Development and Evaluation of a Framework for Monitoring Peer-to-Peer Systems

Master Thesis
KOM-D-351



TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Electrical Engineering
and Information Technology
Department of Computer Science
(Adjunct Professor)

Multimedia Communications Lab - KOM
Prof. Dr.-Ing. Ralf Steinmetz

Submitted by
Christian Groß
on 31.03.2009

Advisor
Prof. Dr.-Ing. Ralf Steinmetz

Tutor
Kálmán György Graffi

1 Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit 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.

Darmstadt, den 31.03.2009 .....

Contents

1 Ehrenwörtliche Erklärung i

2 Introduction 1
- 2.1 Motivation . . . 1
- 2.2 Contribution . . . 3
- 2.3 Outline . . . 3

3 Related Work 5
- 3.1 Introduction to Distributed Information Management Systems . . . 5
  - 3.1.1 Astrolabe . . . 5
  - 3.1.2 Willow . . . 6
  - 3.1.3 SOMO . . . 7
  - 3.1.4 SDIMS . . . 8
  - 3.1.5 CONE . . . 9
  - 3.1.6 T-MAN . . . 10
  - 3.1.7 Proactive Epidemic Aggregation . . . 11
- 3.2 Discussion of Distributed Information Management Systems . . . 11

4 Background 15
- 4.1 Peer-to-Peer Systems . . . 15
- 4.2 Distributed Hashtables . . . 16
- 4.3 Self-Optimization Cycle . . . 18
  - 4.3.1 Monitoring . . . 19
  - 4.3.2 Data Analysis . . . 20
  - 4.3.3 Decision Making and Deployment . . . 21
- 4.4 General Requirements for a Distributed Information Management Framework . . . 22
- 4.5 Data Aggregation . . . 22
- 4.6 Capacities And Statistics . . . 25
- 4.7 Aggregation Schemes . . . 27
- 4.8 Tree-based Aggregation And Propagation Schemes . . . 30
- 4.9 Basic Problems . . . 31
- 4.10 OSGI . . . 32
  - 4.10.1 The Architecture . . . 33
  - 4.10.2 Bundle Dependencies . . . 36
  - 4.10.3 The OSGI Console . . . 37

5 SkyEye.KOM 39
- 5.1 Introduction . . . 39
- 5.2 Assumptions . . . 39
- 5.3 The SkyEye.KOM Approach . . . 39

5.4 Querying Data . . . 41
5.5 Optimizations . . . 41
5.6 Evaluation . . . 42

6 Architecture **43**
6.1 Introduction . . . 43
6.2 Module Overview . . . 43
6.3 Design Decisions . . . 46
6.3.1 Used Libraries . . . 48
6.3.2 Software Patterns . . . 49
6.4 Accessing Low-Level System Information through Java . . . 50
6.5 Querying Performance Data on Microsoft Windows Vista And XP . . . 51

7 Implementation **53**
7.1 Bundle Overview . . . 53
7.2 Bundle Interaction . . . 56
7.2.1 Updating / Propagating Statistical Data . . . 56
7.2.2 Receiving Statistic Updates . . . 56
7.3 Data Structures and Content Types . . . 57
7.4 Data Collection . . . 57
7.5 Data Storage And Aggregation . . . 61
7.6 Querying Data . . . 62
7.7 Extension Points . . . 64
7.8 Integration into LifeSocial.KOM . . . 66
7.9 Integration into PIPE . . . 68
7.10 Export Formats . . . 69

8 Evaluation **71**
8.1 Evaluation Criteria . . . 71
8.2 Evaluation Setup . . . 71
8.3 Presentation and Analysis of the Large-Scale-Test Results . . . 72
8.3.1 Number of peers . . . 72
8.3.2 Distribution of Peers . . . 73
8.3.3 Communication Costs . . . 74
8.3.4 Resource Consumption . . . 79
8.3.5 List-based Aggregation . . . 80

9 Conclusion **83**
9.1 Summary . . . 83
9.2 Future Work . . . 84

A Appendix **87**
A.1 Configuration File . . . 87
A.2 Starting the Monitoring Framework . . . 88
A.2.1 PIPE . . . 88
A.2.2 LifeSocial.KOM . . . 89

A.2.3 FreePastry 90
A.3 Configuration.ini File 91
A.4 Eclipse.ini File 91
A.5 Bat File for Starting 10 Peers 91
A.6 Further Simulation Results 92

List of Figures

2.1 Closed loop for optimizing P2P systems . . . 2
3.1 Example of an Astrolabe tree with three child zones. Taken from: [22] . . . 5
3.2 Example of a Willow tree using three bit identifiers. Taken from: [24] . . . 7
3.3 The data structure implemented in SOMO. Taken from: [43] . . . 7
3.4 An example of a SOMO tree. Taken from: [43] . . . 8
3.5 An example of a Cone tree. Taken from: [3] . . . 9
3.6 Construction of a torus topology using T-MAN. Taken from: [14] . . . 11
4.1 Communication paradigms . . . 15
4.2 Routing table of a Pastry node. Taken from: [26] . . . 17
4.3 Routing example of a message within FreePastry. Taken from: [26] . . . 17
4.4 Example for a Chord Ring . . . 18
4.5 Feedback loop concept for controlling dynamic system behavior. Taken from: [33] 19
4.6 Concept of flooding . . . 28
4.7 Overview of update/propagate strategies. Taken from: [20] . . . 30
4.8 The OSGI framework. Taken from: [38] . . . 32
4.9 The life cycle of an OSGI bundle. Taken from: [9] . . . 34
4.10 Example of declarative services in OSGI. Taken from: [1] . . . 36
4.11 Cyclic dependencies in OSGi . . . 37
5.1 Construction of the aggregation topology within SkyEye.KOM . . . 40
5.2 Support peer concept. Taken from: [29] . . . 42
6.1 Overview of the architecture . . . 44
6.2 Running the prototype with the LifeSocial.KOM communication bundle . . . 52
7.1 Bundle dependencies . . . 53
7.2 UML class diagram of the sensor bundle implementation . . . 55
7.3 Bundle interaction in case of sending statistical data . . . 56
7.4 Bundle interaction in case of receiving statistical data . . . 57
7.5 Class diagram of the basic data structures . . . 58
7.6 Class diagram of the basic data structures . . . 58
7.7 Query resolving . . . 63
8.1 Detected number of peers during the large scale test . . . 72
8.2 Number of peers by tree level averaged over time . . . 73
8.3 Number of peers by tree level . . . 74
8.4 Bandwidth utilization per sec upload / download . . . 75
8.5 Bandwidth utilization per second by tree level averaged over time . . . 75
8.6 Correlation between the number of peers and the download bandwidth utilization 76
8.7 Correlation between the number of peers and the download bandwidth utilization 77
8.8 Number of messages sent and received per second by tree level averaged over time 77
8.9 Average number of hops needed for delivering messages . . . 78
8.10 Amount of zipped and unzipped data send and received . . . 78
8.11 Average CPU and network usage . . . 79
8.12 Heap memory used . . . 80

8.13 Tree visualization with about 44 peers 81
8.14 Tree visualization with about 150 peers 81
8.15 Tree visualization with about 500 peers 82
9.1 Closed loop for optimizing P2P systems 85
A.1 Average CPU usage 92
A.2 Average download usage 93
A.3 Average upload usage 93
A.4 Average number of messages 94
A.5 Free disc space 94
A.6 Total number of messages 95
A.7 Average number of processors 95
A.8 Number of received messages per sec per level 96
A.9 Average uptime 96

List of Tables

3.1 Taxonomy for DIMS 13
3.2 Taxonomy for DIMS 13
4.1 Aggregation functions 24
4.2 List of statistics 26
4.3 List of capacities 27
7.1 Query commands 62

2 Introduction

The following chapter deals with the motivation, contribution and outline of this thesis. The goal of this chapter is to point out the reason, why a framework for monitoring Peer-to-Peer systems is needed in order to accomplish self-optimizing and adaptive Peer-to-Peer systems.

2.1 Motivation

In the recent years the Peer-to-Peer (P2P) paradigm has become more and more important since distributed systems have become present due to mobile phones / PDAs and the availability of pervasive network connectivity. According to current studies conducted by the Ipoque GmbH more than 50 percent of the overall traffic in the Internet are caused by P2P applications [12]. In contrast, the well known and commonly used client-server approach suffers from high costs and load balancing problems. E.g. the costs for maintaining the client server infrastructure of Facebook are estimated to be about 1.05 $ per user and year. With 98 million users in 2007 the total amount of money spent to keep the platform up and running rose to 98 million $ in 2007. Conversely, P2P systems have proven to be much more efficient and economic as they distribute the load and the costs among the users. Furthermore, most P2P systems scale, which means that they support millions of users without performance loss.
Many P2P applications have been developed offering a variety of services: distributed storage applications, chat, Voice over IP (VoIP) - telephony, mail, file sharing / content distribution, multimedia streaming and much more. However, services provided by current P2P application mostly do not fulfill availability or quality of service requirements, because on the one hand peers are free to join or to leave the network. On the other hand, services may be provided by "weak" peers offering only a small amount of resources. In order to improve the availability and the quality of services provided by peers, optimization mechanisms are necessary, ensuring that services will be deployed only on capable peers. Furthermore, peers should be able to adjust system parameters to improve the system according to some quality and performance measure. In order to accomplish self-optimizing and adaptive P2P systems the optimization cycle as shown in Figure 2.1 should be used. The optimization cycle consists of four states: Monitoring, Analyze, Plan and Execute. In the first step data about the P2P system is gathered by installing a monitoring component on each peer in the network. By using the data reported by the monitoring component, an analytic model is used to determine the global system state of the P2P systems. The global system state contains information about whether all system indicators are within normal parameters. If no system indicator has fallen below or exceeded a certain threshold, no optimization is needed. Otherwise, the system state and certain QoS requirements is passed to a decision model. During this planning step, possible adjustments for system parameters are considered and a decision towards an optimized system state is made. Finally, this decision is deployed at the network during an execution step.
During the first step of the optimization cycle the following two problems arise:

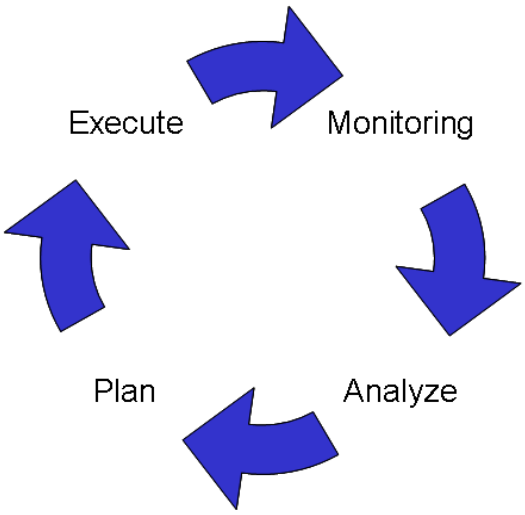


Figure 2.1: Closed loop for optimizing P2P systems

- Due to the distributed nature of a P2P system, the information about the current system state cannot be obtained at a central instance as known from client-server systems. Nevertheless, the information is needed by the optimization process.

- In order to deploy services only on capable peers, one must be able to search for peers fulfilling certain requirements.

To overcome these two problems, Distributed Information Management Systems (DIMS) have been developed, offering the possibility of finding capable peers. Furthermore, they are able to gather and aggregate data about each peer in the network in order to obtain knowledge about the global system state. Using DIMS, it is possible to solve a variety of management and self-configuration problems like finding the best entry point for recently joined peers.
DIMS should satisfy the following basic requirements: DIMS should provide a good responsiveness regarding to requested information and the overhead for aggregation and retrieving information should be low. Furthermore, DIMS should be robust and scalable both in terms of nodes and attributes being used. It should be possible to introduce new attributes and aggregation functions at runtime in order to ensure a great flexibility for new applications using the distributed monitoring infrastructure. As known from other types of applications, security concerns also should to be taken into account. The information provided by DIMS should be up-to-date, and time needed for updating and calculating a global view of the P2P system should be low as the state of distributed systems changes rapidly.
Most developed DIMS are built on top of a Distributed Hashtable (DHT), because DHTs have proven to be a robust, efficient and scalable approach for building distributed applications. In order to support different well known DHTs, DIMS should be overlay-independent.

2.2 Contribution

At the beginning of this thesis a survey of state-of-the-art Distribute Information Management Systems had been conducted. Based on the results of the survey a taxonomy for DIMS has been derived, such that DIMS can be classified.
Furthermore, a framework for DIMS has been developed, implementing SkyEye.KOM a DIMS, which has been introduced in [11]. In order to prove the applicability and extendability of the framework, it has been integrated into two existing P2P applications. The first application is LifeSocial.KOM, a P2P-based social community platform. The second application the framework is integrated in, is PIPE, a decentralized IDE plugin for Eclipse. To verify the quality of the used mechanism for gathering data about the global system state and to demonstrate the validity of the monitoring data, an evaluation process using a large testbed of up to 500 peers was conducted.

2.3 Outline

This thesis is structured as follows: First the state-of-the-art approaches for DIMS are introduced and discussed in Chapter 3. Afterwards, the basics needed in order to understand the developed framework are explained in Chapter 4. The core technology for gathering and aggregating the distributed peer data, called SkyEye.KOM, is explained in Chapter 5. The architecture as well as design decisions taken during the development process are discussed in Chapter 6 followed by the implementation details such as data structures and package descriptions explained in Chapter 7. The evaluation setup as well as the evaluation results are presented in Chapter 8. Finally, the conclusion and future work are introduced in Chapter 9.

3 Related Work

In this section related work is introduced in order to give an overview of the currently existing Distributed Information Management Systems (DIMS). Furthermore, the pros and cons of each system will be discussed in Chapter 3.2 and a taxonomy for DIMS will be presented.

3.1 Introduction to Distributed Information Management Systems

In the recent years a variety of Distributed Information Management Systems has been developed in order to obtain information about the performance and the current system state of P2P networks. The information can be used to help developers to find performance bottlenecks in their P2P applications or to optimize the P2P systems by adjusting system parameters accordingly. Optimization is needed in order to handle the arising complexity and bandwidth consumption caused by todays P2P systems.

3.1.1 Astrolabe

One of the most prominent DIMS is Astrolabe which was introduced in [22]. Astrolabe is able to "collect large scale system states, permits rapid updates and provides on the fly attribute aggregation" [22]. The authors claim that a variety of management and self-configuration problems can be solved using Astrolabe like finding the best entry point for new joining nodes or reconfiguration tasks caused by failing nodes. Astrolabe was built with scalability, flexibility, robustness and security in mind. Furthermore, aggregated data can be accessed due to the integration of JDBC [40] or ODBC [41].
Information in Astrolabe is collected, disseminated and aggregated in so called non-overlapping *zones*. The zones are recursively defined, forming a tree as shown in Figure 3.1. Each zone has a unique zone name, which is composed of the zone names seen, while traveling the path from the root of the tree down to the zone, separated by slashes (e.g. "/Darmstadt/TUD/PC2"). In

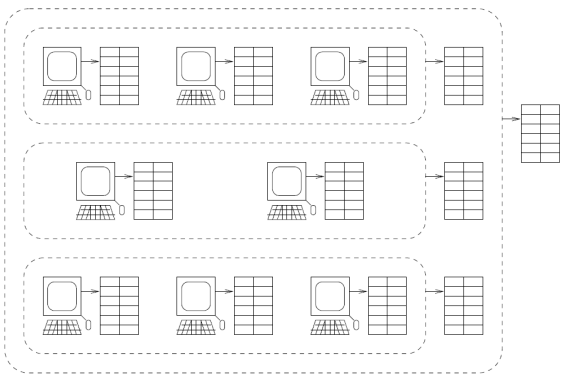


Figure 3.1: Example of an Astrolabe tree with three child zones. Taken from: [22]

addition, a list of attributes is associated with each zone called *Management Information Base* (MIB). Attribute values are calculated using *aggregation functions*. The tree structure described above consists of two types of nodes: inner nodes and leaf nodes. Leaf nodes form an exception as they contain virtual child zones, which are directly writable and not generated by aggregation functions. Each aggregation function uses the information of the MIB of the virtual child zones to produce the MIB for the inner nodes. *Aggregation function certificates* (AFC) describe which information should be aggregated and how.
Astrolabe has been designed in such a way, that connections between nodes are transparent to applications built on top of it. Developers can think of Astrolabe as a querying service, where data can be obtained using common SQL queries.
Queries are handled as follows: Once a query has been initiated, the nearest local zones are inspected first. If no matching data can be found, fulfilling the query terms, the query traverses higher levels of the tree, until it reaches a zone containing the requested data. If the query reaches the root and no data element satisfies the query terms, the result of the query is empty.
In addition, Astrolabe supports dynamic aggregation, which allows users to search for specific information in a distributed environment by introducing new aggregation functions. Using this functionality, publish / subscribe or multicast facilities can be implemented as described in [22].
Communication in Astrolabe is done using *Gossip* [6], a fast scalable and secure epidemic P2P protocol. The basic idea behind this protocol is, that random peers exchange and merge information about their MIB periodically. "In this manner the state of Astrolabe agents will tend to converge with time" [22]. For each zone a set of agents is elected arbitrarily or based on some metric, which gossips on behalf of the zone. Periodically each agent runs the gossip algorithms. First it decides at which level in the zone tree it will gossip. Afterwards, he selects a randomly chosen contact agent from the target zone and transfers the data of all its child zones. The recipient updates his MIB with the received information and sends an update including his MIB to the sender.
Node failures are detected by checking whether an update of a specific zone has arrived within the last T seconds. If not, the corresponding MIB of the failed node is removed.

3.1.2 Willow

In [24] the authors introduced Willow, a P2P based protocol supporting resource location, point-to-point and application multicast, publish/subscribe and data aggregation. Willow is based on Astrolabe [22] (see 3.1.1) and extends it in such a way, that its design is closer to a traditional DHT.
Willow uses a binary tree which is build up from 128 bit identifiers, which each node has to choose one randomly at startup as shown in Figure 3.2. Starting with the first bit of the identifier at the root of the Willow tree, a "0" bit determines the left child of a node, and a "1" bit the right child. Following this scheme, a tree consisting of 129 levels is constructed. A node in the tree is called *domain*, and all child nodes sharing the same prefix with a current node are members of it´s domain. In addition, every node contains attributes. In case of a leaf node the attributes are directly written by the node itself. In terms of an internal node, the attributes are determined by aggregating the attributes of the left and right child node. Doing so, the root will contain the aggregated attributes of all nodes in the network. In terms of the failure or arrival of a peer, the aggregation value is updated in a bottom up fashion. Willow offers no guarantees

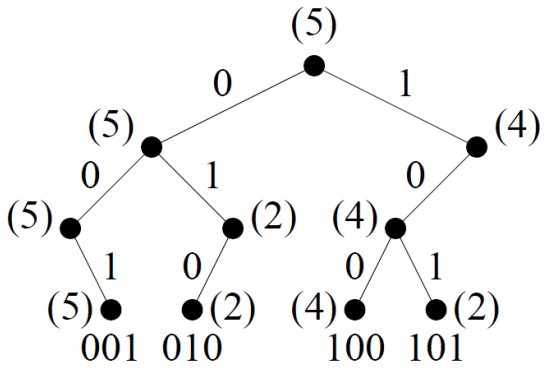


Figure 3.2: Example of a Willow tree using three bit identifiers. Taken from: [24]

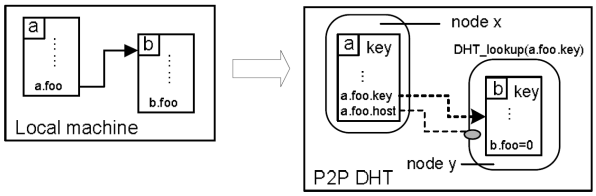


Figure 3.3: The data structure implemented in SOMO. Taken from: [43]

of consistency for aggregation values being observed by different peers, but all values will converge quickly to the same state.

Monitoring is done by installing a query at a peer. Afterwards, the query in propagated among all other peers, which will start to retrieve, aggregate and report the requested data. Furthermore, Willow is able to support publish / subscribe and multicasting functionality. Given any domain in the tree, Willow can forward messages down the tree to both children. An optional given SQL statement determines whether the message should be forwarded into the left or right subtree.

All communication in Willow is done using TCP so that no data can be lost during transmission. Evaluation showed that in terms of churn, a broken Willow tree can be restructured efficiently within only $O(log(n))$.

3.1.3 SOMO

In [43] the concept of a data overlay is introduced, which enables the construction of a robust infrastructure for resource management called SOMO (Self-Organized Meta-data Overlay). Resource management is done by gathering and disseminating system meta-data, which is stored in the data overlay. The following challenges for a Self-Organized Meta-data Overlay are pointed out by the authors: on the one hand the infrastructure has to grow along the hosting DHT. On the other hand, it has to be agnostic to the specific protocols and performance of the underlying DHT. Furthermore, one of core functionalities needed for gathering and disseminating system meta-data is a distributed data structure. So the authors introduced an approach for implementing an arbitrary data structure on top of a DHT as shown in Figure 3.3. From the authors point of view, every distributed data structure has to satisfy the following two conditions: "Each

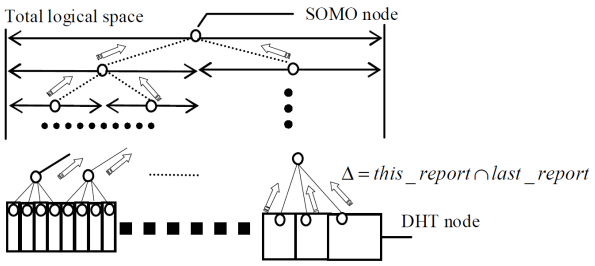


Figure 3.4: An example of a SOMO tree. Taken from: [43]

object must have a key, obtained at its birth" [43] and each attribute of an object is expanded into the two pointers *key* and *host*. The former is a conventional pointer to the key and the latter "the soft state containing the last known hosting DHT node of the key" [43]. This distributed data structure is called data overlay and is independent from the underlying DHT performance, thanks to the *host* shortcut. Without internal reliability support only soft-stated, distributed data structures can be implemented.
The second focus of the paper is the implementation of SOMO, which builds up a tree based meta-data overlay, with a degree of k. Information is gathered at the leaves and propagated and aggregated bottom up in the tree. The system state is available at the root of the tree, after all meta-data has been aggregated. Finally, the system state is triggered down to the leaves of the tree.
The construction of the tree is done using a top down approach, by dividing the responsibility interval of a node into k equidistant smaller intervals as shown in Figure 3.4. The responsible node for each interval is determined by the center of the sub interval. This recursive construction method is done periodically to ensure that the tree gets pruned from deleted or missing nodes. The whole tree topology can be built up within only $O(log_k(N))$ time and the system state is calculated in worst case with a delay of $log_k(N) * T$ with T denoting the time between two update cycles. The authors tested SOMO by finding super-nodes in a prefix-based overlay in order to improve the load balancing. In this context super-nodes were nodes, which provided more capacities than others. Two types of tree topologies where tested and an optimization algorithm was applied to them. The optimization algorithm contained the following four phases: statistic data collection, load balancing, capacity selection and entries dissemination. The authors found out, that in all configurations the load balancing algorithm converges in $O(log(N))$ time and that higher capacity nodes were taking over the load.

3.1.4 SDIMS

In [42] the authors point out the challenges and the benefits of a Scalable Distributed Information Management System (SDIMS), which can serve as a basic building block for a great variety of distributed applications. As key requirements for such a building block, the authors denoted scalability, flexibility, autonomy, isolation and robustness.
To prove their idea, the author developed three real world application using the SDIMS prototype to demonstrate, that the SDIMS prototype fulfills the above mentioned key requirements.
From the authors point of view aggregation is the natural abstraction of a SDIMS. Aggregation is able to support "scalability by allowing a node to view detailed information about nodes in his

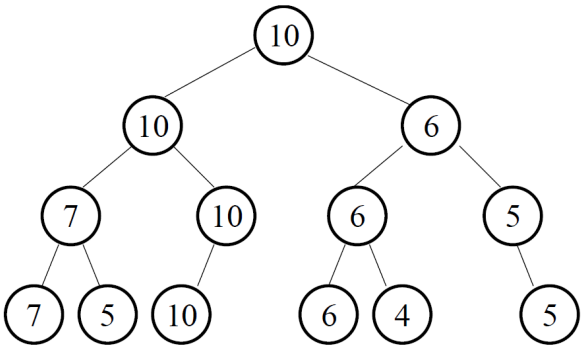


Figure 3.5: An example of a Cone tree. Taken from: [3]

neighborhood and a coarser-grained summary of system data of larger subsets of nodes" [42]. Aggregation appears along with the questions where to compute the aggregation value, which aggregation function should be used and which computation / propagation scheme should be applied. In general the aggregation function used must satisfy the *hierarchical computation property*, which ensures that aggregates are recursively computable. The decision of the right computation / propagation strategy is of importance, because a wrong strategy will either lead to high bandwidth consumptions or to high query latencies. So one of the main points of the paper is the development of an interface providing functionality such that, scalable trees for aggregation can be considered. This interface is called aggregation abstraction, which is naturally supported by DHTs, because the aggregation tree structure fits to their underlying routing mechanism.
In order to support scalability in the number of nodes and in the number of attributes, the authors propose the usage of multiple aggregation trees, because from their point of view it leads to a better load balancing and scalability. In addition, multiple tree aggregation schemes offer the ability of supporting a wide range of applications as newly added attributes can be handled by adding additional trees for aggregation. So, this approach enables the handling of a large numbers of attributes, but makes it harder to resolve multiple-trees-spanning queries.
Furthermore, autonomy as mentioned above is of importance because the availability, security and delivery of meaningful results should be guaranteed in cases of domain specific queries. From the authors point of view this can be achieved using *Autonomous Distributed Hash Tables* ADHT, which can be obtained from normal DHT by modifying the systems routing table construction, such that single exit points for each domain and key can be guaranteed. Furthermore, an access control for domain specific attributes can be established using ADHTs.
Robustness concerns are addressed by masking and isolating failures of nodes at DHT level and performing replication with respect to space and time.

3.1.5 CONE

[3] deals with the subject how DHTs can be extended in order to support resource discovery for better load balancing. The core functionality of resource discovery is to find a resource offering capacities above a given size. From the authors point of view the natural data structure abstraction for answering this questions is a heap as shown in Figure 3.5. Thus, the contribution of the paper is: how to implement and to maintain a generic distributed data structure on top

of a given DHT. In other words, the authors introduced a generic augmentation approach for DHT, such that more complex applications on top of a DHT can be considered. Furthermore the authors used their proposed augmentation approach to introduce their heap-like distributed data structure called CONE. "Similar to a heap, the CONE data structure is a tree with an aggregation key at the root of each subtree" [3]. The aggregation key can be the result of any known aggregation function like SUM, MIN or MAX.
The construction of the CONE tree is done as follows: A new node *R* joining the DHT also joins the CONE tree at the lowest level. To find the definite entry point in the tree, a node determines its closest neighbor *S* by finding the node with the longest common prefix. Afterward, *R* computes the *least common ancestor* (LCA) in the tree it shares with *S*, which forms the entry point for *R* in the tree. After this joining procedure a tree update is necessary to ensure that the right node is at the top of each subtree. Starting bottom up from the LCA, the new node trickles up in the tree. Every node determines whether the new node has a greater key than itself. If so, the new node becomes the new parent. Otherwise, the update procedure stops and the old node remains the parent. The update is done recursively until it reaches the root of the tree which contains all nodes in the DHT.
In terms of a failing node the heap structure of the tree resolves to several disconnected components. In order to reconnect them, every root of a subtree uses the DHT, to find its new parent.
To find a node offering capacities greater than a given threshold, a query can be initiated at any node and traces up in the tree until it reaches a node with a key greater or equal the given threshold. Afterwards, the node fulfilling the query conditions is returned. Due to its tree topology, the CONE heap suffers from load balancing problems. For this reason the authors introduced two approaches for improving the distribution of load of the CONE heap.

3.1.6 T-MAN

The topology of a network has a great impact on the efficiency, costs and robustness of functions like routing, searching or sorting. So the authors in [14] introduced a generic protocol for topology management named T-MAN, which offers the possibility for changing the topology of a network at runtime. As key requirements the authors denoted robustness, scalability, flexibility and simplicity. Using T-MAN only one topology management component per peer is needed, to manage different types of topologies in a network. In order to construct different types of network topologies such as a ring, mesh or binary tree topology, a specific ranking function is used, which enables nodes to find their "right" neighbors based on a given topology. In easy cases like ring, mesh or binary tree topologies a simple distance function can be used for ranking peers. However, more complex topologies cannot be described by simple distance functions and other approaches are necessary. In general, a ranking function returns an ordered list of nodes denoting the neighbors of a node. Every node is maintaining a partial view of size *c* of the network, containing *c* other nodes in the network. By exchanging and updating their views using a gossip based protocol, nodes can optimize their views by finding better nodes relative to a given ranking function. At the beginning of the protocol all node views are initialized with a random subset of nodes, which ensures the convergence of large diameter topologies against the topology given.
Although T-MAN is not a synchronous protocol, the authors speak of cycles which take $\frac{T}{2}$ time, during which each node is updated once in average. The evaluation of T-MAN showed that it

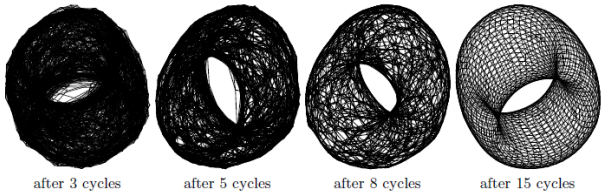


Figure 3.6: Construction of a torus topology using T-MAN. Taken from: [14]

is very efficient because topologies converge very fast as only $O(log(N))$ cycles are needed to establish a given topology as shown in Figure 3.6. Surprisingly, the speed of convergence is independent of the view size c and the given topology. In the presence of churn, where up to 10 % of the nodes are reinitialized per cycle, the efficiency of the protocol decreases significantly due to the fact that a newly joining node needs up to 10 cycles in average to optimize his view, which is the same amount of time a node stays in the network in average before it is reinitialized. Depending on the task (sorting, clustering, aggregation) another problem that might occur is that of clustering. "Regular distribution of nodes over the whole ID-Space will hardly happen instead of dense clusters and unrepresented regions"[14]. In terms that the task is clustering there will be no problem, but in cases of sorting, clustering must be prevented by using direction depending sorting. In their conclusion the authors proposed that optimal ranking functions can be found by a monitoring service.

3.1.7 Proactive Epidemic Aggregation

In [13] the authors introduced an efficient, simple and lightweight protocol for solving the proactive aggregation problem using anti-entropy aggregation. Anti-entropy aggregation works as follows: By randomly selecting nodes from their neighborhood, nodes periodically exchange messages containing the aggregate of their local data. Once a node receives a message, it merges the information received with its local data and answers the message, including the result of the merging process. This proposed approach delivers fast results due to an exponential convergence rate because of the massive parallel computation of aggregates. With increasing network size the amount of messages needed increases linearly. Hierarchical structures as used by e.g. Astrolabe lead to reduced costs of finding required aggregates, but on the other hand the construction and maintenance of the structure induces additional overhead. Conversely, anti-entropy approaches do not use an ordered structure and for this reason no further costs arise. Furthermore, they are extremely simple, lightweight and targeted to unstructured, highly dynamic environments and can be used for collective decision making or automated system maintenance.

