
Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Diploma Program in Computer Science

Diploma Thesis

Dynamics in Large-Scale Peer-to-Peer-Networks

submitted by

Thorsten Schneider

on 12th of October 2006

Supervisor: Prof. Dr. Gerhard Weikum

Advisors: Dipl.-Inf. Matthias Bender
Dipl.-Inf. Sebastian Michel

Reviewers: Prof. Dr. Gerhard Weikum
Prof. Dr. Christoph Koch

Databases and Information Systems Group
Max-Planck-Institute for Computer Science

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, 10/12/2006

First of all, I want to thank Matthias Bender and Sebastian Michel for their tremendous support. Their help and notes influenced this thesis considerably. Many thanks to my family and Carla Haid for bearing with my mood swings and supporting me all the time. A special thank to Kim Herzig who kept a lot of other work away from me. Last but not least, I want to thank Gerhard Weikum for making this diploma thesis possible.

Thank you all.

Abstract

Performing automated tests can help to identify errors with much less effort than testing complex programs manually. Setting up such tests on Peer-to-Peer networks is not an easy task because many machines have to be synchronized while peers should follow a join and leave pattern similar to the real-world behavior. This work develops real-world user behavior models and a simulation framework which is subsequently used to evaluate Minerva, a Peer-to-Peer Web search prototype system developed at MPI. The simulation framework is deployed on the MPI cluster to set up large-scale networks in a fully automated way. Measurements are conducted on the freshness and availability of data in Minerva and compared to theoretical forecasts that are calculated with help of the user behavior models. The experimental results show that the general system design is scalable and the implementation of Minerva is correct.

Zusammenfassung

Automatisiertes Testen von komplexer Software kann dabei helfen, Fehler mit deutlich geringerem Aufwand zu finden, als dies bei manuellen Tests der Fall wäre. Die Anwendung automatisierter Tests auf Peer-to-Peer Systeme ist allerdings schwierig, da viele Rechner aufeinander abgestimmt werden müssen und gleichzeitig das Verhalten der Peers der realen Welt nachempfunden sein sollte. Diese Arbeit entwickelt Modelle für ein realistisches Benutzerverhalten und ein Simulations-Framework, das anschließend zur Evaluation von Minerva, einem vom MPI entwickelten Prototypen zur Peer-to-Peer Web Suche, verwendet wird. Das fertige Framework wird auf dem MPI Cluster eingesetzt um große Netzwerke vollautomatisch aufzubauen. Auf diesen Netzwerken werden Messungen der Frische und Erreichbarkeit von Daten in Minerva durchgeführt und anschließend mit theoretischen Werten verglichen, die mit Hilfe der Benutzermodelle vorausgesagt wurden. Die experimentellen Ergebnisse zeigen, dass das generelle System-Design von Minerva skalierbar und die Implementierung korrekt ist.

Contents

1	Introduction	1
2	Peer-to-Peer in General	3
2.1	Unstructured Peer-to-Peer Systems	4
2.1.1	Centralized Peer-to-Peer Networks	5
2.1.2	Pure Peer-to-Peer Networks	5
2.1.3	Hybrid Peer-to-Peer Networks	7
2.2	Structured Peer-to-Peer Systems	7
3	Chord	9
4	Minerva	13
4.1	Why Peer-to-Peer Web Search	13
4.2	System Architecture	14
4.3	Related Work	16
5	Design of Experiments	17
5.1	Modeling User Behavior	18
5.2	Expected Test Results	20
5.2.1	Pre-requisites	20
5.2.2	Metadata Freshness	22
5.2.3	Metadata Availability	24
6	Simulation Framework	27
6.1	Design	27
6.2	Implementation	28
6.2.1	ProfilerApp	28
6.2.2	MainController	29
6.2.3	NodeController	33
6.2.4	Optimization	37
6.2.4.1	Increase of Network Size	37

6.2.4.2	Parallel Shutdown of Peers	39
6.2.5	Measurements	39
6.2.5.1	Chord Ring Check	39
6.2.5.2	Metadata Freshness	41
6.2.5.3	Metadata Availability	43
6.2.6	Plot Engine	43
7	Deployment on the Cluster	45
7.1	SSH	45
7.2	Grid Engine	47
7.3	Initialization of an Experiment	51
8	Experimental Results	55
8.1	Metadata Freshness	55
8.2	Metadata Availability	60
8.3	Discussion	64
9	Conclusion	65

List of Figures

2.1	Evolution of Peer-to-Peer systems over the years	4
3.1	Mapping keys onto peers in the Chord ring	10
3.2	Send a query through the ring	11
3.3	Send a query using the finger tables	12
4.1	Minerva System Architecture	15
5.1	Poisson distribution for $\lambda = 4$ and $\lambda = 6$	20
5.2	Exponential distribution for $\mu = 0.1$ and $\mu = 0.25$	21
5.3	Probability that a term is still in the directory peer in round r	24
6.1	Class Diagram of the <i>ProfilerApp</i> class	29
6.2	Class design of <i>ControlSequence</i> and <i>ControlAnswer</i> . . .	32
6.3	Class Diagram for the main Controlling tasks	34
6.4	NodeController	36
6.5	Class Diagram of the <i>PlotEngine</i> class	44
7.1	Start peers on remote machine using the startup perl script	46
7.2	MPI cluster with Grid Engine	48
7.3	Setup of the controlling environment	53
7.4	Starting a peer using the <i>NodeController</i>	54
8.1	Setup of the Metadata Freshness Test Environment . . .	56
8.2	Metadata Freshness - Experiment 1	58
8.3	Metadata Freshness - Experiment 2	59
8.4	Setup of the Metadata Availability Test Environment . .	60
8.5	Metadata Availability - Experiment 1	62
8.6	Metadata Availability - Experiment 2	63

List of Tables

3.1	Comparison of Peer-to-Peer Systems	12
8.1	Metadata Freshness - Experiment 1	58
8.2	Metadata Freshness - Experiment 2	59
8.3	Metadata Availability - Experiment 1	62
8.4	Metadata Availability - Experiment 2	63

Chapter 1

Introduction

The usage of Peer-to-Peer networks for downloading data from other users all over the world increased significantly since Napster or Gnutella. Better approaches have been made and several generations of Peer-to-Peer networks have been developed with the goal of a better scalability and stability. Initially becoming popular in the context of sharing music and films, Peer-to-Peer networks are used today in several other areas where information can be shared. Theoretically every application that needs or wants to distribute data over the network and also needs to find it again in an efficient way is predestinated to use a Peer-to-Peer network as underlying layer.

One of these projects is Minerva, a software developed at the chair of Prof. Weikum. It uses Peer-to-Peer structures to share whole databases full of information to provide a Web search service. In small networks of up to ten peers it is possible to manually check if the software does what it should do. But if you want to be sure that everything works fine in daily usage, for example in a network of about 500 or 1000 peers, non-interactive tests are needed to check this in a fully automated fashion.

To set up fully automated tests in such a large network there are several tasks to bother with. First of all, a framework has to be created which has an overview of the whole network and can manipulate the network in an automatic way (e.g. add new peers or shutdown peers after a while). On the other hand, the code that should be monitored and controlled has to be instrumented in a way that the normal behavior is not influenced, because that could falsify the test results.

This thesis deals with the problem of setting up such a suite in which the full range of a Peer-to-Peer network can be tested automatically using a cluster infrastructure. To mimic real-world behavior in the simulation, a model for the user behavior in the network is build and integrated in the test suite. This test suite is then used for evaluating the Minerva project in

different real user behavior models which are based on measurements of the user behavior in the Gnutella network.

The overall goal of the thesis is to show if Minerva and the underlying Peer-to-Peer network is scalable in the size of the network, i.e., the number of peers. In this context scalability means:

- stability of the underlying network
- availability of the shared information
- fault tolerance if, for example, one or more peers leave the network unexpectedly
- behavior under different user behavior (e.g. shortly staying peers against a network with less fluctuation)

The remainder of this thesis is organized as follows. Chapter 2 introduces the basic techniques of Peer-to-Peer networks over the last few years. In Chapter 3, the Peer-to-Peer network Chord, that is used by Minerva, is presented in more detail to give a better overview of the aspects that have to be controlled. Minerva itself and an overview of its Web search techniques is presented in Chapter 4. Chapter 5 contains the detailed explanation of the experiments and models that are planned to be implemented. In Chapter 6 the implementation is described while the deployment on the MPI cluster is shown in Chapter 7. The results of the experimental evaluation of Minerva are presented in Chapter 8 followed by a short conclusion in Chapter 9.

Chapter 2

Peer-to-Peer in General

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 [13] [16].

Peer-to-Peer systems have been known since the late 1960s when the ARPANET has been set up. The goal was to share resources over a few points in the U.S. But during the growth of the ARPANET to the today known internet this characteristic was lost. The commercialization of the internet necessitates techniques which work against the Peer-to-Peer nature of the initial network. Firewalls, NAT, and dynamic IP addresses eliminated more and more the possibility of connections between arbitrary peers in the huge internet.

The consequence of these changes was a network that has stuck to a client/server-model from the beginning of the 1980s (Fig. 2.1.a). The most popular client/server-applications in the internet were WWW, FTP and eMail. In those days this was enough for the standard internet user.

But in 1999 everything changed when Napster came into play. It was the first successful attempt of bringing the Peer-to-Peer model back on stage. But the reason of the tremendous popularity of Napster was not the new technique, but the availability of a huge mass of music, software and (later on) movies (and some other more or less legal stuff) within the network for free. Because of this historical reasons Peer-to-Peer networks still have the reputation of illegality.

One of the most important problem that the developers of Peer-to-Peer systems have to cope with is where to store and how to retrieve content in the network as there is no centralized directory service available. Designing these *lookup functions* makes it necessary to think about efficient ways

of storing and accessing information describing the location of the content, with respect to the scalability, reliability and robustness of the network.

Over the years there have been many approaches to this problem which can be classified into two main categories: *Unstructured Peer-to-Peer systems* and *Structured Peer-to-Peer systems*.

While the first implementations have not paid attention to any structure in the network (new peers are connected randomly depending on the points where they join), the latest Peer-to-Peer networks arrange the peers in order to set up a structure to improve the lookup mechanism.

Some of the architectures of the last years are shown in Figure 2.1 and will be discussed in more detail on the next pages.

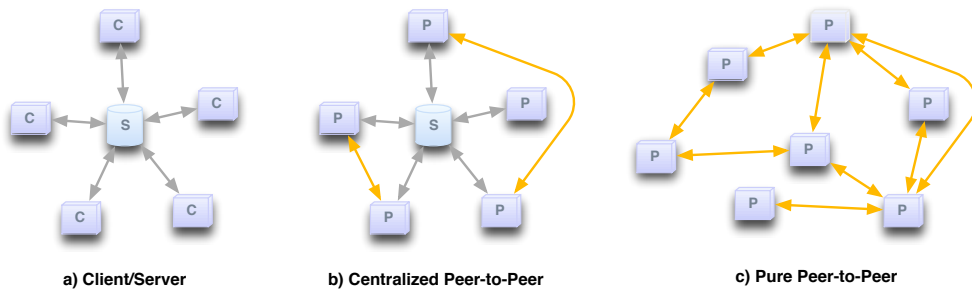
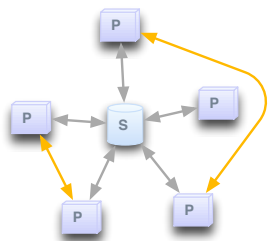


Figure 2.1: Evolution of Peer-to-Peer systems over the years

2.1 Unstructured Peer-to-Peer Systems

There are several approaches to a stable Peer-to-Peer system scheme. The early approaches (e.g., Gnutella) can be classified *Unstructured Peer-to-Peer systems* because the content stored on a given node and its position in the network (IP address) are unrelated and do not follow any specific structure. The most popular unstructured systems are shown below.

2.1.1 Centralized Peer-to-Peer Networks



The first popular Peer-to-Peer network (Napster) relied on a central lookup server. Contrary to the classical server, this lookup server only stores the IP addresses where some content can be found but not the content itself.

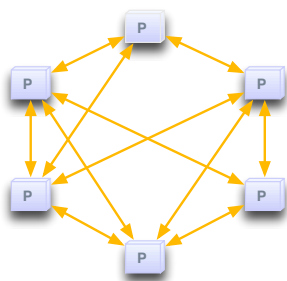
If a peer p_i joins the network, it connects to the server and sends information about the content it provides. If some peer p_j is looking for some content, it asks the server for the peer p_k that stores the content. After this information is retrieved by p_j , a direct connection to p_k is established to download the content.

This technique reduces the load on the server to just the search queries which is a significant improvement against the classical client/server-model. However many of the problems in the client/server-model remain in these so called *centralized Peer-to-Peer networks*:

- If a peer wants to join the network, it has to know in advance the IP address of the lookup server.
- If the server has a breakdown, the whole system is not functioning anymore (*single point-of-failure*).
- If the number of peers increase the server will eventually be overloaded. This means that the network is not scalable.

In fact, this system is not a real Peer-to-Peer system, because of its centralization. The central server knows all connected peers which means one "peer" is loaded with information of $O(N)$ other peers while the other peers do not know anything about other peers.

2.1.2 Pure Peer-to-Peer Networks



Because of the popularity of Napster and the resulting huge number of peers it was necessary to find a solution which is more robust against shutdown or crashes of the lookup server. Out of this necessity arose the first real decentralized Peer-to-Peer network, the *pure Peer-to-Peer network*, which was the base of the first Gnutella release (Version 0.4). The main innovation of this model was its full decentralization, which makes a central lookup-server needless.

If a peer wants to join the network, it connects to any peer that is currently a member of the network. Initially it could be necessary to download a list of currently available peers from a bootstrap server. While the peer spends time in the network it will cache a bunch of available peers, for example peers it set up a connection with to download something. This cached list can then be used to get into the network for the next session without using the bootstrap server again as long as one of the peers on the list is still available. As long as the peer stays in the network, the number of other nodes it knows to perform a lookup is limited to its neighbors. This number is constant ($O(1)$) and does not depend on the size of the network.

After the login process is finished, the new peer does two things. On the one hand, it explores the network in its vicinity and on the other hand it floods the network with information about its offered content (*signaling traffic*). The messages that get flooded over the network take (normally) a maximum of seven hops through the network which avoids infinite paths and loops. To control that, a TTL (Time-To-Live) is used which is decreased by every peer that gets the message. If the TTL is 0 the message is discarded by the peer, otherwise it is sent to all of the peer's neighbors.

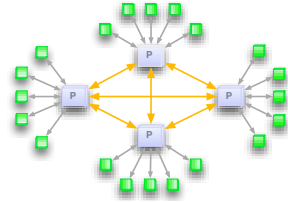
Because of these changes, the pure Peer-to-Peer networks gain a significant better stability, because there is no single point-of-failure anymore. If a peer fails, there are enough other paths available to hold the network together and to provide the lookup functionality.

However, the full decentralization has been paid off with a lot of traffic. Because there is no central lookup service, the information and the lookup queries are flooded over the network causing huge amount of traffic. In fact, the signaling traffic dominates the internet traffic in some cases even today.

This means that the scalability of pure Peer-to-Peer networks significantly depends on the available bandwidth and the number of peers in the network. When new peers enter the system, the number of messages can increase exponentially because of the flooding. If one imagines that in average every third message in the Gnutella network crosses the Atlantic several times because the physical network differs significantly from the overlay topology [16], it is easy to see that this system does not really scale.

Another disadvantage is the fact, that it cannot be guaranteed that all relevant nodes will be reached by a lookup query. This means that possibly not all content that is available in the network can actually be accessed. This happens in networks where the flooded message over seven hops does not reach the peers that store relevant content.

2.1.3 Hybrid Peer-to-Peer Networks



The huge amount of traffic in pure Peer-to-Peer networks made it necessary to revise the basic structure of this network type. The result was the *hybrid Peer-to-Peer network*, which has some hierarchical structure to reduce the traffic and was the base architecture for a new Gnutella release (Version 0.6).

