

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik

Masterthesis

Benchmarking a Distributed Framework for Social Networks

Kangping Liu
Student ID: 6672590
E-Mail: kangping@uni-paderborn.de

Paderborn, October 27, 2013

supervised by
Jun.-Prof. Dr.-Ing. Kalman Graffi
Prof. Dr. Christian Scheideler

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt 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.

Paderborn, den October 27, 2013

Kangping Liu

Contents

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Testing P2P Framework . . . 2
1.3 Overview on Goals . . . 3
1.3.1 Complete Monitoring . . . 3
1.3.2 Synthesis Test . . . 3
1.3.3 Random Test . . . 3
1.3.4 Random Test without Churn . . . 3
1.3.5 Long Term Test . . . 3
1.3.6 Alpha Test . . . 4
1.3.7 Evaluation of Test Results . . . 4
1.4 Outline . . . 4

2 Background **5**

2.1 Peer-to-peer Network . . . 6
2.1.1 Unstructured P2P Overlay . . . 6
2.1.2 Structured P2P Overlay . . . 6
2.1.3 FreePastry . . . 7
2.2 P2PFramework . . . 8
2.3 Plugin and Apps . . . 8
2.4 Terms . . . 8

3 Testing **11**

3.1 Goal . . . 11
3.2 Metrics . . . 12
3.2.1 New Metrics . . . 13
3.3 Setup Parameters . . . 15
3.3.1 Test Environment Parameters . . . 16
3.3.2 Social Parameters . . . 17
3.3.3 LifeSocial Parameters . . . 17
3.4 Test Case . . . 22

4 Monitoring **31**
4.1 Monitoring . . . 31
4.2 SkyEye.KOM . . . 31
4.3 Problem of Monitoring Mechanism . . . 34
4.4 Improvements of Monitoring Mechanism . . . 38

5 Test Setup **45**
5.1 Test Environment . . . 45
5.2 A Standard Test Process . . . 48
5.2.1 Runtime Environment of LifeSocial . . . 48
5.2.2 Deployment . . . 48
5.2.3 Testplan . . . 49
5.2.4 Monitoring Data Processing . . . 49
5.3 Synthesis Test . . . 50
5.3.1 Deployment . . . 51
5.3.2 Testplan . . . 53
5.3.3 Monitoring Data Processing . . . 54
5.4 Random Test . . . 55
5.4.1 Deployment . . . 55
5.4.2 Testplan . . . 55
5.4.3 Monitoring Data Processing . . . 56
5.5 Long Term Test . . . 56
5.5.1 Deployment . . . 56
5.5.2 Testplan . . . 58
5.5.3 Monitoring Data Processing . . . 59
5.6 Alpha Test . . . 62
5.6.1 Deployment . . . 62
5.6.2 Testplan . . . 62

6 Results Evaluation **63**
6.1 Synthesis Test - 25 instances . . . 63
6.1.1 Test Results . . . 63
6.1.2 Conclusion . . . 73
6.2 Synthesis Test - 100 instances . . . 74
6.2.1 Test Results . . . 74
6.2.2 Conclusion . . . 81
6.3 Random Test - 25 and 100 instances . . . 81
6.3.1 Test Results . . . 82
6.3.2 Conclusion . . . 87
6.4 Random Test without Logout - 25 and 100 instance . . . 87
6.4.1 Test Results . . . 87
6.4.2 Conclusion . . . 92

6.5 Long Term Test - 10 and 5 instances 92
6.5.1 Test Result 92
6.5.2 Conclusion 95
6.6 Alpha Test 96
6.6.1 Problem Documentation 96
6.6.2 Conclusion 96

7 Modifications and Extensions **97**
7.1 Memory leak 97
7.2 OSGI commands 97
7.3 Alpha test 97

8 Conclusion **99**
8.1 Achievements 99
8.2 Future Enhancements 100

List of Figures **103**

Bibliography **193**

1 Introduction

1.1 Motivation

Peer-to-peer (p2p) networks have earned itself a significant place nowadays. There are uncountable numbers of p2p users all over the world. The main reason that it is getting more and more popular is that, first, the more users in p2p network the more powerful the network will be; second, it endows users the ability to establish their own network and brings more freedom.

In a p2p network, every individual peer consumes and contributes to the network in the same time. Each peer can be a user and these contributed and consumed resources can be network bandwidth, storage or processing power. In contrary to a centralized network, the loads of p2p network are distributed among all the peers, it makes such network have no bottleneck. Also due to its decentralized characteristic, all the data in the network are distributively stored among peers, which brings it another strong point in terms of security, that it has no single point of failure. Furthermore, in a structured p2p overlay, the distributed hashtable based routing enables users to search for files efficiently, even if the searched file is extremely rare.

However every stick has two ends, due to lack of central censorship, p2p technique also has earned itself a bad reputation in copyright infringement and spreading pornographic contents. One well know lawsuit related to p2p infringement is Napster case. But even though Napster lost the case finally, it still gave rise to the trend of p2p technique. A lot of useful and popular p2p based software emerge, such as Skype, and it often acts as auxiliary to help some central server based technique.

In the meanwhile social network has become a worldwide favourite. Many famous Social Network Services (SNS), such as Facebook, Twitter etc., play an essential role in people's daily life now. Such SNSs represent each user in the network by his personal pages where user can share their identity information, interests or even moods with their friends. However since most these services are web-based server/client structure, they are facing many problems. According to statistics, up to last year Facebook has over one billion active users and has about 180 petabytes of data a year and grows by over half a petabyte every 24 hours [2]. It might be currently affordable for Facebook's servers, but

according to this half petabyte per day growing speed, it will definitely become a severe problem in the future. In addition, the complaints towards the privacy policy of Facebook never stop, because Facebook tracks users' data and the account cannot be deleted once created.

P2P technique can be a solution to such problems. Since the power of p2p network is proportional to the number of users, with this over one billion users and the growing computing speed of personal computers, the capacity of the network will be tremendous. Moreover, there is no central server in p2p network which means users store the data by their own devices. With a commonly reached consensus in privacy policy, users will not be worried about their personal data being peeped.

LifeSocial is a p2p-based social network which is developed in University of Paderborn for years. It is a framework built on the FreePastry p2p overlay, and the application on the top of it is s social network. New features and functions are added to LifeSocial every year. Before put it in the real industry, LifeSocial needs to be exhaustively tested. The goal of this master thesis is to complete the monitoring function and run three sequential tests, synthesis test, long term test and alpha test.

1.2 Testing P2P Framework

During the development of LifeSocial, each new implementation will be tested by JUnit test and small scale test. JUnit test is a test run in a simulated environment where the number of nodes, the behaviours of nodes can be preset. The whole test is run in an individual machine. A small scale test includes several machines, each machine acts as one or two nodes, developers conducts the machines to perform certain behaviours to test whether they are functional. Both these tests are done during the development, but they lack of getting the insight of the behaviour of p2p network during a long term run. That is why a large scale test is necessary for testing.

A large scale test can be set among machines or real users. The essential elements in such test are : test cases, testplan and monitoring. Test case is the concrete action of each peer/user, such as upload a photo. TestPlan is how the sequence of test cases are ordered, and a random testplan is also necessary. Monitoring records different metrics of the network and saves the statistics into files which can be plotted afterwards. In a real user large scale test, the sequence of test cases is always random since the behaviour of users cannot be controlled.

1.3 Overview on Goals

The goal of this master thesis is to benchmark a distributed framework for social networks. To reach this goal the following sub-goals should be achieve.

1.3.1 Complete Monitoring

The monitoring function of LifeSocial is already implemented, however it still has some shortages in different aspects now. Basically two parts need to be completed in monitoring, the first one is monitoring metrics. There are already many metrics monitored now, but they are not enough, new and valuable metrics are required. The second one is monitoring mechanism, since the monitoring tool used in LifeSocial is called SkyEye.KOM [5]. It builds a new tree structure overly on top of the LifeSocial and it communicates by LifeSocial's message channel which brings a problem that, if the message channel of LifeSocial is busy, the monitoring messages will be delayed and the gathered statistics are not accurate.

1.3.2 Synthesis Test

The first test to be carried on is synthesis test, all the nodes in the network start executing the same testplan at the same time. This test is quite helpful in finding whether the modules of LifeSocial are functioning correctly and testing the robustness of the distributed framework under intensive test cases.

1.3.3 Random Test

Random test is as same as Synthesis test except the sequence of test cases in testplan is totally random. This test aims at imitating real user behaviours so that nodes will come in and out network, so called "churn". The performance under churn is one of an important criterion of judging a p2p network.

1.3.4 Random Test without Churn

The only thing this test is different from the random test described above is that, there is no "churn" in this test. All nodes in this network perform test cases randomly but not leave the network.

1.3.5 Long Term Test

Maintaining a network within hours sometimes is not enough to prove it can withstand a long term run. The long term test is an extended synthesis test, the difference is it will be run for days. The testplan in a long term test will

not be as intensive as synthesis test, since the goal is that the network can be maintained for a long time.

1.3.6 Alpha Test

The last and the only involving real user test is alpha test. Alpha test is the first time the let real users contact the product. It is not expected that no error and bugs will appear during this test, on the contrary, the aim of this test is to find and correct the development direction of the product. It is also a prepare work for the beta test, since once users get familiar with the product in alpha test, they know what to expect in the beta test.

1.3.7 Evaluation of Test Results

After having run all these tests and gathered the test results, the statistics will be analyzed and evaluated.

1.4 Outline

In the following sections, first a introduction to the background of p2p network and LifeSocial is in Section 2. After that the goal and related content of testing is introduced in Section 3. The shortage and improvements of monitoring mechanism of LifeSocial is introduced in 4 The setups of these tests are introduced in section 5. Result evaluation is in Section 6. Some modifications and extensions of LifeSocial is documented in Section 7. The conclusion is drawn in Section 8.

2 Background

This chapter includes all the fundamentals of this master thesis. In section 2.1 first all the basics about p2p network are introduced, then FreePastry is specially explained in detail, since it is the overlay of LifeSocial. Section 2.2 describes the P2P-Framework which is built on FreePastry overlay. With these knowledges Section 2.3 presents how the plugins and apps work on this P2P-Framework. Finally in Section 2.4 the concept and terms used in LifeSocial are introduced. Figure 2.1 shows the structure of LifeSocial.

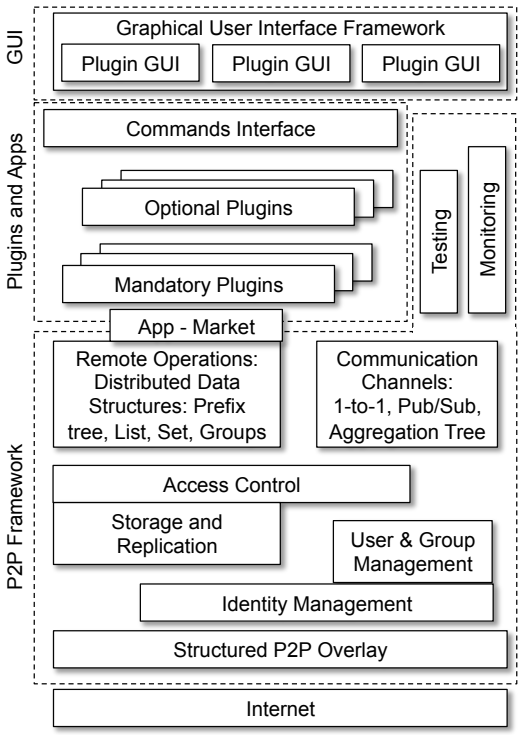


Figure 2.1: Structure of LifeSocial

2.1 Peer-to-peer Network

As the name implies, peer-to-peer (p2p) simply means two nodes communicate with each other without central server, each peer acts as both client and server. When multiple nodes within a network are simultaneously functioning together without any server, it is called a p2p network. From a structural point of view, P2P network can be divided into two categories: structured and unstructured. In structured overlays data items are assigned to specific nodes, thus routing is possible. In unstructured networks, there is no information, where content might be, routing is not possible. Since the most difficult problem of p2p network is that, there is no central server to provide each peer a global view of the network, each node has no clue where other nodes are. In these two different structures, different routing mechanisms are used. Here the structure means the topology imposed on the network, not the physical connections between peers.

2.1.1 Unstructured P2P Overlay

In an unstructured p2p overlay, all peers are randomly connected. The advantage of unstructured p2p overlay is because it is easy to be built. Any outside node is able to join the network by connecting to an existing node inside the network in advance. Due to each node only has to keep the connections to its neighbours, the network is able to handle very high rates of c̈hurnẅhich means nodes frequently join and leave network. The lack of mapping the nodes to data files leads to that, in a homogeneous unstructured p2p overlay, the most common routing protocol is flooding search. Because there is no information about the content, the query message for the content has to be sent as many as possible, namely f̈looding search.̈ Obviously it is not a scale approach which causes a very high load on the network. Moreover, the popular content can be quickly retrieved in unstructured p2p overlay, but it does not guarantee rare content the same story. Even if the rare content is retrieved, the query might have been flooded all over the network.
Besides the homogeneous unstructured p2p overlay, such as Freenet there are also centralized and heterogeneous p2p overlay, such as Napster and eDonkey respectively. Also there are improved routing protocols based flooding search, such as expanding ring or random walk. But none these overlays and routing protocols is not good enough at routing efficiently.

2.1.2 Structured P2P Overlay

Structured p2p overlay impose a specified topology on the network. Peers are connected according to the designed protocol. Structured p2p overlay guaran-

tees efficient routing to the queried data, including very rare data items. Most structured p2p overlay implements the distributed hash table (DHT). A DHT maps nodes and data into (key, value) pairs, any value can be efficiently search by the key.

Distributed hash table Hash table maps each data item into a key, sometimes maps the IP of the responsible node into the key. Distributed hash table means the (key, value) pairs are distributively stored among all the nodes in the network. Thus the variation of one node will not cause a big impact on the whole network. In Figure 2.2 a simple DHT is illustrated, three data items are hashed into three different values respective, and then stored in three different nodes in the network. If *node a* wants to retrieve *DATA1*, it sends out a query for the key *D9EC704*, according to the routing protocol the query will be forwarded to *node b*, then *node c* will answer back directly.

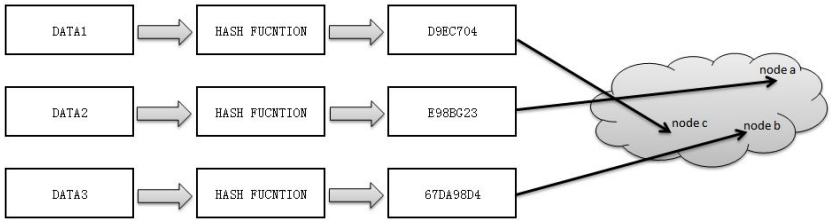


Figure 2.2: Distributed Hash Table

2.1.3 FreePastry

FreePastry [1], a structured p2p overlay, which is developed by Rice university, is explained as an example here. It a java-based prototypical implementation of Pastry. Each node in FreePastry has an unique, uniform and random identifier. Both nodeID and objectID are mapped into a 128-bit key space. A query for a certain key will be routing to the node whose ID is numerically closest. The routing mechanism in FreePastry combines DHT and prefix-routing. It requires

each node, besides routing table, also holding a leaf set. Leaf set contains the nodes whose IDs are numerically closest. Based on the routing function provided by FreePastry, there are two main applications implemented on top of that : PAST and SCRIBE. PAST provides a archive storage and SCRIBE provides group communication/event notification. Both of them are generic, scale and efficient. These two applications are most often used by the p2pframework on top of it.

2.2 P2PFramework

P2P-Framework uses functions provided by overlay, moreover it categorizes and features these functions and provide them to the upper layer. In Figure 2.1 overlay is also counted into P2P-Framework since it is a p2p overlay. Except this overlay, framework can be divided into several modules as shown in the picture. Main functions of this framework can be categorized into : storage and communication as the basic mandatory functions of a network. Monitoring, Testing and App-Market are all developed as a plugin of LifeSocial, but they also implement there own required functions in framework.

2.3 Plugin and Apps

Plugins are basic elements which invokes methods provided by framework, at meanwhile provide logic functions to its own GUI. These plugins provide all services LifeSocial to users. As a social network, these services can be categorized into : mandatory and optional concepts. For instance chat with friends, send friend request to friends, upload photo, these are mandatory plugins. Besides that, new features can be developed as optional plugins, such as establish a forum, voting, these services make LifeSocial colourful. App-Market provides an API to develop and customized users' own plugins.

2.4 Terms

DDS DDS is short for distributed data structure, it is special data structure which is provided by p2pframework of LifeSocial. It is useful for many applications, such as hierarchical file system where a lot of data has to be stored. If all files were stored in one single node, it means the node has to deal with a lot of request and store huge amount of data which increases the risk of single point of failure. DDS helps to store the files among different nodes and the request sent to the nodes are balanced.

OSGI LifeSocial makes uses of Framework. It is based on Eclipse Equinox which implements the OSGI specification. It is consisted of a series of bundles, namely "plugin" afterwards. The basic functional logics are provided by these plugins. The code of the user interface are packed as bundles, it results in that the code of user interface and functional logics are separated.

3 Testing

The main theme of this chapter is to describe the testing in detail. For each different testing first the it needs a goal, the well designed procedure, the set up and predefined metrics etc. With all these well prepared work, the result of different tests are deserved to be evaluated.
Section 3.1 defines goal of testing and the motivation of each kind of test. Section 3.2 describes the metrics monitored in all testings and new added metrics in this master thesis. Section 3.4 introduces the test cases including the original ones and new added ones.

3.1 Goal

The ultimate goal of testing is to put LifeSocial under different environment to observe the performance of users, p2pframework and FreePastry overlay. In synthesis test, it creates an environment that all nodes synthetically performs same test case at the same time. It is out of touch to a real network, but it gives the least pressure and the goal of this test is to check the correctness of each function of LifeSocial.
Random test creates churn in the network so that nodes will join and leave network. The environment of network is more like a real user network and it puts a fairly high pressure on LifeSocial, since data objects stored in the network might by lost, nodes requested might be offline. The goal of this test is to examine how well LifeSocial performs under the high pressure.
Random test without churn is to more like a compromised random test, in this test no churn is concerned. In this test all nodes perform test case randomly, it is like a real network that all users are online. By comparing the result of this test and random test, it can be found whether LifeSocial is able to handle churn well.
Long term test is to check the performance of LifeSocial in a long term run. In a long term run the software may behaves abnormally compared to a run of several hours, memory may increase and queue of message channel may be full. This test will give the result that show how LifeSocial perform after a long time run.

Alpha test is the first time to put a software in a real user group. Usually in alpha test user group is a special testing team of the organization of this software. During this test all kinds of bugs and problem reports are welcomed, since the aim of this test is to find the problems and bugs that not being considered and found during development. It is also a preparation work for the beta test which will put the software in a real user group, instead of a testing team. It is the first time to put LifeSocial in a alpha test.

3.2 Metrics

Metric represents the attribute of a parameter at each time step. It can be collected every where inside LifeSocial. With the help of SkyEye monitoring tool, the global metric of the network is able to be generated according to each node's local metric in the runtime. The attribute of one metric contains four basic elements : name, aggregation strategy, value and unit.

Name describes the characteristic of the measured parameter, for instance it can be looked like :

$$STORAGEDISPATCHER_FRAMEWORK_DATA_SENT\ .$$

STORAGEDISPATCHER indicates in which java class the metric is collected and *FRAMEWORK_DATA_SENT* indicates the content of the metric. Similarly there will be also *STORAGEDISPATCHER_FRAMEWORK_DATA_RECEIVED* which stands for the framework data received.

Aggregation strategy tells SkyEye in which way the metric is aggregated. The most used strategies are : *sum*, *mean*, *minimum* and *maximum*. For instance if *STORAGEDISPATCHER_FRAMEWORK_DATA_SENT_min* has been collected from all the nodes in the network and the aggregation strategy is *minimum*, then the smallest one of all the values will be chosen as the value of this metric at this time step.

Value is as the name implies, it can be the sum of data from beginning to now or the value at this very point. In addition, value also can be the data rate of a certain metric. LifeSocial contains a $RateBuffer$ function especially for calculating data rate for a certain metric. The theory of $RateBuffer$ is show in Figure 3.1 and Figure 3.2, first the data rate $newRate$ after last collect time is calculated, then $newRate$ will be put in a rate buffer. This buffer only stores the latest X values and X can be customized. The final value of the data rate of a certain metric is the average value of the rate buffer. This theory makes sure

that the result graph will be smooth if the rate buffer has a large size. And it is still able to generate the actual value of the data rate if the buffer size is set to 2.

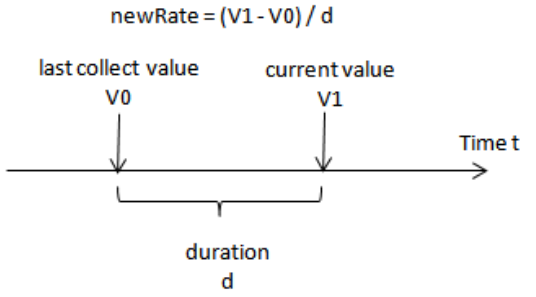


Figure 3.1: The value in this graph is the sum of a certain metric. New rate is the data rate after last collect time.

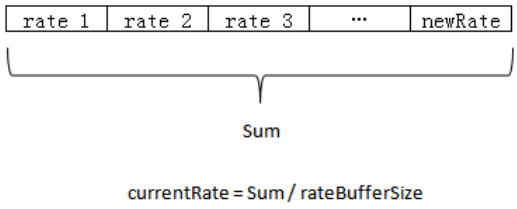


Figure 3.2: Current rate is the final data rate which will be assigned to the metric as the value.

Unit of the metrics is fully customized. It depends on the property of the measured metric.

3.2.1 New Metrics

The original metrics have covered most aspects of LifeSocial, however they do not provide a view of the data flow which starts from plugin layer to FreeP-astry layer. As shown in Figure 3.3, the data flow starting from plugin layer will go through information cache, DDS or directly framework. Moreover the social metrics, such as number of friend or album size are not measured in the runtime.

Data Flow

The most invoked functions which provided by the below layers are : store and retrieve data functions; send and received message functions. Usually most plugins will directly invoke these functions from framework layer, so the new metrics as *FRAMEWORK_SENT/RECEIVED_PLUGINNAME* can be easily added from framework layer. However the data flow through these two parts are too complicated to be sorted out. So the data through these two parts are measured as *FRAMEWORK_SENT/RECEIVED_DDS/INFORMATIONCACHE* and these metrics will stand for the data flow from plugins above these two parts.
In the framework layer, the sums of all the data flows coming from above and below layers are measured as *STORAGE_DATA_SENT/RECEIVED* and

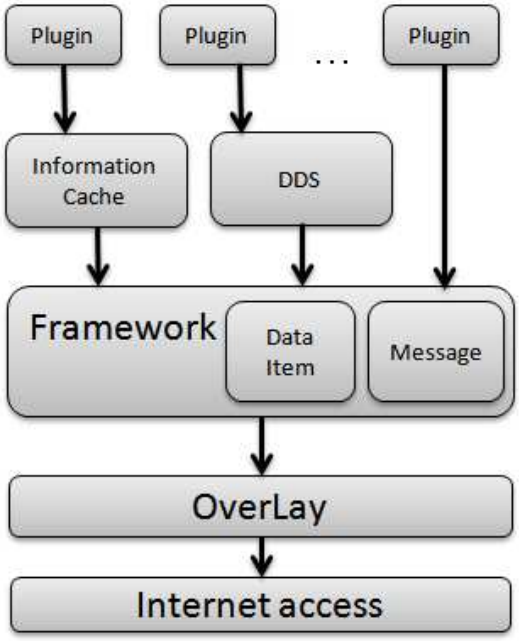


Figure 3.3: Data flow through LifeSocial.

MESSAGE_DATA_SENT/RECEIVED respectively. The combination of these two metrics *FRAMEWORK_DATA_SENT/RECEIVED* is also measured. It is supposed to repeat everything again in the FreePastry's PAST layer or Scribe layer, however, the result of experiments shows that the calculation of the size of data items and messages requires time and PAST or Scribe are very sensitive to the extra delays being introduced by third parties. In these experiments many nodes are not able to join the network due to the time delays caused by object size measuring. Therefore the data flows are not classified any more in the FreePastry layer. The combination of framework data flows and data flows generated by FreePastry are calculated as *PASTRY_DATA_SENT/RECEIVED*.
All the new metrics described above, their minimum, mean and maximum value and data rate are both measured too. Table 3.1 shows all the new added metrics for measuring data flow.

Social Metric

As a social network software, it is also necessary and interesting to monitor the social metrics in LifeSocial. These metrics will be useful for analyzing user behaviour in a social network. The social metrics here mainly measured how many times the user performs a certain action within a time period, for instance the sent friend request or created photo album. The global value of these

metric name	min,mean and max
FRAMEWORK_SENT/RECEIVED_PLUGINNAME	YES
FRAMEWORK_SENT/RECEIVED_DDS	YES
FRAMEWORK_SENT/RECEIVED_INFORMATIONCACHE	YES
STORAGE_DATA_SENT/RECEIVED	YES
MESSAGE_DATA_SENT/RECEIVED	YES
FRAMEWORK_DATA_SENT/RECEIVED	YES
PASTRY_DATA_SENT/RECEIVED	YES

Table 3.1: Table of new metrics for measuring data flow

plugin name	monitored metric
FileStorage	get file; store file
Forum	create post; create thread
Friend	accept friendship request; send friendship request; number of friends
Group	create folder; create private group; create public group; delete group invite user to group; join group; leave group
LiveChat	send chat message
Login	number of login; number of logout
Photo	add photo; create new album ; delete album; load photo
Profile	create basic profile; get own profile; request user profile
Search	number of search
Voting	create voting
Wall	add wall post; add wall comment; retrieve wall entries

Table 3.2: Monitored social metrics in different plugins

metrics are also generated by SkyEye, since a global view of these metrics provides a deep insight of the network. Table 3.2 lists all the metrics which are monitored in different plugins.

3.3 Setup Parameters

There are many parameters which can be configured for each test. Most parameters do not need to be changed in different tests, the other parameters has to be changed in corresponding to different test requirements. In previous master thesis all the parameters are divided into three categories: `Test Environment`, `Social Network` and `FreePastry Parameters`. There are also many controllable parameters in LifeSocial, most of them work like *trigger* of a certain function. Some functions are needed in a certain test and some are not. Some of following parts are referenced from a previous master thesis which is

done by Pascal [6].

3.3.1 Test Environment Parameters

Test environment parameters adjusts configuration of the infrastructure of a test.

Number of machine This parameter controls how many machines are used in the test. It is closer to a real P2P network if more machines are involved in. The accessible machines are limited to 20. Thus the range of this parameter is from 1 to 20.

Number of instance This parameter controls how many instances are involved in a test. There are around 50 machines available inside university, however the number of machines which have been configured in a public/private key system are 20 and it sometimes causes accessibility problems if machines outside this 20 is used. Each machine is able to initiate multiple instances. Moreover instances also can be initiated on the virtual machines. 10 virtual machines inside Uni-Paderborn is reserved for this master thesis.
However if too many instances are initiated in the same machine or webserver, it will decrease the degree of realism of the test, but indeed the more instances the closer to a real P2P network. Thus the choice of this parameter is a trade-off.

Operating system The LifeSocial is developed in Linux system, thus it runs most stable in it. In a real user network, the operating system of each user is heterogeneous. However the machines pool or the webservers in Uni-Paderborn are all installed Opensuse operating system which is one of Linux systems.

Machine performance The performance of machines in a real user network is heterogeneous. In the machines pool or the webservers in Uni-Paderborn it is homogeneous.

Peer bandwidth The machines can be connected via wireless LAN, Gigabit Ethernet or UMTS. This parameter is also can be homogeneous of heterogeneous. All the test machines in Uni-Paderborn are connected via Gigabit Ethernet and under same router. Since LifeSocial has not implemented the function to deal with NAT problem, it will cause problem if there are instances outside router. The bandwidth between machines can be as high as $200MB/s$.

3.3.2 Social Parameters

The parameters is related to the social network under test.

Churn Churn indicates how often and how many users come in and out the network within a certain time interval. Churn rate used in random test of this master thesis is based on the design of previous thesis [6] as shown in Figure 5.7.