3.2 Discussion of Distributed Information Management Systems

In the following the introduced DIMS are discussed and the pros and cons are pointed out. A list of important attributes for comparison of DIMS is given in [11], which will be extended in order to cover all aspects of a DIMS.
The most important criteria for categorizing a DIMS is the topology used for aggregation. Ac-

cording to this criteria we can define structured and unstructured DIMS, whereas DIMS using a structured approach have been much more studied due to the hierarchical nature of aggregation. Structured DIMS maintain an aggregation topology for gathering data about a P2P system. Conversely, unstructured DIMS mostly use a gossip-based communication protocol for exchanging information about the system state and do not maintain a aggregation topology.
Structured DIMS can be classified into single and multi-tree approaches. Single-tree DIMS like[11],[43] and [22] maintain only one tree for aggregating attribute values, whereas multi-tree approaches like [20] construct multiple trees for attribute aggregation, mostly one for each attribute. As already mentioned unstructured approaches do not maintain a tree structure for aggregating data, which leads to less communication overhead. Conversely, structured approaches produce higher communication costs for constructing and maintaining a structured over-overlay, which might lead to a performance bottleneck in terms of high churn rates.
In order to compute a global system state unstructured DIMS suffer from a high communication overhead due to the gossip-based communication protocol used, where nodes randomly exchange messages. In contrast, structured DIMS mostly require only $O(log(N))$ messages for gathering information about the current system state of a P2P system. So in summary one could say, that structured and unstructured DIMS vary in the costs for constructing the aggregation topology and in the costs needed for gathering and computing a global system state.
The decision of using a single-tree vs. a multi-tree approach has a great impact on the load balancing capability of a system. Due to their distributed nature, multi-tree topologies are much more load balanced, than single-tree topologies. Nevertheless, multi-tree approaches might also suffer from performance bottlenecks in higher tree levels as pointed out in [20].
Resolving complex queries within multi-tree approaches is more difficult than resolving queries in a single tree, because for each attribute a separate query message containing the attribute specific conditions must be initiated at the corresponding attribute tree. After receiving the results for each query message, they must be merged. When using single-tree approaches only a single query message must be sent which can be resolved directly at the answering node.
Another important aspect of a DIMS is a the update / propagate scheme used for gathering and distributing data within the P2P system, which will be discussed in 4.8.
It is noticeable, that most of the current systems do not consider any security issues although critical data might be exchanged during the aggregation process. Some of the possible security threads are lined out in Chapter 4.9.
A classification for DIMS has been developed using the discussed criteria. Table 3.1 and 3.2 give an overview about the DIMS discussed above and how they can be classified.

Attribute	CONE	Willow	SOMO	SDIMS
Topology	single tree	single tree	single tree	multiple trees
List-based Aggregation	no	yes	no	yes
Aggregation Functions supported	yes	yes	yes	yes
Overlay Independent	no	no	no	no
Supporting Complex Queries	yes	yes	yes	no
Load Balanced	no	yes	no	yes
Churn Resistant	no	yes	yes	yes
Communication Protocol	DHT	DHT	DHT	DHT
Security Issues considered	no	no	no	no
Update Scheme	push	push	push	push
Propagation	no	yes	yes	yes
Heterogeneity	no	no	no	no

Table 3.1: Taxonomy for DIMS

Attribute	Astrolabe	T-MAN	Proactive Epidemic Aggregation
Topology	single tree	multiple topologies	n/a
List-based Aggregation	no	yes	yes
Aggregation Functions Supported	yes	yes	yes
Overlay independent	no	no	no
Supporting Complex Queries	yes	yes	yes
Load Balanced	no	yes	yes
Churn Resistant	yes	no	yes
Communication Protocol	Gossip	Gossip	Gossip
Security Issues Considered	yes	no	no
Update Scheme	push	random exchange	random exchange
Propagation	yes	no	yes
Heterogeneity	no	no	no

Table 3.2: Taxonomy for DIMS

4 Background

The following chapter deals with the basics of Peer-to-Peer (P2P) systems, which are needed in order to understand the functionality of the developed framework. The general concept of Distributed Hashtables (DHT) is introduced and two of the most common DHT implementations are discussed. In addition, the requirements for Distributed Information Management Systems as well as the OSGI framework are introduced.

4.1 Peer-to-Peer Systems

"A Peer-to-Peer system is a self-organizing system of equal, autonomous entities (peers), which aims for the shared usage of distributed resources in a networked environment avoiding central services." [21]. Peer-to-Peer (P2P) systems have become quite popular in the recent years due to file sharing applications like Kazaa and VoIP telephony based on Skype. But P2P is much more than just a technology for file sharing, which has become a heavily discussed topic due to copyright issues. "Rather, it forms a fundamental design principle for distributed systems. It clearly reflects the paradigm shift from coordination to cooperation, from centralization to decentralization and from control to incentive." [21]. In contrast to the traditional client server approach, which suffers from high costs, scalability- and single-point-of-failure issues, Peer-to-Peer systems have proven to be robust, scalable and self-organizing. In addition, due to the usage of the peer resources Peer-to-Peer systems do not cause high costs except bandwidth usage. A current study provided by the Ipoque GmbH [12] has shown that more than 50 percent of the total Internet traffic in 2008 was caused by P2P applications. This fact demonstrates how important the P2P paradigm has become in the recent years. Thus, it is essential to understand how P2P systems work and how they could be improved and optimized.
Currently there exist tree type of communication paradigms as shown in Figure 4.1. Besides there is a hybrid alternative to the already mentioned traditional client-server-approach, where peers are coordinated by a central instance. Following this approach, communication between peers is accomplished by first querying the location of a peer at the central instance and then initiating the P2P communication in a second step. As shown in Figure 4.1 the third commu-

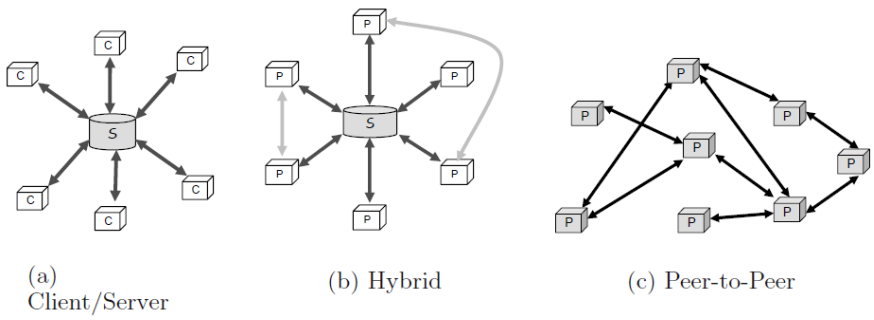


Figure 4.1: Communication paradigms

nication paradigm deals with P2P systems without central instance such that peers and their communication are managed in a fully decentralized fashion.
Furthermore, P2P systems can be divided into structured vs. unstructured P2P systems. Unstructured P2P systems were commonly used during the first generation of P2P systems, which used a central server for organizing peers. Communication between peers takes place only after the location is looked up at the central server. Another commonly used approach for unstructured P2P systems is flooding, where query requests are flooded through the network of peers until either a peer is able to answer the request or the time-to-life counter of the query request expires. In contrast to unstructured P2P systems, structured P2P systems define a "distributed content-addressable data storage" [21] called *Distributed Hashtable* (DHT), which are explained in Chapter 4.2.
Beside the already mentioned advantages, P2P systems also have some weaknesses in contrast to the traditional client server approach. In case of time-critical services such as streaming applications, P2P systems often suffer from high response times. Furthermore, high degrees of reliability as known from client-server systems can only be accomplished, with high degrees of redundancy (aka: replication factor) especially in terms of high churn rates.
Another often neglected issue is that of Security. Ensuring the basic security principles such as trustworthiness, authentication and authorization in a distributed environment is hard to accomplish.
Conflict resolution in terms of concurrent modifications has always been a hot topic in the recent years, especially when data items are replicated in order to ensure high degrees of reliability.

4.2 Distributed Hashtables

Distributed Hashtables (DHT) have proven to be the state-of-the-art technology for creating robust, scalable and self-organizing P2P applications. The concept of DHTs has been introduced concomitantly by MIT and Microsoft Research in 2001 for the first time. DHTs commonly use a m-bit identifier space in order to address nodes and content within the DHT. Mostly hash functions like SHA-1 or MD5 are used to compute m-bit identifiers under which nodes or content can be accessed. Every node in the P2P network participating in the DHT, is responsible for a certain subset of the m-bit ID space. In addition, DHTs provide a routing mechanism, which ensures, that "given a message or a node with a key from the ID space, the message or the node will be routed to the currently responsible node" [26]. Hence, this mechanism ensures on the one hand, that a newly joining node can be integrated into the DHT according to the nodes own ID by routing it to the corresponding node, which is responsible for the joining node´s ID. On the other hand using the routing mechanism, content can be stored in and retrieved from the DHT.
DHTs work quite efficiently as they offer the possibility for retrieving objects within a complexity of $O(logN)$ with N denoting the number of peers. Adding new peers or content is almost done within a complexity of $O(logN)$ or $O(log^2N)$ [21]. In the following, the most prominent concepts for DHTs are introduced and explained, in order to get a deeper understanding on how a DHT works.

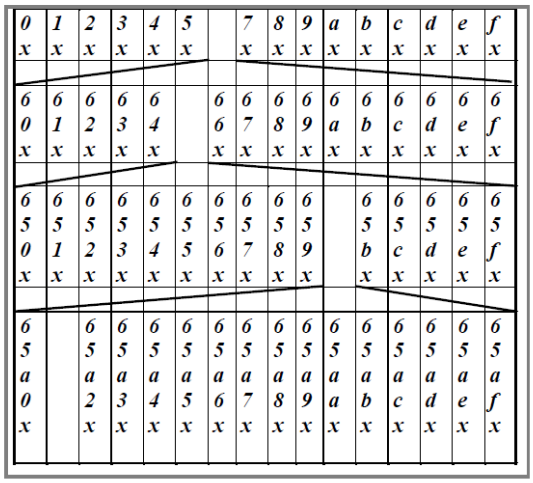


Figure 4.2: Routing table of a Pastry node. Taken from: [26]

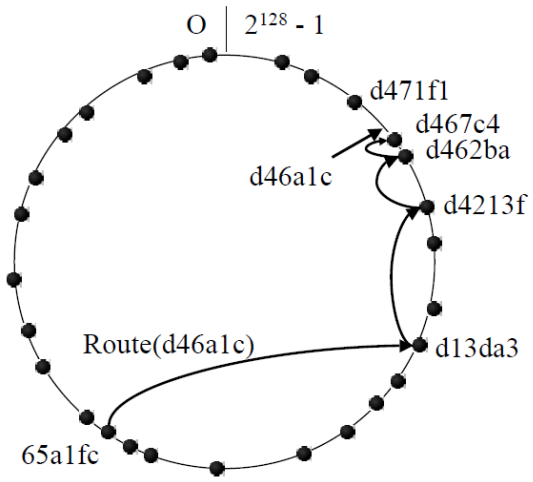


Figure 4.3: Routing example of a message within FreePastry. Taken from: [26]

FreePastry

The FreePastry DHT has been developed by the Rice University since 2001. FreePastry organizes peers in a 128-bit circular identifier space and uses a prefix routing mechanism for delivering messages at the receiving nodes. "Given a message and a key, Freepastry will deliver the message to the node with an ID numerical closest to the key". [26]. To be able to route messages to the corresponding node, each node maintains a routing table with $2^b - 1$ rows and $\lceil log_{2^b} N \rceil$ columns, with b denoting a configuration parameter with typical value 4 and N denoting the number of nodes. The routing table is organized such that "the $2^b - 1$ entries in row n of the routing table each refer to a node whose nodeID matches the present node´s nodeID in the first n digits, but whose $n+1$th digit has one of the $2^b - 1$ possible other values than the $n+1$ the digit of the present node´s ID." [26]
Figure 4.2 shows an example routing table for a node with ID $65a1x$, with x representing an arbitrary suffix. In addition, Figure 4.3 shows the routing path of a message with ID $d46a1c$ send by a node with ID $65a1fc$. The routing of a message is done as follows: given a message m and a node r whose ID shares the same prefix of b digits. The current node r normally forwards the message m to a node s whose ID shares a prefix of at least $b+1$ digits with the ID of the message. If no, such node s can be found, the message is delivered to the current node r. So in each routing step more prefix information of a message´s ID is used leading to a distance reduction between the message and the receiving node. Following the routing scheme FreePastry delivers messages within a complexity of $O(logN)$ routing steps. In order to detect node failures, nodes periodically send keep-alive messages among each other. In case of a node crash, only $O(log_{2^b} N)$ messages are needed to update the routing tables of the affected nodes.

Chord

Another commonly used DHT is Chord [5],which was developed at MIT. Like FreePastry, Chord organizes peers in a m-bit circular ID space using consistent hash functions for mapping nodes to IDs. In contrast to FreePastry, every Chord node maintains a finger table with $O(logN)$ entries instead of a routing table, which is organized as follows: the k-th entry in

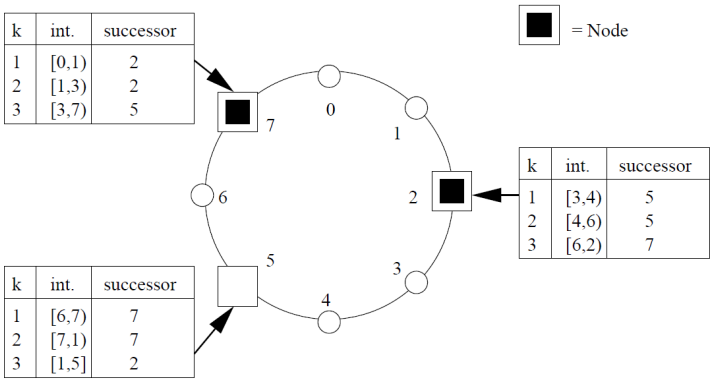


Figure 4.4: Example for a Chord Ring

the finger table of a node r is the smallest node s, such that its ID is greater or equal than $r + 2^{k-1}$ [5]. Each entry of the finger table at position k is called the order-k successor of node r. Another interpretation of the finger table is that the m entries of the finger table define m identifier intervals. The k-th interval of the finger table of node r could be calculated as follows: $](r + 2^{k-1}) \bmod 2^m, (r + 2^k) \bmod 2^m]$.
Routing in Chord is done as follows: a search request f initiated at a node r is handled by checking whether the ID of f is between r and its successor. If so, the search is forwarded to the successor of r and terminates. Otherwise, the query is forwarded "to the largest node in node r´s finger table that precedes f"[5]. Following this scheme, the search request is forwarded until the search terminates at the responsible node.
Figure 4.4 shows an example for a chord ring, containing three nodes with the IDs 2,5 and 7. Assuming a search with ID 6 initiated at node 2. First node 2 checks, whether the ID is between 2 and its successor (5). Since the ID is not between 2 and 5, the biggest node that precedes ID 6 is located, which is the node with ID 5. So the search request is forwarded to that node. The node now checks again, whether the search ID lies between his ID and the ID of its successor (7). As the search ID lies between the two IDs, the search request terminates at the node with the ID 7, which is responsible for the ID 6.[5]

4.3 Self-Optimization Cycle

As already mentioned in Chapter (2.1) at the beginning of this thesis, the goal is to develop self-optimizing and adaptive P2P systems that are able to react on changes in their dynamic and distributed environment by adjusting their system parameters. This approach is very similar to a feedback loop as known from automatic control engineering. Figure 4.5 shows the basic concept of a feedback loop, which can be used for controlling dynamic system behavior. The system output / system state is measured by a sensor. The output of the sensor together with a reference signal form the input for a controller component, which calculates a new setting for the system parameters. An example for such a dynamic system could be an automobile, where the driver provides a certain desired or referenced speed (reference signal) and where the current speed depicts the system output. The system parameter to be controlled in this scenario is the position of the engine´s throttle[33].

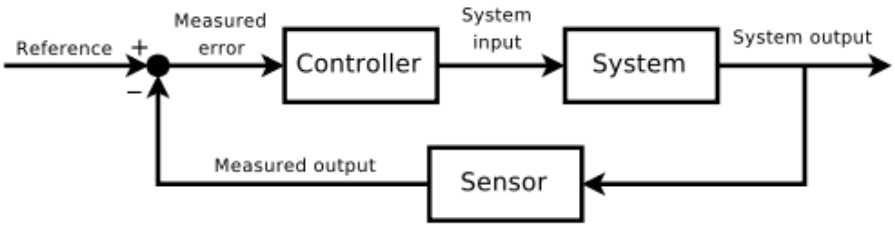


Figure 4.5: Feedback loop concept for controlling dynamic system behavior. Taken from: [33]

The challenge is to design a controller component, that is able to deliver new values for the system parameters in order to control the system. In case of P2P systems the parameters for controlling the system behavior have not been identified and explored in detail yet as well as their impact on the system stability and reliability.
In the following section the three steps of the optimization cycle are introduced and problems arising at each step as well as possible solutions are discussed.

4.3.1 Monitoring

While developing a framework enabling the monitoring of distributed systems like P2P systems, it is necessary to have a clear definition in mind about what monitoring is and which problems may occur during the monitoring process.
"Monitoring is an umbrella term for all types of direct systematic recording, monitoring or surveillance of an operation, system or process by means of technical aids or other observation systems. The objective of monitoring is to gain knowledge about a systems state in order to verify hypotheses and to gain a deeper understanding on the monitored object."[36]
There exists a great variety of application areas for monitoring such as Bio-Monitoring, System-Monitoring, System-Theory, Network-Monitoring and so on.
Monitoring implicates the measuring of data by using measuring devices, which need to be installed around, integrated into or applied to the object being monitored. This presence of a measuring device leads to one of the basic problems of measurement technology which is called Rückwirkungsabweichung [39]. Due to the presence of the measuring device, the system behavior is likely to change. E.g. given a sensor application which is able to gather the current CPU usage. By installing and running the sensor application, additional CPU power and memory is needed. So, the value for the CPU usage delivered by the sensor application includes the CPU usage caused by the sensor application itself. Hence, whenever a measuring device is used to capture an indicator, the indicator itself is affected by the presence of the measuring device due to a change in the system behavior. So the difficulty is to construct sensors such that they affect the target system as few as possible.

Measuring error

Another issue that must be kept in mind, is that measured data always might be erroneous. The question that arises is how far the measured value deviates from the actual value for some given indicator. In order to calculate the absolute deviation F the following formula is used:

$$F = x_m - x_a$$

with x_m denoting the measured value and x_a denoting the actual value. In addition, the relative deviation f can be calculated by using the following formula:

$$f = \left(\frac{x_m - x_a}{x_a} \right) * 100$$

The problem is, that the actual value x_a is unknown in most cases. So it is necessary, to determine how great the impact of the measuring error on the measuring result is. Assuming a normal distribution of the measured values v_i the mean $\overline{v}$ and the empirical standard deviation s could be calculated as follows:

$$\overline{v} = \frac{1}{N} \sum_{i=1}^{N} v_i$$

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} \left(v_i - \overline{v} \right)^2}$$

The mean and the standard deviation are the parameters for the normal distribution. Due to the finite number of measuring values the mean will spread, which is called uncertainty (u).

$$u = \frac{1}{N} * s$$

The uncertainty will decrease with an increasing number of measuring values (N). The uncertainty and the mean together denote the range $\overline{v} - u \ldots \overline{v} + u$ in which the actual value x_a is expected[35].

4.3.2 Data Analysis

After gathering data about the global system state, the next step is to analyze the data in order to classify and recognize the current system state. To overcome this problem data mining algorithms could be applied on the monitoring data to compute an analytic model for recognizing the current system state.
Therefore it first is necessary to define the entities the data mining algorithms should deal with. The entities can be found by taking a closer look on the Feedback Loop scenario mentioned above. The first entity are the system parameters like the size of the routing table. All system parameters together form a setting for the system. The second entity are the system indicators, which can be measured by a sensor such as the query response time. All system indicators together form the current system state, the third possible entity for data mining techniques. The challenge is to uncover the correlations between the system parameters and the system indicators. The values of the system indicators are expected to be a direct result of the chosen values for the system parameters.
A possible simple approach for using data mining algorithms in order to obtain an analytic model could be as follows: First it is necessary to identify possible system parameters like the size of the routing table, the message size or the message hop count. In addition, the important measuring indicators must be defined. Afterwards sensors must be applied to the system in order to measure these system indicators. In a next step the P2P system is started using a predefined

setting of system parameters and snapshots of the system state, represented by a collection of system indicators, are made periodically. Each snapshot is classified with a label indicating e.g. whether the system was stable or unstable. Following this scheme several settings for system parameters could be tested and the resulting monitoring data including the class information (the system state) are stored. Afterwards, this data could be used for calculating a system model by training a rule set or a decision tree [34]. The system model could be used to automatically "recognize" whether the system state is e.g. stable or unstable. Decision rules learned by a rule learner could e.g. look like as follows:

$$\text{Parameter-A}=3 \;\&\&\; \text{Indicator-B}>20 \rightarrow \text{Stable-System-State}$$

Another possible approach for an analytic model could as follows: First, desired thresholds for each system indicators are chosen. In the second step several settings for the system parameters are tested again and the resulting values for the system indicator are stored. Using this data a rule learner could be used to learn a rule set of the form:

$$\text{Parameter-A}<a \;\&\&\; \text{Parameter-B}>b \rightarrow \text{Indicator-A}>a$$

$$\text{Parameter-C}>=c \;\&\&\; \text{Parameter-D}<d \rightarrow \text{Indicator-B}<b$$

Given this type of rule deviations of the system parameters could be detected easily. One problem that arises in this context is, that some of the system indicators may be system parameters as well and vice versa like the number of peers, which typically cannot be adjusted due to a miss of a central authority.

4.3.3 Decision Making and Deployment

In the last section data mining techniques were introduced to compute an analytic model, that describes the system behavior and delivers informations about which system indicator currently is out of range. The question that arises is, how the system could be optimized based on the information delivered by the analytic model. Following the optimization cycle as mentioned in Chapter 2.1 a decision model must be developed, which is be able to take decisions based on the information delivered by the analytic model.
When talking about making decisions in a distributed system like P2P system the following basic questions should be kept in mind:

- How are decisions made?
- Where are decisions made?
- How do decisions look like?
- How can decisions be deployed to the network?

A first possible decision model could be realized using the decision rules found be the first data mining algorithm suggested above: In case that the analytic model detects an unstable system state, the decision model selects all rules from the rule set with head "Stable-System-State". Among these rules, the rule, nearest to the current indicator setting according to some metric,

is selected. Finally, all system parameters occurring in the selected rule are adjusted according to the rule.
When looking at the optimization cycle it becomes obvious that it is very similar to the management of a company, where businessmen make decisions in order to optimize their business processes with the objective of gaining more profit. For this reason, the most common management models have been analyzed and evaluated and how they could be applied to P2P systems in [44]. One basic requirement for the decision making process is, that it should be done in a decentralized fashion in order to avoid interferences with the distributed nature of P2P systems.

4.4 General Requirements for a Distributed Information Management Framework

In the following the requirements for a Distributed Information Management Framework are lined out in order to understand the challenges and problems that arise within the development.
In general, one of the basic requirements for a framework is the support of modularity and extendability, which means that additional functionality can be added easily. In order to enable application-specific monitoring, the framework should be able to be integrated into existing applications without great overhead. Beside this general requirements for a framework there arise a number of additional requirements for a Distributed Information Management System as the framework is running on top of a DHT in a distributed environment.
One common problem that arises in a distributed environment is, that participating nodes join and leave the network periodically. For this reason the Distribute Information Management Framework should be able to handle churn, which means that the framework should continue delivering useful data even if nodes join and leave.
Furthermore, the framework should scale in the number of peers participating in the network and in the number of attributes being aggregated. To avoid bottlenecks and stability problems, the framework should support load balancing techniques to ensure a distribution of the system load. In summary one could say that the framework built on top of the DHT should inherit all its properties like scalability, robustness and self-organization.
In order to support different underlying DHT implementations, overlay-independency is desired.
To ensure fast results about the current system state, the time needed for an information item to travel trough the aggregation topology should be as low as possible especially in terms of high churn rates, when the aggregation topology changes fast. Hence, a logarithmic scale is desired for the aggregation process.
As already mentioned in Chapter 4.3.1 the impact of the monitoring / sensor application should be as low as possible in order to ensure unbiased measuring results. This means that a monitoring application should cause as few resource consumption as possible to avoid interferences with other applications.

4.5 Data Aggregation

The core functionality of the developed prototype is gathering and aggregating data about the current system state. There exist two fundamentally different approaches for aggregating data.

Aggregation Functions

The first one are aggregation functions, which compute a single numeric value based on several numeric input data items. An overview of the used aggregation functions is given in Table 4.1. Every aggregation function used must fulfill the *hierarchical computation property* as mentioned in [42] in order to ensure useful results while being recursively computed. Aggregation functions have the advantage, that the size of the aggregate remains constant during the computation. Aggregation functions are used to compute several statistics about the current system state like current CPU usage averaged over all nodes in the network or the sum of the available disc space of all nodes.

List-based Aggregation

The second aggregation type is called list-based aggregation, which means that several single data items are merged / concatenated to a list. In contrast to aggregation functions, the size of the aggregate increases constantly with an increasing number of data items being merged. Due to the increasing size, the bandwidth needed for computing the list-based aggregate will increase, which leads to performance bottlenecks as weak peer might be unable to handle great quantities of data. To overcome this problem well known techniques for data reduction like data compression, incremental updates or filter mechanisms could be used.
To used data compression in Java the DeflaterOutputStream could be used as shown in Listing 4.1, which leads to a data reduction of up to 70%.

```
1 Deflater f = new Deflater(Deflater.BEST_COMPRESSION);
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
3 DeflaterOutputStream dos = new DeflaterOutputStream(baos,f);
  ObjectOutputStream oos = new ObjectOutputStream(dos);
5 oos.writeObject(s);
  oos.flush();
7 dos.finish();
```

Listing 4.1: Usage of the DeflaterOutputStream

By using incremental updates only significant new data items are sent in order to save network bandwidth consumption. This approach might decrease the amount of data being sent dramatically but also leads to consistency problems especially in terms of churn as the incremental update and the data stored on the receiving node might not fit together.
The third approach for data reduction are filter techniques, where data items are ranked according to some score function. Only the best n data items are sent and added to the aggregate. The remaining data items are neglected in order to reduce the size of the aggregate. The challenge is to create score functions such that only "important" and significant data items are integrated into the aggregate.
List-based aggregation could be used to enable capacity-based peer search in order to find a subset of peers offering specific resource called capacities like free disc space or CPU power. This implicates that peers have to include information about the capacities they are offering, into the list-based-aggregate.

Aggregation Functions
$\min(a_1, \ldots, a_n) = \min(a_1, \min(a_2, \ldots \min(a_{n-1}, a_n) \ldots))$
$\max(a_1, \ldots, a_n) = \max(a_1, \max(a_2, \ldots \max(a_{n-1}, a_n) \ldots))$
$sum(a_1, \ldots, a_n) = \sum_{i=1}^{N} a_i$
$count(a_1, \ldots, a_n) = n$
$mean(a_1, \ldots, a_n) = \frac{sum(a_1, \ldots, a_n)}{count(a_1, \ldots, a_n)}$
$sum_squares(a_1, \ldots, a_n) = \sum_{i=1}^{N} a_i^2$
$variance(a_1 \ldots a_n) = \frac{sum_squares(a_1, \ldots, a_n)}{N} - \left(\frac{sum_squares(a_1, \ldots, a_n)}{N}\right)^2$
$standard_deviation(a_1, \ldots, a_n) = \sqrt{variance(a_1, \ldots, a_n)}$

Table 4.1: Aggregation functions

4.6 Capacities And Statistics

The following section gives an overview of the capacities and statistics being used. Capacities are aggregated using the list-based-aggregation process whereas the statistical values are computed using the aggregation functions as explained in Section 4.5. The challenge is to discover those statistical indicators that are relevant for the optimization process. The goal is to find statistical indicators that discriminate as well as possible between the given system states. Table 4.2 shows the statistical indicators that are currently supported by the developed framework.

Statistic	Unit	Description
TIMESTAMP	sec	the average CPU utilization averaged over the last 10 measuring cycles
BANDWITH_UTILIZATION_ UPLOAD	MB	total amount of data sent
BANDWITH_UTILIZATION_ DOWNLOAD	MB	total amount of data received
BANDWITH_UTILIZATION_ UPLOAD_ZIPPED	MB	amount of zipped data sent
BANDWITH_UTILIZATION_ DOWNLOAD_ZIPPED	MB	total amount of zipped data received
BANDWITH_UTILIZATION_ UPLOAD_PER_SEC_AT_LEVEL_X	KB/s	average upload utilization at level X of the SkyEye.KOM tree
BANDWITH_UTILIZATION_ DOWNLOAD_PER_SEC_ AT_LEVEL_X	KB/s	average download utilization at level X of the SkyEye.KOM tree
BANDWITH_UTILIZATION_ UPLOAD_PER_SEC_ZIPPED_AT_ LEVEL_X	KB/s	average upload utilization cause by zipped content at level X of the SkyEye.KOM tree
BANDWITH_UTILIZATION_ DOWNLOAD_PER_SEC_ZIPPED_ AT_LEVEL_X	KB/s	average download utilization cause by zipped content at level X of the SkyEye.KOM tree
BANDWITH_UTILIZATION_ UPLOAD_PER_SEC_ZIPPED	KB/s	average upload utilization per sec
BANDWITH_UTILIZATION_ DOWNLOAD_PER_SEC_ZIPPED	KB/s	average download utilization per sec
BANDWITH_UTILIZATION_ UPLOAD_AT_LEVEL_X	MB	total amount of data uploaded at level X of the SkyEye.KOM tree
BANDWITH_UTILIZATION_ DOWNLOAD_AT_LEVEL_X	MB	total amount of data downloaded at level X of the SkyEye.KOM tree
BANDWITH_UTILIZATION_ UPLOAD_ZIPPED_AT_LEVEL_X	MB	total amount of zipped data uploaded at level X of the SkyEye.KOM tree
BANDWITH_UTILIZATION_ DOWNLOAD_ZIPPED_AT_LEVEL_X	MB	total amount of zipped data downloaded at level X of the SkyEye.KOM tree
AVERAGE_UPLOAD_ NETWORK_USAGE	MB	average upload network usage of the local computer
AVERAGE_DOWNLOAD_ NETWORK_USAGE	MB	average download network usage of the local computer

continued on next page

Statistic	Unit	Description
AVERAGE_TOTAL_NETWORK_USAGE	MB	average total network usage of the local computer
AVERAGE_CPU_USAGE	MB	the average CPU utilization
SUM_HOP_COUNT	#	sum of the hop counters of all received messages
AVERAGE_HOP_COUNT	#	sum of the hop counters of all received messages over the number of received messages
NUMBER_OF_SEND_MESSAGES	#	the total number of all sent messages during the uptime of the node
NUMBER_OF_RECEIVED_MESSAGES	#	the total number of all received messages during the uptime of the node
NUMBER_OF_NODES	#	the total number of nodes in the network
UPTIME	min	the uptime of the node
HEAP_MEMORY_MAX	MB	max granted heap memory by the VM
HEAP_MEMORY_INIT	MB	heap memory usage at startup
HEAP_MEMORY_USED	MB	current heap memory usage
NON_HEAP_MEMORY_MAX	MB	max granted non heap memory usage by the VM
NON_HEAP_MEMORY_INIT	MB	non heap memory usage at startup
NON_HEAP_MEMORY_USED	MB	current non heap memory usage
FREE_DISK_SPACE	MB	available disc space
NUMBER_OF_PROCESSORS	#	number of processors installed at the local computer

Table 4.2: List of statistics

Statistical values are always numeric values whereas capacities could have a nominal or numeric expression. Currently the capacities listed in Table 4.3 are supported.

