high performance

(throughput and responsiveness). We have also established that LibreSocial is scalable and can support up to as least 2000 concurrent logged in nodes.

5.5 Summary

In this thesis we have concentrated on benchmarking LibreSocial and therefore have taken a closer look at its monitoring and testing mechanisms. We have shown how the monitoring plugin operates and its importance to the testing plugin by enabling the collection of valuable metrics about the system during the tests.

We decided to design and carry out three different test scenarios that will emphasize various quality aspects. The pseudo-random behavior test scenario was intended to test LibreSocial in a broad manner with a random user behavior algorithm, inevitably we found out that a fault in randomization algorithm exists that leads to failures and crashes of nodes. After changing the `enableAutomaticLogout` parameter to `true` we managed to decrease the failure rate and obtained valuable results. Despite the failing nodes, we have managed to show that the leafset size and routing table size are dynamically adjusting to the size of the network to maintain efficient communication between nodes. Due to the unexpected failure of the nodes we have indirectly shown that LibreSocial is a reasonably robust application that can handle unexpected events like internal failures.

The main purpose of the stability test is to ascertain whether LibreSocial can support a large number of nodes and stay stable. To this point, LibreSocial was not tested with such a massive number of connected nodes at once, therefore we wanted to evaluate LibreSocial's scalability and stability. We logged all nodes into LibreSocial and performed several tests, next we logged a large amount of random nodes out to check how well LibreSocial handles churn. From this test we learned that LibreSocial is very stable and can easily be scaled up. The network can deliver messages in an efficient manner and it stays stable under large node fluctuations.

For the replication factor test we wanted to vary the replication factor and evaluate the performance and cost quality aspects. We have shown that an increasing replication factor value takes a toll on the storage times and on overall throughput such as the message rate. From our result evaluation we feel that a replication factor of 16 is ideal to the network sizes that we have tested, because it does not overwhelm the network, does not have high costs,

delivers the most consistent results and performs the best, in our opinion. We can say with confidence that LibreSocial's performance is acceptable and the costs are relatively low. In addition to the mentioned quality aspects, we have indirectly shown that LibreSocial can scale even more than in the stability test as we conducted test with up to 2000 nodes and the network remained stable and outperformed our expectations.

In conclusion, we feel that the conducted experiments were successful and we have received valuable results for this thesis work and for future work. For all the defined quality aspects of distributed systems that are relevant in this thesis to LibreSocial, we have shown that the application is stable and scalable with a good performance degree and fairly low computational costs.

Chapter 6

Conclusions

As presented in the introduction of this thesis, most social networks use a centralized architecture to store user's data. This architecture model was deemed to have too many security and privacy drawbacks on the user's side and distributed architectures solves most of those issues. LibreSocial is a fully distributed online social network that provides data security and has high quality service.

In this thesis, we aimed to evaluate several quality aspects of LibreSocial, in particular the stability, scalability, cost and performance. We started by fixing numerous test cases that were failing and proceeded to add a couple of metrics that were crucial to the evaluation of LibreSocial. Next, we designed three test scenarios and executed them on the university's high performance cluster.

From our experimental work and the gathered results we have seen that LibreSocial is scalable up to at least 2000 nodes and regardless of the number of connected nodes, LibreSocial remains operational in a very stable and reliable manner. We have tested LibreSocial's stability even further by increasing the replication factor and subjected the network to a considerable amount of exchanged messages and subjected the storage dispatcher to significant loads of data. For the tested number of nodes we have found a suitable replication factor value, for which the application was stable and had high performance. After summarizing the results we have further shown that LibreSocial has low computational costs while maintaining its high performance.

In summary, despite the challenges we have experienced that will be further discussed in

the next section, we can confidently conclude that in addition to LibreSocial being a secure distributed social network application, it is also stable, scalable, has high performance degree at low costs.

6.1 Challenges

During the work on this thesis several challenges came up. Some of them were overcame, but a few were out of our control. We want to briefly present those challenges.

Cluster Queue Time and Outage For our testing we have utilized the computational power of the university's high performance cluster, but we were not the only ones using the cluster and therefore job queue times must be taken into consideration. There was no problem executing tests with up to 200 nodes and they usually started right away. For larger tests with 1000 and more nodes we have experienced queue times of several days, sometimes even up to a week. Furthermore, the cluster undergoes constant maintenance during which it is not accessible for several days.

The COVID-19 Virus During the process of the experimental work, the COVID-19 virus outbreak occurred, which affected the progress of our work. Due to the rapid spread of the virus, new health regulations instructed people to stay at home and avoid public gathering. Hence, the experimental environment, the cluster, could not be accessed and the experimental work could not be conducted.

6.2 Future Work

We have presented the results that were achieved in this thesis and addressed the challenges that presented themselves. From our work we have noticed two key areas that must be addressed, those will be introduced in this section.

Firstly, as described in the result evaluation of the pseudo-random behavior test scenario,

the usage of the randomization algorithm led to some major failures and test crashes. The algorithms should be further analyzed in order to identify the source of the problem.

Secondly, when executing tests on the cluster we found out that tests would interfere with one another, because master nodes would recognize slave nodes from other tests and connect them to their own network. In those situations the master node sends a command to start a test plan or to terminate a test plan to all slave nodes, even to ones from other tests. Therefore, we had to execute a single test at a time and could not utilize the cluster to its full potential this caused a big setback. A new mechanism could be developed that established a unique connection between master and slave nodes. So in case that many different instances of LibreSocial run concurrently (for example, in the case of the high performance cluster) no foreign interferences can lead to a test failure.