A small amount of peers are connected as a pure Peer-to-Peer network. The peers in that network are called *supernodes* and stay in the network for a long time. Regular peers connect then directly to one of the supernodes. Because of the resulting tree-like structure, these regular peers are also called *leafnodes*. The supernodes store the information about content available on the connected peers together with their IP addresses. Thus, the superpeers are often able to answer incoming requests immediately by providing the respective IP address. In this way, the average number of hops that is required during the search process is reduced. As a direct consequence the signaling traffic is reduced, too.

But the other problems that already existed in the pure Peer-to-Peer networks still exist in the hybrid networks, for example that it cannot be guaranteed that all stored content in the network is reachable.

2.2 Structured Peer-to-Peer Systems

The problems inherent in unstructured networks regarding an increase of the network size, lead to efforts trying to find better solutions. The main task was to design a more scalable method for content localization.

The new approaches try to solve the main problems of unstructured networks:

- Reduce the traffic that is caused by the message flooding in pure and hybrid networks significantly.
- Guarantee that all content that is stored in the network is reachable by every connected peer.
- Distribute the information for answering search queries over the network, so that there is no bottleneck like in centralized networks.

Distributed Hash Tables (DHT) promise to be a suitable method for this purpose. In the realm of Peer-to-Peer systems, these approaches are also

often called *structured Peer-to-Peer systems* because of their structured and proactive procedures.

Distributed Hash Tables provide a global view of data distributed among many nodes, independent of the actual location. Thereby, location of data depends on the current DHT state and not intrinsically on the data. Because of this characteristic, a DHT has several advantages:

- Each DHT node (peer) manages a small number of key/value-pairs. This means each node is responsible for a subset of all keys.
- Queries are routed via a small number of nodes to the target node.
- The load for retrieving items is balanced nearly equally among all nodes.
- DHT's are considered to be very robust against random failures and attacks.
- A DHT guarantees that all data that it is stored in the system will be found.

Concerning the performance of the lookup function the DHT is a great improvement over unstructured Peer-to-Peer networks. In fact, the lookup function can be used very easily to implement a put/get-interface that provides the same functionality to the user as a centralized hash table.

Chord [17], the Peer-to-Peer system that has been used within this thesis, implements exactly this idea of Distributed Hash Tables and will be discussed in detail in the next chapter.

Chapter 3

Chord

The fundamental problem that confronts peer-to-peer applications is to efficiently locate the peer that stores a particular data item.

As described in 2.1.2 early systems like Gnutella rely on unstructured architectures in which a peer forwards messages to all known neighbors. Even though studies show that this message flooding works remarkably well in most cases, there are no guarantees that all relevant nodes will be reached. Additionally, many of the generated messages are unnecessary and prevent the network from being a highly scalable architecture.

In the architecture Napster used, there was no communication overhead, but the full workload was concentrated on one machine.

Chord [17] tries to solve all of these problems by providing the functionality of a distributed hash table (DHT). Like normal hash tables DHT's store key/value-pairs. Searching for the key returns the corresponding value. Because the entries of the hash table are not all stored on the local machine but all over the network, a method is needed that finds the correct entry in the system. This is done by the lookup function: given a key, it maps the key onto a node. Chord uses a variant of *consistent hashing* [11] to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys.

The nodes in Chord are arranged in a cyclic structure and every node in the cycle gets a unique ID out of a cyclic ID space (for example 0 to $2^{16} - 1$). The keys are mapped onto the same cyclic ID space.

Let p_i denote the peer with i . Assume that we have keys k_j , where j is also an ID of the ID space. Then k_j is assigned to its closest successor p_i in the ID space, such that $i \geq j$. This means that every node is responsible for every key between its own ID and the ID of its predecessor.

For example, consider Figure 3.1. There are ten nodes distributed across

the ID space. Key k_{54} is mapped onto node p_{56} as its closest successor node, and the keys k_{24} and k_{30} are both assigned to one node, namely p_{32} .

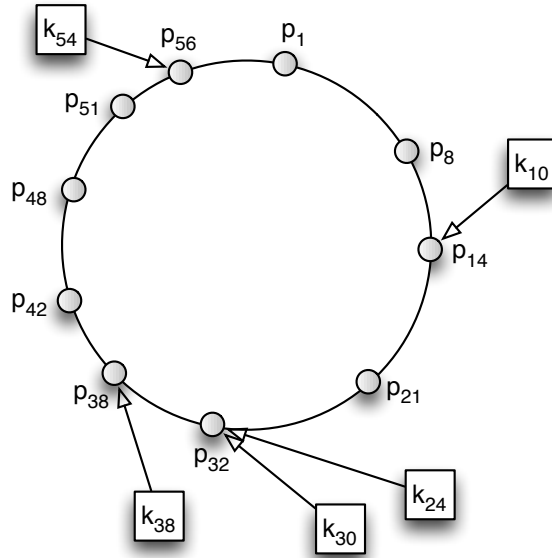


Figure 3.1: Mapping keys onto peers in the Chord ring

Now let us assume that the node p_8 sends a query to find the key k_{54} . As each node knows how to contact its successor there exists a very naive way to answer this query, illustrated in Figure 3.2. The query is passed around the circle until the responsible node has been found (in this case p_{56}).

Looking for the node in this way is the same as searching a linear list and has therefore an expected number of $O(N)$ hops to find the target node in the ring. Because the nodes just need to know about their successors at each node there is only $O(1)$ information required about other nodes.

To accelerate the lookup operation, each peer p_i stores a routing table called *finger table*. The setup of these finger tables is as follows: The m -th entry in the table of node p_i contains a pointer to the first node p_j that succeeds p_i by at least 2^{m-1} on the identifier circle. This means that every node stores $O(\log N)$ pointers to other nodes

This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Secondly, a node's finger table does not necessarily contain enough information to *directly* determine the node responsible for an arbitrary key

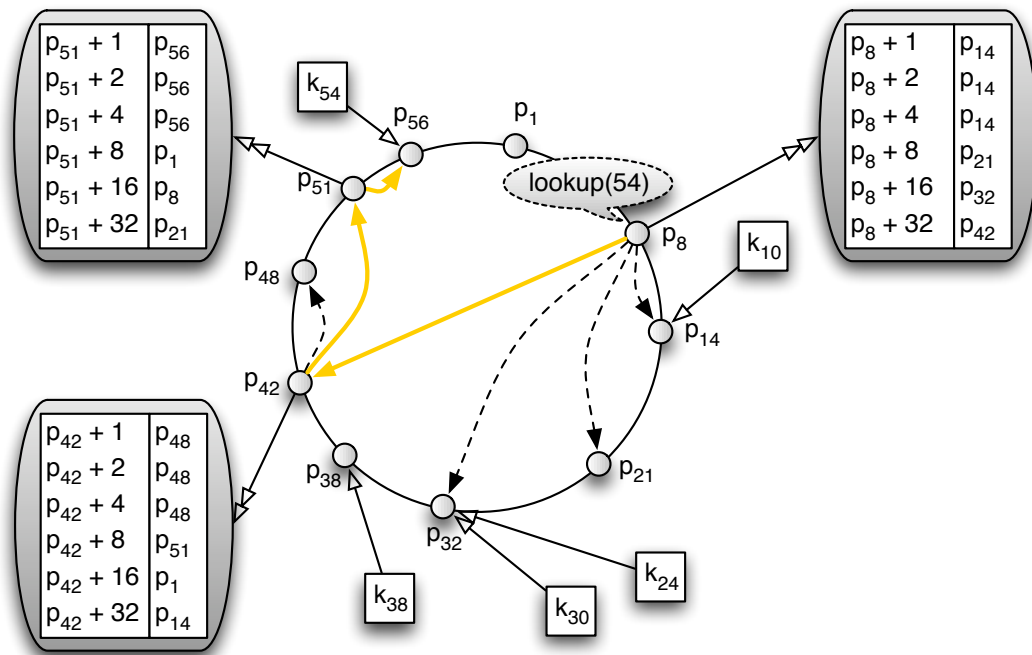


Figure 3.3: Send a query using the finger tables

Chord can provide lookup services for various applications, such as distributed file systems or cooperative mirroring. However, Chord by itself is not a search engine, as it only supports single-term exact-match queries and does not support any form of ranking.

	Communication Overhead	Node State
Napster	$O(1)$	$O(N)$
Gnutella	$\geq O(N^2)$	$O(1)$
Chord	$O(\log N)$	$O(\log N)$

Table 3.1: Comparison of Peer-to-Peer Systems

Chapter 4

Minerva

As presented before Chord implements an efficient lookup method to find keys in the distributed hash table which can be used to provide lookup services for various applications. Minerva is one of these applications that uses DHTs, such as Chord, in order to implement a Peer-to-Peer Web search service [8, 9]. Chord itself delivers the efficient lookup of single-term exact-match queries, while Minerva extends this functionality to enable for example multi-key queries and ranking.

4.1 Why Peer-to-Peer Web Search

One would wonder why a Peer-to-Peer based Web search should be necessary, given the good job Google does. The first and the most obvious reason is the fact the Peer-to-Peer approach facilitates the sharing of huge amounts of data in a distributed and self-organizing way. These characteristics offer enormous potential benefit for Peer-to-Peer based Web search, powerful in terms of scalability, efficiency, and resilience to failures and dynamics. On the other hand, the information, that is distributed over the network, is often user related and can be better sorted by relevance if a user sends a query. The third disadvantage of Google is that it is again a centralized service. If Google dies, all stored information is not accessible while in a Peer-to-Peer network the data normally is still reachable over another path through the network.

Additionally, such a search engine offers great opportunities for collaborative search and recommendations, benefiting from the intellectual input of a large community participating in the data sharing network.

Finally, but perhaps even more importantly, a Peer-to-Peer Web search engine can also facilitate pluralism in informing users about internet content,

which is crucial in order to preclude the formation of information-resource monopolies and the biased visibility of content from economically powerful sources.

4.2 System Architecture

This section gives a short introduction to the Minerva architecture. But it will be limited to the aspects that are relevant for this thesis. More detailed information about the project can be found in [8, 9].

Each peer is fully autonomous and has its own local search engine and a local index that can be build from own Web crawls or imported from external sources. The information that is stored on the peer reflects the user's thematic interest profile, and is therefore the first choice of information to search through. In fact, every peer can do a quick local index search and, if the results are unsatisfactory, extend the search over the network.

To let other peers benefit from the information locally stored, peers can share their local indexes (or specific fragments of local indexes) by posting metadata into a Peer-to-Peer network. This metadata contains compact statistics and quality of service information, and effectively forms a conceptually global (but physically distributed) directory. Minerva does *not* distribute index lists or even documents across the directory, but only metadata *describing* the peers' local indexes.

If the results obtained from its local index are unsatisfactory, a peer can use the directory to identify candidate peers that are most likely able to provide good query results and retrieve their local results.

Directory maintenance, query routing, and query processing work as follows. In a preliminary step (see Figure 4.1 - step 0), every peer publishes statistical information (*Posts*) about every term in its local index to the directory. Chord is used in order to determine the directory peer currently responsible for a term (e.g., P4 is responsible for term "b" in Figure 4.1). This peer maintains a *PeerList* of all Posts for this term from peers across the network. Posts contain contact information about the peer that posted it, and every Post is valid for a previously declared time-to-live interval (*TTL*). Within this interval the Post must be refreshed or the directory peer will drop it to deal with churn, i.e., peers leaving the system without prior notification. If, for example, P3 posts its term "b" into the directory, P4 (that is responsible for "b") will get a message that P3 stores some content concerning "b". But it will also get a TTL (e.g., 5 time intervals) for that Post. P3 then has to send a new Post within the next 5 time intervals, otherwise

P4 will erase it from its list.

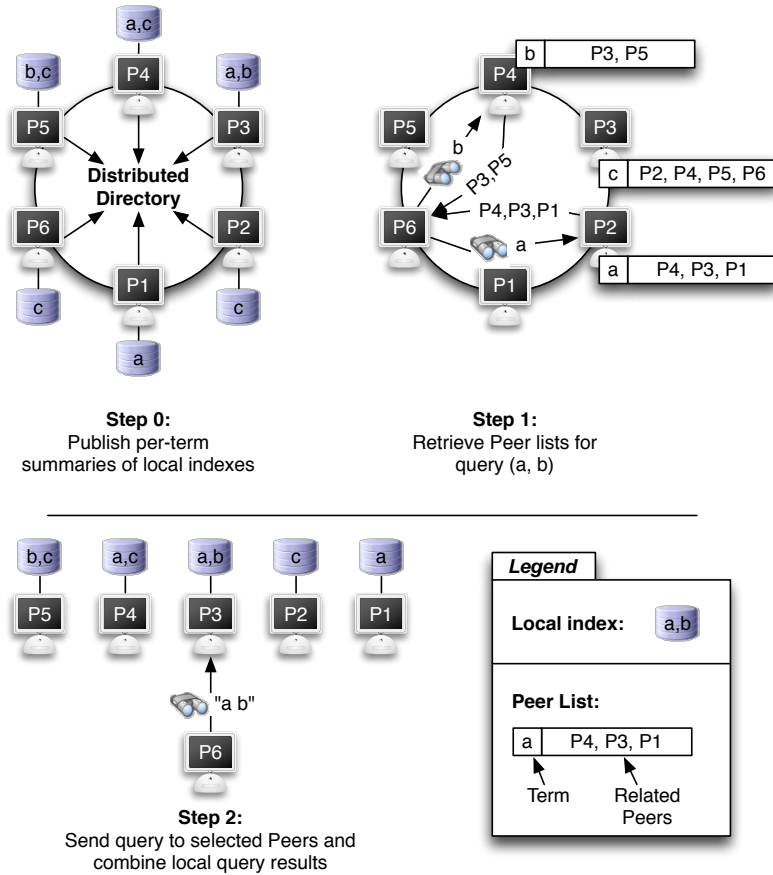


Figure 4.1: Minerva System Architecture

If a peer (e.g., P6) sends a query (here "a b") first of all the query is split into sub-queries if it is a multi-key query and every sub-query is then processed independent form each other. In Figure 4.1 (step 1) P4 is responsible for the term "b" and P2 for the term "a". As described in the chapter before, the peers do not necessary know the responsible peer directly. Therefore each query is routed through the network until it gets to the responsible peer using the lookup method of Chord. P4 in this case sends then the list of peers which store content related to the term "b" back to P6. P2 does the same for term "a".

P6 takes these results and merge them together (*query routing*) to determine auspicious peers. The full query ("a b") is then send to these peers. In this example P3 is the most promising peer, because it has term "a" and term "b" in its local index. So P6 sends its query to P3 (Figure 4.1 - step 2).

The results of this query request (in large networks normally more than one peer will be asked) are used to set up a priority list applying some clever combination strategies for *result merging*. In this way the user is presented an ordered list with the most relevant documents at the beginning. If the user goes for one of this proposal, P6 contacts the corresponding peers directly to download the content that is needed without any detour.

4.3 Related Work

Many other approaches have been proposed for Peer-to-Peer Web search. Some of them should be shortly addressed at this point.

- **Galanx** [19] is a Peer-to-Peer search engine implemented using the Apache HTTP server and BerkeleyDB. A site's Web servers are the peers of this architecture; pages are stored only where they originate from, thus forming an overlap-free network.
- **PlanetP** [10] is a publish-subscribe service for Peer-to-Peer communities, supporting content ranking search. The global index is replicated using a gossiping algorithm.
- **Odissea** [18] assumes a two-layered search engine architecture with a global index structure distributed over the nodes in the system. It actually advocates using a limited number of nodes, in the spirit of a server farm.

Chapter 5

Design of Experiments

As presented in the chapters above, a lot of effort has been spent to set up a Web search engine on top of a Peer-to-Peer network. To be sure the systems do what they should do, extensive testing is indispensable. To check basic functionality (e.g., the communication between the peers) just a few peers (about 10) are necessary, which can be controlled manually. But to be able to evaluate the scalability of the system and the availability of data it is necessary to create a network with far more peers (more than 100). This cannot be done manually, but must be automated.

Beyond this, it is very interesting to test these things also in a dynamic context. This means that the network should change over time. Peers join the network and leave it again after some time what means that the content the peer brought into the network leaves. In the following sections it is shown what this can do to the overall availability of data in the network.

The following experimental evaluations could be useful to assert the quality of a Peer-to-Peer system designed for large-scale deployment:

- Stability of the underlying routing infrastructure.
- Availability of metadata (are all peers listed in the directory peers still in the network?).
- Freshness of metadata (is all data in the network reachable according to the directory peers?).
- Resistance against churn.