Capacity	Unit	Description
PEER_ID		the current ID of the peer normalized to 0-1 range
OPERATING_SYSTEM		the operating system the node is running
RESPONSIBILITY_START		the start ID of the node´s responsibility range normalized to 0-1 range
RESPONSIBILITY_END		the end ID of the node´s responsibility range normalized to 0-1 range
ARCHITECTURE		the system architecture the node is running on
FREE_DISK_SPACE	MB	the available disc space on the node
TREE_LEVEL		the current tree level of the node within the Sye-Eye tree
PARENT_COORDINATOR		the node´s parent coordinator within the Sky-Eye.KOM tree

continued on next page

Capacity	Unit	Description
MAX_NETWORK_TOTAL	KB/s	the total maximum bandwidth offered by the node
MAX_NETWORK_UPLOAD	KB/s	the maximum upload bandwidth offered by the node
MAX_NETWORK_DOWNLOAD	KB/s	the maximum download bandwidth offered by the node
VM_VERSION		the current version of the VM the framework is currently running in

Table 4.3: List of capacities

4.7 Aggregation Schemes

In the following section the most common concepts for aggregating data are discussed in order to provide a deeper understanding on the advantages and disadvantages that every approach appears with. When looking at the related work as discussed in Chapter 3 it is noticeable that most systems depend on a tree-based scheme for data aggregation ([22], [3], [43], [42],[24], [24], [11]). This is not astonishing as a tree topology seems to be the natural expression for aggregating data in a structured and organized fashion. The challenge is to design an aggregation scheme, that is able to compute aggregates over a large number of nodes while assuring consistency in terms of joining and leaving peers causing fluctuations in the available aggregation data. In [2] some additional requirements for the aggregation process are presented: The aggregation scheme should support generality, which means that it should support any aggregation function. In addition, the aggregation scheme should deliver results within a certain amount of time without causing deadlocks. In other words, the aggregation process should support the termination property. Furthermore, the results delivered by the aggregation scheme should be valid according to a definition of practical validity and consistency. Seeking for strong validity and consistency is unrealistic as in it cannot be achieved in a distributed environment due to joining and leaving nodes. At last the aggregation scheme should be independent from the underlying P2P protocol being used and the communication and computation overhead caused by the aggregation scheme should be as low as possible.

Flooding-based Aggregation Schemes

Beside the tree-based approaches some flooding-based approaches have been introduced in [13] and [2]. The naive approach for gathering data about a unstructured P2P network is to initiate a query at a node and flood / multicast it through the network as known from Gnutella (see Figure 4.6). Every peer satisfying the query, sends his answer to the requesting node. It is obvious that this approach will lead to performance problems "as possibly each peer in the network may respond to the query"[2]. With thousands of responding nodes it is very likely that the receiving node will collapse.

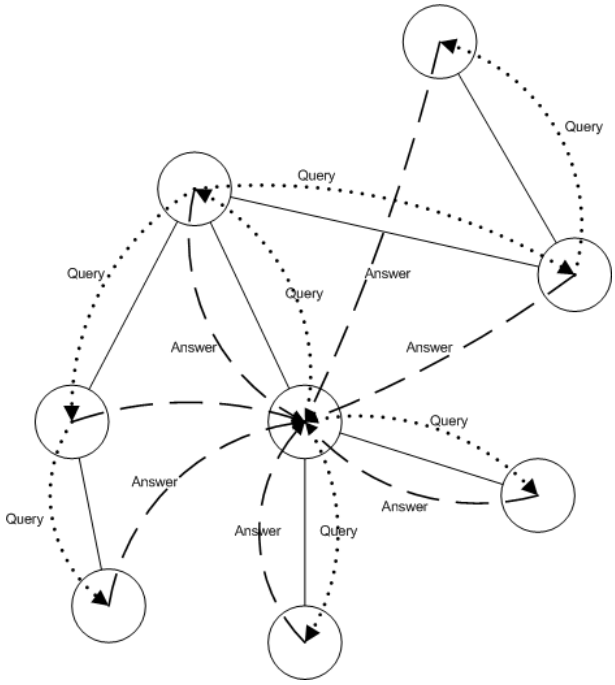


Figure 4.6: Concept of flooding

So it is necessary to define a more organized flooding-based scheme for aggregation. For this reason [2] introduced three approaches for flooding-based aggregation: a single-tree, a propagate-to-all and a multi-tree approach. All three approaches have in common, that the aggregation process is split up into two phases: a broadcasting phase within the aggregation topology is constructed and an aggregation phase within the aggregate is computed using the topology constructed in the previous phase. For each initiated query, a new topology is constructed using broadcast messages. This depicts the great difference to the later discussed tree-based aggregation schemes in Section 4.7.
Using the first approach, a single-tree topology is constructed. "Each node u in the network identifies as its parent node w that first transmitted the broadcast message to u"[2]. In the second step the aggregate is computed using the constructed single-tree. Single-tree approaches suffer from being very sensitive to node failures. In case that a node crashes, all data of the subtree rooted by the node will be lost. Conversely, tree approaches cause low communication and computation overhead as every information is sent and computed only once.
To overcome the problem of failing nodes during the aggregation process the propagate-to-all approach was introduced. A querying node q starts the aggregation process by sending a broadcast message to all its neighbors. All nodes are considered to be inactive until they receive the broadcast message. After receiving the broadcast message, the nodes become active by continuing forwarding the broadcast message and sending their aggregation data to their neighbors. A node receiving aggregation data recalculates its local value and broadcasts the new value to its neighbors. Each node stops broadcasting and calculating his aggregation data after $2D\Delta t$ times, with D denoting the diameter of the network and Δt denoting the time needed for one message hop in the network. This approach is highly reliable in terms of churn but causes a great communication and computation overhead as data is aggregated on every node in the network. In addition, the approach requires detailed knowledge about the diameter D of the network, which will not be available in most cases.

The multi-tree approach introduced by the authors is a tradeoff between high communication, computation costs and reliability. Starting at a querying node q, k independent spanning trees are constructed. This is done as follows: to each node v in the network a level $l(v)$ is assigned to. Initially, the root q has the level $l(q) = 0$ and all other nodes have the level $l(v) = \infty$. "When a node v receives the broadcast message from a node u with level $l(u)$, it first checks whether its level is $l(v) = \infty$. If so, the node sets its own level to $l(v) = l(u) + 1$ and forwards the query with level $l(v)$ to all its neighbors. Otherwise, v stores the level of u as $l(u)$"[2]. Afterwards, each node v selects k random parent nodes from the subset given as $P(u) = v \in N(u)|l(u) < l(v)$. Following this scheme, k independent spanning trees are constructed, which offers higher reliability in case of failing nodes as there exist multiple paths from a node to the root.
All introduced flooding-based approaches can be used within unstructured P2P systems as no overlay is needed.

Tree-based Aggregation Topologies

In contrast to the flooding-based approaches mentioned in the last section, tree-based approaches maintain a fixed topology for aggregating data. For each query initiated on a node the same topology is used. Mostly the aggregation and query process are separated, which means that data is aggregated not just only in case that a query is initiated. Typically, the nodes in the tree pull or push their data periodically up in the tree until it reaches the root, where the global system statistics can be obtained.

Single-Tree Aggregations Schemes

There exist two different approaches for organizing nodes and their data in a tree-based fashion: single and multiple tree approaches. Using the first one, a single tree is constructed for all nodes. The advantage of this approach is, that the number of messages needed for gathering and aggregating the nodes data is within $O(N)$ with N denoting the number of nodes. In addition, querying data in a single tree using complex queries can be done easily as the query must be resolved only within one tree. Beside these advantages the single-tree approach is sensitive to node failures and from a certain point of view the root of the tree depicts a single-point-of-failure. If a node crashes, all the data of the subtree rooted by the node will be lost. Furthermore, it is likely that nodes near the root and especially the root itself will suffer from high bandwidth consumption as the amount of data will increase towards the root as mentioned in Chapter 4.5. In case that weak peers are chosen in the higher levels of the tree, it is possible that the aggregation topology becomes either unstable or even collapses.

Multi-Tree Aggregations Topologies

Using the second approach, a single tree for each attribute is constructed. Consequently, this will result in a better load balancing as the load is spread among several peers. In contrast, the number of messages needed will be within a complexity of $O(A * N)$ with A denoting the number of attributes as for each attribute and tree the information must be encapsulated in a separate message. Resolving complex queries spanning over multiple trees is more difficult that in a single tree scenario as the querying node has to initiate a separate query for each attribute being requested. Furthermore, after receiving the query results for each attribute the node must merge all the data in order to obtain the answer for the initial complex query. This approach also

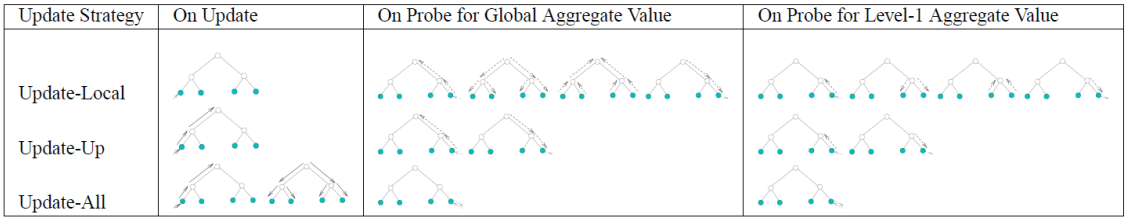


Figure 4.7: Overview of update/propagate strategies. Taken from: [20]

suffers from the single-point-of-failure issue and is also sensitive to node failures as the crash of single node will lead to a data loss and inconsistent information about the global system state.

4.8 Tree-based Aggregation And Propagation Schemes

As discussed in Chapter 4.7 mostly tree-based aggregation topologies are used in Distributed Information Management Systems. While computing aggregates in order to obtain information about the global system state, the question arises, which update- propagate strategy should be used. In [20] three fundamental strategies for updating and propagating data in a tree topology are introduced. Each strategy has its advantages and disadvantages with respect to read- and write accesses, the computation overhead caused and the time needed to answer potential queries. Figure 4.7 gives an overview about the three possible strategies and shows which steps are necessary in case that data is updated on a node or a query is initiated requesting the global aggregate or the aggregate of a specific tree level. In case of the *Update-Local* strategy only the local node updates its data. If a query for the global systems state is initiated it is necessary to compute the global aggregate first. So this strategy has the advantage of causing a low computation overhead as the global aggregate is not computed until it is requested by some query. On the contrary the time needed for answering potential queries will be high as the computation and propagation process cannot be done instantly.
In contrast to the *Update-Local* strategy, the *Update-All* strategy aggressively computes and propagates changes immediately through the tree which causes additional computation- and communication overhead. Queries requesting the global aggregate can be answered immediately as the requested aggregate is available on every node.
The intermediate strategy between the two former approaches is the *Update-Up* strategy. In case of changing node data, the update is send immediately up in the tree until it reaches the root where the new global system state is computed. In contrast to the *Update-All* strategy the global system state is not propagated down the tree afterwards. In case that a query requests information about the global system state, the query is forwarded up to the root, where the query is resolved and sent back the query originator. So the *Update-Up* strategy depicts a compromise between the caused communication and computation overhead and the time needed for answering potential queries.
Finally, the question arises which strategy should be used? The answer depends on how often queries are initiated requesting information about a certain aggregate. In case that the information is needed periodically on several different nodes, a *Update-All* strategy would be the best approach. Conversely, if the data is hardly requested, the *Update-Local* strategy depicts the best solution.

4.9 Basic Problems

In the following section, basic problems of Distributed Information Management Systems (DIMS) are discussed.

Heterogeneous peers

One of the fundamental problems of DIMS arises from the fact, that participating peers in the network can be heterogeneous. In case of DIMS this heterogeneity may cause data loss as aggregation data cannot be processed by peers, which are unable to handle aggregation data due to a miss of the required functionality. So it is necessary to establish mechanisms that avoid data loss due to heterogeneous peers. This can be achieved by establishing mechanisms, which allow nodes to determine whether other nodes are offering a required functionality or not. In case that a peer does not support the required functionality, the sending node can choose another peer according to some conflict resolution mechanism. In context of tree-based aggregation, a possible conflict resolution could be, that the sending node skips the level of the incapable node and forwards the data to the parent of the incapable node.
Another solution is to ensure that only capable peers are connected to each other over an application specific overlay. This approach comes with the drawback, that only a part of the system state can be obtained as incapable peers are neglected.

Security

Another often in context of P2P systems neglected problem is that of security. In all implementations of DIMS, as discussed in Chapter 3, the entire communication is realized without encryption, which becomes a critical issue in case that the global system statistic contains sensitive data. So it is necessary to establish mechanisms which ensure a secure and encrypted communication like symmetric and asymmetric encryption techniques.
In addition, the current DIMS do not check whether the statistics or capacity data included by a node into the aggregate is useful or not. So it possible for a single node to corrupt the entire system statistic by including incorrect or junk data like negative values for the current CPU usage. To avoid this kind of "spam" it is necessary to establish control mechanisms in order to check whether the included data is useful or not.
Furthermore, in the current implementation of the DIMS, malicious nodes mostly have the possibility to flood the network with junk data. By establishing filter mechanisms and excluding malicious peers from the network it might be possible to solve this problem. Beyond that, malicious nodes may perform Denial-of-Service attacks by requesting specific data items thousands of times. Doing so, the responsible node for the requested data item is likely to collapse.

Consistency and validity of data

As nodes join and leave the network, enforcing strict consistency and validity within the collected data is a hard task and nearly impossibly to achieve. So the authors of the current state-of-the-art DIMS introduced a more practical definition for consistency and validity, that can be realized within a distributed environment. One possible definition for *practical validity* is given in [2].

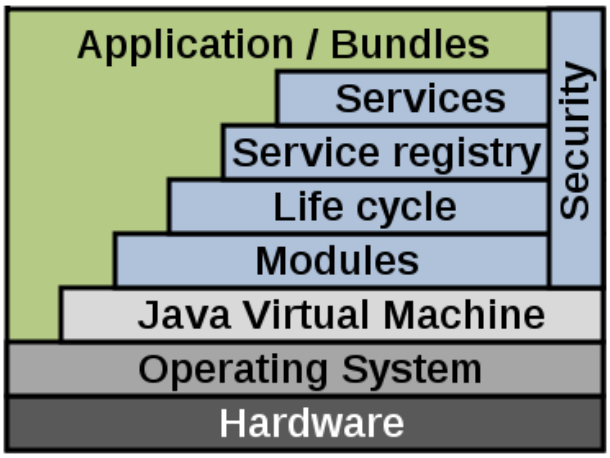


Figure 4.8: The OSGI framework. Taken from: [38]

Evaluation of statistics and capacity data

The evaluation of the gathered systems statistics and capacity data also is a hard task, because global knowledge about the real statistics and capacity data is only available in context of simulations. In case that the monitoring is used within a real world application, the gathered data cannot be evaluated in detail due to the missing reference. The only aspect that can be evaluated is whether the collected data is useful or not. Nevertheless, there exist two possible approaches for dealing with this issue in context of large scale testbeds. The first one is to establish an additional central logging server such that every peer in the network sends and further copy of its logging data to it.Afterwards, the logging data of the DIMS and the logging data of the central server could be compared to each other.
A second solution would be, to install several different DIMS simultaneously on each peer of the network. Afterwards, the results of each DIMS could be compared to each other in order to get an impression on the validity of the data. In case that different DIMS all deliver the same result for the system state, the probability that the data is valid is high.

4.10 OSGI

The following section deals with an introduction to the OSGI [4] framework Equinox [8] as the developed monitoring framework is entirely built on top of it. The OSGI (Open Service Gateway Initative) Service Platform is a dynamic module system and was initially developed for embedded systems. However, the problem of increasing software complexity and monolithic software architectures is very common and for this reason the OSGI Service Platform has become quite popular in many application areas. Eclipse, a widely used IDE for developing Java applications, is entirely built on top of Equinox. "The OSGI standard provide an environment for the modularization of applications into smaller bundles. Each bundle is a tightly-coupled, dynamically loadable collection of classes, jars, and configuration files that explicitly declare their external dependencies (if any)"[37].

4.10.1 The Architecture

Figure 4.8 shows an overview of the Open Service Gateway Initiative Architecture. The architecture is divided into several layers. Each layer summarizes functionality that logically belongs together. The two bottom layers are the *Hardware* and the *Operating System* forming the base of the architecture. On top of the *Operating System* layer the *Java Virtual Machine* layer is located, defining the Runtime Environment for Java applications.

Modules

The *Module Layer* is the first layer belonging to the OSGI framework and defines bundles as elementary modularization unit. Technically, a bundle is a jar file containing class files and resources that offer a certain functionality. In addition, the jar file contains a Bundle Manifest that contains meta information about the bundle called Manifest Header like required packages, a unique bundle identifier, exported functionality / packages and so on. Listing 4.2 shows an example *Manifest.MF* file, which is located in the *META-INF* folder of every bundle.

```
1  Manifest-Version: 1.0
   Bundle-ManifestVersion: 2
3  Bundle-Name: FreePastryCommunication
   Bundle-SymbolicName: quap2p.dims.communication_freepastry.osgi.core
5  Bundle-Version: 1.0.0
   Bundle-Activator: quap2p.dims.communication.osgi.core.Activator
7  Bundle-ActivationPolicy: lazy
   Bundle-RequiredExecutionEnvironment: JavaSE-1.6
9  Import-Package: org.osgi.framework;version="1.3.0",
    org.osgi.util.tracker;version="1.3.3"
11 Require-Bundle: quap2p.dims.util.osgi.core;bundle-version="1.0.0"
   Bundle-ClassPath: lib/FreePastry-2.1beta.jar,
13  .
   Export-Package: quap2p.dims.communication.osgi.core.api
```

Listing 4.2: An example for a MANIFEST.MF file

In order to get a bundle running properly, all imported packages (Import-Package directive) and required bundles (Require-Bundle directive) defined in the Manifest must be present at runtime. One of the characteristics of the OSGi framework is, that dependencies are managed explicitly, which means that if a bundle imports for example a package *foo.par* there must be a second bundle that exports a package named *foo.par*. Otherwise the import remains unresolved and the bundle can not be started. [9]

Life Cycle Management

The above mentioned *Modules* layer defines only static aspects of the bundle concept whereas the *Life Cycle* layer defines all the dynamic aspects of a bundle. The layer specifies the life cycle of a bundle within the OSGi framework with all its possible states. Figure 4.9 shows a state machine of the life cycle of an OSGi bundle. After installing a bundle, the OSGI service platform checks whether all required imports and bundles are present. If so, the bundle gets *resolved*. Otherwise the bundle remains in the state *installed*. In case that all dependencies can be resolved, the bundle can be started, which causes the bundle to changes its state from resolved to starting. During the starting process the service platform checks whether the user has enough rights to start the bundle. If so, the bundle changes its state to *active*. Otherwise, the

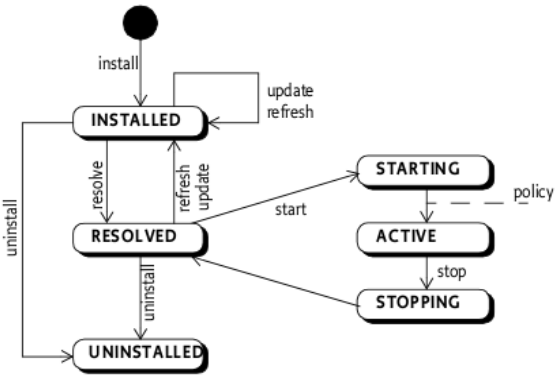


Figure 4.9: The life cycle of an OSGI bundle. Taken from: [9]

bundle state remains resolved. In case that a bundle is stopped it changes its state from active to the intermediate state *stopping* and finally to *resolved*. During the intermediate state *stopping* the bundle may perform several cleanup steps e.g. unregistering services at the f are possibly provided by the bundle. [9]

Service Registry

The *Service Registry* layer is responsible for managing services provided by bundles. Every bundle can register its services at the Service Registry with a specific interface name, such that other bundles can query the Service Registry for the registered service. Bundles are able to register and unregister services at runtime. This means that a specific service may not always be available. In order to ease the work with dynamic services, the OSGi framework offers the concepts of *Service Listeners*, *Service Trackers* and *Declarative Services*.

```
//registering ILoggingService at the Service Registry at bundle A
ILoggingService loggingService = new LoggingServiceImpl();
context.registerService(ILoggingService.class.getName(), loggingService,new
    Hashtable());
loggingService.logMessage("Logging Service registered!", LOGGING_LEVEL.INFO,
    Activator.class);

//querying ILoggingService at the Service Registry at bundle B
ServiceTracker loggingTracker = new ServiceTracker(context, ILoggingService.class.
    getName(), null);
loggingTracker.open();
ILoggingService service = (ILoggingService) loggingTracker.getService();
if (service != null){
  service.logMessage("Hello again...",LOGGING_LEVEL.INFO, Activator.class);
}
```

Listing 4.3: ServiceTracker example

Listing 4.3 shows an example of a bundle *A* registering a logging service at the Service Registry under the name *ILoggingService*. The requesting bundle *B* first initiates a ServiceTracker for the *ILoggingService* and queries the Service Registry for the required service by invoking *loggingTracker.open()* and *loggingTracker.getService()*. Afterwards, it is necessary to check whether

the reference returned by the *getService()* method is null as the ILoggingService may not be available at the Service Registry.

Service Layer

The topmost layer of the OSGI architecture contains all the functionality for creating services, that can be registered and unregistered at the service registry. There exist two different approaches for registering and unregistering services. The first one uses the *org.osgi.framework.BundleActivator* interface, which offers two methods *start(BundleContext context)* and *stop(BundleContext context)* for starting and stopping a bundle.

```
public class Activator implements BundleActivator {

  public void start(BundleContext context) throws Exception {

    ITopologyManagementService service = new TopologyManagementServiceImplSykEye(
        context);
    context.registerService(ITopologyManagementService.class.getName(),
        service, new Hashtable());

    ServiceReference ref = context
        .getServiceReference(ILoggingService.class.getName());
    ILoggingService serv = (ILoggingService) context.getService(ref);

  }
  public void stop(BundleContext context) throws Exception {
    ServiceReference ref = context
    .getServiceReference(ILoggingService.class.getName());
    context.ungetService(ref);
  }
}
```

Listing 4.4: Activator example

Listing 4.4 shows an example implementation of an Activator class, which registers and unregisters an instance of the *ILoggingService* in case that the bundle is started respectively stopped. In order to find the implementation of the Activator class, the Manifest.MF file contains the *Bundle-Activator* Header, where the location of the Activator class is specified (cf. Listing 4.2).
The second approach for starting an OSGI bundle is to use *Declarative Services*. In order to use Declarative Services within Equinox the add-on bundle *org.eclipse.equinox.ds* must be installed. The general concept of Declarative Services are the Service Components which consist of two parts: a class file containing the service implementation and a service description given as a XML file, which contains e.g. information about whether the component declares dependencies to other services or offers a service by itself. In case that a Service Component references other services, the service description contains the definition for the *bind()* and *unbind()* methods for passing the required reference to the external service. The XML file containing the service description must be located in the OSGI-INF folder of a bundle.
All Service Components are initiated and managed by the Service Component Runtime (SCR). In case that a Service Component needs to be started, the SCR checks, whether all dependencies to other services can be resolved. If so, the service component is started and a reference is passed to all other service components that reference the started service component by invoking the components corresponding *bind()* method. Figure 4.10 shows an example of a declarative

service in OSGI. The "Foo" bundle offers a specific service which is consumed by another bundle called "Bar". In case that an instance of the service provided by the Foo bundle is registered at the Service Registry the *setFoo()* method of the Bar bundle is invoked.
Declarative Services should be used in case of a high number of bundles being used because they minimize the memory and startup time needed by applications as just required services are activated. Unused services remain inactive.

4.10.2 Bundle Dependencies

As already mentioned, the OSGI framework offers two possibilities for defining dependencies to other bundles or packages. Originally, the OSGI specification only contained the Manifest Header *Import Package*. With Release 4 of OSGI the *Require Bundle* Manifest Header was added. Although these two possibilities for defining dependencies look very similar, there are fundamental differences between these two techniques. The *Import Package* Header defines a dependency to a external package and does not care about which bundle is offering the required functionality. Conversely, the *Require Bundle* Header defines dependencies to other bundles and causes a bundle to import all packages that are exported by the required bundle. The difference between the two approaches is, that the *Require Bundle* Header defines a dependency to a concrete implementation offered by a specific provider (bundle). In contrast to that, the *Import Package* Header only specifies the name of the package to be imported and does not indicate which provider (bundle) should be used. So the Import Package Header depicts a more flexible way of defining dependencies.

When developing an application using OSGI and defining dependencies to other packages, it is important to avoid cyclic dependencies between bundles as shown in Figure 4.11 as they are prohibited by the OSGI framework. This prohibition makes fully sense when thinking about the idea of creating modularized software architectures fulfilling the requirements of high cohesion, low coupling and unity[1]. In case that two bundles depend on each other causing a cyclic decency, the functionality of the two bundles should be merged to a single bundle.

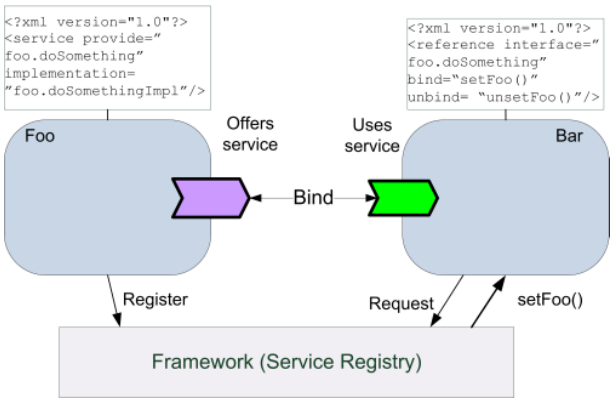


Figure 4.10: Example of declarative services in OSGI. Taken from: [1]

4.10.3 The OSGI Console

The OSGI Console depicts a simple way for managing OSGI bundles. The OSGI Console offers a variety of commands e.g. for installing, starting and stopping bundles. In addition, commands for debugging and displaying unsatisfied bundle dependencies are provided. In case that a developer wants to create a console application, the OSGI console can be extended by custom defined commands. Bundles defining custom commands need to implement the *CommandProvider* interface and must register themselves as *CommandProvider* at the Service Registry as shown in Listing 4.5.

```
public class MyCommandClass implements CommandProvider {

  public MyCommandClass(BundleContext context) {
      // register command at registry
    context.registerService(CommandProvider.class.getName(), this, null);
  }
  //the new command
  @Override
  public void _MyNewCommand(CommandInterpreter ci) {
    SwingUtilities.invokeLater(new TreeWindow(this));
  }
}
```

Listing 4.5: Registering a bundle as CommandProvider

In addition, all new commands defined by the class implementing the *CommandProvider* interface have to fulfill the method signature *public void _<CommandName> (CommandInterpreter ci)*.

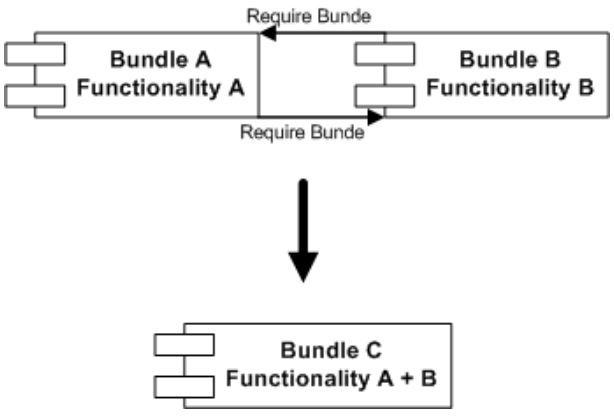


Figure 4.11: Cyclic dependencies in OSGi

5 SkyEye.KOM

This chapter deals with the mechanism implemented in the framework for gathering system statistics and capacity data of a P2P system called SkyEye.KOM which was introduced in [11].

5.1 Introduction

SkyEye.KOM is an "information management (IM) over-overlay which is meant to gather data on a whole P2P network" [11] in order to enable capacity-based peer search and statistics on a P2P system. In this context, capacities can be CPU usage, memory usage, bandwidth utilization or disc space. The benefit of capacity-based peer search is that more suitable peers can be found for other functionality layers, a P2P application is based on. Furthermore, by analyzing the gathered statistics, developers gain a deeper insight on the performance of P2P systems and gives them the opportunity to optimize and debug their systems.

5.2 Assumptions

SkyEye.KOM can be built on top of every normal DHT in case that the DHT offers the following two basic methods:

- *void route(key, message, hint)*
- *boolean isResponsible(key)*

Like regular DHTs, the API should offer the possibility for sending and receiving messages to and from other peers. With the second method peers are able to determine whether he is responsible for a given key or not. The method is required in order to build the aggregation topology, which will be explained shortly.
Furthermore, SkyEye.KOM does not take any security issues into account, which means that all peers are considered to behave in a protocol-compliant way. In addition, it is assumed that on every peer the SkyEye.KOM software is installed.

