# U.P.

## JOB DISTRIBUTION NETWORK FOR MASS COMPUTING PROJECTS

***Master's Thesis***

*Anders Rune Jensen*
*Lau Bech Lauritzen*
*Ole Laursen*

June 2005

AALBORG UNIVERSITY

Aalborg University
Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

**Title:**

U.P. – Job Distribution Network for Mass Computing Projects

**Project period:**
DAT6, February 1st – June 2nd, 2005

**Project group:**
d607b

**Members of the group:**
Anders Rune Jensen
Lau Bech Lauritzen
Ole Laursen

**Supervisor:**
Gerd Behrmann

**Number of copies:** 7

**Number of pages:** 76

**Abstract**

This thesis describes the design of U.P., a decentralised job distribution system for mass computing projects. The design is based on the concept of distributed hash tables, which when combined with keyword-based indexing and load balancing provides a robust and scalable peer-to-peer network for mass computing projects to distribute jobs.

An implementation of the system is developed in C++ for the p2psim simulator. Realistic and synthetic tests of the implementation are conducted to explore the intrusiveness, scalability and robustness, with promising results, although a few problems remain.

Hence, we conclude that a decentralised architecture can be a viable alternative to the currently deployed centralised designs with better scalability, high availability and easier and cheaper deployment for new projects. We also believe that the decentralised architecture can be used to provide the foundation of a more general computational grid.

# Preface

This thesis documents the design of a decentralised, robust and scalable job distribution network for mass computing projects. The design has been developed as part of the DAT6 semester at the Department of Computer Science, Aalborg University.

We would like to thank Josva Kleist for providing us with access to the departmental cluster for our tests. We would also like to thank Ben Segal and Markku Degerholm at CERN for providing us with data from LHC@home and Jinyang Li and Jeremy Stribling for assistance with the use of the p2psim simulator.

*Aalborg, June 2005,*

*Anders Rune Jensen* *Lau Bech Lauritzen*

*Ole Laursen*

# Contents

# Chapter 1

# Introduction

This chapter introduces the idea of mass computing and presents an analysis of a mass computing project. Based on this, we then sum up the important issues for a mass computing system and suggest a different idea for how to architecture such a system.

## 1.1 Mass Computing

The idea of using the idle cycles of personal computers to run large parallel applications, known as mass, desktop or public-resource computing, has become very popular during the nineties with the growth of fast household computers and the Internet. The original idea of cycle scavenging dates back to the Worm project [55] on the local area network at the Xerox Palo Alto Research Center and was later in the eighties further developed when Condor [37] was started to harvest the idle cycles of department machines.

Department machines are relatively uniform with high speed network connections and under administrative control. This is in contrast to the heterogeneous mass of computers that the Internet has brought with very different setups, often slow network connections and no means of control. Mass computing projects are relying on the goodwill and service provided by ordinary computer users [2].

To avoid abolishing this goodwill, mass computing applications must refrain from interfering with the users' normal work. This further restricts the network bandwidth and memory resources available to the application. The limited amount of network bandwidth available combined with restrictive firewall settings makes it infeasible to run parallel jobs that require communication. Instead the work is divided into completely independent parts and distributed to the clients.

Mass computing has attracted a large number of users that have formed communities, e.g. based on country of residence or workplace. Each commu-

nity has a leader board which ranks users based on the amount of work done and each community is also ranked based on the total amount of work done by all the members. The rankings provide incentives for people to donate their spare resources.

One of the largest mass computing projects, SETI@Home [2], began in 1999 and now has more than 400 000 active users worldwide. For completely independent tasks, the aggregate computing power of SETI@Home exceeds even the fastest supercomputers today. While most successful in terms of users, it is predated by the Great Internet Mersenne Prime Search (GIMPS) [24] from 1996 and the Distributed.net project [19] from 1997 which aims at brute-forcing encryption algorithms. Folding@home [54] started in 2000 with the goal simulating protein folding, and data from the simulations has been used in more than 20 scientific papers. ClimatePrediction.net [15] is investigating the sensitivity of climate models used to predict the weather of the next century in the face of global warming. It does not use redundant computing since the simulations are probabilistic and cannot be compared.

The Berkeley Open Infrastructure for Network Computing (BOINC) project [1] was started in 2003 as a framework for building mass computing projects. The goals of the project are to reduce the barriers of entry for new mass computing projects, share resources among different projects, support diverse applications and rewarding participants.

The architecture of a BOINC project is centered around a relational database storing information about the project such as information about workunits, results and communities. The clients connect to a server to get new workunits and deliver results. BOINC ensures that workunits are reassigned if a client does not reply to avoid starvation, and will optionally also ensure that workunits are processed redundantly by several clients to work-around erroneous results and cheating by malicious users.

When a result has been confirmed to be correct, the client is given a certain amount of credit corresponding to the work done. Credit listings are periodically exported from the database and displayed as web pages. BOINC also supports a concept known as trickling where clients periodically report back to the project with how far they have gotten with their workunit. Thus their credit status can be updated faster, which is important for the impatient users with very long workunits.

## 1.2 Analysis of a Mass Computing Project

In order to better understand the usage patterns for mass computing projects, we have examined data from the LHC@home project [36]. LHC@home was launched September 2004 with the aim of verifying through simulation the long-term stability of the Large Hadron Collider apparatus at CERN.

### 1.2.1 Analysis of Clients

We analysed the clients from LHC@home to find out how heterogeneous they are. It may be important to for a mass computing project to address any client heterogeneity. Our analysis revealed the following characteristics.

**Bandwidth** The bandwidth distribution is illustrated in Figure 1.1. An asymmetric relationship between upstream and downstream bandwidth is clear from the figure: 65% of the nodes have between 10 and 500 kB/s downstream while 15% are below 10 kB/s. For upstream, 65% lie between 1 and 80 kB/s while 25% have below 1 kB/s. But due to the small amounts of data transferred when uploading, these latter findings may have been obscured by measurement error and hence not give an accurate view of the available bandwidth.

**CPU performance** Figure 1.2 shows the CPU performance distribution where 80% of the hosts have between 0.750 and 2 gigaflops/s, roughly corresponding to 1–3 GHz processors. The average is 1.186 gigaflops/s.

**Memory** The memory distribution in Figure 1.3 shows the most common configuration to be 512 MB memory, with 256 and 1024 MB memory configurations taking almost 20% each. It is worth noting that almost 10% have 128 MB total memory or less.

Since LHC@home is only available on x86 processors, we were not able to collect statistics about different architectures. Instead we examined the public available host files for SETI@Home and discovered that x86 constitutes 97.5% with 173 642 hosts, PowerPC 1.9% with 3362 hosts, Sparc 0.4% with 747 hosts and other architectures 0.2% with 462 hosts. We also examined the the operating system distribution. Various versions of Microsoft Windows constitutes 89.2% of the hosts, Linux 8.3%, MacOS X 1.9% and the rest, all Unix variants, 0.6%.

Overall one can conclude that the machines available to a distributed project have very fast processors, a fair amount of memory available, reasonable downstream connections but limited upstream connections. Also, it may be profitable to differentiate between clients with respect to memory and bandwidth. For instance, one could distribute jobs with high memory requirements to the 60% of the clients with 512 MB memory or more. Or similarly distribute jobs with large amounts of data to the 45% of the clients with more downstream bandwidth than 100 kB/s. The extent to which this abundance of resources can be used of course depends on what the users are willing to accept.

### 1.2.2 Analysis of Jobs

We also analysed the jobs for LHC@home. The characteristics of the jobs are interesting because they may exhibit heterogeneity that it would be beneficial

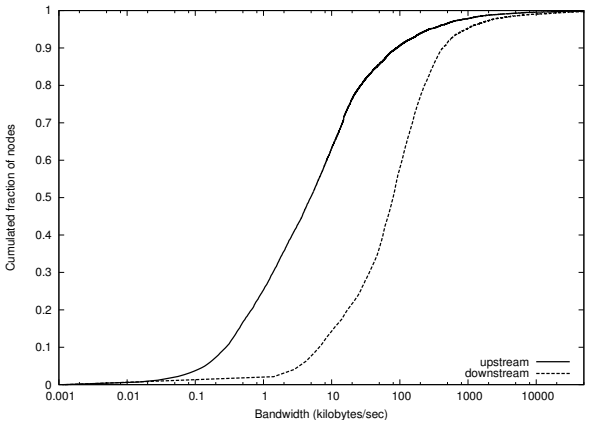


*Figure 1.1: The downstream and upstream bandwidth distributions.*

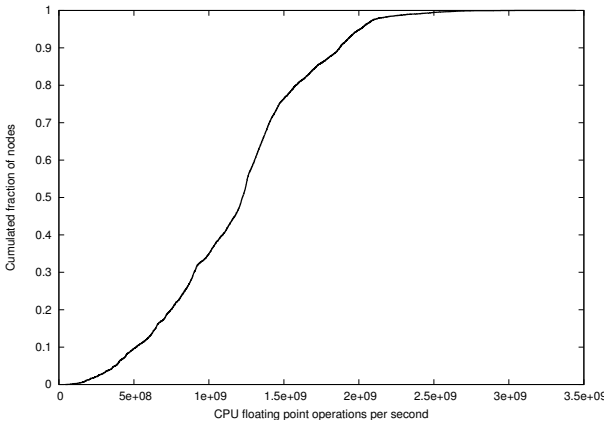


*Figure 1.2: The distribution of CPU performance in floating point operations per second.*

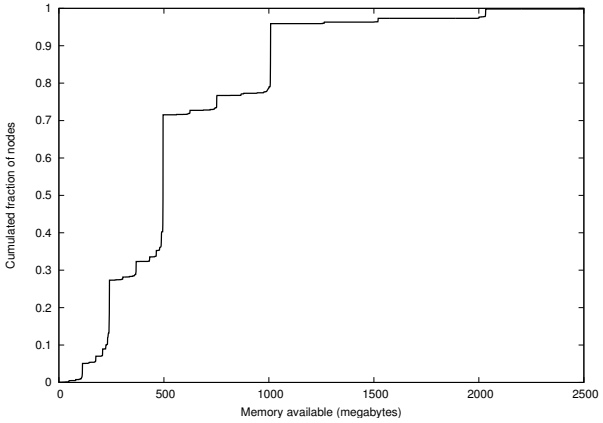


*Figure 1.3: The memory distribution of the hosts.*

to take into account when distributing them. The data had 615 444 distinct jobs – with redundancy and failure handling this resulted in 2 524 177 workunits being sent out.

**Bandwidth requirements** Out of the 615 444 jobs, 95% required downloading a 277 kB file whereas the remaining 5% required downloading a 390 kB file. The upload files were all bounded by 102 kB.

**CPU time** With respect to CPU time, 92% of the sent workunits had an estimate of 6000 gigaflops (corresponding to 84.3 minutes with the host average of 1.186 gigaflops/s) and took on average 30 minutes to complete. The remaining 8% had an estimate of 60 000 gigaflops (corresponding to 14 hours and 3 minutes) and took on average 3 hours and 15 minutes.

A plot of the distribution of CPU times is shown in Figure 1.4. The plot shows that 41% of the workunits are completed within 10 minutes.

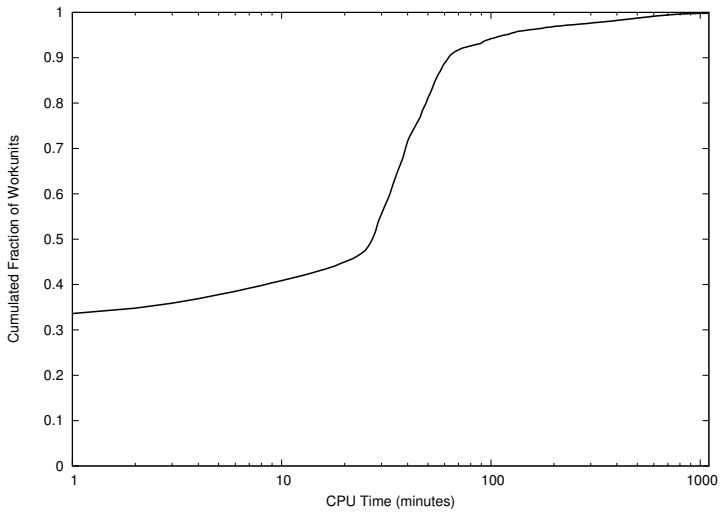


*Figure 1.4: Cumulative distribution of the CPU time spent on workunits.*

**Memory requirements** 51% of the jobs had an upper bound of 60 MB memory. The remaining 49% had an upper bound of 100 MB.

**Disk space** No further disk space is used beside the client itself (about 4 MB) and the downloaded job data.

We can conclude from these data that the jobs are homogeneous with respect to the bandwidth, memory and disk space requirements. The difference is in the spent CPU time where some jobs take considerably more time than others.

We also examined the failure rate of the workunits. 80% of the workunits were completed correctly. 2.8% did not get any further than the initial state – almost all of them are assigned to hosts and returned, but for some reason never

get any further. 5.6% failed due to a client error; it mostly seems to be a problem with bad downloads. 11.3% timed out before getting a reply back from the client. 2.4% were rejected because the job that the workunit corresponded to had already had enough results in to be accepted. None failed due to validation errors. The relatively high error rate suggest that it might be necessary to overbook some workunits if a deadline is to be met.

The above discussion gives the impression that most mass computing jobs are small and completed quickly. Since the jobs are specific to each project, we examined some of the other projects to see whether this holds. For SETI@Home the data appears to be of the same size, but the processing time for the workunits is about 6 hours. For Folding@home the jobs may take at least 14 days to complete and require downloading about one megabyte. For ClimatePrediction.net the workunits have processing times up to a month and require downloading a couple of megabytes.

### 1.2.3 Server Load

Our communications with the maintainers of the LHC@home project revealed that server load was sometimes a problem, in particular with the disk system because of the number of database requests. To count these, we have used the approximation of counting when a workunit is sent and when it is received – both events are registered in the database. Note that this is a lower bound on the total number of database request; for instance, we are not counting maintenance requests and unsuccessful client requests that do not find a suitable workunit.

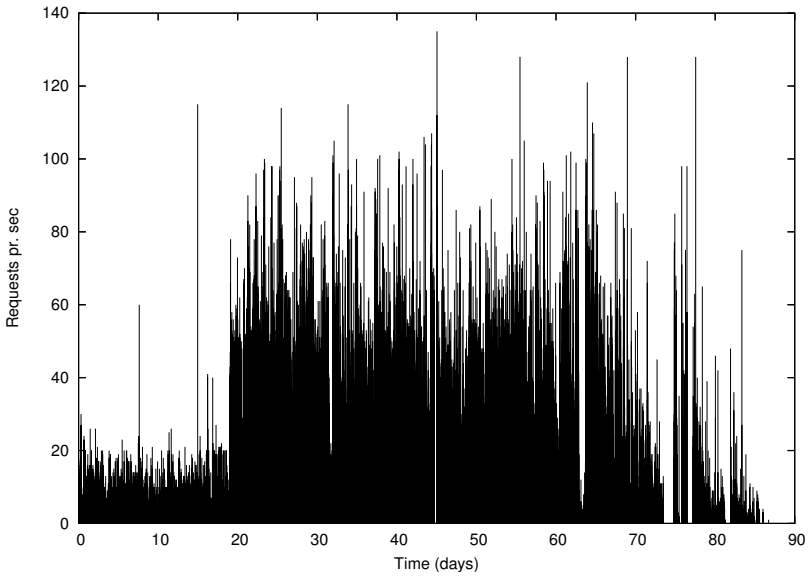


*Figure 1.5: The number of workunits sent and received per second.*

Figure 1.5 shows the number of workunits taken and completed per second over time from the start of the project. The first twenty days are relatively quiet followed by two months with substantially higher peak load. The increase of requests is partly due to an increase of users (1000 to 6000) and partly because the average CPU time needed to complete a workunit is halved. The short periods with much fewer or even no requests are due to the project running out of workunits. On average the server receives 0.6 requests per second, but as the figure illustrates peak situations burden the server with up to 135 requests per second.

From day 40 to day 50 where the spike of 135 requests per second is located (the average is here 0.9 and the standard deviation 3.7), the number of active hosts is 8181. Since the objective of mass computing in general is to exploit large numbers of clients, the 8181 hosts is a relatively small number and even in that setting, server load seems to be a severe problem. With peak loads of at least 135 requests per second the database server will clearly have a problem scaling to more users.

## 1.3 Project Motivation and Goals

Having introduced what encompasses mass computing, we will now present the idea of this project.

Part of our motivation comes from the costs of the centralised architecture that the current mass computing projects are operating with. Even a relatively small project such as LHC@home requires an expensive and powerful server to cope with the load of the job distribution. One way to alleviate this problem is to add more servers that each distribute their own set of jobs – e.g. Folding@Home has more than 20 servers. But this increases the costs of running a project since the extra hardware must be bought and maintained.

As an alternative, this project explores a decentralised architecture based on the recent research in peer-to-peer designs. Some of the issues that a mass computing system encounters are [1]:

1. Job distribution through a job pool with job selection. Issues that may complicate the job selection algorithm:
    - Mapping heterogeneous jobs to heterogeneous clients.
    - Impeding malicious users by enforcing redundant results.
    - Exploiting data locality if clients can reuse job data.
    - Ensuring that jobs do not starve indefinitely.
2. Management of input and output data:
    - Distributing job data.
    - Uploading job results.

3. Providing user incentives:
    - Creating an environment for community-building.
    - Maintaining low degree of intrusion on client systems.
    - Supporting a fair credit system (trickling may be necessary).
4. Assisting with application-specific project implementation:
    - Splitting a problem into separate jobs.
    - Checking incoming results.
    - Piecing valid results together.

Of these issues, due to the time frame of our project, we restrain it to cover only job distribution and selection. In doing so, however, we must address client system intrusion and also support a credit system in order to have a viable solution. Data management, community web site management and a framework for application building are on the other hand largely orthogonal issues.

Our basic idea is to setup a decentralised network of voluntary client machines and let them manage the job pool in cooperation. Different mass computing projects can submit jobs to the shared pool and later harvest the results. An index is maintained to enable the clients to find appropriate jobs efficiently. Asymmetric cryptography is used for authentication of projects as well as clients to avoid abuse and ensure that credit can be assigned.

An immediate disadvantage of this sketched decentralised system is loss of control compared to a centralised solution where the servers are maintained in the organisation running the project. Also, it is not trivial to design such a system since network and node failures as well as security issues must be addressed. But in return, we get a great abundance of resources.

These resources can be used to relieve the requirements for running a mass computing project. With our idea, the project machine is just submitting jobs to the job pool and does not have to be available on the network all the time or be able to handle the massive peak load found in Section 1.2.3. Hence an ordinary workstation might suffice instead of a powerful server with its associated maintenance. This makes it cheaper and also easier to deploy new projects once the shared job pool network is established.

With the extra resources from the decentralised system we can also afford to add in redundancy so that the progress of a project does not depend on any single component. And with a decentralised peer-to-peer network comes far greater scalability since the available resources grow with the number of participants. SETI@Home currently has about 400 000 active users – scaling to a greater part of the entire Internet population such as 4 000 000 users may require ten times the server hardware.