A very important task that has to be done before setting up these experiments is the build of a model that can simulate real user behavior. In fact, different user behavior can significantly influence the evaluation of the network and, thus, should be addressed carefully.

The goal of this chapter is building models that predict the behavior of the system under some dynamic parameters. These forecasts can be used later to assert discrepancies by comparing them to the experimental results that are measured with the developed test suite.

5.1 Modeling User Behavior

Because the goal is to set up a large Peer-to-Peer network in an automatic way that should behave like human users, initially there is the need to find a model for the dynamic behavior of users in the network. So first of all, there is the question what could be dynamic in such a network. There are four dynamic components that influence the network:

- **Peers joining the network**

A joining peer influences the network significantly, because several tasks have to be processed before the peer is fully integrated in the network. First of all it has to be sorted into the Chord ring, and possibly take over some directory responsibilities from the peer that is its successor from now on. On the other hand the new peer needs to disseminate its metadata to the directory.

- **Nodes leaving the network**

If a node leaves the network the strength of the influence depends on the information the node shared before and how much information it stored as directory peer. If the node stores a large number of Posts, these Posts are temporarily lost. This means that the content that is related to these Posts can not be found by other peers, as long as the corresponding peers do not send a repost.

- **The life time of a broadcasted term (TTL)**

Because of the fact peers leave the network without prior notice, directory peers can have references to peers that are not alive anymore. To avoid those dead ends a Post is erased from the directory if and only if the directory peer receives no refresh signal during the TTL.

- **The refresh interval**

As mentioned before, the directory peers need a refresh signal for a term if the term should not be removed after a while. But there is another effect as well. If a directory peer that is responsible for a special Post leaves the network, this Post is no more available as long as the corresponding peer sends a *repost*. So it makes sense to have

different values for the TTL and the repost interval. We will see that some mechanisms in the network are highly influenced by the setting of this parameter later in this chapter.

Very interesting for modeling the real world are peers that join the network and peers that leave the network because these are the parameters that are noncontrollable after a network has been released to the wilderness.

There have been studies on real world systems like Napster and Gnutella to find out how actual peers behave [15]. In these studies, the network traffic and some user related information have been collected. During the evaluation of that data the researchers found out that two mathematical models come very close to the real user behaviour [14, 12] and, therefore, are well-suited to model user behavior for automated testing:

- **Poisson distribution simulates joins**

If one knows the average number (λ) of peers joining the network per time interval, the probability that in the next time interval n peers join the network follows a Poisson distribution:

$$P_{\lambda}(n) = \frac{\lambda^n}{n!} e^{-\lambda} \quad (5.1)$$

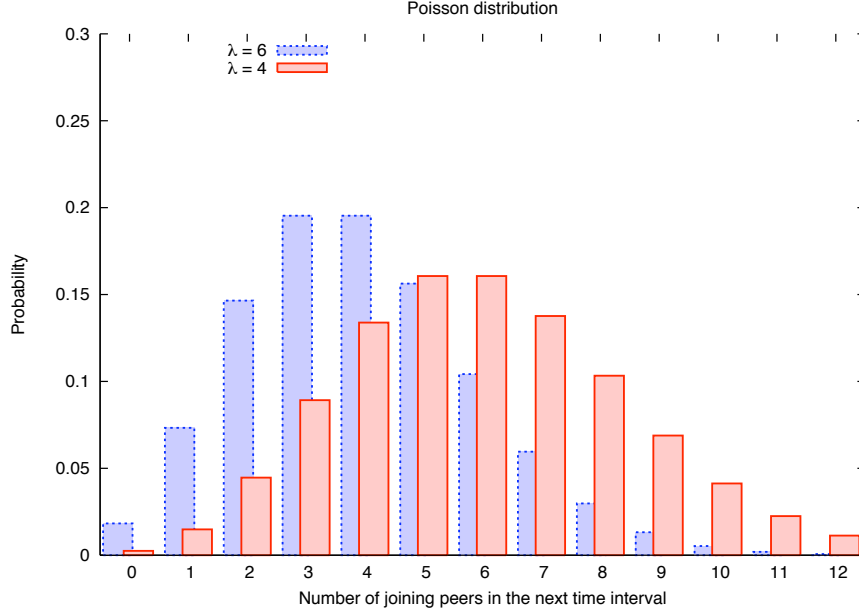
In Figure 5.1 the Poisson distributions for parameters 4 and 6 are illustrated. If, for example, the average number of peers joining the network per time interval is 6, the probability that in the next round another 6 peers join the system is 0.16. The possibility of 3 joins is 0.09, of 8 joins 0.10. More information about the Poisson distribution can be found in [7].

- **Exponential distribution simulates leaves**

The parameter μ gives the average number of drop outs in one time interval depending on the number of running peers. If μ is for example 0.01, 1 percent of the running peers leave the network in this round. This means that the exponential distribution gives the probability that a peer leaves the network after a certain time interval. The distribution can be described by the following formula:

$$F(x) = 1 - e^{-\mu x} \quad (5.2)$$

In Figure 5.2 the exponential distributions for parameters 0.1 and 0.25 are illustrated. If, for example, 10% of currently running peers leave the network per time interval on average, the probability that a freshly started peer leaves in round 5 is 0.39, in round 10 it is 0.63, and in round 15 it is 0.78. More detailed information about the exponential distribution and its characteristics can be found in [1].

Figure 5.1: Poisson distribution for $\lambda = 4$ and $\lambda = 6$

5.2 Expected Test Results

5.2.1 Pre-requisites

First of all it is necessary to predict how many peers are running in a given round depending on the join and leave parameters in order to evaluate discrepancies when testing the system. This is achieved by the following recursive formula:

$$F(j, l, r) = \begin{cases} j + F(j, l, r - 1) \cdot (1 - l) & : r > 0 \\ 0 & : r \leq 0 \end{cases} \quad (5.3)$$

where j is the number of peers that join the network each time interval on average, l is the percentage of peers that leave the network on average depending on the number of running peers one time interval ago, and r is the time interval that is the point of interest.

Every time interval, j peers join on average and l percent of the current running peers leave on average. That means $1 - l$ percent of the currently

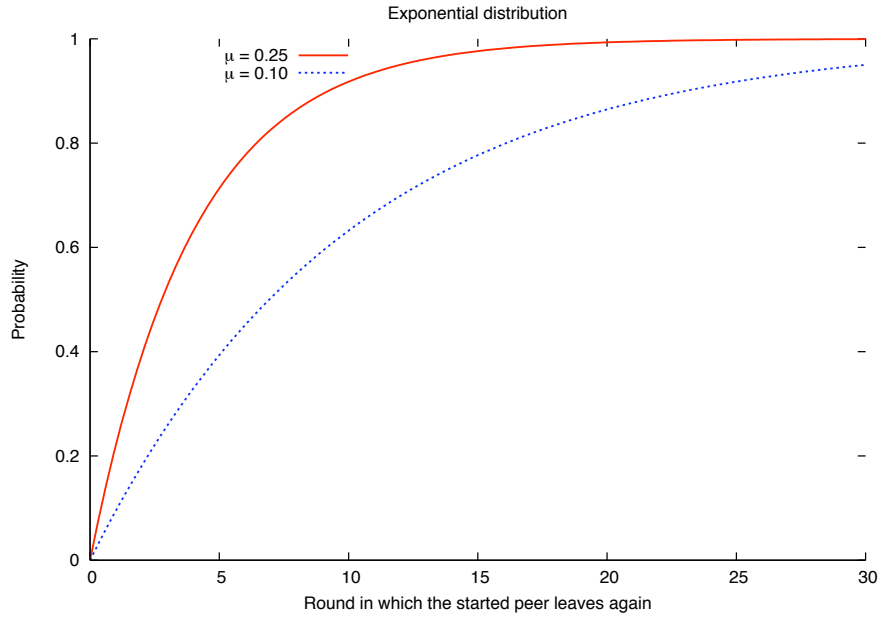


Figure 5.2: Exponential distribution for $\mu = 0.1$ and $\mu = 0.25$

running peers stay in the network for this time interval while the new peers join.

To get rid of the recursivity, the following closed formula can be used:

$$F(j, l, r) = \begin{cases} j \cdot \frac{1-(1-l)^r}{l} & : r > 0 \\ 0 & : r \leq 0 \end{cases} \quad (5.4)$$

Proof. The equality of the equations 5.3 and 5.4 can be shown by induction:

1. Basis: For 0 is trivial. For 1 is as follows:

$$\begin{aligned} F(j, l, 1) &= j + F(j, l, 0) \cdot (1 - l) \\ &= j \end{aligned}$$

$$\begin{aligned}
&= j \cdot \frac{l}{l} \\
&= j \cdot \frac{1 - 1 + l}{l} \\
&= j \cdot \frac{1 - (1 - l)^1}{l}
\end{aligned}$$

2. Inductive hypothesis: The formula is correct for some $n \in \mathbb{N}$

3. Inductive step:

$$\begin{aligned}
F(j, l, n + 1) &= j + F(j, l, n) \cdot (1 - l) \\
&\stackrel{(IH)}{=} j + j \cdot \frac{1 - (1 - l)^n}{l} \cdot (1 - l) \\
&= j \cdot \left(1 + \frac{1 - (1 - l)^n}{l} \cdot (1 - l) \right) \\
&= j \cdot \left(1 + \frac{(1 - l) - (1 - l)^{n+1}}{l} \right) \\
&= j \cdot \left(1 + \frac{1}{l} - \frac{l}{l} - \frac{(1 - l)^{n+1}}{l} \right) \\
&= j \cdot \frac{1 - (1 - l)^{n+1}}{l}
\end{aligned}$$

□

As it is now possible to predict how many peers are running in a given round, it is also possible to forecast how many peers leave in a given round:

$$\text{leaving peers in round } r: F(j, l, r - 1) * l \quad (5.5)$$

With these equations it is now possible to set up two interesting measurement studies that will be used during the practical part of this thesis.

5.2.2 Metadata Freshness

One common scenario is that peers leave the network without explicit notice, especially don't remove their metadata from the distributed directory. That means it is possible the network "thinks" the information is available but because of the missing peer the information is no longer accessible. So the first equation deals with the question how many of the peers that should be there (according to the directory peers), are really available. This can

be calculated as follows (for simplicity reasons every peer posts exactly one term):

$$G(j, l, r, t, rp) = \frac{F(j, l, r)}{F(j, l, r) + \underbrace{\sum_{i=r-t}^r F(j, l, i-1) \cdot l \cdot \min(1, \frac{i-r+t}{rp})}_b} \quad (5.6)$$

where j is the join rate, l the leave rate, r the round for which we want the availability, t the time-to-live for the terms (TTL) and rp the time after which a refresh is send to the directory

Equation 5.6 gives the average availability of the terms for the specified round r . To better understand the formula it is a good idea to split it up and look at the parts separately.

The nominator represents all peers that are really up and running, as shown in equations 5.3 and 5.4.

The denominator represents all peers that the system thinks that are running. These are all peers that are really running plus the peers that left the network before, but whose information has not yet been erased from the directory. So the interesting part is how to determine the number of those "ghost-peers". This is encapsulated in the sum on the right hand side.

The point-of-interest are peers that leave between round $r-t$ and r (now). The metadata of all peers that left before that time is no longer stored in the directory because of the TTL. All other peers *could* still be listed in the directory depending on the time of their last repost. That is described by part b of equation 5.6.

Because of the fact that a term is reposted every rp rounds, there exists a probability of $\frac{1}{rp}$ that a term is reposted in a certain round, if there is no knowledge about other rounds. Looking at a range of rounds, the accumulated probability that a term has been reposted in the meantime raises from one round to the other by $\frac{1}{rp}$ (see Figure 5.3). If, for example, a peer leaves in round $r-t+3$ it could have reposted within the the last t rounds (seen from round r) with a probability of $\frac{3}{rp}$ what means that the post is still listed in round r with the same probability. Under these conditions it is clear that peers that left after the round $r-t+rp$ are definitively listed in the directories in round r , because they have reposted the terms within the last t rounds with a probability of 1, what means that the TTL of the term has not yet been expired.

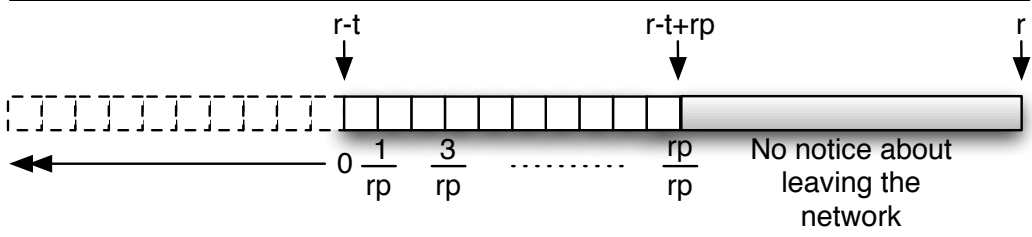


Figure 5.3: Probability that a term is still in the directory peer in round r

5.2.3 Metadata Availability

Another situation that can come up is that a directory peer leaves the network and takes all the information about terms and peers along out of the network i.e., its share of the distributed metadata directory. Until the next refresh, those terms are not available, because no directory knows about them and therefore no other peer can find the corresponding peers.

$$\begin{aligned}
 & \sum_{i=r-rp}^r \underbrace{F(j, l, i-1) \cdot l \cdot \frac{tpp}{F(j, l, i-1)} \cdot F(j, l, i)}_a \cdot \underbrace{\frac{i-r+rp}{rp}}_b \quad (5.7) \\
 &= \sum_{i=r-rp}^r l \cdot tpp \cdot F(j, l, i) \cdot \frac{i-r+rp}{rp}
 \end{aligned}$$

Equation 5.7 predicts the number of terms that are unavailable in round r . The calculation is very similar to the one before. Again the last few rounds are examined. In this case the last rp rounds are of interest, because every term that was lost before $r - rp$ definitively has been reposted until round r and is available again. Every term that gets lost within this time interval has a certain probability that it is reposted until round r .

To gain a better understanding of that, we first take a look at the subterm a of the formula. For simplification it is assumed that the hash-value of the terms are uniformly distributed and every peer posts exactly the same terms. tpp is the number of terms each peer posts. Under these conditions the subterm $\frac{tpp}{F(j, l, i-1)}$ gives the number of terms a directory peer is responsible for in round $i-1$. This especially means that if one directory peer leaves the network in round $i-1$ the number of terms a directory peer is responsible for multiplied with the number of running peers results in the number of terms that are no longer available in round i . Because more than one directory

peer can leave the network in one round this number has to be multiplied with the number of leaving peers ($F(j, l, i - 1) \cdot l$). Altogether the subterm 5.7.a predicts the actual number of terms that are lost in round i .

To consider terms that are reposted again before round r , another factor is added (Equation 5.7.b). As mentioned above, the probability that a term is reposted in one certain round is $\frac{1}{rp}$. So if the directory peer had left one round ago, the probability that a repost happened since then is $\frac{1}{rp}$, two rounds ago $\frac{2}{rp}$ and so on until rp rounds ago the probability of a repost is 1. Because in this case, the number of terms that have not been reposted is the point-of-interest, the multiplier is 0 in round $r - rp$ and raises in $\frac{1}{rp}$ steps to 1 until round r .

Summing up these values from round $r - rp$ up to round r it exactly gives the number of terms that are not reachable in round r .

To be able to say how many terms are available in proportion to the terms that are really in the network, Equation 5.7 is divided by the actual number of terms provided by the running peers. This result is then subtracted from 1 to calculate not the missing but the available terms:

$$H(j, l, r, rp) = 1 - \frac{\sum_{i=r-rp}^r l \cdot tpp \cdot F(j, l, i) \cdot \frac{i-r+rp}{rp}}{F(j, l, r) \cdot tpp} \quad (5.8)$$

$$= 1 - \frac{l \cdot \sum_{i=r-rp}^r F(j, l, i) \cdot \frac{i-r+rp}{rp}}{F(j, l, r)} \quad (5.9)$$

This formula predicts the behavior of the basic Minerva functions. This means that in a real network the ratio could be improved by using some more advanced functions, e.g, the so-called *replication* that set more than one directory peer responsible for one term.