Application Load Application load is generated by user behaviour in the network, such as upload photo, send chat messages. This behaviour is shown in Figure 5.6 which is based on the work of previous thesis.

3.3.3 LifeSocial Parameters

The currently configurable parameters in LifeSocial is listed in Figure 3.4, 3.5 and 3.6. It includes the configuration parameters from bottom overlay to the top plugin layer.

enableHeterogeneityAwareness This parameter specifies whether the heterogeneity function is enabled in FreePastry. Heterogeneity function enables the overlay to distribute workload based on peer's capacity information so that the peer with few capacity will not be overloaded and the free capacity of strong peer will not be wasted. However since all the test machines or webservers have the same capacity, this function is turned off, since turning on this function will generate more messages and data traffics in the network. Heterogeneity is implemented by the project group 2012 $TeamOverlay$. The detail of this function is in the project group's final documentation.

heterogeneityCacheSize This parameter specifies how much cache in memory is needed for heterogeneity function. Due to heterogeneity function is not needed in this master thesis's test, the size of it is set to $1MB$.

CanYouProvideTimeOut This parameter controls how long the $CanYouProvideMessage$ from heterogeneity function will be timeout. It does not matter if the heterogeneity function is turned off.

providerListLength This parameter controls how many providers every responsible node stores. It does not matter if the heterogeneity function is turned off.

replicationFactor This parameter defines how many replicas is made for one data object.

objectSizeCalculationEnabled This parameter controls the object size calculation in PAST layer. It calculates and monitor the data objects passed through PAST. Since this calculation brings delay and PAST layer is very vulnerable to delays, it is turned off always.

randomSelectionModeEnabled This parameter controls whether the heterogeneity function picks the suitable provider based on the capacity information or a random selection policy. It does not matter if the heterogeneity function is turned off.

measuringData This parameter controls object size calculation in `Framework` layer and `Plugin` layer. It calculates and monitors data objects pass through `Plugin` layer and `Framework` layer. Monitoring these data flows is newly added in this master thesis and it is turned on for all tests.

socialMetricCollection This parameter controls whether the collection of social metrics is enabled. It collects all the social network related metrics, such as friend request, wall post etc. It is newly added in this master thesis and it is turned on for all tests.

autoStartMonitoring This parameter controls whether the monitoring function is automatically started once the user logins to network. It is needed when every user's statistics has to be collected or there is no user interface to start the monitoring function, as if LifeSocial is initiated from command line. This function is turned on in all tests, especially when LifeSocial is deployed in webserver and it has to be started from command line.

disablePIRouting PIRouting is short for parallel and iterative routing. Parallel routing enables user to not only send message to one node, instead a message is sent to X nodes and X is configurable. Iterative routing enables user to track the sent message in case that malicious nodes intercept messages. After a certain sender sends a message, once the message reaches a receiver, this receiver has to get the next receiver and sent it back to the sender, the sender will send it to the next receiver personally. By experiments these functions are not perfect yet, for instance due to parallel routing, a friend request might be received multiple times. Thus these two functions are turned off for all tests.

justParallel If this parameter is turned on and $disablePIRouting$ is turned on, only the parallel routing will be used, otherwise both mechanisms are used.

parallelity This parameter defines how many messages are sent in parallel in parallel routing. It does not matter if parallel routing is turned off.

combinedParameter This parameter specifies how many rounds are combined together in monitoring mechanism. The detail is explained in 4.4. The value of this parameter is set to 3 usually.

autoFriendRequestAccept This parameter controls whether one friend request is accepted automatically. It is especially important for testing, because it is impossible to manually accept friend request for each instance. It is always turned on in $Synthesis$ and $Random$ tests.

friendRequestAcceptedGUI This parameter controls whether A GUI is shown when the sent friend request is accepted. Since this GUI will keep alive if it is not manually closed, if too many unclosed GUI accumulated, it is dangerous that the GUI of LifeSocial might be stuck. It is turned off in $Synthesis$ and $Random$ tests.

messageAutoDeleted In LifeSocial messaging plugin, this parameter controls whether stored messages will be deleted after 1000 seconds. In all tests, this function is disabled, especially important for alpha test.

onlyGnuplot The monitoring function of LifeSocial generates three different kinds of files to store monitored statistics. They are $.csv$, $.dat$ and $GNUPlot.dat$ files. To plot all the statistics into graphs the software $GNUPlot$ is used, so only the $GNUPlot.dat$ file is useful. And in a long-term test, for instance 4 days, the each file is as large as around $80MB$. It is afraid that the quota of the machine might be exceeded, thus this parameter is turned on if it is necessary to save storage space.

mobileNodes Mobile nodes is a feature implemented by project group 2012 $OverlayTeam$. If a user login to network as a mobile node, it does not have to be responsible for message routing, data storage, but it still can route messages, store and retrieve data in network. However this feature is not perfect yet, it is turned off always. If it is turned on, a node logins to network as a mobile node if the port number it chooses is even, otherwise logins as a strong node.

monitorDataVisual The statistics can be visualized by LifeSocial in the runtime. However this visualization needs to store all the monitored data from the time instance initiated to now which keeps the memory increasing all the time. Therefore this function is turned off in a long term test.

bulletinURL This parameter specifies the url of the content which is show in the bulletin board of LifeSocial.

```
# enables the heterogeneity awareness (1 = enabled, 0 = disabled)
enableHeterogeneityAwareness = 0

# size of the heterogeneitycache in MB
heterogeneityCacheSize = 1

# timeout for a canyouprovidemessage in ms
CanYouProvideTimeOut = 5000

# maximal number of providers for each data object
providerListLength = 1

# number of replications of a data object in the system
# NOTE: It is important that #strongNodes >= replicationFactor+1
# in case weak nodes are present
replicationFactor = 5

# enables the object size calculation for the monitoring
# (1 = enabled, 0 = disabled)
objectSizeCalculationEnabled = 0

# enables the random selection mode (1 = enabled, 0 = disabled)
randomSelectionModeEnabled = 0

# enables the measuring data function, the data measured located in
# Plugin, Framework levels (1 = enabled, 0 = disabled)
measuringData = 1
```

Figure 3.4: LifeSocial parameters - 1

```
# enables the collecting of social metrics, such as friend requests
# (1 = enabled, 0 = disabled)
socialMetricCollection = 1

# auto start monitoring (1 = enabled, 0 = disabled)
autoStartMonitoring = 1

# disable the parallel and iterative routing
# (1 = enabled, 0 = disabled)
disablePIRouting = 1

# only enable parallel routing (1 = enabled, 0 = disabled)
justParallel = 0

# configure the parallelity, since this number is not constant,
# do not change it if necessary
# (0 = disabled, else = the parallelity u want )
parallelity = 1

# in monitoring, how many rounds you want to combine
combineParameter = 3

# enables the auto friend request accepted
# (1 = enabled, 0 = disabled)
autoFriendRequestAccept = 0

# enables the accepted friends request GUI
# (1 = enabled, 0 = disabled)
friendRequestAcceptedGUI = 1

# enables message(email) automatically deleted after 1000 seconds
# (1 = enabled, 0 = disabled)
messageAutoDeleted = 0

# only create GNUPlot monitoring data file
# (1 = enabled, 0 = disabled)
onlyGnuplot = 0

# mobile nodes, if enabled, the odd port number stands for strong
# node, the even port number stands for weak node
# (1 = enabled, 0 = disabled)
mobileNodes = 0
```

Figure 3.5: LifeSocial parameters - 2

```
# visualization of monitored data in runtime, this function will
# keep the memory usage increasing! (1 = enabled, 0 = disabled)
monitorDataVisual = 0

# url to the bulletin board
bulletinURL = a url
```

Figure 3.6: LifeSocial parameters - 3

3.4 Test Case

Test case defines a certain action with conditions and variables which is performed by the software or system under a certain environment. The result shows whether the software or system has passed the test case. The action of the test case in LifeSocial is minimized to the most basic social network methods of LifeSocial. It can be sending a friend request or adding a wall post. To execute a test case the precondition and parameter of it are necessary. The available test cases are described in the following table 3.3, table 3.4 and table 3.5, it is firstly concluded in previous master thesis [6].

Two more test cases are created in Project Group 2012 of LifeSocial, they are : `testcaseForum` and `testcaseStoreFile`. However due to `testcaseStoreFile` lack some conditions in OSGI 2.4, it is not working yet and it is work of the `Plugin Team` in the project group. Another two new plugins created by this project group is `App Store` and `LiveChat`, they are developed by the `Framework Team` and `Plugin Team`. However they did not create the test case and provide command to the OSGI. The difficulty for `App Store` to provide commands is that the whole functionality of `App Store` is not completely programmed as an individual plugin, and half of it is programmed in framework. To fix these three test cases is beyond the scope of this master thesis, but it is helpful and necessary to complete these three test cases in the future. Table 3.6 describes the test cases `testcaseForum` and `testcaseStoreFile`.

In previous thesis [6], it depicts the dependencies between all the test cases as in Figure 3.7. The following description for the figure 3.7 is quoted from his thesis: *"The shaded boxes with a label on the top left corner visualize preconditions. Each test case within a shaded box needs that precondition to be complied. For example, the Register test case is inhabited inside of the Bootstrap Server Running precondition box. Thus, to register to the network, there needs to be at least one bootstrap server running. Furthermore, test cases with outgoing arrows activate certain preconditions and therefore enable new*

Test Case :	**Register**
Description :	Registers a new user and logs into the network
Preconditions :	Bootstrap Server running
Parameters :	Bootstrap-IP, Bootstrap-BootPort, Own-Ip, Own-BindPort, Login-Name, Password, City, Country.
Test Case :	**Login**
Description :	Logs into the network with credentials of an existing user.
Preconditions :	Bootstrap Server running, Username already exists but not currently logged in.
Parameters :	Bootstrap-IP, Bootstrap-BootPort, Own-Ip, Own-BindPort, Login-Name, Password.
Test Case :	**Logout**
Description :	The user logs out of the network.
Preconditions :	Logged in user.
Parameters :	Login-Name, Password.
Test Case :	**Change Profile Picture**
Description :	The user changes the profile picture
Preconditions :	Logged in user.
Parameters :	Path to new profile picture.
Test Case :	**Change Profile Information**
Description :	The user changes a defined field in its profile.
Preconditions :	Logged in user.
Parameters :	Values for updated information and ID of the updated field.
Test Case :	**View Profile**
Description :	The user views the profile of a friend or the own profile.
Preconditions :	Logged in user, Friendship to other user(s).
Parameters :	Username of the owner of the requested profile.
Test Case :	**Send Chat Message**
Description :	The user sends a chat message to a user's friend and initiates a possible chat conversation.
Preconditions :	Logged in user, Friendship to other user(s).
Parameters :	ID of a friend, Text/Content of message.
Test Case :	**Reply to Chat Message**
Description :	The user replies to an active chat conversation.
Preconditions :	Logged in user, Friendship to other user(s), Incoming chat message / Active chat conversation.
Parameters :	Text/Content of message.

Table 3.3: Table of test cases - 1

Test Case :	**View Chat History**
Description :	The user browses a chat history
Preconditions :	Logged in user, Friendship to other user(s) Active chat conversation.
Parameters :	-
Test Case :	**Send Wall Post**
Description :	User comments a wall entry.
Preconditions :	Logged in user, Existing Wall Post.
Parameters :	ID of existing wall post, Text/Content of comment.
Test Case :	**Comment Wall Post**
Description :	User comments a wall entry.
Preconditions :	Logged in user, Existing Wall Post.
Parameters :	ID of existing wall post, Text/Content of comment.
Test Case :	**View Wall**
Description :	User browses a friend’s wall.
Preconditions :	Logged in user, Friendship to other user(s).
Parameters :	ID of a friend.
Test Case :	**Create Photo Album**
Description :	User creates a new photo album.
Preconditions :	Logged in user.
Parameters :	Title for photo album, Comment for photo album.
Test Case :	**Upload Photo**
Description :	User uploads a new photo to photo album.
Preconditions :	Logged in user, Existing photo album.
Parameters :	Path to photo, Comment for photo, ID of a photo album.
Test Case :	**View Friends Photo**
Description :	User views a photo from a friend’s photo album.
Preconditions :	Logged in user, Friendship to other user(s), Existing photo.
Parameters :	ID of a friend, ID of a photo album, ID of a photo.
Test Case :	**Search User**
Description :	User searches for a certain user.
Preconditions :	Logged in user.
Parameters :	Search term.

Table 3.4: Table of test cases - 2

Test Case :	**Search Group**
Description :	User searches for a certain group.
Preconditions :	Logged in user.
Parameters :	Search term.

Test Case :	**Send Friend Request**
Description :	User sends a friend request to a certain user.
Preconditions :	Logged in user.
Parameters :	Username of friend, Message to send along.

Test Case :	**Accept Friend Request.**
Description :	User accepts an incoming friend request.
Preconditions :	Logged in user, Incoming friend request.
Parameters :	-

Table 3.5: Table of test cases - 3

Test Case :	**Forum**
Description :	User creates a forum thread and add three posts in it.
Preconditions :	Logged in user.
Parameters :	-

Test Case :	**Store File**
Description :	User stores a file in the network.
Preconditions :	Logged in user.
Parameters :	File path.

Table 3.6: New test cases : Forum and Storage

test cases to be executed. Precondition boxes can be within other precondition boxes, which represents a chain of precondition. As example, to execute the View Profile test case, the preconditions Logged in User, Existing User Currently Offline and Bootstrap Server Running need to be complied. In the same way, a chain of test case executions can be obtained to activate a specific test case, by following the path from the specific test case backwards. Imagine, the creation of a new group is the goal. Before executing the test case Create Group, it is checked if the precondition Logged in User is complied. If not, try to execute the Login test case in advance, since this test case enables the Logged in User precondition. Prior to the execution of the Login test case, its preconditions are checked. It is checked, if there already exists a user that is currently not logged in for the given parameters. This procedure is repeated until all preconditions are fulfilled and the origin test case can be executed."
However as a result of the work of Project Group 2012 of LifeSocial, the `Group Plugin` is recreated. It is credited to `Plugin Team`. It introduces a new concept and functionality in `Group Plugin` which is called `"Forum"`. In Figure 3.9 a folder is first created in `Group Plugin` user interface, this creation of folder is mandatory for creating a forum, afterwards a forum "test" is created in this folder. Figure 3.9 shows the graphical user interface of creating a thread in forum. Figure 3.10 shows how to create a post in the forum thread and Figure 3.11 illustrates the view of viewing other user's post in a certain forum thread. The creation of a forum thread and adding a post in this thread, these two actions are included in the `testcaseForum`.
Another new feature added in LifeSocial is "Storage" plugin, it is credited to `Plugin Team`. It enables user to store files in the network and retrieve it by the `Storage key` of the file.
The messaging in LifeSocial is also improved by the `Plugin Team`, originally LifeSocial only supports one-to-one chat, now a new functionality "chat room" is developed. The chat room is initiated by a certain user and this user decide which friends he wants to invite to this chat room, then the invited friends will receive the notification.
Since the `Group Plugin` is recreated and two more features have been developed in LifeSocial, the Figure 3.7 needs to be modified, even though there are no correctly functional test cases for `Storage Plugin` and `LiveChat Plugin`. Since first the test cases for these two new features will be created in the future; second this Figure 3.7 not only illustrate the dependency between test cases, but also illustrate the available user actions in LifeSocial which will be helpful in analyzing the user behaviour in an `Alpha test`. The new test case dependency is shown in Figure 3.12. The black blocks in the Figure indicates the test cases need to be created in the future.

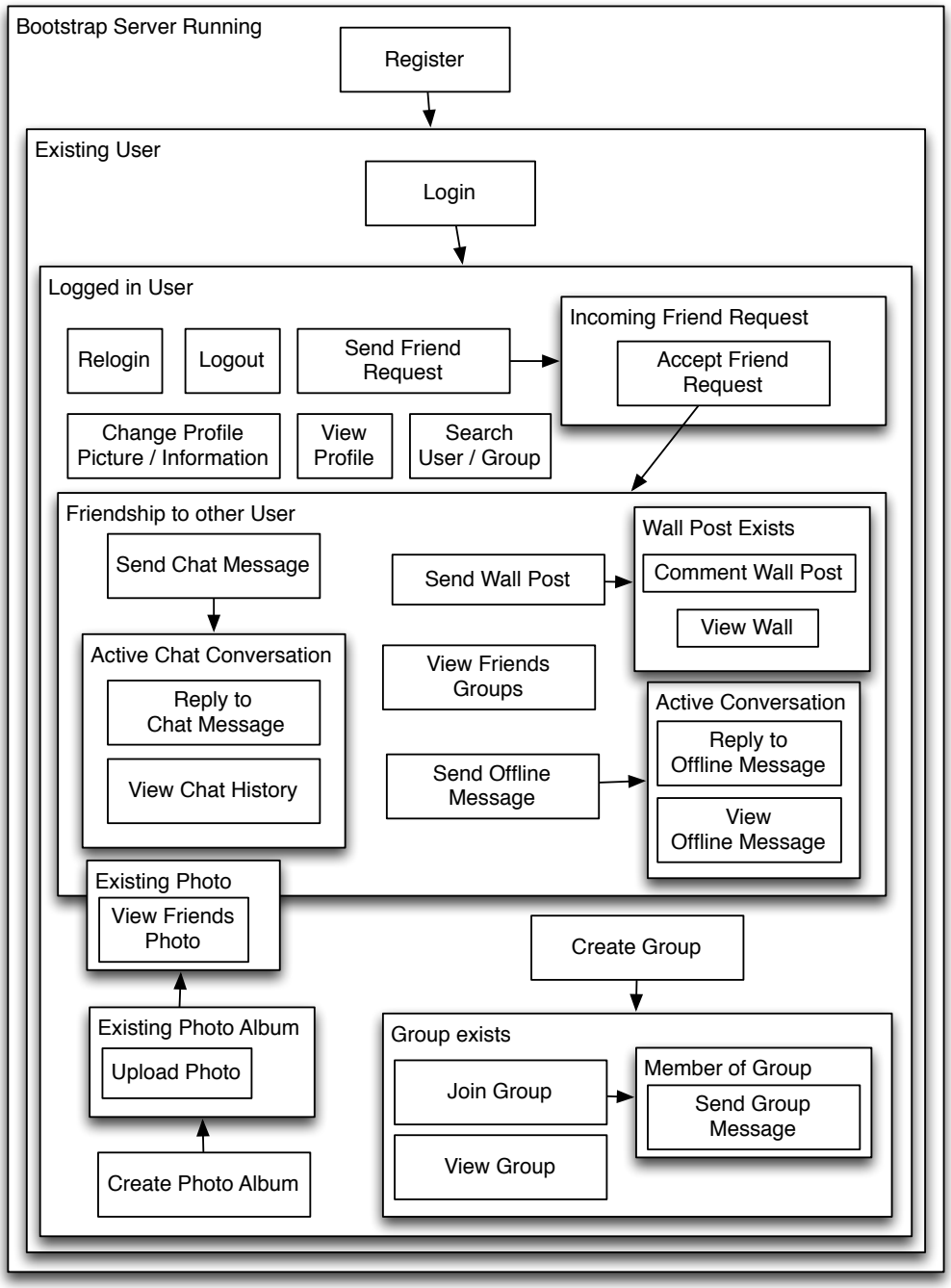


Figure 3.7: Dependencies between test cases.

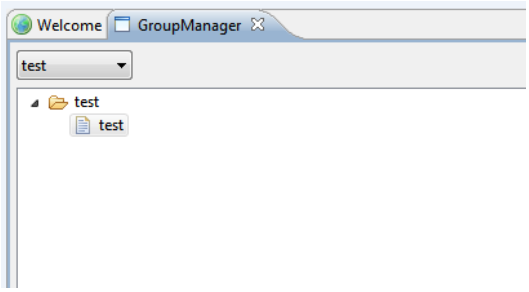


Figure 3.8: Create a forum.

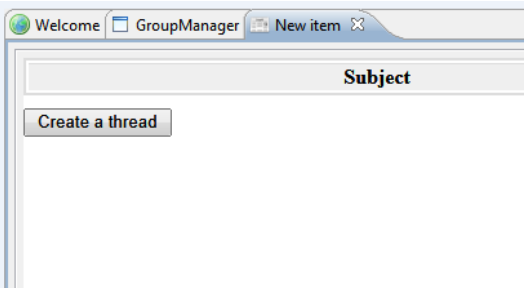


Figure 3.9: Create a forum thread.

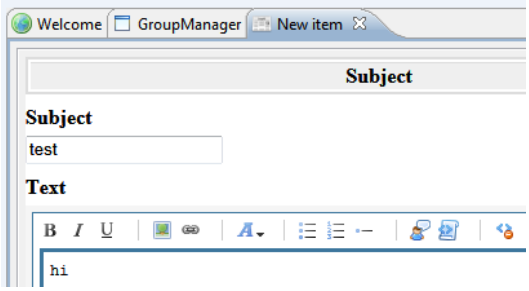


Figure 3.10: Forum user interface.

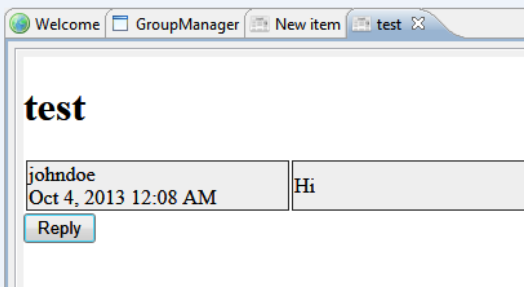


Figure 3.11: View other user's forum.

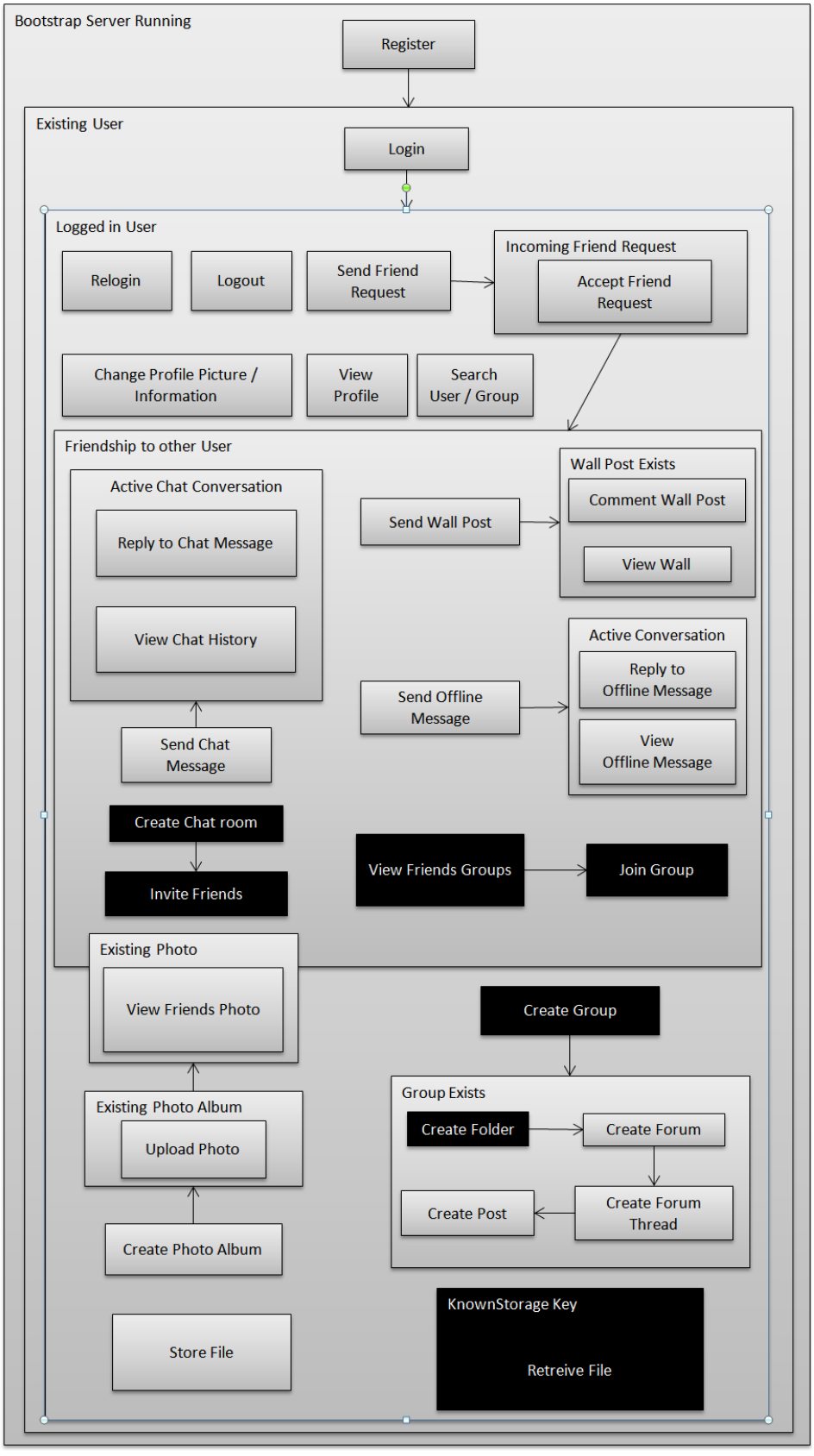


Figure 3.12: Current test case dependencies.

4 Monitoring

This section introduces the monitoring on p2p overlay. Section 4.1 introduces the concept and challenge in p2p network monitoring. Section 4.2 introduces a monitoring mechanism called SkyEye.KOM [5]. Section 4.3 shows the monitoring problem before this master thesis. Section 4.4 shows the improvements made to the monitoring.

4.1 Monitoring

Monitoring p2p network enables the evaluation of quality of service (QoS). The goal of monitoring is to monitor and modify system metrics to provide constant quality. Since it is unlike a service based system, to provide the constant quality in a p2p system meets many challenges. First the constant quality in a p2p network consists of various scenarios, for instance distributed storage; content delivery or discovery and contacting of users, none of them can be controlled in a centralized way. Second a p2p network has many uncontrollable elements, mainly due to the "churn" in the network which has a great impact on the connectivity, accessibility and many critical property of the network. Thirdly the capacity and connectivity of peers are different from each other.
Monitoring enables obtaining global knowledge on the system statistics. The monitoring should not bring a fairly large data traffic into the network. Usually the monitoring of p2p network is down by first recording the local metrics and then aggregating the local metrics based on a certain mechanism according to different monitoring categories, finally the global statistics is calculated and redistributed to each peer. This aggregation function has to be scale on different size of the network.
The modification of system based on gather statistics usually has to be on-the-fly and requires no-deployment, however currently LifeSocial is not able to be reconfigured during runtime.

4.2 SkyEye.KOM

SkyEye.KOM is a tree based monitoring mechanism, it assumes the monitored p2p-overlay is structured and it aims at high precision with low overhead. It

continuously collects system state information and redistribute to each node in the runtime. It introduces a new layer on top of the p2p overlay which means it has a wide applicability. The new layer uses the p2p overlay to route monitoring messages. Each node is assigned a `Domain ID` in SkyEye.KOM layer.
Once a node joins the network, it will be assigned a `Domain ID`. The value of `Domain ID` is the middle value of the domain interval. A domain interval can be [0,0.5] or [0.25,0.375], the largest domain interval is [0,1]. Thus when the first node joins the network it will be assigned a `Domain ID` of 0.5 and the second will be 0.75 or 0.25 depending on which domain interval it is assigned to. These `Domain IDs` build a tree topology which will not change over time. As illustrated in Figure 4.1, 7 nodes in the P2P overlay and each of them is assigned a `Domain ID` in the SkyEye.KOM layer. In this figure the node degree is 2 which results in a logarithmic tree size. This node degree is configurable and usually it is called as `split factor`. Each peer can be a coordination at different levels. It is obvious that every node in the network has a constant in and out message channels.