5.3 The SkyEye.KOM Approach

SkyEye.KOM uses a k-tree topology to gather all the statistic and capacity information needed. This is done by using the responsibility ranges maintained by the DHT.
Every DHT comes with its own ID space (S_{OID}). In order to accomplish overlay independency, all IDs are mapped from the S_{OID} onto the SkyEye.KOM ID space S_{ID} using mapping functions. For

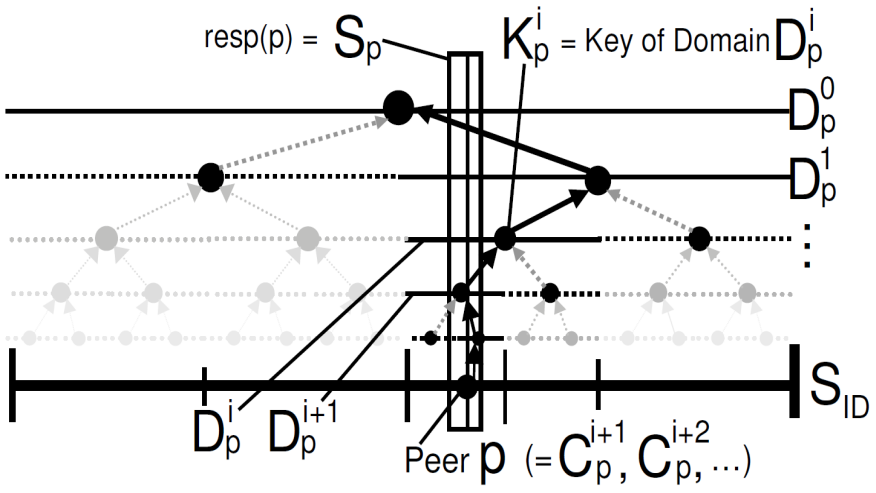


Figure 5.1: Construction of the aggregation topology within SkyEye.KOM

example, given a Chord DHT with an ID space ranging from 0 to $2^{160}-1$ the following mapping function could be used:

$$IDmap_{chord} : \left[0, 2^{160}-1\right] \rightarrow [0,1], i \rightarrow \frac{i}{2^{160}}$$

Mapping functions for other DHTs can be constructed accordingly. The denominator of the fractions should be even in order to avoid periods and problems during devision. In general, the mapping function has to fulfill the following requirements:

- preserve the convexity of identifier subsets
- preserve the responsibility ranges of nodes
- being invertible

The construction of the k-tree is done by dividing the ID space of SkyEye.KOM recursively into k smaller non-overlapping sub ranges, called *Domains*, forming the discrete levels of the k-tree as shown in Figure 5.1. To each Domain D and level l a so called *Domain Key* K is assigned. Given the responsibility function provided by the DHT API a peer p is responsible for a certain Domain at level l if and only if it is responsible for the Domain Key K_p^l. If so, peer p is considered to be the Coordinator C_p^l of the Domain D_p^l. In order to construct the tree topology, each peer p calculates its *Parent Coordinator* $C_p^{l_{min}-1}$ located at level $l_{min}-1$ by using the following formula:

$$l_{min} = min(i \in \mathbb{N} \text{ with } K_p^i \in S_p)$$

with S_p denoting the subset of the ID space peer p is responsible for, which is defined as:

$$S_p = q \in S_{ID} \text{ with } res(q) = \text{true}$$

Each peer p in the network determines for which Domains it is responsible for by calculating the *Domain Keys* K_p^l for each level l that are relevant for it. A Domain D at level l is relevant for

a peer p, if its identifier $ID_p \in S_{ID}$ is within the interval $\left[min(D_p^l), max(D_p^l)\right]$ of the Domain.
In order to compute the Domain Key K_p^l at level l a peer p first calculates the limits of the Domain D_p^l by using the following two formulas:

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

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

with k denoting the splitting factor of the SkyEye.KOM tree. Afterwards, the Domain Key K_p^l at the middle of a Domain could be calculated as follows:

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

Finally, after determining its own level and Parent Coordinator, every peer in the network starts sending updates of his statistics and capacity data upwards in the tree to his Parent Coordinator. The responsible coordinator receives all the update messages from the peers he is responsible for, aggregates the information received and sends the aggregated information to the coordinator one level up in the tree. So the information is processed bottom up in the tree at each level until it reaches the root, where the global system state about all peers in the network can be obtained.
In order to handle outdated data, statistics and capacity data is provided with a timestamp. In case that a predefined age of the data is exceeded, the data is deleted.

5.4 Querying Data

As already mentioned, SkyEye.KOM offers the possibility to search for specific peers fulfilling certain requirements. Capacity-based peer queries like "Give me 10 peers offering more than 10 MB of disk space" are processed in the same way like the update messages. A peer starting a query sends a request to the coordinator in its domain. The coordinator checks whether he has information about peers fulfilling the query. If so, it answers the query. Otherwise, it forwards the query to the coordinator one level up in the tree. By doing so, a query takes at most $log(N)$ hops in the tree to get answered, at latest by the root of the tree.
In addition, SkyEye.KOM is able to handle complex queries easily due to the use of a single aggregation tree. In case of multiple trees being used for aggregation, resolving complex queries consisting of several clauses becomes complicated as for each attribute a separate query message must be initiated within the corresponding tree containing the required information.

5.5 Optimizations

In [11] the authors proposed two approaches for solving load balancing problems and the problem of weak peers during the aggregation process. Due to the non-uniform distribution of peer

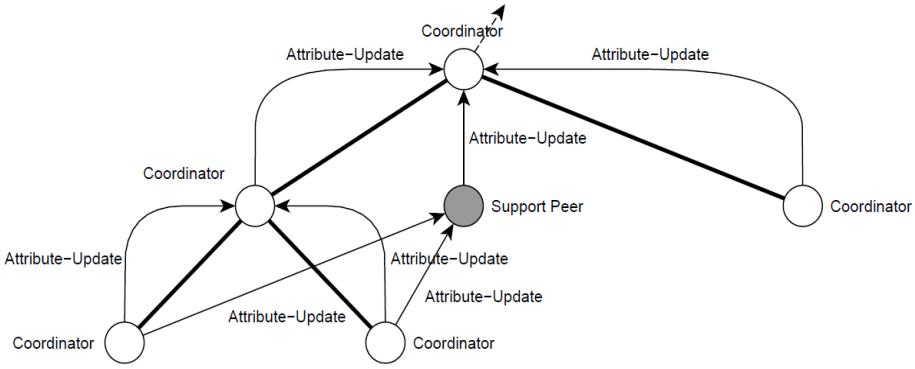


Figure 5.2: Support peer concept. Taken from: [29]

IDs, the SkyEye.KOM tree may become unbalanced. Within regions with a high ID density the tree degenerate to a list of peers. This leads to inefficiently long update times with mostly underloaded peers along the aggregation path. To overcome this problem, peers periodically check for how many child peers they know to be responsible for. In case that the number of child peers at a Coordinator C_p^l drops below a defined threshold T_{MIN}, the child peers are ordered to send their next N updates to Coordinator C_p^{l-1}. Using this mechanism the height of a tree could be decreased and underloaded peers are avoided.
Conversely, if "weak" peers are located in higher tree levels, it is likely that they become instable and aggregation data might be lost. To prevent "weak" peers from being overloaded, a support peer concept has been introduced by the authors. According to the concept each peer defines a threshold T_{max} of incoming connections the peer is able to handle. In case that the number of connections exceed the defined threshold, a support peer is chosen by the "weak" peer as shown in Figure 5.2. If the chosen support peer is willing to take over the load of the weak peer or parts of it, then the weak peer orders his child peers to send their updates for a given time period to the support peer.

5.6 Evaluation

The Evaluation of SkyEye.KOM showed that it inherits the properties of the DHT like robustness and self-organization, because it is entirely built on top of it. Node failures are handled efficiently because failing coordinators are replaced instantaneous, resulting in no further costs in the over-overlay. Many other IM systems are not load balanced in contrast to SkyEye.KOM, which is able to react on high load appearance by introducing support peers, to which update and query requests are dispatched. Furthermore the introduced approach of SkyEye.KOM is scalable, because of the used b-tree topology.

6 Architecture

In the following chapter the architecture of the developed prototype is introduced and design decisions taken during the development process are explained in order to achieve a deeper understanding on how the application works and to give a short review on the different application components.

6.1 Introduction

Today´s software must meet several requirements like modularity, reusability and low coupling. For this reason, several frameworks have been developed to ease the development of applications fulfilling the requirements mentioned above. One of these frameworks was introduced by the Open Service Gateway Initiative (OSGI) [4], which has been used for building the prototype.
OSGI offers the possibility to modularize components of a software easily and to enable or disable software components at runtime. There exist many implementations of the OSGI framework like Equinox [8] which is the most popular framework due to the fact that the Eclipse IDE has been entirely built on top of Equinox. The DIMS prototype developed during this master thesis is based on Equinox as well and consists of several components, which are shown in Figure 6.1. Every module is designed as an OSGI bundle in order to allow for an easy exchange or extension of existing functionality. Each module offers (exports) a predefined interface which can be used (imported) by other modules to ensure low coupling of functionality. The basic idea behind the architecture is to enable an easy integration into existing software projects also using OSGI, e.g. LifeSocial.KOM or PIPE. In addition, the architecture design enables an easy exchange of modules for instance the Topology Module can be exchanged with a new implementation in order to support other aggregation topologies as described in Chapter 4.7.

6.2 Module Overview

In the following, each module and its functionality are described in detail starting with the basic modules, followed by all other modules depending in this basic modules. In general, each module offers a specific functionality to other modules, which is called *Service* as explained in Chapter 4.10.

Communication Module

The Communication Module offers an interface for intermediate communication between nodes and services as an abstraction layer for communication. As shown in Figure 6.1, the communication module is the point where the integration into other applications like LifeSocial.KOM and PIPE takes place. In Chapter 6.3 possible approaches for an easy integration into these application are discussed. Modules depending on the communication module register

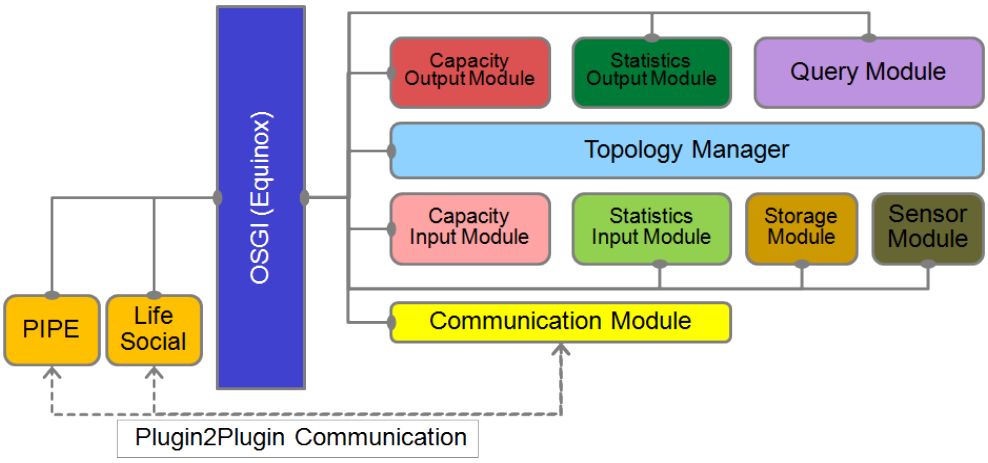


Figure 6.1: Overview of the architecture

themselves as observer at startup. For this reason the *ICommunicationService* interface provides the following methods:

- *public void addCommunicationObserver(CommunicationObserver arg)*
- *public void removeCommunicationObserver(CommunicationObserver arg)*

Whenever that a message arrives at the communication module all observers are notified using the *notifyObservers(Object o)* method. Depending on the type of the data received, each observer decides whether the content is processed or not.

Storage Module

The storage module offers an interface for storing and retrieving local statistics and capacities as well as received data from other nodes.

Sensor Module

The Sensor Module contains all sensors for measuring all statistic values as explained in Chapter 4.6. All values collected by the sensors are stored using the *IStorageService* interface provided by the storage module.

Statistic Input Module

The Statistic Input Module handles the received statistic data and stores it on the local node using the storage service. It order to be able to receive statistic data, the statistic input module registers itself as a *CommunicationObserver* at the communication module. When data has been received, the module is notified via the *update(Observable ob, Object o)* method.

Capacity Input Module

The Capacity Input Module handles the received capacity data and stores it on the local node using the storage service. It order to be able to receive capacity data, the capacity input module registers itself as a *CommunicationObserver* at the communication module. When data has been received, the module is notified via the *update(Observable ob, Object o)* method.

Topology Manager

The topology management module is responsible for maintaining the aggregation topology needed for the aggregation process. The topology manager offers an interface containing the following methods:

- *public Vector<TopologySendingInformation> getSendingInformation();*
- *public TopologyInformation getTopologyInformation();*

By invoking the *getSendingInformation()* method a node gets a vector containing several *TopologySendingInformation* objects. Each of them represents an abstract information unit including the following information:

- ID of the sender
- ID of the receiver
- transmission type (update or propagate)

The interface is designed in such a way, that the underlying topology used for the aggregation process and the aggregation process itself are uncoupled. The topology management process should be as much independent from the aggregation process as possible in order to ease the exchange or extendability of the topology management process.
In order to be able to inform child nodes about load balancing and support-peer cycles the topology module depends on the communication module and registers itself as *CommunciationObserver* at the *ICommunicationService*.

Capacity Output Module

The capacity output module is responsible for processing the node´s capacity output. In order to know which information is to be sent and where it is to be sent, the capacity output module periodically queries the topology management module. After obtaining the required sending information, the capacity output module fetches the data according to the sending information from the storage module and sends it to the receiver using the communication service provided by the communication module. The data is then received by the receivers capacity input module.

Statistic Output Module

The statistic output module generates the statistics output of a node. In order to know which information is to be sent and where it is to be sent, it periodically queries the topology management module. After obtaining the sending information needed, the statistic output module fetches the statistical data to be send from the storage module and sends it to the receiver using the service provided by the communication module. The data send is received by the receivers statistic input module.
The statistical data to be sent is the aggregate of the statistical data received from child nodes and the node´s own statistical data.

Query Module

The query module provides the functionality needed for querying different kinds of data like the current snapshot of the global system statistics or a list of peers fulfilling several specified requirements. Queries can be initiated trough the OSGI console. In order to enable custom

defined console commands the query module implements the *CommandProvider* interface as explained in Chapter 4.10.

6.3 Design Decisions

In the following some major design decisions taken during the development process are described in order obtain a deeper understanding on the architecture as well as the problems that occurred and how they were solved.

Bundle vs. Fragment Bundle

One of the most important design decision was how the integration into LifeSocial.KOM and PIPE could be achieved. The challenge was to design a communication module that easily can be used and integrated within other P2P applications. Designing only one bundle which is able to communicate within LifeSocial.KOM, PIPE and Freepastry is impossible due to the fact, that for each application specific classes must be imported, which all certainly will not be available at runtime. For example, given a module able to perform plugin2plugin communication within LifeSocial.KOM [16], there are several dependencies on specific classes provided by LifeSocial.KOM. Hence, these dependencies can not be resolved in case that the *CommunicationModule* should be integrated into another application, e.g PIPE, due to the missing LifeSocial.KOM classes. One simple way to overcome the problem of unresolvable dependencies, would be to create a new bundle for each new application that the DIMS framework should be integrated in. Yet, this approach comes with the disadvantage that every dependency linking to the Communication Module needs to be altered which depicts a great overhead. To solve this problem, modules using the communication module only import the api-package exported by the communication bundle instead of defining a dependency on the communication module. To run the prototype within different applications like for e.g. LifeSocial.KOM, the corresponding communication bundle needs to be activated (see Figure 6.2). The advantage of this approach is that dependencies are resolved at runtime. After starting the communication bundle, the communication interface is exported, which then can be used by other bundles. So it does not matter which communication bundle is used. The only important thing is that the the required communication api package is exported by the bundle. In a prior version of the communication module the fragment bundle concept of OSGI was used to enable the integration into several application. Later this approach was discarded as it turned out to be very error-prone because the Java reflection API and OSGI did not seem to not work together very well. Nevertheless, the fragment bundle concept is introduced briefly.
Fragment bundles consist of two parts: one host bundle and several fragment bundles referencing it. First a host bundle is created, providing the interface definition for communication. Each module using the communication ability is referencing the host bundle. Afterwards, a new fragment bundle for each P2P application is created, referencing the host bundle and containing the specific implementation of the communication interface. Due to the fact that the host bundle has no knowledge about the fragment bundles referencing it, the Java reflection API [32] is necessary to create instances of the classes provided by the fragment bundles. The following code snippet (see listing 6.1) illustrates how an instance of the class *CommunicationServiceImplFreepastry* is created at the communication host bundle using the reflection API of Java:

```
public void start(BundleContext context) throws Exception {

  Class<?> c = (Class<?>) Class
      .forName("CommunicationServiceImplFreepastry");

  Constructor<ICommunicationService> constructor =
      (Constructor<ICommunicationService>) c
          .getConstructor(BundleContext.class);

  ICommunicationService comServ = constructor.newInstance(context);
  context.registerService(ICommunicationService.class.getName(), comServ,
      new Hashtable());
}
```

Listing 6.1: Java Reflection API

The communication module is designed as an OSGI host bundle to enable easy integration into other P2P applications. Depending on the application the framework module should be integrated in, the corresponding fragment bundle is installed and activated. As mentioned before, the advantage of fragment bundles is, that the host bundle has no knowledge about the fragment bundles referencing it, but offers the possibility to include specific classes in the classpath at runtime. Furthermore, fragment bundles itself always need their host bundle to be able to get activated.

Java Native Interface

Obtaining low level performance data using Java is a hard task due to fact that the Java API does not offer any functions for querying any kind of low-level system data. In Java 1.6, "SUN" extended several performance beans, now offering the possibility of querying the average system load *OperatingSystemMXBean.getSystemLoadAverage()*. However, in Windows the method is not supported and always returns -1 instead of the current average CPU usage.
In order to obtain the average system load in Windows XP and Vista, the Java Native Interface (JNI) was used. JNI enables the usage of libraries written in C++, such that low level performance data can be queried easily using the Microsoft Performance Data Helper API (PDH). How JNI works and how it can be used will be explained in Chapter 6.4.

SQLite vs. Simple Storage

Another important design decision concerning the storage module was taken. When storing data, the question arises how it can be stored such that it can be queried fast and efficiently. In a prior version of the storage module, a Java library from Zentus [28] for the SQLite [27] was used. SQLite is a lightweight derivative of a SQL server supporting two different operating modes: Either, data can be stored non persistently in the memory or persistently in a data file on the hard disk drive. The first operating mode is very fast, but limited in its storage capacity. Furthermore, only one instance of the SQLite server can be started on one computer simultaneously as they all share the same memory. Hence, the memory operating mode is not suitable for a P2P application with multiple instances running on one machine.
In contrast, the second operating mode would allow multiple SQL-server instance by assigning a new file to each new instance, but it turned out that the query performance was significantly worse than using the memory operating mode.

So the decision was made to implement the storage module from the scratch without SQLite, capable of storing data in the memory as well as supporting multiple instances running on one machine. The disadvantage of this approach was that no convenient query language like SQL was present such that a complete query language had to be designed and implemented from the scratch as well.

Communication without ACK

Communication within a P2P application should be as easy as possible because in terms of churn complex communication protocols might lead to errors causing an unforeseeable system behavior. This means, that the communication protocols used for exchanging information should have as few states as possible. For this reason, the decision was made to implement a stateless communication protocol without sending ACK messages. On the one hand this decision results in a less reliable communication mechanism, but even in case of a message loss, the data be will automatically resent due to the update interval. So a reliable communication is not needed. In addition, bandwidth is saved due to only half of the messages are sent.

Activator vs. Declarative Services

As described in Chapter 4.10, OSGI offers two approaches for registering services at the service registry. The first one requires an *Activator* class being present in each bundle, which is responsible for registering services provided by a bundle at bundle startup. This approach comes with the disadvantage that all bundles are loaded at startup, possibly resulting in a high memory consumption. To solve that problem, Declarative Services have been developed, offering the possibility to load bundles dynamically at runtime only if they are really used. For example imagine the LifeSocial.KOM application, which is composed of several bundles/plug-ins like a friends-list plug-in, photo-gallery plug-in and so on. As long as the user does not request any functionality provided by the photo plug-in, there is no need to load it.
Nevertheless, the decision was made to use the *Activator* based approach for starting bundles as it is unlikely that several bundles provided by the monitoring framework remain unused.

Query Language

Another decision had to be made concerning the query language used for the capacity based peer search. If SQLite had been used for the storage module, it would have been obvious to use SQL as the query language, as query requests formulated by the user, using SQL, could have been passed directly to the storage module in order to get answered. Yet, it has been decided not to use SQLite and implement a storage module from the scratch. This means that a query language had to be implemented from the scratch as well. In a prior version of SykEye.KOM, a sum-of-product-based query language has been introduced, which has proven to be expressive enough to fulfill the requirements for capacity based peer search.

6.3.1 Used Libraries

In the following, third party libraries used by the monitoring framework are presented and described shortly.

Log4J

Logging is one of the basic functionalities that every application should provide. *System.out.println()* is quite a fast way for producing logging output, but is hardly configurable. For this reason more sophisticated solutions for logging should be considered such as *Log4J* [7] or *java.util.logging* [31], which offer the possibility to define different logging levels like *INFO*, *ERROR*, *DEBUG* and are able to redirect logging data to several outputs such as to the console or to a HTML file. In case of the developed monitoring framework, Log4J (Version: 1.2.15) has been chosen for handling the logging output, as it is one of the state-of-the-art logging libraries.

FreePastry

SkyEye.KOM is an overlay independent Distributed Information Management System which means that various Distributed Hashtable implementations can be used. As LifeSocial.KOM and PIPE are build on top of FreePastry [23], it has been decided to use FreePastry for the standalone version of Skyeye.KOM as well as in order to minimize the development overhead. At development time FreePastry was available at version 2.1 beta.

JFreeChart

JFreeChart [17], an open source chart library, has been chosen for visualizing the gathered systems statistics. JFreeChart has the advantage of being written in pure Java and not depending on some external program like for example JGnuPlot [15], which needs GNUPlot [10] being installed. In addition, JFreeChart enables real time plotting of statistical data. Furthermore JFreeChart "supports many output types, including Swing components, image files (including PNG and JPEG), and vector graphics file formats (including PDF, EPS and SVG)" [17]
Nevertheless, GNUPlot is one of the state-of-the-art open source programs for plotting and for this reason all statistic data is also exported using the dat-format, which can be read by GNUPlot easily.

6.3.2 Software Patterns

In this section, design patterns used by the framework are discussed briefly and possible alternatives are introduced.
The observer pattern has been used to uncouple the communication bundle from all bundles depending on it. Without using the observer pattern cyclic dependencies between the bundles would arise due to the fact that a bidirectional communication between the bundles is taking place, where both communication partners have to know each other. Using the observer pattern, the communication module becomes the *Observable* which only maintains a list of *Observers*. Bundles that want to use the communication bundle register themselves as an observer at the communication bundle. In case of an arriving message at the communication bundle, the *notifyObservers(Object o)* method is invoked in order to notify all observers about the arrived message. The message is delivered to each observer by calling its *update(Observable ob, Object o)* method.
An alternative approach would have been to use an intermediate storage for arriving message as known from LifeSocial.KOM. All messages arriving at the communication module are stored in an intermediate storage, which is periodically polled by all bundles that depend on the communication bundle. The disadvantage of this communication approach is that a periodic lookup at

the intermediate storage is needed, which causes an additional overhead. By using the observer pattern, periodic lookups are not necessary.

6.4 Accessing Low-Level System Information through Java

One of the core functionalities of every Distributed Information Management System is the ability to obtain information about the current system state. This includes low-level data about the current CPU performance, the memory- and network usage or a list of the available hardware. Obtaining this kind of low-level system data is hard to accomplish within Java as the Java VM offers no API for querying low level system data. Since Java 6.0 Sun has added the *information.management* package including several so called *beans* offering access to information like current heap memory usage, the operating system or the average CPU usage with some restrictions. For example using the *CPUMXBean*, the average CPU usage is only available under Linux, while querying the information under Windows returns always -1 as the functionality has not yet been implemented under Windows.
In order to gather the required information under Windows anyhow, a workaround is necessary. Currently two possible ways have been proposed to solve this problem. [25] Both approaches suffer from platform dependability as they use platform dependent programs or libraries. The first approach introduced suggests using the *Runtime.exec()* method in order to invoke a platform specific command line program like *ps*, which is able to deliver the required information and parse its output. The advantage of this approach is that it can be implemented easily and fast but at the cost of reliability as it strongly depends on the output of the external program. Updating or uninstalling the external program might lead to failures easily.
A more reliable solution is provided by the Java Native Interface (JNI) [30], which offers a standardized API for invoking platform specific functions and methods written in C/C++. On the one hand this approach is more reliable due to the implementation of platform dependent library accessing the system API, which is unlikely to change. On the other hand the second approach requires more effort as it is necessary to recompile the library on every platform. Furthermore, the usage of C++ DLLs might lead to *OutOfMemoryErrors* due to possible memory leaks caused by insufficient memory management or grubby source code. Developers who want to write a DLL in order to access system functions, have to take the following steps:

- create a Java class introducing the native API. This is done by defining function bodies using the *native* keyword like e.x. *public native double getAverageCPUUsage()*
- the created java class must include the *System.loadLibrary("name-of-dll")* function call
- create a class file (*.class) using the javac compiler e.g. javac path.to.java.file.java
- create a C++ header file (*.h) using the javah e.g javah path.to.java.file
- create a Visual Studio DLL Project and import the created header file
- create a cpp file (*.cpp) implementing the created header file
- build the DLL

- copy the created DLL-file to the Java project directory, so that the DLL can be found at runtime of the Java project

When a native function e.g. *public native double getAverageCPUUsage()* is called from the Java application, the corresponding C++ function is called provided by the DLL which has been specified by the *System.loadLibrary()* command.

6.5 Querying Performance Data on Microsoft Windows Vista And XP

The question that now arises is how to query performance data on Microsoft Windows using C++. There exists a convenient way for Windows developers to obtain information about performance counters. The Performance Data Helper (PDH) [18] DLL published by Microsoft offers a query interface for accessing a variety of performance measures called counters. "Counters are used to provide information as to how well the operating system or an application, service, or driver is performing. The counter data can help determine system bottlenecks and fine-tune system and application performance. The operating system, network, and devices provide counter data that an application can consume to provide users with a graphical view of how well the system is performing." [18] Consumers of counter data can decide whether they want to consume performance data from a real time source or from log files. In our scenario we use the real time performance data sources. In order to gather data through the PDH interface the following steps are necessary:

1. a query handle must be created using the *PdhOpenQuery()* function
2. performance counters must be added to the query using *PdhAddCounter()*
3. afterwards the query can be executed using *PdhCollectQueryData()*
4. in order to print a formated version of the queried data, the *PdhGetFormattedCounterValue()* function can be used

Performance counters are organized in a hierarchical fashion and can be specified using the following URL-like syntax:

```
\<NameofComputer>\<ClassOfCounter>\<Resource>\<CounterValue>
```

The first part of the query specifies on which computer the query data should be gathered. The next part determines which class of counter data should be queried like for instance CPU usage or network traffic. The *Resource* part specifies whether the data should be collected from a specific resource on the computer. For example if there are two network cards installed on one computer, you can decide whether to measure the data from only one of the interfaces or the total network usage. If the total network usage should be queried the Resource part is left empty. The last part of the query appoints the detailed counter value. For example the current CPU usage can be queried be using the following query string:

```
\LocalComputer\Processor (*)\ProcessorTime %
```

PDH Query strings must always be written in the default operating system language. Otherwise, the DPH API does not deliver the requested performance data.

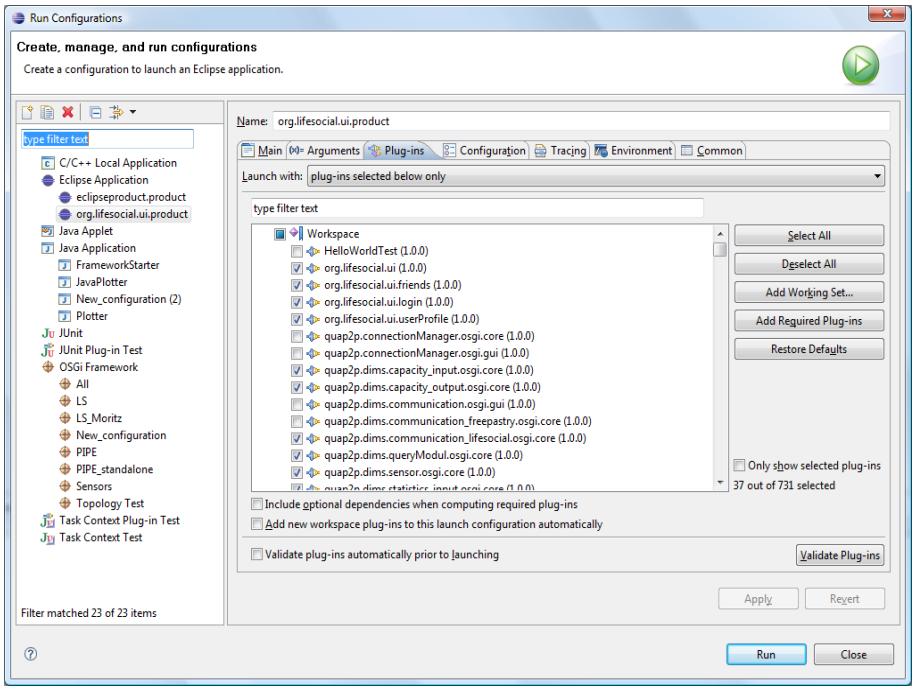


Figure 6.2: Running the prototype with the LifeSocial.KOM communication bundle

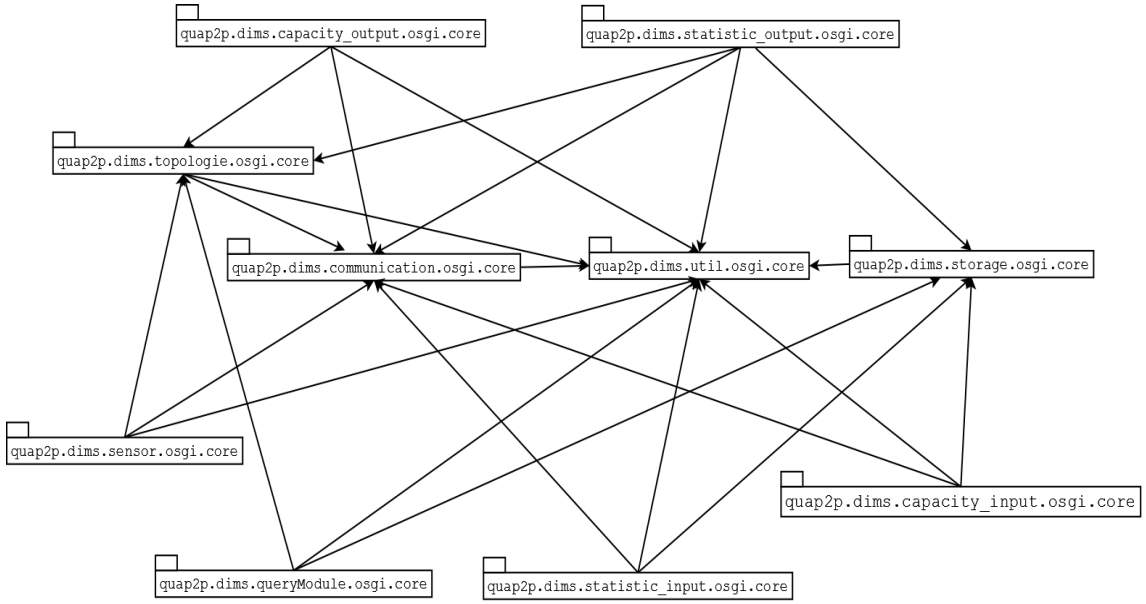


Figure 7.1: Bundle dependencies

7 Implementation

This chapter deals with the implementation details of the developed distributed monitoring framework. It addresses developers, who are interested in integrating the framework into new applications or extending existing functionality of the framework. For this reason the integration of the framework into LifeSocial.KOM and PIPE is described in detail. In addition, the implementation of the sensor bundle is introduced as well as the basic data structures and the supported export formats. Furthermore, the bundle interaction for the most important use cases is explained.