Chapter 6

Simulation Framework

With the approaches from the last chapter, it is possible to predict the behavior of the network. The next step is to set up a real network and do measurements to check if these expectations are fulfilled by the network if it runs within a controlled testbed. This step is discussed in detail in the following sections.

6.1 Design

To be able to do measurements on a large-scale network, a system has to be developed that can cope with all aspects of setting up, control and trigger the network but do not influence the core of the implementation that should be tested. Therefore within this thesis a round based discrete event simulator is developed, that can automatically control and monitor the network. According to the approaches in the last chapter this simulator can especially start and stop peers on basis of the user behavior models.

On the monitoring part the simulator is able to check the stability of the routing infrastructure (the Chord ring) and measure the availability of metadata and the freshness of the corresponding directories.

The goal is to produce the above mentioned mechanisms working in an automatic but customizable way. To be able to compare the results from several test runs and to influence the workload on the network the experiments are set up in a round based environment. In every round there are three (during measurement four) steps that have to be processed:

1. **New peers join the network**

As shown before the number of peers that join the network follows the poisson distribution (see Equation 5.1). That means that in every

round a number of peers (depending on the parameter for the distribution) have to be started and connected to the network automatically.

2. **Every peer gets a message that a new round begins**

It is important that the peers realize that a new round has been started because of term aging. Depending on the values for TTL and repost times the peers have to do some administrative things. If the TTL of a term has expired the corresponding directory peer has to erase the term out of its directory. If the repost time has expired, the corresponding peer has to send a repost message into the network.

3. **Peers leave the network**

When a peer starts up it is determined when it should leave again according to the exponential distribution that has been described before. There has to be a functionality to tell the peer that it should leave the network in a particular round. Again this should take place automatically.

(4. **Measurement tasks**)

As we have seen before there are two main things we want to measure. But it is also possible to let the system run without any measurement. If the measurement mode is activated some commands are executed in this phase of the round. For example one can ask every peer for its directory to check which terms are available. But we will have a more detailed look at this in a later chapter.

6.2 Implementation

During the implementation several classes and helper classes have been written. The ones that build the core of the system and make the most of the work will be presented in detail in the following sections.

6.2.1 ProfilerApp

The *ProfilerApp* class is just a container for the peer itself. The only task of the *ProfilerApp* class is to setup the peer, initialize it with reasonable values and tell the MainController the peer is up and running.

Although it seems the *ProfilerApp* does not do much, it encapsulates the very basic settings that a peer needs to startup correctly. Using the ProfilerApp makes it very easy to create a new peer because it takes the values that are necessary for the peer's first initialization and sets them at

the correct point in the peer itself and the helper objects of the peer (e.g., the event handler for incoming network traffic). Besides that, introducing this class makes it possible to leave the *Peer* class implementation of Minerva untouched.

Starting a peer using the *ProfilerApp* works as follows. First of all a new empty *ProfilerApp* object is created. After that the *setup()* method can be used to set the variables the peer should be initialized with. The next step depends on the initial work the peer should do. If a new chord ring should be created the *ProfilerApp*'s *createChordRing()* method must be called, otherwise the *joinChordRing()* method must be called with the port and hostname of a peer that this peer should connect to.

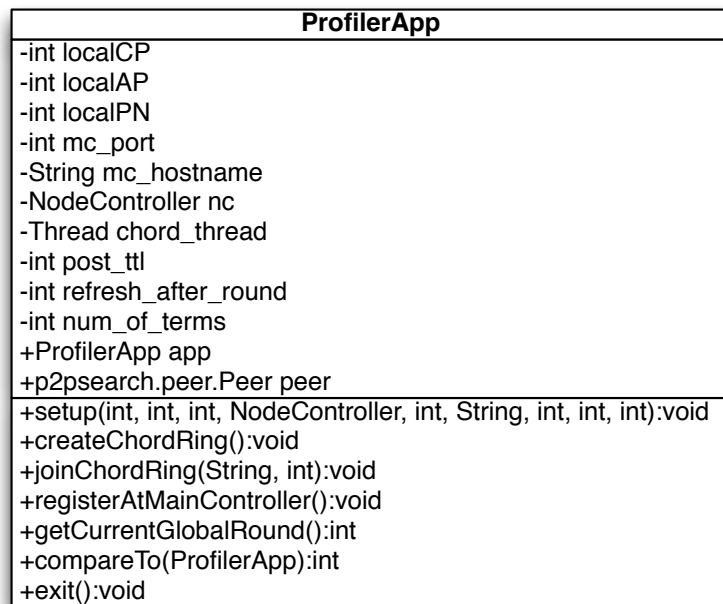


Figure 6.1: Class Diagram of the *ProfilerApp* class

6.2.2 MainController

As the goal of this thesis is making measurements with a huge number of peers, it is necessary to implement a kind of superuser that has the full control over the network and has knowledge of all peers that are running. This user must also coordinate and automatically initiate all events in the network that are normally performed by normal users (e.g., start and shutdown of peers).

All these tasks are encapsulated in one class. Because this is the heart of the implementation this class is called *MainController*.

The *MainController* is the first piece of the system that is started. In the initial implementation of this class the first round (as described above) starts immediately with the initialization of new peers. This is done by creating a *ProfilerApp* object for every peer that should be started. To keep track of the running peers, the *MainController* uses the helper class *Instance*. For every peer that starts, one *Instance* object is created, holding information about the running peer. It seems that this is redundant because the *ProfilerApp* itself holds this information. If all peers are started on the same machine as the *MainController* this is correct. But it is indispensable to distribute the peers over several physical machines and distinct Java Virtual Machines for the intended large-scale experiments. In this case the *ProfilerApp* is no longer directly accessible for the *MainController*. The distribution of the peers will be described in detail in the next chapter.

As mentioned before the *MainController* creates a certain number of peers at the beginning of each round. This number is calculated by using the poisson distribution described in Section 5.1. For every peer that is started, it is also calculated in which round the peer has to leave the network again using the exponential distribution also described in Section 5.1. This information is also stored in the corresponding *Instance* object, so that the *MainController* can quickly determine which peers must be shut down in a certain round. The calculations are done in two helper classes: *PeerLeave* and *PeerJoins*. Objects of these classes are created with the parameters also described in Section 5.1. After the initialization the *getValue()* method returns the number of peers that should be started in the next round (*PeerJoins*) or the number of rounds the peer stays in the network (*PeerLeave*).

Some of the tasks that have to be done by the *MainController* need a network connection to a peer. To be able to talk to each peer over the network, the *MainController* uses the helper classes *ControlSequence*, *ControlAnswer*, and *ControlCommunicator*. On the side of the peer, the *SocketBasedEventHandler* must be extended to react to the requests coming from the *MainController*. Because a peer already has the functionality of receiving network messages, there is just this class that has to be extended by the new messages it should handle.

- **SocketBasedEventHandler**

The *SocketBasedEventHandler* is extended by a new message type (*CONTROL_REQUEST*). By doing so the implementation of Minerva or Chord is untouched. The commands that are received (encapsulated

in a request) also use already implemented methods in the peer itself.

- **ControlSequence** (see Figure 6.2)

Objects of this class are sent over the network to the peers (in fact it is also used to send messages to the *NodeController* and the *MainController*). Every object encapsulates a command that should be executed on the peer's side and, if necessary, some options related to the command. In fact the sent command is one of the known commands in the peer's event handler (otherwise nothing happens). This means the number of commands can easily be increased by adding them to the *SocketBasedEventHandler* (or the *Main-/NodeControlEventHandler* respectively).

- **ControlAnswer** (see Figure 6.2)

Because some of the commands that are sent to the peer require an answer, the event handler can send back an object of the *ControlAnswer* class. This container can store various data types. The *ANSWER_TYPE* is used to remember what kind of data type is stored.

- **ControlCommunicator**

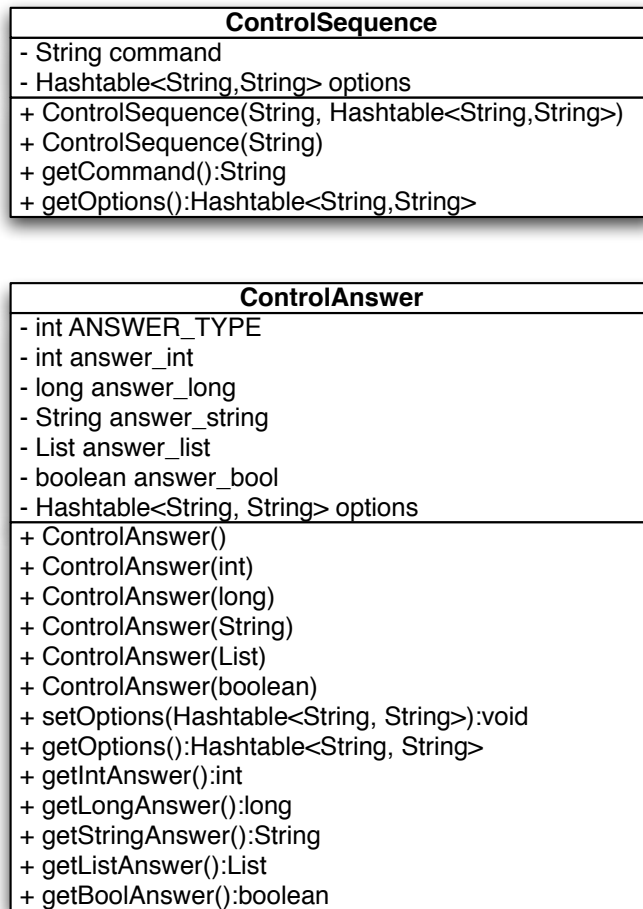
Every *Instance* object holds an instance of the *ControlCommunicator* class. The *ControlCommunicator* is a simple helper class that initially opens a connection to the corresponding peer and sends on demand a *ControlSequence* object as a *CONTROL_REQUEST* message to the event handler of the peer. The command that is encapsulated in the object is executed by the event handler and the result of the execution is send back encapsulated in a *ControlAnswer* object.

Because the class implements three methods (*sendMainRequest(Object)*, *sendNodeRequest(Object)* and *sendInstanceRequest(Object)*), it can be used universally for all network communication that is necessary in the scope of this thesis.

These classes cover the whole network communication to control the running peers.

Because it turned out during the first measurement tests that it is not very scalable to always control every peer individually over the network another class is introduced. The so-called *NodeController*. How this class works is described in detail in the next section. But it has to be mentioned here, because it makes some extensions to the *MainController* necessary.

As shown in the next chapter it is necessary that the *NodeController* registers at the *MainController*. This means the *MainController* no longer just sends messages over the network, but also needs the ability to receive

Figure 6.2: Class design of *ControlSequence* and *ControlAnswer*

messages. Therefore, a new helper class is introduced (*MainControlEventHandler*), that is in fact a reduced copy of the peer's *SockedBasedEventHandler*. It only knows a new message type (*MAIN_REQUEST*) and just one command (*register*). The *MainController* has to wait for all instances of the *NodeController* class to register before the first round can be started. Thus, there is a loop added before the simulation starts, that is left if and only if all *NodeController* instances are up and running.

The last helper class that should be introduced at this point is the *Machine* class. This class is used during the distribution of peers on several machines, which makes sense in a network with many peers. For every machine that can be used to start peers one instance of the *Machine* class is created. Beside the network information, like the host name, it stores how

many peers are already running on that machine, which is interesting for load balancing.

The last thing the *MainController* has to cope with is the shutdown of peers as soon as their lifetime is elapsed. To determine the peers that have to leave next, *Instance* objects are stored in a list, that can be sorted by the round in which the corresponding peer will leave. The peers at the beginning are the ones that will leave before the other ones. After the sorting, the *MainController* just have to send a **shutdown** message to the first entry of the list, if the round it should leave is equal to the current round. After that the entry is dropped. Because more that one peer can leave in one round, this is repeated as long as the first peer in the list should leave in a later round. If the shutdown message has been sent, the peer receiving the message does the rest. It tells the *NodeController* that it will leave now and kill itself afterwards.

The relationships between the *MainController* and its helper classes are illustrated in Figure 6.3

6.2.3 NodeController

As stated before, controlling all peers from the *MainController* sometimes does not scale very well (e.g., during the check of the Chord ring stability, which is explained in Section 6.2.5). The *NodeController* is introduced to improve scalability for these critical tasks, to make the distribution of peers over several machines easier, and for some cluster-specific reasons, stated in the next chapter.

The *NodeController* can be seen as a kind of container for peers. The initialization of the *ProfilerApp* objects is no longer done by the *MainController*. Instead, the *MainController* sends a network message to the *NodeController* a peer should be started on. If a peer is started, the *NodeController* acts in the same way as the *MainController* before. It also stores a list of *ProfilerApp* instances which have been started.

In this way the *NodeController* can be used to distribute peers over several machines by just starting one *NodeController* on every machine that should be used and tell the *MainController* that it can use this machine. For some optimization reasons it is a good idea to tell the *MainController* just the number of *NodeController* instances that can be used during this run. If the *MainController* is successfully started, the *NodeController* instances are started on the corresponding machines and register themselves over the network at the *MainController*. As answer to their registration the *NodeController* instances get a range of Chord IDs for which they are responsible. Peers with a Chord ID in this range will be started on the responsible *Node-*

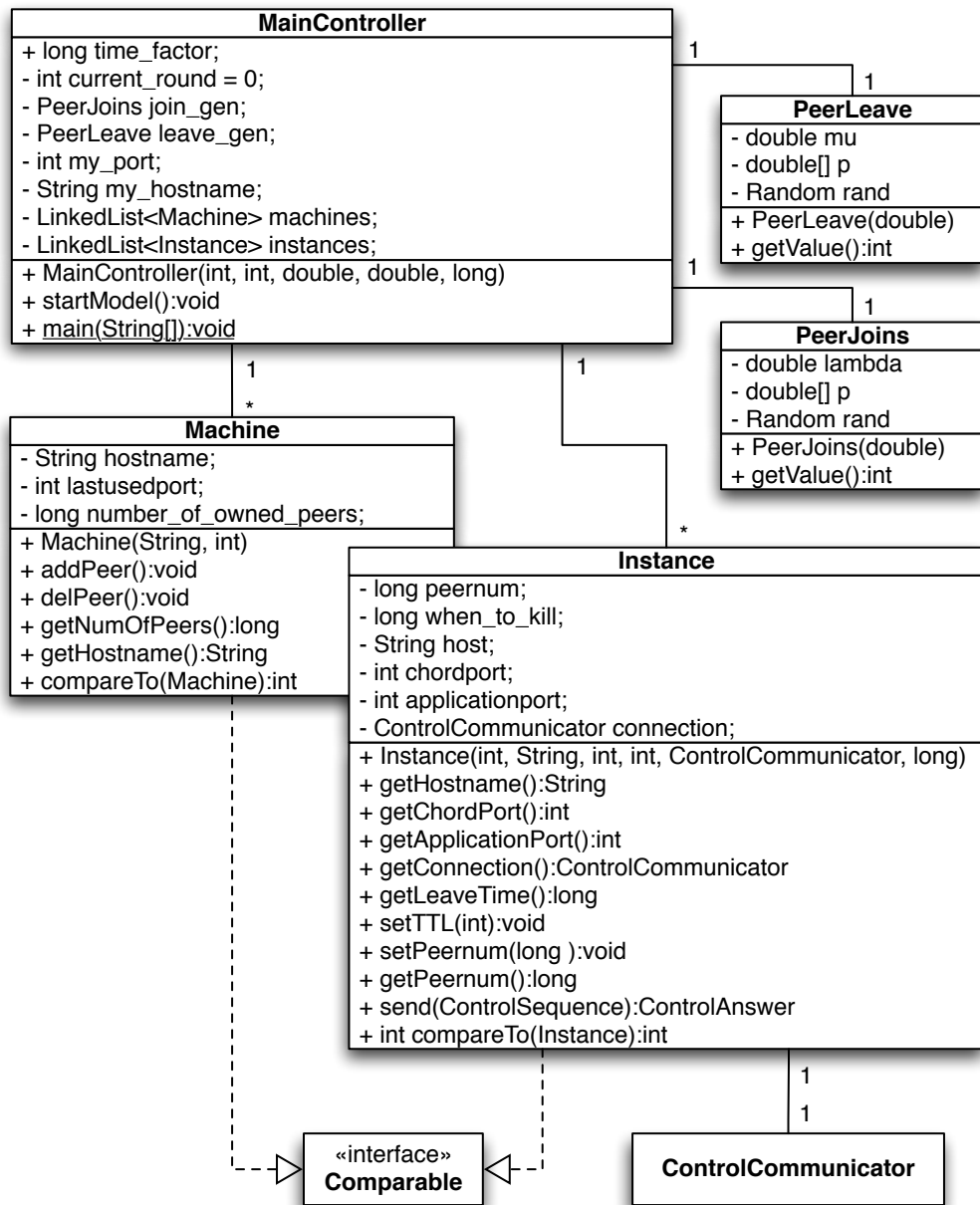


Figure 6.3: Class Diagram for the main Controlling tasks

Controller. This means that peers with just a short range between their Chord IDs are very probable on the same machine. The network communication between these peers will be much faster and the network workload is decreased.