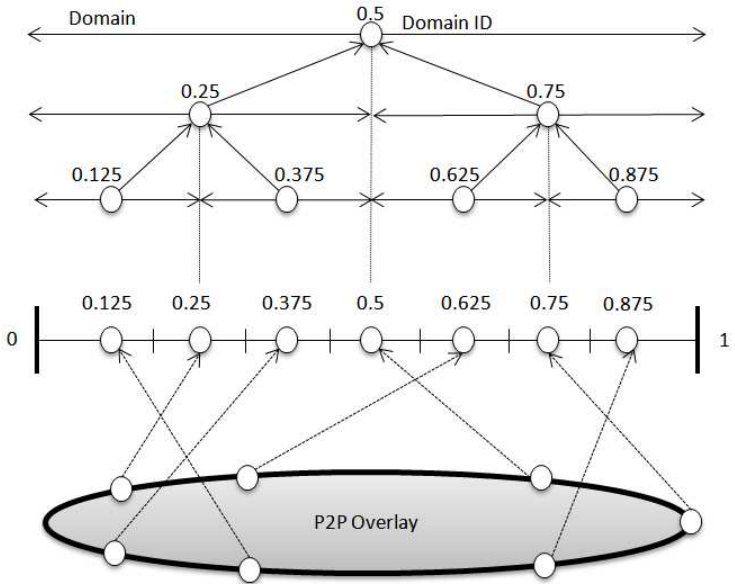


Figure 4.1: SkyeEye.KOM ID Domain.

Based on this tree topology, each peer periodically sends its local metrics to its parent. The parent aggregates its child nodes' metrics and the metrics of itself, afterwards it sends the aggregated metrics to its parent. This local aggregation mitigates the workload of the root node in the SkyEye.KOM layer, otherwise the root node has to calculate the metrics for the whole network which means the mechanism is not scale any more. Once the root node receives messages from its child nodes, it aggregates it child nodes' metrics and its owns, then it sends

Aggregation Strategy	Description	Aggregated Value Type
sum	Get the sum of all values	Long, Integer, Double
mean	Get the mean value of all values	Long, Integer, Double
greatest	Get the maximum value of all values	Long, Integer, Double
smallest	Get the minimum value of all values	Long, Integer, Double
smallestNonNegative	Same as smallest, but only get the non-negative value	Long, Integer, Double
assign	Default aggregation strategy, just assign the latest value to the metric	Any type

Table 4.1: Aggregation strategies

back the result, the global metrics, to its child nodes and the child nodes send back to their child nodes and so on.

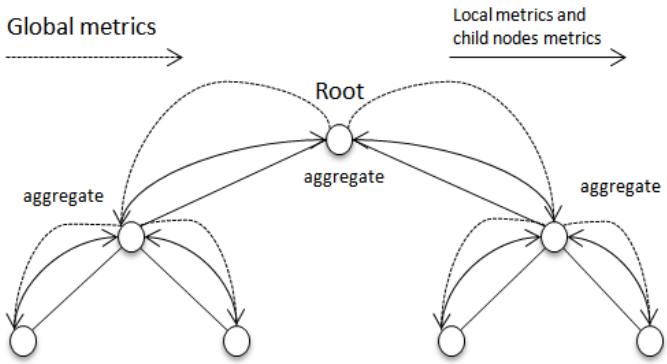


Figure 4.2: Message routing in SkyEye.KOM layer.

In Figure 4.2, it shows a three level tree graph, the aggregation happens in every level except the bottom level. The current available aggregation strategy is show in table 4.1. It is also possible for developer to customize strategies. From this graph it is observed that every coordinator in different layers generate the global view of the sub-tree under itself and send this sub-tree global view to its parent and every sent monitoring message is acknowledged. The final global view is aggregated in root node, then disseminate to child nodes.
Figure 4.3 shows monitoring activity diagram of one node. Once the node initiate the monitoring function which means it joins the new layer generated by SkyEye.KOM, it starts a scheduled task that it periodically sends monitoring message to parent, the period is T and it is configurable. During this time T the node receives monitoring messages from parent and child nodes. If the monitoring message is from parent, it will forward it to the child nodes, otherwise, it get the metrics from the message and stores it. Once the time reaches the end

of T, it first aggregates the metrics it stores with its own metrics. Afterwards if it is the root node, it will send back the aggregated metrics as global view to its child nodes, otherwise it will send a monitoring message which contains all the aggregated metrics to its parent.
It is noted that the messages which contains local metrics parent receives from child node and messages which contains global view child nodes receives from parent node are not synchronized. It means that child nodes are always sending the freshest metrics to the parent, however metrics that the nodes located in the high level receives is possibly elder that its own metrics. This problem occurs in a fairly deep tree or a in a network which has a slow message communication channel, it can be mitigated by the enlarge the node degree. If the node degree is enlarged to 8, a network with 100 nodes only has 3 levels. The parameter T control the precision of global metrics, if every node sends its metrics to the parent in a high frequency, therefore the global view will be updated in a high frequency.

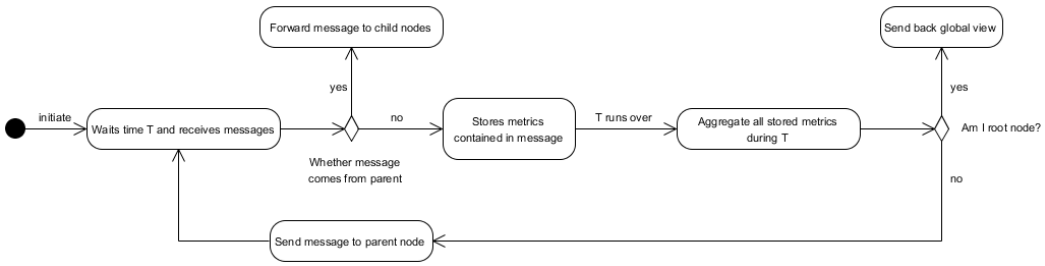


Figure 4.3: Monitoring activity diagram for one node.

4.3 Problem of Monitoring Mechanism

Since SkyEye.KOM monitoring mechanism uses p2p overlay to route messages, it leads to one potential problem that if the p2p overlay has a fairly large delay in message routing or there is a big jam in message communication, it will cause a negative impact on the monitoring result.
Since to introduce new monitored metrics in LifeSocial 3.2.1, the measurement for data objects is applied. Every message has to be measured size before being sent out which cause a delay in message channel. For example, the metric *GLOBAL_NETWORK_NODES* presents how many nodes currently in the network. If in a 4 nodes network and the node degree is 3, as depicted in Figure 4.4.
Nodes 2, 3 and 4 sends monitoring message to 1 periodically hence node 1 is supposed to receive 3 monitoring messages every period. However if there is a high delay in messages routing, the number of monitoring messages node 1

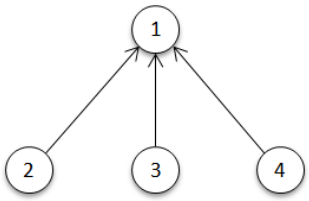


Figure 4.4: A simple 4 nodes structure.

receives every period can be less than 3, as illustrated in Figure 4.5. The circle with dashed border depicts not received message. Thus, the value of global metric $GLOBAL_NETWORK_NODES$ will be less than 4.

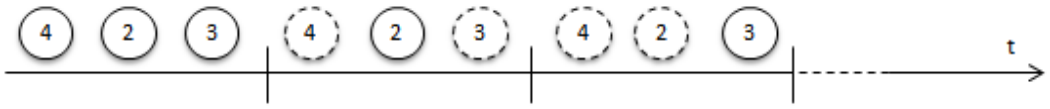


Figure 4.5: Monitoring in communication channel with high delay.

Figure 4.6 shows the metric *GLOBAL_NETWORK_NODES* measured by original monitoring mechanism in a 25-instance synthesis test. In this synthesis test every node manually logins into network one by one. It can be observed from the graph that in the first 30 minutes, the curve increases reasonably. Once all the nodes login to the network, the curve should keep horizontal until the nodes begin to logout. In this synthesis test, there is no logout test case or test case including logout action. However in Figure 4.6, after every node starting executing testplan, the curves starts to fall down frequently, especially between 40 and 65 minutes. During this period the $Wall$ related test cases are being executed, they are $testcaseSendWallPost$, $testcaseCommentWallPost$, $testcaseViewFriendsWall$, $testcaseViewOwnWall$ and also $testcaseForum$. Since these test cases implements the distributed data structure (DDS 2.4), it needs to exchanges messages more frequently than usual. The last half of the testplan is photo related test cases, they are $testcaseCreatePhotoAlbum$, $testcaseUploadPhoto$ and $testcaseViewFriendsPhoto$. They do not require large amount of message exchanges, but every uploaded photo needed to be measured which still generate delays before the upload photo is stored in the responsible node. As observed in Figure 4.6, the last half time, from 80 minutes to the end, the graph does not contain a low and deep ”black” part compared to the first half.

To prove that this distortion of graph is indeed caused by the message delay caused by data object measuring, the Figure 4.7 shows the metric *GLOBAL_NETWORK_NODES* measured in a synthesis test in previous master

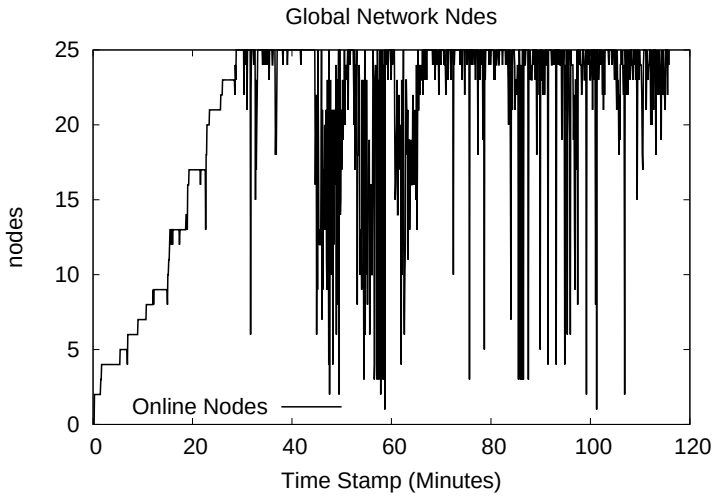


Figure 4.6: The metric *GLOBAL_NETWORK_NODES* measured by original monitoring mechanism.

thesis [6]. It is also a 25-instance test and the testplan is quite intensive. However the result graph keeps horizontal all the time. But it is observed that in previous master thesis's graph there is values that are larger than 25, which means within one time period, there are duplicate messages received from same node. It is caused by, the message which should be received in last period is received in current period, due to the delay. The original monitoring mechanism does not care the message is sent from which node so that the values of delayed message and the message received this period will be both aggregated, resulting in the aggregated value is larger the it is supposed to be. As depicted in Figure 4.8, the circle with bold line is the duplicate message, as a result within the third period, the aggregated global value of metric *GLOBAL_NETWORK_NODES* is 4. This is why there are values larger that 25 in Figure 4.7. This problem is fixed by specifies that from one node only one monitoring message can be received within one time period.
GLOBAL_NETWORK_NODES is aggregated based on the sum strategy, the other metrics, for instance data rate related graphs are better but still unsatisfied. As the *GLOBAL_NETWORK_PASTRY_DATA_SENT* shown in Figure 4.9, this graph shows how much data is sent by overlay during the whole test, the trend and basic figure can be recognized, but still not satisfied.

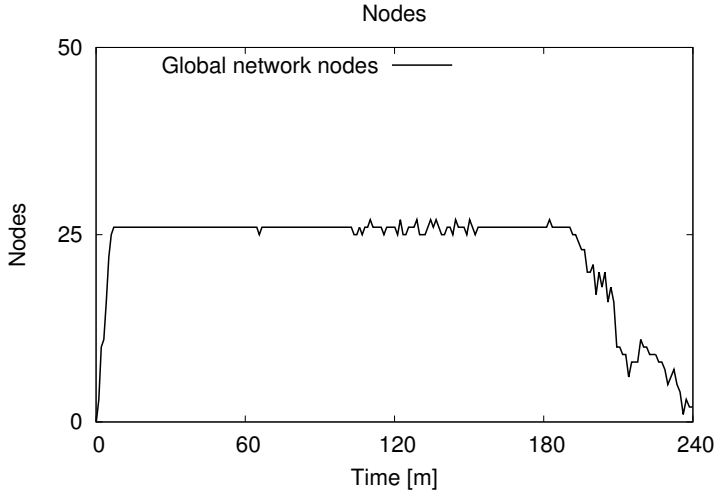


Figure 4.7: *GLOBAL_NETWORK_NODES* measured without data measuring function.

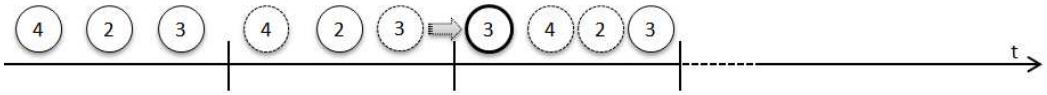


Figure 4.8: Delayed and duplicate message.

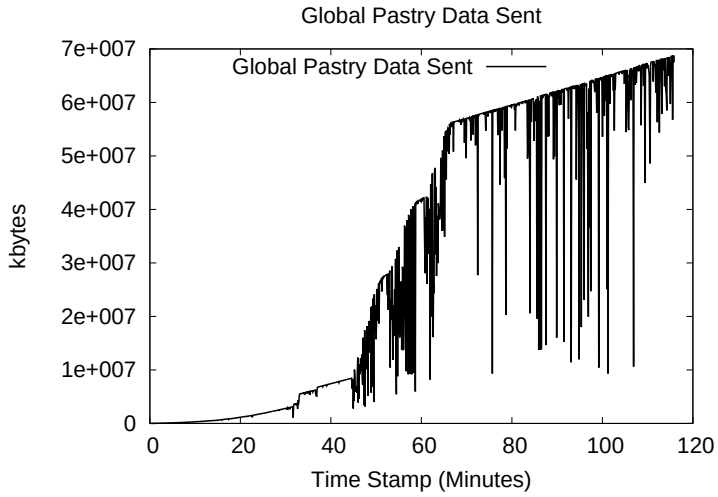


Figure 4.9: *GLOBAL_NETWORK_PASTRY_DATA_SENT* measured by original monitoring mechanism.

4.4 Improvements of Monitoring Mechanism

The solution to this problem is using historical-based smoothing. It means each coordinator will not only aggregate the metrics received in this period, but also consider the metrics received last periods. The trade-off of this solution is that while it smooth the result graphs, the freshness of the data is compromised. However it is obvious that Figure 4.6 is almost totally unrecognizable except the part before testplan started.
The first attempt of this solution is to reuse last two rounds' metrics. Every time before coordinator starting aggregating metrics, it first get all the latest messages received from different nodes in this rounds and last two rounds. Thus each child node's message at most has three versions: $thisRound$, $lastRound$ and $lastLastRound$. Similarly, each metric's value has at most three versions. From these three values, a final value which will be sent to parent has to be calculated. Thus all the metrics from one child node will be generated a final version from these three rounds' metrics. And these extracted metrics will be stored to be used of next round. Therefore for each child node's message, there might be 7 different situations as show in table 4.2. If in this three rounds, from one child node only one message is received, then the value of this round is taken; if two messages are received, then the absent message will be compensated by last extracted result, and from these three rounds a mean value will be calculated for each metric; if all three messages are received, then a mean value will be calculated for each metric. The pseudo-code of this mechanism is in 4.4

```
Receive Phase :
For every received message M do
```

lastLastRound	lastRound	thisRound	Decision
✓	✓	✓	mean
✓			lastLastRound's
	✓		lastRound's
		✓	thisRound's
✓	✓	last extracted result	mean
✓	last extracted result	✓	mean
last extracted result	✓	✓	mean

Table 4.2: 7 different situations of one child node's monitoring message in 3 consecutive rounds, from coordinator's perspective.

```
 if message M is global, do forward M to child nodes;
   else extract metrics metrics and sender's ID ID from M,
   stores ID and metrics in HashMap thisRound < ID, metrics >

Send Phase :
for every ID do
  if only round X of lastLastRound, lastRound, thisRound
  contains ID;
  put ID and X.get(ID) in the HashMap Final < ID, metrics >;
  else if round X and Y of lastLastRound, lastRound, thisRound
  contains ID;
  compensate the absent round with lastFinal < ID, metrics >;
    for every metric do
      Get mean value from three rounds' values;
      put metric in metricsBuffer;
  put ID and metricsBuffer in Final;
  else if all rounds contains ID;
    for every metric do
      Get mean value from three rounds' values;
      put metric in metricsBuffer;
  Aggregate Final and LocalMetrics;
  if myself is root do send back global view;
  else send to parent;
  lastLastRound = lastRound;
  lastRound = thisRound;
  lastFinal = Final;
  clear thisRound;
```

The result of *GLOBAL_NETWORK_NODES* of this improved mechanism is shown in Figure 4.10. It is a test with 15 instances. Compare to Figure 4.6, it is clear that the low and deep "black" in Figure 4.10 is mitigated. The curve does not stay in the low part for a long time. However this graph is still not satisfied enough, the curve still falls down to low parts frequently. It is assumed that, if more previous rounds are combined, the graph will be more smooth.
However, with more combined rounds the freshness of data will be more compromised. This trade-off between freshness of data and smoothness should be able to be controlled by the actual situation. If the network has unobstructed communication channels and the instance is not producing delays, the freshness of data can be increased while the result graphs can still keep smooth, otherwise it has to be decreased so that the result graphs can keep smooth.

Thus it is convenient to provide a parameter $combinedRounds$ to control how many rounds it is needed to be combined. This combine action is only applied on root node so that other nodes do not introduce more delays.

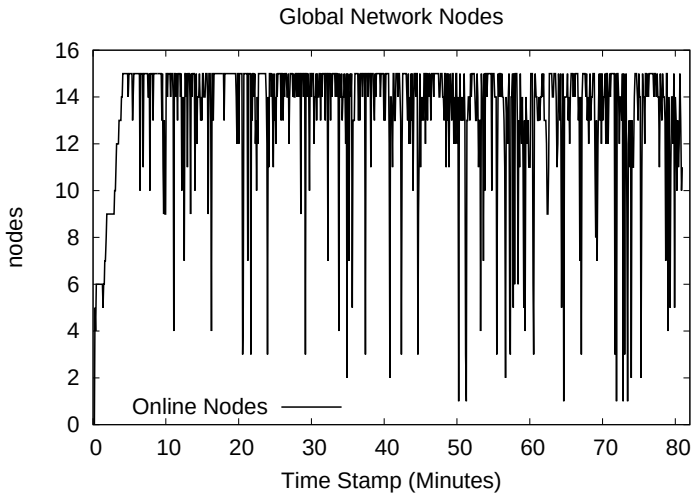


Figure 4.10: Result of metric *GLOBAL_NETWORK_NODES* measured by the new monitoring mechanism that reuse last round's information.

In this improved configurable monitoring mechanism, root coordinator initiate a Java Vector() once the monitoring functions started. This vector works like a FIFO bag, it stores every round's received metrics in corresponding to the sender's ID in a Java HashMap(). After the coordinator sending message to its parent, the vector discards the last elements it stores and initiate a new empty HashMap to store the metrics received in next round, as depicted in Figure 4.11. In this figure the $combinedRounds$ is configured to 5. Every time after the message is sent, the last element of the vector is discards and the first element is used to store the received metrics in current round. The precondition of this graph is that, the vector is already filled with 5 elements, otherwise the vector will just keep increasing and nothing will be discarded after the message sent until it is full.

If $combinedRounds = X$, every round's final metrics is generated based on $X-1$ last rounds' and this round's information. The pseudo-code above is modified as below from root node's perspective. This time for the sum aggregation strategy the max value from last X rounds is chosen; the $mean$ aggregation strategy the $mean$ value from last X rounds is chosen; the $smallest$ aggregation strategy the $smallest$ value from last X rounds is chosen; the $greatest$ aggregation strategy the max value from last X rounds is chosen. The reason to change the choose policy in this way is, by tests this choose policy generates the smoothest result

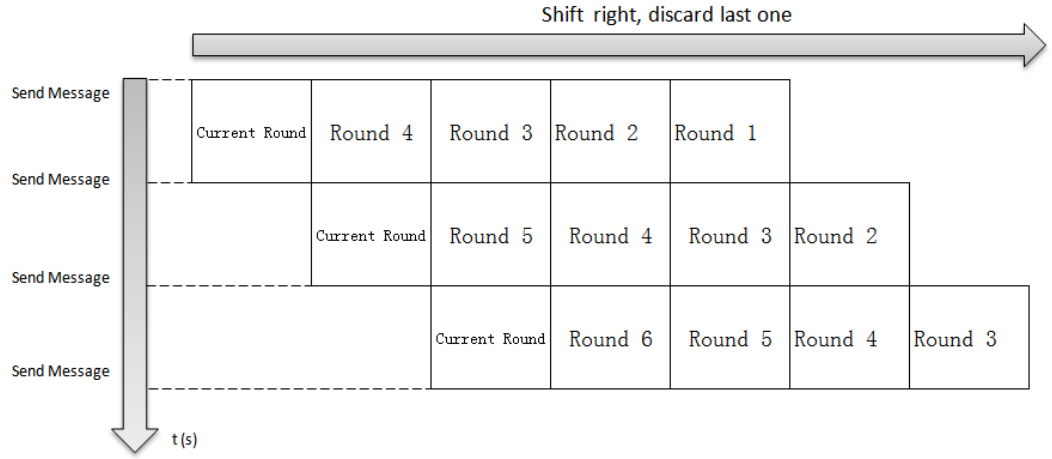


Figure 4.11: Shift the stored rounds every time after the monitoring message is sent to parent..

graphs.

Initiation Phase:

initiate Vector v;

initiate an empty HashMap as the first element of v;

Receive Phase :

For every received message M **do**

Extract metrics $metrics$ and sender's ID ID from M,

stores ID and $metrics$ in the first element of v

Send Phase :

```
for every ID do
Get metrics received from last X − 1 rounds and this round;
for every metric do
  if aggregation strategy == sum do
    Get the max value from X values;
    Stores metric in metricBuffer;
  if aggregation strategy == mean do
    Get the mean value from X values;
    Stores metric in metricBuffer;
  if aggregation strategy == smallest do
    Get the smallest value from X values;
    Stores metric in metricBuffer;
```

```
    if aggregation strategy == greatest do
      Get the max value from X values;
      Stores metric in metricBuffer;
  Aggregates metricBuffer with local metrics;
  Sends back to child nodes;
  Shifts vector elements;
```

For all the other nodes in the SkyEye.KOM layer except root node, each of them do not use any historical information of received metrics and just handle metrics in the normal way.
The result of global network nodes and pastry data sent by this $combinedRounds$ methods is shown in Figure 4.12 and Figure 4.13. It is a 50-instance test. In these two graphs the figure of curve can be clearly recognized. However, there are many huge impulses in the graphs, the reason of these impulses remains unclear, even though it has been debugged for many times. Due to this reason, in this $combinedRounds$ mechanism, it is not using the metrics sent last round to compensate the null ones in the current round.

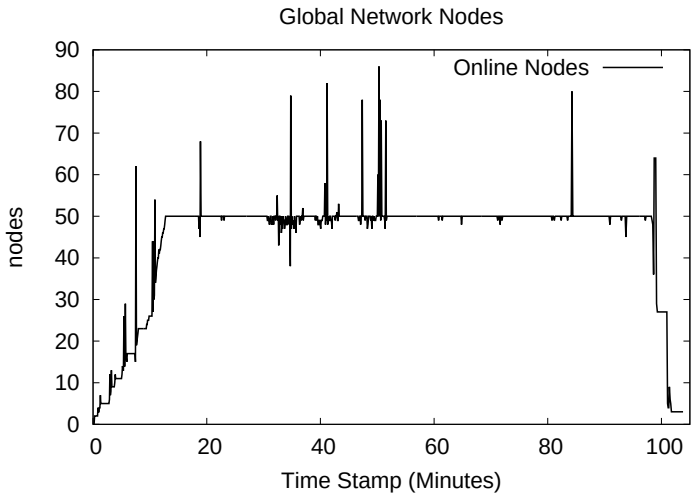


Figure 4.12: *GLOBAL_NETWORK_NODES* measured by new mechanism.

This is all about current monitoring function, the control parameter *combinedRounds* can be configured in

./org.lifesocial.core.utilities/config/LifeSocialConfig.txt

If the data measuring function is opened and LifeSocial instance running with quite intensive tasks, the recommended $combinedRounds$ is 5.

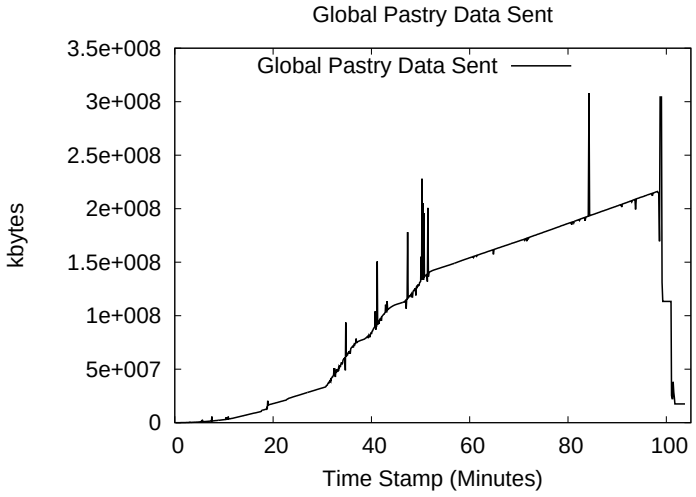


Figure 4.13: *GLOBAL_NETWORK_PASTRY_DATA_SENT* measured by new mechanism.

5 Test Setup

This section introduces the basic test environment first. The it first introduces a standard test process and afterwards it introduces the setups of synthesis test, random test, long term test and alpha test in detail. Each test includes three basic parts : deployment of instances, testplan and monitoring data processing. Random test without logout is not introduced in detail, since the only difference it with the random test is that slaves will not logout during test.
Section 5.1 introduces test environment. Section 5.2 introduces the standard process of running a test. Then section 5.3, section 5.4, section 5.5 and section 5.6 introduce $Synthesis$, $Random$, $Longterm$ and $Alpha$ tests respectively.

5.1 Test Environment

This section introduces the test area of this thesis. The machines pool is in University Paderborn shown in Figure 5.1. There are 20 machines in this pool and each machine is equipped with the following components:

- Processor : Intel i7-2600
- RAM : 8 GB
- Operating System : openSUSE 3.41.1-x86_64
- Network Interface : Gigabit Ethernet
- Java VM : 1.7.0_40 build 24.0-b50

In addition to these machines, 10 virtual machines located in the university's they are used to initiated long term instances for alpha test and long term test. These virtual machines are equipped with the following components:

- Processor : Intel(R) Xeon(R) CPU
- RAM : 2 GB
- Operating System : openSUSE 3.41.1-x86_64
- Network Interface : Gigabit Ethernet



Figure 5.1: Machine pool of University Paderborn.

- Java VM : 1.7.0_40 build 24.0-b50

5.2 A Standard Test Process

This section introduces a standard test process which can be a framework for all kinds of tests. It includes the basic information of runtime environment of LifeSocial, deployment of instances, testplan and monitoring data processing. All kinds of test can be extends based on these information.

5.2.1 Runtime Environment of LifeSocial

The current version of LifeSocial is developed in java, including its three main parts : overlay, framework and plugins and GUI. The executable operating systems of LifeSocial are : Linux, Windows and MacOS. The memory consumption for initiation LifeSocial is at least 40MB. After initiation the memory usage of LifeSocial keeps increasing but slowly, the reason is explained in Section 7.1. All the above requirements are listed as follows:

- Operating system : Windows, Linux, MacOS
- Java 1.7
- Memory : at least 40MB for initiation

5.2.2 Deployment

LifeSocial instance can be deployed anywhere fulfilling the requirements listed in section 5.2.1. However due to some characteristics of LifeSocial, two more things should be mentioned in deployment.

Graphical user interface First, the initiation of LifeSocial requires to render its graphical user interface, usually a monitor is equipped if it is initiate in a real machine. If it is initiated in a virtual machine or some where without monitor from command line, it needs a display server to work as a monitor. Two solutions are known currently. The first solution uses `X virtual framebuffer` and the second one uses `VNC`. In the `Long term test` and `Alpha test` the first solution is used, because the $Xvfb$ component is pre-installed in the virtual machines while VNC is not. And student does not have root rights in University-Paderborn.
$XVFB$ is a display server implementing the X11 display server protocol. According to wiki:Xvfb [3], this virtual server does not require the computer it is running on to even have a screen or any input device. Only a network layer is necessary. To start the server first make sure the $Xvfb$ component is installed, the following command is used:

```
Xvfb :1 -ac -screen 0 1024x768x8&
```

The server will listen for connections as server number 1 without access control restrictions, and screen 0 will be depth 1024x768x8. Afterwards using to the following command to set the server run on the current screen (which is not the screen created by $Xvfb$ command) :

```
export DISPLAY=:1
```

NAT issue NAT is short for Network Address Translation. It is a process of translating IP address while message is being sent through routers. NAT issue is that messages can not be received or sent out by the instances behind a router. The current version of LifeSocial can not handle this issue perfectly, thus all instances have to be deployed under the same routing, or within the same LAN.
Except these two precautions, a normal deployment of test of LifeSocial includes one master and multiple slaves. The master's address is know by all instances.