7.1 Bundle Overview

As described in Chapter 4.10, most bundles provide a specific service that is registered at the service registry at bundle startup. Every bundle of the framework contains at least two packages: an **.api* package and a **.component* package, e.g. *quap2p.dims. util.osgi. core.logging.api* and *quap2p.dims.util.osgi.core. logging.component*, which contains the logging service interface and its implementation. The **.api* package always contains the service interface which is exported via the *Manifest.MF* file. The service implementation is located in the **.component* package and should not be visible to other bundles.
The framework is structured as follows: For each module introduced in Chapter 6.2 there exists a separate Eclipse plugin project representing an OSGI bundle. Below, each bundle and its implementation-details are explained.

quap2p.dims.util.osgi.core

The basic data structures as well as the basic functionality like logging or loading of configuration files are located in the *quap2p.dims.util.osgi.core* package, which is used by every other bundle in the framework.

quap2p.dims.communication.osgi.core

For each application using the monitoring framework (LifeSocial.KOM, PIPE, FreePastry) a separate communication bundle has been implemented:

- *quap2p.dims.communication_lifesocial.osgi.core*
- *quap2p.dims.communication_pipe.osgi.core*
- *quap2p.dims.communication_freepastry.osgi.core*

All these bundles have in common that they contain and export the *ICommunicationService* interface. At startup of the framework, only one of the bundles should be activated. Otherwise the framework will not start properly due to multiple registrations of the *ICommunicationService* at the service registry. How each of the bundles works in detail is described in Chapter 7.8 and 7.9. Figure 7.1 illustrates that the *quap2p.dims.util.osgi.core* and the *quap2p.dims.communication_(x). osgi.core* bundles are the most basic bundles because many other packages depend on them.

quap2p.dims.communication.osgi.gui

The *quap2p.dims.communication.osgi.gui* bundle contains an minimalistic GUI for starting a new FreePastry ring or connecting to a bootstrap node and was meant to work together with the *quap2p.dims. communication_freepastry.osgi.core* bundle. For PIPE and LifeSocial.KOM it was not necessary to use the GUI, as they already have a GUI of their own for managing the connectivity.

quap2p.dims.storage.osgi.core

The *quap2p.dims.storage.osgi.core* bundle is responsible for storing the nodes´ local data as well as data received from other nodes. For this reason the storage bundle provides the *IStorageService* interface which is implemented by *StorageServiceSimpleImpl*. As seen in Figure 7.1 the interface is used by the capacity and statistics in and output bundles in order to store and retrieve data.
In order to avoid sending outdated data of nodes that left the network, the storage service has a cleanup thread, which periodically checks all stored data. If no update for a specific data item is received within a certain period of time, the data is deleted. Stored data may become obsolete due to leaving or joining nodes and the related alternation of the SkyEye.KOM tree topology. The time period for deleting data has to be chosen carefully. A too aggressive cleanup policy would cause a loss of all stored data in the worst case, whereas a too conservative cleanup policy might include outdated data in the global system statistics. To avoid data loss, the update interval of the *ICapacityOutputservice* and the *IStastisticOutputservice* must always be smaller than the time period after which data is considered to be outdated.

quap2p.dims.sensor.osgi.core

The *quap2p.dims.topology.osgi.core* bundle contains all sensor implementations like CPU and network usage sensor. Every sensor implementation is a subclass of the abstract class *AbstractSensorService* which implements the *ISensorService* interface and extends *java.util.Thread* as shown in Figure 7.2. Every sensor is running in a separate thread enabling a different update intervals for each sensor. In addition, the multi-threaded sensor concept has the advantage that, if a sensor crashes due to an error, the other sensors will remain operational. On the other hand this approach causes additional overhead at the storage service as the service must support concurrent write accesses.

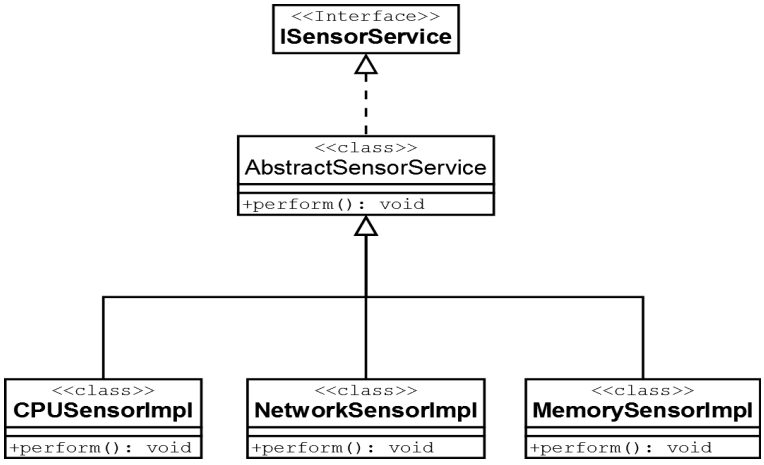


Figure 7.2: UML class diagram of the sensor bundle implementation

quap2p.dims.topology.osgi.core

The *quap2p.dims.topology.osgi.core* bundle provides the *ITopologyService* and is responsible for maintaining the SkyEye.KOM over-overlay. For the purpose of load balancing the topology bundle uses the communication service to send and receive load balancing instructions.

quap2p.dims.capacity_input.osgi.core

All received messages containing capacity data are handled by the *ICapacityInputService*. The service checks whether the data is for the purpose of update and stores it on the local node. In order to receive incoming capacity data, the capacity input service registers itself as an observer at the communication service.

quap2p.dims.statistics_input.osgi.core

All received messages containing statistical data are handled by the *IStatisticsInputService*. In contrast to the capacity input service, it receives update and propagate data as global system statistics are updated upwards and propagated downwards in the tree. The service checks weather the received message contains update or propagate data and stores it accordingly. In order to receive incoming statistical data, the statistic input service registers itself as an observer at the communication service.

quap2p.dims.capacity_output.osgi.core

The capacity output bundle contains the functionality for sending capacity output data upwards in the tree. For this reason it periodically queries the *ITopologyService* for the receiver´s id (parent coordinator id). Afterwards, it fetches all received capacity data from the *IStorageService* and sends it to the receiver.

quap2p.dims.statistics_output.osgi.core

The *IStatisticOutputService* provided by the *quap2p.dims.statistics_output.osgi.core* bundle works much like the *ICapacityOutputService*. It also periodically queries the topology service, fetches the statistical data from the local storage and sends it to the parent coordinator. In case that a node is the root of the SkyEye.KOM tree, it starts propagating the global system state down the tree.

quap2p.dims.queryManager.osgi.core

The *quap2p.dims.queryManager.osgi.core* bundle contains the functionality for accessing the global system statistics or searching for peers using the capacity based peer search. The query service extends the OSGI console such that queries can be initiated through the console. Tn order to accomplish this, the query service implements the *CommandProvider* interface as introduced in Chapter 4.10.3.

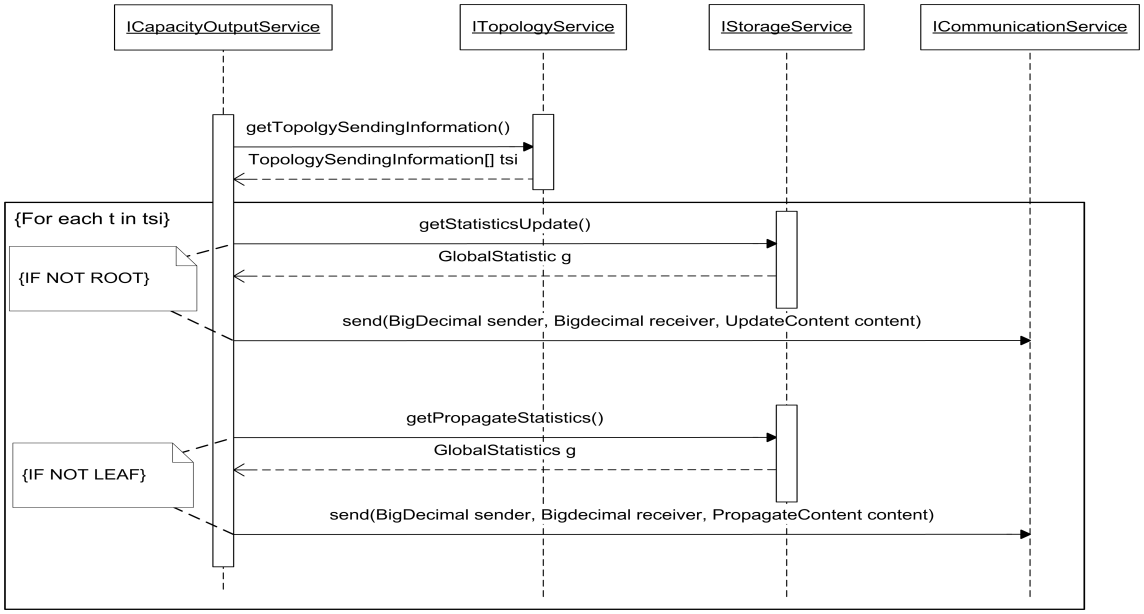


Figure 7.3: Bundle interaction in case of sending statistical data

7.2 Bundle Interaction

In the following the interaction between the different bundles for the most important use cases is explained in order to allow for a deeper understanding of the processes within the framework.

7.2.1 Updating / Propagating Statistical Data

In order to send statistic update or propagate data, the *IStatisticOutputService* first queries the *ITopologyService* to get an array of *TopologySendingInformation* objects, which contains the information about the receiver and the type of data (update or propagate) to be included in the message. Depending on the type, either the *getUpdateStatistics()* or *getPropgateStatistics()* method of the *IStorageService* is invoked, returning the data to be sent. Afterwards, the *send(Bigdecimal sender, Bigdecimal receiver, Serializable s)* method of the *ICommunicationService* is invoked. The sequence diagram as shown in Figure 7.3 visualizes the update / propagate process.

7.2.2 Receiving Statistic Updates

Figure 7.4 shows the interaction between the different bundles in case of an incoming update for the global system statistic. The message is received by the *ICommunicationService* via the *receive()* method. As explained in Chapter 6.3.2 the ICommunicationService extends Java´s *java.util.Observable*. For this reason all registered observers, e.g. *IStatisticsInputService*, are notified about the arrived message by calling the *notifyObservers(Object o)* method, which causes the invocation of the observer´s *update(Observable ob, Object o)* method. After receiving the update data the *IStatisticsInputService* checks whether the data is an instance of *GlobalStatistics*. If so, it calls the *storeGlobalStatistic()* method of the *IStorageService* in order to store the received update for the global system statistics. Propagate data is received the same way. The only difference between receiving update and propagated data is, that the *storeProgagateStatistic()* method is invoked instead.

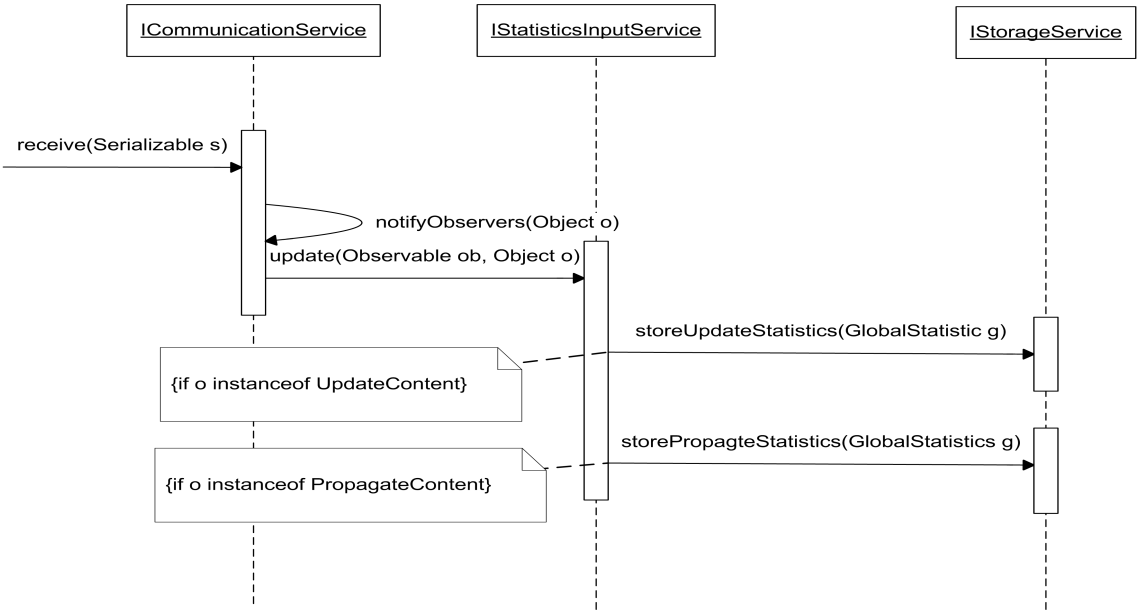


Figure 7.4: Bundle interaction in case of receiving statistical data

7.3 Data Structures and Content Types

In the following, the basic data structures as well as the content types are introduced, which are located in the *quap2p.dims.util.osgi.core* bundle. Figure 7.5 shows the data structures used by the statistic update/propagate process. The data structures for the capacity based peer search are shown in Figure 7.6. The global system state of a P2P network or parts of it are represented by the *GlobalStatistics* object, which contains a collection of *SingleStatistics* objects each representing a single metric. The *SingleStatistics* class offers the possibility of storing different kinds of aggregates for a metric like minimum, maximum, sum, and so on. In contrast, capacity data is organized as follows: The *PeerList* object stores a collection of *Peer* objects. Peers can be added or removed by using the *addPeer(Peer p)* or *removePeer(Peer p)* method. Every *Peer* instance contains zero or more *Attribute* objects. Due to the fact that an attribute value may be expressed as a nominal and/or numeric value, the *Attribute* class offers three different constructors. In case that the attribute value can be expressed meaningfully as numeric and nominal data, e.g. version numbers or number of processors, the first constructor should be used as shown in Figure 7.6. In the other case, that the attribute value can only be expressed as nominal data like the name of the operating system, the corresponding constructor should be invoked. Furthermore, to every attribute or statistic value measured by some sensor, a timestamp is attached in order to determine the age of the data.

7.4 Data Collection

As described in Chapter 7.1 the *quap2p.dims.sensor.osgi.core* bundle contains all the functionality for measuring local statistic and capacity data. The framework was designed in such a way that new sensors can be added easily. How to add a new sensor is explained in Chapter 7.7.
Sensors can be divided into internal and external sensors. External sensors do not depend on the other bundles except the storage bundle and gather data from outside the application like the CPU usage. Internal sensors gather data provided by other bundles like the topology or communication bundle.
Most sensors periodically query a data source and store the query result using the *IStorageService*. Depending on the type of sensor data being collected, the measuring process can either be done easily

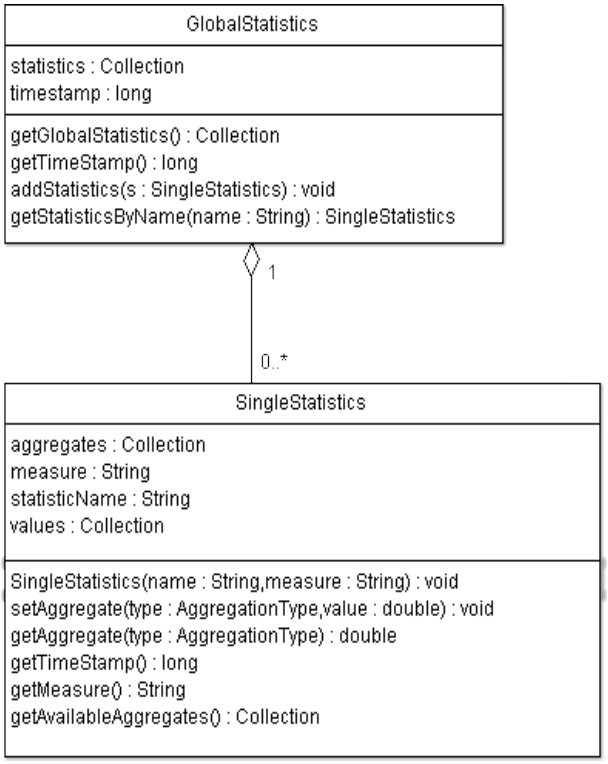


Figure 7.5: Class diagram of the basic data structures

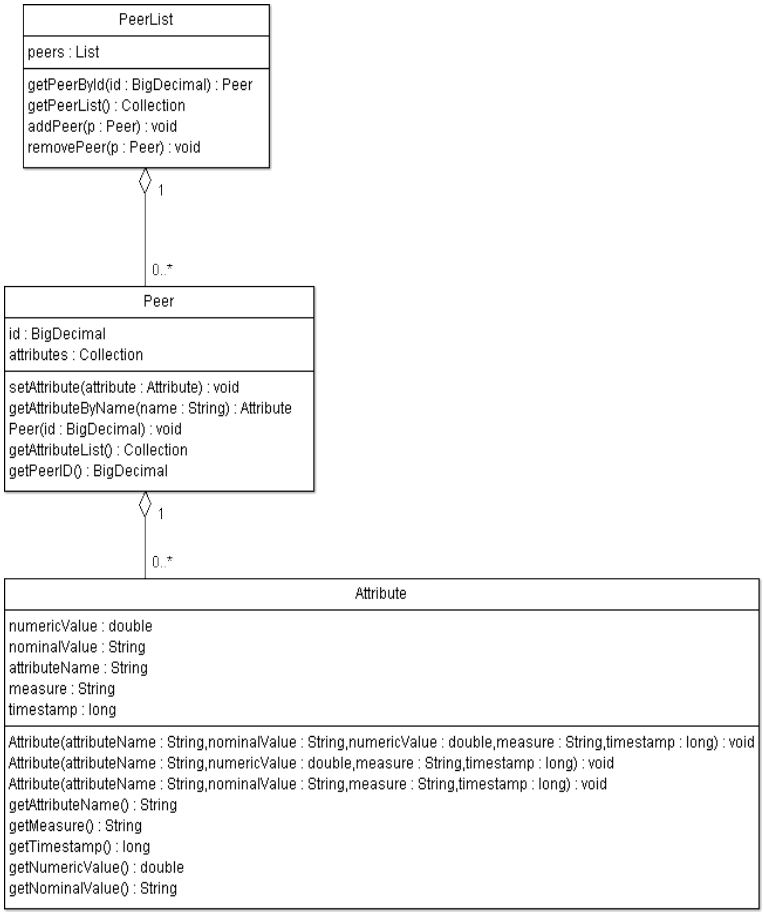


Figure 7.6: Class diagram of the basic data structures

or not. Low level system data e.g. is very difficult to obtain as the Java API hardly offers support for gathering low level system data. As explained in Chapter 6.3 JNI is used to overcome this problem at the expense of interoperability, because for each operating system used, a separate platform specific version of the DLL library must be implemented. In order to access the methods provided by the DLL, the *NativeSystemConnector* class has been implemented, which is used by the network and CPU sensor. Listing 7.1 shows the source code of the *NativeSystemConnector* class. As one can see, depending on the operating system used the required DLL file is loaded via the *System.loadLibrary(name)* command. Furthermore, the native methods for accessing the DLL are shown in the listing.

```
1  public class NativeSystemConnector {

3    static {
       // loading a native lib in a static initializer ensures that it is
5      // available done before any method in this class is called:
       try {
7        if (System.getProperty("os.name").equals("Windows Vista")) {
           System.loadLibrary("DLL_VISTA");
9        } else if (System.getProperty("os.name").equals("Windows XP")) {
           System.loadLibrary("DLL_XP");
11       } else {
           System.out.println("Unsupport system");
13       }
       } catch (UnsatisfiedLinkError e) {
15       System.out.println("native lib not found in 'java.library.path': "
             + System.getProperty("java.library.path"));
17       throw e; // re-throw
       }
19   }
     public static synchronized native double getProcessCPUTime();
21
     public static synchronized native double getCurrentTotalStream();
23
     public static synchronized native double getCurrentDownStream();
25
     public static synchronized native double getCurrentUpStream();
27 }
```

Listing 7.1: NativeSystemConnector

In the following sections the most important sensor implementations and problem solved during the development process are explained in detail.

Handling Fluctuating Data

During the implementation of the sensors some obstacles occurred. Several sensors like the network sensor, delivered very fluctuating and unsteady data, which led to unusable statistical results. To overcome this problem, the following approach was chosen: the last ten measured values are stored, the lowest and the highest value are neglected and the average value is calculated using the remaining eight data points. In practice this approach has proven to delivers good results even if the input data is very volatile.

Network Utilization - Bandwidth Estimation

Another issue that needed to be solved was the problem of bandwidth estimation because querying the bandwidth of the network device would always deliver 100 or 1000 Mbit. This might be true in case that the application is running only in a LAN but the bandwidth information would be wrong in case

that the application is communicating over the Internet as a standard Internet connection in Germany usually offers not more than 6 Mbit bandwidth downloadstream and 1 Mbit uploadstream. So it was necessary to estimate the available bandwidth that can be used for communication. An easy approach was chosen to solve this problem: the greatest ever seen value for the current network and download usage is stored. The average percentage of network usage is then calculated using the following formula:

$$\left(\frac{current_value}{max_value}\right) * 100$$

In order to query the current network utilization the *getCurrentTotalStream()*, *getCurrentDownStream()*, and *getCurrentUpStream()* methods are used. Currently the network sensor does not support Linux as no Linux specific DLL has been implemented yet.

CPU Sensor

As already mentioned above, the CPU sensor is using the *NativeSystemConnector* class. The current CPU usage is queried by invoking the *getProcessCPUTime()* method and averaged over the last ten measured values. Under Linux the CPU usage can be queried by invoking the *getSystemLoadAverage()* method provided by the *OperatingSystemMXBean*.

Communication Sensor

The communication sensor periodically invokes the *getCommunicationStatistics* method offered by the *ICommunciationService* interface, which returns a hash map containing values for several communication metrics like the average hop count, average size per message, amount of sent and received data, amount of sent and received zipped data, number of sent and received messages, and so on. Furthermore, the communication sensor queries the *ITopologyService* to get the information about the current level of the node within the SkyEye.KOM tree topology. This information is combined with the information obtained from the communication service in order to get level-specific statistics on the bandwidth utilization, messages sent, average message size and so on.
To be able to measure the average hop count of messages being sent and received, the underlying communication bundle must support hop counting. In order to support hop counting every message must maintain a hop counter and the communication bundle must be able to increase the hop counter in case that a message is forwarded. For example the FreePastry *Application* interface contains the *forward(RoutingMessage m)* method, which can be implemented using the following code snipped in order to support hop counting:

```
1  @Override
   public boolean forward(RouteMessage arg0) {
3    try {
       if (arg0.getMessage(endpoint.getDeserializer()) instanceof FreePastryMessage
           ) {
5        FreePastryMessage m = (FreePastryMessage) arg0.getMessage(endpoint.
             getDeserializer());
         m.increaseHopCount();
7      }
     } catch (IOException e) {
9      e.printStackTrace();
     }
11   return true;
   }
```

Listing 7.2: Implementation of the forward() method

Currently hop counting support is available only for the FreePastry standalone communication bundle and the PIPE communication bundle.

Memory Sensor

The memory sensor gathers information about the currently available and used Heap Memory of the Java Virtual Machine. This information can be accessed easily via the *MemoryMXBean*, which is part of the Java API.

Storage Sensor

The information about the total free disc space can be obtained using the following code snipped:

```
long freeDiskSpace = 0;
File[] roots = File.listRoots();
for (File file : roots) {
  freeDiskSpace += file.getFreeSpace();
}
//calculate megabytes
freeDiskSpace /= (1024 * 1024);
//store information
this.getStorageService().storeStatisticAttribute("FREE_DISK_SPACE",freeDiskSpace
    , "MB");
this.getStorageService().storePeerAttribute("FREE_DISK_SPACE",String.valueOf(
   freeDiskSpace).toString() , "MB");
```

Listing 7.3: Obtaining available disc space

The information about the free disc space is part of the peer attributes as well as part of the global system statistics as the information can be used meaningfully in both cases.

7.5 Data Storage And Aggregation

In the following, the data aggregation process of local and received capacity and statistical data is explained in order to make the fundamental difference between these two aggregation approaches understandable.
The data aggregation ability is one of the core functionalities of the monitoring framework. Data aggregation is done using the following two approaches: capacity data is aggregated using a list-based data aggregation wheres as statistical data is aggregated by computing a singe value per metric. On every peer, several sensors are running which periodically store their results using the *IStorageService*. For this reason the *IStorageService* offers two methods for storing local peer attributes (capacity data) or local statistical data:

- *storeStatisticAttribute(String identifier, double value, String measure)*
- *storePeerAttribute(String identifier, String value, String measure)*

For each statistical data item stored by using the storeStatisticAttribute() method a new *SingleStatistics* object is created where as for each new peer attribute being stored a new *Attribute* item is created. This local data will be merged with the data received from other nodes later during the aggregation process.

Aggregation of Capacity Data

The capacity data aggregation is done by the *ICapacityOutputService*. As explained in Chapter 7.3 capacity data is represented by *PeerList* objects, which contain a collection of *Peer* objects. Each *Peer* object in turn has a collection of *Attribute* objects. In a first step all received *Peer* objects are fetched from the *IStorageService*. Then a new *PeerList* object is created and all *Peer* objects are added to the *PeerList*. Afterwards, the service queries the local node´s own attribute list from the *IStorageService*, creates a new *Peer* object including its own attribute list and appends it to the newly created *PeerList* object. In case

Command	Description
getGlobalStatistics	Retrieves the current snapshot of the global system statistics
getGlobalPeerList	Retrieves a list of all known peers in the network
getAllCapacityNames	Retrieves a list of all known capacity name
executeQuery "QueryString"	Executes a capacity query. The query string must satisfy the introduced query language
getResult [id]	Retrieves the result for a given query id, if no id is specified, the last received query answer is returned

Table 7.1: Query commands

that the node is not the root of the tree the created *PeerList* object is sent to the nodes parent coordinator one level up in the tree. Using this approach, *PeerList* objects are forwarded upwards in the tree until they reach the root of the SkyEye.KOM tree, where the global capacity list, containing information about every peer in the network, can be obtained.
The list-based aggregation has the disadvantage of increasing messages sizes towards the root. Depending on the number of nodes the aggregate could reach several hundreds of MB in size. To solve this problem it is necessary to establish different filter mechanisms to reduce the amount of data forwarded upwards in the tree. The challenge is to select the best peers according to some score, without loosing capable peers, because rejected peers might not be found later during the capacity based peer search.

Aggregation of Statistical Data

In contrast to the capacity data aggregation, statistical data is aggregated by computing a single value per metric such as sum, count, min, max and so on and is done by the *IStatisticsOutputService*. As already mentioned in Chapter 7.3, the global system state is represented by the *GlobalStatistics* class, which contains a collection of *SingleStatistic* objects. Starting from the leaves of the SkyEye.KOM tree, nodes create a *GlobalStatistic* object containing their local system state and send it to their parent coordinator. The parent coordinator merges all received *GlobalStatistics* objects including his local system state using the *mergeGlobalStatistics(GlobalStatistics a, GlobalStatistics b)* method provided by the *IStorageService*. For each *SingleStatistics* included in the *GlobalStatistics*, a single aggregate is computed as explained in Chapter 4.5. The advantage of single value aggregation is that the message size remains almost constant in the whole tree. There might just be slight increases in case that a new metric is introduced by a node.

7.6 Querying Data

All data gathered by the monitoring framework can be accessed through a query interface provided by the *IQueryService*. The *IQueryService* uses the possibility of custom defined console commands as described in Chapter 4.10. Currently, the commands listed in table 7.1 are supported. Resolving a query in a distributed environment like a P2P network might take some time due to the fact that the query might used to be forwarded several times until it can be resolved. To prevent the console from blocking during a query initiated by a user gets answered, the query process has been divided into two parts. After invoking executeQuery the console prompt returns immediately. Meanwhile, the query is resolved by SkyEye.KOM and the result is sent back to the requesting node. When a node receives a result for a query request, an information messages is displayed in the console. Afterwards, the user is able to display the query result be executing the *getResult* command. In addition to the console commands, the IQueryService offers a query interface that can be used by other bundles.

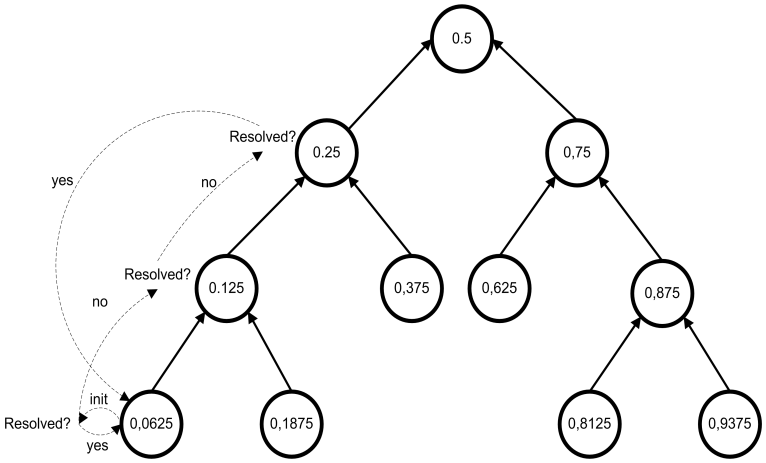


Figure 7.7: Query resolving

Query Resolving

Figure 7.7 illustrates how queries are resolved by the framework. If a query is initiated on a peer, the *IQueryService* first checks whether the query can be resolved locally. If so, the query is answered immediately. If not, the query is forwarded to the parent coordinator one level up in the tree. The coordinator also checks whether the query can be resolved or not. If so, he sends an answer to the query originator. In case that he is not able to answer the query, it is further forwarded up the tree. Following this scheme, a query travels upwards in the tree until it either is resolved by a coordinator or it reaches the root of the SkyEye.KOM tree. If neither one coordinator nor the root are able to answer the query, it remains unresolved and a corresponding message is sent back to the query originator. One question concerning the resolving of queries arises in case that a peer is able to answer a query only partially. If e.g. a user has requested 30 peers fulfilling certain requirements and a peer on level n has found 10 peers matching the query, the question arises whether the 10 peers should be included in the query request which will be forwarded to the coordinator on level $n-1$. If so, it is likely that the query can be answered faster, but the peers that are found may not be the best peers fulfilling the requirements, especially when filter mechanisms are used to reduce the size of the capacity update messages. Furthermore, including the peers in the query request will result in an increasing size of the query messages while traveling up in the tree causing additional bandwidth utilization/overhead.
During the implementation of the current version of the framework the decision was made, not to include the peers, because the query answer with and without including the already found peers will be the same as no filter mechanisms are implemented yet.

Query Language

Besides the ability of gathering system statistics the framework offers the possibility of searching for peers fulfilling specific requirements. Queries are formulated using a sum of products (ger: disjunktive Normalform) based query language which will be explained in the following using the Backus-Nauer-Form.

```
<QueryString> ::= "<NumberOfSearchResults>;<Condition>"

<NumberOfSearchResults> ::= * | <DigitSequence>

<DigitSequence> ::= <Cipher>| <Cipher><DigitSequence>

<Cipher> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Condition> ::= <AND-Clause> | <Condition>||<Condition>
```

```
10
   <AND-Clause> ::= <Clause> | <Clause>&&<AND-Clause>
12
   <Clause> ::= <CapacityName><Comparator><Value>
14
   <CapacityName> ::= OPERATION_SYSTEM | ... |  DISC_SPACE
16
   <Comparator> ::= < | > | <= | >= | == | !=
18
   <Value> ::= <CharOrCipher> | <CharOrCipher><Value>
20
   <CharOrCipher> ::= A | ... | Z | a | ... | z | 0 | ... | 9 | " "
```

Listing 7.4: Query Language

Given the query language as shown in Listing 7.4, a valid examples for a query string is shown in Listing 7.5.

```
1  "3;OPERATING_SYSTEM==Win  XP&&DISC_SPACE>500||NETWORK_USAGE<5"}.
```

Listing 7.5: Query Language Example

The answer to this example query would contain three peers fulfilling the following requirements: either they are running under Windows XP and offer more than 500 MB of free disc space or their network usage is below 5 KBs.

7.7 Extension Points

The monitoring framework was designed with extendability in mind. Adding new functionality should be done easily and should affect as few bundles as possible. In the following, possible scenarios for adding functionality to the framework are introduced in order to provide an overview of additional useful functionality for future development. In addition, this section should enable developers to get a closer look on how functionality can be added to the framework.