Furthermore, although we only examine the case of building the job distribution design on top of a decentralised infrastructure, having this infrastructure with its associated resources in place makes it possible to change more of the work flow in the future.

For instance, the decentralised network could be modified to also manage the input and output data for the jobs so that the data does not have to pass through a central server – apparently, storing the data at the clients is already being considered with BOINC [1]. If redundant results are being computed, the upstream bandwidth requirements for a project file can be reduced by at least about 2/3 by distributing the data once to the network which then takes care of serving it multiple times to the clients.

Our last motivating factor also has to do with future prospects. As a part of the preliminaries for this project, we designed in [12] another job distribution system intended for computational grids, i.e. grids that connect computing clusters into large batch systems. In an attempt to be flexible enough to support more than just job distribution and enable tight control, that system ended up being clearly unsuitable for mass computing projects because of the scale involved. We believe the same can be said for most other existing grid systems, in particular those based on the Globus Toolkit [25].

But in some sense, mass computing *is* grid computing – on a much larger scale but with simpler organisational concerns. By shifting the focus of our design efforts to cooperation on the large scale we believe it is possible something useful for computational grids can also turn up.

Hence, the goal of our project is to build a system that enables us to answer the following questions:

- Is a decentralised architecture competitive with the centralised architecture? Can we exploit the prospects offered without at the same time:
  - Introducing too much overhead?
  - Reducing the possible uses of the system?
  - Being severely affected by the failures of the large number of individual components?
  - Compromising security through the loss of control?

  In order to answer these questions, we must in the design address *intrusiveness*, e.g. by avoiding overloading single nodes; *scalability*, to avoid reducing the use of the system to only small projects; *robustness*, to enable the system to continue seamlessly in spite of frequent node failures as a prerequisite for being able to scale; *flexibility*, to be able to accommodate the needs of future projects; and *security*.

- Is a decentralised design addressing the needs of mass computing a sensible foundation for computational grids?

## 1.4 Project Overview

Having introduced mass computing and the motivation and goals of our project, we will now provide an overview of the rest of our work.

In order to be able to build on the latest advances in peer-to-peer techniques, we have reviewed a number of articles. A summary of the relevant techniques we have found are provided in Chapter 2.

We have then come up with a decentralised design based on these techniques. The design is discussed in Chapter 3. It handles multiple mass computing projects submitting jobs, efficient job search with keywords to support heterogeneity, clients claiming jobs and sending in the results, and indexing of the finished jobs for efficient retrieval by the projects; all in a decentralised manner. The amalgamation of techniques needed for a scalable and robust mass computing system is our main contribution.

In order to evaluate the design, we have implemented it in a simulation framework and conducted tests to explore its intrusiveness, its scalability and its ability to handle failures of the participants. The implementation and results of the tests are described in Chapter 4.

Finally, in Chapter 5 we summarise the project, state our conclusions and present some issues for future work.

## 1.5 Related Work

Decentralised job distribution systems have been proposed before. One example is Messor [42] in which every node in the network can submit a job. The jobs move around on the network according to simple principles that collectively results in good load balancing. A related example is the Organic Grid [11] where the job submitter becomes the root of a dynamic tree as other nodes gradually request work from it. Both projects are phrased in terms of mobile agents. Another example of a job distribution system is Jalapeno [58] in which self-organised managers distribute the submitted jobs through ad-hoc hierarchies.

Our project differs from these systems in our choice of architecture which we base on structured overlay network techniques and indexing. Without the structured techniques, it is not clear whether the other projects can provide enough control over where jobs are directed, e.g. in the situation that everyone is not both equally capable of and willing to work on all tasks. Another difference is that these systems apparently cater to the idea of everyone submitting jobs, although the security implications of this is unclear, whereas we only consider a selected few job submitters.

There are also several examples of using a pull model of job selection in which each resource on its own pulls a job from a job pool when it needs something to do. BOINC [1] and other mass computing architectures, e.g. the sys-

tems running Folding@home [54], GIMPS [24] and distributed.net [19] do this. Other more grid-like systems also use the pull model. One example is DIRAC (Distributed Infrastructure with Remote Agent Control) [59] where clients submit jobs to a central job server and agents running on the computational resources request jobs from it. Our project is similar to these in the conceptual model, but again differs in the architecture where we aim at decentralising the job pool.

CCOF (Cluster Computing on the Fly) [38] is an example of the reverse situation where the architecture is decentralised but a push model with registration of resources and an explicit scheduler is used. Centralised architectures based on the push model are commonly used for grids for connecting clusters, e.g. NorduGrid ARC (Advanced Resource Connector) [20], Teragrid [10] and UNICORE (Uniform Interface to Computing Resources) [60].

The advantage of the push model is the ability to optimise the usage of the grid globally, e.g. with respect to response time or overall grid throughput. The foremost disadvantage is limited scalability. Alone the problem of centrally maintaining up-to-date information about all grid participants is too expensive for large grids such as a mass computing project. This is our reason for differing from these systems by not using the push model.

Finally, there are several other decentralised systems that use indexing on a decentralised structured overlay network for efficient retrieval of items. In [49] an indexing system for document retrieval is presented; an issue here is how to extract keywords from the documents. KSS (Keyword-Set Search System) [26] is a similar system where an MP3 file-sharing application is considered. Arpeggio [14] is a complete file-sharing network design that supports keyword searching; the main difference from our application domain is that Arpeggio is dealing with less dynamic contents, but much larger amounts of data.

# Chapter 2

# Peer-to-Peer Techniques

This chapter reviews some of the recently invented techniques for peer-to-peer networks. We first present the idea of structured overlay networks, then review approaches to indexing and load balancing.

## 2.1 Structured Overlay Networks

The original decentralised peer-to-peer networks for file-sharing impose no particular structure in the way the nodes in the network are organised [13; 27], except for limiting the number of neighbours to each node. The resulting networks may have good average performance in practice because of naturally arising small-worlds clustering as is the case in Freenet [13]. But in the worst case, locating an uncommon piece of information may require searching all of the nodes in the network. This severely affects the latency of operations and hampers the scalability of the network.

Instead, much research in decentralised peer-to-peer networks in the recent years has focused on what one can think of as distributed data structure design. The idea is to setup a data structure in a distributed fashion to enable efficient queries with guaranteed good worst-case performance. The data structure is partitioned and distributed to the nodes in the network so that each node only has a partial view of the complete structure. This is necessary to be able to scale to an Internet-wide network.

The different parts of the distributed data structure is traversed by routing messages from node to node in the overlay network until they arrive at the destination. As with ordinary data structures, the structure is intended to reduce the number of locations that must be visited. Node failures will happen frequently in a large overlay network consisting of ordinary computers, so redundant routing information in each node must be maintained as a background overhead to ensure robustness.

There are many suggestions of how to structure the overlay network. The

Internet DNS is an early example of a hierarchical structure where the nodes are arranged in a tree. The tree guarantees logarithmic lookup, but does not divide the responsibility for maintaining the data structure evenly between the nodes since all requests in case of cache misses has to go through the root. Instead, recent research on structured overlay networks has focused on other data structures that are not based on hierarchically partitioned trees. One example is SkipNet [31] which is based on a circular skip list [45].

### 2.1.1 Distributed Hash Tables

To ensure that the load in the network is balanced, most designs are, however, based on hash tables [30; 32; 39; 48; 52; 57; 63] with a simple *put(key, item)* and *item get(key)* interface where the network nodes act as buckets.

Excessive movement of data items when nodes join and leave the network is avoided by using consistent hashing [33]. The idea in consistent hashing is to hash both data item keys and node identifiers, and place each data item on the node that it is closest to for some notion of distance in the space of hash values. This means that a joining or leaving node has a localised impact on the network as data items only has to be transferred to or from its neighbours.

Although hashing potentially yields $O(1)$ network hops for looking up a data item, this requires knowledge of many nodes in the network which is costly for a large dynamic network. To be able to scale to Internet-wide networks, most designs settle for $O(\log n)$ lookup network hops in return for only having to maintain $O(\log n)$ space at each node, with $n$ being the total number of nodes.

The individual designs differ in what kind of network topology they employ. Usually, a node maintains knowledge about nodes in its vicinity and a few selected nodes farther away. The former keeps the network together whereas the latter means that messages can be routed over a few long-distance hops to get close to even far-away destinations. For instance, Chord [57] arranges the nodes in a ring with extra pointers $\frac{1}{2}$-way, $\frac{1}{4}$-way, $\frac{1}{8}$-way, etc., through the ring for each node. With these pointers the distance to any destination can at least be halved at each step of the routing algorithm, which results in a logarithmic number of network hops in the worst case.

Another idea comes from prefix-based routing, used by Tapestry [63] and Pastry [52], where the routing is based on matching the prefix of the destination address. Kademlia [39] has a similar topology but phrases it in terms of using XOR to calculate the distances between hash values.

CAN [48] arranges nodes in a $d$-dimensional space with wrap-around and routes via neighbours. Koorde [32] utilises de Bruijn graphs to achieve either $O(\log n)$ network hops with $O(1)$ routing state or $O(\log n / \log\log n)$ network hops with $O(\log n)$ routing state.

Kelips [30] is an example of an almost one-hop network that achieves $O(1)$

network hops with $O(\sqrt{n})$ routing state by dividing the nodes into groups and having each node in a group maintain knowledge through gossipping about all other members of that group plus a few members from each of the other groups.

### 2.1.2 Practical Issues

Any of the structured overlay network designs mentioned (there are more in the literature) provides a scalable and robust routing foundation. Whereas the scalability guarantees stem from their structure, the robustness is achieved by keeping redundant routing state and updating it periodically to ensure that failed neighbours can be routed around. The dynamic self-repair mechanisms also help making the networks self-organising so that they can grow and shrink automatically without human intervention. Furthermore, the basic mechanisms are quite simple so implementations of the designs tend to be quite small.

Thus the structured overlay networks have some properties that are very desirable in practice. Some remaining practical issues must be addressed, however.

#### Data Loss

Even though a structured overlay network provides a robust routing foundation, the data stored in the network must also be maintained in a fault-tolerant manner. This is solved by introducing redundancy which is maintained periodically by what is usually referred to as republishing. There are usually two choices for republishing: either the creator of the data keeps republishing it, or the part of the network where the data is placed assumes responsibility.

There are also two choices of redundancy. Either a number of replicas, say $r$, are stored in the network. This scheme can tolerate that up to $r - 1$ nodes fail within a republishing interval without losing data. Or one can use an erasure-coding scheme [40]. The erasure-coding scheme works by splitting the data into $n$ blocks and generating $m > n$ blocks from the $n$ blocks with an erasure code. The $m$ blocks are then stored in the network, and because of the special properties of the erasure code, it will suffice to retrieve any $(1 + \epsilon)n$ of the blocks, where $\epsilon$ is a small constant that represents a trade-off between encoding/decoding time and space overhead.

Replication is simple and gives slightly better average read latency than erasure coding since the data can be retrieved from the fastest node of $r$ nodes instead of the $(1 + \epsilon)n$ fastest nodes of $m$ nodes [17].

On the other hand, erasure coding provides vastly better fault-tolerance with the same space usage because the data is distributed over more nodes. Hence, if the availability expectation is kept constant, erasure coding can save

much space and much bandwidth for the data writers when dealing with large amounts of data.

We do not expect our job distribution system to deal with large amounts of data since we do not address data management. Hence we prefer replication.

### Latency

Another problem with the structured overlay network designs is that the intermediate network hops in the routing path imposes a considerable latency. To address this problem, network locality must be taken into account. One possible solution is to adapt to the network by modifying the structure in the network; Coral [22] is an example of such a system.

Another possibility is to adapt to the physical network within the limits of the existing structure. There are two separate issues [17] – maintaining low-latency query routes, referred to as proximity neighbour selection, and selecting a low-latency server to download the data from.

The idea in proximity neighbour selection is to examine several nodes and pick the one with the lowest latency whenever the network structure permits a choice, which it usually does for the far-away destinations. This simple idea works surprisingly well in practice and can on average reduce the communication latency to a small constant factor (e.g. about 1.5 for Chord [17]) within the limit of the direct IP route between two nodes. For the idea to work, it is important that the network structure is flexible in what neighbours it permits [28].

Since our job distribution system is essentially a large batch system, the latency of the operations is a minor issue. Hence, there is no need to use an overlay network with a structure that is specifically tuned for low latency.

### Churn

A very practical problem with peer-to-peer networks is churn, that nodes join and leave continuously. For the pioneering file-sharing networks like Gnutella and Napster, the mean session time for a node has been measured to be under half an hour [53]. Although the implications of this result has been disputed by measurements [4] on a later file-sharing network, Overnet, the conclusion is still that many file-sharing nodes temporarily join and leave the network several times a day.

Since other data [39] indicate that long-lived nodes are likely to stay up for even longer, one possible solution to the problem of churn is to only include in the network itself a selected subset of stable hosts. The rest of the hosts act as clients.

Otherwise if every peer is considered equal, there are two distinct problems that must be solved. The first is to maintain a sound routing foundation with reasonable lookup latency and a well-connected network in spite of frequent

joins and leaves. It has been shown [50] that a careful choice of network maintenance protocols combined with dynamically adjusted timeout periods can solve these issues with a low background bandwidth overhead.

The second problem is to prevent data loss and ensure availability of the data stored in the network. With high churn, it is necessary with a large degree of redundancy to be able to ensure availability if all nodes are treated equally. Some of the redundancy can be eliminated by monitoring the node availability and adjusting the redundancy dynamically for each data item [5]. But if the overlay network includes many low-bandwidth nodes, e.g. modem users, and maintains large amounts of data, another architecture than a simple hash table design treating every node more or less equally may be needed.

Since we are not addressing data management, our system will only deal with small amounts of data, so churn should be a minor problem as long as the distributed hash table implementation is robust enough.

#### Network Partitions

If a part of the physical network infrastructure fails, some nodes in the overlay network may not be able to contact the other nodes. This effectively partitions the network. The self-repairing mechanisms will repair the structure of each part of the network so that after some intermittent query failures two separate overlay networks will be running in parallel.

Since the nodes from these two networks cannot communicate with each other, they may create conflicting data in their respective networks. This data then has to be reconciled when connectivity is established again.

One solution for this problem is to design the application on top of the network so that this reconciliation is possible to do automatically without loss of information. This limits the kind of systems that the overlay network can support, though.

Another approach is to try to detect network partitions and stop executing on the nodes in the smallest part of the network while a partition is occurring. Unfortunately, this approach still seems to be an open problem in the literature. For it to be feasible, network partitions must be detectable reliably without security risks. Since we do not intend our project to explore these detection issues, we design our system to support automatic reconciliation.

#### Security

Even though structured overlay networks can cope with failing node and network partitions, some problems remain if the overlay network includes malicious nodes. We will detail solutions to some of the problems in this section based on the analysis in [9]. Apart from denial of service assaults, possible attacks include:

1. Running many nodes with carefully selected identifiers to obtain a large part of the hash space. This can be used to isolate some nodes from the rest of the network nodes or to gain control of all replicas of a data item and cause data loss or corruption.

2. Inserting adversaries into the routing tables of other nodes when they join or refresh their routing tables.

3. Misrouting, corrupting or dropping messages to make lookups fail or return the wrong data items.

The node identifier assignment problem can be solved by using certificate authorities to grant access to the overlay network. The certificate authorities can ensure that it is not possible to assume an arbitrary identity. They can also add the overhead necessary to ensure that it is not possible to run so many nodes that the network is overtaken, e.g. by requiring new nodes to solve a computationally hard problem.

The problem of routing table maintenance is mainly a problem because most overlay network design have loose constraints on which nodes are accepted for the routing tables to allow latency optimisations. A possible solution is to maintain two routing tables: a latency-optimised table with loose constraints for normal usage and a restricted fall-back table that only allows certain nodes according to the structure of the network.

To solve the last issue of incorrect message forwarding, a timer can be used to detect dropped messages and the message be resend via another route. Misrouted or corrupted messages can be detected if the data returned is signed or self-certifying [41]. In the case of retrieving data items that are replicated, multiple nodes can also be contacted to obtain a quorum decision.

## 2.2 Indexing Techniques

With a basic distributed hash table it is only possible to retrieve a data item if the key for the data item is known beforehand. More complex queries or queries with only partial knowledge can only be satisfied by flooding the network with requests until the desired data is found, which ultimately can involve contacting every node in the system.

Since the flooding approach does not scale well and since searching is essential to support selecting appropriate jobs for our application, this section reviews some approaches to indexing the data items of a distributed hash table. The index is a data structure that provides an efficient mapping from an index key to data item keys through index entries of the form (*index key*, *data item key*). It is easy to setup and maintain an index with a central server, but since a centralised solution will not scale as the network grows, it is desirable to distribute the index among the participating nodes.

### 2.2.1 Keyword-Based Indexing

One way to index the data items is to use keywords. A lookup in the index for a keyword will then return all data items that are associated with that keyword. Such an index can be mapped to a distributed hash table by storing the index entries for each keyword under the hashed value of that keyword. Queries that involve more than one keyword are more difficult to deal with, however.

#### Indexing with Distributed Joins

The work of [49] focuses on optimising queries with multiple keywords when the index entries for each keyword are stored under that keyword only in the distributed hash tables. The improvements are achieved by incorporating a variety of techniques such as Bloom filters, caching and incremental results.

A search involving multiple keywords, such as $a \wedge b$, is performed by forwarding the query to a node that is responsible for one of the keywords, say $A$. Node $A$ locally finds the index entries containing the keyword it is responsible for and forwards them to node $B$ together with the original query. Node $B$ then computes the final result based on the data from $A$ and the index entries it is responsible for itself. If the final computation of the result was done at the client instead of by $B$, all index entries matching the keyword $B$ is responsible for would have to be transferred to the client instead of just those matching $a \wedge b$.

However, the set of index entries sent from $A$ to $B$ may be quite large. The amount of data transferred can be reduced considerably by sending a Bloom filter instead of the set itself. A Bloom filter is a hash-based data structure that summarises set membership. The summary compresses the set at the cost of accuracy since some elements may be falsely reported as belonging to the set – but false negatives do not occur.

The efficiency is further improved by caching Bloom filters and by returning incremental results. The idea of incremental results is to compute and send the results gradually instead of all at once since often only part of the results are needed.

#### Indexing with Local Joins

Arpeggio [14] is a file-sharing system built upon a structured overlay network with an indexing scheme similar to the one described. The main difference is that for each indexed file, all metadata keywords are attached to all index entries. This consumes more space but allows the index nodes to answer complex queries locally without having to do distributed joins.

In [26] it is proposed to group multiple keywords and use that as the key for some index entries. For instance, the keywords $a, b, c$ results in the combinations $a, b, c, ab, ac, bc, abc$. This effectively precomputes the joins. With more

knowledge of the application domain it is possible to reduce the number of combinations by only grouping keywords that are frequently searched in conjunction.

### 2.2.2 Hierarchical Indexing

An alternative to flat keyword indexing is to build a hierarchical index. In [23] the approach is to use query strings as keys in the index and let general queries point to more specific ones until the leaf level of actual data pointers is reached.

