NUBIS

A DECENTRALISED, FLEXIBLE, FAULT-TOLERANT AND SCALABLE FOUNDATION FOR COMPUTATIONAL GRIDS

December 2004

AALBORG UNIVERSITY
Group B2-201

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

Title:

Nubis – A Decentralised, Flexible, Fault-Tolerant and Scalable Foundation for Computational Grids

Project period:
DAT5, Sep. 3rd – Dec. 22th, 2004

Project group:
B2-201

Members of the group:
Thomas Christensen
Anders Rune Jensen
Rasmus Aslak Kjær
Lau Bech Lauritzen
Ole Laursen

Supervisor:
Gerd Behrmann

Number of copies: 7

Report – number of pages: 82

Abstract

This report describes the design of Nubis, a decentralised, flexible, fault-tolerant and scalable foundation for computational grids. The design is based on a distributed file system with support for access control, mutual exclusion and notification of file changes. The distributed file system is based on the concept of distributed hash tables, resulting in a scalable, robust information service. On top of this, the report describes the design of a data service and outlines the design of grid resource and user components.

A prototype implementation of the information service is developed in C++ and tested. The tests are performed with a network of 994 participating nodes and are mostly promising.

The design is in its infancy and there are many unanswered questions that the report does not address, in particular with respect to the resource components and scheduling. However, the design is robust compared to the currently deployed grids, and also appears to be more scalable and flexible.

Preface

This report documents the design of a decentralised, flexible, fault-tolerant and scalable foundation for computational grids. The design has been developed as part of the DAT5 semester at the Department of Computer Science, Aalborg University.

We would like to thank Josva Kleist for letting us use part of the university cluster for our tests and for suggesting problem areas in NorduGrid, and Henrik Thostrup Jensen for helping us understand NorduGrid.

Aalborg, December 2004,

Thomas Christensen — *Anders Rune Jensen*

Rasmus Aslak Kjær — *Lau Bech Lauritzen*

Ole Laursen

Contents

Chapter 1

Introduction

In this chapter we present the idea of a computational grid, going into some detail with a particular implementation, NorduGrid Advanced Resource Connector, and the current problems with that implementation. We then propose a different architecture, review recent advances within the field of distributed file systems for supporting our proposed architecture, and finally state the system requirements.

1.1 Computational Grids

There is some controversy over what constitutes a grid, and the term has been much hyped in the recent years. In the following, we adopt this definition by Rajkumar Buyya [8]:

> *Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements.*

The term is a reference to the power grid [16]. The users of the power grid are oblivious to the exact nature of the grid and simply plug in their appliances, expecting that the appropriate amount of electrical current is delivered to them. Furthermore, the power producers can range from a farmer with a small windmill in his field to large national power plants. Potentially anyone can participate in the power grid as a producer or consumer.

This concept has given birth to the idea of computational grids, i.e. that computers from diverse locations are interconnected to let users share the resources. The idea is illustrated in Figure 1.1. Users should be able to connect to the grid and submit tasks without having to concern themselves with where the tasks are run (unless they want to), and resources should be able to connect

to the grid and receive tasks that are appropriate for them. Resources can be computational resources or storage, or even access to specialised equipment such as measurement instruments. The goal is to have better utilisation and wider access to resources, even across traditional organisational boundaries.

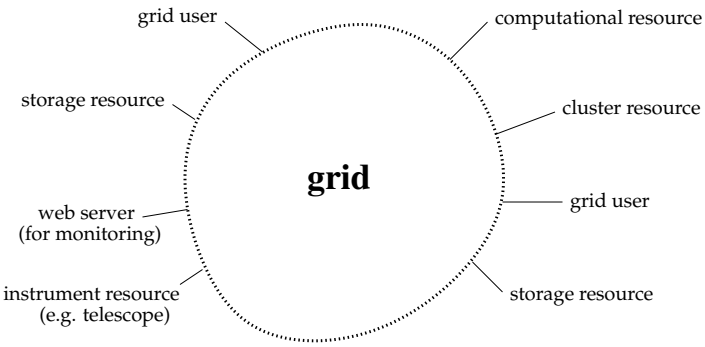


Figure 1.1: A conceptual view of a grid with resources and users.

Taking the idea to the extreme one can imagine one giant computational grid consisting of all computers connected to the Internet. But currently most users seem to be scientists with demanding computational tasks, and the grid designs are either geared towards utilising high-performance computing clusters as the resources, or screen saver science where ordinary desktop machines work on small part of a problem when the processor is idle. A popular example of the latter is the SETI@home (Search for Extraterrestial Intelligence) project [43], which is an effort to analyse radio telescope data for signals from intelligent beings.

1.1.1 Challenges

Since a grid by nature is a large distributed system that spans many organisations, there are several challenges that must be overcome in the design of a grid:

- Security. The security model must be flexible enough to support modelling the diverse conditions in the various organisations – the local resource policies must be respected – and modelling of virtual organisations [17] spanning multiple organisation for resource sharing. This problem includes authentication and authorisation of users and resources, and enforcement of access control.

- High availability and fault-tolerance. A large grid will continually suffer hardware failures and must be able to cope gracefully with them and at the same time stay up. It should also take precautions against losing jobs and data.

- Flexibility. Much is expected of grid technology, so a grid design should preferably be flexible enough to be able to support future use cases.
- Ease of use. A successful grid relies on many parties adopting the idea, reaching a critical mass, so it must not be too difficult or cumbersome to use or setup a resource for the grid.
- Scalability. A grid must be able scale to a large number of users and resources. This affects both administration of the grid and the operation of the software itself – e.g. is it not possible for a design that relies on a central authority to scale to something the size of the Internet.
- Privacy. Protecting the privacy of information when submitting jobs to the grid. Jobs can work on valuable data sets, and situations can arise where it is unacceptable that these data can be intercepted and read by anyone other than the job itself and the submitter.
- Accounting. High performance computing resources are not cheap to acquire and maintain. So as it is the case with the power grid, accounting of resources spent by consumers is needed if the grid is not to be operated gratis. This can be imagined to evolve into grid economy where resources receive micropayments from job submitters in return for running jobs.
- Portability and heterogeneity. The portability of the grid software is also important. Ideally there should be no technical boundary preventing resources from participating in the grid. But even if the grid software itself is portable, the problem of making the job software run on diverse architectures remains.