If all *NodeController* instances have registered, the first round of the simulation is started. As told before, every peer has to be informed about a new round, because of the term aging. With such a hierarchy this can be done much faster and much more efficient than telling every peer about that, because just a fraction of the network messages has to be send. The *MainController* sends a message to all *NodeController* instances. The *NodeController* itself gives this message to all of the peers it started before. Because the *NodeController* and the peers are running in the same virtual machine, this can be done very efficiently by just calling the corresponding method in the peer's instance without any additional network traffic.

To provide the possibility of a network connection, the *NodeController* uses a helper class called *NodeControlEventHandler*. An instance of this class is created on startup of the controller and runs in an own thread. In this way the *NodeController* itself is never blocked for having a look on the network events (e.g., incoming requests). The event handler itself looks for a free port on and the hostname of the machine. As soon as the event handler is up, the *NodeController* sends a register message to the *MainController* including the information determined by the event handler, so that the *MainController* knows how to connect.

The class diagram of the *NodeController* and the *NodeControlEventHandler* is shown in Figure 6.4. Except for the startup phase the *NodeController* does nothing without a command from the *MainController*. In the current implementation the *NodeController* knows three commands that can be sent by the *MainController*:

- **nextround:** This command is used to tell every peer in an efficient way that a new round is going to start. The *MainController* sends this command to each *NodeController*, which tells all peers it is responsible for about the new round. This last step is not done via the network, but in a direct method call. This is possible because the peers and the controller are running in the same virtual machine.

Technically, the command is received by the event handler which calls the *nextRound()* method of the *NodeController* instance.

- **getpeerids:** With this command, the *NodeController* is told to give a list of the IDs of all peers that the controller is responsible for. This command is used for example to check the Chord ring stability.

The event handler also just calls a method of the controller, like before: *allPeersInContainer()*.

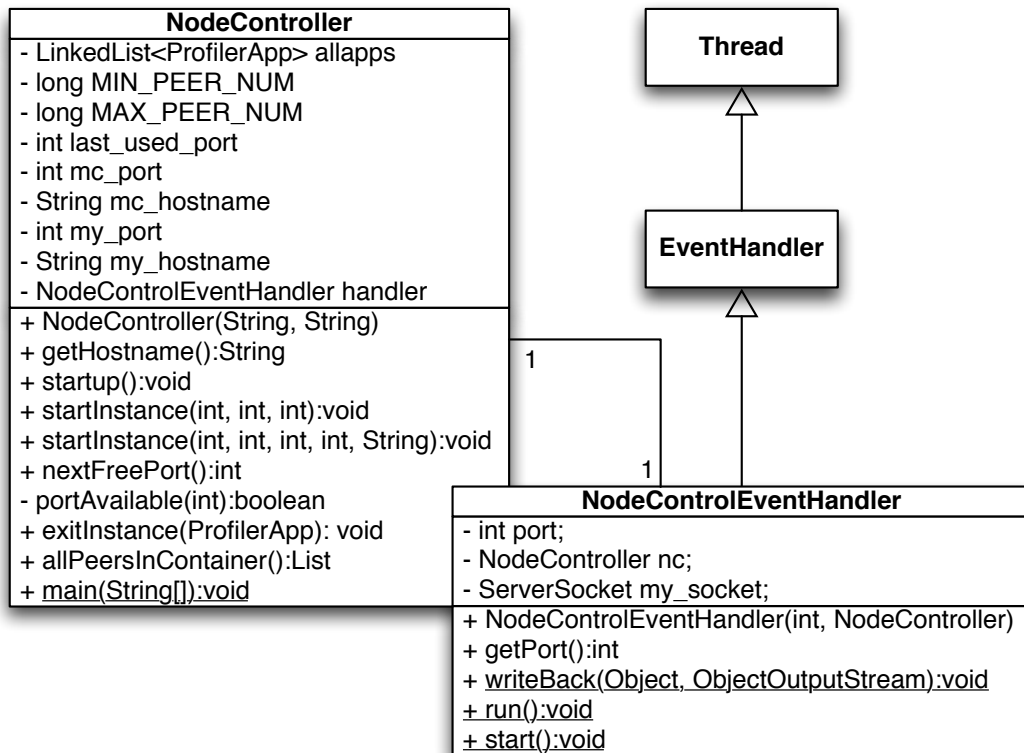


Figure 6.4: NodeController

- **startinstance:** This command is used to give the *MainController* the ability to start new peers using the *NodeController*. With this command some options come along, for example if the new peer should join or create a network. These options are used to determine which method of the controller should be called and which parameters to give to this method. Either
 - *startInstance(ID, cp, ap, rp, rh, terms)* for peers that should join to an existing network or
 - *startInstance(ID, cp, ap, terms)* for peers that should create a new network

is called, where *ID* is the Chord ID the peer should have, *cp* the port on which the peer should listen for Chord commands, *ap* the port on which the peer should listen for Minerva and controlling messages, *rp* the Chord port and *rh* the host name of the peer the new peer should

join at, and *terms* the number of pseudo-randomly generated terms the peer should post.

With the technique that the *NodeController* implements it is very easy to add new commands that influence the peers running in the same virtual machine together with the controller itself. In this way, the workload can be splitted and delegated from the *MainController* to the running *NodeController* instances.

To extend the *NodeController* part by a new command, only two things have to be done. First a method that implements the new command is added to the *NodeController* and second *NodeControlEventHandler* is extended by a new command pattern. Out of the new method all (public) methods available in a peer can be called.

6.2.4 Optimization

6.2.4.1 Increase of Network Size

During the first tests with a large-scale setup, some problems appeared that are based on the overload of a single machine. Using the following three mechanisms made it possible to start a significant higher number of peers all over the network and hold it in a stable state without losing a *NodeController* due to a crash.

- **Load Balancing**

As noticed during the implementation phase of this thesis, it is not a good idea to start many peers on one machine while another machine has to cope just with a few peers. The more peers are running on a machine the faster the fixed number of available sockets is reached which causes the crash of the whole *NodeController*. Therefore the *MainController* tries to distribute the peers in a fair way over the machines.

To do this, the list of *Machine* instances is sorted by the number of peers that are running on it. The machine with the lowest number of running peers is then the next machine that will get one more peer to start.

- **Ring Partitioning**

Another improvement that has been implemented at the same time is the distribution of the Chord IDs. When a *NodeController* registers at

the *MainController* a range of Chord IDs is attached to that *NodeController*. If a peer is started on that *NodeController* it starts with one of the Chord IDs in that range. In this way two peers with small differences in their Chord IDs are typically located on the same machine, accelerating the communication between these peers.

- **Peer Limitations**

Using the *NodeController* brings many improvements. Especially that the *NodeController* and the peers are running in the same virtual machine resources like memory and CPU time are saved and the possibility of direct method calls makes the communication between *NodeController* and peers much faster. However, there is one real disadvantage in sharing one virtual machine. For the operating system it is just one process that is running. While processor and memory capacity is limited only by the hardware, there are resources that are limited by the operating system itself. File descriptors are one of these limited resources, and it is the one that makes trouble in our case. File descriptors are a kind of identifier for an open file, but is also used as identifier for stdout, stdin and stderr. The latter file descriptors are used by every program. Unfortunately, sockets are also file descriptors and the number of available descriptors can not be changed (except by the root user). Because a lot of work is done over the network, a lot of sockets are used. If many peers are running on one node it can happen that the process runs out of file descriptors which inevitably results in a crash of the virtual machine and the loss of the *NodeController* and all peers running in it.

To get rid of this problem the number of peers that can be started in one *NodeController* can be limited by a parameter. As mentioned before, there is a real problem in running out of sockets if all peers are started in the same virtual machine. Therefore it is recommended that the limit is set to a smaller value the more peers should be started in the whole network. In experiments with around 500 peers, about 15 peers per node was a reasonable limit.

Another idea was to start a second virtual machine if the first one gets near a critical bound. In our case this is not possible because of the cluster organization (just one process per machine can be submitted).

6.2.4.2 Parallel Shutdown of Peers

Depending on the join and leave rates it is possible that a huge bunch of peers leave the network in a particular round. In the beginning this was done sequentially. This means that a peer can only leave the network if no other peer is leaving at the moment. Of course that is not nearby a real world behavior and above that it unnecessary prolongs every round and thus the whole experiment. Therefore the shutdown of several peers in one round should be done in a parallel way.

To do this an inner class of the *MainController* has been introduced. It is derived from the Java internal *Thread* class to run independently after the initialization. For every peer that should shutdown one instance of this class is created which does exactly two things:

1. Send the shutdown command over the network to the corresponding peer
2. Wait for an answer to be sure the peer left the ring

After the peer gave the feedback, the *MainController* deletes the information of the peer from its internal structures. This technique decreases the time needed to shutdown peers to a nearly constant value depending on the network connection, what fastens the experiments and thus make more individual experiments possible in the same time.

6.2.5 Measurements

In Section 6.2.2 the tasks of *MainController* have been presented in detail. But one thing is missing there: The measurement phase. The measurements are controlled by the *MainController*, too. However, the implementation is independent from the controller. Therefore the implementation of the experiments is described separately but also with a look on the *MainController* in the following.

6.2.5.1 Chord Ring Check

The basic requirement for our tests is a stable overlay infrastructure. This means that it is necessary to know if all peers that have been started are really up and running and if they are all connected to the same Chord ring. To ensure this, an initial test is created that checks the Chord ring stability at the end of every round. The first implementation of the check was done in a very naive way. During the first tests it turned out that it is not efficient enough

in large networks. However, with the introduction of the *NodeController* it is possible to do this check a lot more efficient.

- **The naive approach**

Because the *MainController* maintains a list of all peers that should run, the most obvious approach is to create a list of peers that can be accessed in the ring and compare it to the maintained list. The implementation of Chord already provides such a method, which works as follows.

One of the running peers is asked for its successor. The approach traverses over the ring by requesting the successor of that successor and terminates as soon as the returned successor is equal to the initial peer. Because the peers are sorted in a ring form, all available peers in this ring have been addressed by the peer that initializes the traversal.

Every peer that has been addressed in this traversal is stored in a list. This list obviously represents all peers that are in the ring. The *MainController* takes this list and compares it to the list of peers maintained by itself. If the lists are equal, all peers that should run actually do so, but additionally they are all in **the same** ring. If the lists are not equal there are two possibilities. Either the peers that are missing crashed before the check or they form another, independent ring. Both is in fact not wanted and can falsify the tests.

- **The partial ring check**

As described above the naive check of the stability of the Chord ring, in large networks, takes a huge amount of time, which may end up in network timeouts of the *MainController* waiting for an answer from the initially addressed peer. Besides, the experiments take very long.

During the implementation of load balancing a technique has been developed that assigns every *NodeController* a range of IDs in the Chord ring. All peers started on a node have an ID out of this subspace. Within the ring the peers are sorted by their IDs. This means that a peer is a successor of another peer if its ID is higher (except for one special case: The peer with the lowest ID is the successor of the peer with the highest ID, so that the ring can be formed). Because the *MainController* knows all peers and their IDs and also the ranges of IDs every *NodeController* is responsible for, there is a more clever approach to check the stability of the ring.

First of all the *MainController* sends a **getpeerids**-request to all *NodeController* instances in parallel. Each *NodeController* takes the peer

with the lowest ID in the corresponding ID subspace. This peer is now contacted to give a list of all their successors up to the upper bound of the ID subspace. With this information it is possible to see if the parts of the ring that are encapsulated in the *NodeControllers* are stable. To check if the whole ring is stable every peer has to look for exactly one more successor, the one with the lowest ID that is higher than the upper bound of the corresponding ID subspace. In the *MainController* the ring can then be reconstructed and checked if it fits together and if all peers have been covered.

Using this technique the workload of the peers that construct the list of their successors is minimized. Due to the parallelization the time it takes to check the chord ring decreased to $\frac{1}{\#NodeController}$ of the time it took before the optimization.

6.2.5.2 Metadata Freshness

As shown in Section 5.2.2 we plan to check the influence of the parameters on the availability of terms in the network and if the network refresh itself during the simulation. The first experiment measures the metadata freshness and deals with terms that left the network with their corresponding peers, but should be still accessible because their TTL is not yet elapsed. These terms are then still listed in one of the directory peers, but their reference to a connected peer will point to the nonentity.

To set up this experiment, one peer, that will act as the only directory peer in the network, is initialized before the regular simulation is started. This peer gets a specific Chord ID and will never terminate during the experiment. The *NodeController* that contains this peer will not be requested to start another peer later on. This is necessary because this peer will be the most busy one if the network gets huge and therefore should not be disturbed by other peers running on the same machine.

Every peer that is started during the simulation runtime, will be advised to post a term whose hash value is the same than the ID of the peer mentioned above. This means that the first peer that has been started will be the directory peer for all peers that join the network later on.

With this configuration it is very easy to determine what percentage of metadata refer to peers that actually are available. Because the *MainController* has knowledge of all running peers at any time, all that has to be done is creating a list of peers that should be accessible according to the network. To get this list a lookup in the Chord ring is performed with the term that has been posted by all peers. Because of the controlled posting of exactly one term from every peer, the answer comes from a directory peer (by the

way the peer we started first) that got posts from all peers that ever joined the network. While a peer stays in the network this post is refreshed from time to time to avoid getting lost because of the TTL. If the peer leaves, the refresh is missing and the post is eventually kicked from the directory. This means that the directory peer returns also peers that already left the network. So this is exactly the list that should be created to determine the ratio.

Again most of the things that are needed to set up this experiment are already available in the implementation of the peer. This means that significant changes or extensions to the part of the Chord and Minerva code are not necessary. The event handler of the peer is extended by another command to call the corresponding methods on an incoming request. The main extensions take place in the `MainController`.

- The part that prepares the test arrangement is added before the regular simulation startup. Using a parameter on startup the *MainController* can be instructed to perform the measurement or not (see Section 7.3).
- The measurement phase in a round is processed if an experiment is in progress. In the experiment described in this section the *MainController* sends a term request to the special directory peer that has been created initially. The answer is a list of peers that have posted this term before (excluding the ones that are erased because of the TTL). In fact, some of these peers could have left the network in the meantime but are still listed. The ratio between really running peers and the pseudo-available ones is then given to the *PlotEngine* to write it into a file.

One general change is done in the posting engine of Minerva. Because of the change from a time based to a round based run of the model, the TTL and repost machinery has to be switched to the round based technique, too. This could be done with a minimal change in the Minerva code. First of all the TTL value that is stored in the peer itself becomes initialized with a round value rather than with a time value. Secondly this value is no more compared to the current time, but to the current round.

At the beginning of every round the `MainController` sends a message to all `NodeController` instances indicating that a new round has been started. Every `NodeController` forwards this message to the peers on its machine and calls the check if a repost is necessary.

6.2.5.3 Metadata Availability

This experiment was also presented in Section 5.2.3. It checks if the directory peers are correctly refreshed from time to time. If a directory peer leaves the network and does not send its share of the directory to the new directory peer, this information is lost until the peers itself send a refresh post.

To check this, the new peers are advised to post a special amount of terms into the network. Again, as mentioned in the experiment above, the *MainController* keeps the overview of the peers.