5.2.3 Testplan

A testplan or the actions performed by each instances during the runtime can be divided into three categories : synthesis testplan, random testplan and real user actions. Synthesis testplan is used in synthesis test. The goal of this testplan is to examine the functionality of software, at the same time generating data traffics in the network so that to observe the performance of p2p network. Random testplan is designed to imitate real user's behaviour. This testplan can create and control the "churn" rate in the network so that the performance of the p2p network under different "churn" rate can be monitored. The real user actions will be monitored and stored for analysis user behaviour afterwards. It also generates data traffics and "churns" in the network.

5.2.4 Monitoring Data Processing

After the test, each instance generates a $SkyEye$ folder which contains all monitored data. There are two $.dat$ files and one $.csv$ file in this folder. Data is stored in "**GNUPlot.dat" file which can be plotted by $GNUPlot$ tool. The format of the file is shown in table 5.1. The first part of the file lists all the monitored metrics in a column and indicated by the symbol $\#$. The first metric is always $TimeStamp$. Under all the metrics a blank row is added. After this blank row the second part stores each metric's value. In the second part, the first column stores the value of first metric which is always $TimeStamp$, and second column stores the value of second metric and so on as depicted in table 5.1
$GNUPlot$ is a portable command-line driven graphing utility for many platforms. It supports many different types of files, the file format in table 5.1 is

```
1   # TimeStamp
2     # metric1
3     # metric2
4     # metric3
...
```

1	2	3	4	
t1	metric1Value	metric2Value	metric3Value	...
t2	metric1Value	metric2Value	metric3Value	...
t3	metric1Value	metric2Value	metric3Value	...

Table 5.1: GNUPlot .dat file.

one of them. A script is used for plotting these *.dat* files. The script looks as follows:

```
set termoption dashed
set ylabel ''TIMESTAMP''
set xlabel ''Time Stamp (Minutes)''
set ytics nomirror
set xtics nomirror
set terminal pngcairo size 800,600
set output 'TIMESTAMP.png'
plot \
"data1.dat" using ($1/60):1 title "Node 0" linewidth 3 with lines, \
"data2.dat" using ($1/60):1 title "Node 1" linewidth 3 with lines, \
"data3.dat" using ($1/60):1 title "Node 2" linewidth 3 with lines
replot
```

After the run of test, each instance generates a *SkyEye* folder which contains all monitored statistics.

5.3 Synthesis Test

In Synthesis test all machines starts testplan at same time. It is useful in examining the functionality of the software.

5.3.1 Deployment

All the instances are deployed in machines pool in University Paderborn. There are three different levels of synthesis test, 25 instances, 50 instances and 100 instances. The idea to have different number of instances is to analysis the p2p network behaviour under different network scale.
The code of LifeSocial in each machine is synchronized by $smartgit$. Every test each instance generates a $storage$ file and $SkyEye$ statistics. Since for every new test the historical information from last test is unfavourable, the $storage$ file needs to be delete after every test. $SkyEye$ statistics are mandatory for generating test results, it needs to be collected. It will be a lot work if these deletions and collections are done manually. Therefore a public/private key system is established in the network. From one single node the deletion and collection actions can be easily done by two shell scripts.
One master is initiated in the network, every slave logins via the master. Master keeps all slaves' information and is responsible for issuing commands to slaves. Master instance hold two additional views: `Master View`, Figure 5.2 and `TestCaseDelegater` 5.3. The master view shows online user, register user and bootstrap server. Online user indicates the user online; register user indicates the registered user which might be not online; bootstrap server indicates from which server the user logins to network, to login to network user first contact master, afterwards master allocates a bootstrap node for user to login.

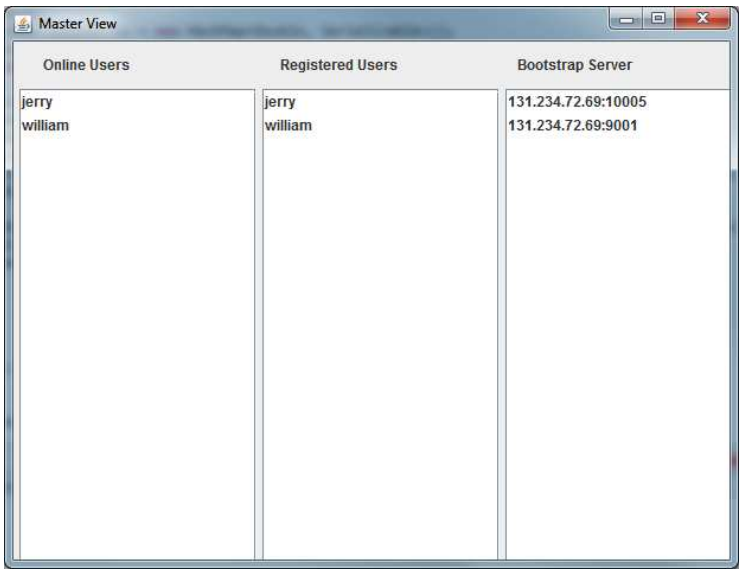


Figure 5.2: Master View

TestCase Delegater is designed to issue commands to slaves. The bottom mid-

Command	Description
ClearTestPlan	Empty the currently being executed testplan
EnableRandomMode	Slave starts running random mode
DisableRandomMode	Slave stops running random mode
ExecuteTimedTestPlan	Slave starts executing timed test plan
LoadTestPlan	Slave starts executing test plan
PrintTestPlan	Slave console prints the currently executed testplan
Wait	Slave thread wait for a certain period

Table 5.2: Commands in test delegater.

dle scroll panel shows the online users. The `Send to all` check box if checked, then all the users are selected. The top left dropbox controls which command is sent to slaves. The current available commands besides some test cases, are shown in table 5.3. The right dropx box indicates the execution priority of the sent commands. The top left box controls how many times the commands have to be executed by the slave. After all these are set, the command can be sent by the *send* button.

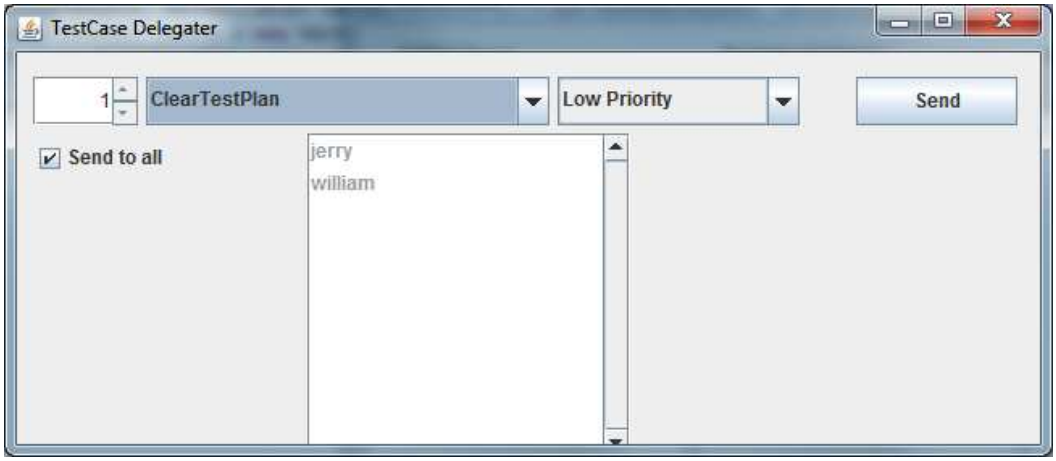


Figure 5.3: TestCase Delegater

After all slaves login to network, master sends out the command *ExecuteTimedTestPlan*. The difference between timed testplan and testplan is that there is time bound for each test case. As an example of timed testplan shown as following:

20x TestCaseCreatePhotoAlbum 10m

$20x$ means how many times the test case should be executed. After that is the

test case to be executed. $10m$ defines the time bound for this test case, if the test case has not been executed 20 times within 10 minutes, the rest repetitions will be skipped. The purpose of using timed testplan is to synchronize all the instances.

5.3.2 Testplan

This section describes the testplan used in synthesis test. The goal of the testplan is to first generate data traffic so that the behaviour of P2P network under much data traffic can be monitored; second is to examine the functionalities of LifeSocial. The testplan is shown below:

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

Besides test case $wait$, 10 test cases are executed in 85 minutes. And by experiments this testplan generates quite a lot data. The parameter of the test case $wait$ between other test cases are configured based on experiments. Similarly the time bound of each test case is also based on experiments.
First two test cases are profile related, they work as ”warm up” test cases which let nodes in the network exchange messages with each other and get knowing each other. The next 4 test cases are wall related test cases, including $TestCaseForum$ these test cases are using the distributed data structure.

DDS requires nodes to exchange more messages in which results more memory consumption, since each sent message has to be measured size and this measurement consumes memory. The last part is photo related test cases. These test cases functions normally and safely.
$SendFriendRequest$ test case is not included in the testplan. Based on previous tests, this test case will cause some instances stuck. Since all users information are stored in master while each slave has no idea of which users exists in the network. At the beginning of test case $SendFriendRequest$, slave invokes RMI methods to get a username from master. If too many requests are sent to master, due to some reason, some slave are stuck. However if the master send $SendFriendRequest$ command to slaves one by one, it works finely.
Another skipped test case is $SendChatMessage$. It also leads some instances to stuck in the runtime. The reason remains unclear. A test of executing this test case 200 times without data measurement with 2 instances leads to one of them stuck.
Other test cases are skipped because they are not generating too much data traffic.

5.3.3 Monitoring Data Processing

Since there are still some flaws in the improved monitoring mechanism and the reason remains unclear. As shown in Figure 4.12 and 4.13, there are many impulses in these two graphs. Since at least the each instance's local metrics are monitored and stored correctly. One brute-force solution created in this master thesis is to manually generate the global $GNUPlot.dat$ file based on each instance's local metrics.
A java program is created to manually produce this global $GNUPlot.dat$ file. The basic mechanism of this program is simple. It first reads in all instances' local metrics and stores them in a HashMap as follows :

```
HashMap<nodeID, HashMap<metricName, metricAttribute»
```

Since each instance has different number of metrics, the first step is to get all the metrics considering all instances'. After wards due to the timeline of each instance is not synchronized, so a common timeline is made. It is made in this way, the start time of this timeline is surely 0, the end time is the last node which left network and the timeline is segmented by a customized interval size, by default the interval is 5 seconds long. Then at every 5k ($k = 0, 1, 2, 3...K \in N$) seconds for every metric M, the program traverse each instance's timeline and finds the closest time to 5k, and search for the value of M at this time. After the data of this new $GNUPlot.dat$ is collected, it is written into a new $.dat$ file. The execution of this java program requires a pretty long time and

memory consumption. A powerful webserver with 16GB memory is used. The manually generated graphs of metric $GLOBAL_NETWORK_NODES$ and $GLOBAL_NETWORK_PASTRY_DATA_SENT$ are shown in Figure 5.4 and Figure 5.5. It is the result from a 25-instance test. It can be found that there is no impulse in the graphs and the graphs look smooth. This program currently is not durable for the result from a long term test or alpha test, since the memory consumption for a several days long $.dat$ file will be incredibly huge.

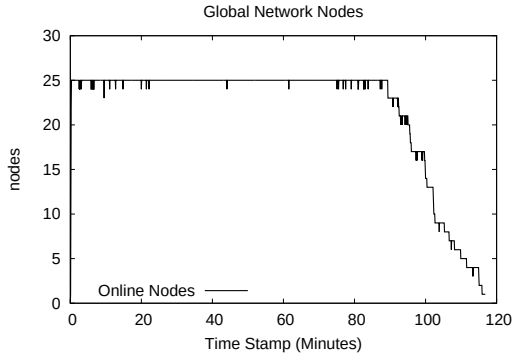


Figure 5.4: Manually created *GLOBAL_NETWORK_NODES*.

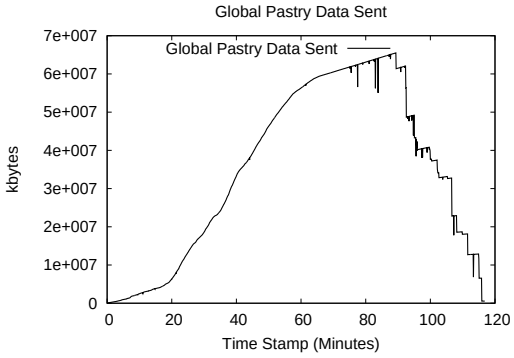


Figure 5.5: Manually created *GLOBAL_NETWORK_PASTRY_DATA_SENT*.

5.4 Random Test

This section introduces random test. In random test every user logins to network for a randomly selected long duration, afterwards logs out.

5.4.1 Deployment

In random test the deployment is as same as synthesis test.

5.4.2 Testplan

There is no fixed testplan for each instance to execute. Once a user logins to network, it first executes a start script which contains several test case. Instead of a timed testplan, the start script is a normal testplan since it does not need all the users to execute it synchronously. In a normal testplan all the test case are arranged directly one after another without any interval or time bound. The format of a normal testplan is as following:

```
1x TestCaseChangeProfilePicture 5x TestCaseChangeProfileInformation
```

The test cases are directly one after another, only the number of repetitions of the test case is specified. The start script used in random test is as following:

```
1x TestCaseSendFriendRequest
2x TestCaseChangeProfilePicture
2x TestCaseChangeProfileInformation
2x TestCaseCreatePhotoAlbum
2x TestCaseUploadPhoto
2x TestCaseViewFriendsPhoto
```

Although $TestCaseSendFriendRequest$ is problematic which will cause some instance stuck, there is no better solution yet. Having a friend is the precondition for most test cases, it is very important for a new login node to request a friendship in the network.

5.4.3 Monitoring Data Processing

The monitoring data processing in random test is done in a standard way.

5.5 Long Term Test

Long term test is to examine the functionality of the software under a long term run.

5.5.1 Deployment

In the long term test, 5 instances are deployed in 5 virtual machines in the webserver respectively. These five machines are in the same LAN. and one of them is master. They use $Xvfb$ server to work as display server.
Besides the fixed display server problem, another obstacle in running instances from command line is, without graphical user interface, two views shown in Figure 5.2 and Figure 5.3 are missing. After all slaves login to network, it is impossible to send commands to slaves. There are two solutions to this problem. First, the master instance can be initiated in a real machine which has a monitor for rendering graphical user interface. After all slaves login, master sends commands to them and logs out network. It is why there is a 10 minutes test case $Wait$ in the long-term test's testplan. It is for all instances to recover from master's logout.
Another solution is to add commands to the $OSGI$ console to control each instance. $OSGI$ commands can be added by implementing

`org.eclipse.osgi.framework.console.CommandProvider` interface. The interface contains only methods for the help. The console should use inspection to find the commands. All public commands, starting with a '_' and taking a CommandInterpreter as parameter will be found. Since `com.p2pframework.tester.component` class has already implemented this interface, it is a suitable place for adding the commands needed in long-term test. An example of implementing this interface is shown below.

```
1  public class TestComponent implements CommandProvider
2  {
3    public String getHelp()
4    {
5      System.out.println("--- Test Component Commands ---");
6      System.out.println("  The commands available here are : ");
7    }
8
9    public void _test(CommandInterpreter ci)
10   {
11     String arg = ci.nextArgument();
12     if (arg == null)
13     {
14       getHelp();
15       return;
16     }
17     System.out.println("User has typed in the console : " + arg);
18   }
19 }
```

In the above example, the `getHelp()` method is the inherited abstract method from interface `CommandProvider`. A manual about the command provided by this class can be written here. To provide a *test* commands as shown above, a *underline* has to be added before the name of the method. As well as the method parameter, `CommandInterpreter` has to be added as method parameter. Once LifeSocial instance is initiated, to call the command *test* just type in console :

`CommandName + arguments : test hello`

Using method `ci.nextArgument()` returns what user has typed after the command as *String*. If there is anything wrong in using this command, `getHelp()` method should be called. The needed command in long-term test is to ask all slaves to execute the timed testplan. It is added as following shows:

```
1  public void _executeTimedTestPlan(CommandInterpreter ci)
2  {
3    Collection<String> recept = new ArrayList<String>();
4    recept.add("all");
5    master.sendTestCaseCommand(1, "ExecuteTimedTestPlan", Priority.LOW, recept);
6    System.out.println("TimedTestPlan is sent!");
7  }
```

The collection $recept$ contains all the receiver of this command. If only string `all` is in this collection, then all slaves will receive this command.

5.5.2 Testplan

To enable this long term test running for several days, the testplan should make the instances generate some data traffic instead of keeping idle all the time. Thus it is designed as a long term synthesis test. The timed testplan used in long term test is loose and long. It keeps the instance performing a certain action but not in a intensive manner. One period of the test plan is as follows:

```
1x Wait 10m
1x TestCaseChangeProfilePicture 1m
1x Wait 5m
1x TestCaseChangeProfileInformation 3m
1x Wait 5m
1x TestCaseSendWallPost 10m
1x Wait 5m
1x TestCaseCommentWallPost 10m
1x Wait 5m
1x TestCaseViewFriendsWall 10m
1x Wait 5m
1x TestCaseViewOwnWall 10m
1x Wait 5m
1x TestCaseForum 10m
1x Wait 5m
1x TestCaseCreatePhotoAlbum 10m
1x Wait 5m
1x TestCaseUploadPhoto 10m
1x Wait 5m
1x TestCaseViewFriendsPhoto 10m
1x Wait 60m
```

At the beginning a 10 minutes $Wait$ test case is executed. Afterwards several test cases are executed in order, in the end is a 60 minutes test case $Wait$. This test case works as a pause for every instance. This testplan is repeated for tens of times, the length of this testplan is 7-day long. However logically these instances will be crashed due to memory exhausted.

User Behaviour

The imitation of real user behaviour in the random test is already implemented by previous master thesis [6]. In his thesis he designs user behaviour based on the studies of Benevenuto et al. [4]. Based on this reference, previous master thesis first implements "churn" in the test. He designs each user joins the network and stays in the network for a predefined duration. After this duration this user will logout the network and a new user will join the network after a short period. The online duration distribution designed by previous master thesis is shown in Figure 5.7. He points out that *As example, approximately* 17.11% *of the users spent 7 minutes and 48 seconds online. However, it does not reflect the fact, that this time may be split into several sessions, since the users may log in multiple times, up to 4.1 times per day, during the 12-day period. Nevertheless,* 63% *of the monitored users only logged in once during the 12-day period, so the presented data may suffice for the online duration time of the peers during the tests.*
Previous master thesis also described the available user activities and categorize them into 6 categories: $Profile\&Friends$, $Wall$, $Communities$, $Photos$, $Search$ and $Messages$. Since after his thesis LifeSocial has been added new features and removed some features, the current available activities are shown in table 5.3. The transition from one activity to another depends on the previous activity. Based on the studies of Benevenuto et al. [4] previous master thesis designed the $CategoryTransitionProbability$ for the categories of LifeSocial and it is shown in Figure 5.6. The only different in this thesis's tests is in $community$ category there is only $Forum$ test case know since the whole group plugin has been recreated.

5.5.3 Monitoring Data Processing

The monitored data in long-term test is processed in a standard way.

Category	Activity / Test Case
Profile & Friends	Change Profile Picture Change Profile Information View Own Profile View Own Wall View Friends Profile View Friends Wall View Friends Groups
Wall	Send Wall Post Comment Wall Post View Own Wall View Friends Wall
Communities	Forum
Photos	Create Photo Album Upload Photo View Friends Photo
Search	Search User Search Group Search City Search Country
Messages	Send / Reply Chat Message View Chat History

Table 5.3: Categorization of activities in LifeSocial

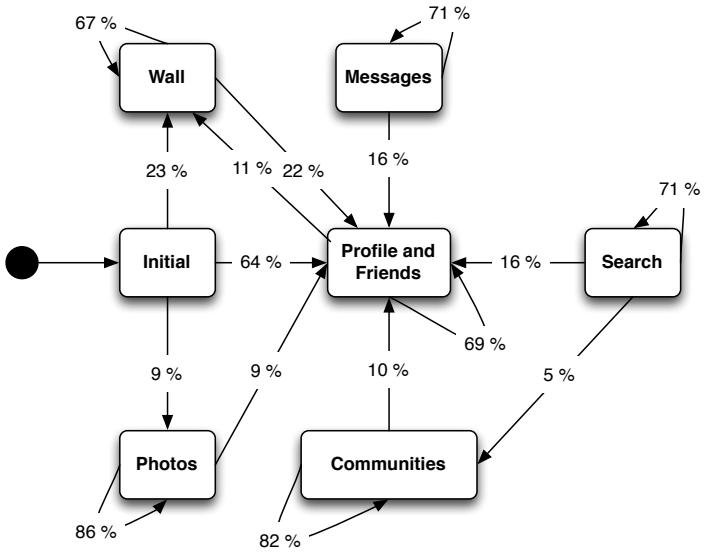


Figure 5.6: Category Transition Probability for the LifeSocial categories by previous master thesis [6].

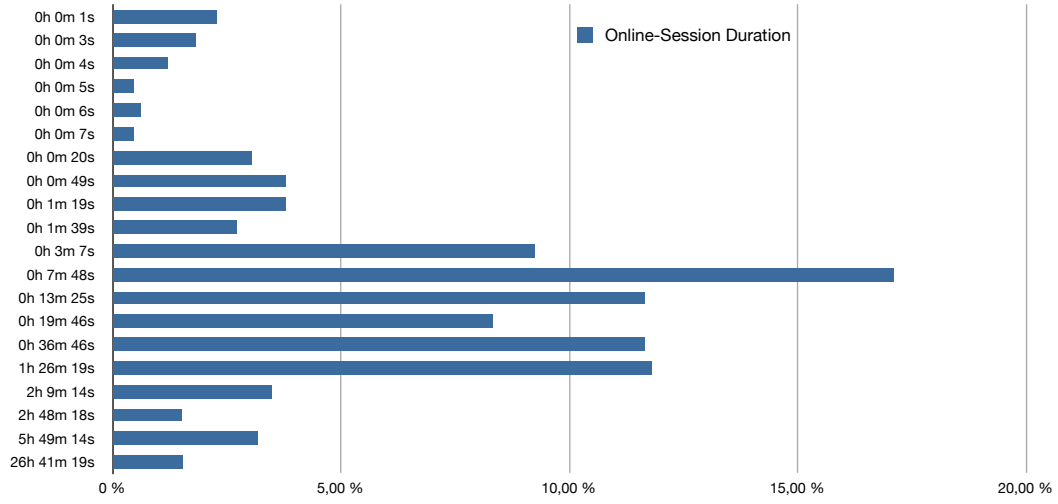


Figure 5.7: Total online distribution based on Bavenuto et al. [4]

5.6 Alpha Test

The alpha test of LifeSocial aims at letting a group of users to use LifeSocial for one week to find the problems and generate the load on the network. So more bugs can be fixed and the quality of the network can be evaluated. It will be also a prepare work for the future beta test.

5.6.1 Deployment

In alpha test, a long existing network should be established so that every alpha test user can login to this network any time. Moreover the node in this network should be able to be rebooted quickly. There are 10 accessible virtual machines, so 7 of them are picked as master and backup nodes in alpha test. Each backup node logins by a specific name.

5.6.2 Testplan

The testplan in alpha test is real user actions. However since users do not who else are in the network, they are not able to find potential friends. A meet point for these users is master node’s wall. It is required that once every real user logins to network, they should

6 Results Evaluation

This chapter evaluates the results from different tests. Section 6.1 and Section 6.2 evaluates results of synthesis tests 25-instance and 100-instance. Section 6.3 and Section 6.4 evaluates random tests and random tests without logout. Section 6.5 evaluates result of long term test. Section 6.6 introduces lessons learned in alpha test.

6.1 Synthesis Test - 25 instances

6.1.1 Test Results

Figure 6.1 shows all nodes are initiated in the first about 15 minutes. After all nodes having being booted and registered in the network, `TestCaseSendFriendRequest` is sent to each user manually by test plugin. From about 23 minutes all instances started executing timed testplan and the value of `global network nodes` keeps 25 to the end. Logically this value should drop to 0 progressively after the testplan is finished. In Figure 6.1 it does not decrease in the end is because that, only the last stopped instance will record this descend of global value. Since the shut down of this test was done by multiple people, it is not possible to find the last instance in the network. However this descend of metric `global network nodes` does not matter too much. Figure 6.2 shows mean value of `Join Network Time` and `Registration Time`. `Registration Time` includes `Join Network Time` as the first executed part, the rest of `registration` is first to try to retrieve user's login item if the user has registered before, otherwise new user login item will be created. From this graph it is found that `Join Network Time` increases at the initiation phase. This ascend is because, it is known that in FreePastry before a new node is booted into network it has to perform some actions such as request and store leafset of other nodes. At the very beginning of the test, all nodes connect into network via master node only, moreover there are less nodes in the network so that new join node needs not to send requests to too many nodes. `Registration Time` increases at the initiation phase too, because except the join network part, with the increasing number of nodes in the network, to try to retrieve its login item requires more and more time. The most difference between `Registration Time` and `Join Network Time` is about `1s`.

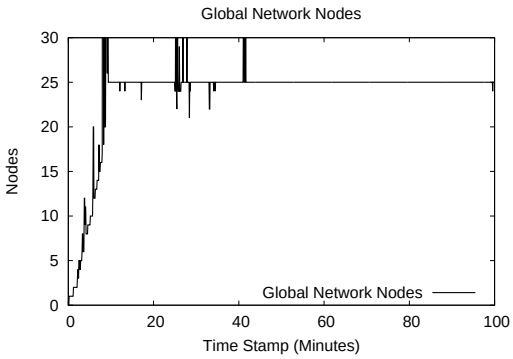


Figure 6.1: Global Network Nodes.

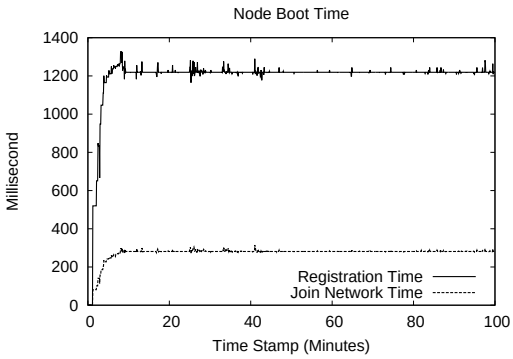


Figure 6.2: Global Node Boot Time.

Figure 6.3 shows global stored elements and Figure 6.4 shows max, mean and min values of it. The replication factor is 5. In both two figures, there are 5 ascends: before and after 20 minutes, 50 minutes, 70 minutes and 75 minutes. These ascends are caused by both store and retrieve actions in the network. Such as highest ascend before 80 minutes which caused by `TestCaseUploadPhoto`. After that, there is also a small ascend after 80 minutes which is caused by `TestCaseViewFriendsPhoto`. Totally 14000 elements are stored in the network. On average each node stores 600 elements, which is 4% of total. The node which is responsible for most stored elements stores nearly 750 elements, which is 5% of total. The node with least elements stores about 500 elements, which is 0.3% of total. The difference of stored elements between most and least nodes is 250 elements, which is 1% of total.

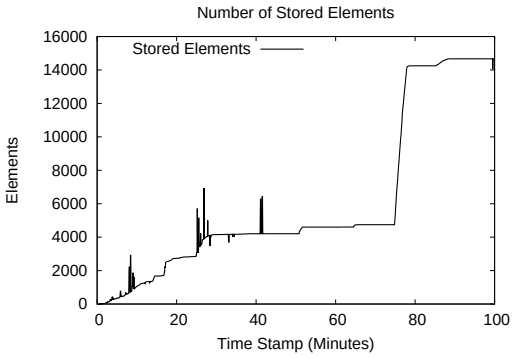


Figure 6.3: Global stored elements.