None of these issues are trivially solved. The current grid designs, of which some are reviewed in the next section, have mostly dealt successfully with only a subset of them. A single design that is flexible enough to cover all grid use cases and at the same time proves to solve the other issues has yet to reveal itself.

1.1.2 Grid Efforts

There exists several grid toolkits for aiding the implementation of grids and work has also been done to form standards for grid architectures.

The Globus Alliance is developing a toolkit for building a grid, including frameworks for building information systems for registration of resources and user authentication, replica location systems for management of data, and direct interfaces to resources. The Globus Toolkit is currently working on a fourth version, where the major focus seems to be on standardising communication as web services [19].

Parts of the Globus Toolkit has been used as the basis for several grid solutions, both in national grids and international efforts. One of the larger currently operating grids is NorduGrid, discussed in further detail in Section 1.2. Other large grids include the Large Hadron Collider Grid Computing Project [29] at CERN, the Grid Physics Network (GriPhyN) [5] and Teragrid [9] (which both are developed and run by American universities).

There are, however, other grids that are not based on Globus. Uniform Interface to Computing Resources (UNICORE) [50] is a mostly German grid solution where the middleware is implemented entirely in Java to facilitate running the software in heterogeneous environments.

Distributed Infrastructure with Remote Agent Control (DIRAC) [49] is a grid middleware programmed in Python and based on a pull model of scheduling where resources themselves request tasks from the central job queue when they are ready, as opposed to e.g. NorduGrid where the resources passively receive jobs.

Grid Datafarm [46] is both a grid middleware and a plan for how to arrange the hardware for data intensive computing. The idea is to setup the participating clusters so that each cluster node has its own fast and large disk system (through RAID striping). The grid middleware takes care of moving input data to the nodes so that they have direct access to it – it tries to do this efficiently by taking data locations and transfer costs into account in its scheduling.

All of the above are basically aimed at combining clusters, large batch systems, into a grid as an even larger batch system. Another model is to try to hide the distributed nature of the grid by adopting a distributed operating system or object system. Legion is an effort to assimilate computers into one world-wide virtual computer. In contrast to the Globus Toolkit, Legion is an integrated solution that does not use existing services and technologies, but instead has its own unified object model [28].

The grid models that are aimed at utilising idle desktop machines also provide a specialised architecture. Some examples of projects are Berkeley Open Infrastructure for Network Computing (BOINC) [2] which is the underlying technology used for SETI@Home among others, distributed.net [12] which is cracking encryption algorithms, and Alchemi which is a .NET framework [1].

1.2 NorduGrid

The NorduGrid project started in May 2001 as a collaborative effort between research centers in Denmark, Norway, Sweden and Finland. The project was initially named the Nordic Testbed for Wide Area Computing and Data Handling [15]. Continuous development and challenges has matured the project beyond the state of a testbed and into a stable grid. At present the grid development, resources and usage are administered by the Nordic Data Grid Facility (NDGF), which is part of the North European Grid Consortium. The grid,

however, is still refered to as NorduGrid.

The middleware developed by NDGF is called the NorduGrid Advanced Resource Connector, or simply the NorduGrid ARC. This section presents the purpose and the present state of NorduGrid and the architecture of the NorduGrid ARC, and concludes with a discussion of the problems in the architecture, design and implementation.

1.2.1 History of the Project

The purpose of the NorduGrid project was to create a testbed for a Nordic grid infrastructure and eventually to develop a grid that meets the requirements for participating in the ATLAS experiment [15]. ATLAS, A Toroidal LHC ApparatuS, is a detector for conducting high-energy particle physics experiments that involve head-on collisions of protons with very high energy. The protons will be accelerated in the Large Hadron Collider, an underground accelerator ring 27 kilometres in circumference at the CERN Laboratory in Switzerland. The ATLAS experiments have around 1800 physicists participating from more than 150 universities in 34 countries; the actual experiments are planned to begin in 2007 [4].

The experiments are expected to generate 12-14 petabytes data per year that needs to be analysed. To prepare the participating entities for these massive amounts of data, the ATLAS Data Challenges have been posed. The first of these data challenges ran throughout 2002 and was completed in 2003. NorduGrid was the Scandinavian contribution to this challenge [14].

During the initial phase of designing the grid, the fundamental idea was that NorduGrid should be built on existing pieces of working grid middleware and the amount of software development within the project kept at a minimum. Once the basic functionality was in place the middleware was to be further extended, gradually turning the testbed into a production grid. And at the start of the NorduGrid project in May 2001 a general design philosophy was formulated as follows:

- Start with simple things that work and proceed from there
- Avoid architectural single points of failure
- Should be scalable
- Resource owners retain full control of their resources
- As few site requirements as possible
  - No dictation of cluster configuration or install method
  - No dependence on a particular operating system or version
- Reuse existing system installations as much as possible

- The NorduGrid middleware is only required on a front-end machine
- Computational resources are not required to be on a public network
- Clusters need not be dedicated to grid jobs

Existing toolkits were then examined to decide on which to base the NorduGrid middleware. The available possibilities were narrowed down to two alternatives; the Globus Toolkit and the software developed by the European DataGrid project (EDG). But the initial idea to use only existing middleware components, and not having to develop any software in the project soon proved impossible to realize.

The Globus Toolkit lacked job brokering facilities and the Grid Resource Allocation Manager seemed unable to handle the large amounts of input and output data. The EDG software seemed to alleviate many of the deficiencies of the Globus Toolkit, but was not available in a mature state (in the beginning of 2002). Furthermore, the software from EDG employed a centralised resource brokering scheme possibly becoming a major bottleneck. The Globus Toolkit was decided upon as the basic foundation for the NorduGrid ARC. Hence, the NorduGrid ARC was built on top of Globus by extending and modifying the existing components of the toolkit [15].

1.2.2 Architecture and Design

The NorduGrid tools are designed to handle the aspects of using, maintaining and administrating a grid. This includes job submission and management, user management, data management and monitoring. The three most important parts of the NorduGrid ARC architecture are:

- A client-server architecture for the resources where each cluster has runs a server, the grid manager, which is contacted directly by the grid users that wish to submit jobs.
- A hierarchically distributed system, the information service, for registering the resources so that the users can discover them.
- A client-server architecture for the data distribution in the form of file servers that are accessed with an extended version of FTP, GridFTP.