With the techniques introduced in the experiment for Metadata Freshness it is pretty easy to get a list of all distributed terms. The *MainController* just sends a request for their term lists to all running peers. The number of entries in this list should then be equal to the the number of running peers multiplied with the number of terms posted by every peer if there is no lack of freshness.

Because of the fact that the before implemented features are used, there are only changes to the *MainController*. There is additional code that performs the list requests and evaluates them against the stored number of running peers. Besides a new parameter has been introduced to give the user the possibility to choose one of the two tests that have been presented (see Section 7.3).

6.2.6 Plot Engine

As described before the only measurement that has been done in the initial implementation was the check of the chord ring stability after every round. Because this is not really a relevant experiment to produce evaluation results, it was enough to print the result of that check on the standard output. For the "real" experiments this is not applicable, because the standard output normally gives information in a human readable format including data that is not needed for automatic evaluation and therefore would have been converted separately into a machine readable format. To avoid this extra work, the *PlotEngine* can be used (see Figure 6.5).

To be able to encapsulate the results from one experiment in a file of a well-defined format a small engine has been developed that collects the information of a certain experiment and put this in a file that can be processed later on. The filename itself contains the most influencing arguments like join rate, leave rate or name of the experiment, so the files can be archived in a proper way. Thus it is very easy to write scripts that process just one or perhaps a correlated bunch of result files to get for example a graphical interpretation of the data.

This engine is encapsulated in a small class file and instantiated in the *MainController* because all data is collected there. Depending on the experiment that should be processed, the relevant data is chosen out of information that is already stored in the *MainController* and the on-demand results that comes from requests to *NodeController* instances or peers by the plotting engine. This data is then written to the corresponding file after a number of rounds that can be set as it is needed.

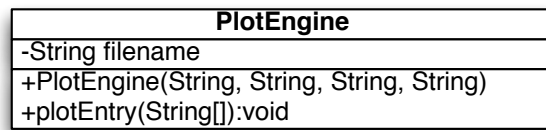


Figure 6.5: Class Diagram of the *PlotEngine* class

Chapter 7

Deployment on the Cluster

To start the experiments with the simulation framework presented before, a cluster owned by the MPI is used. The core of the cluster currently consists of 94 machines. Each machine provides two processor units; i.e., there is a total of 188 processors available. Every system consists of the same hardware and is equally installed. This always provides the same environment to every process that is started on the cluster. How this cluster is used to start our experiments is explained in this chapter.

7.1 SSH

At the beginning of the thesis, the cluster was just a huge number of machines put together in one room and connected to the internal network of the MPI. This means every cluster node was accessible like a normal computer. So the first approach to distribute peers over the cluster was the use of SSH.

Most users of SSH use it to login to a remote machine and work on it as on the local one. But SSH can also be used to execute programs on remote machines. This can be done by just typing the following command:

```
ssh hostname command
```

where *hostname* is the hostname or IP address of the remote machine and *command* is the command that should be executed. The command must be available on the remote machine.

Although it seems that the remote startup is very easy, the standard SSH configuration does not work for our purposes, because a password has to be typed in to log on to the remote machine. Fortunately, SSH supports public key authentication, such that a login is possible without interaction. This makes it possible to fully automate the distribution process. How to

configure these settings is explained in [6], while a more detailed description of the used public key generation program *ssh-keygen* can be found in [5].

As told above it is necessary to have the program available on the remote machine. In the case of this thesis, the whole project must be accessible on all machines of the cluster that should be used. In fact, that is no problem, because the home directories, stored on a separate fileservers, are already available on every cluster node.

With all these requirements regarding remote startup of programs fulfilled, the *MainController* can use this underlying layer to distribute the peers over the available cluster nodes (see Figure 7.1).

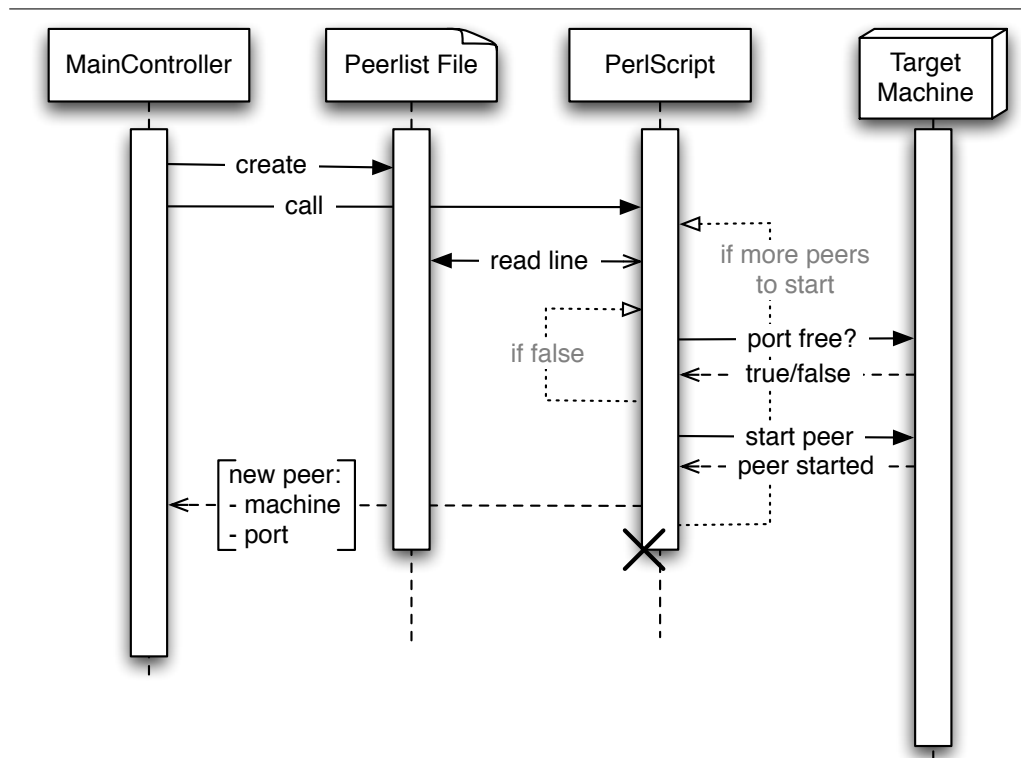


Figure 7.1: Start peers on remote machine using the startup perl script

As in every round a bunch of peers is started the remote startup has been outsourced to an external perl script. This minimizes the system calls out of the Java application. The startup of peers in a round can then be categorized into two groups:

- **Preparation Phase**

In this phase the *MainController* determines how many and on which machines the peers should be started. From this information a file is created that holds the hostname of each machine followed by the number of peers that should be started on this machine. When this file is written, the preparation phase is finished.

- **Execution Phase**

This phase starts with a system call to execute a perl script. This perl script reads the file created in the preparation phase. For every peer that should be started on a certain remote machine the first step is to find two available ports on that machine, one for the Chord communication and one for the Minerva communication. This is done using the *netstat* and *grep* command. If, for example, it should be checked that the port 9000 is free on mpiao9520 the following command is executed by the script:

```
ssh mpiao9520 netstat -a -n | grep 9000
```

If the output of this command is empty, port 9000 is definitely free. If the port is not free the number is incremented by one and checked again. As soon as two free ports have been found the peer can be started using the found ports as arguments.

These steps are repeated for all peers that should be started. As soon as a peer has been started the perl script sends the information of the new running peer to the *MainController*. When all peers are up and running the script terminates and the *MainController* continues with the next step in the current round.

7.2 Grid Engine

Although the solution to start peers on remote machines using SSH worked remarkably well, the cluster itself runs into problems if it provides this service. Machines that are reachable via SSH can be used by every person that has a login on that machine. This means it can happen that the experiments of one user disturb experiments of another user, making experiments unreproducible because the available resources are not always identical. Therefore the *Grid Engine* has been introduced. It schedules the jobs that are transmitted in a queue and blocks a cluster node if a job is already running on it. This means that every job gets the same environment and the same resources without being disturbed by another process. Another improvement

of the queueing technique is that the cluster is optimally used because the *Grid Engine* distributes the queued jobs over the available machines what frees the user from searching for an unused machine. However, one does not know on which machine the job will be started by the *Grid Engine* what lead to some extra effort in our case.

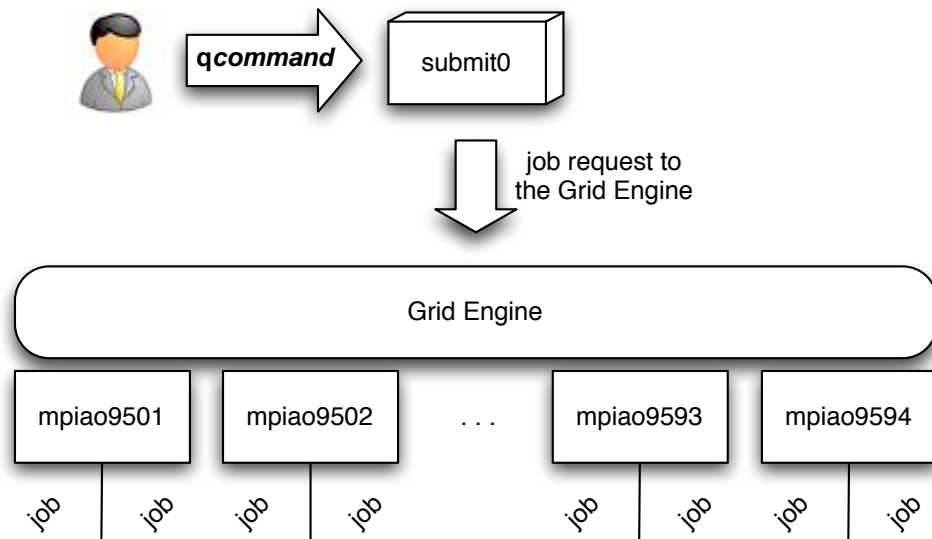


Figure 7.2: MPI cluster with Grid Engine

As mentioned above about 190 processor units are available on the cluster. Some of them are reserved for short time experiments (less than 4 hours). Others can be used without limitations. One can place a job on a processor unit and until the job is finished this unit is exclusively reserved just for that job.

The *Grid Engine* can be accessed from exactly one machine. In our case this is `submit0.mpi-sb.mpg.de`. Everyone who has an account on this machine can login via SSH and use *Grid Engine* commands to start jobs, kill jobs and get some status information about running jobs and available machines.

The most important commands are shortly described in the following. A more precise description of the commands and their parameters can be found at [2, 3, 4] or in the manual pages installed on `submit0`:

- `qsub (qsub [options] [command | - [command_args]])`

This command is used to send jobs to the grid engine.

One of the most commonly used option is *-cwd* which executes the job from the current working directory.

If one wants to submit a binary file and not a script, which was necessary for this thesis at some point, the *-b* option can be used to tell the grid engine that the command could also be a binary:

-b y[es]	<i>command</i> may be a binary or script. Nothing except the path of the command will be transferred to the host that should execute the job.
-b n[o]	<i>command</i> needs to be a script and it will be handled as script. It will be transferred to the host that should execute the job.

- **qstat**

This command shows the current status of the available *Grid Engine* queues and the jobs associated with the queues. Selection options allow you to get information about specific jobs, queues or users. Without any option *qstat* will display only a list of jobs with no queue status information.

qstat has so many options that it is not possible to cover them here all. The most commonly used option during this thesis was the *-u* option which makes it possible to show the jobs of a particular user.

- **qdel** (*qdel* [options] [-u *wc_user_list*] [*wc_job_range_list*])

This command is used to delete queued or running jobs from the cluster.

If one wants to delete a single or some particular jobs this can be done by adding the corresponding job numbers to the "job range list". If one wants to delete all jobs owned by himself the *-u* option followed by the own user name can be used.

-u <i>wc_user_list</i>	Deletes only those jobs which were submitted by users specified in the list of usernames. Non-admin users can only delete their own jobs. If no job is specified all jobs from that user are deleted.
<i>wc_job_range_list</i>	A list of jobs which should be deleted

- **qlogin** (*qlogin* [options])

qlogin can be used to get shell access on one of the nodes. It can

be compared with SSH with the difference that you do not know on which machine you will login. Normally this is only used for debugging sessions because the node is blocked as long as the user is logged in, even if there is no workload on the machine.

A job that is submitted with *qsub* is transferred to one of the cluster nodes. If there is no node available because they are all blocked by a job, new jobs are queued and transferred to the first node that is free again. The structure of the MPI cluster is shown in Figure 7.2 including the host names.

Although the *Grid Engine* is a good enhancement for experiments, the switch to this technique makes it necessary to change the distribution approach described in Section 7.1 significantly, because a direct connection via SSH is no more possible. The most obvious solution of this problem is using the *qlogin* command which is provided by the *Grid Engine*. While *qlogin* gives an SSH-like connection method it is not possible to run more than one process on one machine without using some nasty tricks. Beyond this it is not guaranteed that the node that should be used is available. Although it is possible to use *qlogin* to connect to a special node, the connection is only possible if the node is not used by another process. Otherwise the request will be scheduled and executed as soon as the running process terminates. In fact, a long time can elapse before the peer is really started.

With the introduction of the *NodeController* this problem is solved. On every machine that should be used as a "peer-container" one *NodeController* is started. This has the following advantages:

- Every *NodeController* blocks one cluster node. As the *MainController* waits until the *NodeController* instances are all running, it is guaranteed that the machines that should be used for an experiment are really available.
- Starting peers do not need a system call anymore, because the *NodeController* initializes a new peer when the corresponding request comes from the *MainController* over the network.

As mentioned before, the *MainController* is started first. It is told by a parameter how many *NodeController* instances will be started. As soon as the *MainController* is up and running the given number of *NodeController* instances can be started. This environment setup process is shown in Figure 7.3.

For simplification a new bash script was developed that automatically starts the *MainController* and the *NodeController* instances using the *qsub* command of the *GridEngine*. It takes a bunch of parameters and uses them

during the startup of the controllers. It waits for the *MainController* and automatically starts the *NodeController* instances just in time.

As mentioned above the startup of new peers does not need any system calls if the *MainController-NodeController* environment is used. The new peers are created by the *NodeController* in the same virtual machine. The distribution of the peers is done over the network (see Figure 7.4).

To start a new peer the *MainController* has to send the request to the corresponding *NodeController* using the stored *Machine* instance and the *ControlCommunicator*, respectively. The *NodeController* observes the startup of the peer and sends the network information (e.g. the port the peer listens on) as answer back to the *MainController*.

7.3 Initialization of an Experiment

The easiest way of starting an experiment is using the *startupall* script. It takes a few parameters that directly influence the behavior of the system:

```
startupall $NC $PPN $TTL $RP $JR $LR $SCENARIO
```

\$NC	The number of <i>NodeController</i> instances that should be started
\$PPN	The upper bound of the number of peers that can be started on one machine
\$TTL	The time-to-live value of terms
\$RP	The number of rounds after which a term should be reposted to avoid that this term is deleted from the directory
\$JR	The join rate of peers
\$LR	The leave rate of peers
\$SCENARIO	The experiment that should be processed <i>term</i> : Test for Metadata Freshness <i>directory</i> : Test for Metadata Availability <i>none</i> : Run the system without measurements

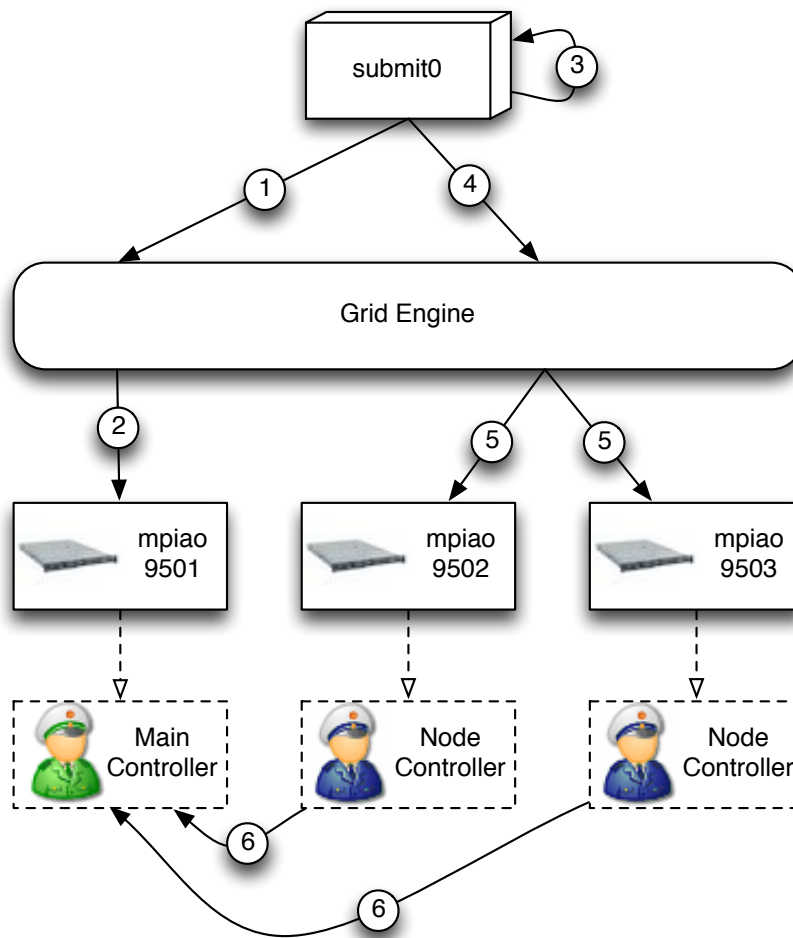
Beside this fully automated solution, the controller instances can also be started manually. That is important if one want to give some options to the *qsub* command. To start the *MainController* the following command should be typed:

```
qsub mainstartup.sh $NC $JR $LR $PPN $TTL $RP $SCENARIO
```

If the *MainController* has been started on a machine the *NodeController* instances can be started as follows:

```
qsub nodestartup.sh $HOST [$PORT]
```

where `$HOST` is the hostname of the machine the *MainController* is running and `$PORT` is the port on which the *MainController* is listening for the registration request. If the port is not given, the default port (9000) is used.