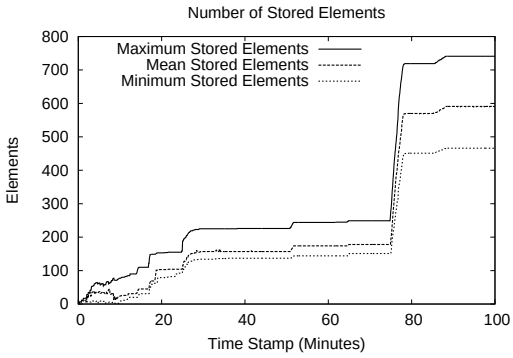


Figure 6.4: Global Stored elements - Max Mean Min.

Figure 6.5 shows global received `InsertMessage` and Figure 6.6 shows max, mean and min values of it. Once an `InsertMessage` is received by a certain node, it supposes to stores a replica. Thus the ascend in these two graphs

can only be caused by a store action in the network so that there are less apparent ascends compared to Figure 6.3 and Figure 6.4. However, the number of `InsertMessages` is keeping increasing all the time, it means that besides the several apparent ascends which are caused by the store actions made by test cases, FreePastry is sending `InsertMessges` all the time. Totally almost 200000 `InsertMessages` are sent, on average each node receives about 8000 `InsertMessages`. In Figure 6.6 shows that the difference between max and min value of received `InsertMessages` is huge, which is 15000-2000=13000 messages, 6.5% of total. This gap seems to be getting bigger and bigger if test continues. It means some nodes in the network are always chosen to send `InsertMessage` which will cause some nodes overflowed finally. In addition, it might be also caused by the fact that, 25 instances actually forms a small ring in the network and replication factor is a little bit large compare to this ring. Since each stored object has to be replicated 5 times, some nodes might be often chosen as the target to store replicas.

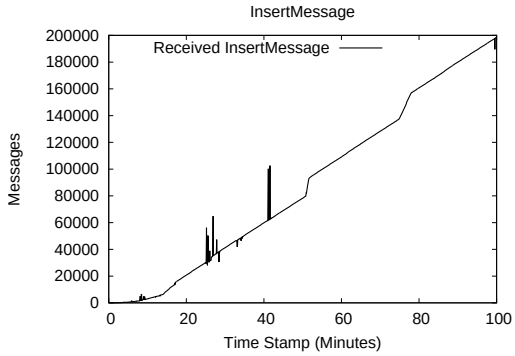


Figure 6.5: Global Received InsertMessages.

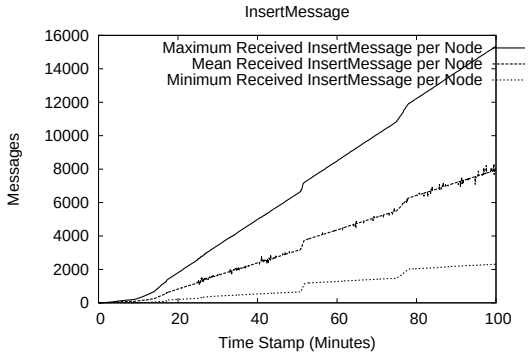


Figure 6.6: Global Received InsertMessages - Max Mean Min.

Figure 6.7 shows the total storage space and used storage space, it is clear that the storage space is almost empty and the used storage space can be barely seen. In Figure 6.8 shows the used storage in max, mean and min. It can be found that even `InsertMessage` is not evenly sent, the difference between most and least stored elements and used storage space are not increasing.

Figure 6.9 and Figure 6.10 show global framework sent and received messages respectively. Both graphs are in the same figure, even the quantities are the same. It is due to every sent message leads to a response messages, thus it means that there is no message lost during communication. In both graphs there are some rapid ascends, it is due to the wall related test cases and `TestCaseForum`. Since these test cases implement `DDS` function, it requires more messages exchange than usual. On the contrary, all the photo related test

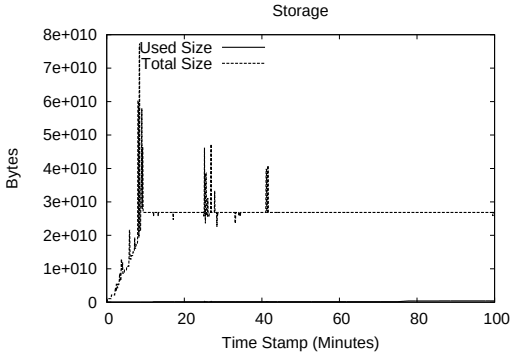


Figure 6.7: Global Used Storage and Total Storage.

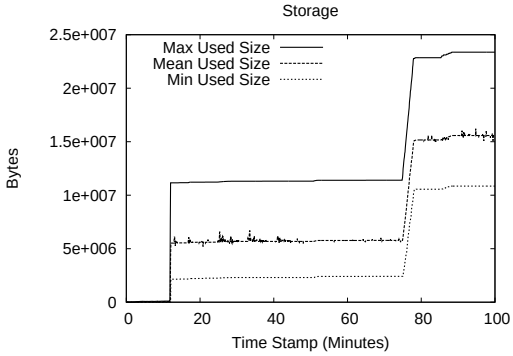


Figure 6.8: Global Used Storage - Max Mean Min.

cases in the last half phase do not produce too many messages in framework layer.

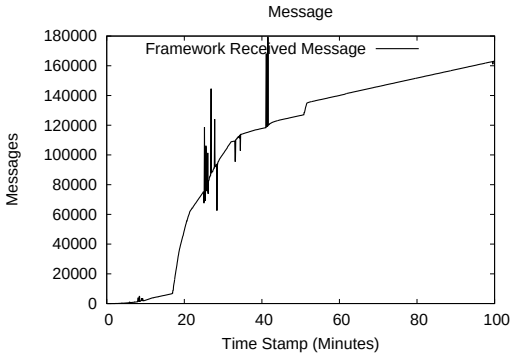


Figure 6.9: Global Framework Received Messages.

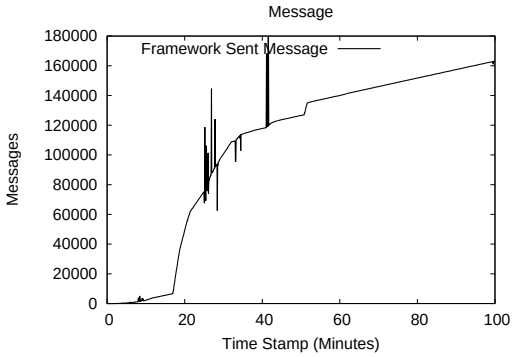


Figure 6.10: Global Framework Sent Messages.

Figure 6.11 and Figure 6.12 shows pastry received and sent messages respectively. Similarly, the figure and quantities of two graphs are same. Compared to framework received and sent messages, Freepastry started exchanging messages from the very beginning. Till about 17 minutes the curve of both graphs increases rapidly, and this rapid ascend conforms the first 17 minutes of Figure 6.1 when nodes login to network. Since when nodes login to network, they need to exchange their leafset and replicate the object they stored and so on, which generates a large amount of message exchanges. Another two rapid ascend is first after all nodes login, the other one is after 60 minutes when nodes started creating photo albums, in this test case each node has to first load its existing photo albums which result in sending out many `LookupMessages`. As shown

in Figure 6.13 and Figure 6.14, exactly the same that after 60 minutes, number of received `LookupMessages` has a rapid ascend. Similarly, number of sent `LookupMessages` is as same as received messages. It is noted that the figure of `LookupMessages` is as same as `PastryMessages`. t means that `LookupMessages` is one of the main reason that generates data traffics in message exchange of Freepastry. In the end of test, the totally sent/received `LookupMessages` is 12% of totally sent/received `PastryMessages`.

The messages sent/received by framework in the end is 4% of pastry sent/received messages, since Freepastry is sending around many regular messages to keep the connection of network. Moreover, this percentage mainly depends the insensitivity of the testplan. But it still can be found from these 4 graphs that framework indeed controls the trend of message traffic in Freepastry. As in Figure 6.11 and Figure 6.12 there is a short flat curve before 20 minutes, it is the time that all nodes have login to network no test case is being executed. Similarly in Figure 6.9 and Figure 6.10 at the same time, there is not too many messages are sent by framework. After the testplan is executed, the first executed test is `TestCaseChangeProfileInformation` which requires to retrieve profile item in the network. It looks that in framework layer this test case does not produce too many traffic, however in pastry layer it indeed makes a rapid ascend in the number of sent messages. After that whenever there is lookup actions performed by test cases, there will be an ascend in the graph of `PastryMessage` and `LookupMessage`.

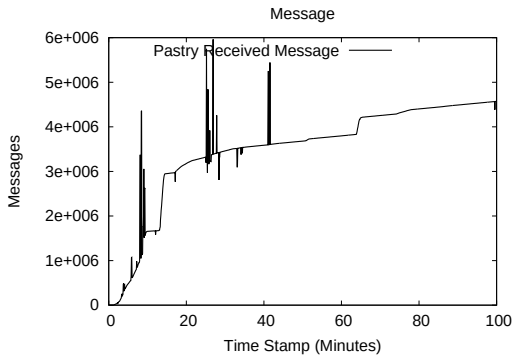


Figure 6.11: Global Pastry Received Messages.

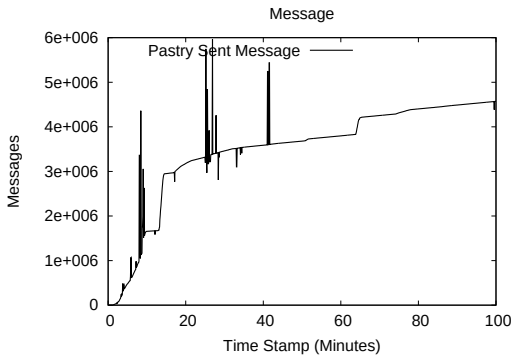


Figure 6.12: Global Pastry Sent Messages.

Figure 6.15 and Figure 6.16 shows framework and pastry received message rate. Since sent message rate of these two are same as received rate respectively, it is not shown here. In Figure 6.16, in the login phase of test, the message rate is very high, since nodes are exchanging many network maintenance messages. In framework layer, message rate gets high when the `DDS` related

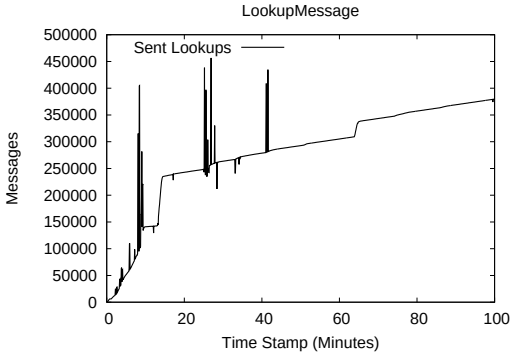


Figure 6.13: Global Sent LookupMessages.

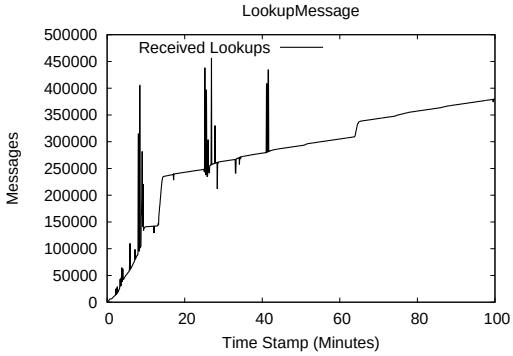


Figure 6.14: Global Received LookupMessages.

test cases are executed.

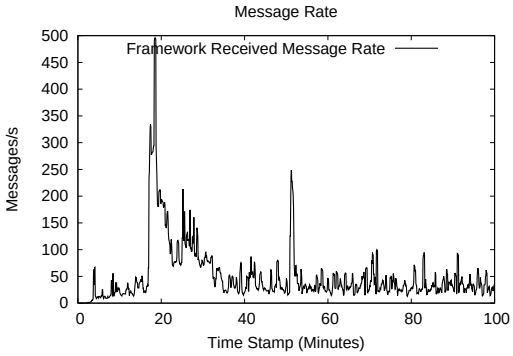


Figure 6.15: Global Framework Received Messages Rate.

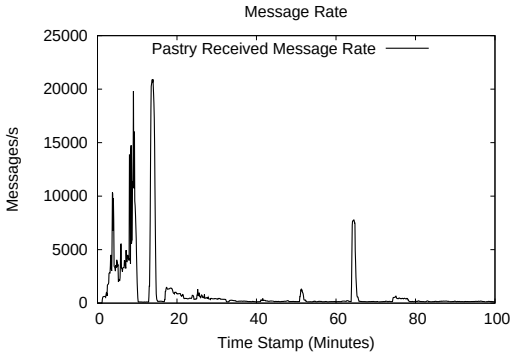


Figure 6.16: Global Pastry Message Received Rate.

In framework layer storage module, two main actions can be performed: `store` and `retrieve`. In Figure 6.17 and Figure 6.18 shows mean `store` and `retrieve` time. In the beginning of test, the highest peck in both graphs is caused by `TestCaseChangeProfilePicture`. However in the last half phase of test there seems to be all objects can be retrieved or stored in very short time. It is because that the photos uploaded by photos plugin has been scaled by image scaling tool. The size of the photo after being scaled is only around 30 kbytes. Compared to this size, the monitoring message sent around every 5 seconds is even 28 kbytes. Therefore it is normal to see that the store or retrieve time does not increase in the end. However, the retrieve time increases during 20 to 50 minutes. During this period the `DDS` related test cases are being executed. According to the mechanism of `DDS`, data items which belong to the same cate-

gory are stored among multiple nodes in a tree structure. To retrieve a certain item may require requesting multiple child nodes in the tree. Thus it can be found that DDS introduces some inefficient in the network.

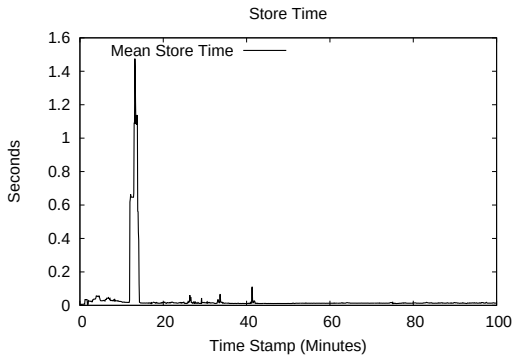


Figure 6.17: Global Mean Store Time.

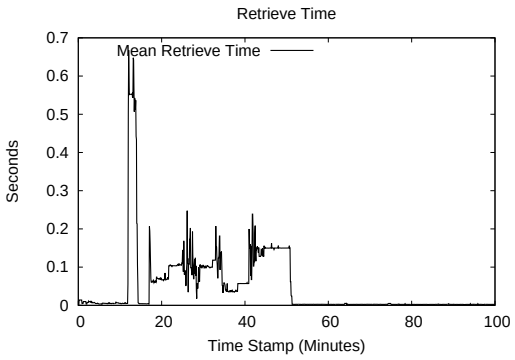


Figure 6.18: Global Mean Retrieve Time.

Figure 6.19 and Figure 6.20 show the stored and retrieved items in the network. The total number of retrieved items are much more larger than stored items. Since DDS is also using the retrieve function in StorageDispatcher, it might due to this reason that more items are retrieved than stored. In total 60000 items are stored and about 430000 items are retrieved. The retrieved items are about 7.1 more than stored items.

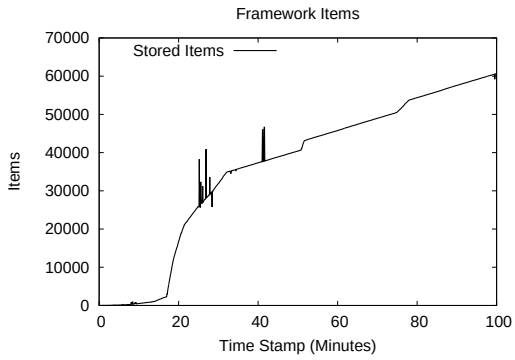


Figure 6.19: Global Stored Items.

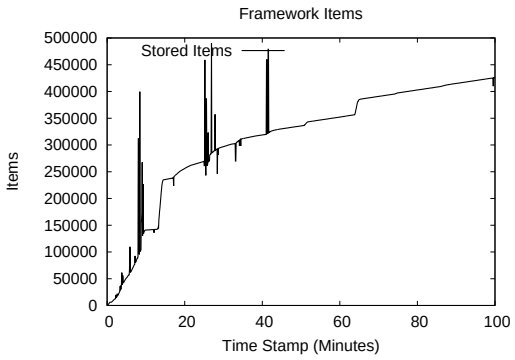


Figure 6.20: Global Retrieved Items.

Similarly, global lookup time basically follows retrieve time. Since the retrieve function in framework layer always invoke the `lookup` function in PAST. As Figure 6.21 shows the large file and DDS introduce more delays in lookup request. But the longest lookup time is only 250ms $\approx$ 0.25s which is totally durable. Figure 6.22 shows missed object size calculation from all nodes. The figure illustrate the sum of missed calculations. Still the curve increases rapidly during 20 and 40 minutes when DDS related test cases are being executed. It is due to

DDS requires more message exchanges and each message has to be measured size before being sent out, thus much memory is consumed. In case memory consumption overflows quota, it is designed that whenever FreeMemory is less than 0.25 of total memory, the object size calculation is omitted. In photo related test cases there is no missed calculation. In total there are only 14 missed calculations, on average 0.56 to each instance.

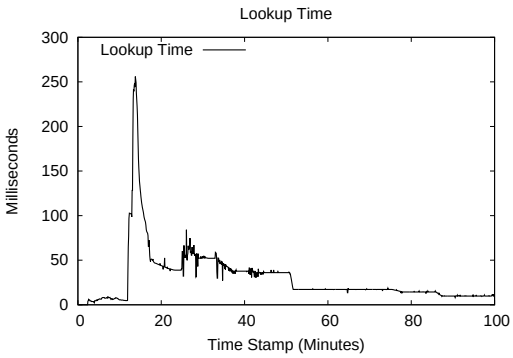


Figure 6.21: Global Lookup Time.

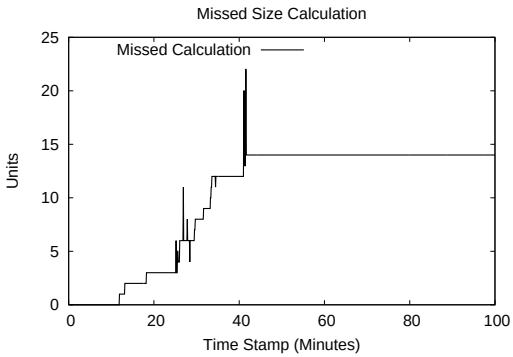


Figure 6.22: Global Missed Size Calculations.

Figure 6.23 and Figure 6.24 shows framework data send and framework data received respectively. Inside framework, two modules are mainly responsible for passing data from upper layer to overlay, they are: StorageDispatcher and MessageDispatcher. In these two figures the two StorageDispatchers' curves are not same, it is because that to retrieve a file is not lead by a stored action. Similarly, to store a file does not mean another node will retrieve it. It can be found that most data traffics is generated by MessageDispatcher in both figures, since there is always messages sent around, such as monitoring messages which will be sent and received and each message is of size 29 kbytes. Under this testplan, received data traffic generated by MessageDispatcher is 86% of total and StorageDispatcher is 14% of total. Sent data traffic generated by MessageDispatcher is 88% of total, and StorageDispatcher is 12% of total. In framework data read figure it is noted that read data traffic increasing quickly when DDS related test cases are executed. It means DDS function requires to retrieve a large amount of data, and to retrieve these data many messages are read. After that, the messages almost keep horizontal to the end. It is interesting to find that, even the numbers of received and sent message in framework layer are same, the size of them are different. As shown in these two figures, the final sizes of total received and sent data by MessageDispatcher are different, received data is 4.75 times larger than sent data. From the tendency of the curve MessageDispatcher Data Received it is observed that, at

the beginning it keeps steady and later it increases quickly when DDS related test cases are being executed. The reason is not clear but it worth to be studied in future work.

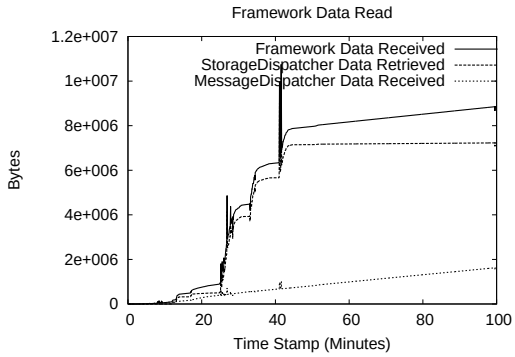


Figure 6.23: Global Framework Data Received.

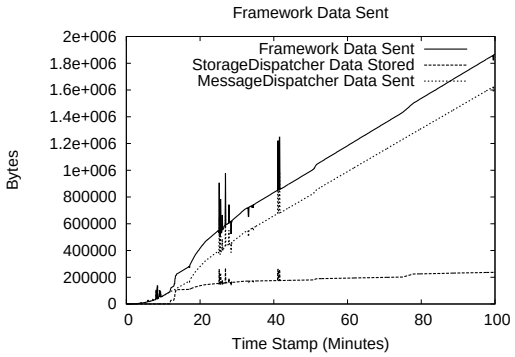


Figure 6.24: Global Framework Data Sent.

Figure 6.26 and Figure 6.25 show sent and received data rate of MessageDispatcher and StorageDispatcher. Figure 6.27 and Figure 6.26 show received and sent data rate of framework. It is clear that received data rate is always higher than sent data rate, even though the numbers of received and sent messages are same.

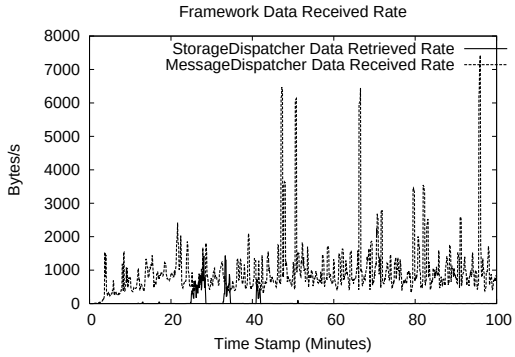


Figure 6.25: Global Framework Modules Data Received.

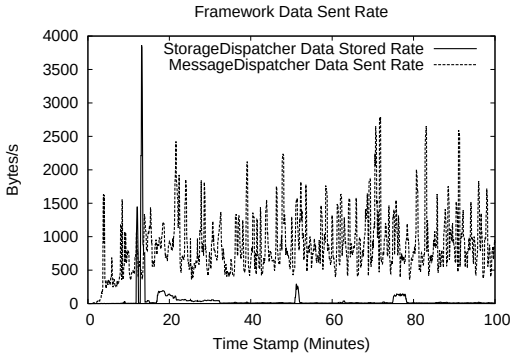


Figure 6.26: Global Framework Modules Data Sent.

Figure 6.29 and Figure 6.30 illustrates data sent and read by framework and FreePastry. It is found that compared to FreePastry data sent and read, framework just produces few data traffic, since FreePastry is keep sending around messages to keep the connection of network and in this test, framework does not produce too much data traffic. The received data by framework is 14% of

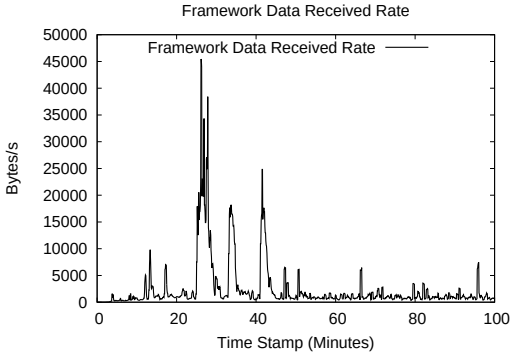


Figure 6.27: Global Framework Data Received Rate.

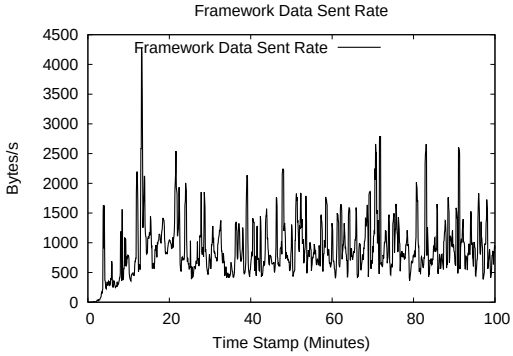


Figure 6.28: Global Framework Data Sent Rate.

FreePastry received data and sent data is 1% of FreePastry.

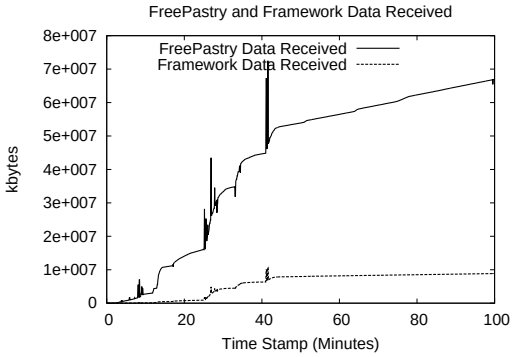


Figure 6.29: Global Framework and FreePastry Data Received.

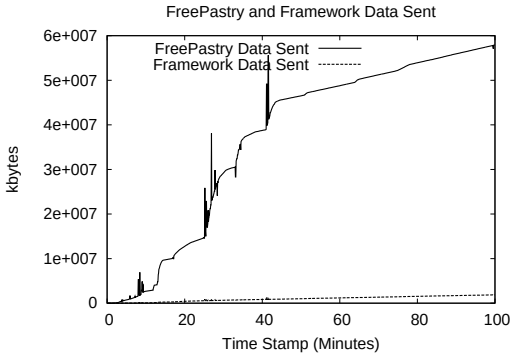


Figure 6.30: Global Framework and FreePastry Data Sent.

The data sent and read (if necessary) by each module above `MessageDispatcher` and `StorageDispatcher` is also monitored. As shown in Figure 6.31 and 6.32 monitored global data sent and read of `DDS` and `Photos Plugin`. In DDS data received is larger than data sent which conforms what is observed in framework data received and data sent. In DDS received data is 10MB larger than sent data. In photos plugin the sent data is much larger than received data since photos plugin's sent data test case `TestCaseUploadPhoto` is executed 50 times and `TestCaseCreatePhotoAlbum` is executed 20 times, but receive data test case `TestCaseViewFriendsPhoto` is only executed 20 times in the end.

Some social metrics are monitored too, as shown in Figure 6.33 and 6.34 sent friend requests and created photo album. Both basically conform to what is

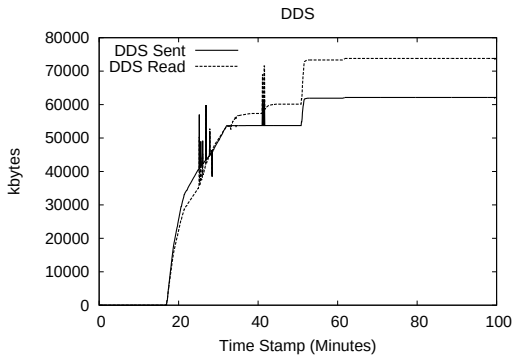


Figure 6.31: Global DDS Data Sent and Read.

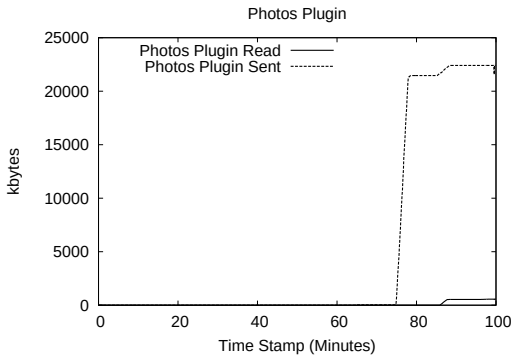


Figure 6.32: Global Photos Plugin Data Sent and Read.

designed in testplan.