We first illustrate the architecture with an example of the task flow when submitting a job and then afterwards describe some of the components of the design.

Job Submission Task Flow

A job submission scenario is depicted in Figure 1.2 where the numbers indicate the order of the tasks involved in submitting a job.

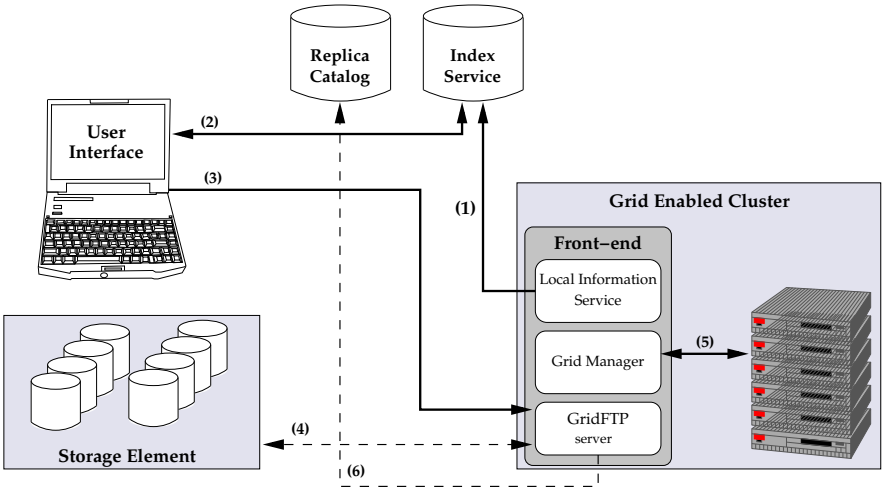


Figure 1.2: An overview of the NorduGrid ARC architecture. The bold lines indicate the flow of control and data in the operation of the grid. The dashed lines indicate optional data flow depending on job specific properties.

1. To enter NorduGrid, a cluster (or resource in general) must run three distinct services; a grid manager service, an information service and a GridFTP service. The information service collects local information and, as illustrated by the first arrow, registers itself with the index service to signal its arrival in the grid.

2. The user prepares a job description and submits it through the user interface. Before actually submitting the job, the user interface performs resource brokering by matching the resource requirements specified in the job description with existing clusters in the grid. Any cluster that matches the requirements are eligible for processing the job and the user interface selects one based on scheduling policies.

3. Upon selecting the destination cluster where the job will run, the user interface uploads the job description to the selected cluster by means of the GridFTP server running at the front-end of that cluster.

4. The uploaded job description is detected and subsequently parsed by the grid manager running at the receiving cluster. The grid manager creates a session directory where all information and data pertaining to the job will eventually be stored. Depending on the job description the input data for a job is either fetched from a remote location, as illustrated by the fourth arrow, or uploaded along with the job description in the previous step.

5. When the input data for a job is available in the session directory, the grid manager at the cluster submits the job to the local resource management system for execution. Currently NorduGrid supports a variety of such

systems, e.g. OpenPBS, PBSPro, TORQUE, Condor and N1 Grid Engine [27].

6. When a job has completed successfully the grid manager processes the output data according to the job description and places the result either at a remote location, as illustrated by the sixth arrow, or simply leaves the output data in the session directory. Finally, if requested by the user, the grid manager notifies him about the completion of the job, e.g. by email.

Grid Manager

The grid manager is the primary interface between the grid and the local resource. It runs on the front-end machine of every cluster in NorduGrid and handles incoming job submissions, including the pre- and post-processing of data related to the jobs.

The grid manager is implemented as a layer on top of the Globus Toolkit with integrated support for a replica catalogue, which is used for sharing of cached files among users and for staging input/output data. The grid manager interfaces with a GridFTP server for job submission.

For every job the grid manager creates a session directory and stores the input files for the job in it. Since the gathering of job and input data is performed by the cluster front-end in combination with a specific user interface, there is no single point that all jobs have to pass through in NorduGrid.

Users can either upload files directly to the grid manager or it can be instructed to download them on its own using a variety of protocols, such as HTTP, FTP, GridFTP, etc. When all input files are present in the session directory for a given job the grid manager creates an execution script that, besides executing the job, handles the required configuration of environments for third party software or libraries.

After a job has finished executing, output files can be transferred to remote locations by specifying it in job description, or alternatively be left in the session directory for later retrieval by the user. The grid manager can also register files with replica catalogues if the job description request it.

Replica Catalogue

The replica catalogue in NorduGrid is used for registering and locating data sources. It is based on the replica catalogue provided by the Globus Toolkit with a few minor changes for enhanced functionality.

The changes primarily improves the ability to handle the staging of large amounts of input and output data for jobs and the adds the ability to perform authenticated communication based on the Globus Security Infrastructure mechanism. Objects in the replica catalogue are created and maintained by the grid managers running at each resource in NorduGrid, but can also be accessed by the user interface for resource brokering.

Information System

The NorduGrid ARC implements a distributed information system which is created by extending the Monitoring and Discovery Services (MDS) provided by the Globus Toolkit. In NorduGrid there is an MDS-based service on each resource and each of these is responsible for collecting information on the resource on which it is running.

The MDS is a framework for creating grid information systems on top of the OpenLDAP software. An MDS-based information system consists of the following:

- An information model defined as an LDAP schema.
- Local information providers.
- Local databases.
- Soft-state registration mechanisms.
- Index services.

We describe each in turn in the following.

The **information model** employed by the original MDS is oriented towards single machines as the unit of computation and as such not well suited to describe the cluster-based approach of NorduGrid. The NorduGrid information model is a mirror of the architecture and hence describes its principal elements, ie. clusters, users and jobs. The information about these elements is mapped onto an LDAP tree, creating a hierarchical structure where every user and every job has an entry. Replica managers and storage elements are also described in the information system although only in a simplistic manner.

Local information providers are small programs that generates LDAP entries as replies to search requests. The NorduGrid information model requires that NorduGrid specific information providers are present on the front-end of each cluster. The information providers interfaces with the local batch system and the grid manager to collect information about grid jobs, users and the queueing system of the grid. The collected information is used to populate the local databases.