1. Send qsub command to the *Grid Engine* to startup a MainController
2. The *Grid Engine* chooses a free node and starts the MainController
3. Check if the MainController is up and running using the qstat command
4. If the MainController runs send qsub commands to the *Grid Engine* to start NodeController (in this case two)
5. The *Grid Engine* again chooses free nodes and starts the NodeController instances
6. The NodeController instances register themselves at the MainController

Figure 7.3: Setup of the controlling environment

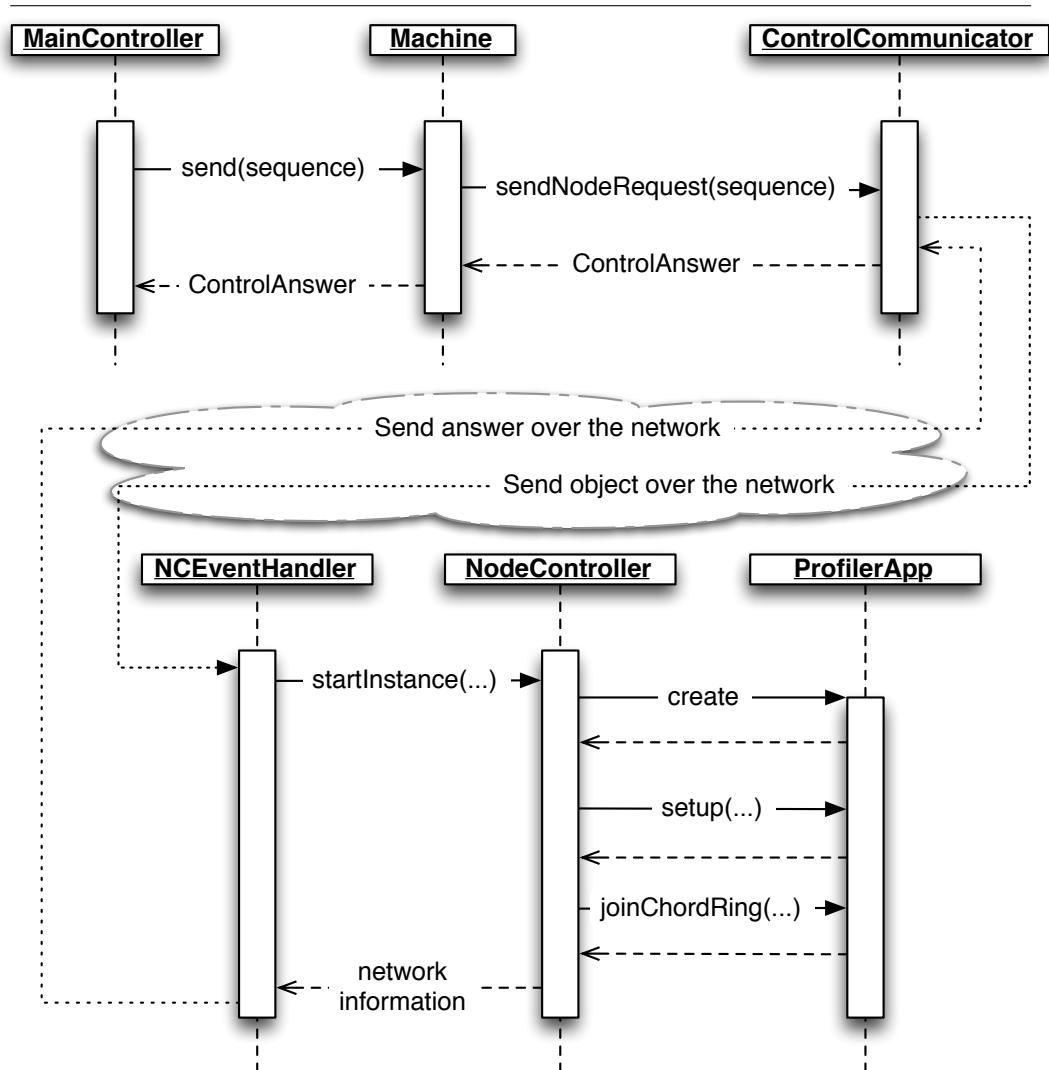


Figure 7.4: Starting a peer using the *NodeController*

Chapter 8

Experimental Results

As stated in the earlier chapters a simulation framework has been designed and implemented for running tests on large scale Peer-to-Peer networks. This framework is now used to evaluate Minerva on the MPI cluster. In this chapter the results of these tests are presented and compared with the expected results from the theoretical analysis. The tests study the metadata freshness (see Section 5.2.2) and availability (see Section 5.2.3). During this special tests the general network stability is also tested regularly, but the appearance of an instability was very rare over all tests, so it will not be discussed in more detail.

The basic setup of the tests is the same for both experiments. The simulation framework is started with the appropriate parameters. In round zero the network is empty and the peer population for the subsequent rounds follow the join and leave models that have been presented in Section 5.1. For all experiments we fix the join rate to two.

We want to study metadata freshness and availability for two different TTL/repost parameter pairs, namely 20/10 and 40/20, yielding four measurement studies. For each study we vary the leave rates between 0.01 and 0.05 and present averages over 10 runs each in the following sections.

8.1 Metadata Freshness

If a peer leaves, the metadata objects in the directory will not vanish immediately but only when the TTL of the corresponding post expires. This leads to inconsistency between the peers that are still alive in the network and the list of peers that is stored in the directory. Exactly these inconsistencies will be studied in this section.

In this experiment exactly the same term is posted by all peers. The first

peer that is started will never leave the network and will get a Chord ID so that it is responsible for the chosen term and, thus, will be the only directory peer in the network that will receive posts (see Figure 8.1). We measure the ratio of metadata objects in the directory that point to peers that are still alive over all objects stored in the directory. To get this data we need the list of all currently running peers (this is a list maintained by the *MainController* anyway) and the list of metadata objects from the directory peer.

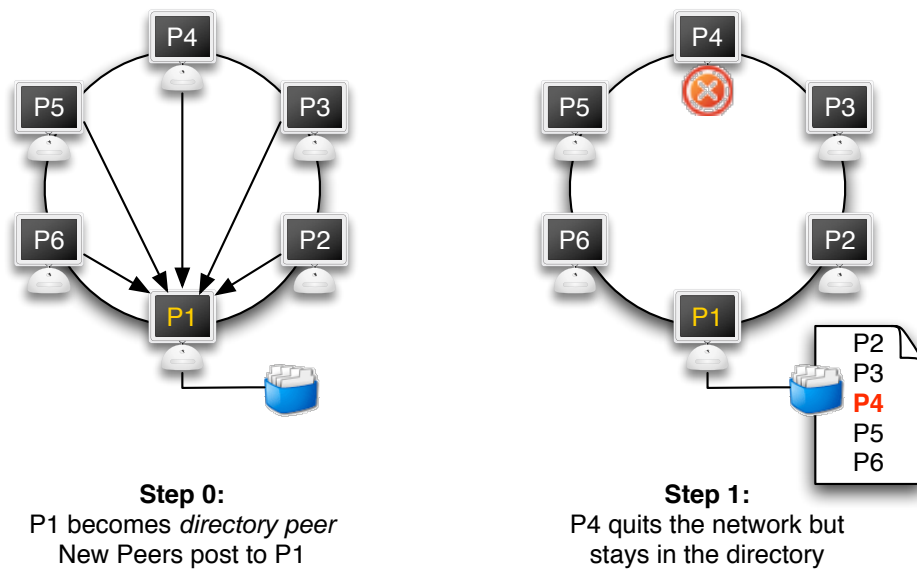


Figure 8.1: Setup of the Metadata Freshness Test Environment

In Table 8.1 (TTL 20 - Repost 10) and in Table 8.2 (TTL 40 - Repost 20) the results of the experiments are illustrated. For every run we calculate the ratio of fresh metadata objects. In fact this represents the freshness of the directory at the moment of the run. If the ratio decreases there are more stale references in the directory which refer to peers that are no longer available in the network.

The two bottom lines in the tables compare the averaged experimental results to the expected ones using the predictions from Section 5.2.2. In Figure 8.2 and 8.3 this is also illustrated in a graphical way.

The experimental results are pretty close to the expected ones (see Figure 8.2 and 8.3) and show a clear tendency. For example, when the leave rate increases from 0.01 to 0.05, the freshness ratio drops from 0.878 to 0.627, as more of the peers referenced in the directory leave the system before their posts disappear from the directory. On the other hand, for the same leave

rate of 0.05, the longer TTL of 40 causes the freshness to drop again from 0.627 to 0.509. In fact better freshness ratios are desirable but lead to more network traffic because reposts have to be sent more often (under the same user behavior).

As expected one can see that the directory freshness decreases if the leave rate and/or the TTL/Repost intervals are increased. This leads to the conclusion that Minerva was implemented correctly within this scope, as the freshness depends as expected directly on the number of leaving peers and the refreshing intervals of their posts.

Run	Experiments (Leave Rate)				
	0.01	0.02	0.03	0.04	0.05
1	0.94	0.85	0.88	0.92	0.86
2	0.91	0.82	0.75	0.75	0.53
3	0.84	0.86	0.78	0.78	0.69
4	0.89	0.83	0.76	0.68	0.69
5	0.88	0.79	0.72	0.75	0.56
6	0.92	0.84	0.71	0.63	0.48
7	0.90	0.79	0.79	0.60	0.57
8	0.82	0.78	0.76	0.63	0.63
9	0.81	0.74	0.69	0.57	0.61
10	0.87	0.79	0.73	0.69	0.65
average freshness	0.878	0.809	0.750	0.700	0.627
expected freshness	0.899	0.813	0.740	0.678	0.625

Table 8.1: Metadata Freshness with Join Rate 2, TTL 20 and Repost 10

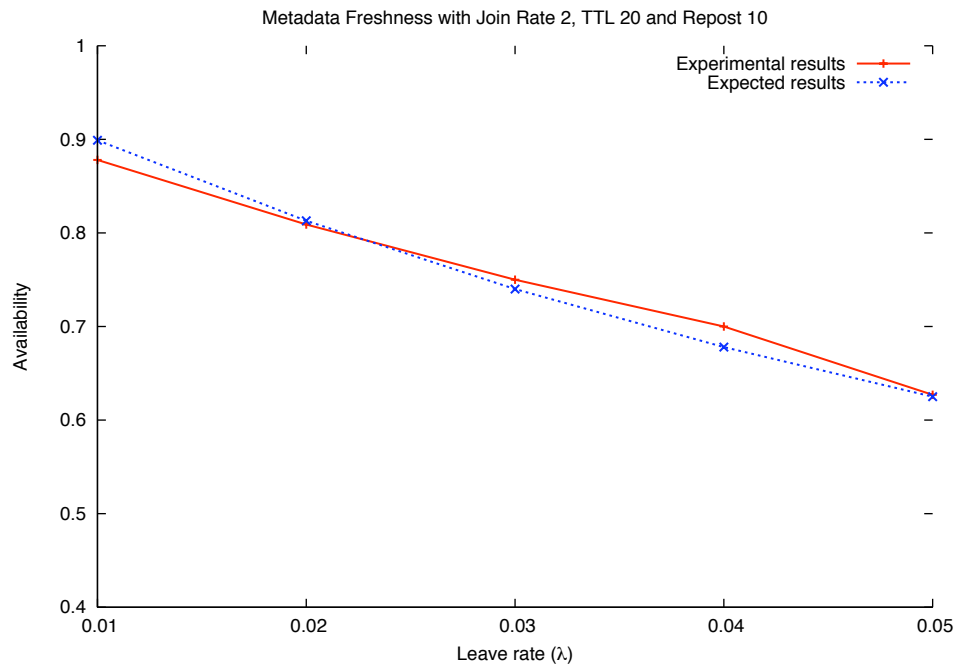


Figure 8.2: Metadata Freshness with Join Rate 2, TTL 20 and Repost 10

Run	Experiments (Leave Rate)				
	0.01	0.02	0.03	0.04	0.05
1	0.84	0.95	0.76	0.93	0.86
2	0.91	0.82	0.77	0.53	0.68
3	0.81	0.78	0.62	0.49	0.58
4	0.86	0.74	0.59	0.44	0.57
5	0.85	0.65	0.61	0.54	0.41
6	0.80	0.67	0.62	0.49	0.48
7	0.78	0.65	0.59	0.38	0.39
8	0.77	0.67	0.54	0.46	0.41
9	0.81	0.66	0.54	0.49	0.36
10	0.80	0.66	0.55	0.51	0.35
average freshness	0.823	0.792	0.619	0.526	0.509
expected freshness	0.844	0.725	0.632	0.560	0.502

Table 8.2: Metadata Freshness with Join Rate 2, TTL 40 and Repost 20

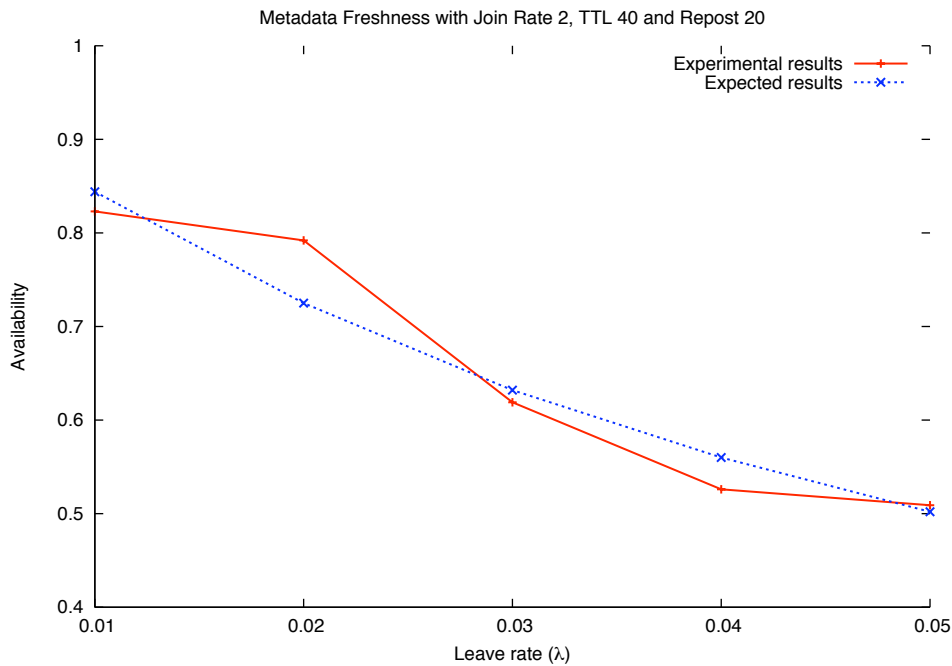


Figure 8.3: Metadata Freshness with Join Rate 2, TTL 40 and Repost 20

8.2 Metadata Availability

When a directory peer leaves the network, it takes its share of the metadata directory out of the network. To avoid the total loss of these references, the peers repost their term-specific metadata regularly. With more directory peers leaving the network the number of terms for which currently no metadata are available increases. In this section the behavior of Minerva within this scenario is analyzed.