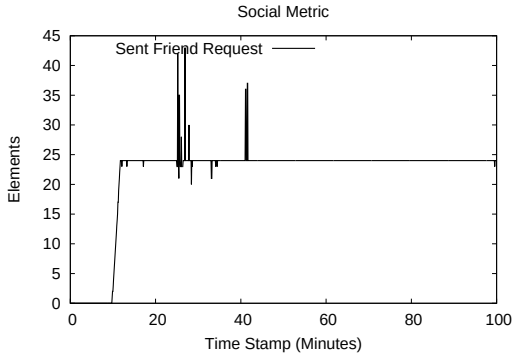


Figure 6.33: Global Number of Friend Request.

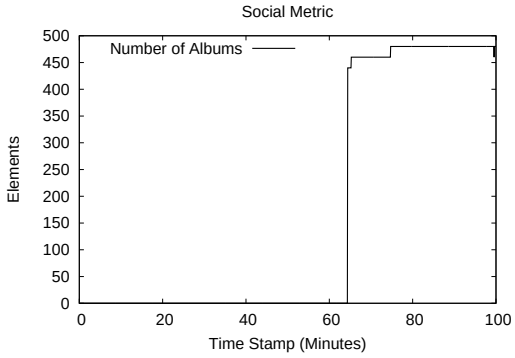


Figure 6.34: Global Create Photo Albums.

6.1.2 Conclusion

In this 25-instance synthesis test, the functionality of each test case is tested and the result is successful. By evaluating result graphs, accuracy of monitoring function is tested by analysis the activities in the network. Relationships between different modules and layers are studied. Characteristics and strange parts are found. All in all, framework behaviour of a small scale 25-instance network is monitored and analyzed in this test.

6.2 Synthesis Test - 100 instances

This section compares the test results of 100 instances with 25 instances'. Since the relationship between testplan and test results has been explained in last section, it is more interesting to observe the change of network behaviour from a larger scale setup, 100 instances. Same testplan is used in this test.

6.2.1 Test Results

Figure 6.35 and Figure 6.36 shows global network nodes and global boot time. In global network nodes the curve keeps value 100 until 100 minutes, then all nodes logs out thus the curve should drop down. However, first, since all nodes logs out very quickly, the monitoring mechanism cannot react this very quick change just before the end of test; second the historical information is used in current monitoring mechanism so that it even slows the reaction of the changes in the network. Therefore the final value of global network nodes keeps 100. Compare to Figure 6.1, it is found that the random impulses of both graphs occur at roughly the same position, which means `DDS` function indeed cumbers monitoring function. The global boot time graphs shows nearly same result as result from 25-instance test Figure 6.2, except the initiation phase. Since in a 100-instance test it requires more time to start all instances, there will be more instances user other nodes except `master` to finish registration. Both curves in global node boot time indicate the mean value, thus in a 25-instance test it might that almost all instances are booted from master. However increase the scale of test 4 times and the increased boot time is only about 0.1 second which shows the boot function of LifeSocial is scale.

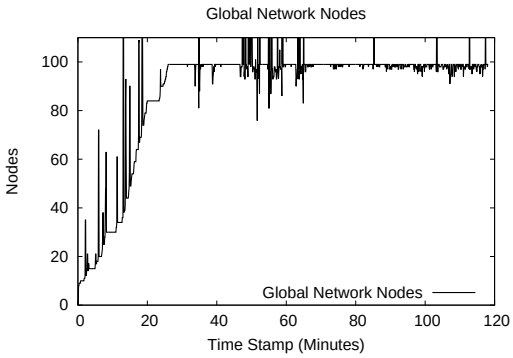


Figure 6.35: Global Network Nodes.

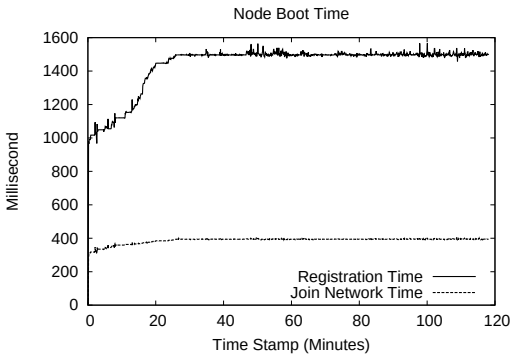


Figure 6.36: Global Node Boot Time.

Figure 6.37 and Figure 6.38 show global stored elements and its max, mean and min value. Compared to Figure 6.3 final number of global stored elements in a 100-instance network is 60000 elements more, increased 4 times, which

conforms to a 4 times larger network. Figure 6.38 shows the mean stored elements in each node keeping same value as in a 25-instance network which is around 600 elements. The values of max and min are different from 25-instance test. The max value is increased 100 elements more while the min value is decrease 100 elements less. However by comparing the result from a 50-instance test, which is attached in the appendix, this differences between max and min values of different tests are totally random.

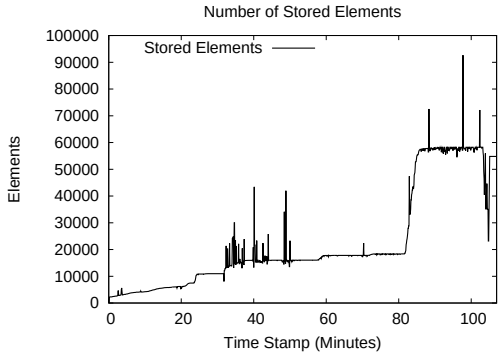


Figure 6.37: Global stored elements.

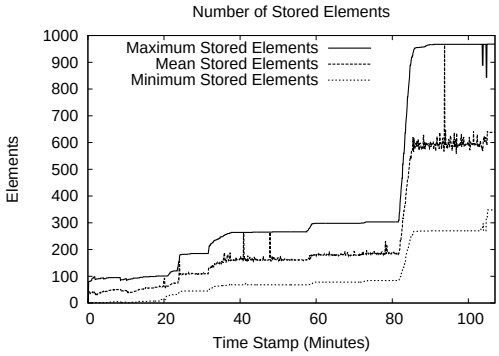


Figure 6.38: Global Stored elements - Max Mean Min.

Figure 6.39 and Figure 6.40 show global used storage and its max, mean and min curves. It is found that, compare to total storage only few storage is used, since this testplan does not generate too much storage. The mean storage used by each node is around $2x10^7$ which is a little bit more than the result from 25-instance test, but totally reasonable.

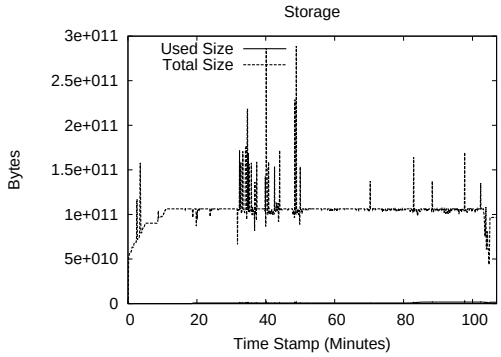


Figure 6.39: Global Used Storage and Total Storage.

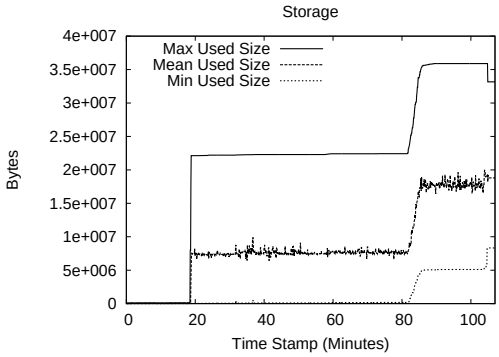


Figure 6.40: Global Used Storage - Max Mean Min.

In Figure 6.41 and Figure 6.42 they show the stored and retrieved items in the

network. It conforms to the result of 25-instance test that retrieved items are more than stored items. this time number of stored items is 230000 and of retrieved items is $1.7x10^6$. Thus the number of retrieved items is 7 times more than stored items. This result conforms to the result from 25-instance test.

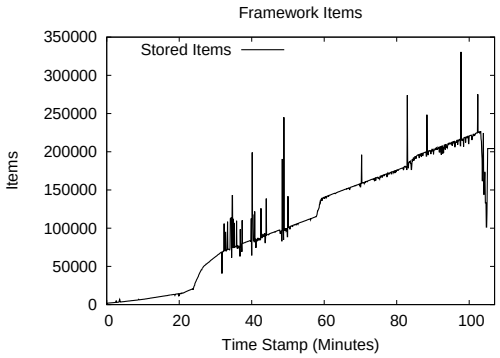


Figure 6.41: Global Stored Items.

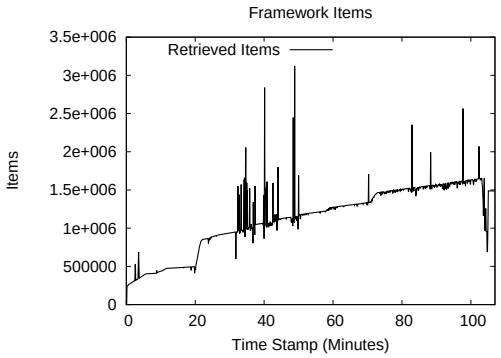


Figure 6.42: Global Retrieved Items.

Figure 6.43 and Figure 6.44 show the globally received `InsertMessage` and its max, mean and min value. It is found that received `InsertMessage` is 5 times more than what is received in 25-instance test. Moreover, the max and min curves show that the difference between the most received and least received nodes are getting bigger and bigger, as time and size of network increases. It means some nodes are ignored in sending replicas to them.

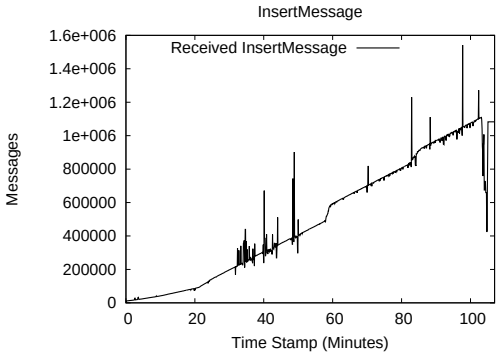


Figure 6.43: Global Received InsertMessages.

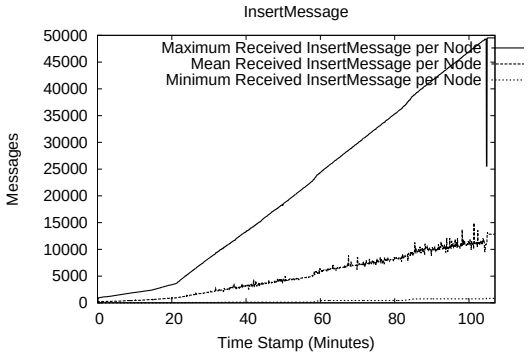


Figure 6.44: Global Received InsertMessages - Max Mean Min.

Figure 6.45 and Figure 6.46 show global received and sent messages by framework layer. Same number of messages are sent and received globally. $5x10^5$ messages are sent and received, compared to 25-instance test, 3 times more messages are sent which means increase the size of network 4 times results in

3 times more messages by framework. The trend of the curve basically follows Figure 6.10 and Figure 6.9 in 25-instance test. Similarly, in the wall related test cases, many random impulse occur. Compared to that, less impulses occur in photo related test cases.

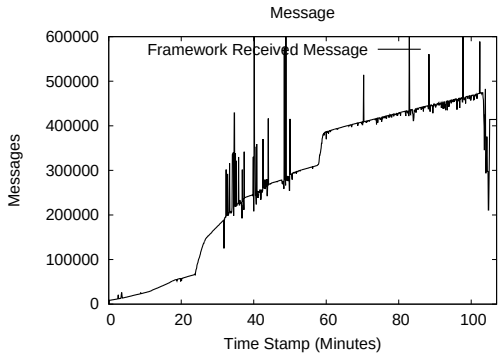


Figure 6.45: Global Framework Received Messages.

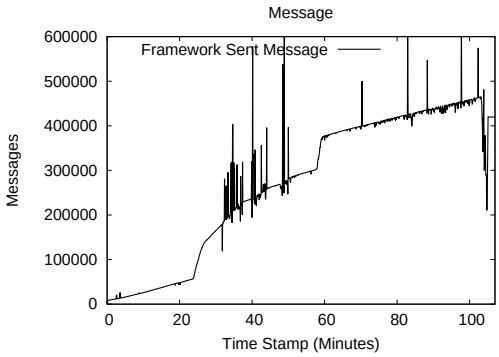


Figure 6.46: Global Framework Sent Messages.

Figure 6.47 and Figure 6.48 show global message received and sent by FreePastry. The amounts of sent and received messages are same which means there is no message lost during the test. Compared to 25-instance test, 4.44 times more messages are sent and received by FreePastry which approximately equals to the increased size of network.

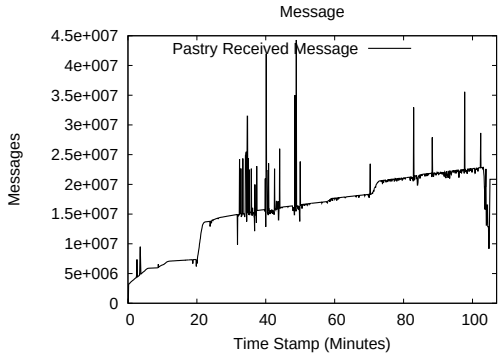


Figure 6.47: Global Pastry Received Messages.

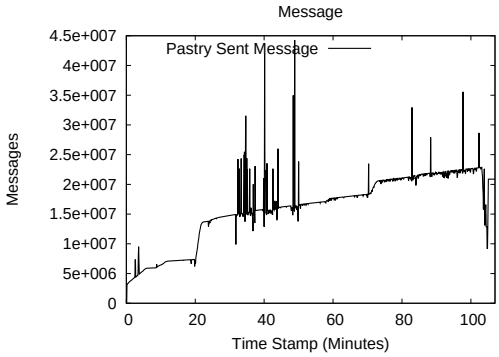


Figure 6.48: Global Pastry Sent Messages.

Figure 6.49 and Figure 6.50 show the sent and received `LookupMessages`. The numbers of received and sent `LookupMessage` keep same. Compared to the result from 25-instance test, the number of `LookupMessage` sent or received is 4.5 times more which approximately equals to the increased size of network.

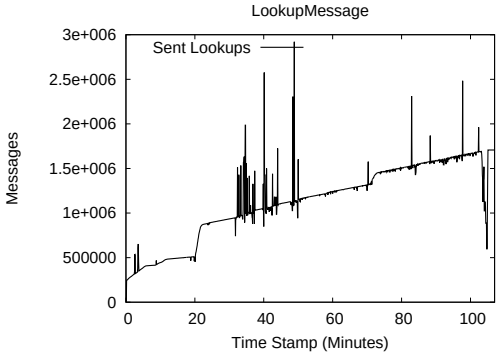


Figure 6.49: Global Sent LookupMessages.

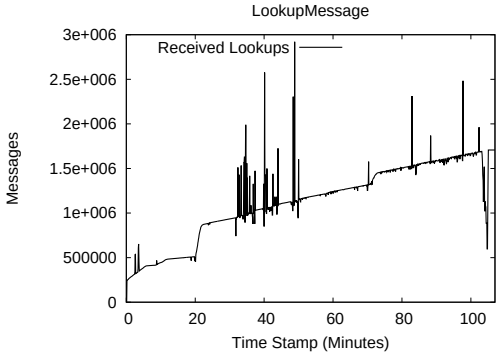


Figure 6.50: Global Received LookupMessages.

Figure 6.51 and Figure 6.52 show the message rates of framework and pastry respectively. Compared to the result of 25-instance, the figure of these two graphs are almost totally same. Only the message rate is roughly doubled in 100-instance test.

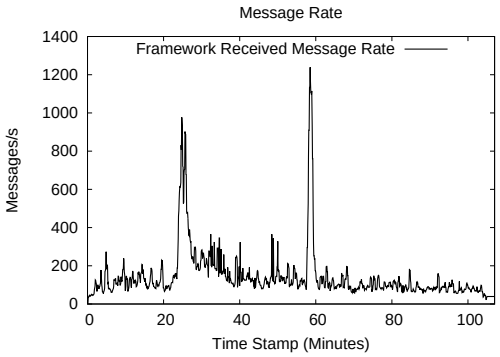


Figure 6.51: Global Framework Received Messages Rate.

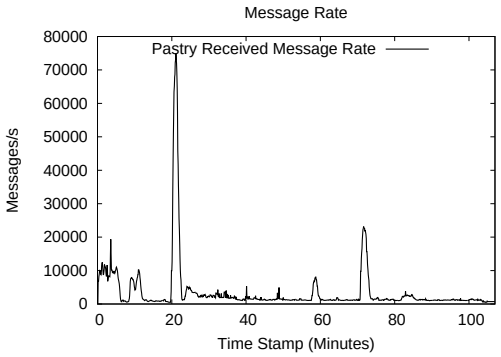


Figure 6.52: Global Pastry Received Messages Rate.

Figure 6.53 and Figure 6.54 show $mean$ store and retrieve of a certain item time in the network. The figure of curves of these two graphs conform to the result of 25-instance test. However, in the store time graph, the highest peak which is caused by change profile test case is twice larger than the result of 25-instance test. Except this peak, the cost of retrieve time is only a little bit higher than 25-instance test's, almost the same. Similarly retrieve time in this test has a peak as high as 1.8 second and except the peak it is a little bit higher than 25-instance test's result. On average, while the network size is 4 times larger, to store and retrieve a data object cost twice time more

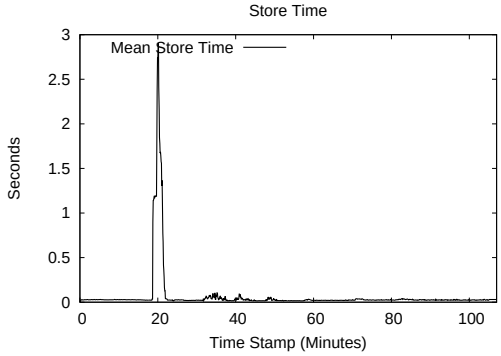


Figure 6.53: Global Mean Store Time.

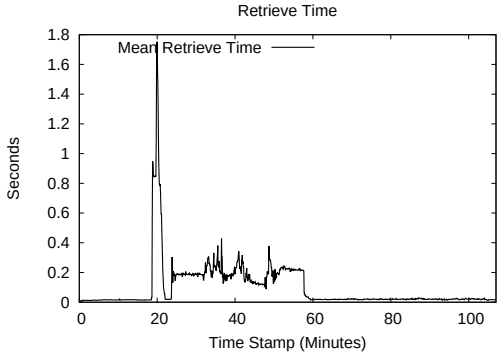


Figure 6.54: Global Mean Retrieve Time.

Figure 6.55 and Figure 6.56 show global mean lookup time and missed size calculation. The global lookup time follows the figure of the curve from 25-instance test, and the value is roughly the same. Combine the result graph of retrieve time in this test which is twice long than the result of 25-instance test, the reason that the lookup time is not increased by the retrieve time is increased might be: first, the message is enqueued in overlay not returned to framework layer immediately; second, the object size calculation time is included in this retrieve time calculation, thus it might because in 100-instance test, multiple instances are running on same machine which results in faster memory consuming and it leads to that the object size calculation requires longer time, but this calculation is limited to be within 40ms, otherwise the measurement thread is interrupted. However it still shows that the lookup function is scale even the network size is 4 times larger. The missed size calculation is in total 250 and on average each instance encounters 2.5 missed calculation which is 2 more times than 25-instance test's. It proves the assumption above that memory consume more in this 100-instance test.
Figure 6.57 and Figure 6.58 show framework data received and sent, including two modules : `StorageDispatcher` and `MessageDispatcher`. Still the figure of curves conform the result of 25-instance test, the amount of data is much more than 25-instance result's. The total data received by `Framework`, `MessageDispatcher` and `StorageDispatcher` is 5 times more than 25-instance result. The total data sent by `Framework`, `MessageDispatcher` and `StorageDispatcher` is 4 times more than 25-instance result.
Figure 6.59 and Figure 6.60 show the compare of `Framework` and `FreePastry` data received and sent. These graphs has the same figure as the result of 25-instance test. Still, the data generated by framework does not occupy too much in all the data sent or received by FreePastry. The total data received and sent by FreePastry is increased fivefold compare to 25-instance result.
It is interesting to find that, the increased size of network also leads the in-

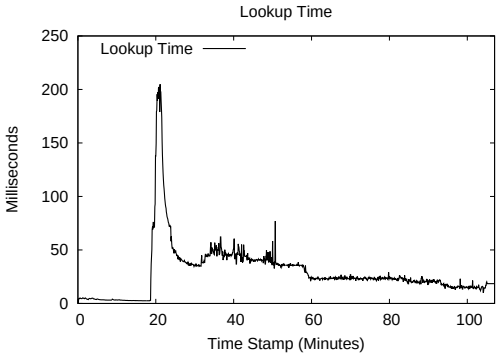


Figure 6.55: Global Lookup Time.

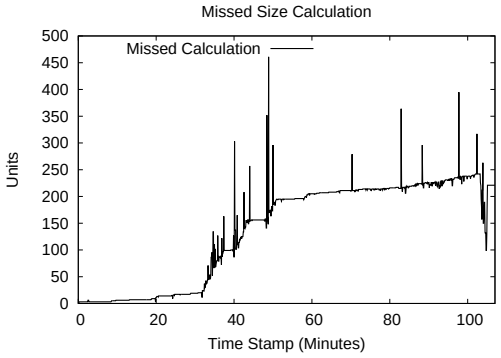


Figure 6.56: Global Missed Size Calculations.

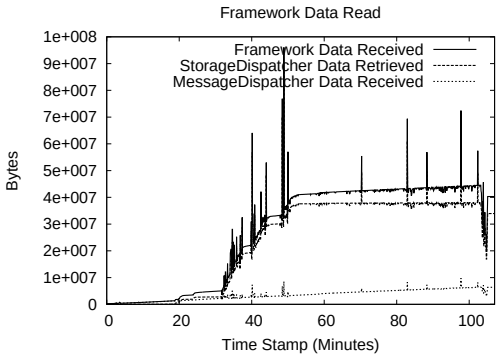


Figure 6.57: Global Framework Data Received.

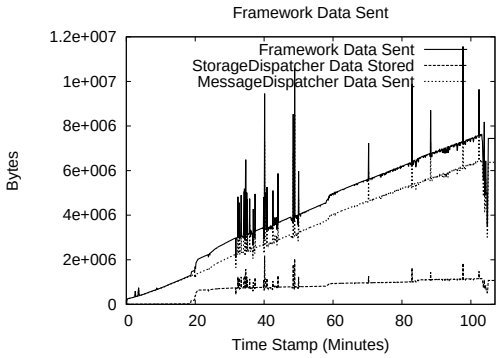


Figure 6.58: Global Framework Data Sent.

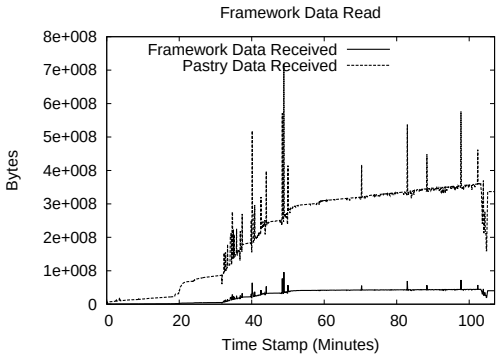


Figure 6.59: Global Framework and Pastry Data Received.

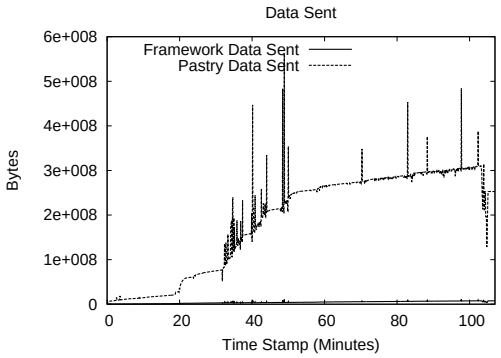


Figure 6.60: Global Framework and Pastry Data Sent.

creased sent and received data by each plugins. As shown in Figure 6.61 and 6.62, the data sent and received by `DDS` and `Photos Plugin` is increased compare to 25-instance result. The data sent and received by `DDS` is increased 3 times. the data sent and received by `Photos Plugin` is increased 4 times.

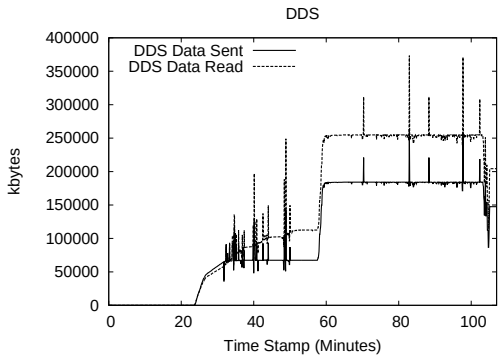


Figure 6.61: Global DDS Data Sent and Read.

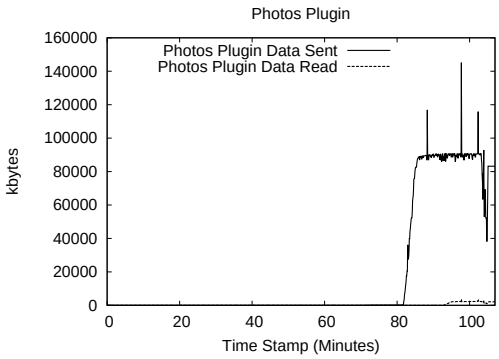


Figure 6.62: Global Photos Plugin Data Sent and Read.

6.2.2 Conclusion

By comparing 100-instance test result with 25-instance test result, it is found that the increased size of network leads to the increase of many metrics, such as sent or received message, data. This increase also has an impact on the framework layer, the data sent and received by framework is increased about 4 or 5 times. Moreover, plugin layer data traffic is also influenced, the increased size is around 4 or 5 too. But the increased size does not change the performance too much, such as the store and retrieve time, or lookup time, these metrics are also increased but still keeps a fairly good value.

6.3 Random Test - 25 and 100 instances

In this section, two random test results of 25 and 100 instances are compared. In a random test, each user stays in the network for a randomly picked online duration. The distribution of this online distribution is defined by Pascal based on the study of Benevenuto [4] which is shown in Figure 5.7. By observation during these two tests some nodes can be considered as $dead$, the behaviour or result of these $dead$ nodes can be defined into following categories:

Stuck in a certain test case Some nodes just stuck in executing a certain test case and do not throw any exception. These test cases

which cause instances stuck usually are `DDS` related test cases, plus `TestCaseChangeProfilePicture` and `TestCaseSendFriendRequest`.

Cannot login after logout Some nodes cannot login again after it logs out. Previously this problem is even worse, since the logout node will try to use the last bootstrap node to login, if the last bootstrap node is not in the network any more, it will consider master as bootstrap node. However this design causes some nodes to keep sending request to previous bootstrap node forever. Therefore now the design is the logout node will use master as bootstrap node directly. However it is still found that some nodes are stuck in login and keeping printing out in the console that `Messaging Plugin : You are not logged in`.

Full of MessageLostMessage Some nodes are having `MessageLostMessage` all the time. The scenario can be, while it is executing a certain test case which requires a certain data object, thus it sends out request for this object but all the nodes which have this object have disconnected from the network.
Another issue is, the current monitoring function is not perfect for random test. A main problem is it cannot detect the $dead$ nodes in the network. Further, the tree structure of monitoring mechanism cannot handle the churn perfectly.

6.3.1 Test Results

Figure 6.63 and Figure 6.64 show the network nodes graphs of random test 25-instance and 100-instance. In 25-instance test the random mode is enabled after 10 minutes. In 100-instance test the random mode is enabled after about 25 minutes. It is found that in 25-instance test the number of nodes drops slowly and gradually to 14 finally. In 100-instance the number of nodes drops to 23 immediately after the random mode is enabled and then this descend becomes slowly till the end. It is supposed to be the problem of `TestCaseSendFriendRequest`. Since this test case requires each slave to invoke the method in master's side to get a random slave's name, thus too many requests will cause problem and lead to slave stuck. Since in the beginning of the random test, each slave shall execute the start script, in this script `TestCaseSendFriendRequest` is executed at first, therefore there will be 100 requests at the same time. After the node number drops to 23, the network keeps steady and works normally for these about 20 to 40 nodes.
Figure 6.65 and Figure 6.66 show global join network time used in 25-instance test and 100-instance test respectively. First, it is clear that at the beginning, 25 or 100 nodes booted into network, the required time is negligible, and the curves are as same as synthesis 25-instance and 100-instance tests'. After random mode is enabled, the time required to join the network in both tests is

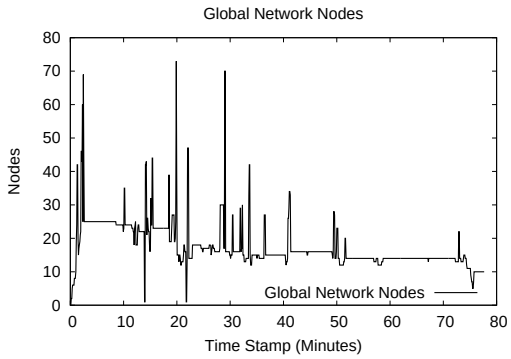


Figure 6.63: Global Network Nodes - 25 instance.