The **local databases** are responsible for implementing the first-level caching of the LDAP entries generated by the information providers. Furthermore, they are responsible for providing the requested grid information used for replying to queries through the LDAP protocol. Globus includes an LDAP back-end called the Grid Resource Information Service (GRIS). NorduGrid uses this back-end as its local information database. The local databases in NorduGrid are configured to cache the output of the information providers for a limited period.

The contact information in the local databases is registered as **soft-state** in the index services which in turn can use soft-state registration in other indices.

This means that resources must keep registering themselves periodically to avoid being purged from the information system.

The **index service** of NorduGrid is used to maintain dynamic lists of available resources. A record in the index service contains the contact information for a soft-state registered resource. A user must then query the local databases at the resources for further information which reduces the overall load on the information system. The Globus developed back-end for the index service, the Grid Information Index Service (GIIS), uses a hierarchical topology for the indices, as illustrated in Figure 1.3, where local information providers register with higher level GIIS services which in turn register with the highest level GIIS services.

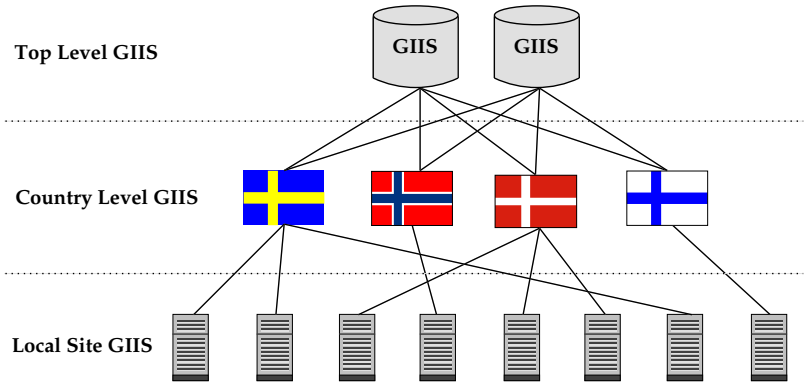


Figure 1.3: The hierarchical structure of the information system in NorduGrid. Higher level GIIS hosts should be replicated to avoid a single point of failure in the information system.

User Interface and Brokering

Users of NorduGrid interact with the grid through the user interface which has commands for submitting jobs, for querying the status of jobs and clusters and for killing jobs. Commands for managing input and output data on storage elements and replica catalogues are also included.

The user interface is also responsible for scheduling the job by choosing an appropriate cluster to run it on.

GridFTP server

GridFTP is the transfer protocol used for all data transfers in the NorduGrid ARC even for job submission. To submit a job, a job description is uploaded to a cluster using GridFTP. GridFTP is a modified FTP server provided by the Globus Toolkit, however, NorduGrid uses its own modified implementation that is better suited for interfacing with the NorduGrid ARC.

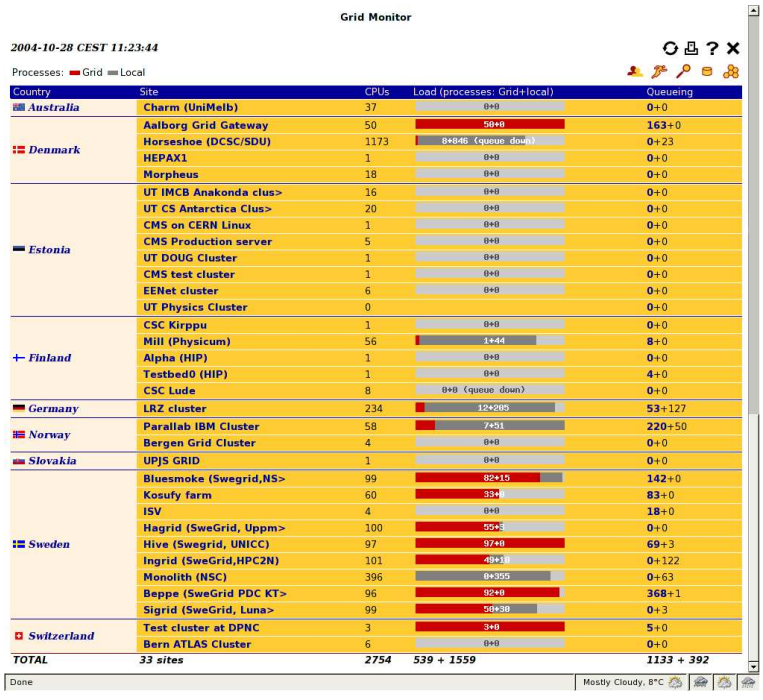
Storage Element

A storage element is a separate service that handles storage of data in the grid. A job description can specify that input data should be downloaded from a storage element. In its current incarnation storage elements are mere GridFTP servers but recently effort have been put into extending the capabilities of the storage element service. The NorduGrid Smart Storage Element (SSE) is supposed to be a replacement of the current storage element, will be based on standard protocols such as HTTPS, Globus GSI and SOAP, and will provide flexible access control, data integrity between resources and support for autonomous and reliable data replication [34].

Monitoring

NorduGrid provides a web-based monitoring tool for browsing the grid information system. It allows users of the grid to view all published information about the currently active resources.

The structure of the monitoring tool corresponds to the hierarchical structure of the information system itself. Hence, the initial screen of provides an overview of the entire grid, e.g. the number resources in terms of CPUs, the length of queues etc. The user can browse deeper into the hierarchy and at the lowest level inspect the queues, available software, hardware platform and so forth on a selected resource.