If the first issue will be addressed, then LibreSocial could be better tested in a manner that represents real user behavior. If the second issue will be addressed, then the application will be all the more stable. The solution of both issues will enable a more comprehensive benchmarking of LibreSocial.

Bibliography

[AH00] Eytan Adar and Bernardo A Huberman. “Free riding on Gnutella”. In: *First monday* 5.10 (2000).

[All18] O Alliance. “Osgi service platform core specification, 2007”. In: *Más información en http://www. osgi. org* (2018).

[AV12] Kavitha Jagannath Pascal Kallien Jonathan Schlüßer Andreas Volk Daniel Breitlauch. *Project Group 2011: A Distributed Framework for Social Networks*. Tech. rep. Universität Paderborn, Apr. 2012.

[Ban08] Eric Bangeman. *Study: BitTorrent sees big growth, LimeWire still 1 P2P app*. 2008. URL: https://arstechnica.com/uncategorized/2008/04/study-bittorren-sees-big-growth-limewire-still-1-p2p-app/ (visited on 01/11/2020).

[Bar17] Kendall Bartsch. “The Napster moment: Access and innovation in academic publishing”. In: *Information Services and Use* 37 (Oct. 2017), pp. 343–348. DOI: 10.3233/ISU-170842.

[Ben+09] Fabrício Benevenuto et al. “Characterizing user behavior in online social networks”. In: *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*. 2009, pp. 49–62.

[BH03] Jean Bacon and Tim Harris. *Operating systems: concurrent and distributed software design*. Pearson Education, 2003.

[Cas+02] Miguel Castro et al. “SCRIBE: A large-scale and decentralized application-level multicast infrastructure”. In: *IEEE Journal on Selected Areas in communications* 20.8 (2002), pp. 1489–1499.

[DB14] Weifeng Peng Dheeraj Balabhadruni. “Systematic Evaluation of a Decentralized Social Network”. Master’s Thesis. Universität Paderborn, Dec. 2014.

[DR01] Peter Druschel and Antony IT Rowstron. “PAST: A large-scale, persistent peer-to-peer storage utility.” In: *HotOS*. Vol. 8. 2001, pp. 75–80.

[ESS13] W. Effelsberg, R. Steinmetz, and T. Strufe. *Benchmarking Peer-to-Peer Systems: Understanding Quality of Service in Large-Scale Distributed Systems*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013. ISBN: 9783642386732.

[GM20] Kalman Graffi and Newton Masinde. “LibreSocial: A Peer-to-Peer Framework for Online Social Networks”. In: (Jan. 2020).

[Gra+08] Kalman Graffi et al. “Skyeye. kom: An information management over-overlay for getting the oracle view on structured p2p systems”. In: *2008 14th IEEE International Conference on Parallel and Distributed Systems*. IEEE. 2008, pp. 279–286.

[Gra+10a] Kalman Graffi et al. “LifeSocial.KOM: A P2P-based platform for secure online social networks”. In: *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)* (Jan. 2010), pp. 1–2.

[Gra+10b] Kalman Graffi et al. “LifeSocial.KOM: A P2P-Based Platform for Secure Online Social Networks”. In: Sept. 2010, pp. 1 –2. DOI: 10.1109/P2P.2010.5569977.

[Gra+11] Kalman Graffi et al. “LifeSocial.KOM: A Secure and P2P-based Solution for Online Social Networks”. In: Feb. 2011, pp. 554 –558. DOI: 10.1109/CCNC.2011.5766541.

[GVVE08] Dorothy Graham, Erik Van Veenendaal, and Isabel Evans. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.

[Hoy09] Jeff Hoye. *Pastry-A substrate for peer-to-peer applications*. 2009. URL: https://www.freepastry.org/ (visited on 12/19/2019).

[Jor13] Paul C Jorgensen. *Software testing: a craftsman’s approach*. Auerbach Publications, 2013.

[Kal13] Pascal Kallien. “Systematic Evaluation of a Peer-to-Peer Framework in the Use Case of a Social Network”. Master’s Thesis. Universität Paderborn, Feb. 2013.

[Laf17] Robert Lafore. *Data structures and algorithms in Java*. Sams publishing, 2017.

[Lig12] S. Ligus. *Effective Monitoring and Alerting: For Web Operations*. O'Reilly Media, 2012. ISBN: 9781449333485. URL: https://books.google.de/books?id=cTqbdNzH7UUC.

[Liu13] Kangping Liu. "Benchmarking a Distributed Framework for Social Networks". Master's Thesis. Universität Paderborn, Oct. 2013.

[MG20] *Peer-to-Peer based Social Networks: A Comprehensive Survey*. Jan. 2020.

[PK15] Jack Peterson and Joseph Krug. "Augur: a decentralized, open-source platform for prediction markets". In: *arXiv preprint arXiv:1501.01042* (2015).

[RD01] Antony Rowstron and Peter Druschel. "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility". In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 2001, pp. 188–201.

[RD10] Rodrigo Rodrigues and Peter Druschel. "Peer-to-peer systems". In: *Communications of the ACM* 53.10 (2010), pp. 72–82.

[SW05] Ralf Steinmetz and Klaus Wehrle. *Peer-to-peer systems and applications*. Vol. 3485. Springer, 2005.

[VST17] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 19. März 2020 Liat Khitman



Diese CD enthält:

- eine *pdf*-Version der vorliegenden Bachelor's Thesis
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelor's Thesis samt aller verwendeten Skripte
- **[anpassen]** die Quelldateien der im Rahmen der Bachelor's Thesis erstellten Software XYZ
- **[anpassen]** den zur Auswertung verwendeten Datensatz
- die Websites der verwendeten Internetquellen