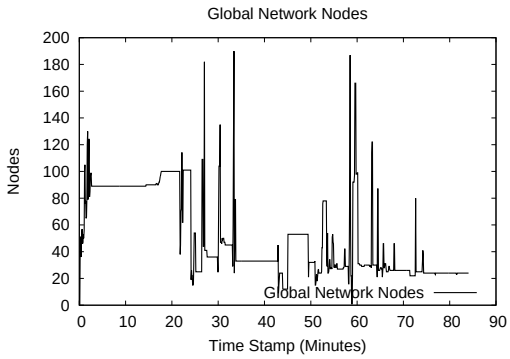


Figure 6.64: Global Network Nodes - 100 instance.

getting high. In 25-instance test the join network time is around 20 seconds and in 100-instance test it is getting higher and higher after 50 minutes, and the highest value is above 100 seconds which is fairly long.

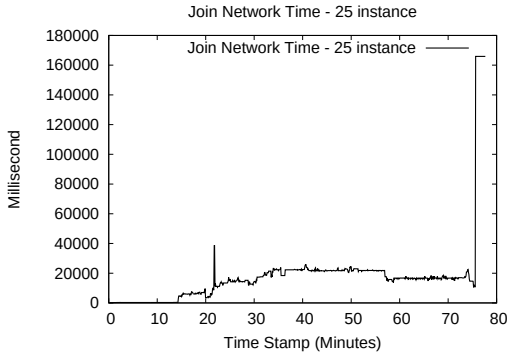


Figure 6.65: Global Join Network Time - 25 instance.

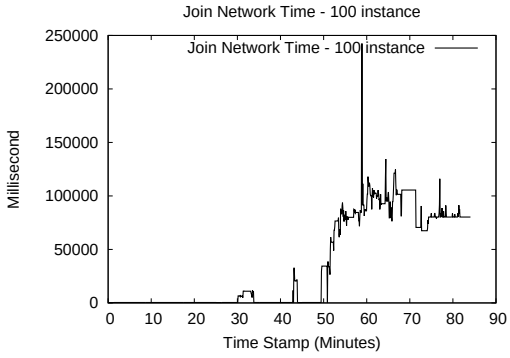


Figure 6.66: Global Join Network Time - 100 instance.

Figure 6.67 and Figure 6.68 show global retrieve item time of both tests. These rises of retrieve time is randomly located in both graphs since the test cases which requires to retrieve items are randomly picked to be executed. Compared to synthesis tests, retrieve time used in these two tests are much more longer, the longest one in 25-instance is 18 seconds and in 100-instance is 15 seconds. Since many nodes came and left network, to retrieve a data item might need to send requests to multiple responsible nodes in sequence resulting in requiring longer time. Similar results are found in global store item time in Figure 6.69 and Figure 6.70.

Figure 6.71 and Figure 6.72 show framework message sent and received during

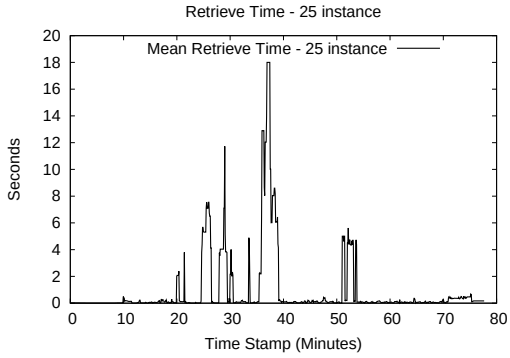


Figure 6.67: Global Retrieve Item Time - 25 instance.

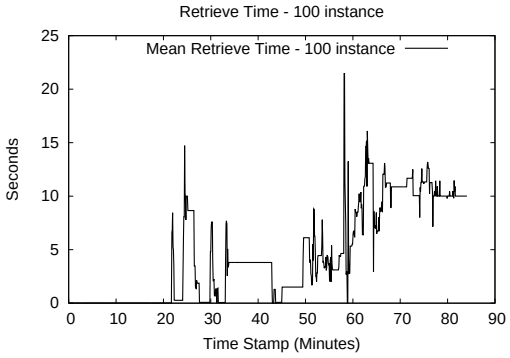


Figure 6.68: Global Retrieve Item Time - 100 instance.

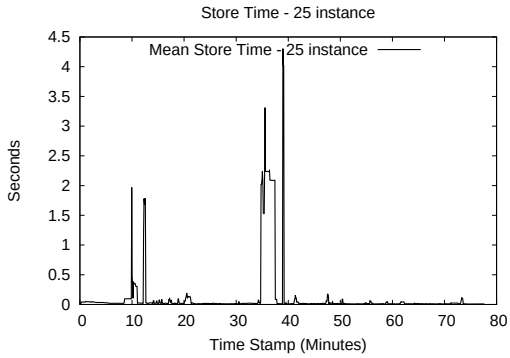


Figure 6.69: Global Store Item Time - 25 instance.

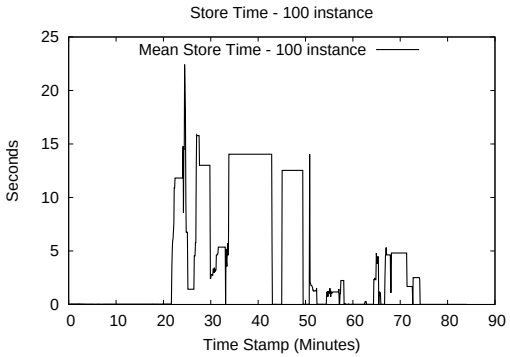


Figure 6.70: Global Store Item Time - 100 instance.

25-instance test. It is found the global messages sent and received keep same which means the monitoring function based on alive nodes in the network still functions correctly. Compared to synthesis 25-instance, framework in and out messages are much more less. The reasons are, first the random test time is 20 minutes short than synthesis test; second the test cases are not arranged as intensive as synthesis test's; third many nodes logout or crashed results in no more messages sent or received. Similar results can be found in 25-instance pastry received and sent messages in Figure 6.73 and Figure 6.74. The result in 100-instance test is same only the amount of sent and received messages in framework or FreePastry is larger.

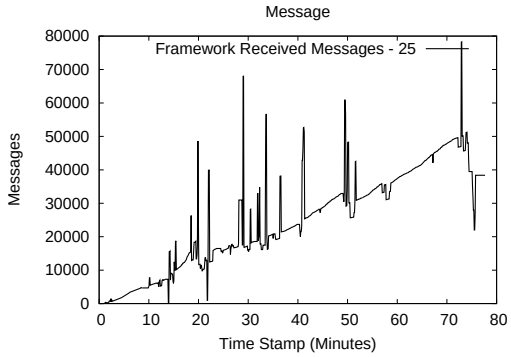


Figure 6.71: Global Framework Message Received - 25 instance.

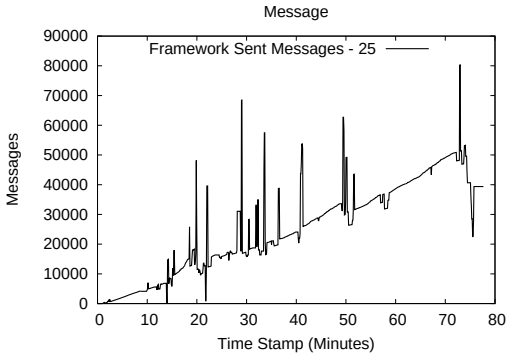


Figure 6.72: Global Framework Message Sent - 25 instance.

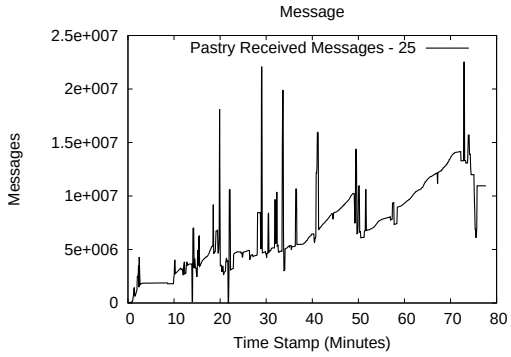


Figure 6.73: Global Pastry Message Received - 25 instance.

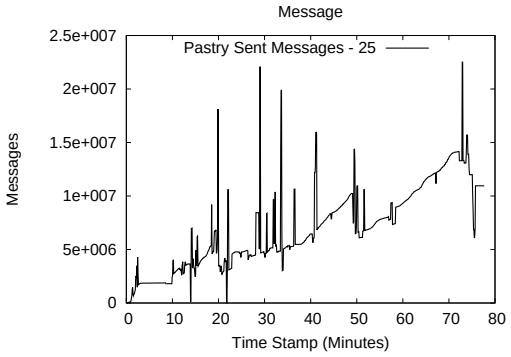


Figure 6.74: Global Pastry Message Sent - 25 instance.

Framework and FreePastry message received rate are shown in Figure 6.75 and Figure 6.76. Compared to synthesis 25-instance test results, framework message rate is decreased, the reason has been explained above, mainly due to

there is not too much data flow generated by framework. The message rate in FreePastry is almost same as results in 25-instance synthesis test, it is a little bit higher than that one, the reason can be more message-lost messages are sent. Same results are found in 100-instance tests in Figure 6.77 and Figure 6.76.

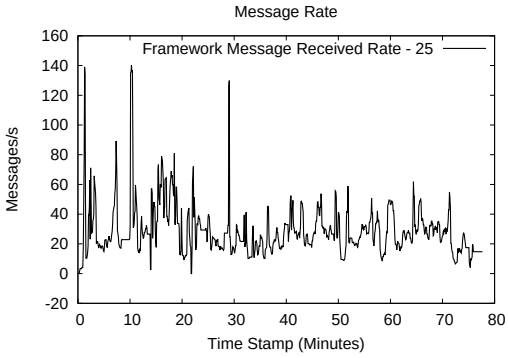


Figure 6.75: Global Framework Message Received Rate - 25 instance.

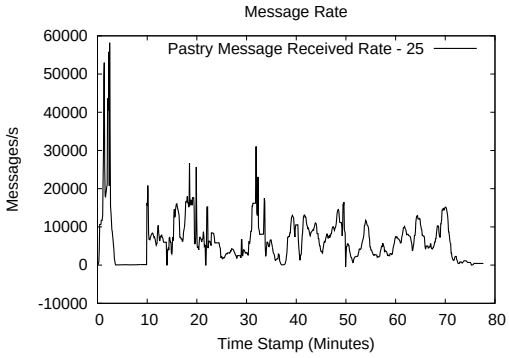


Figure 6.76: Global Pastry Message Received Rate - 25 instance.

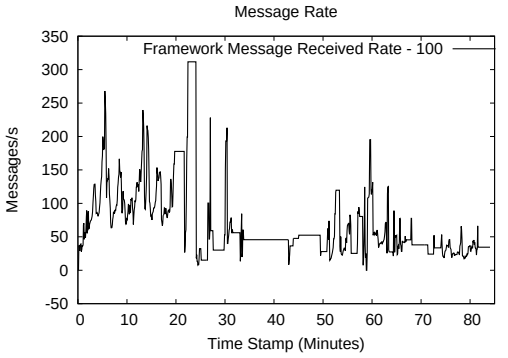


Figure 6.77: Global Framework Message Received Rate - 100 instance.

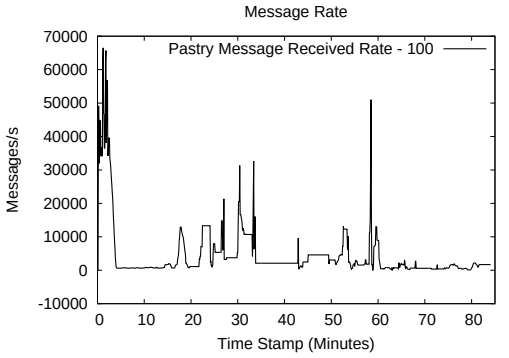


Figure 6.78: Global Pastry Message Received Rate - 100 instance.

Figure 6.79 and Figure 6.80 shown data sent and received by DDS in both 100 and 25 instances tests. It is found that data received by DDS is still more than data sent by it. Also both data sent and received in 25-instance network are larger than the results of 100-instance network.

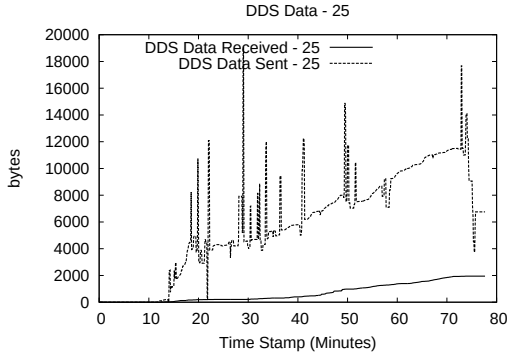


Figure 6.79: Global DDS Data - 25 instance.

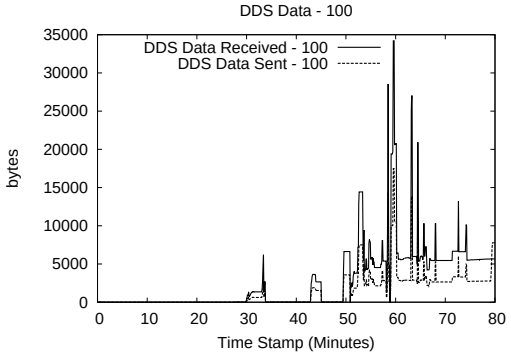


Figure 6.80: Global DDS Data - 100 instance.

6.3.2 Conclusion

In this section the results of 25-instance and 100-instance tests are compared with each other and with results of synthesis test 25-instance and 100-instance. It is found that even though there are nodes crashed due to send friend request test case at the beginning of the test in both two random tests, after that both network behaves normally, such as the numbers of framework and pastry sent or received messages are equal. Indeed friend request test case is the culprit of the main failure in both tests and it cannot be avoided in both tests. A better way to handle this is that, each node randomly picks a node in its routing table to send friend request to.

6.4 Random Test without Logout - 25 and 100 instance

This section introduces the random tests without logout. The goal of this test is to examine the network performance when all the nodes in the network execute random test cases. It is like a network with real users and they will not logout network. Thus the churn rate in this test is 0. Similarly in this test two scales of network are tested, one with 25 instances and the other one with 100 instances.

6.4.1 Test Results

Figure 6.81 and Figure 6.82 show global network nodes in each test. It is found that in 25-instance test all nodes keep online until test is finished. However in 100-instance test about 4 or 5 nodes lost after the random mode is enabled. By observation on site it is found that these nodes are stuck because of `TestCaseSendFriendRequest`. Except these lost nodes all other nodes keep

online until the test finished. The reason that these nodes which are crashed by `TestCaseSendFriendRequest` is much more less than random test with logout is, in this random test without logout, after the random mode is enabled, the first test case each slave choose may not be `TestCaseSendFriendRequest`.

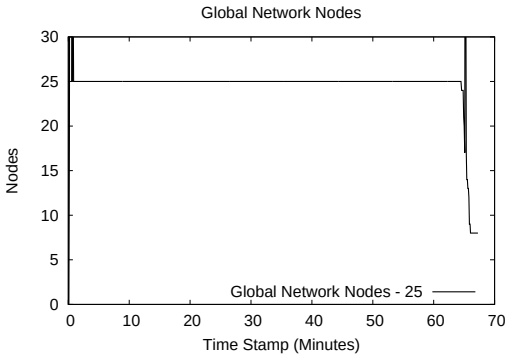


Figure 6.81: Global Network Nodes - 25 instance.

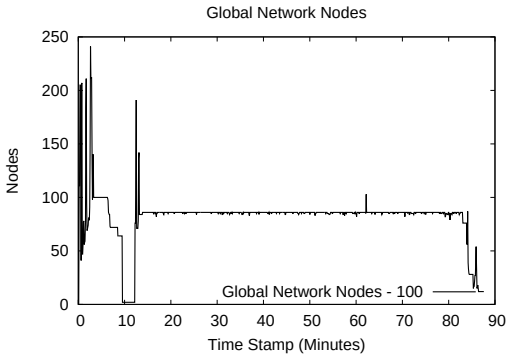


Figure 6.82: Global Network Nodes - 100 instance.

Figure 6.83 and Figure 6.84 show the lost message in both test. In 25-instance test, there is no message lost except when nodes logs out network in the end. In 100-instance test, after several nodes crashed due to `TestCaseSendFriendRequest`, about 6000 message lost in the network, however after several minutes the network recovered and no more message are lost.

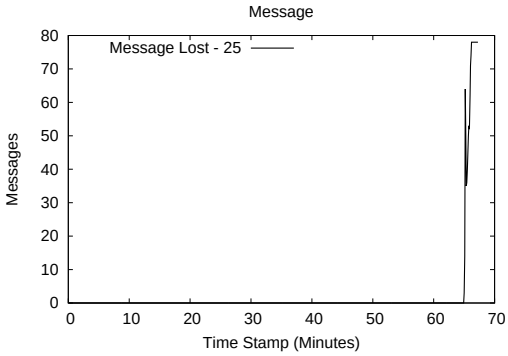


Figure 6.83: Global Message Lost - 25 instance.

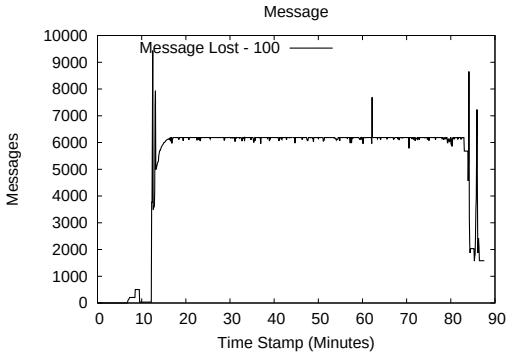


Figure 6.84: Global Message Lost - 100 instance.

Figure 6.85 and Figure 6.86 show mean stored elements by each instance in the network. The average stored elements by each node in 25-instance and 100-instance are same.

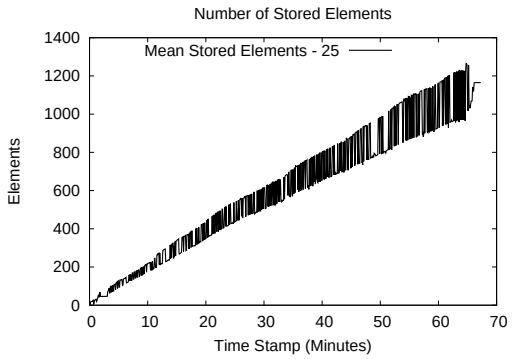


Figure 6.85: Global Stored Elements- 25 instance

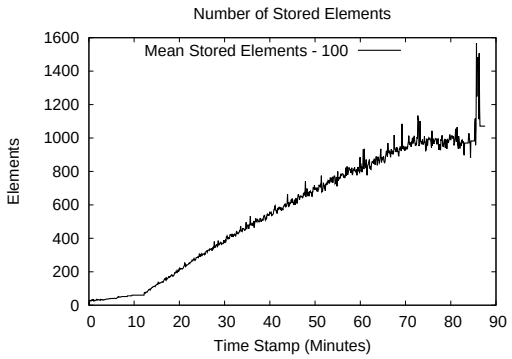


Figure 6.86: Global Stored Elements - 100 instance.

Figure6.87 and Figure 6.88 show the mean retrieve time in 25-instance test and 100-instance test. In both graphs during most time the value of retrieve time keeps very low. Only in the beginning of test, retrieve time is long compare to the value afterwards. It means retrieve time in the framework layer converges along with time. The same result is found in store time test in Figure 6.89 and Figure 6.90.

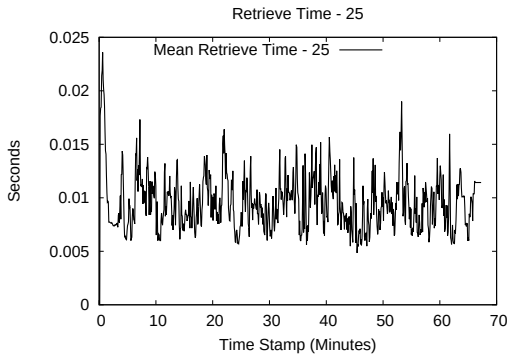


Figure 6.87: Global Retrieve Time - 25 instance.

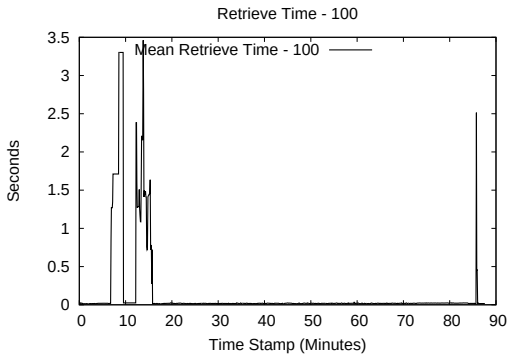


Figure 6.88: Global Retrieve Time - 100 instance.

Figure 6.91 and Figure 6.92 show framework message rate of both tests. Since the sent and received messages are same, only the received message rate is shown here. It is found that framework message rate keeps get high till the end of test. The message rates in both tests are roughly same, the rate in 25-instance test is a little bit high. The total message sent by framework of 25-instance is more than 100-instance's.

Figure 6.93 and Figure 6.94 show message rate of FreePastry in 25-instance

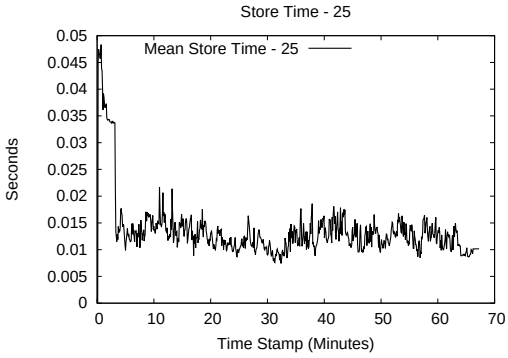


Figure 6.89: Global Store Time - 25 instance.

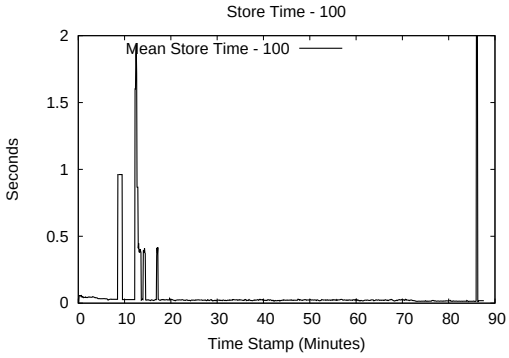


Figure 6.90: Global Store Time - 100 instance.

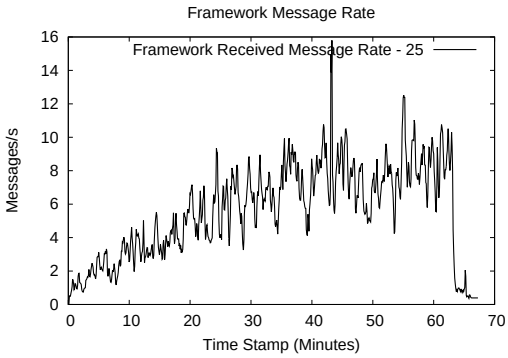


Figure 6.91: Global Framework Message Rate- 25 instance.

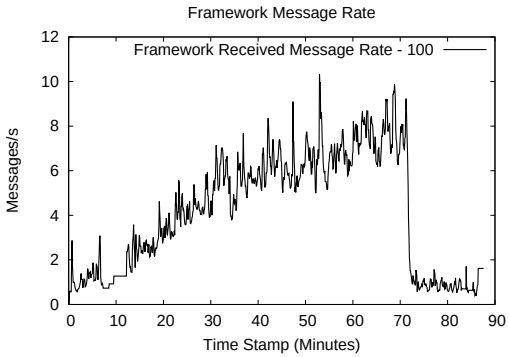


Figure 6.92: Global Framework Message Rate - 100 instance.

and 100-instance test. Result is similar to the framework message rate. Still, message rate in 25-instance is higher than 100-instance's.

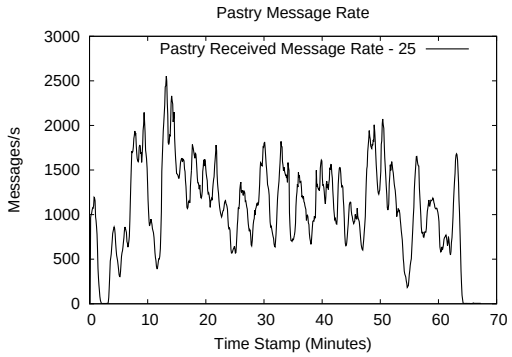


Figure 6.93: Global Pastry Message Rate - 25 instance.

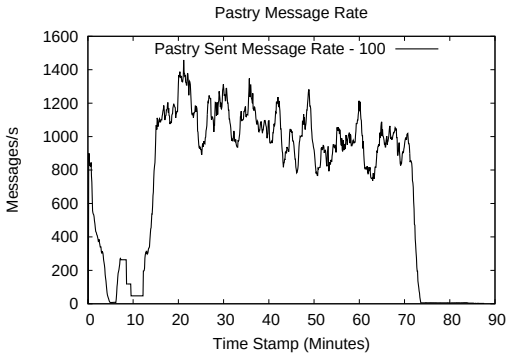


Figure 6.94: Global Pastry Message Rate - 100 instance.

Figure 6.95 and Figure 6.96 show the lookup time in both tests. The lookup time keeps almost horizontal after the random mode is enabled. The results of 25-instance test is roughly as same as the result of 100-instance test.

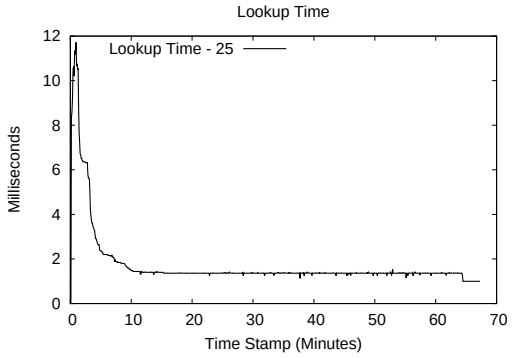


Figure 6.95: Global Lookup Time Average - 25 instance.

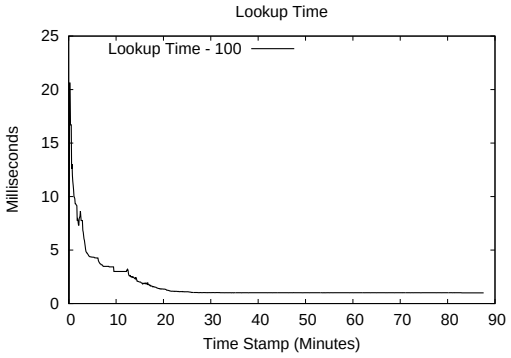


Figure 6.96: Global Lookup Time Average - 100 instance.

Figure 6.97 and Figure 6.80 show DDS data rate in 25-instance and 100-instance tests. In both test received data rate is higher than sent data rate. Send data rate keeps horizontal almost, on the contrary the received data rate keeps increasing till the end. Moreover, data rate in 100-instance test is higher than 25-instance's.

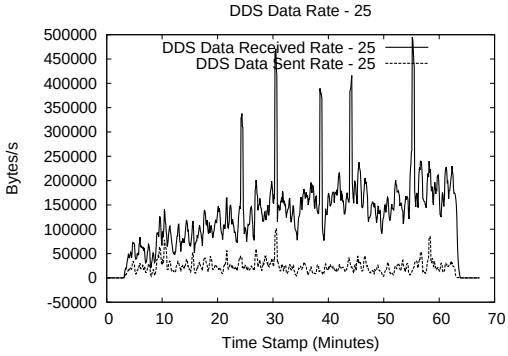


Figure 6.97: Global DDS Data Rate - 25 instance.