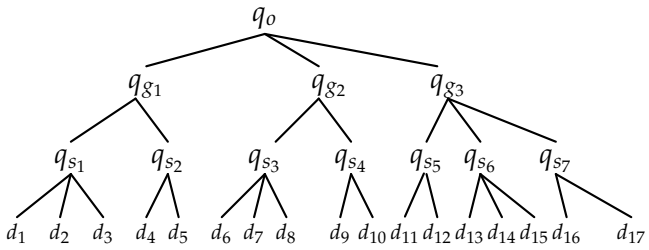


*Figure 2.1: An example of a hierarchical index. Pointers near the leaves are more specific.*

Consider the example in Figure 2.1. Here the overall query $q_o$ points to the general queries $q_{g_1}$, $q_{g_2}$, $q_{g_3}$ that in turn point to specific queries. The specific queries then point to the data items.

The query tree is mapped to the distributed hash table by hashing each internal node and placing the pointers originating at that internal node at the corresponding place in the hash space. When searching for data items, the traversal starts from the root and branches out towards the leaves until encountering the data items. For instance, with the index in Figure 2.1 the data item $d_9$ is found by looking up $g_o$ which gives $q_{g_1}$, $q_{g_2}$ and $q_{g_3}$, selecting $q_{g_2}$ which gives $q_{s_3}$ and $q_{s_4}$ and then selecting $q_{s_4}$ to find $d_9$ and $d_{10}$, i.e. three hash table lookups for the index itself.

The hierarchical index is flexible in that it is possible to perform complex queries still using the same tree-traversal search operation. However, in practice it is not possible to store all conceivable query strings in the index and furthermore a deep index will delay queries. The latter problem can partly be overcome by inserting shortcuts to data items for popular lookups higher in the index.

### 2.2.3 Conclusions

We expect that the keywords needed to index jobs are so few that it is possible to attach them to the index entries and use local joining, as with the file-sharing application of Arpeggio. This is faster and much easier than using distributed joins or the hierarchical indexing technique.

We have not discussed how to index numeric values, e.g. to support range queries, but from the data obtained from the mass computing project in Section 1.2 it seems that a fine-grained index is not necessary. For instance, for memory the index could simply split the jobs up into requiring little memory (less than 128 MB), medium memory (between 128 MB and 512 MB) or lots of memory (more than 512 MB). Thus we will not go through possible ideas for supporting range queries, although several suggestions do exist [3; 6; 46].

A problem with all of the techniques is that multiple index entries end up at the same nodes in the network because they have the same hashed key. In order to be able to handle this problem, we will proceed by discussing some means of balancing the load.

## 2.3 Load Balancing

In a heterogeneous environment there are four factors that can lead to load imbalance in a distributed hash table. Uneven placement of nodes and data items in the hash space, both resulting in some nodes being responsible for more data items than others; heterogeneous node capacity where even with a uniform load some nodes cannot cope with it because they have less capacity than the average; and finally non-uniform data access in which a single data item key can be several times more popular than other keys.

The problems with uneven distribution of data items and heterogeneous hosts can be addressed with the same approach, whereas the last issue with hot spots requires different techniques.

### 2.3.1 Uneven Node Placement and Capacity

Even the original Chord paper [57] acknowledged the problem of uneven node placement since it is known that placing the nodes with uniform probability in the hash space results in an non-uniform distribution of the portion of the hash space that is assigned to each node – with high probability the assigned portion for a node may exceed the average by a factor of $O(\log(n))$, with $n$ being the number of nodes in the network.

A solution examined in [57] is for each node to run several virtual nodes. This helps somewhat with ensuring that nodes do not suffer too much from being assigned a larger part of the hash space since it is unlikely that all virtual nodes of a node will end up with large assignments.

A more drastic approach is, however, examined in [47] where protocols are suggested to allow heavily-loaded nodes to actively exchange virtual nodes with lightly-loaded nodes. This directly mitigates the problem of some nodes having too many data items since a host with too many data items can swap one of its heavily-loaded virtual nodes with a lightly-loaded virtual node from another host. Furthermore, to solve the problem of heterogeneous hosts, each

node determines the number of virtual nodes it maintains based on its capacity, e.g. nodes with twice the capacity have twice the number of virtual nodes.

One problem with virtual nodes is the maintenance cost associated with the extra nodes. In [35] it is suggested to consider a number of virtual placements in the hash space for each node and only maintain a node at the placement that balances the load the most. Then the overhead is reduced to monitoring the different placements instead of active participation – while maintaining the security measure that it is not possible for a node to arbitrarily select its position in the hash space.

There remains a problem, however, if the data items are not evenly distributed since many items could potentially end up in the part of the hash space belonging to a single node. Another algorithm is proposed in [35] where each node still maintains only one position, but periodically probes random other nodes to determine their load. If a heavily-loaded node finds a lighly-loaded node, or vice versa, the lightly-loaded node moves its position to that of the heavily-loaded node so that load of the latter is divided between the two.

Common to all of these load-balancing solutions is that there is an unavoidable cost associated with them, in particular with migration of nodes.

### 2.3.2 Hot Spots

Even if each node has an equal share of the data item keys, popular keys can skew the request rates so that some nodes become hot spots. Introducing virtual nodes or moving nodes around in the hash space cannot do away with this problem because the requests may be for a single key, and only one node will be responsible for that key. Consequently, in order to address hot spots effectively a solution must enable delegating the responsibility of one node to several other nodes.

The delegation has to be done in a dynamic fashion that takes the structure of the overlay network into account. For instance, although it is common to distribute replicas of a data item to the other nodes closest to it for fault-tolerance, this does not alleviate hot spots. Partly because a constant number of nodes cannot scale to arbitrary request rates, and partly because most query requests still have to go through the node closest to the key with most routing architectures.

Note that for our application, we expect hot spots to occur not because a single data item is very popular, but because many small index entries have the same key. A client request does not have to return all of the index entries, however, but at most a few since the client only needs a few jobs at a time.

Our application also has the property that each index entry will only be relevant for a limited duration of time, since an index entry will be irrelevant when the corresponding job has been found and claimed by a client. Hence, both the client query load and the load from having to store entries may be

very dynamic.

### Caching and Active Replication

The first family of solutions we consider are well-known in the literature (e.g. [17; 22; 39; 61]). The idea is to distribute replicas of popular data backwards along the incoming routing paths of requests to a hot spot.

A simple way to do this is through adaptive caching [17; 39]. Cache entries are dynamically created on the search paths in response to high load on the nodes caching the data items. Each node independently monitors the request rate for keys and decides to replicate based on this. Even a simple algorithm with a fixed rate limit works surprisingly well in practice [22].

An active replication strategy is proposed in [61]. The backward routing paths to the hot spot are viewed as a tree and replicas are maintained in this tree so that a front of nodes at the same level all have a replica as Figure 2.2 illustrates. The algorithm has been proved to result in equal load on all replicating nodes under some assumptions in a Pastry-like overlay network.

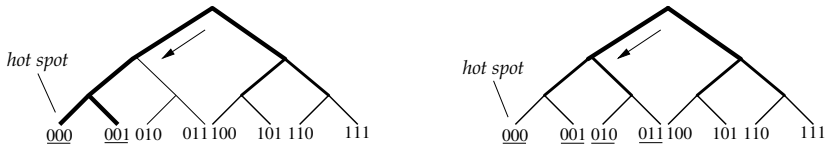


*Figure 2.2: To the left, the two underlined nodes store a replica. Since the load is too high, the data is replicated one step further up in the tree at twice the number of nodes, to the right, resulting in half the load on the two first nodes.*

These two proposals suggest replicating data items because they consider a file-sharing application where single files result in hot spots. In our situation the set of index entries must instead be split into disjoint subsets and shared between the nodes. For the nodes to share the entries, it is necessary with some amount of communication between them.

Hence, although the simple adaptive caching approach is attractive because it enables each node to make load-balancing decisions independently, it cannot readily be applied in our application because it does enable a replicating node to communicate with an arbitrary other node from the group of nodes that are sharing the load of the key.

The active replication strategy is better because the set of nodes is temporarily fixed with the current front, but it is still not obvious how to distribute the index entries dynamically.

### Multiple Hash Functions

Another approach to load balancing is to use multiple hash functions to determine where data items are replicated.

In [7] a small fixed number $d$ of hash functions are used. When writing a data item, its key is hashed to $d$ positions in the hash space and the least loaded node out of the $d$ is selected to store the value. It is suggested to store a pointer to the selected node at the other $d - 1$ nodes to be able to find the item with at most two lookups, but this is a bad idea for an index application since it just increases the load. Apparently, the technique is quite competitive compared to using virtual nodes for data item balancing, even when $d = 2$, but since the number of choices of nodes is limited to $d$ for a single key, the technique cannot scale to handle arbitrarily large hot spots.

In [62] a large number of hash functions is set aside, e.g. 10 000, and a dynamic number of these are used for each key based on the popularity of the key. The paper suggests replicating the data item to all placements specified by the used hash functions, which corresponds to sharing the index entries among them. The problem is then how to determine how many hash functions are in use at a given time for a particular data item, e.g. when reading or writing an index entry. A solution proposed in [62] is to use a binary search algorithm, appropriately randomised to avoid creating query hot spots, as illustrated in Figure 2.3.

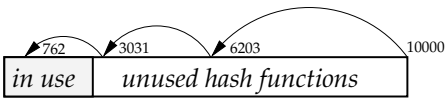


*Figure 2.3: The available hash functions, ordered by number. The randomised binary search begins by choosing a random number between 1-10000, namely 6203 which is an unused hash function. Then it chooses a random number between 1-6203, namely 3031, again unused, and a number between 1-3031, namely 762 which is in use and thus terminates the search.*

One can also imagine other solutions. For instance, the network could gossip about the current number of hash functions in use for heavily-loaded keys, e.g. using the algorithm in [16], or periodically broadcast the information using a broadcast algorithm for overlay networks, e.g. [8; 21; 44]. This makes the number of hash functions for each key global state, which scales linearly in the number of different load-balanced keys and hence is limited in the number of index keywords it can support.

The basic idea in using multiple hash functions is quite simple, and the technique spreads the load well. Unfortunately, the binary search algorithm introduces a logarithmic overhead which for a distributed hash table with $O(\log(n))$ lookups results in $O(\log^2(n))$ performance. The algorithm also requires compaction of the used hash functions from time to time, a somewhat complicated operation to do efficiently [62]. Compaction is required because some nodes in the middle of the used hash functions may crash or run out of useful index entries. Another disadvantage is that communication between the members of the group of nodes responsible for a key requires a separate

lookup for each member.

### Subgroups

In [34] an efficient subgroup concept is presented. It allows an arbitrary group of nodes in an overlay network to form a subgroup by supporting a $O(\log(n))$ subgroup-lookup operation that returns the next member in a subgroup after a given key. The presentation is in terms of Chord, but the idea appears to be applicable to other structured overlay networks as well. We can use it for dispersing hot spots by creating a subgroup for each popular key and vary the size of the group as a function of the key popularity.

The technique works by embedding in the Chord ring a virtual tree for each subgroup. The tree is constructed with the special property that the root node of any subtree $\tau$ is the same node as the root of the rightmost subtree of $\tau$, as shown in Figure 2.4. For example, $H$ is repeated down along the rightmost path in the whole tree and $D$ is repeated along the rightmost path in its subtree.

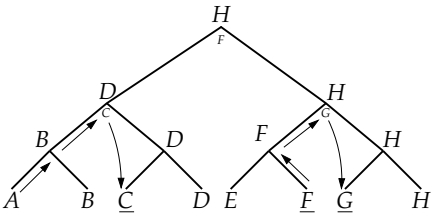


*Figure 2.4: A virtual tree for a subgroup. C, F and G are group members.*

The root of the whole tree is simply the node closest to the identifier of the subgroup. The edges in the virtual tree are mapped to the ordinary Chord fingers in practice so that traversing the tree is efficient.

The idea of the tree is that if the root node of a subtree $\tau$ stores a pointer to the leftmost subgroup member in the right subtree of $\tau$, then any node can find the member to the right of it by querying nodes on the path towards the root. For instance, node $A$ in Figure 2.4 finds the group member next to it by traversing the tree until it reaches $D$ which can route it directly to $C$. Similarly, node $F$ finds its successor by asking first itself and then $H$.

A node inserts itself in the subgroup by traversing the tree towards the root until reaching the appropriate node to place a pointer to itself at. In Figure 2.4, node $C$ has inserted itself at node $D$, node $F$ and $G$ have inserted themselves at different locations of node $H$. The insertion operation is performed periodically in case a node fails – this is the only thing needed to ensure fault tolerance since the structure of the tree is repaired by Chord itself.

The benefit of the virtual tree is that if random nodes issue lookups for a subgroup with $k$ randomly inserted nodes in the group, then the resulting query load will be roughly balanced among $\Omega(k)$ nodes in the system [34].

Hence, we can use the subgroup protocol for load balancing by distributing data items with a given key randomly to the members of a subgroup for that key. An advantage is that it allows control of which nodes participate in a group in contrast to the completely randomised approach of using multiple hash functions.

### Using Wildcards

SkipNet [31] has a concept of constrained load balancing where keys can be specified to be load balanced among a subset of nodes based on the URL. This idea has inspired us to another solution for alleviating hot spots.

The problem with the index entries is that they all are mapped to exactly the same key. But if we introduce some random noise in the low-order bits of this mapping so that they are mapped to almost, but not quite, the same keys, then they will not end up in precisely the same spot in the hash space but just close to each other. This means that the load-balancing algorithm presented in Section 2.3.1 that moves nodes around will work. For instance, it would be possible to move another node into a heavily-loaded area of the hash space and share the index entries, as illustrated in Figure 2.5.

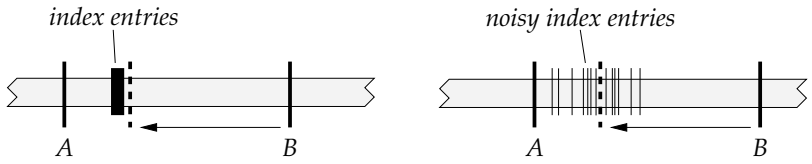


*Figure 2.5: To the left, A is responsible for a hot spot of index entries and moving node B does not help. To the right, some noise has been introduced in the index entries and moving node B now cuts the load on A in half.*

One way to express this idea is to replace a fixed number of bits in the last part of the data keys with a wildcard, e.g. the key 0101 becomes the key $01*$ meaning that it can mapped to any of the locations 0100, 0101, 0110 or 0111. To write a data item, the wildcard in the key is replaced with a random bit string and the data sent to the node closest to this key. Another node looking for data items with the key 0101 will similarly generate a random replacement for $01*$ and ask the node closest to the result.

This means that all nodes are not guaranteed to find a particular data item with only one lookup since they may contact the wrong nodes, but this is a minor problem as long as it is likely that other useful data items are found. A node that writes data items can always record the keys it has chosen and thus find the items immediately.

The result of this idea is more fine-grained addresses for nodes than for data items, as illustrated in Figure 2.6 for a circular hash space. Intuitively, it seems that this could make some data item keys clash and require flooding

local parts of the network to find the necessary data. But if we increase the length of the node locations instead of decreasing the significant length of the data item keys, then we have the same probability of clashes.

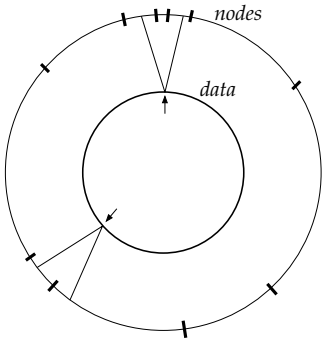


*Figure 2.6: One data item key maps to several node locations, which enables multiple nodes to share the load of a single key.*

Another related issue is that the difference in graininess between the node locations and data keys places a fixed limit on the number of nodes that can be responsible for a data key. But we can choose node locations to be so large that this is not an issue in practice. For instance, we could make node locations 64 bits larger than data keys, which ensures that $2^{64}$ nodes can assume responsibility of a single key.

The wildcard approach has the property that the nodes responsible for a key are next to each other in the hash space. The advantage of this is that any necessary communication between the members of the responsible group can be done via neighbour links, which is simple and cheap. The disadvantage is that a concentration of nodes in a small part of the hash space might skew the routing infrastructure enough to disturb its fault-tolerance and load-balancing guarantees. Whether this is a problem is an open question.

### Conclusions

Three of the four approaches we have identified, multiple hash functions, subgroups and data key wildcards, are immediately applicable. But the subgroup approach appears more attractive than using multiple hash functions since it allows more control and stays within $O(\log(n))$ message overhead for a distributed hash table with $O(\log(n))$ lookup cost. It does require more maintenance, though.

Our idea of using data key wildcards is simpler than the subgroup protocol and allows cheap group communication, but has a potential disadvantage of skewing the hash space. Whether this is a problem in practice has to be clarified through analysis or simulation.

# Chapter 3

# Design

This chapter describes the design of our system. We outline the architecture and the workflow and then proceed with a discussion of the job pool, how it is indexed, load balancing, assignment of credit, some security considerations and finally how the system handles network partitions.

## 3.1 Overview

With the techniques from Chapter 2, we will now outline the architecture of our system which we base on an overlay network. Numerous actors are involved, both inside and outside the overlay network. The actors inside the overlay network are the machines that keep the network going, store submitted jobs and maintain an index structure for efficient job searching. The actors outside the network are the mass computing project maintainers that submit and collect jobs, and the clients that run the jobs, as shown in Figure 3.1.

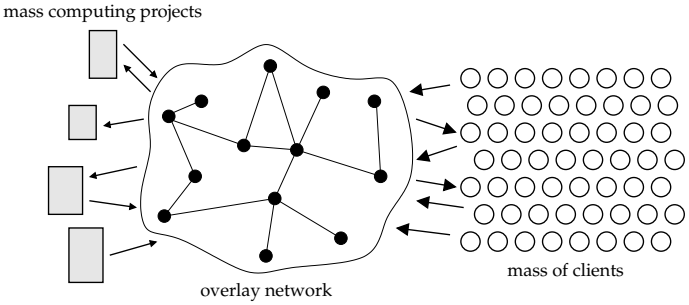


*Figure 3.1: The mass computing projects to the left submit jobs and retrieve the results from the overlay network that operates according to our design, and the clients to the right find jobs and compute and deliver the results.*

We propose using a distributed hash table to provide a basic key-value mapping infrastructure which we map the jobs onto in the following manner.

Each job must be described by a job specification with an associated unique random hash key. The nodes in the overlay network that are closest to the hash key are responsible for keeping track of the job. This entails storing and maintaining replicas of the job specification, replying to client requests for the job and keeping track of the job status and results.

The jobs also need to be indexed in order to enable the clients to find them efficiently. Each job specification should include a list of keywords to find the job with. The nodes storing a job should then maintain the index from this list by inserting the job specification hash together with the complete list of keywords as the value at the closest node for each hashed keyword. With the complete list of keywords available the nodes that store index entries can answer boolean queries using only local information.