Country	Site	CPUs	Load (processes: Grid+local)	Queueing
Australia	Charm (UniMelb)	37	0+0	0+0
Denmark	Aalborg Grid Gateway	50	50+0	163+0
	Horseshoe (DCSC/SDU)	1173	0+846 (queue down)	0+23
	HEPAX1	1	0+0	0+0
	Morpheus	18	0+0	0+0
Estonia	UT IMCB Anakonda clus>	16	0+0	0+0
	UT CS Antarctica Clus>	20	0+0	0+0
	CMS on CERN Linux	1	0+0	0+0
	CMS Production server	5	0+0	0+0
	UT DOUG Cluster	1	0+0	0+0
	CMS test cluster	1	0+0	0+0
	EENet cluster	6	0+0	0+0
	UT Physics Cluster	0		0+0
Finland	CSC Kirppu	1	0+0	0+0
	Mill (Physicum)	56	1+44	8+0
	Alpha (HIP)	1	0+0	0+0
	Testbed0 (HIP)	1	0+0	4+0
	CSC Lude	8	0+0 (queue down)	0+0
Germany	LRZ cluster	234	12+205	53+127
Norway	Parallab IBM Cluster	58	7+51	220+50
	Bergen Grid Cluster	4	0+0	0+0
Slovakia	UPJS GRID	1	0+0	0+0
Sweden	Bluesmoke (Swegrid,NS>	99	82+15	142+0
	Kosufy farm	60	33+0	83+0
	ISV	4	0+0	18+0
	Hagrid (SweGrid, Uppm>	100	95+5	0+0
	Hive (Swegrid, UNICC)	97	97+0	69+3
	Ingrid (SweGrid,HPC2N)	101	49+10	0+122
	Monolith (NSC)	396	0+355	0+63
	Beppe (SweGrid PDC KT>	96	92+0	368+1
	Sigrid (SweGrid, Luna>	99	50+30	0+3
Switzerland	Test cluster at DPNC	3	3+0	5+0
	Bern ATLAS Cluster	6	0+0	0+0
TOTAL	33 sites	2754	539 + 1559	1133 + 392

Figure 1.4: The initial window of the NorduGrid monitoring web site.

The web server that provides the grid monitoring tool runs on a machine independent from the grid itself. Hence, it does not in itself incur any load on the information system. However, the web interface refreshes its information every 30 seconds by performing LDAP queries, which ultimately does burden the information system.

1.2.3 Identified Problems

A closer examination of the NorduGrid ARC and conversations with the users reveals a number of issues. The problems relate both to the architecture and design and to implementation-specific details. Below is a brief description of some of the major identified problems. We have categorised the issues according to two of the central desirable properties for a grid in general, robustness and scalability.

Robustness Issues

An architectural problem is that the hierarchical topology of the index servers is not robust since it results in single points of failure. When requesting information for a job submission, the tree must be traversed from the root towards the leaves so if a higher-level server crashes, it will take down the part of the information system below it. This makes some resources disappear from the grid even if they are perfectly reachable manually.

The problem is known and supposed to be solved by having a replicated fail-over server for each high-level server, but this is really an undesirable solution since it requires twice the resources and twice the maintenance. It is currently not done in NorduGrid either, probably for this reason.

Furthermore, the client-server job submission architecture where a user submit his job directly to the grid manager at a resource makes the resource a single point of failure. If the resource crashes, then the job has disappeared from the grid and has to be resubmitted by the user.

A related problem is that the storage destination for the output of a job also is contacted in a client-server manner, so if the destination server has failed, the output may be lost because the grid manager does not have room for the data locally and cannot send the results elsewhere. This means a successful job run that may have taken hours to process is lost, and the job must be resubmitted.

There are also some problems with the engineering of the implementation. The approach of reusing pieces of existing middleware results in a large incoherent system consisting of a lot of distributed subsystems that are difficult to understand. The plethora of programming languages used does not help.

Scalability Issues

There are two inherent scalability issues with the hierarchical structure of the information system. One problem is contention at the tree root – since all queries visits the root, it will become a hot spot when the number of users grow. DNS has the same problem, even though it utilises an aggressive caching scheme.

The other problem is that scheduling a job involves traversing all nodes in the information system, including all leaf nodes. With many resources, this becomes too costly and there is no facility in NorduGrid ARC for supporting other means of scheduling.

Even with the current number of resources, it means that the latency of a scheduling operation is at least the latency of the slowest traversal from the root to a leaf node. A slow resource thus dominates all scheduling operations. Combined with the fact that much of the grid manager design is based on polling, a simple job that just calls the UNIX command `uname` takes several minutes to execute even when the resource running the command is otherwise free.

The scheduler also has a problem with load balancing. Because of the inherent delays in the design, it is possible for several users to submit their jobs simultaneously to the same lightly-loaded resource and overwhelm it. This is more likely to happen with many users. A related problem is that a job with few resource requirements, e.g. with respect to memory, may be submitted to a resource that can run jobs with high requirements. If such jobs later arrive at the resource, it would have been beneficial to migrate the job with few requirements to another resource.

Another problem is the way access control is managed. Users are required to obtain a certificate from the NorduGrid certification authority which makes it possible to set up access control at the resources by maintaining a database with trusted certificates. Unfortunately, these databases are currently maintained manually and in a quite centralised manner, as is the certification process. This does not scale at all to a large number of users, and even today burdens the administrators and involves high latencies.

1.3 A Simpler Grid Architecture

Some of the current problems in NorduGrid are software engineering issues that a major reimplementation could alleviate. But some of the problems are inherent in the architectural model that is being used. The NorduGrid ARC is based mostly on a traditional client-server architecture like the World-Wide Web where job submitters are clients and each resource server is operating completely independently of the other resource servers. The problem with this approach is that tasks like scheduling and smarter job handling on the grid need coordination and cooperation, which a collection of completely indepen-

dent servers are unable to do.

To solve this problem, the NorduGrid employ extra distributed systems to take care of any issues that require coordination, such as the information service for resource registration. These extra systems must be designed, implemented and tested separately which is a complicated task given their distributed nature. As the case of NorduGrid ARC shows, the result is not as robust as it could be. Also, both grid users and grid resources must support several communication protocols. The resulting software grows large and tends to become incoherent and difficult to understand.

A much simpler model, and what we propose, is an architecture that replaces the myriads of distributed systems with a single global grid entity similar to the original grid idea illustrated in Figure 1.1. The global grid entity acts as an intermediary between the participants in the grid and is used to coordinate the resources available, reducing the interface for both grid users and resources to the connection to the grid entity. [1]

The question is how to design such a global grid entity. We assert that the basic problem to be solved is that of information sharing; the coordination and decision making of a global grid entity rests upon having access to all necessary information and for the grid participants being able to update the information as jobs are submitted and processed and resources come and go.