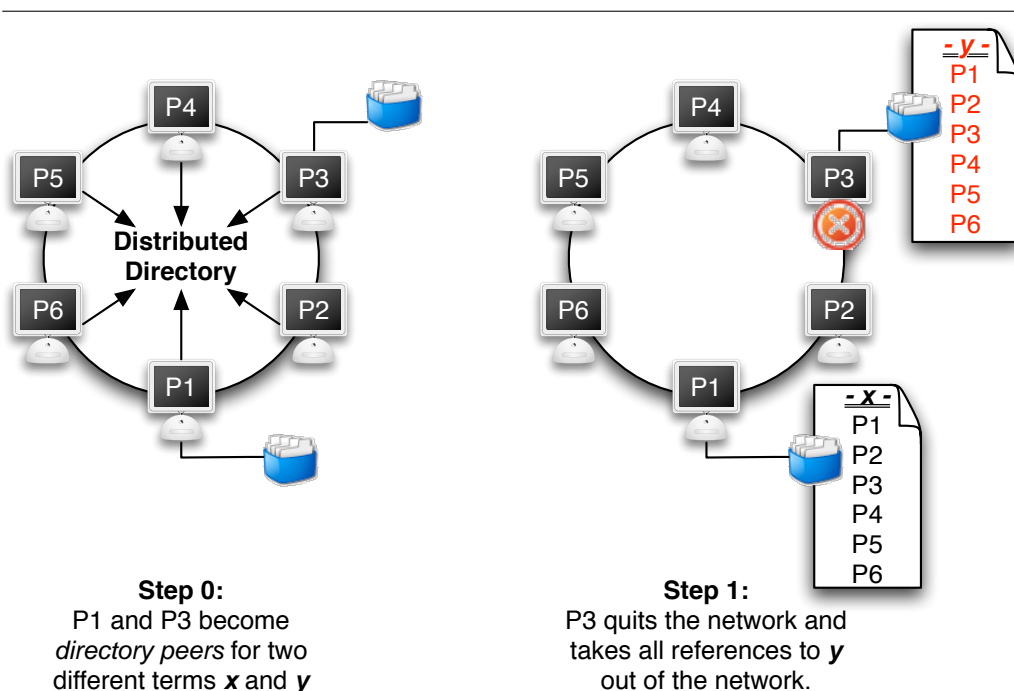


Figure 8.4: Setup of the Metadata Availability Test Environment

In this experiment, each peer posts metadata for the same 100 terms. We measure the ratio of posts in the network that belong to currently running peers over the expected number of posts, which is by design of the experiments 100 times the number of currently running peers (because every peer posts metadata for exactly 100 terms). To calculate the ratio we retrieve the PeerLists from the directory peers for every term. Metadata that is stored in the directory that refers to peers that are no longer running is not considered and thus removed from the list.

In Table 8.3 and 8.4 the experimental results of the two test series are shown. Again the average over all test runs is calculated to compare it to the

predictions from Section 5.2.3. Every entry in the table gives the availability of metadata at the moment of the run.

The experimental results of this experiment are also very close to the expected ones (see Figure 8.5 and 8.6) and also show an obvious tendency. For example, when the leave rate increases from 0.01 to 0.05, the availability ratio drops from 0.941 to 0.799, as more peers leave the network and take its share of the directory with them. On the other hand, for the same leave rate of 0.05, the longer repost interval of 20 causes the freshness to drop again from 0.799 to 0.634.

The availability of the metadata depends significantly on the leave rate and the repost interval, as expected. In comparison to the metadata freshness results, one can see that the repost value has a higher impact on the metadata availability. In networks with a high leave rate one would therefore try to shorten the repost interval to raise the quality of the directory. However, this unfortunately leads to more network traffic which could slow down the network depending on its size.

Run	Experiments (Leave Rate)				
	0.01	0.02	0.03	0.04	0.05
1	0.98	0.93	0.91	0.88	0.95
2	0.96	0.89	0.90	0.89	0.91
3	0.97	0.88	0.87	0.85	0.89
4	0.93	0.95	0.89	0.82	0.80
5	0.93	0.89	0.82	0.88	0.63
6	0.91	0.97	0.89	0.85	0.62
7	0.93	0.84	0.92	0.90	0.72
8	0.92	0.90	0.85	0.81	0.87
9	0.96	0.89	0.88	0.82	0.83
10	0.92	0.91	0.90	0.84	0.77
average availability	0.941	0.904	0.883	0.854	0.799
expected availability	0.959	0.917	0.875	0.831	0.788

Table 8.3: Metadata Availability with Join Rate 2, TTL 20 and Repost 10

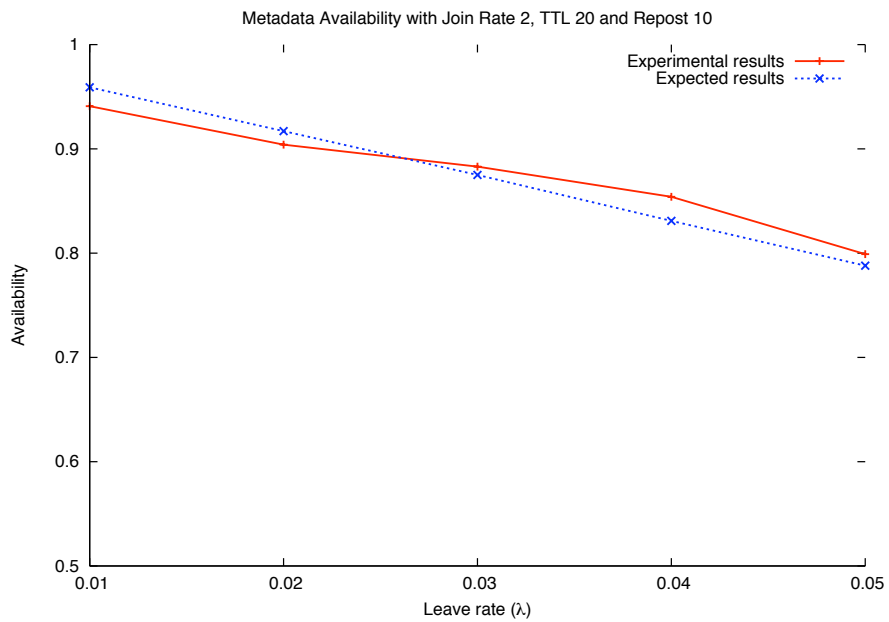


Figure 8.5: Metadata Availability with Join Rate 2, TTL 20 and Repost 10

Run	Experiments (Leave Rate)				
	0.01	0.02	0.03	0.04	0.05
1	0.98	0.88	0.83	0.70	0.67
2	0.98	0.90	0.80	0.63	0.59
3	0.89	0.91	0.72	0.69	0.61
4	0.91	0.85	0.76	0.72	0.58
5	0.91	0.81	0.74	0.64	0.55
6	0.82	0.88	0.79	0.65	0.69
7	0.87	0.89	0.81	0.71	0.65
8	0.94	0.83	0.79	0.63	0.68
9	0.84	0.86	0.78	0.64	0.64
10	0.89	0.81	0.74	0.68	0.68
average availability	0.903	0.862	0.776	0.669	0.634
expected availability	0.921	0.839	0.754	0.667	0.579

Table 8.4: Metadata Availability with Join Rate 2, TTL 40 and Repost 20

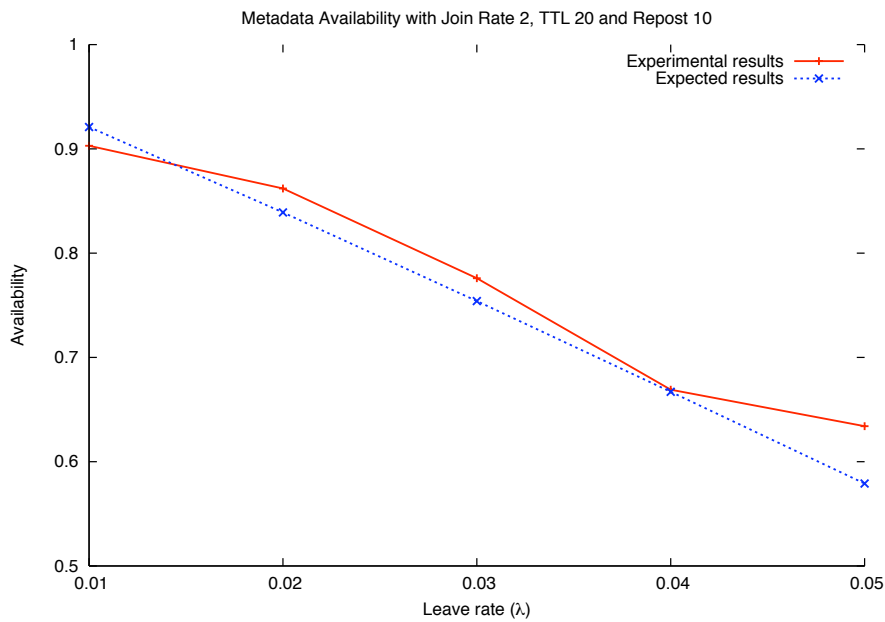


Figure 8.6: Metadata Availability with Join Rate 2, TTL 40 and Repost 20

8.3 Discussion

During the tests that have been presented in this chapter, the only limitations of this work came with the cluster infrastructure. With a reasonably low number of twenty peers on one cluster node it was possible to raise the number of stable running peers to a value of about 800. Because the cluster was also used by other people, it was not possible to get more than about 40 nodes. With a bigger number of available nodes in the cluster it would be possible to raise this number to a significantly higher value.

The limitation of the maximum number of peers that can be run on one cluster node has its origin in the limitation of resources on the machine itself which is fixed by the operating system and can only be changed by the administrator. If the network got unstable, typically one of the machines ran out of sockets, which causes the *NodeController* and all peers running in it simultaneously. In real world this would be equal to the breakdown of a part of the internet.

As the measurement engine produces some network overhead and the cluster itself was heavily loaded by other people, the number of running peers in the created networks was lower than it would be possible in a full featured test run. But it was absolutely sufficient to compare the experimental results with the expected ones.

Chapter 9

Conclusion

Testing large networks in an automatic way makes it possible to evaluate applications with a minimum amount of work. The framework that results out of this thesis makes such tests possible, especially under the aspect of different user behavior models. The network can for example be stressed with a lot of joining and leaving peers. On the other hand it can also be tested with a large number of running peers which stay for a longer time depending on the things that should be checked.

This work has introduced mathematical models that mimic the behavior of real users, to serve as a basis for meaningful experiments. A framework for automated tests based on these models was designed and implemented. An example deployment was performed using the MPI cluster infrastructure and experimental studies based on the framework have been conducted on Minerva to test its scalability, data availability, and data freshness.

The experiments showed that Chord as underlying Peer-to-Peer system was a good choice. Although there are problems if the network grows quickly the overall stability is not affected and the tests of Minerva's basic functions gave very good results. This leads to the conclusion that not only the system design but also the implementation work has been correctly done.

In the future, additional tests that consider some more enhanced features of Chord and Minerva, e.g. *replication*, could be performed by extending the framework. Although this work focused on Chord as underlying Peer-to-Peer system, hereafter the framework could be extended to be able to test applications based on other Peer-to-Peer overlay networks (e.g., *Pastry*).

Although it is out of scope for this immediate work, it should be mentioned that using the test framework showed up some minor bugs in the Minerva implementation that affect single peers but not the stability of the whole network. These results could be used to improve Minerva in the future.

Bibliography

- [1] Exponentialverteilung. <http://de.wikipedia.org/wiki/Exponentialverteilung>. Last visited July 18, 2006. German language.
- [2] GridEngine command qdel. <http://www.hpc.dtu.dk/GridEngine/man/qdel.html>. Last visited August 22, 2006. English language.
- [3] GridEngine command qstat. <http://www.hpc.dtu.dk/GridEngine/man/qstat.html>. Last visited August 22, 2006. English language.
- [4] GridEngine commands: qsub, qsh, qlogin, qrsh, qalter, qresub. <http://www.hpc.dtu.dk/GridEngine/man/qsub.html>. Last visited August 22, 2006. English language.
- [5] Manual pages - ssh-keygen. <http://www.openbsd.org/cgi-bin/man.cgi?query=ssh-keygen>. Last visited September 11, 2006. English language.
- [6] Openssh public key authentication. <http://sial.org/howto/openssh/publickey-auth/>. Last visited September 11, 2006. English language.
- [7] Poisson-Verteilung. <http://de.wikipedia.org/wiki/Poisson-Verteilung>. Last visited July 18, 2006. German language.
- [8] M. Bender, S. Michel, G. Weikum, and C. Zimmer. The MINERVA Project: Database Selection in the Context of P2P Search. In *BTW 2005*, Karlsruhe, Germany, 2005.
- [9] Matthias Bender, Sebastian Michel, and Gerhard Weikum. P2P directories for distributed web search: From each according to his ability, to each according to his needs. In Roger S. Barga and Xiaofang Zhou, editors, *2nd International Workshop on Challenges in Web Information Retrieval and Integration (WIRI2006) @ICDE2006*, ICDE Workshops (ICDEW), page 51, Atlanta, GA, USA, 2006. IEEE Computer Society.

- [10] F.M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. Technical Report DCS-TR-487, Rutgers University, 2002.
- [11] D. Karger, E. Lehmann, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.
- [12] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242. ACM Press, 2002.
- [13] Andy Oram. *Peer-To-Peer; Harnessing the Benefits of a Disruption Technology*. O'Reilly, Beijing - Cambridge - Farnham, 2001.
- [14] Gopal Pandurangan, Prabhakar Raghavan, and Eli Upfal. Building low-diameter peer-to-peer networks. *Selected Areas in Communications, IEEE Journal*, 21:995–1002, Aug. 2003.
- [15] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, San Jose, 2002.
- [16] Ralf Steinmetz and Klaus Wehrle. Peer-to-peer-networking & -computing. *Informatik Spektrum*, 27(1):51–54, 2004. Springer, Heidelberg.
- [17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures and protocols for computer communications*, pages 149–160. ACM Press, 2001.
- [18] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasunderarm. Odiessa: A peer-to-peer architecture for scalable web search and information retrieval. Technical report, Polytechnic University, 2003.
- [19] Y. Wang, L. Galanis, and D. J. de Witt. Galanx: An efficient peer-to-peer search engine system. Available at <http://www.cs.wisc.edu/yuanwang>.

Index

C	
central lookup server	5
Chord	9
consistent hashing	9
cyclic ID space	9
D	
DHT <i>see</i> Distributed Hash Table	
directory maintenance	14
Distributed Hash Table	7
E	
Exponential distribution	19
F	
finger table	10
G	
ghost-peers	23
Gnutella	19
V0.4	5
V0.6	7
Grid Engine	47
L	
leafnode	7
Load Balancing	37
local index	14
lookup function	3, 9
M	
MainController	30
metadata	14
Metadata Availability	24
Metadata Freshness	22
Minerva	13
N	
Napster	3, 5, 19
NodeController	33
P	
Peer Limitations	38
Peer-to-Peer system	
structured	8
unstructured	4
centralized	5
hybrid	7
pure	5
PeerList	14
Poisson distribution	19
Post	14
Q	
qdel	49
qlogin	49
qstat	49
qsub	48
query processing	14
query routing	14, 15
R	
replication	25
repost	18
result merging	16
Ring Partitioning	37
S	
signaling traffic	6
SSH	45

ssh-keygen	46
startup script	
mainstartup	52
nodestartup	52
startupall	51
supernode	7

T

TTL	14
-----------	----