Conceptually, the nodes closest to a job key assume the full responsibility of ensuring that the job is run and the results made available; they act as proxies on behalf of the mass computing project that has submitted the job. Nodes with index entries passively store the entries as soft state and answer queries, and it is up to the nodes storing the jobs to keep the index updated.

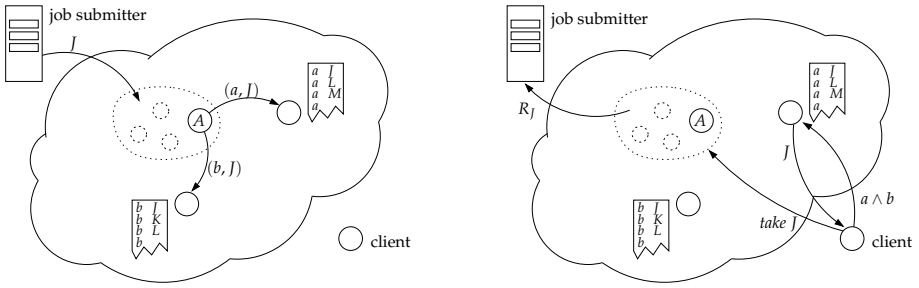


*Figure 3.2: The workflow of our design.*

With this architecture, the overall workflow is as illustrated in Figure 3.2. A job specification $J$ is sent by a mass computing project to the node $A$ closest to its key in the structured overlay and replicated to the neighbours of the node for fault-tolerance. Node $A$ then inserts index entries with the job keywords $a$ and $b$ in the overlay network.

Afterwards a client can lookup an index entry, use it to find the job and claim it. Upon receiving the request for the job from the client, node $A$ withdraws the index entries. The client retrieves any necessary job data and starts working on the job. When it has finished, it informs the nodes storing $J$ of the result $R_J$. Later, the mass computing project retrieves $R_J$ so that $A$ and its neighbours can eventually delete the job.

In the following sections, we will discuss issues arising from this architecture in detail, beginning with how the jobs themselves are handled. The overlay network nodes are participating out of goodwill, and we will assume them not to be malicious. We will, however, assume that they may be tempted to try

cheating in order to gain extra credits.

## 3.2 Job Pool

When designing the common job pool which is maintained by the nodes in the overlay network, we must decide how to distribute the jobs to the nodes, how clients claim jobs and how they submit the results again when they have computed them.

### 3.2.1 Job Distribution

The first issue to consider is how the mass computing projects should distribute their jobs. This is complicated by the requirement of having to support redundant computing to prevent clients from cheating.

If the nodes in the network can be trusted not to cheat then the closest nodes to a job can take care of running the necessary redundant versions of it on different clients, thus relieving the mass computing project from having to deal with this issue. A protocol is then needed for reaching consensus on when enough redundant results has been computed to form a quorum.

There are two problems with this approach, however. The first is that since the nodes are likely to run clients too, it does not seem reasonable to expect them not to cheat.

The second problem is that it makes it easier for ordinary clients to cheat by retrieving multiple versions of the same job and reusing the result computed for one version (or even a phony result) when submitting the results for the other versions to gain extra credits for free. Multiple clients must be colluding with each other to do this, since a single client will not be allowed to retrieve multiple version of the same job. The server in a centralised setting has some protection against such a conspiracy simply because of the sheer number of jobs – if the jobs are distributed randomly to clients, there is little chance of retrieving the same job multiple times. But we expect the number of jobs at a single node in our system to be relatively low, and hence the nodes to be more susceptible to the attack.

If the nodes in the network cannot be trusted to handle the redundancy aspect, which is the assumption of our design, then this aspect must be taken care of by the project itself. This means that it must distribute each redundant version of a job as a separate job itself. Each of these jobs is given a unique random identifier, and hence ends up at a random place in the network which makes it unlikely that clients can cheat, even if they are helped by a job node.

But before the mass computing project can actually submit each of these jobs, a job specification must be produced. A job specification consists of a description of how to run the job intended for the client software, references to

the required input files and a signature by the project that produced the job for proving the authenticity of the job.

The job specification is then duplicated to provide enough redundant results, and finally each copy is assigned a random key and submitted to a node in the network. The mass computing project should await that the job has been replicated to $r$ nodes in the network before considering the job submitted to prevent data loss in case of node failures. After this the fault-tolerance properties of the overlay network will keep the jobs safe for as long time as required.

Since we do not address data management, we will not describe how the input files are managed. But the input file references can be made flexible enough to support both storing the files on an external file server (e.g. via HTTP) and storing them directly in the overlay network if it is extended to support data storage.

### 3.2.2 Claiming Jobs

With the jobs safely submitted to the network, the next issue to consider is how to enable the clients to claim jobs so that they can run them without duplicating efforts. A node with a claimed job should refuse other clients to claim the job. Each of the nodes closest to a job must be informed of the claim so that the claim can survive node failures.

Merely sending the claim to all the nodes to collect positive and negative replies and then follow the decision of the majority is, however, not safe since another client may have a slightly different view of which nodes are closest to the job if a node has failed or joined recently. Thus in this erroneous situation two clients that are simultaneously trying to claim a job may both end up believing that they succeeded.

Instead we require that none of the replies must be negative. A client requesting a job orders the $r$ closest nodes by their distance to the job identifier, and starts requesting the job according to this ordering, one node at a time. If any of the $r$ closest nodes declines the claim, the client must abort and try to find another job.

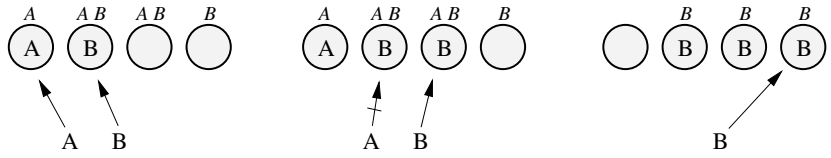


*Figure 3.3: Client A and B trying to claim a job at the same time with slightly different views of the closest nodes ($r = 3$). To the left, both A and B get one claim from the node they believe to be closest to the job. In the middle, A must abort because it encounters a node that has granted the job to B. B can, however, continue and claim the job on r nodes.*

The ordering prevents a situation where a job is never claimed because the

clients keep blocking each other from getting $r$ positive replies since the ordering ensures that a losing client has not contacted any of the nodes that the client it has lost to needs to contact, as shown in Figure 3.3.

Hence, as long as the clients agree on at least one common node that they believe to be one of the $r$ closest nodes to the job, the algorithm guarantees with few messages that there is exactly one client that wins. This is under an assumption of no failures. If a node fails, the client can simply skip it if it knows enough nodes to continue.

Guarding against client failures requires using timeouts on the nodes. When a node accepts a job claim, it should await a confirmation message from the client before making the claim permanent. The client sends the confirmation to each of the $r$ nodes after it has received a positive reply from all of them. Each message must include the signed and timestamped replies from the other $r - 1$ nodes to enable the nodes to verify that the client is not cheating. If the nodes do not receive the confirmation messages within a short time interval $T_{\text{claim}}$, they stop considering the job claimed.

There is a final minor point. In order to ensure that the algorithm is not impeded by minor inconsistencies, we allow a node that receives a claim for a job it does not store to respond with a "do not care" message. This allows a client to proceed even if a few of the $r$ nodes have not yet received the job, for instance because they have recently joined the network.

### 3.2.3 Submitting Results

When a client has finished computing the results for a job, it submits them to the $r$ closest nodes. The nodes must then store the results until they have been picked up by the mass computing project that submitted the job. To avoid data loss in case the project machine fails, the project should first retrieve the results from the node closest to the job, store it somewhere safe and then inform all $r$ nodes that the job has been collected. After some time $T_{\text{collect}}$ nodes can then delete the job. $T_{\text{collect}}$ needs to be large enough to ensure that the state of the job can overwrite other inconstant states as explained in the next section.

There is, however, ample room for the client to stop running the job or for some other reason fail to report the results back. Recall from the analysis of the LHC@home project in Section 1.2.2 that 11.3% of the jobs timed out before getting a reply back from the client. The nodes storing the job should detect this and allow another client to process the job.

They can to this by attaching a timeout $T_{\text{finish}}$ to a claimed job, initially set to a reasonable upper bound of the time it takes to run the job and submit the results. The timeout should be set individually for each job by the mass computing project, since job running times will vary. Once the job has timed out, the job is considered ready again.

### 3.2.4 Job Republishing

Fault-tolerance is assured by having the $r$ replicas of each job specification on the nodes that are closest to the job identifier. Each node in the network periodically contacts its neighbours to ensure that $r$ replicas are present even when nodes join and fail. This is all part of the normal operation of the distributed hash table, and the details depend on the particular overlay network used.

Apart from the job specification itself, a job also has an associated state on each replicating node; one of *ready*, *temporarily claimed* by client $x$, *claimed* by client $x$, *finished* when the results have been submitted and *collected* when the project has the retrieved the results.

The protocols described so far do not ensure that the replicating nodes all have a consistent view of what state the job is in. For instance, a short-lived node may join and receive a message instead of one of the other $r$ previously closest nodes before failing. Or clock drift may cause a node to cancel a claim before the other $r - 1$ nodes. The periodic replication algorithm should resolve these inconsistencies to avoid that they grow serious.

It can do this in the following manner. If the two states are different, then if one is *collected*, both should be *collected* since the project for sure has retrieved the results. Else if one state is *finished*, both should be *finished* as there is no reason to do more work when a result is available. Else if one state is *claimed*, it should overwrite the other state unless the difference can be attributed to skewed clocks – this might happen if one node cancels a claim before the other nodes, either because it got the claim a little before the other nodes or because its clock ticks faster. This case can be detected by examining the timeout; if there is less than $T_{\text{skew}}$ left, the state should not be overwritten.

The remaining case is if one state is *temporarily claimed*. Since this is just a temporary state used by the claiming algorithm, there is no reason to replicate it.

## 3.3 Indexing

In the previous section we described how to design the common job pool. However, the simple job pool is not enough because it does not support finding jobs without knowing the job identifiers beforehand. To solve this problem we embed a keyword index of the jobs in the overlay network. The benefits of having this index include:

1. It allows clients to efficiently find only jobs from a selected subset of the projects that use the job pool. Without an index, a client would have to resort to flooding or random probing which makes it costly to find jobs from small projects with few jobs.

2. Jobs can be selected based on indexed job attributes, which is a premise

for mapping heterogeneous jobs to heterogeneous clients.

3. Data locality can be exploited when dealing with large amounts of data. A client that has run a job on specific input data can cache these data and use the index to find other jobs that need the same data. Though this requires some extra infrastructure to ensure that the client can not misuse this to gain redundant copies of the same job.

Furthermore, we hope that the flexibility of an index infrastructure can help solve other future issues, e.g. it will also prove useful for projects when they are looking for finished jobs. The most important reason for having the index is, however, number 1 in the above list; without an index, the system would not be able to scale and still provide a good service for small projects.

We first present the basic design of the index, and then describe how keywords can be chosen, how to maintain the index entries and how queries are processed.

### 3.3.1 Basic Structure

The index structure used is the distributed keyword index with local joins as described in Section 2.2.1. An index entry consists of the primary keyword which acts as key, a list of secondary keywords to enable local joins and the identifier of the job that the entry is referring to as the value of the entry.

For each of the keywords for a job, the nodes that are closest to the job construct an index entry with the keyword as the primary keyword and the rest included as secondary keywords. For instance, a job with the keywords $a$, $b$ and $c$ results in an index entry with primary keyword $a$ and secondary keywords $b$ and $c$, an index entry with primary keyword $b$ and secondary keywords $a, c$ etc.

Each index entry is placed on the node which is closest to the hashed value of the primary keyword. The system should not replicate index entries. This is not necessary because they can be reconstructed with the information already available in the network in the job specification that they stem from. When there are no copies of an index entry, it is easier to retract the entry when it has outlived its usefulness, which will usually happen as soon as the entry has been retrieved.

### 3.3.2 Choosing Keywords

Each project that uses the system has to choose a set of keywords that describe the characteristics of the jobs of the project so that the clients can query for them. The choice of keywords should be made with care since the clients have to go through a keyword query for each job they wish to process. Although the choice depends highly on the kind of jobs submitted by a project, we will try to point out some general issues.

While analysing the mass computing project in Section 1.2 we identified a few keywords that seem plausible for many projects. These are *project* (e.g. "LHC" or "SETI") for distinguishing projects, *memory* (e.g. "<50MB", "50-100MB", ">100MB"), *bandwidth* (e.g. "modem", "cable") and *input data* (e.g. the URL of the data).

As mentioned, the *project* keyword allows several projects to coexist in our system since a client can search specifically for jobs from a particular project. The *project* keyword is also a good candidate for being grouped with the other keywords as mentioned in Section 2.2.1 so index entries from distinct projects are not mixed. For instance, if *project* is combined with *bandwidth* one could get the keywords "LHC:modem" and "LHC:cable". Grouping reduces the problem of hotspots to some extent, but makes it more difficult to select jobs across projects.

Once the set of keywords is decided, the clients participating in a project must be informed of them so that they can take them into account when generating queries. One way to do this is to include them in the distribution of the client software.

There is one final keyword which we have not discussed yet. When a client submits the results of a job, the project that has generated the job can collect it. With the design we have described so far, the project would have to periodically probe the nodes that are closest to the job. With many jobs, this is a major task in itself.

Instead the nodes that store the job can insert a special *finished* keyword (e.g. "LHC:finished") in the index. This allows the project to probe the index to discover whether any of its jobs have finished. The retrieved index entries then lead directly to the finished jobs. Note this use case is a bit different from when a client is looking for a job because the project is likely to want to retrieve all index entries as opposed to only one entry; the details of how this is possible depends on the characteristics of the load-balancing algorithm used as discussed in Section 3.4.

### 3.3.3 Index Maintenance

Having discussed what to index, we will now turn our attention to how the nodes storing a job should maintain the index entries of the job. To guard against failures, they must rewrite the index entries periodically.

One possibility is that the nodes all write the index entries. But this results in $r - 1$ redundant messages to the nodes storing the index entries. Instead each node should periodically consider whether to write the index entries, but only actually do so if it is the node closest to the job identifier. With no failures, only one node will then write the index entries, and if the closest node fails, the next node will quickly take over.

When a job is claimed, the nodes storing the job should also coordinate to

withdraw the index entries to avoid that further clients are directed to the job and waste messages trying to obtain a job that is not available.

Since index entries are not replicated, they will always stay on the same nodes that they were written to, unless those nodes fail. This means that the node which is closest to the job can simply store the addresses of the nodes it sent the index entries to and use these to contact them again instead of trying to find the entries with a lookup in the overlay network. In case the node closest to the job fails, the index entries will time out and be deleted automatically anyway.

### 3.3.4 Query Processing

When a node receives an index query, it goes through its index entries to find all that satisfies the query. Of these it must then select the requested number of entries – for a client looking for job, this would result in one entry and for a project this would result in all finished index entries. This selection is actually a scheduling decision since it determines which jobs are run first.

A simple algorithm is to choose the index entries for the oldest jobs first. This will ensure, at least approximately, that jobs are processed in a first-in, first-out manner so that no jobs remain in the system forever.

The problem with this simple solution is that when jobs are indexed with multiple keywords, the nodes that end up storing the corresponding index entries will hand them out with the same ordering. Thus if two clients retrieve index entries with different keywords simultaneously, it is likely that they find the same job so that one of them has to start over.

This problem can be amended by choosing a random entry from the oldest $l$ entries. The value of $l$ should be set dynamically depending on the rate $\rho$ of incoming queries and the average delay $d$ it takes before the clients have contacted the job nodes and these retracted the index entries. Both $\rho$ and $d$ can be observed locally at each node.

The value of $\rho d$ is an estimate of the number of index entries that the node will hand out before getting feedback and thus being in danger of choosing an already claimed job. Hence, $l$ should be considerably more than $\rho d$ to have enough different entries to choose from.

Although the resulting design is not strictly first-in, first-out, it still favors old jobs and should thus prevent jobs from getting stuck for long time.

There is one final issue when selecting index entries. A node storing index entries should give the clients some time to contact the nodes storing the jobs before serving the same index entries again. A simple way of doing this is to put a served entry in quarantine. If the client fails before contacting the nodes storing the job, the node with the index entry can reactivate the index entry after a timeout – otherwise, in the normal case the entry would be deleted by the nodes storing the job.

## 3.4 Load Balancing

The design described in the previous sections results in a system where the load from the jobs will be somewhat imbalanced because of uneven node placement. Furthermore, the few nodes that store index entries will become hotspots when the network and number of jobs grow. The latter is a scalability problem which must be dealt with.

Section 2.3 presented several techniques for load balancing. From the techniques that dealt with hotspots, we identified the two that seemed most suitable: an explicit subgroup protocol and data key wildcards.

In lack of evidence to which is better and lack of time to produce such evidence, we use the wildcard technique because it is simple to implement. It must be combined with an algorithm for placing nodes at arbitrary places to even out the load – we use the last algorithm described in Section 2.3.1 (the second algorithm in [35]) for this purpose.

With this approach, the key for an index entry is the hash of the primary keyword with some of the low-order bits replaced with a random bitstring. The added noise is used both by nodes writing and retrieving index entries. It is not necessary to add noise to job identifiers since they are selected randomly and thus are not expected to clash.

Each node periodically probes another random node for its load to find out whether the load differs by more than a fixed fraction $\varphi$, i.e. $L_x < \varphi L_y$ or $L_y < \varphi L_x$ where $L_x$ and $L_y$ is the load of the two nodes, so that one node should move.

In order to compute $L_x$ and $L_y$, the load from both jobs and index entries must be quantified. The way we do this is as

$$L_x = n_i w_{\text{index}} + n_j w_{\text{job}}$$

where $n_i$ is the number of index entries, $n_j$ is the number of jobs on node $x$ and $w_{\text{index}}$ and $w_{\text{job}}$ are weights that reflect the respective load of storing index entries and jobs.

Only the jobs that node $x$ is closest to should be counted in $n_j$ even though the node also stores replicas of other jobs. If these other job replicas were counted in the load, it would not be possible for another node to half the load of an overloaded node in most cases, since a single node can only relieve the responsibility of the jobs that the overloaded node is the $r$ closest to; it would still be among the closest nodes for the jobs that it before was the $r-1$, $r-2$, ..., closest to.

There is one problem that we have not discussed yet. When a project is looking for finished jobs, it retrieves multiple index entries and if the index entries are divided between several nodes, the project cannot fetch all entries with one request. Because of the noise, multiple requests are necessary to reach all nodes in the interval that the index entries are placed in.

Since the keys are placed randomly, for a given number of requests all index entries will only be retrieved with a certain probability that depends on the distribution of nodes. It is possible to get around this problem by contacting the node in one end of the interval and send requests to each of the following nodes until a node outside the other end of the interval is reached. Since the contacted nodes are neighbours, this operation is not expensive to do. A project would, however, most likely retrieve the results periodically, and thus get to any remaining index entries eventually anyway, so an exhaustive search may not be necessary in practice.