We propose an architecture where the information is placed in a hierarchic distributed file system. This is convenient because a hierarchic file system specifies both a naming system for structuring information through directories and means for getting to the actual data. This seems ideal in providing a firm basis to build the rest of the system on. Setup of virtual organisations, job submission and handling, resource registration, efficient handling of large amounts of data, etc. can all be done with a distributed hierarchic file system.

The amount of data stored in the file system is expected to be for coordination mostly and hence small, thus it can be a light-weight file system. But it must be distributed, robust and scalable. It must also provide some means of synchronisation, e.g. mutual exclusion, to avoid race conditions to be useful for coordination purposes.

Some alternatives to a hierarchic file system are object directory services and database based file systems.

Lightweight Directory Access Protocol [51] (LDAP) is a common object directory protocol, which is widely used as a directory for Internet email adresses. LDAP is tuned towards write-once-read-many and as such not suited for the dynamics of an information system where the data is updated frequently.

Database-based file systems are an emerging technology in desktop systems development with efforts like GNOME Storage [35], Microsoft WinFS [22] and DBFS [21]. The idea is to store much more metadata than with conventional hierarchic file systems, for instance what email did a certain text origi-

[1] We thank the Minimum Intrusion Grid project [33] for inspiration for formulating this model.

nate from or even version control on a line by line basis. These metadata can then be queried through a database interface.

The desktop centric focus of current database file systems is clearly not needed for the information system, however a light-weight database file system could be suitable for the information system. Implementation of a large distributed database file system seems, however, a daunting task.

Since the capabilities of the file system will affect the overall capabilities of the design to a large extent, the following section reviews relevant distributed file systems before we state the system requirements in Section 1.5.

1.4 Distributed File Systems

Distributed file systems have been in use for two decades since Sun Microsystems introduced Network File System (NFS) [45] in 1984. NFS, like the Server Message Block (SMB) and Common Internet File System (CIFS) [3] protocols (used for Microsoft Windows shares), represents a straight-forward client-server model where a central server stores the data and the clients connect to it whenever they need to fetch a file. A virtue of this model is that the data remains under complete control in the central server. For instance, this makes it relatively easy to design an access control mechanism.

Unfortunately, the central server is a single point of failure and becomes a bottleneck under heavy load; it provides neither high availability or scalability. Variations of the centralised idea with multiple replicating servers and advanced caching schemes for improved performance and higher availability include the Andrew File System (AFS) and the Coda file system [11]. But a drawback of these systems is their added complexity and higher cost. Their centralised nature is also still a potential bottleneck and makes it difficult to scale to something at the size of the Internet.

Gnutella [20] abandons the idea of centralisation in favour of making the clients peers that simultaneously retrieve files as clients and store files as servers, keeping track of other known peers. The distribution of data on the resulting peer-to-peer or overlay network has no particular structure, each node in the network simply offers the files that the user on that node has available, so to find a specific file it is necessary to broadcast the query on the network. Several other popular file-sharing systems also operate with peers, including Kazaa [26] and eDonkey [13].

The benefits of the peer approach is that there is no single point of failure, and that the available resources scale with the number of participating nodes, since nodes are both clients and servers. Unfortunately, the idea of query broadcasting does not scale well, since in the worst case the query must be forwarded to every node in the network (else the requested file may not be found). Furthermore, synchronisation mechanisms such as locks cannot easily be added.

A recent approach to storing data in a decentralised overlay network with

more structure is the concept of distributed hash tables [31; 40; 42; 44; 53]. In a distributed hash table, nodes are assigned a unique key when they enter the overlay network, as are data items. This means that one can think of the hosts and the data items as being mapped to a key space.

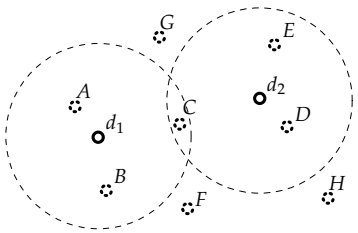


Figure 1.5: Example of a key space with two data items, d_1 and d_2, and a number of nodes (the dotted small circles). With $r = 3$, node A, B and C must store d_1, C, D and E must store d_2 whereas F, G and H need not store anything.

The basic idea is then that a given data item is stored on the r nodes that are closest to it in key space (r is some small integer constant), see Figure 1.5. This of course supposes that one defines a distance metric in the key space. The keys and the distance metric is what gives the overlay network a structure – given a key, a data item is retrieved simply by contacting one of the r hosts closest to the key. Fault-tolerance is ensured by the fact that r replicas are distributed so at least r nodes must crash or leave the overlay network before there is a possibility of data loss – and the network can heal itself by letting the sets of r neighbours contact each other regularly to detect missing replicas.

For actually finding the closest nodes without having to know every node in the network, the nodes must continually ensure that they know at least the r closest nodes to themselves. Furthermore, they also maintain a list of nodes that are farther away to facilitate look-up in sub-linear time, usually $O(\log(n))$ with n being the number of nodes in the network. This feature, that the nodes only know of a subset of the other nodes and still are able to find a particular data item in logarithmic time, has the effect that the structured decentralised overlay networks can scale to become Internet-wide while staying coherent.

Another benefit of the structured overlay networks is that it is possible to exploit the structure not just to store data and locate replicas, but also to implement locks and notification of events (we show how this can be done in Section 2.2).

The main difference between the individual overlay networks is their topology, how nodes are organised with the distance metric. One conceptually simple topology is a ring where the distance between two points on the ring is the size of the angle in the clock-wise direction between them (this is used by Chord [44]). Another topology result from prefix-based routing, used by Tapestry [53] and Pastry [42], where the routing is based on matching the prefix of the destination address. A particularly simple distance metric is to XOR the

keys of two points and interpret the result as an integer. This gives in topology that can be visualised as a tree which lends itself well to an implementation. Kademlia [31] does this.

Using distributed hash tables can lead to hot spots in the network where frequently read keys are not distributed sufficiently. One solution for this is distributed sloppy hashing [18] where nodes organise themselves into clusters based on network proximity, caching keys inside clusters. But this creates cache coherency problems when updating data, as it is expensive to locate all cached copies of the data item to perform the update. Hence, sloppy hashing is most appropriate with few writes and many reads.