Adding New Sensors

One easy scenario for adding new functionality to the framework deals with the implementation of new sensors. As already mentioned in Chapter 7.1 all current sensor implementations are subclasses of *AbstractSensorService* which implements the *ISensorService* interface and extends the class *java.lang.Thead*. Being a subclass of *java.lang.Thead*, each sensor can run as a separate thread. Listing 7.6 shows the implementation of the *run()* method of the *AbstractSensorService* class. The class offers two kinds of operating modes. In case that a metric on a peer will not change over time, e.g. the operating system version, it is not necessary to query the information periodically. To cope with this issue the developer just has to define a negative update interval indicating that the sensor should query the data source only once. Given the operating system example an updateInterval of -1 will query the information about the installed operating system only ones.
In case of metrics that change over time, a positive update interval should be specified, causing the sensor to query the information periodically every $updateInterval$ milliseconds.

```
1  public void run() {
       if (updateInterval < 0) {
3            perform();
       }else {
5        while (!this.isStopped()) {
```

```
        try {
7         Thread.sleep(getUpdateInterval());
          perform();
9       } catch (InterruptedException e) {
          e.printStackTrace();
11      }
      }
13  }
}
```

Listing 7.6: run() method of the AbstractSensorService class

To add a new sensor implementation, it is necessary to create a new class which extends *AbstractSensorService*. Furthermore, the implementation must implement the abstract method *perform()*, which is periodically invoked as shown in Listing 7.6. All measured values must be stored using the two corresponding store methods of the *IStorageService* as demonstrated in Listing 7.3.

Supporting new Topologies

SkyEye.KOM uses a tree-based topology for gathering and aggregating data. Although tree topologies seem to be the natural expression for data aggregation, it is possible that aggregation of data could be also be accomplished by using other topologies. For this reason, the current architecture of the monitoring framework supports the integration of new aggregation topologies. The *quap2p.dims.topology.osgi.core* bundle contains all the functionality required for the topology maintenance process. In case that a new topology management service has to be created, it is only necessary to create a new implementation of the *ITopologyService* interface, which contains the *getTopologySendingInformation()* method. The method returns an array of abstract sending instructions containing information about sender, receiver and type of data to be sent.

Including Filter Mechanisms

The current implementation of the monitoring framework does not support filter mechanisms for reducing the amount of capacity data produced during the aggregation process. Without appropriate filter mechanisms it is likely that bottlenecks towards the root of the SkyEye.KOM tree will occur due to the increasing size of update messages. In order to overcome this problem, it is necessary to forward only the most relevant and interesting parts of the capacity data. This means that only a subset of the received information about the peers and their capacities are forwarded towards the root. To determine which peers are relevant, scoring functions could be used. Based on the current capacity values, a score could be computed for each peer and only the top n peers could be forwarded. This functionality could be added to the current implementation of *ICapacityOutputservice*.

Supporting new Output Formats

Data analysis is the next logical step in the optimization cycle after obtaining data about the global system state. This could be done using data mining tools like WEKA [19]. In order to ease the usage of external programs, the monitoring framework should support different output formats, e.g. writing *.arff files which can easily be interpreted by WEKA. Currently only comma separated values (CSV) and the dat file format used by GNUPlot are supported. To add new output formats it is necessary to implement a class of type *IStatisticWriterService* or to extend the existing functionality located in the *StatisticWriterServiceImpl*.

Integration Into Other Applications

The integration into existing applications is described in detail in Chapter 7.8 and 7.9.

7.8 Integration into LifeSocial.KOM

One of the basic requirements for the monitoring framework was the ability to integrate easily into existing applications like LifeSocial.KOM or PIPE. As already explained in Chapter 6.2, the communication bundle is the point where the framework is connected to other applications. For each new application, the framework should be integrated in, it is necessary to create a new bundle offering a package named *quap2p.dims.communication.osgi.core.api* which in turn must contain the *ICommunicationService* interface. In addition, each implementation of the *ICommunicationService* uses a private *Converter* class to translate between the ID space used by the underlying DHT and the SkyEye.KOM ID space.
In order to enable communication within LifeSocial.KOM, the *quap2p.dims. communication_lifesocial. osgi.core* bundle was created fulfilling the requirements mentioned above. At development time the implementation of LifeSocial.KOM did not offer any mechanism for querying whether the current instance of LifeSocial.KOM is connected to the network or not. Furthermore, it offered no mechanism for notifying other applications about a change of connectivity. So some workarounds were necessary to overcome this problem. The current implementation of *CommunicationServiceImplLifeSocial* uses a separate thread called *ConnectionChecker*, which periodically checks whether the local instance of LifeSocial.KOM is connected to the network or not. As already mentioned, the LifeSocial.KOM implementation did not offer any method for querying the current state of connectivity. So it was necessary to conclude this information by checking if the *MessageDispatcher* used by LifeSocial is null as shown in Listing 7.7.

```
public void run() {

      while (run) {
        try {
          if (Connector.getMessageDispatcher() != null) {
            System.out.println("Connection established");
            initMessageApp();
            run = false;
          }
          Thread.sleep(updateInterval);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
```

Listing 7.7: run() method of the ConnectionChecker class

Listing 7.8 shows the initialization of the message dispatcher right after the local node gets connected to the network.

```
synchronized (node) {
      while (!node.isReady() && !node.joinFailed()) {
        try {
          // delay so we don't busy-wait
          node.wait(500);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        // abort if can't join
        if (node.joinFailed()) {
          this.setUpSuccess = false;
          logger
              .error("Join to the LifeSocial network failed... Please try again
                 later.");
```

```
        }
      }
    }
    if (isSetUpSuccess()) {
      storageDispatcher = ObjectFactory.getStorageDispatcher(node,
          storageDirectory);
      messageDispatcher = ObjectFactory.getMessageDispatcher(node);
      emailSender = ObjectFactory.getEmailSender();
      localStorage = ObjectFactory.getLoacalStore();
      commands = ObjectFactory.getCommands();
      searchEngine = new SearchEngine(storageDispatcher);
      hashGenerator = new HashGenerator();
      pluginMonitor = new PluginMonitor();
    } else {
      Logger().error(
          "LifeSocial couldn't be started...\n"
              + "Please try it again.");
      shutdown();
    }
}
```

Listing 7.8: Code snippet of the constructor of the LifeSocial *system.Connector* class

After a successful connect the *ConnectionChecker* recognizes that the local node is connected to the network and tries to setup the *CommunicationServiceImplLifeSocial* by invoking the *initMessageApp() method* as shown in Listing 7.9. Afterwards the *ConnectionChecker* thread terminates.

```
private void initMessageApp() {
    messageApp = Connector.getMessageDispatcher().getMessageApplication();
    this.endpoint = messageApp.getEndpoint();
    this.setChanged();
    this.notifyCommunicationObservers(converter.convertToSkyEyeId(Connector
        .getNode().getId()));
    logService.logMessage("Endpoint available!", LOGGING_LEVEL.INFO,
        CommunicationServiceImplLifeSocial.class);
    connected = true;
    InfoCacheFetcher f = new InfoCacheFetcher(1000);
    f.start();
}
```

Listing 7.9: *initMessageApp()* method of the *CommunicationServiceImplLifeSocial*

The message application (see Listing 7.9 for *messageApp*) is needed for sending messages within LifeSocial.KOM. To understand how the integration into LifeSocial.KOM is realized, it is necessary to first explain how plugin-to-plugin communication is done within LifeSocial. The basic design concept behind LifeSocial.KOM is a plugin structure. Every plugin is registered at the startup of LifeSocial.KOM. Afterwards, it can communicate within LifeSocial.KOM using its unique plugin identifier. All data received by a LifeSocial.KOM instance is stored in the *InformationCache* labeled with the plugin identifier. All plugins periodically query the *InformationCache* for received data labeled with their plugin identifier.
So in case that a message should be send within LifeSocial.KOM it is necessary to define a *Header* object, which specifies the plugin ID need for communicating over the *InformationCache*. The information that is to be sent is encapsulated in a *BasicPayloadImpl* object, which is attached to a *BasicMessage* object. The *BasicMessage* object contains the information about the sender. In the last step the message application is used to send the message to the receiver as shown in Listing 7.10.

```
public void send(BigDecimal sender, BigDecimal receiver, Serializable s) {
```

```
  if (connected) {
    Header header = new HeaderImpl("monitoringObjectID", "monitoringPluginID", "
        monitoringDataID");
    BasicPayloadImpl payload = new BasicPayloadImpl(s);
    BasicMessage bm = new BasicMessage(new StorageKeyImpl(converter
        .convertToFreePastryId(sender).toString()),header,payload);
    messageApp.routeMsg(converter.convertToFreePastryId(receiver), bm, null);
  }
}
```

Listing 7.10: *send()* method of the *CommunicationServiceImplLifeSocial*

As already mentioned, all received messages are stored in the *InformationCache*. So it is necessary to have a separate thread running on each node, which periodically checks whether new monitoring data has arrived. This is done by the *InfoCacheFetcher* class as shown in Listing 7.11, which is a subclass of *java.lang.Thread*. In case that a new message has arrived, the containing payload is extracted and the *receive()* method of the *ICommunicationService* is invoked.

```
while (run) {
    try {
      while (connected
          && InformationCache.msgAvailable("monitoringPluginID")) {
        Message m = InformationCache.getMsg("monitoringPluginID");
        if (m instanceof BasicMessage) {
          BasicMessage msg = (BasicMessage) m;
          if (msg.getPayload() instanceof BasicPayloadImpl) {
            BasicPayloadImpl mpp = (BasicPayloadImpl) msg.getPayload();
            receive(mpp.getContent());
          }
        }
      }
      Thread.sleep(updateInterval);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
}
```

Listing 7.11: *run()* method of the *InfoCacheFetcher* class

7.9 Integration into PIPE

The second application the framework is integrated in, was PIPE. The *quap2p.dims. communication_pipe. osgi.core* bundle contains all the functionality for communicating within PIPE. The communication mechanisms used in PIPE and LifeSocial.KOM for forwarding data to higher application layers are different. As described in 7.8, LifeSocial.KOM plugins receive their data using a pull paradigm because they periodically start querying the *InformationCache*. Conversely, in PIPE data is pushed towards the receiver using the observer pattern. Furthermore, PIPE offers the possibility of notifying other components about a change in network connectivity. Applications or components interested in receiving updates register themselves as *IConnectionEventListener* as shown in Listing 7.12.

```
ServiceTracker sdCommTracker = new ServiceTracker(context,
        ISDCommunicationService.class.getName(), null);
sdCommTracker.open();
comService = (ISDCommunicationService) sdCommTracker.getService();
```

```
this.cm = comService.getCommunicationModule("myApp", this);

this.connManager = ConnectionManager.getConnectionManager();
this.connManager.registerConnectionEventListener(this);
```

Listing 7.12: Registration as *IConnectionEventListener*

The interface offers several methods through which the observers are notified in case that the local node got connected or disconnected to the network or in case that a connection attempt was unsuccessful.
In order to communicate within PIPE, it is necessary to create an instance of the *CommunicationModule*. This is done by querying the OSGI Service Registry for the *ISDCommunicationService* and invoking the *getCommunicationModule(String appID, Observer o)* method on it. The method expects two parameters: the first one is an application identifier of type String and the second one is of type *Observer*, in order to register the *CommunicationServiceImplPipe* as an observer at the communication module. Following the observer pattern, all received data is delivered to the observer through the *update(Observable o, Object ob)* method as shown in Listing 7.13.

```
public void update(Observable o, Object arg) {

  if (arg instanceof IReceivingObject) {
    IReceivingObject ro = (IReceivingObject) arg;
    receive(ro.getPayload());
  }
}
```

Listing 7.13: update() method of the CommunicationServiceImplPipe

To send data within PIPE it is necessary to encapsulate the *FreePastryID* inside a *IGenericNodeId* object and to create a new *SimpleParameterObject*. Afterwards, the communication module can be used for sending payload to the receiver as shown in Listing 7.14.

```
public void send(BigDecimal sender, BigDecimal receiver, Serializable s) {
    if (connected) {
      IGenericNodeId id = new RawPastryNodeId(converter.convertToFreePastryId(
          receiver));
      SimpleParameterObject spo = new SimpleParameterObject(id);
      cm.send(spo, s);
    }
}
```

Listing 7.14: send() method of the CommunicationServiceImplPipe

7.10 Export Formats

The current version of the monitoring framework supports exporting statistical data using the CSV and the GNUPlotDAT format. Listing 7.15 and 7.16 show an example of how the CSV and the DAT format are organized. Using the CSV format values are separated by comma and for each statistic a new row is created. The first row is always the timestamp information of the corresponding global statistic snapshot.

```
TIMESTAMP;sec;34;64;94;125;
AVERAGE_UPLOAD_NETWORK_USAGE_MEAN;%;16.40;17.84;20.88;18.91;...
AVERAGE_UPLOAD_NETWORK_USAGE_MIN;%;4.84;4.14;8.07;4.97;...
AVERAGE_UPLOAD_NETWORK_USAGE_SUM;%;2279.73;5370.37;3362.03;2931.62;...
AVERAGE_UPLOAD_NETWORK_USAGE_STANDARDDEVIATION;%;10.80;12.85;14.26;...
```

```
AVERAGE_UPLOAD_NETWORK_USAGE_COUNT;%;139.0;301.0;161.0;155.0;...
AVERAGE_UPLOAD_NETWORK_USAGE_VARIANCE;%;116.85;165.22;203.39;234.72;...
AVERAGE_UPLOAD_NETWORK_USAGE_SUMOFSQUARES
    ;%;53632.81;145548.58;102952.93;91830.63;...
AVERAGE_UPLOAD_NETWORK_USAGE_SUMOFSQUARES
    ;%;53632.81;145548.58;102952.93;91830.63;...
...
```

Listing 7.15: CSV Export Format

Using the DAT format data is organized in columns separated by spaces. The first column always contains the timestamp information of the corresponding global statistics snapshot.

```
TIMESTAMP AVERAGE_UPLOAD_NETWORK_USAGE_VARIANCE_%
    AVERAGE_UPLOAD_NETWORK_USAGE_MAX_%...
34 38.45 16.66 10.03...
64 107.50 39.22 20.37...
94 19.17 39.57 32.28...
125 159.81 54.11 39.59...
...
```

Listing 7.16: DAT Export Format

8 Evaluation

The following chapter evaluates the developed framework for monitoring P2P systems in order to provide a deeper insight in its performance. First, the evaluation criteria as well as the evaluation setup are described. Afterwards, the results of the Large Scale Test are presented and analyzed in detail.

8.1 Evaluation Criteria

Before evaluating a system, the evaluation criteria should be determined in order to be able to construct the evaluation setup accordingly. In case of the developed framework for monitoring P2P Systems there are three aspects that should to be evaluated. The first aspect is the framework property. In general, a framework should allow developers to create new components, which can be integrated easily into existing applications. In addition, new functionality should be added without great overhead. In order to evaluate this aspect in detail it is necessary to conduct a user study to obtain statistically significant data on how far the two properties are fulfilled. Due to the fact that the developed framework remains a prototype which will not be used for a large number of other applications, the evaluation of the integration and extendability property by conducting user studies is neglected.
The second aspect that needs to be evaluated is the P2P property. Like every P2P application build on top of a DHT, the framework should inherit its typical properties like self-organization, scalability and robustness. Furthermore, load balancing issues should also be taken into account, because an irregular distribution of load might lead to stability problems.
The gathered statistics and capacity data about the global system state form the third aspect of evaluation. They should be valid and consistent and in addition, the time and the resources needed to collect and compute them should be as low as possible.
In summary the evaluation should deliver insights in the following three main aspects:

1. Applicability of the developed framework
2. Quality of the used mechanism for gathering the system statistics and the capacity data
3. Validity and consistency of the delivered results

8.2 Evaluation Setup

The developed prototype was evaluated using a testbed with up to 500 nodes running on 37 physical computers. Every computer in the test scenario had at least 3 GHz (single core) of CPU power, 2 GB RAM and Windows XP installed. All computers were connected over a LAN without any Firewall or NAT installed. Every 30 seconds each peer appended its current snapshot of the global system statistics to its local log file. The log files were analyzed afterwards. In addition, a snapshot of the current tree topology was made every 5 minutes.
Every 5 seconds each peer in the network periodically sent its capacity and statistics update to its parent coordinator and simultaneously propagated the global system statistics down the tree. The small threshold for the update/propagate time has been chosen in order to simulate the system under a high load and to realize short update times. Given a real world application an update/propagate interval of about 60 seconds should be satisfactory.
The total simulation time was 10,000 seconds (2.7 hours). During the simulation time, different types

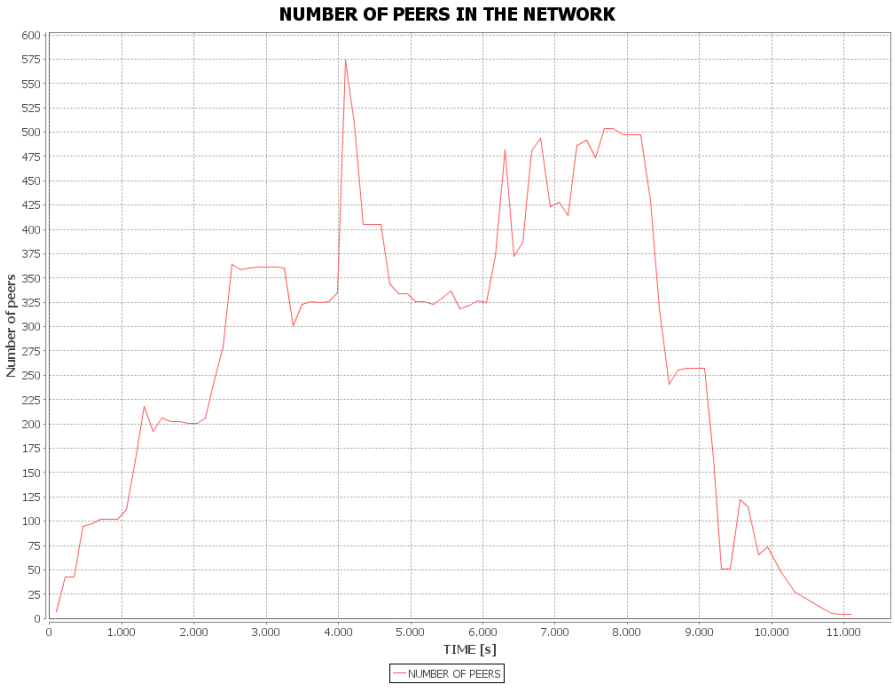


Figure 8.1: Detected number of peers during the large scale test

and levels of churn where tested (10%, 20%, 50% leaving peers and random churn).
Furthermore, in order to evaluate the list-based aggregation process for the capacity data, a GUI component for visualizing the current tree topology was implemented as every node in the network includes information about its responsibility range and its parent coordinator into the capacity update. Snapshots were taken every 5 minutes to document the development of the SkyEye.KOM tree topology.

8.3 Presentation and Analysis of the Large-Scale-Test Results

In the following the results of the large scale test are presented, analyzed and interpreted in order to evaluate whether the criteria outline above were fulfilled.

8.3.1 Number of peers

Figure 8.1 shows the run of the detected number of nodes in the network. Within the first 2500 seconds of the test, 350 peers joined the network successively. After the detected number of peers stabilized, the first churn test with about 10% was made (simulation time: 3500). 250 seconds later the correct number of peers (325) was detected again. Next, additional 75 peers joined the network so that the total number of peers was about 400. After a short phase of stabilization, 20% of the peers were disconnected from the network (simulation time: 4500). The leaving peers were succeesfully detected after about 250 seconds. For the next 1000 seconds a random churn test was conducted during which random peers joined and left the network under the condition that the total number of peers should remain constant. Finally, the total number of peers was increased to 500 and two 50% churn tests (50% leaving peer) were made, which was passed successfully. The results demonstrate that the developed prototype is churn resistant up to churn rates of 50%. The framework delivers useful results short after the underlying DHT has stabilized by reorganizing the responsibilities of all peers. During the large scale test, the time between the start of each churn phase and the stabilization was about 250 sec in mean. In case of leaving nodes

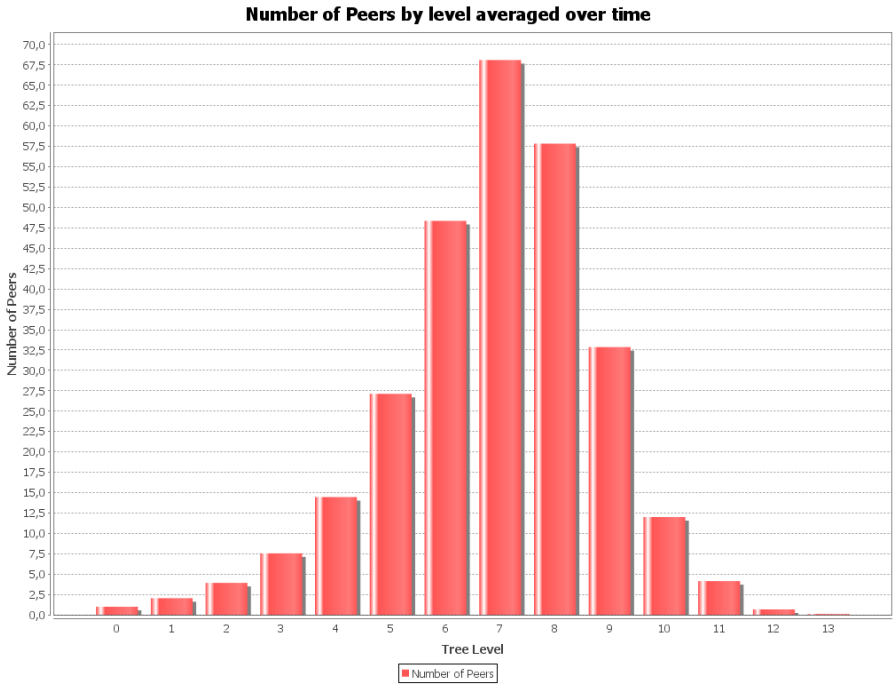


Figure 8.2: Number of peers by tree level averaged over time

the FreePastry DHT needs about 15 sec to recognize the peers that have left or failed. Afterwards, the DHT needs additional 10 sec to reorganize the responsibilies of the nodes. Finally, after the DHT has stabilized the tree structure is reestablished and the aggregate is recalculated along the path to the root of the tree.
Due to joining and leaving peers, the topology of the SkyEye.KOM tree changes which causes temporary errors in the global system statistics. For instance given the increasing number of peers after 4000 seconds. For a short time the number of peers detected was about 575. This overshoot is caused by topology changes that cause some peer aggregates temporarily being included twice into the global system statistics.

8.3.2 Distribution of Peers

As mentioned in the evaluation criteria section, load balancing is an important aspect for system stability. In order to guarantee a good load balancing, the SkyEye.KOM aggregation tree should be as balanced as possible. Furthermore, a well balanced aggregation tree will lead to a minimal tree depth resulting in a decreased time needed for computing aggregates. Figure 8.2 shows the distribution of peers by tree level averaged over the simulation time. As you can see, the distribution of the nodes in the levels 0 to 6 follows the function $f(x) = 2^x$ with x denoting the tree level. From level 7 to 13 the number of peers does not suffice to fill the remaining levels completely. Given the maximum number of 500 peers during the large scale test the optimal balanced tree would have $\lceil \log_2(500) - 1 \rceil = 8$ levels. To achieve a better distribution of the peers over the levels of the SkyEye.KOM tree it would be necessary to achieve a uniform distribution of nodes over the ID space. Another approach has been introduced in [11]: When the number of peers in the the subtree rooted by a node p drops below a certain threshold the child peers are ordered to send their updates to the parent coordinator of node p. Hence, this mechanism leads to a decreased depth of the tree.
Figure 8.3 shows the run of the number of peers at each level of the tree during the large scale test. You can see that the majority of peers was located in level seven and eight during the test. In case of joining

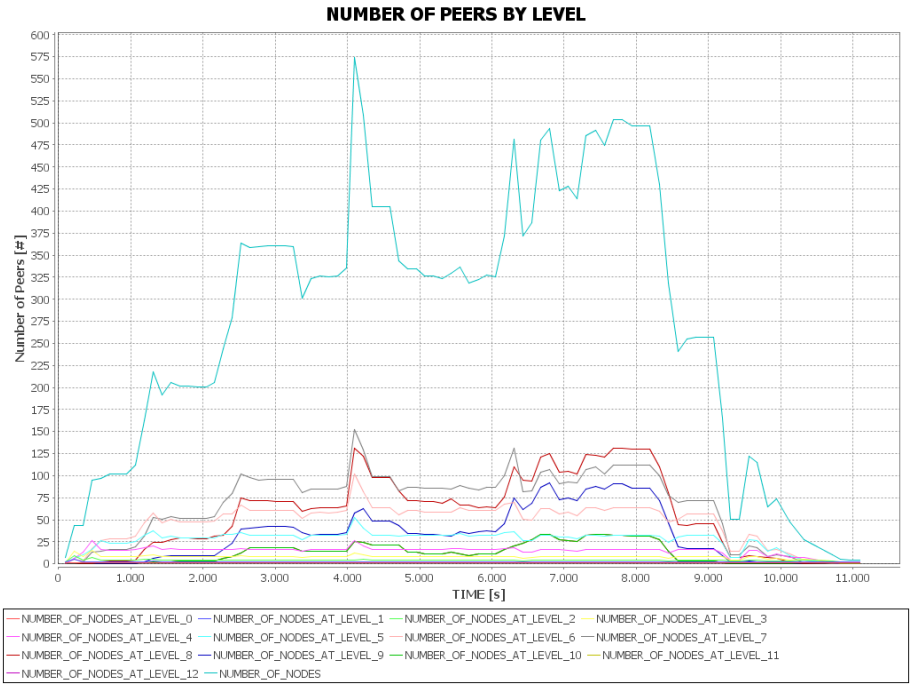


Figure 8.3: Number of peers by tree level

and leaving peers you can establish that the increases and the decreases of peers in the lower levels of the tree are greater than in the higher levels due to the reorganization of the tree. For example after the first test with 50% leaving peers (Time: 8500 seconds) the number of peers at level eight and nine declined harder then in the higher levels five and six.

8.3.3 Communication Costs

The most important parameter affecting the communication costs is the statistics and capacity update interval. During the large scale test an update interval of 5 seconds was used which depicts a very aggressive update strategy.

Average Bandwidth Consumption

Figure 8.4 shows the bandwidth utilization caused by the framework as well as the run of the number of peers in the network. For each data point the minimum, maximum and mean are plotted. The most important fact is that the bandwidth utilization averaged over all peers remains constant over time at a level of 3 KB/s. This means that 3 KB/s are necessary to obtain a global view of the system. On the other hand the maximum bandwidth utilization increases with an increasing number of nodes because of the increasing size of the capacity data due to the use of list-based aggregation.

Average Bandwidth Consumption Per Tree Level

The previous fact is also demonstrated in Figure 8.5. One can see that the bandwidth utilization towards the root increases exponentially due to the increasing amount of capacity data. With such an increasing bandwidth utilization the developed framework will not scale as nodes in the higher levels are likely to fail. So it is necessary to implement additional filter mechanisms or incremental updates such that the amount of data forwarded upwards in the tree remains almost constant. In case of the filter mechanisms the challenge is to construct them such that the capacity-based peer search continues delivering useful results, which means that only the best peers according to some metric should be returned.

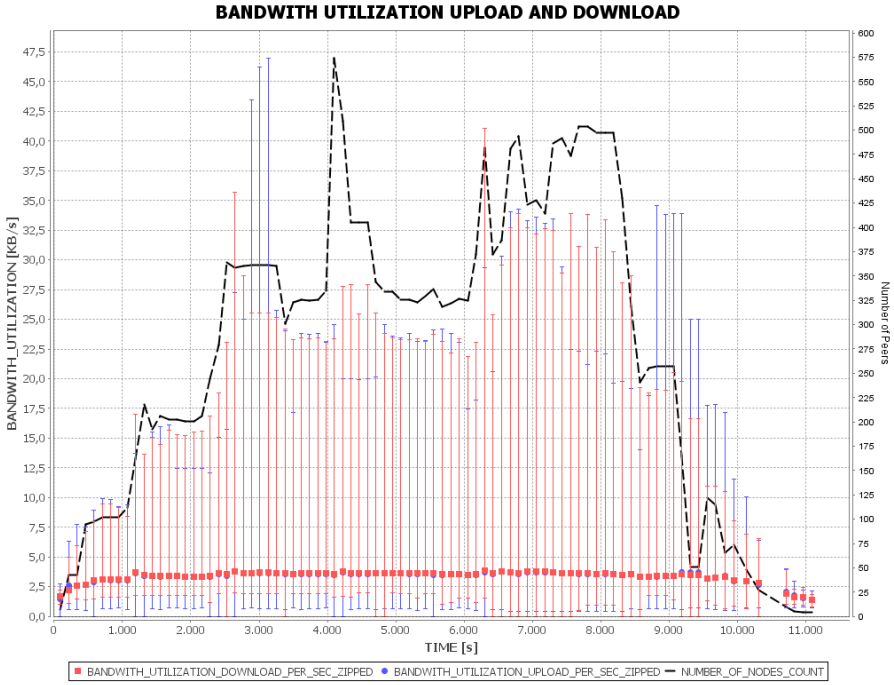


Figure 8.4: Bandwidth utilization per sec upload / download

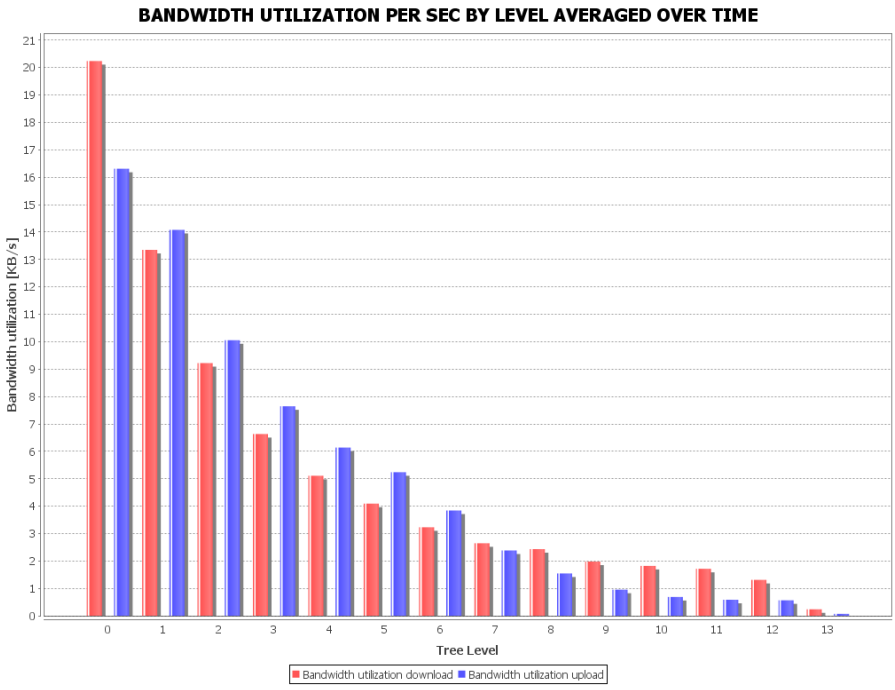


Figure 8.5: Bandwidth utilization per second by tree level averaged over time

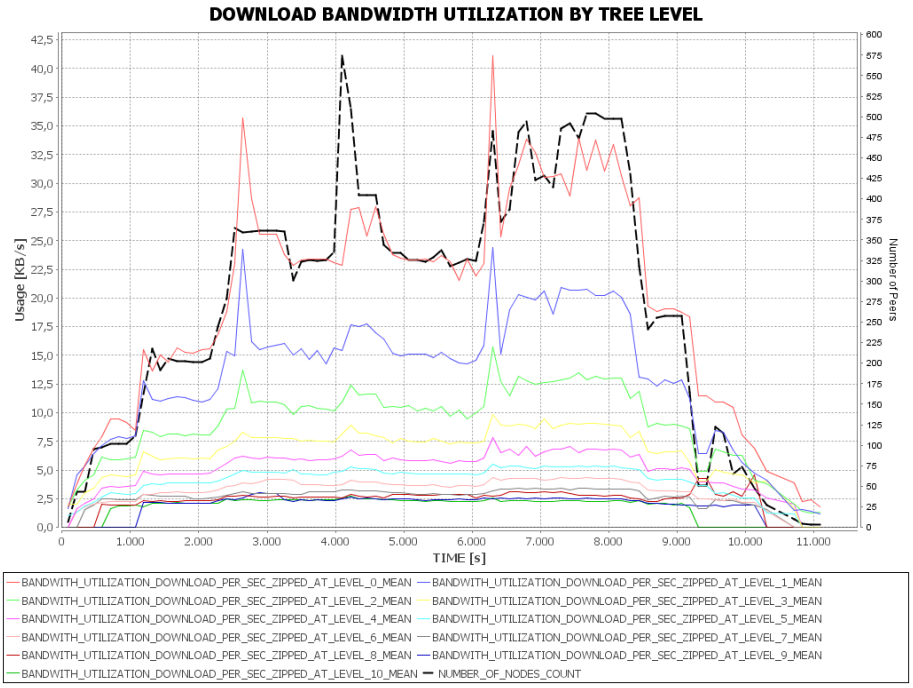


Figure 8.6: Correlation between the number of peers and the download bandwidth utilization

Another effect to notice is that the proportion between the downstream and upstream bandwidth utilization at the root is inverse to those of the lower levels. The reason is that the root is the top-most node and therefore it has no parent coordinator to which updates could be send to. In the lower levels the upstream drops below the downstream bandwidth utilization because the received propagation data send by the root is greater than the update send by these nodes.
An increasing or decreasing number of peers has a varying impact on the different tree levels and their bandwidth utilization, which is shown in the Figures 8.6 and 8.7. In the higher levels of the tree the fluctuating number of peers causes higher fluctuations in the bandwidth utilization. This behavior depict another clue that the higher levels of the tree, especially the root, become a bottleneck as they will not be able to handle the occurring large amount of data in case of a large number of peers participating in the network.