A related problem is the load balancing of index entries means that a single node may no longer be able to answer a query involving multiple keywords with certainty. Instead multiple nodes may need to be contacted in order to find an index entry for a specific query.

## 3.5 Providing Incentives

The design should support assigning credit to provide users with an additional incentive besides goodwill for participating in the mass computing projects. It would also be beneficial if the system could assign credits to the nodes in the overlay network as well as assigning credits to clients that run jobs in order to convince users to contribute to the common infrastructure.

A prerequisite for assigning credits is a notion of identity for the parties that receive the credits. We will assume that both clients and nodes in the overlay network have an account to the system with a certificate and a corresponding private key that they can use to prove their identity. As a proof of authenticity the certificate should be signed by a certificate authority for the system.

### 3.5.1 Client Credits

As discussed in Section 3.2.1, we work with the trust model that job pool nodes cannot be trusted to handle job redundancy. When the overlay network cannot correlate results, it cannot determine whether a result is phony and should not be credited.

Thus the mass computing projects must themselves assign credits when processing results – they can then publish statistics about the credits on a web site. The credit calculation can be done in a manner similar to the current mass computing projects, e.g. based on the number of floating point operations spent.

The information that a particular client has run a job must, however, somehow get to the project. The client can append its identifier to the results it submits to the job nodes, but then a job node could replace the identifier with one of its own choice to claim the credit itself.

Hence, in addition to including its identifier in the job results, the client must also encrypt the resulting chunk of data in such a way that only the project can decrypt it. This ensures that the job pool nodes get an incomprehensible chunk of data that they cannot tamper with. The encryption can be done with a standard asymmetric cryptography technique using the public key of the project.

This approach to credit assignment assumes that the clients finish the jobs completely before they receive any credits. We can extend it to support trickling credits by having the clients report back to the job nodes from time to time. The job nodes would then insert themselves in an index so that the mass computing project can find them in the same manner as when the nodes have a result ready.

### 3.5.2 Node Credits

Being part of the overlay network involves three tasks: participating in the routing infrastructure, being responsible for jobs and serving index entries.

The participation in the routing infrastructure is a basic premise of our system and it appears to be difficult to assign credit to. For instance, simply assigning credit for each message routed allows two nodes that are collaborating to route messages back and forth between them to earn credit.

It is also difficult to assign credits for serving the light-weight index entries because it is a passive task controlled by the node which is closest to the job. Thus if we let this node assign the credits, it is easy for it to cheat. Similarly we cannot let the node with the index entry assign credits to itself without risking cheating.

Contrary to this, it is quite easy to assign credit for being responsible for a job since the set of responsible nodes are well-defined given the job specification and the structure of the network at a given point in time. Although the set of nodes may vary as nodes join and leave the network, but the mass computing project can simply assign credit to the nodes that are responsible for the job at the time the project retrieves the results.

## 3.6 Security Issues

A number of security issues arise from the design outlined in the previous sections. We first go through some possible attacks and discuss how they can be prevented, and then describe how to distribute the various certificates in use in the system.

### 3.6.1 Attacks

Since we have assumed that nodes in the overlay network are not malicious, although they may try to cheat, we will not consider attacks from inside the overlay network. Other attacks include:

**Smuggling alien jobs into the system** This is not possible without access to the private key of a project since both job nodes and clients can recognise unauthorised jobs by checking the signature on the job description.

**Forged job results submission** It should not be possible for a client to submit results as another client. A client $A$ could for instance try to submit incorrect results as another client $B$ to harm the project or raise suspicion that $B$ is malicious or cheating. To prevent this attack, clients should include a signature in the results.

**Excess job claiming** A client may attempt to retrieve as many jobs as possible, either to secure them for himself or as a form of denial-of-service attack. To prevent this attack, the nodes with index entries should clamp the rate with which a client can retrieve index entries from them. They can easily do this locally, but they need to coordinate in order to enforce a global limit.

They can do this by hashing the identity of the client and sending a timestamp to the location of the hash key in the network when an index entry has been sent to the client. Other nodes can then check the timestamps before letting the same client retrieve an index entry from them.

A malicious client cannot circumvent the rate limit by requesting jobs directly since the only way to get a proper job identifier is through an index entry, and a node with a job will only respond to a request for that job if the request includes the proper job identifier.

**Submitting false results** A client can try to return incorrect results, either to cheat, to gain extra credit or of maliciousness. The projects deal with this themselves by means of redundant computing as already mentioned.

**Thwarting the routing infrastructure** External hosts may try to enter the system by pretending they are valid nodes. This boils down to whether they can get themselves inserted in the routing tables of the nodes in the existing network since other nodes will only contact them if they are in the routing table of some node.

To prevent this a new node may only be inserted into routing tables once it has been verified to be an authorised participant of the overlay network. The verification procedure can be executed in the background and consists of asking for the public key of the host which must be signed

by a certificate authority, checking the authenticity and sending a cryptographic challenge that must solved by means of the private key. A successful response results in the new node being inserted into the routing table.

**Submitting phony index entries** A client could try to submit phony index entries to cause inconvenience for clients querying for jobs. This can be prevented by constructing the index entries for each job when it is created by the project and signing them with the project key so that they can be verified at the nodes storing them.

**Denial of service** External hosts can launch denial-of-service attacks on the overlay network by sending a large number of legitimate-looking requests, e.g. routing requests, to selected nodes. The robustness of the network should make it difficult to do more than disturb minor parts of it, but it may be possible to swamp some machines.

If an adversary that launches one or more of the above attacks is discovered, it would be desirable if it was possible to blacklist that client. As with the suggestion for hampering excess job claiming, it would be possible to put a note with the ban at the hashed value of the client identity in the network. Then the nodes could check whether a client is banned before proceeding with serving the request. Unfortunately, this means that even legitimate client requests must be preceded by a lookup operation, which may be too much of an overhead.

Instead we suggest disseminating the blacklist in the overlay network to allow nodes to check a request with the information available locally only. One possibility is to use an efficient broadcast algorithm designed for structured overlay networks, such as [21]. Although a broadcast involves all nodes in the network, we would expect that updates to the blacklist are rare. To avoid that the blacklist itself is used to launch a denial-of-service attack, the nodes in the network should limit the rate at which updates to it may be sent.

### 3.6.2 Distribution of Certificates

There are four types of certificates in the system:

1. The certificate of the certificate authority.
2. Project certificates, used to sign job descriptions.
3. Node certificates that are necessary to be part of the overlay network.
4. Client certificates issued when granting accounts.

The certificate of the certificate authority should be included in the distribution of the software to be available to everyone.

The project certificates must be available to the nodes that are responsible for jobs and the clients so that they can avoid accepting unauthorised jobs. The identity in a signature should include a reference to the certificate; the certificate can then be retrieved and cached for further use. We will not specify where to store the certificates since we do not address data management, but an external HTTP server or the overlay network itself could be used as for the data for jobs. The certificates are signed by the certificate authority so their authenticity can be easily verified.

Each node or client certificate is only needed by a limited number of entities at various times as part of the protocols we have designed. They can thus be sent directly by the owner when needed. For instance, a job node can request the certificate from a client in order to be able to authorise it when the client is attempting to deliver the results of a job.

If a certificate is compromised, a new one should be generated and the old certificate blacklisted as quickly as possible. Data written with the old certificate may have to be discarded since they could be compromised too.

## 3.7 Network Partitions

As described in Section 2.1.2, an overlay network may be partitioned because of physical network failures. This is likely to cause some intermittent problems as the partitioned network recovers, but the most serious problem is if it results in inconsistent data in the network because two independent overlay networks operate independently for some time.

Since index entries are considered to be soft state, they should adapt seamlessly and not cause any problems. For the jobs, there is a potential for inconsistency, however, since a job may be run by different clients in each independent network.

The job republishing operation described in 3.2.4 will ensure that when the two network merges the different replicas will end in the same state. It may, however, be the case that two different clients have claimed a job. In this case, the client to first return the results causes the nodes storing the job to change the job state which prevents the other client from submitting the results.

If two different clients both have submitted their results of a job, for instance while the network was still partitioned, the nodes that end up being closest to the job should agree on which results to keep. A solution that does not favour specific clients is to let the results on the node closest to the job be the master; this is also the results that the project will retrieve.

Hence, network partitions should only have a temporary impact on the function of our system.

# Chapter 4

# Evalution

We have evaluated the design described in the previous chapter empirically. This chapter describes the p2psim simulator which we have used to implement the design for testing purposes, a number of synthetic tests intended to explore the limits of the system and finally a more realistic simulation run based on data from LHC@home.

## 4.1 The p2psim Simulator

The p2psim [43] simulator is a discrete-event network simulator written in C++ that has been used to evaluate different distributed hash table designs. It is based on cooperating user-space threads and consists of a number of separate parts.

An event queue manages the simulated time and takes care of running scheduled events. The overlay network is modeled with a node class with support for remote-procedure calls, both synchronous and asynchronous, and some network-associated information such as IP address. A network topology is used to introduce latency in the remote-procedure calls – each call is encapsulated in a packet which first encounters simulated latency (implemented as an event) on the link between the two nodes, is then executed on the receiving node and lastly experiences latency on the way back to the sender with the reply. Several different network topologies are supported to model latencies on various kinds of networks.

The different protocols are modeled by deriving a class from the node class. Deriving a class makes it possible to store state for each node and provides support for making remote-procedure calls to the other nodes via their IP address – and these two things are all that is needed for implementing the various protocols. Version 0.3 of p2psim that we used has support for Chord, Accordion, Koorde, Kelips, Tapestry and Kademlia.

The protocols are written with the assumption that the remote-procedure

call mechanism is not fault-tolerant but only provides an UDP-like service. It is also possible to specify a failure model so that fraction of the remote calls fail.

There is no support for modeling packet queues or available bandwidth, though, so the failures, and the latencies for that matter, do not realistically approximate what would happen at a node that has saturated its link. Hence, the simulator does not appear to be appropriate for simulating networks where large amounts of data are transferred.

A simulation is started by an event generator that places predefined events, e.g. node joins, crashes, lookups, in the event queue. The node instances can then generate new events as a part of their implementation. Any required statistics and logs for analysis must be produced in the implementation of the simulation. A couple of standard event generators are included in the distribution of p2psim, e.g. one for generating node churn.

## 4.2 Implementation of Design

For our implementation, we derived an application class from a Chord protocol class (ChordFingerPNS) and added about 2700 lines of C++ code. We chose the Chord protocol because its implementation appeared to be the most thoroughly tested (there is considerable overlap between the developers behind p2psim and Chord).

### 4.2.1 Algorithm Implementation

Submitting jobs, finding jobs, the algorithm for claiming jobs as described in 3.2.2 and an algorithm for collecting finished jobs were implemented together with storing, retrieving and deleting index entries. Finally, the algorithms for republishing and load balancing were written.

The algorithm for collecting jobs works by finding a random node in the network and letting it collect at most $H$ jobs by looking up index entries, retrieving the results and informing the $r$ closest nodes that the job is collected. The latter is done asynchronously. The procedure is repeated periodically to simulate a project machine that harvests the jobs continually.

The republishing algorithm periodically compiles a list of jobs that a node is closest to and sends them to its $r - 1$ successors. These compare the jobs with the jobs they already store and resolve the inconsistencies. If they find inconsistent states where their state take precedence or find jobs that they believe the republisher should have sent to them, they add them to a list and send the list back to the closest node which then updates its jobs.

If a node is pushed away from being the $r$ closest to being the $r + 1$ closest to a job, it should delete its replica of the job after some time. We take care of this issue by associating a timeout with each job. Each time a job is received

through republishing, its timeout is reset to ensure that the $r$ closest nodes keep their copy of the replica.

To avoid risking that the periodic runs of the republish algorithm – and also of the load-balance algorithm – happen the same time on many nodes, which increases the peak load, the starting time for each new run is randomised somewhat before scheduling the run.

The load-balance algorithm begins by generating a random key and contacting the closest node to this key. If the loads of the two nodes differ too much (we used $\varphi = 0.4$, $w_{\text{index}} = 1$ and $w_{\text{job}} = 3$), the position of half load for the node with the most load is calculated and the other node moves to this position by first leaving the network and then joining at the new position.

The algorithm is aborted if one of the nodes has not experienced at least one run of the republishing algorithm or if the time for an index to time out plus 5 minutes for the Chord ring to stabilise has not passed. This is intended to prevent that a node participates in two load-balance operations in a row before it has had time to receive or turn over the data items that it is responsible for.

To avoid fluctuations when there is little load in the network we also abort the algorithm if the difference between the loads is less than $\Delta L_{\text{trivial}}$ ($\Delta L_{\text{trivial}} = 100$ was used in the tests). For instance, without this check if one node has load 1 and the other node has load 3, a load-balance operation would occur since $1 < 0.40 \times 3$ even though there is no reason to balance so small amounts of load.

Since a node finds another node to load balance with by choosing a random key, nodes that are responsible for larger parts of the hash space are more probable to be found. This is a benefit for balancing job load since a larger portion of the hash space is likely to contain more jobs. But it is a disadvantage for balancing index load since the wildcard technique tends to cluster nodes close to the hash of an index keyword.

However, since the nodes in such a cluster themselves contact other nodes and can make them move close to them, we did not consider it worthwhile to design and implement an algorithm to choose a random node with uniform probability instead of choosing a random key with uniform probability.

### 4.2.2 Simulator and Chord Issues

Since the default setup of p2psim is geared towards experimenting with the routing infrastructure, we had to modify the Chord class a little to make it return the results of a lookup to the application class. We also had to modify it to support arbitrary node placements as required by the load-balance algorithm, and finally we added a statistics generation function for monitoring the load of remote procedure calls.

The use of threads in the simulator is intended to make the algorithms easier to write, but it turned out to have a negative impact too. Whenever a remote

procedure call is initiated, the thread running the algorithm is suspended while other events happen. These events can change the data structures being used or even crash the node making the remote call while it continues to run.

This meant that we had to add a lot of failure handling and spend much time tracking down hard-to-find errors. The extra failure handling makes the algorithms harder to read and thus to some extent defeats the purpose of using threads. Consequently, an event-based system may have been better after all, even though the event-based paradigm chops the algorithmic flow up into small pieces.

We also discovered a concurrency problem with the Chord protocol as it is described in for instance [18]. The closest node to a data item is assumed to have the master version of the item and be responsible for republishing it to the other $r - 1$ closest nodes. This is necessary because the other nodes know only one predecessor and hence cannot contact all nodes (except the second closest node). The concurrency issue happens if a new node $B$ starts copying the item from the closest node $A$ to be able to join the network and $A$ at the same time receives updates for the item. $B$ will then have an old version, so if it ends up closer to the item than $A$ and never refreshes its version from $A$, it will later overwrite the new version on $A$ with its own old version.

To avoid this problem we do not assume that the closest node has the master version, but instead allow it to be overwritten if the node during republishing learns of a version with a higher precedence according to our inconsistency resolution protocol. This requires the republishing algorithm with two-way communication which is described in the previous section.

Finally, the implementation of the Chord protocol in p2psim does not cache lookup results to avoid looking up the same nodes multiple times in a row. We found the lack of a cache to have a major impact on the performance of our implementation, and hence implemented a cache on top of the Chord implementation which reuses the results returned from the Chord layer in up to 90 seconds. Ideally, the cache would be implemented in the Chord layer itself so that lookups can use it too, but the cache we implemented was enough to avoid some of the negative impact.

## 4.3 Test Setup

Before discussing the tests that we have conducted, we will just briefly describe the general test setup.

In order to simplify the analysis of the results, we use only one keyword ("LHC") for indexing the written jobs. The number of replicas, $r$, was set to 5. Furthermore, we set the short timeout for the temporary claim $T_{\text{claim}}$ to 15 seconds, the timeout before collected job can be removed $T_{\text{collect}}$ to 2 hours, the time index entries are inactive to 2 minutes, the expire time for index entries to 16 minutes, the index rewrite interval between 10 and 15 minutes, the expire

time of jobs to 1 hour and the replication interval between 20 and 30 minutes. $T_{\text{skew}}$ is not used since we do not simulate clock skews.

The network topology used in the simulations is based on the King data set used in [17] in which the latencies from more than 20000 Gnutella clients have been sampled using the King tool [29]. Hence, the latencies should be fairly realistic. The network sizes used are 128, 512 and 1024 nodes.

Most tests have been run a cluster of 2.8 GHz Pentium 4 with 2 GB memory on a Linux 2.6.10 kernel and took less than 3 hours to run. Since the simulator used simulates time internally, the particular hardware used should not have any impact on the test results.

## 4.4 Synthetic Tests

We have designed two types of tests, a static and a dynamic test, with artificially generated jobs to be able to push the system to its limits. The static test is intended to explore the capacity of the system with three phases of activity. In the first phase, jobs are submitted with a constant rate to fill the network with jobs. In the second phase, clients retrieve the jobs with the same rate and in the third phase the job results are submitted, and finally collected by the project. The system is given plenty of time to stabilise between the three phases.

The dynamic test is intended to evaluate how the system handles a continuous stream of incoming jobs. Jobs are submitted with a constant rate and retrieved by the clients with the same rate. The job submissions are given a lead to avoid running out of jobs, but after a few minutes the system should reach an equilibrium where the number of incoming jobs equals the number of finished jobs.

By varying the parameters of these two tests, we can evaluate the scalability of the system. We have also examined how the system copes with node churn and whether the load-balance algorithm manages to reduce the load.

In all simulation runs, the most important criterion is the intrusiveness of the system on the participating overlay network nodes. The simulator allows us to measure the bandwidth utilisation and the number of remote procedure call messages sent and received (what we call the RPC load). We have not attempted to measure other forms of resource consumption. The memory and disk space used should be negligible. CPU usage is more difficult to estimate without a real implementation, but the number of remote procedure calls per second at least gives a hint of the amount of work a node has to do.

The static tests should give an impression of how many resources are needed per node to maintain a persistent storage of a large number of jobs and their related index elements. In the dynamic tests, the intrusiveness with a high system throughput is evaluated.

### 4.4.1 Scalability

The first series of tests are intended to explore how the system behaves with different sizes of networks and different workloads. The static test was run with 50 000, 150 000, 250 000 jobs in total (15 jobs are submitted per second until the total is reached), and the dynamic test was run with a job submission rate of 10, 20, 30 jobs/s. All tests were run on networks of 128, 512 and 1024 nodes. To simplify the interpretation of the results, node churn and load balancing were disabled.

The architecture of p2psim prevented us from running tests with more than 250 000 jobs and and submission rates larger then 30 jobs/s, since the 2048 megabytes of memory available on the simulation machines were not sufficient to store the required events.

#### Results

The results are shown in Figure 4.1 for the static tests and Figure 4.2 for the dynamic tests, both with 1024 nodes.

Each plot in the leftmost column displays the number of active jobs (i.e. jobs that are not in the *finished* or *collected* state) on each node as the green lines, the number of index entries on each node as the blue lines and the maximum number of RPC/s any node has encountered as the red bars. For instance, if there is a red bar at second 130 with height 82 it means that in the 130th second the maximum number of remote procedure calls that any node received is 82. Statistics about RPC loads less than 10 RPC/s were not collected during the test runs (to reduce the load on the test machines), hence if no bar is plotted the maximum RPC load is less than 10 RPC/s.