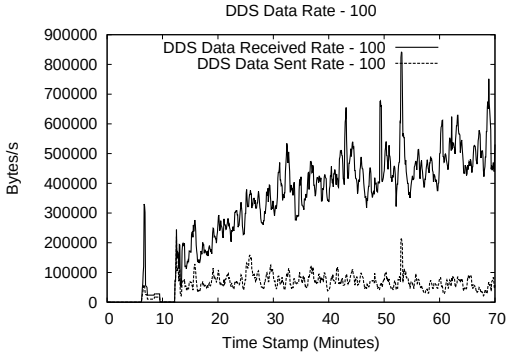


Figure 6.98: Global DDS Data Rate - 100 instance.

6.4.2 Conclusion

In the random test without logout tests, LifeSocial shows a high performance in both framework layer and overlay. Only in 100-instance test 2 nodes crashed due to `TestCaseSendFriendRequest`. And the network shows ability to recover after some nodes crashed. Compared to the random test with logout, it shows that LifeSocial can have a high performance if all instances stay in network all the time.

6.5 Long Term Test - 10 and 5 instances

This section introduces the result of long term test. The test setup is introduced in section 5.5. The reason to run this twice time for 10 instances and 5 instances is, at first 10-instance test showed a result that about after 1 day run, all nodes were full of message lost, and the reason is some nodes' message queue were overflowed. As shown in Figure 6.99, it shows that it is caused by `PriorityTransportLayer`. In previous master thesis, a queue overflowed problem is also discovered, it is overflowed by messages which are not acknowledged and it is already fixed. Now the reason of this problem is unclear, so a test with 5 instances is run in the purpose of finding whether this problem is related to scale of the network.

6.5.1 Test Result

Figure 6.100 and Figure 6.101 show the global network nodes of these two tests. Both figures show that around 1500 minutes, problem occurs and the values of global network nodes drop down. In the end, in both network only the local node is counted, thus the value drops to 1 finally.

```
        at rice.p2p.past.PastImpl.deliver(PastImpl.java:2119)
        at rice.pastry.commonapi.PastryEndpoint.receiveMessage(PastryEndpoint.java:582)
        at rice.pastry.standard.StandardRouter.deliverToApplication(StandardRouter.java:868)
        at rice.pastry.standard.StandardRouter.receiveRouteMessage(StandardRouter.java:608)
        at rice.pastry.standard.StandardRouter.route(StandardRouter.java:210)
        at rice.pastry.commonapi.PastryEndpoint.routeHelper(PastryEndpoint.java:290)
        at rice.pastry.commonapi.PastryEndpoint.route(PastryEndpoint.java:188)
        at rice.pastry.commonapi.PastryEndpoint.route(PastryEndpoint.java:181)
        at rice.pastry.commonapi.PastryEndpoint.route(PastryEndpoint.java:178)
        at rice.p2p.past.PastImpl.sendRequest(PastImpl.java:729)
        at rice.p2p.past.PastImpl.sendRequest(PastImpl.java:747)
        at rice.p2p.past.PastImpl.sendRequest(PastImpl.java:684)
        at rice.p2p.past.PastImpl.getHandles(PastImpl.java:970)
        at rice.p2p.past.PastImpl.doInsert(PastImpl.java:1041)
        at rice.p2p.past.PastImpl.insert(PastImpl.java:1125)
        at com.p2pframework.storage.past.PastryStorageDispatcher$2.receiveResult(PastryStorag
eDispatcher.java:353)
        at com.p2pframework.storage.past.PastryStorageDispatcher$2.receiveResult(PastryStorag
eDispatcher.java:1)
        at rice.p2p.past.PastImpl$10.receiveResult(PastImpl.java:1198)
        at rice.environment.processing.WorkRequest.returnResult(WorkRequest.java:70)
        at rice.environment.processing.WorkRequest$1.run(WorkRequest.java:87)
        at rice.selector.SelectorManager.doInvocations(SelectorManager.java:443)
        at rice.selector.SelectorManager.run(SelectorManager.java:264)
Caused by: org.mpisws.p2p.transport.priority.QueueOverflowException: Queue to /131.234.31.143
:10005:/131.234.31.143:10005 overflowed. Couldn't deliver message java.nio.HeapByteBuffer[pos
=1897 lim=1897 cap=1897]
        ... 60 more
```

Figure 6.99: Queue overflowed exception in long term test.

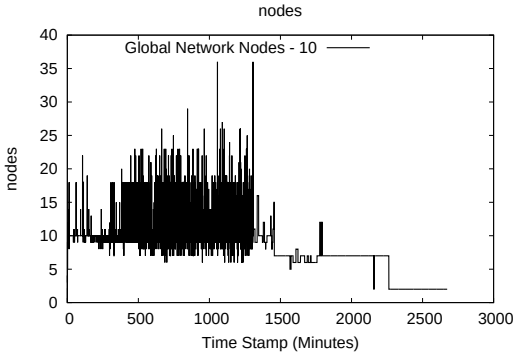


Figure 6.100: Global Network Nodes - 10 instance.

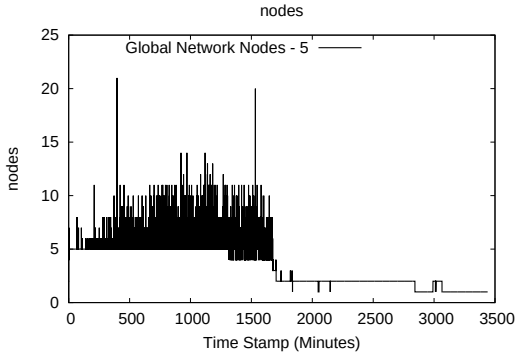


Figure 6.101: Global Network Nodes - 5 instance.

Figure 6.102 and Figure 6.103 show framework received data rate of both tests. Both data rates keep low before queue overflowed. Same results are found in framework data sent rate graphs.

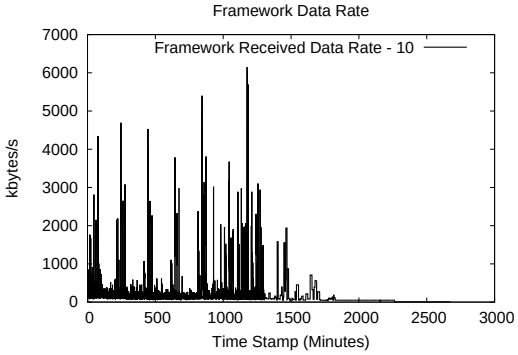


Figure 6.102: Global Framework Data Received Rate - 10 instance.

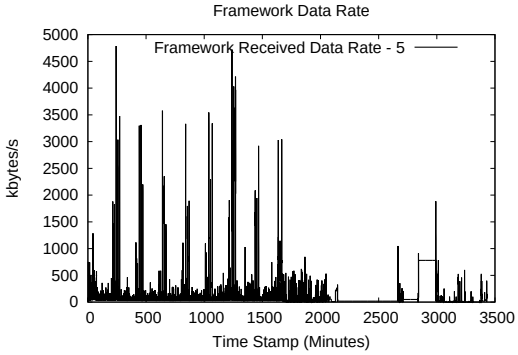


Figure 6.103: Global Framework Data Received Rate - 5 instance.

Figure 6.104 and Figure 6.105 again prove that message lost from the same time as shown in figures above.

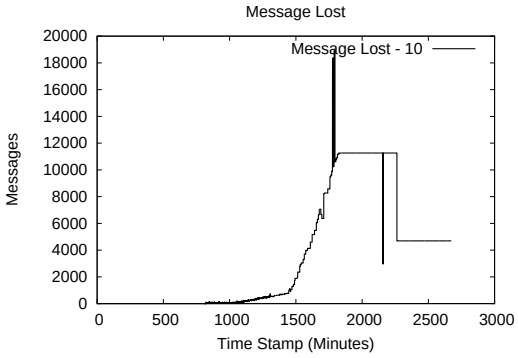


Figure 6.104: Global Message Lost - 10 instance.

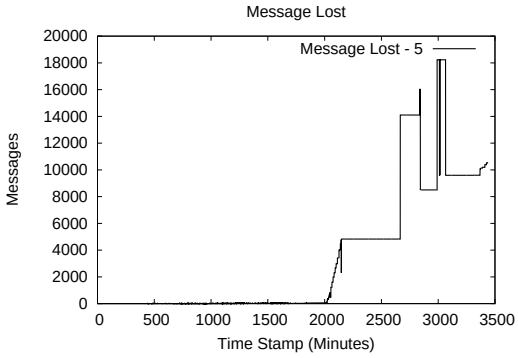


Figure 6.105: Global Message Lost - 5 instance.

Figure 6.106 and Figure 6.107 show pastry received data rate of both tests. Since the testplan is quite loose, as framework data rate shown in figures above which are quite low, thus these results basically show the basic data rate of FreePastry if no traffic load is generated above it. In 10-instance test pastry data rate keeps around 1000 kbytes and in 5-instance test it keeps around 500 kbytes.

Figure 6.108 and Figure 6.109 show memory usage of both tests. Before both networks crashed, the memory got as high as 800MB, it indicates that memory

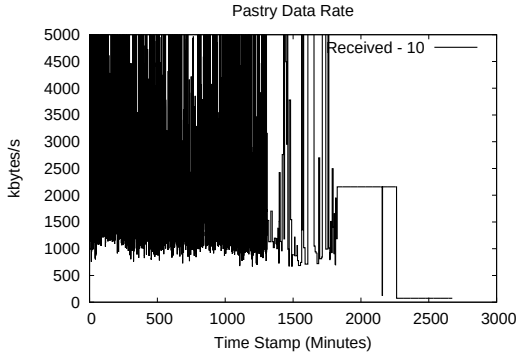


Figure 6.106: Global Pastry Data Received Rate - 10 instance.

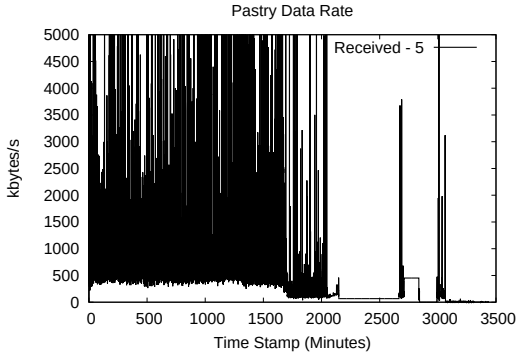


Figure 6.107: Global Pastry Data Received Rate - 5 instance.

usage has nothing to do with network size. Moreover, the testplan is super loose so that it means memory usage has nothing to do with the framework workload. The increase of memory is caused by FreePastry. From FreePastry forum, it is found that there are many discussions about this problem and yet there is no helpful result.

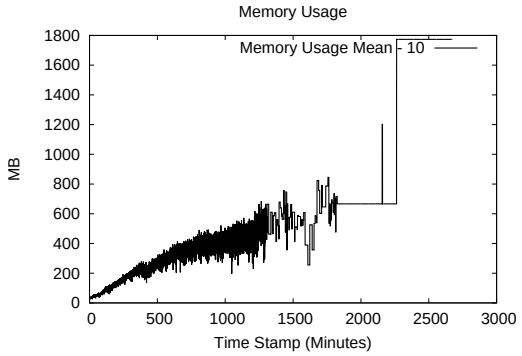


Figure 6.108: Global Memory Usage - 10 instance.

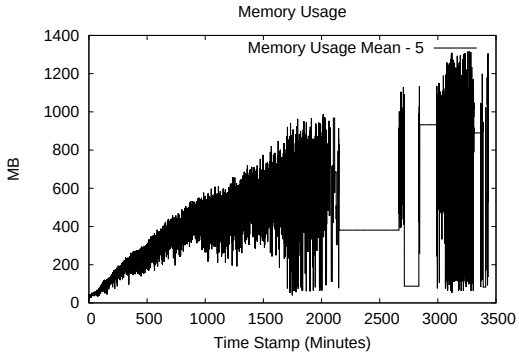


Figure 6.109: Global Memory Usage - 5 instance.

6.5.2 Conclusion

The long term test shows that current LifeSocial is able to run $1500minutes = 25h$ without any exception. After 1500 minutes, this queue overflowed exception is thrown by `PriorityTransportLayer` in FreePastry. Another issue revealed by long term test is, memory usage keep increasing slowly, but all the time during test. By analyzing it is caused by FreePastry overlay.

6.6 Alpha Test

In this test real users involved in, the test lasts two weeks. However, it did not succeed in the end. In this section the problem occurred in this test is documented which might be helpful for future work.

6.6.1 Problem Documentation

This section describes problems occurred in running alpha test.

User group User group in alpha test is a testing team inside the organization. Since the software is incomplete, a testing team is familiar with code and more easily to find bugs. The communication between test manager and user group should be well prepared. A problem about this occurred in this master thesis is, in the beginning of test, a introduction email is sent to the mailing list of user group, however due to unexpected mistake this email is blocked which results in a 2-week delay of test.
The motivation of user group is quite important. It will not be a problem if user group is a testing team inside organization, otherwise an attracting story behind test should be designed. In terms of testing LifeSocial, the story can be a certain activity or game in the network. In this master thesis, a riddle game in the network is designed, there are multiple virtual users are created in the network, the name of each virtual user is the answer of a riddle, users can compete to find as many virtual users as they can. However in this thesis, the user group is outside LifeSocial organization and not motivated at all. Only two users created the user account and did some actions in the network only in one same day. Thus no valuable result is recorded.

Data Duplicated Even though few user participate in the test. The backup nodes in the network are still booted in this two weeks and test manager joins network every day. It is observed that, in wall plugin created wall posts sometimes is duplicated.

Data Loss In the beginning of test, each virtual user is uploaded a profile picture. However after a certain time, some of these pictures are lost.

6.6.2 Conclusion

Even though the alpha test did not succeed in this master thesis, the learned experience and lessons is helpful for future. In addition, the test setup of alpha test has been studied and the difficulties has been solved which is introduced in Section 5.6.

7 Modifications and Extensions

This chapter shortly documents all the modifications and extensions made to LifeSocial before or during different tests. Section 7.1 documents how memory leak problem is solved. Section 7.2 introduces new added `OSGI` commands. Section 7.3 introduces the modification made for `Alpha Test`.

7.1 Memory leak

Memory usage increases quite quickly in LifeSocial before this master thesis. By analyzing it is caused by the monitoring function. Since LifeSocial is implemented a function that the monitored statistics of network can be rendered in the runtime, to be able to render these statistics LifeSocial keeps all the monitored data in memory. After this function is disabled, LifeSocial becomes quite memory saving now.

7.2 OSGI commands

Since the monitored statistics are not able to be rendered during runtime due to memory leak problem, to be able to show the monitoring result in runtime, a new command in osgi is added. By typing :

`monitor` + `Metric Name`

the monitored result of this metric will be shown in the console.

7.3 Alpha test

One problem occurs in the deployment of alpha test is, since these master and backup nodes are booted in the webserver via command line, to configure each instance, such as register and login user, has to be configured by the $OSGI$ console. There are available $OSGI$ commands which is introduced in Section 7.2 including register and login a user, but the GUI is not updated after the user register or login. Therefore other commands will not be able to be executed.

The current testplugin works in the way that, once the master logins to the network, each slave logins to the network by a randomly selected name and the user interface is updated as login state. However in alpha test, each backup node should always login by the same name. To adapt to alpha test, a new

8 Conclusion

This chapter describes the achievements of this thesis and the possible enhancements in the future.

8.1 Achievements

The goal of this master thesis is to benchmark a distributed framework for social networks. There are 4 different kinds of tests are designed to benchmark this framework. They are : `synthesis test`, `random test`, `long term test` and `alpha test`. But `synthesis test` and `random test` have the first priority, even though they have been run by the previous master thesis, new functions and new metrics have been added in LifeSocial which need to be tested again in these two tests. Long term test has the medium priority. Alpha test has the last priority, it is optional in this master thesis. The deployment and prepare work of alpha test have been tried and succeed in this master thesis, however the user group of this alpha test is not optimal, thus the result is not successful. Synthesis test has already been down by a previous master thesis, so this test has a easy start at the beginning of this master thesis. However before this master thesis, a project group had implemented many new features in LifeSocial. The problems made by these new features in overlay are fixed before testing. In addition, some metrics monitored by monitoring plugin are not accurate or actual, thus these metrics are fixed. Another series of metrics of data flow through LifeSocial are added. However these metrics bring a problem that memory leaks during runtime. This problem is finally fixed by turning off the visualization function of monitoring data.
Random test has done by previous master thesis too. This test has churn, at first nearly all nodes stuck since the logout node will login via the previous bootstrap node instead of master node. However this previous bootstrap node might offline, thus this problem is fixed by defining that logout node always login by master node. The final result of random test reveal a problem that requesting friendship function of LifeSocial needs to be improved. Because currently each slave get a randomly picked friend name from master by RMI method, this design causes many slave stuck during test, especially when many slaves request user name from master.

It is the first time to do long term test of LifeSocial. The result shows that LifeSocial is able to run for 25 hours without any exception. After that a queue overflowed exception occurs and nodes either keeping throwing this exception or keeping getting message lost. Another problem found in this test is, memory usage keeps increasing slowly all the time. It will impede a LifeSocial instance to stay online for a long time.
Alpha test of LifeSocial is failed due to users are not motivated to participate in the test. The lessons learned in this test is documented. The deployment of this test is work out and can be useful for the future work.
Another achievement of this master thesis is improving the monitoring function of LifeSocial. It is not a predefined goal of this master thesis, however the insufficient of monitoring function indeed have caused a big problem in evaluating test results. In the end the problem of monitoring function is fixed, but it still has a flaw which is documented in this thesis.
From these test, the conclusions are drawn that first, LifeSocial is able to working correctly if all nodes execute same test case at the same time. Moreover, in synthesis test, the main pressure of framework comes from DDS function. This data structure requires more message exchanges. In random test, the churn indeed brings issues to the network, many nodes are not able to login to network and data objects are missing in the network. However, the result of this test shows that a small scale p2p network is better in handling churn compare to large scale network. A small scale here means 25 nodes and large scale means 100 nodes. From random test without logout test, LifeSocial shows a high performance if all nodes keep staying in the network. From long term test, it is found that LifeSocial is not able to run correctly more that 25 hours and memory usage keeps increasing all the time. Finally, alpha test reveals that a motivated user group is important, and this user group is better to be a test group consisting of developers.

8.2 Future Enhancements

First, a severe problem that impedes random testing is that, `TestCaseSendFriendRequest` leads some nodes to be stuck. Instead of requiring each slave to invoke a method from master side, each slave should be able to randomly choose a node from its routing table and send friend request to.
A second main issue that impedes long term test is that, memory usage in FreePastry is increasing all the time. The reason of this problem is unclear now. It might be due to that messages are enqueued in `PriorityTransportLayer`.
Random test still should be run for LifeSocial in the future, since to put LifeSocial into industry it has to be able to handle a high rate churn in the network.

Moreover after memory leak problem is fixed, a long term test with more instances, such as 25 or 50, is worth to be run. Finally, an alpha test with motivated users will be an very important step for LifeSocial.

List of Figures

2.1 Structure of LifeSocial 5
2.2 Distributed Hash Table 7

3.1 The value in this graph is the sum of a certain metric. New rate is the data rate after last collect time. 13
3.2 Current rate is the final data rate which will be assigned to the metric as the value. 13
3.3 Data flow through LifeSocial. 14
3.4 LifeSocial parameters - 1 20
3.5 LifeSocial parameters - 2 21
3.6 LifeSocial parameters - 3 22
3.7 Dependencies between test cases. 27
3.8 Create a forum. 28
3.9 Create a forum thread. 28
3.10 Forum user interface. 28
3.11 View other user's forum. 28
3.12 Current test case dependencies. 29

4.1 SkyeEye.KOM ID Domain. 32
4.2 Message routing in SkyEye.KOM layer. 33
4.3 Monitoring activity diagram for one node. 34
4.4 A simple 4 nodes structure. 35
4.5 Monitoring in communication channel with high delay. 35
4.6 The metric *GLOBAL_NETWORK_NODES* measured by original monitoring mechanism. 36
4.7 *GLOBAL_NETWORK_NODES* measured without data measuring function. 37
4.8 Delayed and duplicate message. 37
4.9 *GLOBAL_NETWORK_PASTRY_DATA_SENT* measured by original monitoring mechanism. 37
4.10 Result of metric *GLOBAL_NETWORK_NODES* measured by the new monitoring mechanism that reuse last round's information. . . 40

4.11 Shift the stored rounds every time after the monitoring message is sent to parent.. 41
4.12 *GLOBAL_NETWORK_NODES* measured by new mechanism. 42
4.13 *GLOBAL_NETWORK_PASTRY_DATA_SENT* measured by new mechanism. 43

5.1 Machine pool of University Paderborn. 46
5.2 Master View 51
5.3 TestCase Delegater 52
5.4 Manually created *GLOBAL-_NETWORK_NODES*. 55
5.5 Manually created *GLOBAL-_NETWORK_PASTRY_DATA_SENT*. 55
5.6 Category Transition Probability for the LifeSocial categories by previous master thesis [6]. 61
5.7 Total online distribution based on Bavenuto et al. [4] 61

6.1 Global Network Nodes. 64
6.2 Global Node Boot Time. 64
6.3 Global stored elements. 64
6.4 Global Stored elements - Max Mean Min. 64
6.5 Global Received InsertMessages. 65
6.6 Global Received InsertMessages - Max Mean Min. 65
6.7 Global Used Storage and Total Storage. 66
6.8 Global Used Storage - Max Mean Min. 66
6.9 Global Framework Received Messages. 66
6.10 Global Framework Sent Messages. 66
6.11 Global Pastry Received Messages. 67
6.12 Global Pastry Sent Messages. 67
6.13 Global Sent LookupMessages. 68
6.14 Global Received LookupMessages. 68
6.15 Global Framework Received Messages Rate. 68
6.16 Global Pastry Message Received Rate. 68
6.17 Global Mean Store Time. 69
6.18 Global Mean Retrieve Time. 69
6.19 Global Stored Items. 69
6.20 Global Retrieved Items. 69
6.21 Global Lookup Time. 70
6.22 Global Missed Size Calculations. 70
6.23 Global Framework Data Received. 71
6.24 Global Framework Data Sent. 71
6.25 Global Framework Modules Data Received. 71
6.26 Global Framework Modules Data Sent. 71
6.27 Global Framework Data Received Rate. 72

6.28 Global Framework Data Sent Rate. . . . 72
6.29 Global Framework and FreePastry Data Received. . . . 72
6.30 Global Framework and FreePastry Data Sent. . . . 72
6.31 Global DDS Data Sent and Read. . . . 73
6.32 Global Photos Plugin Data Sent and Read. . . . 73
6.33 Global Number of Friend Request. . . . 73
6.34 Global Create Photo Albums. . . . 73
6.35 Global Network Nodes. . . . 74
6.36 Global Node Boot Time. . . . 74
6.37 Global stored elements. . . . 75
6.38 Global Stored elements - Max Mean Min. . . . 75
6.39 Global Used Storage and Total Storage. . . . 75
6.40 Global Used Storage - Max Mean Min. . . . 75
6.41 Global Stored Items. . . . 76
6.42 Global Retrieved Items. . . . 76
6.43 Global Received InsertMessages. . . . 76
6.44 Global Received InsertMessages - Max Mean Min. . . . 76
6.45 Global Framework Received Messages. . . . 77
6.46 Global Framework Sent Messages. . . . 77
6.47 Global Pastry Received Messages. . . . 77
6.48 Global Pastry Sent Messages. . . . 77
6.49 Global Sent LookupMessages. . . . 78
6.50 Global Received LookupMessages. . . . 78
6.51 Global Framework Received Messages Rate. . . . 78
6.52 Global Pastry Received Messages Rate. . . . 78
6.53 Global Mean Store Time. . . . 79
6.54 Global Mean Retrieve Time. . . . 79
6.55 Global Lookup Time. . . . 80
6.56 Global Missed Size Calculations. . . . 80
6.57 Global Framework Data Received. . . . 80
6.58 Global Framework Data Sent. . . . 80
6.59 Global Framework and Pastry Data Received. . . . 80
6.60 Global Framework and Pastry Data Sent. . . . 80
6.61 Global DDS Data Sent and Read. . . . 81
6.62 Global Photos Plugin Data Sent and Read. . . . 81
6.63 Global Network Nodes - 25 instance. . . . 83
6.64 Global Network Nodes - 100 instance. . . . 83
6.65 Global Join Network Time - 25 instance. . . . 83
6.66 Global Join Network Time - 100 instance. . . . 83
6.67 Global Retrieve Item Time - 25 instance. . . . 84
6.68 Global Retrieve Item Time - 100 instance. . . . 84
6.69 Global Store Item Time - 25 instance. . . . 84

6.70 Global Store Item Time - 100 instance. 84
6.71 Global Framework Message Received - 25 instance. 85
6.72 Global Framework Message Sent - 25 instance. 85
6.73 Global Pastry Message Received - 25 instance. 85
6.74 Global Pastry Message Sent - 25 instance. 85
6.75 Global Framework Message Received Rate - 25 instance. 86
6.76 Global Pastry Message Received Rate - 25 instance. 86
6.77 Global Framework Message Received Rate - 100 instance. 86
6.78 Global Pastry Message Received Rate - 100 instance. 86
6.79 Global DDS Data - 25 instance. 87
6.80 Global DDS Data - 100 instance. 87
6.81 Global Network Nodes - 25 instance. 88
6.82 Global Network Nodes - 100 instance. 88
6.83 Global Message Lost - 25 instance. 88
6.84 Global Message Lost - 100 instance. 88
6.85 Global Stored Elements- 25 instance 89
6.86 Global Stored Elements - 100 instance. 89
6.87 Global Retrieve Time - 25 instance. 89
6.88 Global Retrieve Time - 100 instance. 89
6.89 Global Store Time - 25 instance. 90
6.90 Global Store Time - 100 instance. 90
6.91 Global Framework Message Rate- 25 instance. 90
6.92 Global Framework Message Rate - 100 instance. 90
6.93 Global Pastry Message Rate - 25 instance. 91
6.94 Global Pastry Message Rate - 100 instance. 91
6.95 Global Lookup Time Average - 25 instance. 91
6.96 Global Lookup Time Average - 100 instance. 91
6.97 Global DDS Data Rate - 25 instance. 92
6.98 Global DDS Data Rate - 100 instance. 92
6.99 Queue overflowed exception in long term test. 93
6.100 Global Network Nodes - 10 instance. 93
6.101 Global Network Nodes - 5 instance. 93
6.102 Global Framework Data Received Rate - 10 instance. 94
6.103 Global Framework Data Received Rate - 5 instance. 94
6.104 Global Message Lost - 10 instance. 94
6.105 Global Message Lost - 5 instance. 94
6.106 Global Pastry Data Received Rate - 10 instance. 95
6.107 Global Pastry Data Received Rate - 5 instance. 95
6.108 Global Memory Usage - 10 instance. 95
6.109 Global Memory Usage - 5 instance. 95

Bibliography

[1] *FreePastry*. 2009. – http://www.freepastry.org/

[2] *Wikipedia*. 2012. – http://en.wikipedia.org/wiki/Facebook

[3] *Wikipedia*. 2013. – http://en.wikipedia.org/wiki/Xvfb

[4] Benevenuto, Fabrício ; Rodrigues, Tiago ; Cha, Meeyoung ; Almeida, Virgílio: Characterizing user behavior in online social networks. In: *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. New York, NY, USA : ACM, 2009 (IMC '09), S. 49–62. – URL http://doi.acm.org/10.1145/1644893.1644900. – ISBN 978-1-60558-771-4

[5] Graffi, Kalman ; Kovacevic, Aleksandra ; Xiao, Song ; Steinmetz, Ralf: SkyEye.KOM: An Information Management Over-Overlay for Getting the Oracle View on Structured P2P Systems. In: *Parallel and Distributed Systems, International Conference on* 0 (2008), S. 279–286. – ISSN 1521-9097

[6] Kallien, Pascal: *Systematic Evaluation of a Peer-to-Peer Framework in the Use Case of a Social Network*. Master thesis, Uni-Paderborn. 2012