Number of Messages

Another important measure for determining the communication costs is the number of messages sent and received per second at each node as well as the information about the average message size at each level of the tree. Figure 8.8 demonstrates that the number of messages received and sent per second never exceeds the threshold of one. In the lower levels of the tree the amount of received messages per second declines as nodes on the lower levels tend to have less child nodes. In contrast, the root receives twice as much messages per second than sending messages. The reason is again, that the root node has no parent coordinator to which data could be send to.
In case of a fully balanced and complete SkyEye.KOM tree the number of messages received and sent at all levels would be the same except at the leaves and the root of the tree.

Average Hop Count

In order to determine how efficient a DHT handles the communication, the average hop count of a message traveling from the sender to the receiver is a good indicator. As you can see in Figure 8.9, the number of hops needed for delivering a message in mean is about 2 because the size of the DHT routing

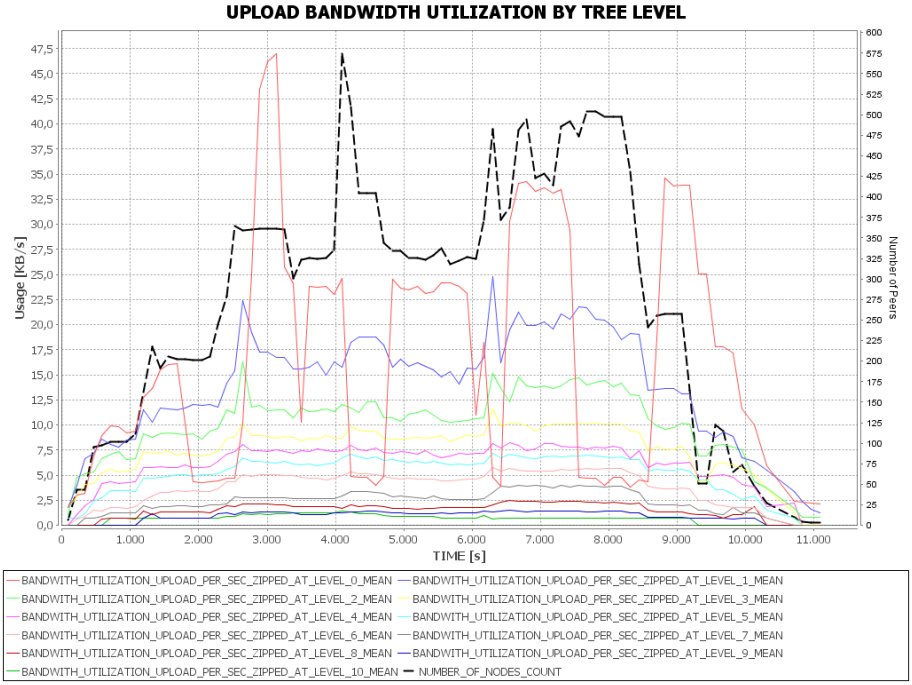


Figure 8.7: Correlation between the number of peers and the download bandwidth utilization

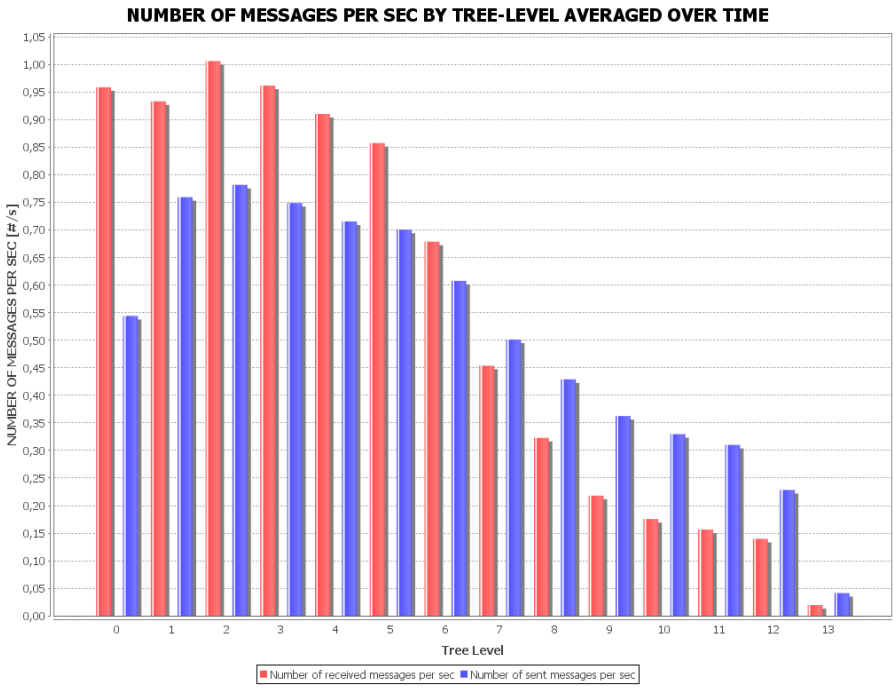


Figure 8.8: Number of messages sent and received per second by tree level averaged over time

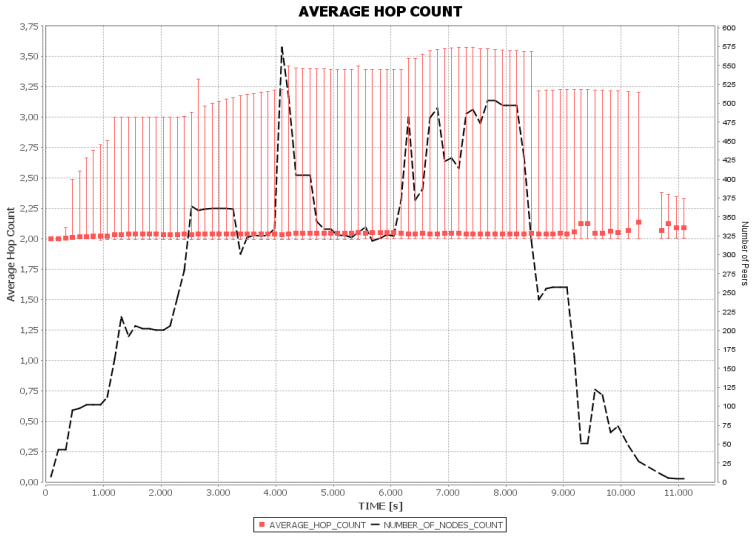


Figure 8.9: Average number of hops needed for delivering messages

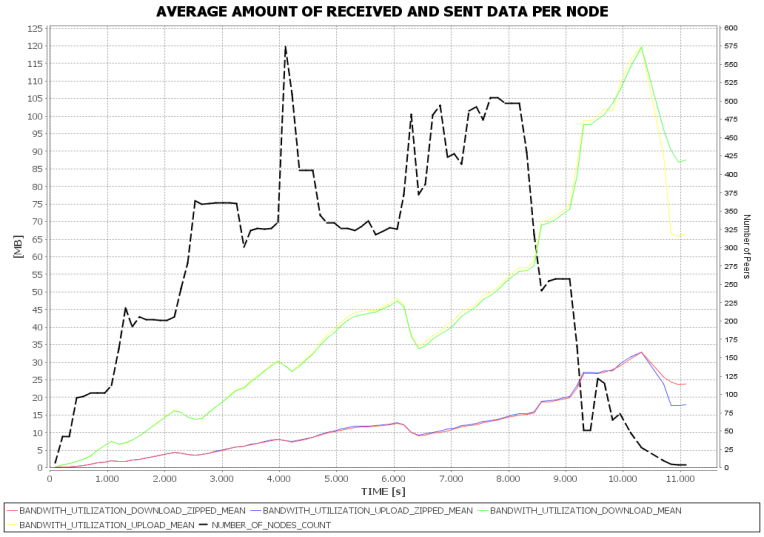


Figure 8.10: Amount of zipped and unzipped data send and received

table maintained by each node compared to the total number of nodes during the large scale test is huge due to the logarithmic scale of the routing table as explained in Chapter 4.2. Thus, messages sent could be delivered quite immediately at the receiver.
Another thing to mention is, that the maximum number of hops slightly increases with an increasing number of nodes.

Zipped vs. Unzippped Data

In order to minimize the bandwidth utilization, all data is zipped before sending leading to a reduction of up to 70% in the amount of data being sent as shown in Figure 8.10. Furthermore, you can see that no data is lost during communication between peers because the amount of sent data in average is always equal to the amount of received data. As one would expect, there is a correlation between the number of nodes and the average amount of sent and received data. If new peers join the network, the average value decreases due to the fact, that the new peers have not received any data yet. The great increase of the data curves during the 50% churn test occured because mostly "young" peers with a small uptime

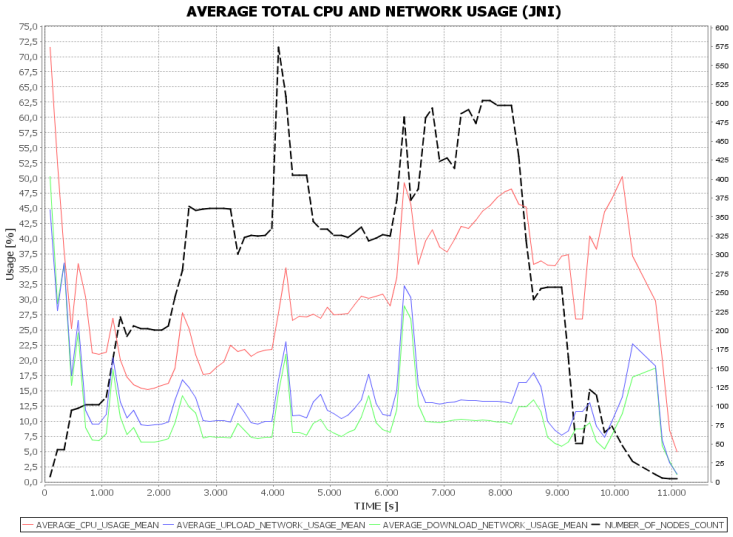


Figure 8.11: Average CPU and network usage

were disconnected. Due to the miss of these "young" peers the fraction of "old" peers, which had sent and received more data, increased resulting in a greater value for the average.

8.3.4 Resource Consumption

The resource consumption of a Distributed Information Management System should be as low as possible because the system is meant to be an additional component and should not interfere with the main application. During the large scale test several measures like CPU, network and memory usage were logged. Figure 8.11 shows the run of the curves for the CPU and network usage as well as the number of nodes. The CPU and the network usage were measured using the Java Native Interface (JNI) together with the Microsoft Performance Data Helper Library as described in 6.5. Thus, the values measured for the CPU and the network usage are system-wide and not limited to the monitoring framework. However, as no other applications were running on the computers except the monitoring framework. The CPU and network usage measured are assumed only being caused by the application. The values for the network usage have been calculated using the the approach as explained in Chapter 7.4.
As you can see, the CPU usage measured slightly increased during the large scale test from 15% in mean up to 50%. This increase was caused by the growing size of log files written during the large scale test by each peer, which caused a large amount of disc IO over time. In addition, it is noticeable that the CPU and network usage correlate because the runs of the three curves are similar. In case of joining and leaving peers an increase of the CPU and network usage is detected as joining and leaving peers, cause an additional communication overhead for reorganizing the DHT.
Figure 8.12 shows the heap memory usage during the large scale test. One can see that the heap memory usage slightly increases from about 20 MB in mean at the beginning of the test up to 80 MB towards the end. The increase is caused by the inefficient mechanism for log file writing, which had to deal with an increasing size of log files during the test. As a consequence an mechanism for incremental writing of log files would be necessary in order to reduce the disk access.

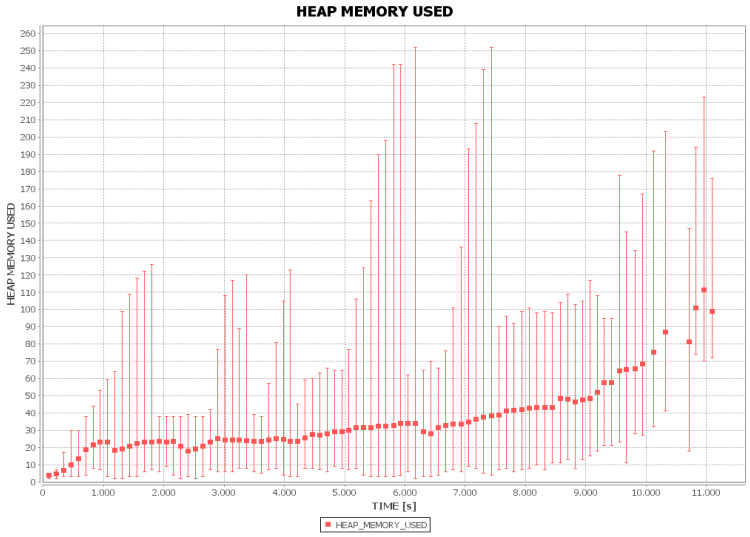


Figure 8.12: Heap memory used

8.3.5 List-based Aggregation

The evaluation of the list-based aggregation enabling capacity based peer search has been done by visualizing the topology information included in the capacity update. Each node appends information about its responsibility range and its parent coordinator to the capacity update. The information travels upwards in the tree until it reaches the root, where the information is parsed to a tree data structure, which is visualized.

The tree figures show the SkyEye.KOM tree during the large scale test with 44, 150 and 500 peers participating in the network. The red vertical lines visualize the responsibility range of each node. In addition, the node ID as well as the tree level are displayed. As you can see, the concatenation based aggregation delivers valid results in all three cases because the entire tree structure including all nodes in the network is displayed. Furthermore, the figures show that in case of a high density of IDs within a certain range, the depth of the tree increases. Conversely, if the IDs are uniformly distributed over the ID space, the tree becomes balanced. In order to have a quality measure for the distribution of nodes over the ID space, the relative deviation Δl of the tree depth from the minimal tree depth given the current number of peers N can be calculated using the following formula:

$$\Delta l = \frac{l_c - \lceil log_2(N+1) \rceil + 1}{\lceil log_2(N+1) \rceil - 1} * 100$$

with l_c denoting the currently observed tree depth.

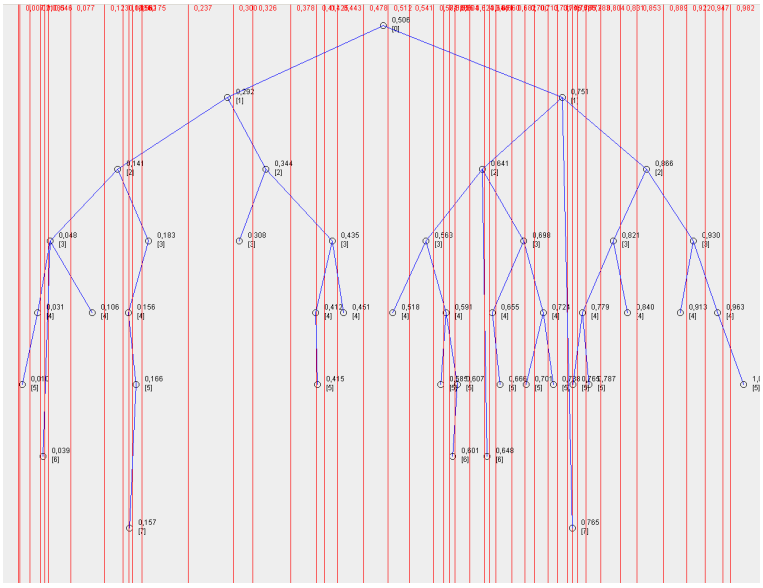


Figure 8.13: Tree visualization with about 44 peers

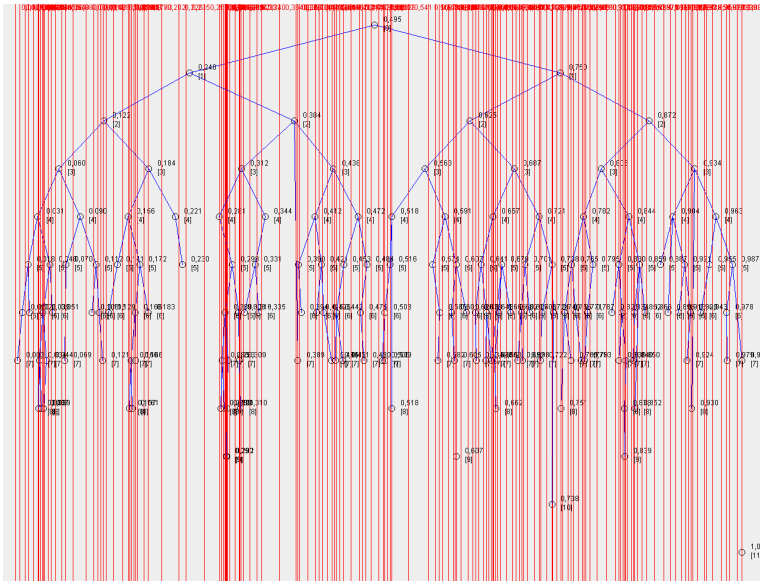


Figure 8.14: Tree visualization with about 150 peers

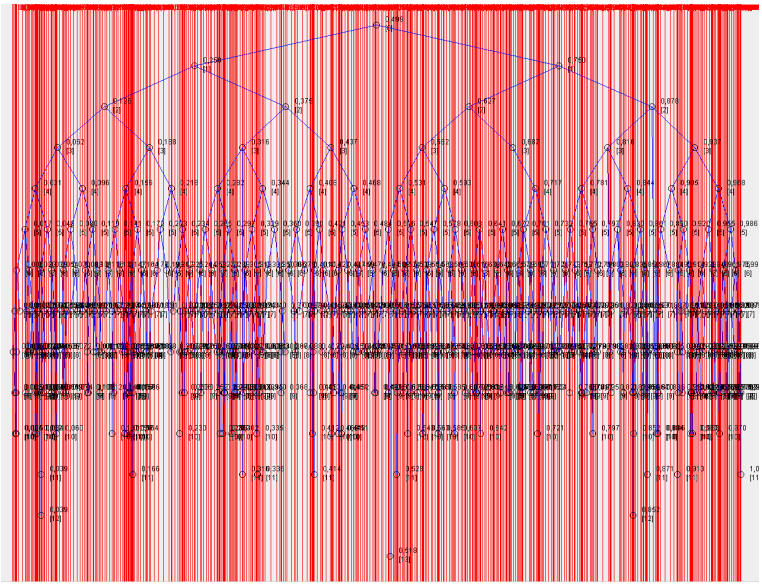


Figure 8.15: Tree visualization with about 500 peers

9 Conclusion

This section deals with the conclusion of this master thesis. Furthermore an outlook on the future work within the research area of self-optimizing P2P systems is given.

9.1 Summary

During this master thesis a framework for Distribute Information Management Systems (DIMS) has been developed enabling the monitoring of P2P systems. Furthermore a literature survey of state-of-the-art DIMS was conducted and a taxonomy has been developed. Using this taxonomy, DIMS can be classified according to the communication mechanism and aggregation topology used. Furthermore, properties like overlay independence, churn resistance, load-balancing and security aspects depict further classification criteria.
The prototype implements SkyEye.KOM, a DIMS which has been introduced in [11] enabling gathering system statistics on structured P2P systems as well as to search for peers offering specific capacities like CPU power, available bandwidth, etc. SkyEye.KOM uses a single tree based topology for aggregating peer data. Furthermore, SkyEye.KOM has the advantage of being overlay independent by introducing its own ID space ranging from 0 to 1. Using mapping functions, the entire ID space of the underlying DHT is mapped completely onto the SkyeEye.KOM ID space.
The architecture of the framework was designed with extendability in mind. Future developer will be able to add new functionality easily, such that new attributes as well as new aggregation topologies can be supported. Furthermore, the architecture was designed to ease the integration into existing P2P applications. In order to accomplish these two properties the framework was built using OSGi (Open Service Gateway Initiative), a service platform for modular Java applications. Consequently, the architecture is composed of a number of modules called bundles. In case that a bundle requires functionally provided by another bundle, the functionality always is accessed through an interface. This approach allows developers to alter or extend existing functionality easily, because bundles only depend on the interface definition and not on the implementation of the interface.
The framework has been successfully integrated into two existing P2P applications which also have been built on top of OSGi. The first application is LifeSocial.KOM, a P2P based social community platform. PIPE, a distributed IDE based on Eclipse, is the second application the framework was integrated in.
The evaluation, using a testbed consisting of up to 500 peers, showed that the mechanism for gathering the system statistics as well as the capacity data is churn resistant with churn rates of up to 50%. Furthermore, by zipping all communication data, a reduction in the bandwidth utilization of up to 70% was accomplished at the expense of CPU power. With peers sending their updates periodically every 5 seconds, only 3 KB/s in mean are necessary to obtain a global view of a P2P system. However, it should be noted that an update interval of 5 seconds depicts an aggressive update strategy. For most existing P2P applications an update interval of about 30 to 60 seconds will propably be satisfactory resulting in a bandwidth utilization of about 200 bytes per second.
Another important aspect that was discovered during the evaluation process was, that due to the concatenation based aggregation, a high bandwidth consumption was caused in the higher levels of the SkyEye.KOM tree. Without additional filter mechanisms or incremental updates, the application does not scale.
The analysis of the resource consumption has shown that the CPU usage increases during the simulation time due to an inefficient mechanism for log file writing. With up to 30 instances running on a single

computer the CPU usage was about 50% in mean and the overall bandwidth cunsumption was about 20% in mean. By improving the mechansism for the log file writing the CPU consumption could be decreased. With this improvement a single instance of the monitoring framework on a single computer can be run without affecting other applications as the resource consumption caused by the monitoring framework will be even low.
At the beginning of evaluation process the following three main aspects for evaluation were pointed out:

1. Applicability of the developed framework
2. Quality of the used mechanism for gathering the system statistics and capacity data
3. Validity and consistency of the delivered results

which now can be answered as follows:

1. The applicable of the framework has been proven with two successful integrations into two existing applications.
2. The mechanism has proven to be churn resistant with churn rates of up to 50% but suffers from scalability problems.
3. The results delivered by the framework are valid and consistent.

In summary, the developed framework for monitoring P2P systems constitutes a first step towards self-optimizing and adaptive P2P systems.

9.2 Future Work

As mentioned in the conclusion, the developed framework only constitutes the fist step towards self-optimizing P2P systems. Hence, a lot of research work remains to be done. Given Figure 9.1, which has already been introduced in Chapter 2.1, one of the next steps is developing an analytic model which is able to interpret the data delivered by the monitoring process. A very promising way for accomplishing this goal seems to be the use of data mining algorithms, e.g. rule or tree learners as they both are able to handle complex explanatory models. The challenge is to discover those attributes of a P2P system that form a major part of the explanatory model. In other words, we need to find attributes such that the system behavior can be explained with a minimum of required data . Finally, the analysis process should deliver a clue which measure of a P2P systems is currently out of range and should be improved.
Given this glue, a decision model decides which parameter of the P2P systems must be adjusted in order to accomplish an optimized system state. The key issues for developing a decision model are:

1. Where are decisions made?
2. What decisions are made?
3. What granularity and specificity do decisions have?
4. How are decisions implemented?

Decisions within P2P systems should be made in decentralized fashion to avoid interferences with the P2P communication paradigm. The second and third issue address the fact, that different underlays might be used within P2P systems. So it is necessary to make decisions of a more general nature and to propagate them downwards to the underlay where the decision is applied. For a P2P system built on top of a Chord DHT. If we have a decision that requires the finger table to be increased in order to optimize the hop count, this decision could only be implemented if a Chord DHT is present. So if the specificity of

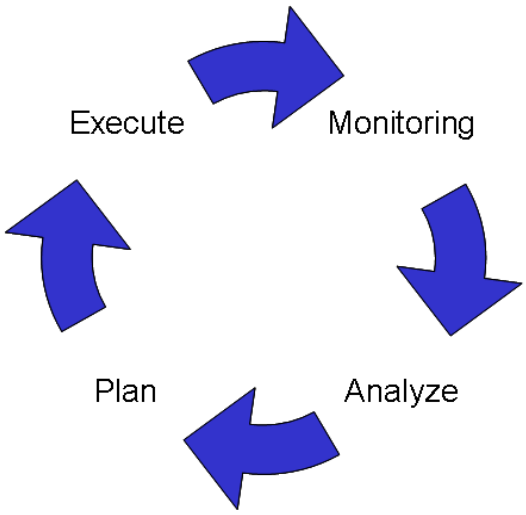


Figure 9.1: Closed loop for optimizing P2P systems

a decision is too high, it cannot be implemented on other P2P systems. This leads to the fourth issue. To overcome this problem, a hierarchical approach for decision deployment would be possible. Decisions made by the decision model should have a general nature like decreasing the hop count. After propagating decisions down to lower functional layers, each layer itself could decide how the decision has to be implemented in order to accomplish an optimized system state.
Furthermore, applications build on top of DHTs and DHTs itself, should provide a simple API for changing system parameters at runtime. Currently, most systems do not support changing system parameters during runtime. In these cases it is mostly necessary to restart the system after a change in the system parameters.
Like most developed applications the current version of the framework offers a potential for optimization. As mentioned in the evaluation chapter the bandwidth utilization caused by the list-based aggregation could be decreased by implementing filter mechanisms or incremental updates. Furthermore, the security issues as lined out in Chapter 4.9 should also be addressed.
The architecture of the framework was designed to support different aggregation topologies. By adding new aggregation topologies to the framework, the performance of the different aggregation topologies could be compared within a larger testbed.

A Appendix

A.1 Configuration File

```
#************************** Topology Configuration
    *********************************
#Tree splitting factor
TREE_SPLITTING_FACTOR=2

#min peers responsible for
T_MIN=-1

#max peers responsible for
T_MAX=2

#number of peers at which support peer is necessary
T_SUPPORT=20

#Loadbalancing time in ms
LOADBLANCING_TIME=60000

#support peer time in ms
SUPPORT_PEER_TIME=60000

#************************** Communication Configuration
    ******************************
#Boot address
BOOT_ADDRESS=192.168.178.21

#Boot port
BOOT_PORT=9001

#Bind port
BIND_PORT=9001

#Communication Wrapper (Pipe, FreePastry, LifeSocial)
#When using Freepastry, do not forget to enable the freepastry lib bundle quap2p.
    dims.freepastry
COM_WRAPPER=FreePastry

#************************** Storage Configuration ******************************
#Heart beat timeout in ms
HEART_BEAT_TIMEOUT=15000

#************************** Statistics Configuration
    ******************************
#Update interval in ms (must be smaller than HEART_BEAT_TIMEOUT, otherwise data
    will be delete before updateing
STAT_UPDATE_TIME=5000
```

```
42 #************************** Capacities Configuration
       ******************************
   #Update interval in ms (must be smaller than HEART_BEAT_TIMEOUT, otherwise data
       will be delete before updateing
44 CAPACITY_UPDATE_TIME=5000

46 #************************** Capacities Configuration
       ******************************
   #Query Timeout Time
48 QUERY_TIMEOUT=10000

50 #************************** STATISTICS OUTPUT Configuration
       ******************************
   #wihtout "/" at the end
52 #define own dir oder type "user.home"
   STAT_OUPUT_DIR=c:/Temp
```

Listing A.1: Configuration file for the framework

A.2 Starting the Monitoring Framework

The framework was integrated into two existing applications. Furthermore, a stand-alone version using only FreePastry has been developed. Hence, for each type a different collection of bundles must be installed and activated.