Consider the job load in the static tests in Figure 4.1. The number of jobs on each node rises until all jobs have been submitted. Since there is no load balancing, the number of jobs on each node varies considerably, from about 50 jobs to about 800 jobs in the test with 50 000. The job load then falls again when the jobs start being claimed by the clients.

One of the two index curves disappears out of the top of the plots during the first few minutes. This is index entries for the "LHC" keyword which are written as the jobs are submitted to the system. Since there is no load balancing of the index entries in these tests, all the "LHC" keywords will reside on one node and that node will then store the same number of index entries as there are active jobs, e.g. 50 000 entries in the test with 50 000 jobs.

The "LHC" index load falls to zero once the jobs have been claimed and are running. This happens after about 15 hours. After the idle period of 12 hours, there is again some index activity when the jobs start having their results submitted. The activity comes from index entries written with the "LHC:finished" keyword. The line fluctuates as the job results are collected. The reason that the fluctuations are above zero is that quarantined index entries for jobs that are

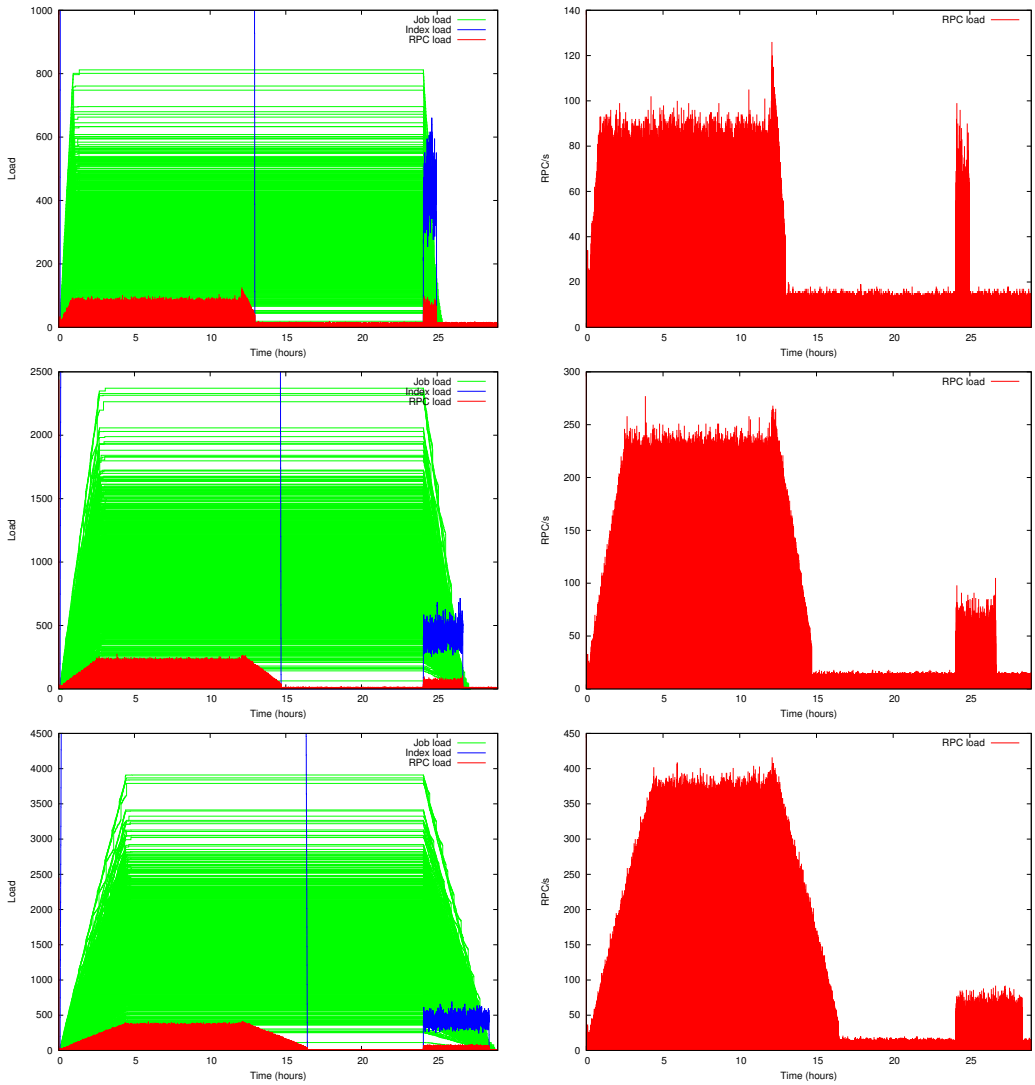


*Figure 4.1: Results of the static scalability test with 1024 nodes. From the top* 50 000, 150 000 *and* 250 000 *jobs. To the left, an overview of the load and to the right a close-up of the RPC load.*

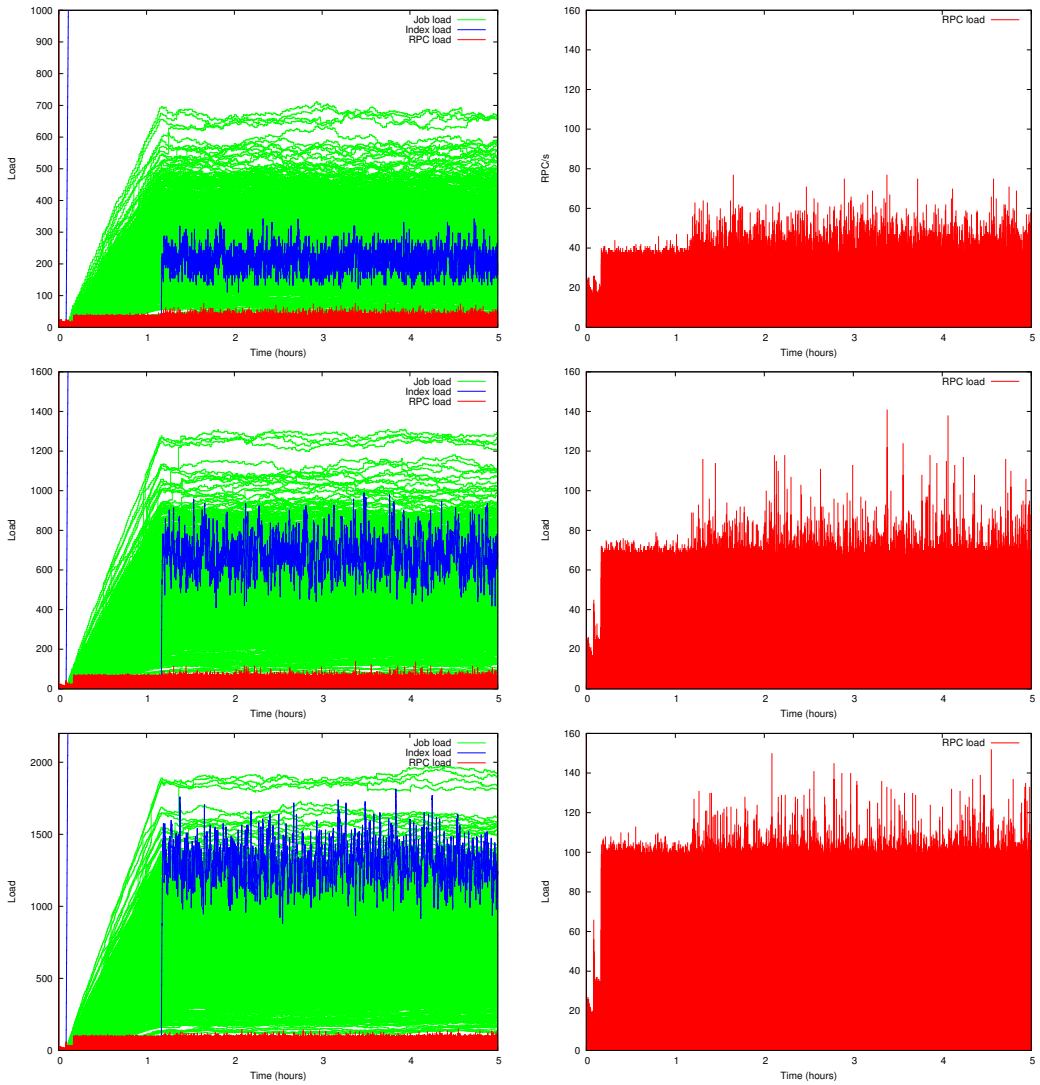


*Figure 4.2: Results of the dynamic scalability test with 1024 nodes. From the top 10 job/s, 20 job/s and 30 job/s submission rate. To the left, an overview of the load and to the right a close-up of the RPC load.*

being collected are counted in – there is a short delay before the index entries are actually deleted by the nodes storing the jobs as the result collector is busy processing the entries it has acquired.

Now consider the maximum RPC load displayed in the rightmost column. Since the jobs and index entries require only little storage space, the RPC load provides a better measure of the actual intrusiveness than the job and index load. The maximum RPC load closely follows the index activity. In the quiet period when the clients are computing the results, the maximum load is only about 20 RPC/s both with 50 000, 150 000 and 250 000 jobs. This suggest that the system scales well with the number of jobs as long as there is no activity. In the period with most index activity, the RPC load is, however, about 100 RPC/s with 50 000 jobs, 250 RPC/s with 150 000 and 400 RPC/s with 250 000 jobs.

Since the plots of the RPC load only display the maximum load, we also computed the average RPC/s for each test. As we do not have the statistics for less than 10 RPC/s, we conservatively approximated the missing loads to be 9 RPC/s. Surprisingly, the estimate of the average RPC/s for all seconds then turns out to be between 9-10 RPC/s.

To investigate this further, a bar diagram of the estimated average RPC/s for each node over its whole lifetime is shown in Figure 4.3 for the test with 1024 nodes and 50 000 jobs (the other tests exhibit the same characteristics). Each bar is calculated as the estimated total number of remote procedure calls for a node divided by the lifetime of the node, and the bars have then been sorted with the most loaded nodes first (the diagram only shows the average RPC/s for first eight most loaded nodes since the rest are not visible anyway). Only two nodes have an estimated average load higher than 9 RPC/s – the two nodes that store index entries. Hence, it would seem that the high RPC loads are caused by the nodes with index entries.

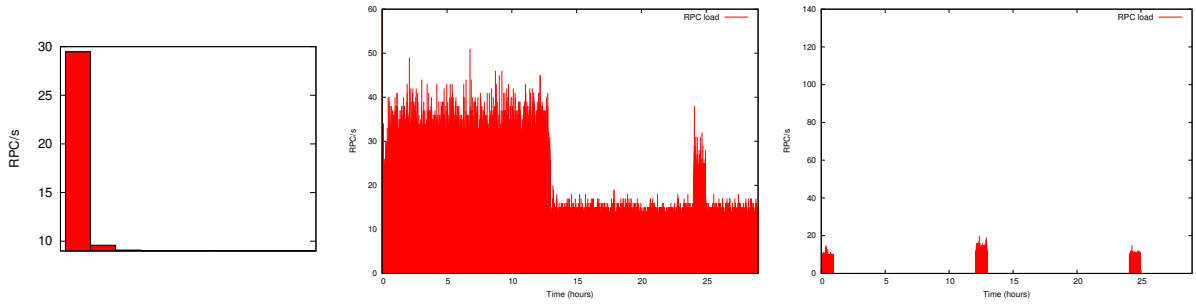


*Figure 4.3: To the left, a bar diagram of estimated average RPC/s for each node over its lifetime from the static test with 1024 nodes and 50 000 jobs. In the middle, the RPC load when the nodes with index entries are excluded. To the right, the load from Chord lookups are also excluded.*

The middle plot in Figure 4.3 which shows the maximum RPC loads when the nodes with index entries are excluded, confirms this. The remaining load is

still about 30-40 RPC/s in the period with high load. Upon investigation almost all of the load turns out to be from Chord lookups; if they are not counted in, the remaining RPC load is less than 9 RPC/s except in the three active phases where it still below 20 RPC/s as shown to the right in Figure 4.3. With a better cache most of the lookups could probably be avoided.

We also measured the maximum bandwidth used. For the test with 50 000, it is below 5 kB/s. If load from index nodes and Chord lookups is excluded, it is below 1 kB/s.

In the dynamic tests in Figure 4.2, the job load ramps up and then flattens out after an hour when the job results start being submitted by the clients (each job takes an hour to run). The index load from the "LHC" keyword again disappears in the top of graph, whereas the load from the "LHC:finished" index entries fluctuates. After the initial minutes, the maximum RPC load is almost constant for an hour until the job results start being submitted and collected.

The three tests with 10, 20 and 30 jobs/s are very similar and differ mostly in the magnitudes of the lines. Note that while the job submission rate is increased from 10 to 30 job/s, the maximum RPC load increases only about 2.5 times (from 40 RPC/s to 100 RPC/s). This suggests that the system scales well in the number of jobs.

We have investigated the RPC load as with the static tests, and again the load is dominated by the index nodes, see Figure 4.4. Without the load from the nodes with index entries, the maximum RPC load is about 30 RPC/s. If the Chord lookups are also excluded, the load drops to about 10 RPC/s. We found the bandwidth usage to be below 6 kB/s. Without the load from index nodes and Chord lookups, it is below 1 kB/s.

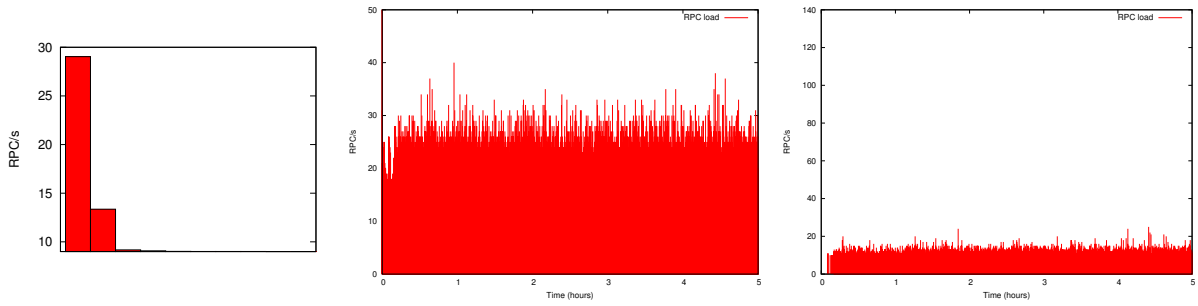


*Figure 4.4: To the left, a bar diagram of estimated average RPC/s for each node over its lifetime from the static test with 1024 nodes and 50 000 jobs. In the middle, the RPC load when the nodes with index entries are excluded. To the right, the load from Chord lookups are also excluded.*

The tests with 128 and 512 nodes had similar results as those shown here. Since there are fewer nodes to store the jobs, each node will have comparatively more jobs, but the maximum RPC load remained the same. We attribute this to the domination of the measurements by the index activity, which suggests that

the tests with load balancing enabled (in Section 4.4.3) should give a better impression of the scalability.

#### Conclusions

Leaving aside for the moment the load from index entries, it appears that the system scales well in the number of jobs. With about 10% of the nodes in the network as LHC@home has clients and with ten times the number of jobs processed in the dynamic test with 10 jobs/s, the system imposes a low average load on the nodes. The maximum load is somewhat higher, but still within the limits of what a node should be capable of, considering that replying to most remote procedure calls does not involve querying a huge database.

The tests highlight two problems, however:

- The index activity overshadows the load from the rest of the system so load balancing is required in order to scale.
- The routing layer should be optimised to reduce the number of lookup messages.

The high index activity when the jobs reside in the system for a relatively long period of time waiting to be processed also suggests that it might be beneficial to use a dynamic scheme for the index rewrite interval. The situation that an index entry has not been retrieved for long can be used as an indication that it will take long before it is going to be taken and that it is thus not as important if it is not maintained as thoroughly in the mean time. Of course, such a scheme must be designed carefully to avoid that some index entries end being difficult to get hold of again.

### 4.4.2 Churn

In a real setting nodes will crash, leave and join frequently so the system must be able to handle it transparently. In order to test whether it is capable of that, we have run the static and the dynamic test with churn.

Each test has a specified degree of churn which is generated by an exponential distribution supplied with the simulator. When a node joins, it calculates its lifetime and crashes when it has reached the end of it. It then calculates its deathtime and remains down until reaching the end the deathtime, at which point it joins as a new node. The lifetime of a node is calculated as

$$T_{\text{life}} = \mu_{\text{life}} \cdot \log(1 - \gamma)$$

where $\gamma$ is a random number, $\gamma \in [0; 1)$, and $\mu_{\text{life}}$ is the mean lifetime. The deathtime is calculated in a similar manner from the mean deathtime $\mu_{\text{death}}$.

We ran tests with $\mu_{\text{life}} = \mu_{\text{death}} = 24$ hours and tests with $\mu_{\text{life}} = \mu_{\text{death}} = 12$ hours. Note that when the mean lifetime is equal to the mean deathtime, the

nodes will be active only about half of the time on average. The static tests were run with 50 000 jobs and the dynamic tests were run with a job submission rate of 10 jobs/s, both with initially 1024 nodes and with load balancing disabled.

### Results

The results of the churn tests are shown in Figure 4.5 for the static test and Figure 4.6 for the dynamic test. These graphs should be compared to their counterparts without churn, Figure 4.1 for the static tests and Figure 4.2 for the dynamic tests. Only one test run is shown for each test; other runs had similar results.

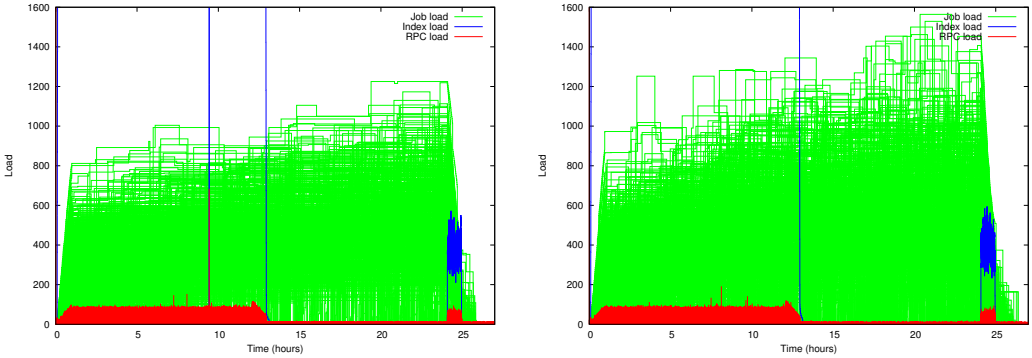


*Figure 4.5: Results of the static tests of churn with 1024 nodes and 50 000 jobs. To the left, the run with a 24 hour lifetime mean and to the right a 12 hour mean.*

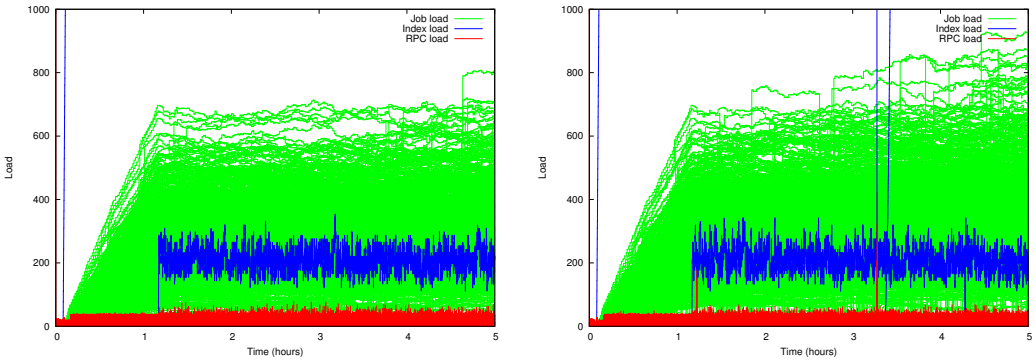


*Figure 4.6: Results of the dynamic tests of churn with 1024 nodes and a 10 jobs/s submission rate. To the left, the run with a 24 hour lifetime mean and to the right a 12 hour mean.*

The results of the two static tests in Figure 4.5 are close to the results of Figure 4.1 with two exceptions. There are a couple of spikes in the tests with churn which are not present in the test without churn – the most noticeable one is when the node with the index entries crashes in the test with $\mu_{\text{life}} = 24$ hours.