1.5 System Requirements

In this section we present the requirements for our grid software. We aim to design a grid system that is functionally equivalent to the current NorduGrid ARC with some exceptions. That is, a grid system that allows resources to be shared among the users of the grid, across administrative and organisational domains. Returning to Figure 1.1, the system to be implemented is the global grid entity.

1.5.1 Basic Requirements

From the point of view of the users, the grid must support:

- Submission of jobs without requiring the user to find a resource, and in a flexible manner that can support diverse types of resources.
- Monitoring of jobs, including getting a notification when a job is completed.
- Managing of jobs. The details depend on the type of job, but at least cancelling the job.
- Collections of jobs so that a large batch can be treated easily.
- Handling of the input data for the job.

From the point of view of the resources, the system must support:

- Retrieving a job and processing it without having to concern the resource with who submitted it.
- Registration of the resource, with flexible commitment so that the resource owner remain in control.
- Efficient handling of data, i.e. retrieval of data from a storage resource and sending output back.

Since the focus is on the current NorduGrid ARC functionality, the system must also specifically support cluster resources and storage resources. The cluster resource must provide a flexible interface to the local cluster software to support different cluster batch systems. The storage resources must provide data space and support handling large amounts of data efficiently, support replication to guard against failures and file collections to make it easy to deal with complex data sets.

Furthermore, the global grid entity must support:

- The basic infrastructure for the grid for distributing information and coordination.
- Modeling of virtual organisations that consists of users, resources and jobs and restricts access for all others.
- Pairing of jobs and resources to avoid having the user do it.
- Any maintenance that goes beyond one grid participiant.

1.5.2 Requirements for Design

There are also some abstract requirements for the system which must be:

- Scalable. It must employ an architecture that is more scalable than the current NorduGrid architecture. Centralised administration must also be avoided to avoid ending with a human bottleneck.
- Robust and highly available, really preventing any single point of failure.
- Flexible. As mentioned in Section 1.1.1, a grid should be designed with flexibility in mind, so that additional features and support for new resources can be added in further development.
- Easy to setup, maintain and access, since grids currently require gentle cooperation and mostly thrive on people's goodwill.

1.6 Project Overview

Based on the considerations in this chapter, we have come up with a design for fulfilling the requirements presented in the previous section. We have also implemented and tested some of the most basic parts of this design. The rest of the report describes our efforts.

Chapter 2 presents the design of a basic infrastructure, the decentralised distributed file system, how it implements the features that are necessary for building the rest of the grid on top of it and how it enables modeling of virtual organisations. A simple design for handling transfers of large amounts of

data is also presented together with an outline of common resource types. Our design enables both scheduling, a *push* model, and simple job selection, a *pull* model.

Due to time constraints we have primarily focused our attention on the foundations of the architecture, and only sketched the upper level components, including the resource and user components.

After having the design of the basics ready, we have implemented in C++ a working prototype of the features needed for the decentralised distributed file system. Chapter 3 presents the tests performed on the prototype to evaluate its performance and feasibility. Access control features, a mutual exclusion algorithm and notification of file changes have been tested and the results are presented and evaluated.

Chapter 4 concludes by presenting a summary of the project, an evaluation of our design and a discussion of future work.

Chapter 2

Design

This chapter presents the design of our proposed system. Section 2.1 gives an overview of the architecture and the components, and the following sections explains the detailed design of each component.

2.1 Overview

As explained in Chapter 1, we base the design on a decentralised distributed file system, the *information service*, and use that for communication and coordination. To be fault-tolerant and highly available, the information service must run on several machines. To find out which, we divide the grid participants into three categories:

1. The *users* who submit and monitor jobs.
2. The *resources* that offer storage or process jobs.
3. The *fabric* that keeps the grid up.

Keeping the information service up is clearly part of the duty of the fabric, whereas conceptually the users and resources will be mere clients of the information service. In reality, most resource machines (such as cluster gateways or storage servers) are probably very stable and reliable and thus prime candidates for being part of the fabric. But some resources may have a more transient connection, e.g. desktop machines, or may for other reasons simply not want to or be unable to contribute to the fabric, for instance because of restrictive firewalls. Hence, we make the distinction between the fabric and the resources.

Apart from the coordination information that is transferred through the information service, there is also a need for transporting large amounts of input and output data efficiently between the users and the resources. These are handled by the *data service*.

Furthermore, we need components for resources and components for users. In total, we split the design into the following parts:

- Information service component
- Data service component
- Resource components (e.g. cluster resource, storage resource)
- User components (for job submitting, WWW monitoring interface)

The basic idea is that a user or resource component sets up the information service component either as a client or as part of the fabric and uses it to communicate with the rest of grid, activating a data service component if it needs to transfer data, see Figure 2.1.

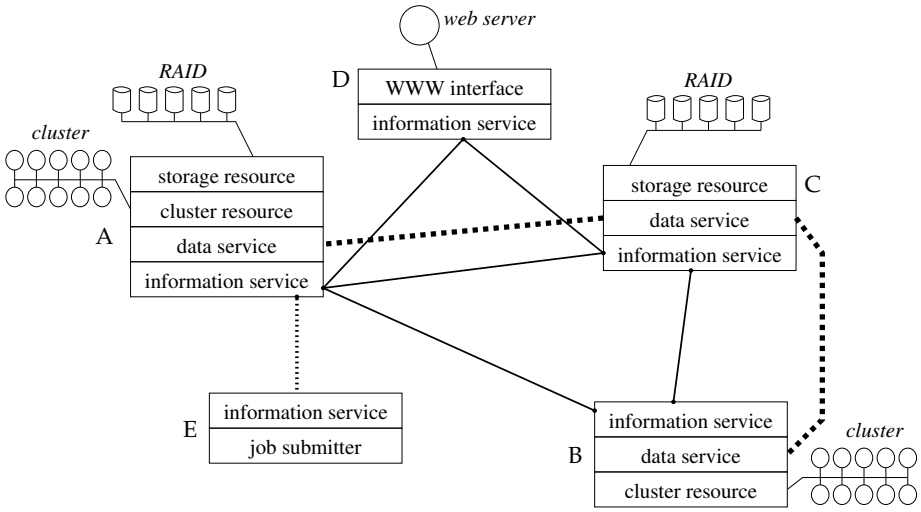


Figure 2.1: An example of deployment of our grid design with five hosts. Host A is a cluster with mass storage attached, host B is a cluster, host C is a mass storage resource, host D is a web server which is connected to the grid for easy monitoring access, and host E is a client with a job submitting interface – the information service component on host E is not part of the fabric. Communication takes place indirectly via the information service with input and output data being transferred directly to the end-points through the data service.