A.2.1 PIPE

The following bundles are necessary to start the framework using PIPE:

- quap2p.basicStorage
- quap2p.chatMessenger.osgi.app
- quap2p.communicationModule
- quap2p.connectionManager.osgi.gui
- quap2p.fw.communicationModule.osgi.core
- quap2p.fw.serviceDispatcher
- quap2p.serviceDispatcher
- quap2p.serviceDispatcher.osgi.gui
- quap2p.storageModule
- tmp.refactoredRevisionStructure
- quap2p.askme.plugin.app
- quap2p.connectionManager.osgi.core
- quap2p.connectionManager.plugin.gui

- quap2p.contactManager.plugin.app
- quap2p.dims.capacity_input.osgi.core
- quap2p.dims.capacity_output.osgi.core
- quap2p.dims.communication_pipe.osgi.core
- quap2p.dims.eclipsegui
- quap2p.dims.gui
- quap2p.dims.queryModul.osgi.core
- quap2p.dims.sensor.osgi.core
- quap2p.dims.statistics_input.osgi.core
- quap2p.dims.statistics_output.osgi.core
- quap2p.dims.storage.osgi.core
- quap2p.dims.topolopgy.osgi.core
- quap2p.dims.util.osgi.core
- quap2p.p2pvcs.core
- quap2p.piki.plugin.core
- quap2p.piki.plugin.gui
- quap2p.serviceDispatcher.osgi.core
- quap2p.serviceDispatcher.plugin.gui

A.2.2 LifeSocial.KOM

The following bundles are necessary to start the framework within LifeSocial.KOM:

- org.lifesocial.ui
- org.lifesocial.ui.browser
- org.lifesocial.ui.common
- org.lifesocial.ui.friends
- org.lifesocial.ui.groups
- org.lifesocial.ui.livechat
- org.lifesocial.ui.login2
- org.lifesocial.ui.messaging
- org.lifesocial.ui.myprofile
- org.lifesocial.ui.photos

- org.lifesocial.ui.search
- org.lifesocial.ui.tests
- quap2p.dims.capacity_input.osgi.core
- quap2p.dims.capacity_output.osgi.core
- quap2p.dims.communication_lifesocial.osgi.core
- quap2p.dims.extensions.osgi.gui
- quap2p.dims.queryModul.osgi.core
- quap2p.dims.sensor.osgi.core
- quap2p.dims.statistics_input.osgi.core
- quap2p.dims.statistics_output.osgi.core
- quap2p.dims.storage.osgi.core
- quap2p.dims.topolopgy.osgi.core
- quap2p.dims.util.osgi.core

A.2.3 FreePastry

The following bundles are necessary to start the framework using only FreePastry:

- quap2p.dims.capacity_input.osgi.core
- quap2p.dims.capacity_output.osgi.core
- quap2p.dims.communication_freepastry.osgi.core
- quap2p.dims.communication.osgi.gui
- quap2p.dims.queryModul.osgi.core
- quap2p.dims.sensor.osgi.core
- quap2p.dims.statistics_input.osgi.core
- quap2p.dims.statistics_output.osgi.core
- quap2p.dims.storage.osgi.core
- quap2p.dims.topolopgy.osgi.core
- quap2p.dims.util.osgi.core

A.3 Configuration.ini File

```
#Product Runtime Configuration File

osgi.splashPath=platform:/base/plugins/quap2p.dims.starter.osgi.core
eclipse.product=quap2p.dims.starter.osgi.core.dims
osgi.bundles.defaultStartLevel=4
osgi.bundles=org.eclipse.core.contenttype,org.eclipse.core.jobs,org.eclipse.core.
    runtime@start,org.eclipse.core.runtime.compatibility.registry,org.eclipse.
    equinox.app,org.eclipse.equinox.common,org.eclipse.equinox.preferences,org.
    eclipse.equinox.registry,quap2p.dims.capacity_input.osgi.core@start,quap2p.dims
    .capacity_output.osgi.core@start,quap2p.dims.starter.osgi.core@start,quap2p.
    dims.communication_freepastry.osgi.core@start,quap2p.dims.queryModule.osgi.
    core@start,quap2p.dims.sensor.osgi.core@start,quap2p.dims.statistics_input.osgi
    .core@start,quap2p.dims.statistics_output.osgi.core@start,quap2p.dims.storage.
    osgi.core@start,quap2p.dims.topolopgy.osgi.core@start,quap2p.dims.util.osgi.
    core@start,quap2p.dims.extensions.osgi.gui@start,org.eclipse.equinox.launcher,
    org.eclipse.equinox.launcher.win32.win32.x86
eclipse.noShutdown=true
eclipse.console=true
```

Listing A.2: config.ini file

A.4 Eclipse.ini File

```
-console
-noExit
-vmargs
-Xms40m
-Xmx128m
```

Listing A.3: eclipse.ini file

A.5 Bat File for Starting 10 Peers

```
set host=192.168.178.21
set timer=15
START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 9001
WAIT %timer%
START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 %host% 9001 9002
WAIT %timer%
START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 %host% 9001 9003
WAIT %timer%
START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 %host% 9001 9004
WAIT %timer%
START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 %host% 9001 9005
WAIT %timer%
START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 %host% 9001 9006
WAIT %timer%
```

```
15 START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 %host% 9001 9007
   WAIT %timer%
17 START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 %host% 9001 9008
   WAIT %timer%
19 START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 %host% 9001 9009
   WAIT %timer%
21 START /B eclipse.exe -quap2p.dims.starter.osgi.core.id1 %host% 9001 9010
   EXIT
```

Listing A.4: bat file for starting 10 peers on a local pc

A.6 Further Simulation Results

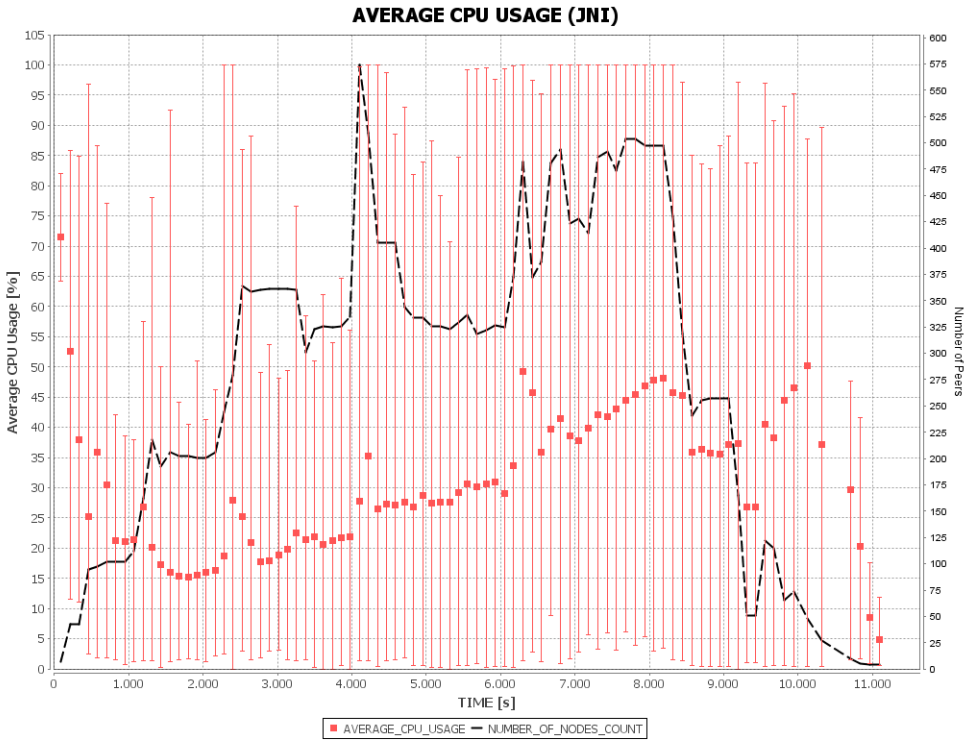


Figure A.1: Average CPU usage

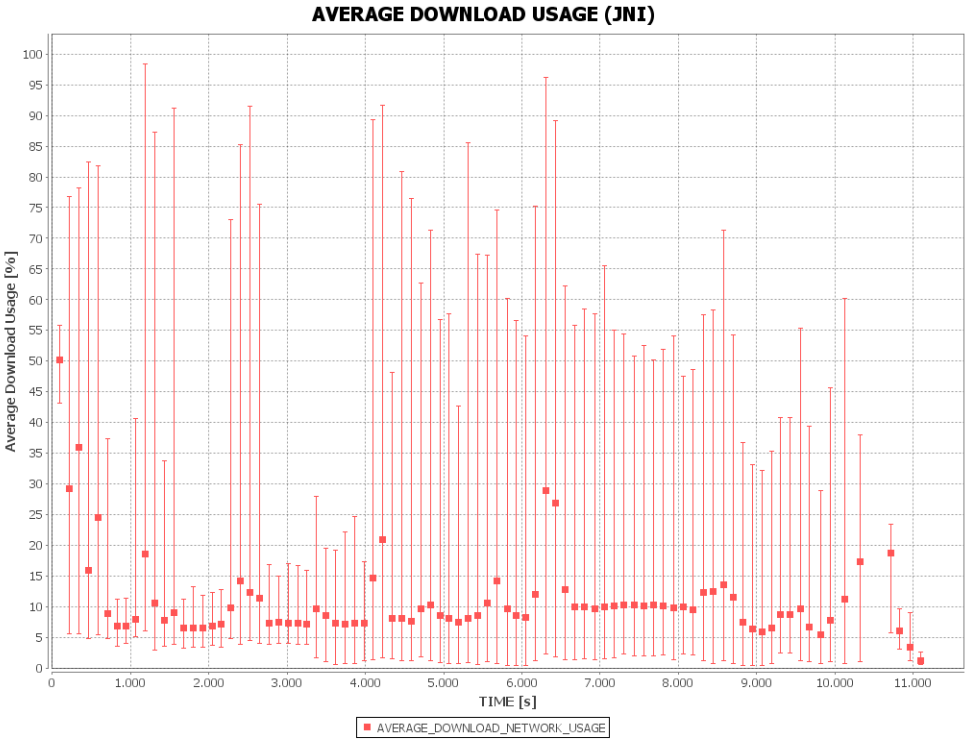


Figure A.2: Average download usage

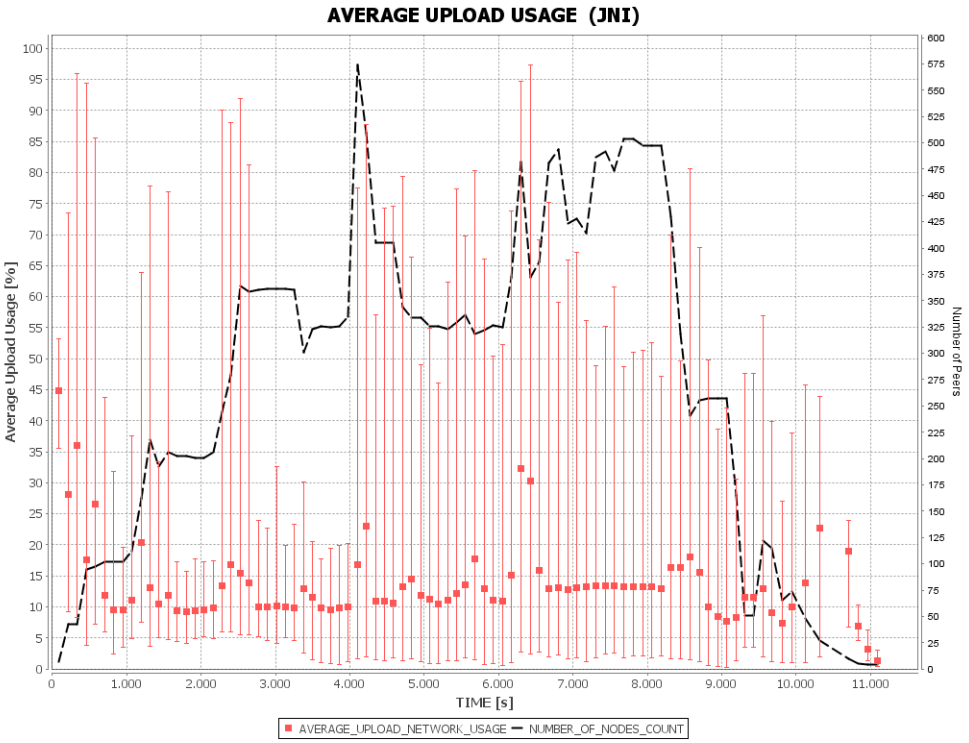


Figure A.3: Average upload usage

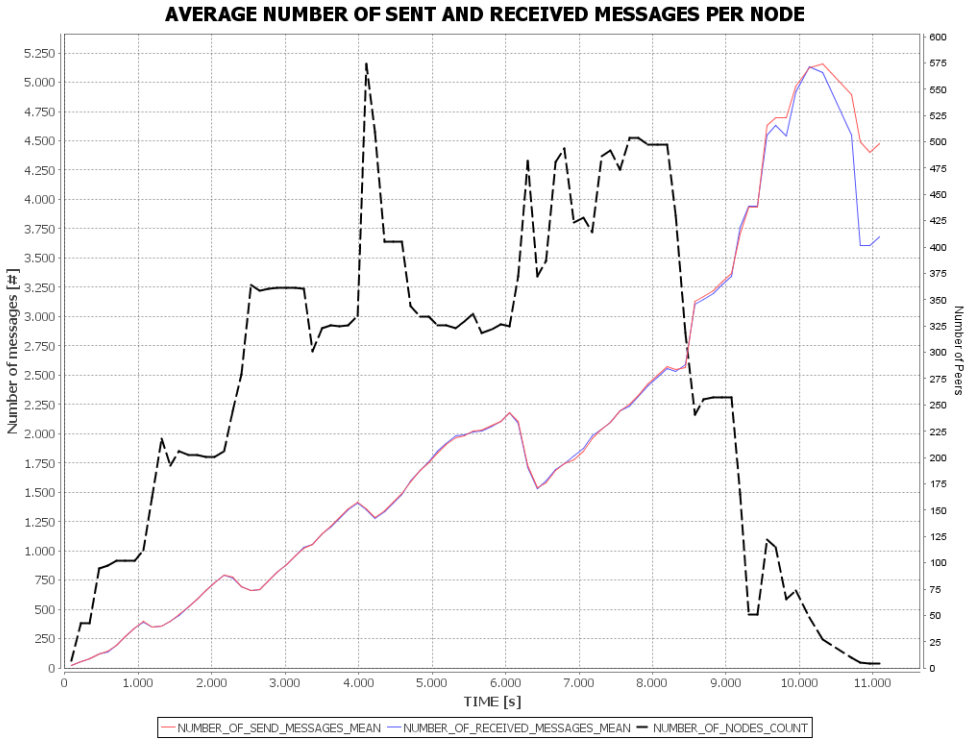


Figure A.4: Average number of messages

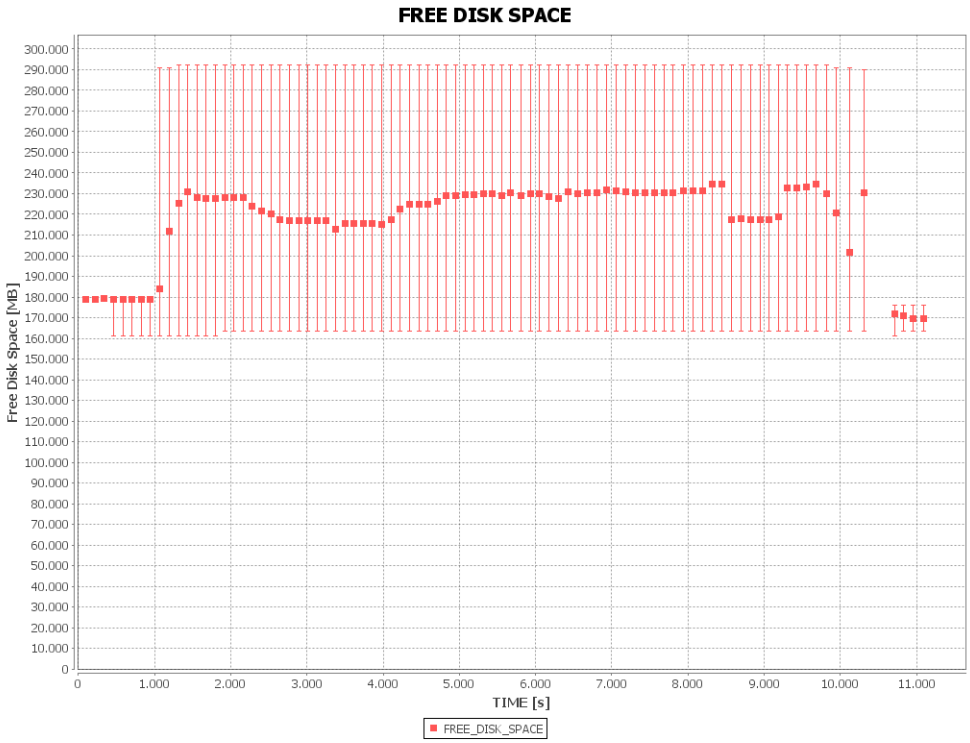


Figure A.5: Free disc space

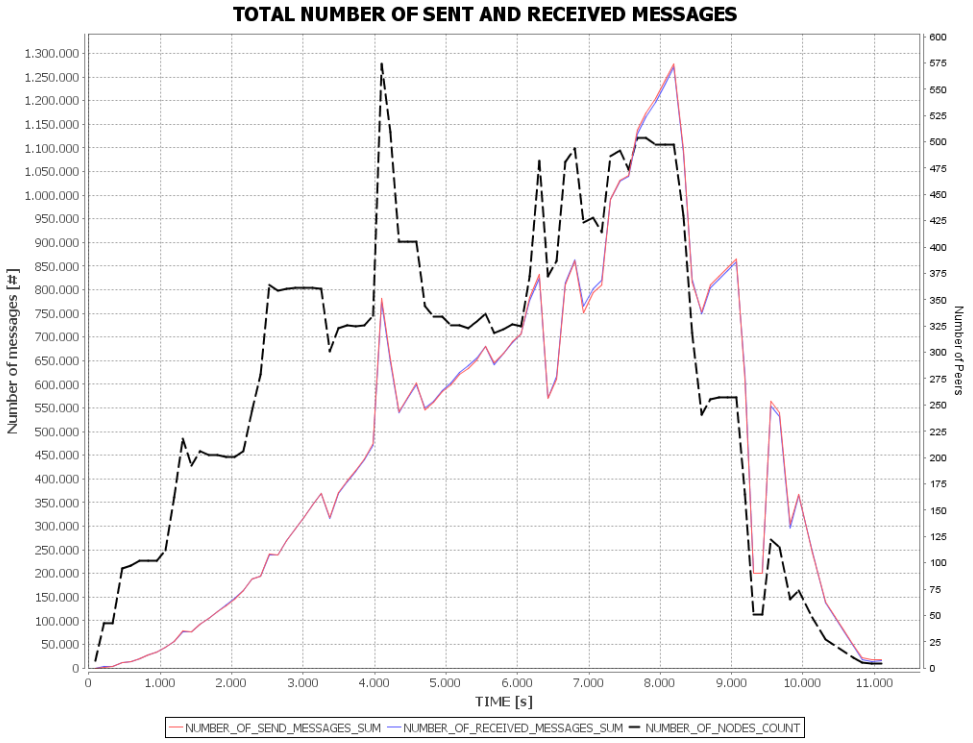


Figure A.6: Total number of messages

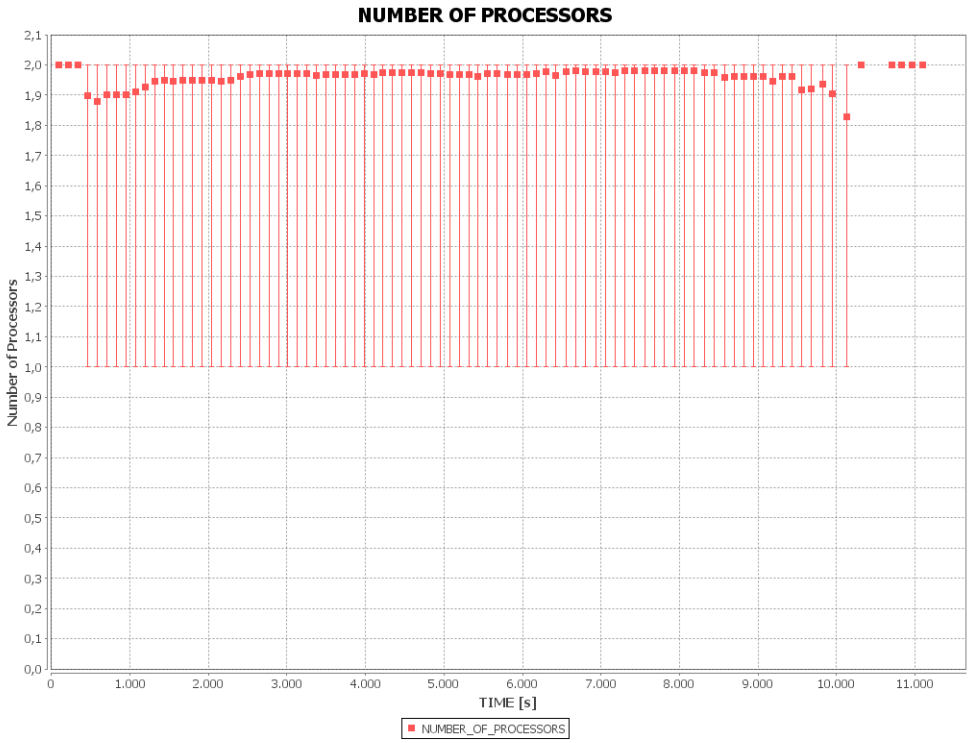


Figure A.7: Average number of processors

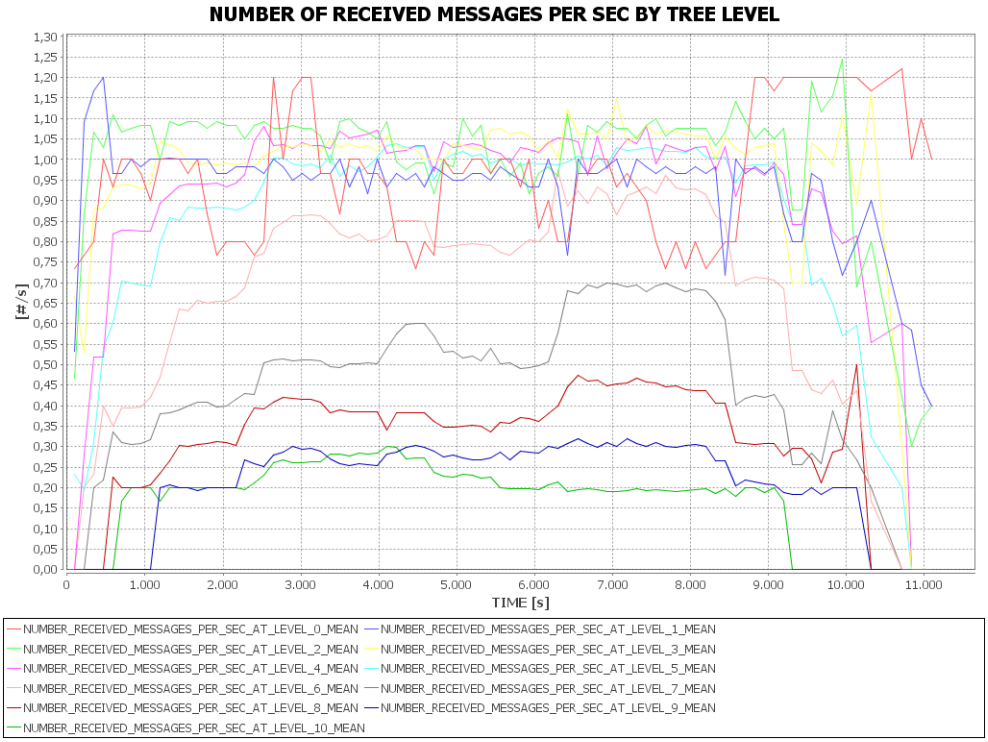


Figure A.8: Number of received messages per sec per level

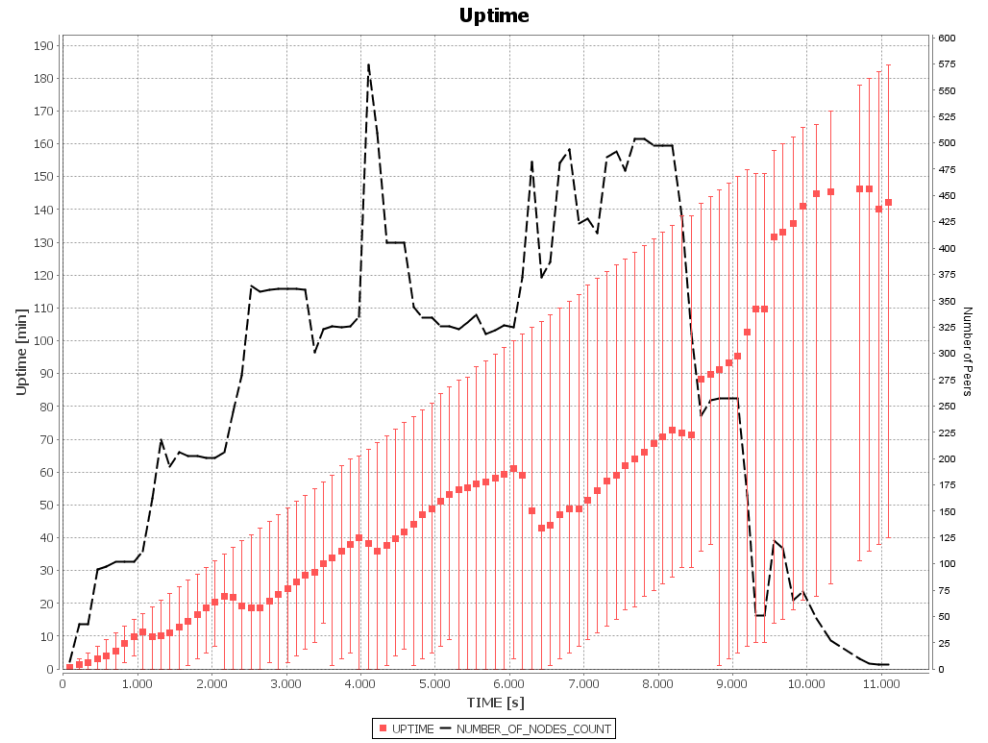


Figure A.9: Average uptime

Bibliography

[1] Luciana Alivite. Design and Implementation of a Service-oriented Dynamic and Modular P2P-based Multimedia Online Community Platform using OSGi. n/a, 2008.

[2] M Bawa, H Garcia-Molina, A Gionis, and R Motwani. Estimating aggregates on a peer-to-peer network. In *n/a*, 2003.

[3] Ranjita Bhagwan, George Varghese, and Geoffrey M. Voelker. Cone: Augmenting DHTs to support distributed resource discovery. Technical report, n/a, 2003.

[4] OSGI Consortium. Open Service Gateway Initiative. `http://www.osgi.org/Main/HomePage`. [Online; accessed December-2008].

[5] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service. *Hot Topics in Operating Systems, Workshop on*, 0:0081, 2001.

[6] A. Demers, D. Greene, C. Hauser, and W. Irish. Epidemic algorithms for replicated database maintenance. *Proc. of the sixth ACM Symp. on Principles of Distributed Computing*, n/a:1–12, August 1987.

[7] Apache Software Foundation. Log4J. `http://logging.apache.org/log4j/`. [Online; accessed Feb-2009].

[8] Eclipse Foundation. Equinox OSGI. `http://www.eclipse.org/equinox/`. [Online; accessed December-2008].

[9] Nils Hartmann Gerd Wütherich. *Die OSGI Service Platform*. dpunkt Verlag, 2008.

[10] GNUPlot. GNUPlot. `http://www.gnuplot.info/`. [Online; accessed Feb-2009].

[11] Kalman Graffi, Aleksandra Kovacevic, Song Xiao, and Ralf Steinmetz. SkyEye.KOM: An Information Management Over-Overlay for Getting the Oracle View on Structured P2P Systems. In *The 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS'08)*, page 8, Melbourne, Australia, Dec 2008. IEEE, IEEE Computer Society Press.

[12] ipoque GmbH. Internetstudie 2008/2009. `http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009`. [Online; accessed Feb-2009].

[13] Márk Jelasity. Epidemic-style proactive aggregation in large overlay networks. In *In Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 04*, pages 102–109. IEEE Computer Society, 2004.

[14] Márk Jelasity and Ozalp Babaoglu. T-Man: Gossip-based overlay topology management. In *In 3rd Int. Workshop on Engineering Self-Organising Applications (ESOA 05*, pages 1–15. Springer-Verlag, 2005.

[15] JGNUPlot. JGNUPlot. `http://jgnuplot.sourceforge.net/`. [Online; accessed Feb-2009].

[16] KOM. LifeSocial - A Distributed Platform for Multimedia Online Communities. `http://www.lifesocial.org/`. [Online; accessed December-2008].

[17] Object Refinery Limited. http://www.jfree.org/jfreechart/. http://www.jfree.org/jfreechart/. [Online; accessed Feb-2009].

[18] Microsoft. Performance Data Helper(PDH). http://msdn.microsoft.com/en-us/library/aa373083(VS.85).aspx. [Online; accessed December-2008].

[19] University of Waikato. Weka. http://www.cs.waikato.ac.nz/ml/weka/. [Online; accessed Feb-2009].

[20] Mike Dahlin Praveen Yalagandula. Research Challenges for a Scalable Distributed Information Management System. Technical report, In Proc SIGCOMM, 2004.

[21] Klaus Werle Ralf Steinmetz. *Peer-to-Peer Systems and Applications*. Springer Berlin / Heidelberg, 2005.

[22] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21:2003, 2003.

[23] USA Rice University, Houston. Freepastry. http://freepastry.org/FreePastry/. [Online; accessed Feb-2009].

[24] Adrain Bozdog Robbert van Renesse. Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol.

[25] Vladimir Roubtsov. Profiling CPU usage from within a Java application. http://www.javaworld.com/javaworld/javaqa/2002-11/01-qa-1108-cpu.html. [Online; accessed December-2008].

[26] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[27] SQLite. www.sqlite.org/. http://www.sqlite.org/. [Online; accessed Feb-2009].

[28] sqlitejdbc. http://zentus.com/sqlitejdbc/. http://zentus.com/sqlitejdbc/. [Online; accessed Feb-2009].

[29] Dominik Stingl. Development of a Self-Optimizing Life Cycle Framework for Structured Peer-to-Peer Systems. Master's thesis, Technische Universität Darmstadt, 2008.

[30] Sun. Java Native Interface (jni). http://java.sun.com/j2se/1.4.2/docs/guide/jni/. [Online; accessed December-2008].

[31] Sun. java.util.logging. http://java.sun.com/javase/6/docs/api/java/util/logging/package-summary.html. [Online; accessed Feb-2009].

[32] Sun. Trail: The Reflection API. http://java.sun.com/docs/books/tutorial/reflect/. [Online; accessed December-2008].

[33] Wikipedia. Control Theory. http://en.wikipedia.org/wiki/Control_theory. [Online; accessed Feb-2009].

[34] Wikipedia. Decision_Tree. http://en.wikipedia.org/wiki/Decision_tree. [Online; accessed March-2009].

[35] Wikipedia. Messabweichung. http://de.wikipedia.org/wiki/Messabweichung. [Online; accessed March-2009].

[36] Wikipedia. Monitoring. http://de.wikipedia.org/wiki/Monitoring. [Online; accessed Feb-2009].

[37] Wikipedia. OSGI. http://en.wikipedia.org/wiki/OSGi. [Online; accessed March-2009].

[38] Wikipedia. Osgi_framework. http://en.wikipedia.org/wiki/File:Osgi_framework.svg. [Online; accessed March-2009].

[39] Wikipedia. Rueckwirkungsabweichung. http://de.wikipedia.org/wiki/Rueckwirkungsabweichung. [Online; accessed Feb-2009].

[40] Wikipedia. Java DataBase Connectivity. http://de.wikipedia.org/wiki/Java_Database_Connectivity. [Online; accessed Oktober-2008].

[41] Wikipedia. ODBC. http://de.wikipedia.org/wiki/ODBC. [Online; accessed Oktober-2008].

[42] Praveen Yalagandula. A scalable distributed information management system. In *In Proc SIGCOMM*, pages 379–390, 2004.

[43] Zheng Zhang. SOMO: Self-Organized Metadata Overlay for Resource Management. In *in P2P DHT. In Proceedings of IPTPS*, 2003.

[44] Moustafa Zohidi. Anwendung von Management- und Führungsstrategien in Peer-to-Peer-Netzwerken. Master's thesis, Technische Universität Darmstadt, 2008.