Also there are lots of vertical lines in the job load; each time a node crashes, its jobs are lost and later replicated on another node.

The results of the dynamic tests in Figure 4.6 are also close to the results of the dynamic test without churn in Figure 4.2. The index load fluctuations from the "LHC finished" index entries increase a bit, and one of the nodes with index entries crash in the test with $\mu_{\text{life}} = 12$ hours which cause the index curve to disappear shortly before the index is recreated.

#### Conclusions

For both the static and the dynamic test, the nodes generally store a larger number of jobs when churn is enabled because fewer nodes are active. But the RPC load is not increased considerably, with the only exception being when a node with index entries crashes. This should not be so much of a problem if the index entries are divided among several nodes as is supposed to happen when load-balance algorithm is enabled, as it will be in the tests in Section 4.4.3. Hence, we can conclude that the system appears to be good at handling churn.

### 4.4.3 Load Balancing

When evaluating the effectiveness of the load-balance algorithm, we are interested in how well it balances the index entries and the jobs, and how it affects the load from remote procedure calls.

Since we found in Section 4.4.1 that the RPC load from the nodes with the index entries overshadowed any benefit the network may have gotten from adding more nodes, we also examined how the RPC load scales as more nodes are added. Hence, we ran the static test with 50 000 jobs and the dynamic test with 10 jobs/s, both with 128, 512 and 1024 nodes and the load-balancing algorithm enabled.

#### Results

The results are shown in Figures 4.7 and 4.8. The corresponding results from Section 4.4.1 are also shown. Note that only one test run is shown for each test; other runs had similar but not identical results. Especially the heights of index load plots vary much in the beginning because the load-balance algorithm is randomised on each node.

In Figure 4.7, the load-balancing mechanism is most visible on the leftmost column with the plots with index entries. After some time (the algorithm runs in 20-60 minutes intervals) the algorithm halves the maximum index load in a series of steps.

In the middle column, the algorithm also manages to distribute the jobs more evenly in all three tests compared to the bottom reference plot from Figure 4.1. The vertical lines in the middle column come from the node leaves and

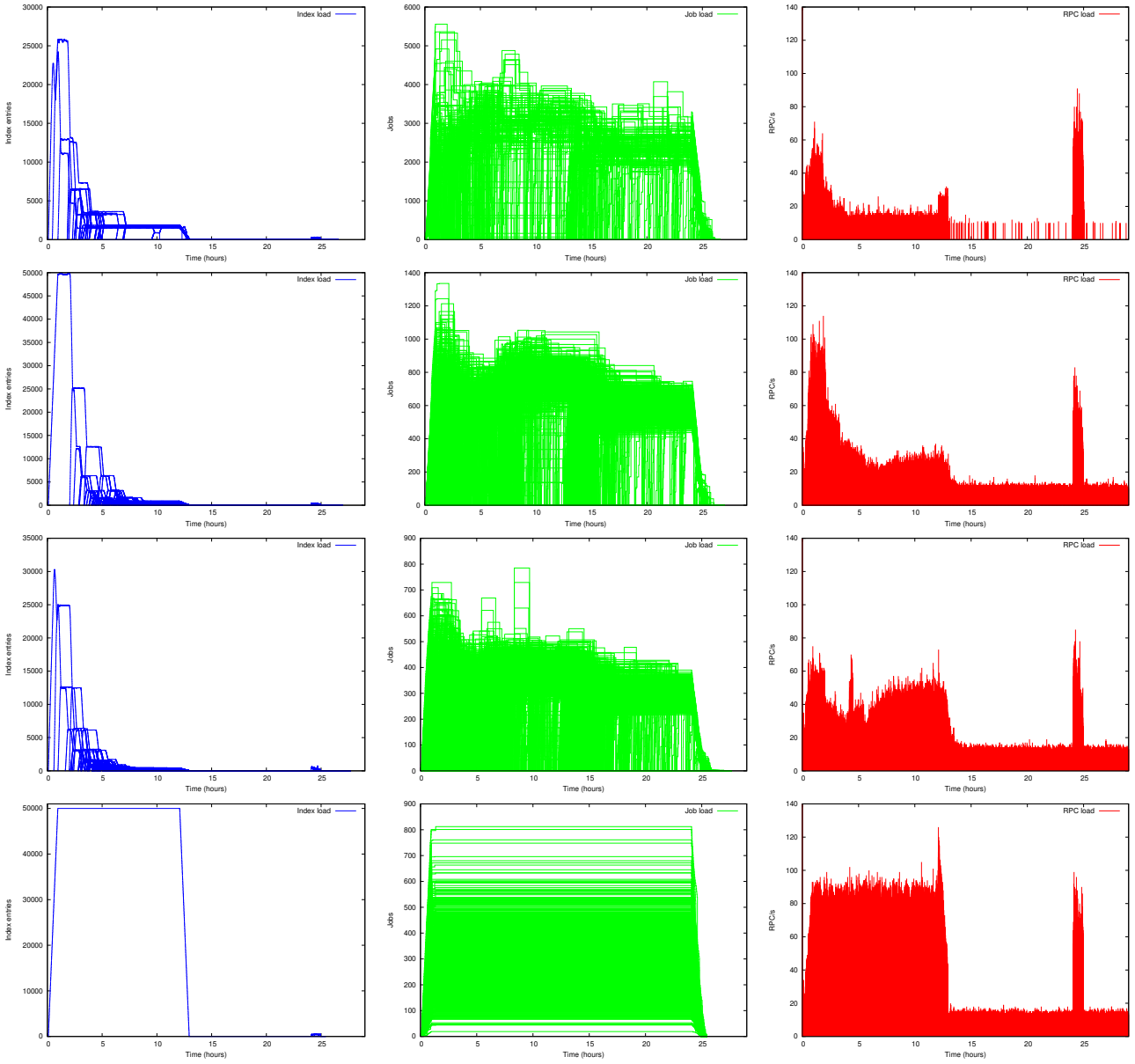


*Figure 4.7: Results of the static test with load balancing enabled and* 50 000 *jobs. The first row is with 128 nodes, the second with 512 nodes and the third with 1024 nodes. The last row is the results from Figure 4.1 (with 1024 nodes) for comparison.*

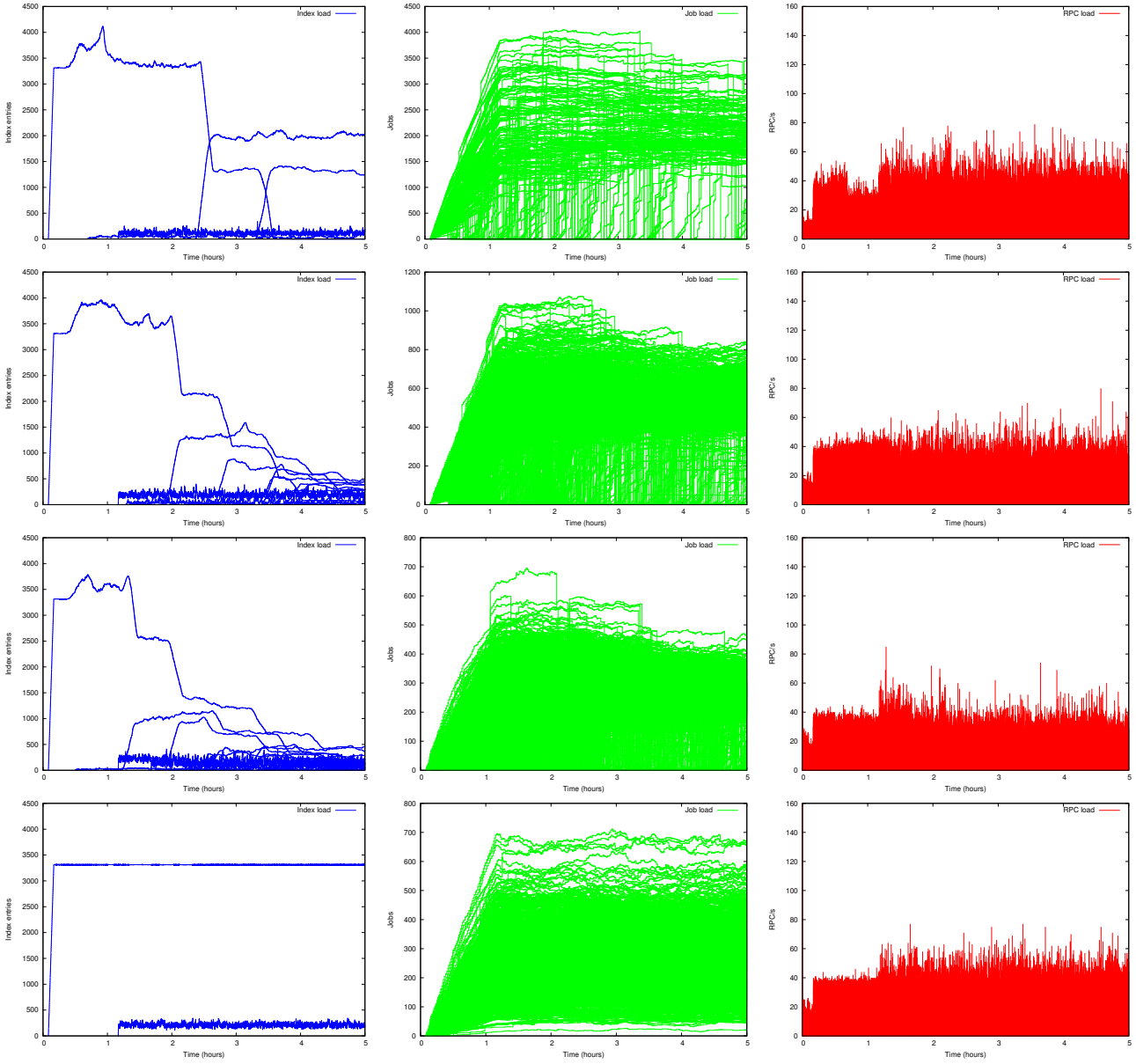


*Figure 4.8: Results of the dynamic test with load balancing enabled and a job submission rate of 10 jobs/s. The first row is with 128 nodes, the second with 512 nodes and the third with 1024 nodes. The last row is the results from Figure 4.2 (with 1024 nodes) for comparison.*

joins that the algorithm uses; as shown in the figure, there is much more load-balancing activity in the test with 1024 nodes (1820 node moves) than in the test with 512 nodes (938 node moves), and correspondingly more activity in the test with 512 nodes than the test with 128 nodes (338 node moves). On average, each node makes about two node moves, so when there are more nodes, there is also more activity.

As the rightmost column shows, this affects the RPC load. Contrary to what we expected before running the tests, the maximum RPC loads are higher in the tests with more nodes even though there are more nodes to share the load. It was found in Section 4.4.2 that the node crashes can cause RPC load spikes, and since node leaves are equivalent to node crashes in the simulation of Chord, we attribute the extra load in tests with larger networks to the node leaves from the load-balancing activity.

Interestingly, the extra load is from Chord lookups. If these are removed, the RPC load for the test with 1024 nodes is as in Figure 4.9 which also contains the corresponding RPC load from Figure 4.3 without load from nodes with index entries and Chord lookups. When Chord lookups are not counted, the load balance algorithm manages to divide the load from the index entries well enough to get close to the ideal situation after some hours.

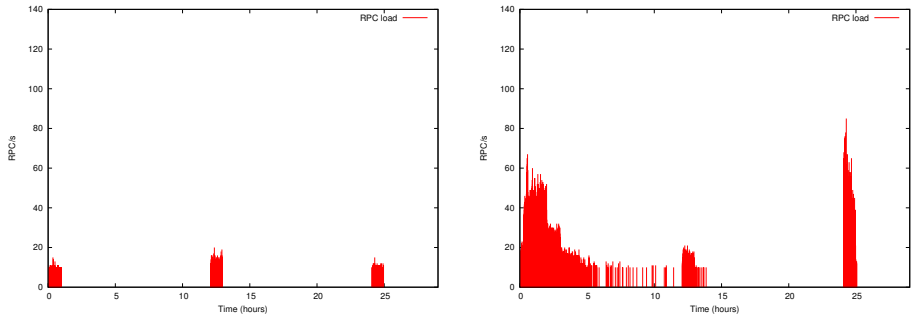


*Figure 4.9: To the left, the RPC load from Figure 4.3 without Chord lookups and load from the nodes with index entries. To the right, the load without Chord lookups from the corresponding test with 1024 nodes and load balancing enabled.*

The rightmost column of Figure 4.7 also shows, however, that the RPC load is considerably lower in all tests than in the bottom reference plot. Hence the load-balance algorithm does manage to balance the RPC load too. Another sign of this is the bar diagrams with estimated average RPC load shown in Figure 4.10. The large column from the node with the most index entries has been evened out.

Figure 4.8 shows the results for the dynamic tests with the results from Figure 4.2 plotted at the bottom for comparison. The same remarks hold here for the first two columns as for the static tests.

The rightmost column with the RPC loads is different. The load-balancing algorithm has not helped as much as for the static tests. However, in the dy-

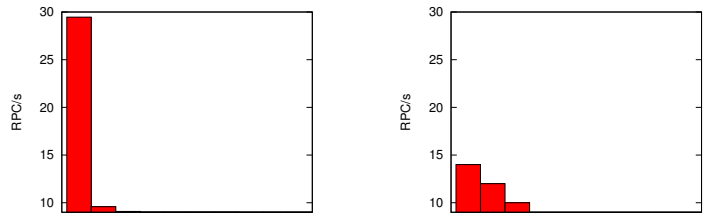


*Figure 4.10: To the left, a bar diagram of estimated average RPC/s for each node over its lifetime from Figure 4.3. To the right, the corresponding diagram from the static test with 1024 nodes and load balancing enabled.*

namic tests there is not a problem with increasing RPC load in tests with more nodes in the network – on the contrary, the load is a bit lower as we expected. One reason for this could be that the number of index entries is much lower than in the static tests and hence results in less activity. Also, the tests only run for five hours so there is not as much time to balance the load. The test with 128 nodes has 101 node moves, the test 512 nodes has 420 node modes and the test with 1024 nodes has 812 node moves. The load without Chord lookups is plotted in Figure 4.11. Apparently, the load-balance algorithm has not managed to balance the index load enough to make it disappear. The bar diagram with estimated RPC load illustrated in Figure 4.12 shows that the algorithm has managed to half the average load of the most exposed node.

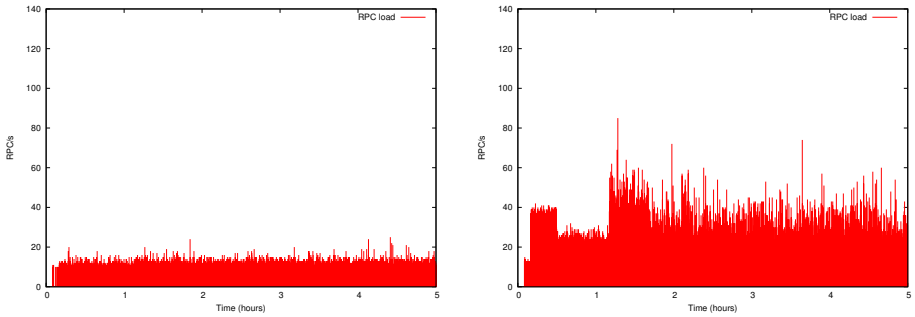


*Figure 4.11: To the left, the RPC load from Figure 4.4 without Chord lookups and load from the index nodes. To the right, the load without Chord lookups from the corresponding test with 1024 nodes and load balancing enabled.*

### Conclusions

Although the load-balance algorithm manages to balance the index entries and the jobs, some problems remain with the RPC load. The algorithm appears to cause too much activity – a better tuned $\Delta L_{\text{trivial}}$ could probably amend part of this problem. Another problem which is visible in these synthetic tests is that the algorithm is slow at adapting to changes, for instance when a large number of index entries with the same keyword are written in small period of time.

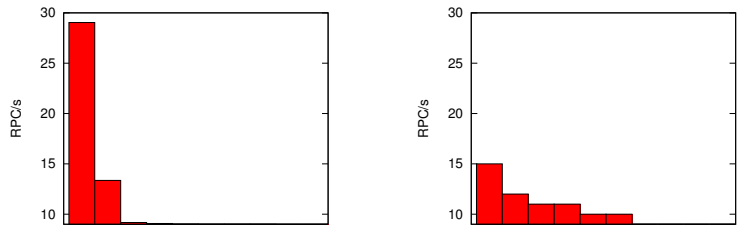


*Figure 4.12: To the left, a bar diagram of estimated average RPC/s for each node over its lifetime from Figure 4.4. To the right, the corresponding diagram from the dynamic test with 1024 nodes and load balancing enabled.*

## 4.5 Evaluation with Realistic Data

This section presents a complete simulation of the first 37 days of activity extracted from the LHC@home project – the test machine did not have memory enough to complete a full simulation of the entire three months. 700 887 jobs are submitted and 612 759 processed during the 37 days. The test was run with 1024 nodes in the overlay network which is about ten percent of the number of LHC@home clients. Load balancing is enabled together with node churn with a lifetime and deathtime mean of 24 hours.

Figure 4.13 shows how the job load varies as batches of jobs are submitted and processed. The submission of each batch of jobs is evened out a bit so that at most 10 jobs/s are submitted. The index load generally follows the job load with spikes when a new batch of jobs is submitted or when a node with index entries leaves the system. Both job and index loads are quite well balanced, most noticeable in the period from day 25 to day 30 when the load-balance algorithm has had plenty of time to adapt to the conditions.

The RPC load is generally quite low, around 40 RPC/s most of the time, but also has some spikes, especially in the beginning where it reaches 200 RPC/s. We attribute these to the load balancing of the index. After the algorithm has stabilised, further load balancing only causes minor spikes. If the Chord lookups are removed, as shown in Figure 4.14, the load spike at the beginning is reduced to 10-20 RPC/s and the RPC load is about halved in the remaining period.

Comparing Figure 4.14 to the first 37 days of the server load found in Figure 1.5, the intention of absorbing the peak load in the overlay network appears to has worked if the lookups are handled better. We attribute the remaining spikes to the slow adaptation of the load-balance algorithm.

Overall the system handles the test well. After the initial period, it adapts itself to the changing load and maintains a low level of RPC load throughout most of the test.

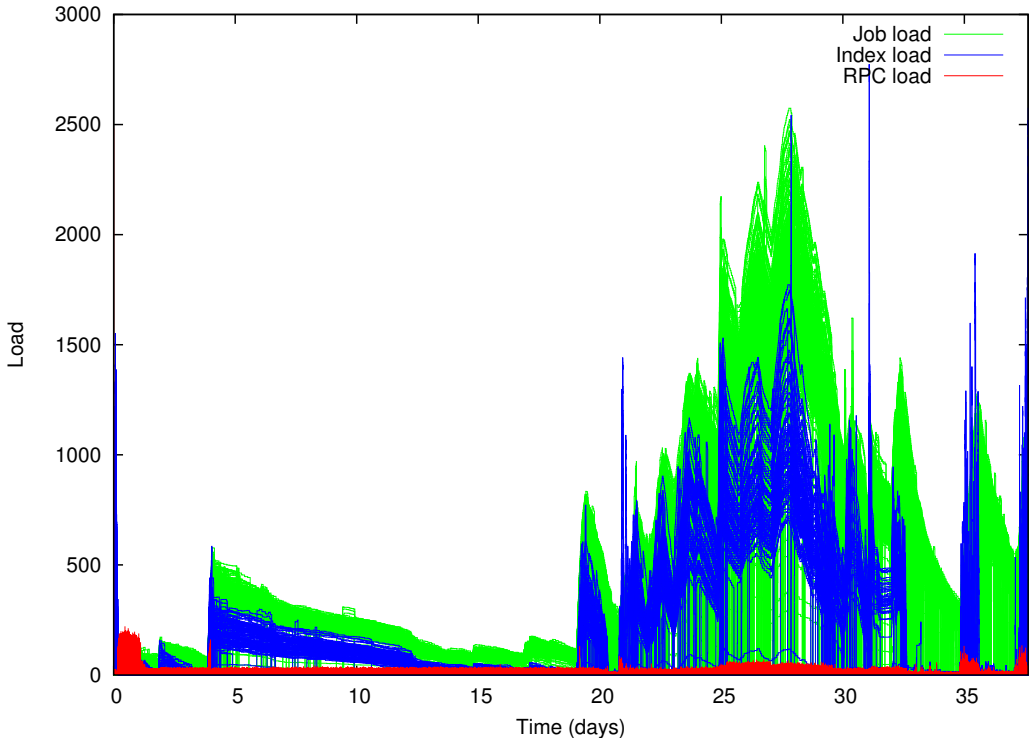


*Figure 4.13: Results of the test with realistic data.*

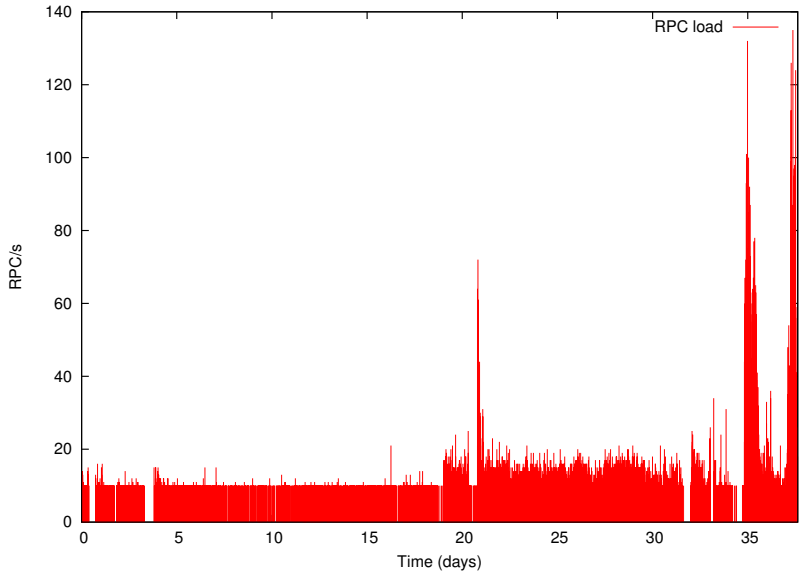


*Figure 4.14: RPC load of the test with realistic data without Chord lookups.*

# Chapter 5

# Conclusion

This chapter summarises the achievements of this project, states the conclusions we have drawn and finally presents some areas of future work.

## 5.1 Summary

In recent years utilising the idle CPU time of desktop machines connected to the Internet has become increasingly popular. In Chapter 1 we presented an analysis of the mass computing project LHC@home. The analysis revealed a heterogeneity in the clients that may be beneficial to take into account by differentiating between them to better make use of their resources. The jobs were found to be mostly homogeneous, but varying significantly in processing time from project to project.

Furthermore the analysis uncovered that the central server in the current mass computing architectures is a bottleneck because of high peak loads. This motivation led us to suggest a decentralised design where some of the clients are involved in the job distribution to relieve the projects from having to maintain clusters of servers.

In order to provide a solid foundation for building such as decentralised system, Chapter 2 reviewed state of the art peer-to-peer techniques. Unstructured as well as structured overlay techniques, in the form of distributed hash tables, and practical issues such as data loss, latency, node churn, network partitions and security were examined. We then reviewed several indexing techniques intended to enable the decentralised system to locate jobs efficiently. In order for an indexing technique to scale to a large amount of clients, the index must be divided among several nodes so we concluded the chapter with a discussion of several techniques for load balancing.

Next we presented, in Chapter 3, the proposed design of our system. The design uses a distributed hash table that it uses to store jobs and index entries. Jobs are submitted to the network by a project and replicated to several nodes

to prevent data loss when nodes fail. Index entries are created from keywords in the job descriptions and maintained by the nodes that store jobs. Clients go through the index entries to find suitable jobs. They then claim the jobs from the nodes storing them and eventually submit the computed results.

The index keywords allow several projects to coexist in the same network without interfering with each other, and also provide a way of differentiating between the heterogeneous clients. The index entries are processed in a roughly first-in, first-out manner. Both jobs and index entries are load balanced, and the system can automatically recover from inconsistencies and thus also cope with network partitions.

In order to evaluate the proposed design, we implemented it in the p2psim simulator framework as described in Chapter 4 and then conducted a series of synthetic tests with the aim of gaining knowledge about the limits of the system.

The tests considered three different aspects of the system. How the system scales in the number of jobs, the impact of load balancing and the ability to handle churn. The scalability tests demonstrated that the system has low average load, but that the load of the nodes storing the index keywords is several times higher than the load of the other nodes. The load-balance algorithm was found to even the number of index entries and jobs but to cause moderate load from extra messages when nodes moves from one position to another, and also to be slow at adapting to quick changes. Finally the churn tests revealed that the system quickly adapts to nodes leaving and the load from remote procedure calls is similar to the tests without churn.

A real-word test was also conducted on data from the LHC@home project and demonstrated how the system handled a real-world scenario. The test showed that the system is capable of handling the changing load and overall has a low degree of intrusiveness.

## 5.2 Conclusions

We have designed and tested a decentralised system for distributing jobs and collecting results for mass computing projects. In our design, we have stressed scalability and robustness and tried to address intrusiveness, flexibility and security. We believe that the tests have shown that the system is scalable and not intrusive if some remaining issues are taken care of, that the system is fault-tolerant and adept at repairing itself, that the index keywords provide a reasonable level of flexibility, and finally that it does not compromise the security if a few precautions are taken as described in Section 3.6.

Hence, we believe that we can answer the question of whether a decentralised architecture can be competitive with a centralised architecture affirmatively. The architecture has the benefits of built-in scalability, high availability, easier and cheaper deployment for new projects and an abundance of

resources.

To sum up, our contributions are:

- A novel approach to job distribution for mass computing projects using recent advances in peer-to-peer technology.
- Experimental evaluation of this approach.
- An analysis of the mass computing project LHC@home.
- The design and test of a new way of balancing load for data items that have the same key and cannot be cached.

Furthermore, we believe that our architecture with its support for distributed job distribution can provide the basis for building a design for very scalable computational grids.

What is foremost missing is support for data management and more certificate infrastructure to allow more than just a few selected projects to submit jobs. We have argued that the keyword index can support numeric ranges too by crude quantification, but a more fine-grained approach may be needed after all in a grid setting – if so it can be built into the overlay network. There is also a lot of talk in the grid community about scheduling, but our system explicitly builds upon a pull model that precludes some forms of scheduling. The index may be able to help, though. For instance, one could index jobs based on priority or target location.

People build grids with different intentions, however, and our use of peer-to-peer technology does have it drawbacks, most importantly a greater dependency on otherwise unrelated parties, potentially all over the world, with the associated loss of control. Our system also follows a best-effort paradigm and does not provide any guarantees for when and where jobs are run. Hence, our architecture may not be appropriate for all uses of computational grids; at least not without further research.

## 5.3 Future Work

While the tests in Chapter 4 have demonstrated the feasibility of the design there is still room for enhancements:

- Currently the design assumes that nodes follow a fail-stop model. This may not be appropriate in practice because it requires that the nodes are not malicious. It would be safer to build the system around a Byzantine failure model where no assumptions on faulty nodes are made. There has been done some work in [51] on handling Byzantine failures in a distributed hash table, although with the assumption that faulty machines can be repaired.

In general it should be possible to adapt well-known distributed algorithms for the various protocols since the distributed hash table abstraction can map each task to a small set of $r$ nodes so that even algorithms with quadratic message complexity can be used as long as they only involve small subsets of the entire system.

- The overhead and intrusiveness of certificate checks, encryption of results and the other means of maintaining the security mentioned in Section 3.6 should be tested.

- We restrained the design to only cover job distribution and selection. Data management, community web site management and a framework for application building are issues that need to be resolved before a real-world application can be build. For the former issue it is worth investigating embedding the data in the overlay network, whereas the two other issues can probably be solved by reusing parts of BOINC.

- Another real world aspect is how machines with restricted Internet connections, e.g. going through NAT or a firewall, can participate in the overlay network. Skype [56], a peer-to-peer Internet telephony program, handles the problem by using supernodes, which are not restricted, as middle-men together with other techniques when two restricted clients communicates. The major drawback with this approach is the added bandwidth usage and latency, although latency is not a problem for our application area.

- The problems uncovered through the experiments with the load-balance algorithm must be addressed. The problem of hotspots when caching is unsuitable is not thoroughly studied yet in the literature and requires further research. One of the other algorithms presented in Chapter 2 could be used, or the currently used algorithm could be extended.

  There appears to be two problems that together cause the currently used algorithm to adapt only slowly to sudden changes:

  1. Index entries are never transferred from one node to another – instead the algorithm relies on the periodic index maintainenance to rewrite the index entries which takes some time. Since this indirect transfer must be completed before another load-balance operation is started to avoid instability, this limits the rate at which load balancing can occur.
  2. Load balancing is initiated periodically which means that it may take some time before a node discovers that it is becoming heavily loaded.

  The first problem could be solved by introducing a protocol for transferring index entries directly from one node to another after a node has

moved to another position. The second problem could be amended by monitoring the rate of incoming requests; it is difficult for a node to determine whether its load is high, though, without comparing it to the load of other nodes, but it would be possible to exchange load information as a side-effect of other operations, e.g. lookups.

- There is also a problem with the calculation of load for the load-balance algorithm. The load is calculated the same way on each node, which means that the nodes are treated as if they were homogeneous. This assumption is hard to justify for nodes on the Internet. Instead the load should be weighted according to the capabilities of the node that calculates it.

- Also the load calculation does not take the request rate into account. If, for instance, the system runs out of jobs for a particular keyword, the clients may still periodically contact the nodes storing the index entries for this keyword – and after some time, the load balance algorithm will move all of these nodes except one away since the load from the index entries will be reduced to nothing as they have all been taken. It is likely that more than one node would be needed to send negative replies to the clients in a very large system. The request rate can be included in the calculation if each node remembers the last few minutes of requests and weight them in.

- With the additional resources available in the overlay network compared to a central server, it would perhaps be possible to extend the system to allow several clients to coordinate to run a large job that must run on several machines simultaneously. Such a feature could potentially open up for more applications of mass computing, although the limited bandwidth available on most clients may restrict the scope; if client heterogeneity is taken into account, it may still be feasible, however.

- Finally a computational market, where clients receive money instead of credit, could be envisioned. This means that the clients have to trust the projects more since they get an incentive to cheat to save money. Some parts of the design may have to be reworked to help prevent projects from cheating and facilitate this trust.

# Bibliography


[1] David P. Anderson. BOINC: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.

[2] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@Home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[3] Artur Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. Technical Report HPL-2002-209, HP Labs, 2002.

[4] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[5] Ranjita Bhagwan, Kiran Tati, Yuchung Cheng, Stefan Savage, and Geoffrey M. Voelker. TotalRecall: System support for automated availability management. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation*, March 2004.

[6] Ranjita Bhagwan, George Varghese, and Geoffrey M. Voelker. Cone: Augmenting dhts to support distributed resource discovery. Technical Report CS2003-0755, University of California, San Diego, July 2003.

[7] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. Technical report, BU Computer Science, 2002.

[8] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, 2002.

[9] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314, 2002.

[10] Charlie Catlett. The TeraGrid: A primer, September 2002.

[11] A.J. Chakravarti, G. Baumgartner, and M. Lauria. The Organic Grid: Self-organizing computation on a peer-to-peer network. *IEEE Transactions on Systems, Man, and Cybernetics*, 35(2), March 2005.

[12] Thomas Christensen, Anders Rune Jensen, Rasmus Aslak Kjær, Lau Bech Lauritzen, and Ole Laursen. Nubis: A decentralised, flexible, fault-tolerant and scalable foundation for computational grids. Student project report (seventh semester), Aalborg University, December 2004. `http://www.cs.aau.dk/library/cgi-bin/detail.cgi?id=1103661917`.

[13] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, January 2002.

[14] Austin Clements, Dan Ports, and David Karger. Arpeggio: Metadata searching and content sharing with chord. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems*, February 2005.

[15] ClimatePrediction.net gateway. Internet home page. `http://www.climateprediction.net/`.

[16] Francisco Matias Cuenca-Acuna, Christopher Peery, Richard P. Martin, and Thu D. Nguyen. PlanetP: Using gossiping to build content addressable peer-to-peer information sharing communities. In *Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*. IEEE Press, June 2003.

[17] F. Dabek, J. Li, E. Sit, J. Robertson, M. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation.*, March 2004.

[18] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[19] distributed.net. `http://www.distributed.net/`.

[20] P. Eerola, T. Ekelof, M. Ellert, J. R. Hansen, A. Konstantinov, B. Konya, J. L. Nielsen, F. Ould-Saada, O. Smirnova, and A. Waananen. The NorduGrid architecture and tools. In *Computing in High Energy and Nuclear Physics*, 2003.

[21] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient broadcast in structured p2p networks. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[22] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 239–252, March 2004.

[23] L. Garcés-Erice, P.A. Felber, E.W. Biersack, and G. Urvoy-Keller. Data indexing in peer-to-peer DHT networks. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, March 2004.

[24] Great Internet Mersenne prime search. `http://www.mersenne.org/`.

[25] The Globus Alliance. `http://www.globus.org/`.

[26] Omprakash D. Gnawali. A keyword-set search system for peer-to-peer networks. Technical report, Massachusetts Institute of Technology, June 2002.

[27] Knowbuddy's Gnutella FAQ. `http://www.rixsoft.com/Knowbuddy/gnutellafaq.html`.

[28] Krishna P. Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM 2003*, August 2003.

[29] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the SIGCOMM Internet Measurement Workshop (IMW 2002)*, Marseille, France, November 2002.

[30] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[31] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of USITS*, 2003.

[32] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[33] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina 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.

[34] David R. Karger and Matthias Ruhl. Diminished chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, pages 288–297, 2004.

[35] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43. ACM Press, 2004.

[36] LHC@home. Internet home page. `http://lhcathome.cern.ch/`.

[37] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[38] V. Lo, D. Zhou, D. Zappala, Y. Liu, and S. Zhao. Cluster computing on the fly: P2P scheduling of idle cycles in the internet. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, 2004.

[39] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, March 2002.

[40] Petar Maymounkov and David Mazieres. Rateless codes and big downloads. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[41] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 124–139, Kiawah Island, South Carolina, December 1999.

[42] Alberto Montresor, Heing Meling, and Ozalp Babaoglu. Messor: Load-balancing through a swarm of autonomous agents. In *Proceedings of the 1st International Workshop on Agents and Peer-to-Peer Computing*, July 2002.

[43] p2psim: a simulator for peer-to-peer (p2p) protocols. `http://pdos.csail.mit.edu/p2psim/`.

[44] Vinay Pai, Karthik Tamilmani, Vinay Sambamurthy, Kapil Kumar, and Alexander Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems*, February 2005.

[45] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[46] Sriram Ramabhadran, Joseph M. Hellerstein, Sylvia Ratnasamy, and Scott Shenker. Prefix hash tree: An indexing data structure over distributed hash tables, 2004.

[47] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured p2p systems. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[48] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[49] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *ACM/IFIP/USENIX International Middleware Conference*, June 2003.

[50] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[51] Rodrigo Rodrigues, Barbara Liskov, and Liuba Shrira. The design of a robust peer-to-peer system. In *10th ACM SIGOPS European Workshop*, Saint Emilion, France, September 2002.

[52] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[53] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking 2002*, January 2002.

[54] Michael Shirts and Vijay S. Pande. COMPUTING: Screen savers of the world unite! *Science*, 290(5498):1903–1904, 2000.

[55] John F. Shoch and Jon A. Hupp. The "worm" programs – early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, 1982.

[56] An analysis of the skype peer-to-peer internet telephony protocol, 2004.

[57] Ion Stoica, Robert Morris, David Karger, M. Francs Kaashoek, and Hari 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.

[58] Niklas Therning. Jalapeno – decentralized grid computing using peer-to-peer technology. Master's thesis, Chalmers University of Technology, 2003.

[59] Andrei Tsaregorodtsev, Vincent Garonne, and Ian Stokes-Rees. DIRAC: A scalable lightweight architecture for high throughput computing. In *5th International Workshop on Grid Computing (GRID 2004)*, pages 19–25, 2004.

[60] UNICORE web site. `http://www.unicore.org/`.

[61] Ye Xia and Alin Dobra. Ideal load balancing techniques in structured peer-to-peer networks. Submitted for publication.

[62] Ye Xia and Vivekanand Korgaonkar. Data replication with multiple hash functions in structured peer-to-peer networks. Draft.

[63] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.