The information service is used to register grid resources and users and user groups, to submit and manage jobs and to coordinate the handling of the data on the grid. To illustrate the work flow, we now give an example of a job submission based on the grid shown in Figure 2.1.

A user *E* needs to have some computing intensive analysis done on a large data set. *E* writes the analysis software, prepares a job description, authenticates with the grid and submits the job. At this point, the job is in the information service and available to the other grid participants so *E* is free to disconnect from the grid.

A resource, say the cluster *B*, then picks up the job. The cluster resource component on *B* reads the job description and requests the necessary input

data from its local data service component, which locates the closest replica of the data and begins retrieving it. When the retrieval is complete, the cluster resource component starts executing the job by placing it in the batch system on the cluster, and keeps monitoring it to be able to report status and any errors that may occur to the information service.

When the execution has ended, the cluster resource components hands any output to the local data service. *E* can then connect to the grid and retrieve the data.

The following sections give an overview of each of the components, and the rest of the chapter discusses each in detail, covering what this simple example does not mention, including job failure and cancellation and how data is actually managed.

2.1.1 Information Service Component

The structured decentralised overlay network designs described in Section 1.4 support a hash table interface. On top of this, the information service must present a file system interface with several features, to some extent similar to what operating systems provide.

First of all, with a large-scale system it is important to be able to structure the information. The classical way to do this is to support directories which gives a tree structure (or just a directed acyclic graph if a file can appear in multiple directories). The information service should support directories by having the file operations in its interface be aware of them and by supplying special operations for creating and listing directory contents.

Furthermore, since the grid is intended to span multiple organisations, it is important to support access control. This implies the need for the concept of a user in the system and for user authentication. To be able to administrate the users conveniently, the information service must also support forming groups of users so that rights can be granted to a certain group. The group system should be flexible enough that it is possible for any users to create arbitrary groups. This, together with the access control, makes it possible to form ad hoc virtual organisations.

As mentioned in Section 1.3, the information service must also support some means of synchronisation. One part of this is to be able to make changes to a file without experiencing race conditions that may lead to a lost update. For instance, if two grid participants at the same time read the same file, change it and write the result in the file system, one of the participants will overwrite the changes of the other participants. To solve this problem, the information service should support file locks, so that a grid participant can lock a file temporarily to exclude others from it.

Another part of coordination is the ability to let other grid participants know that something has happened. Since the information in the information

service is divided into files, it is natural to support subscription to notification of file changes so that when the file is changed, the subscribers are notified by the information service. Without this feature, the other grid participants would have to resort to polling the file which results in either slower response times if the polling interval is long, or a higher overhead if the polling interval is short.

Finally, while most distributed hash tables consider all participants to be peers, the information service must also support a client mode of operation for those participants that are not part of the fabric.

To sum up, the information service is a structured decentralised overlay network that supports:

- A file system interface with support for directories.
- Users, groups and access control.
- File locks for mutual exclusion.
- Notification of file changes.
- Fabric mode and client mode.

2.1.2 Data Service Component

The large quantity of input and output data involved in the current and predicted future grid jobs means that it is important to handle transportation of the data on the grid efficiently. For instance, although it would be theoretically possible to store the 12-14 petabytes pr. year generated in the ATLAS experiment in the information service, doing so would add a severe performance requirement that may be difficult to fulfil given the other requirements of the information service. A separate data service is more flexible – it can focus solely on delivering and storing large amounts of data.

The data service design we describe is quite simple and based on an underlying transportation protocol and storage method, e.g. HTTP, FTP, scp [52] or SFS [32]. The interface should be minimal to make it easy to replace it with another more advanced design, if needed.

From the system requirements, it must be possible to structure the data sets and the management of them needs to be under access control. To avoid two parallel systems of users and access control mechanisms, the access control for the data service should be based on the information service. This can be done by mapping the structure of the data sets into the information service as a directory hierarchy.

Since the data service must support basic replica management to protect against unavailable hosts, the directory hierarchy also provides a place to store information about where the replicas are located. Our simple design should only support registering replicas and transferring them.

However, when a data set is replicated, it is possible to choose where to retrieve data from and the design should not just try a random site, but base the choice on measurements of the network capacity. This makes it possible to avoid scenarios where a far-away replica is chosen instead of a much faster local replica.

In summary, the data service is closely related to the information service and should support:

- A minimal interface for retrieving and storing data, based on an existing transfer protocol. Higher-level structure is achieved through the information service interface.
- Basic replica management to avoid unavailable data.
- Bandwidth measurements to optimise the data transfers.

More advanced designs are possible. For instance, one could design a self-organising data service that automatically handled replication, either designed from the ground up or based on a distributed hash table. Unfortunately, the current distributed hash table designs have not been tried and tested with large amounts of data and suffer to some degree from a very rigid structure that implies loss of control. It would be a major undertaking to address these issues in a design and test the result, and outside the time frame of this project.

2.1.3 Resource Components

Each grid resource is represented by a component that acts as an intermediary between the grid and the resource, e.g. a cluster gateway or a storage server. In general, the component needs to register itself in the information service, continually report its status, receive and run available jobs and deal with failures.

The actual execution of a job and management of the resource is of course specific for the type of resource. To be able to work for a grid like NorduGrid, a cluster resource and a storage resource component must be designed.

2.1.4 User Components

The most basic user component is a job submission component. It should support:

- Submitting jobs via the information service.
- Monitoring and management of the submitted jobs of the user.
- Handle input and output data via the data service.

A deployed grid will most likely have several other user components. For instance, NorduGrid currently has a web monitoring interface which with our

design simply would be another user component with some glue code for extracting information from the information service and formatting it as HTML upon requests from a web server.

Given the design of the other components as explained in the following sections, the job submission component should be straight-forward to design. Hence, we do not discuss it in the following.

2.2 Information Service

As mentioned, the information service is a file system in which files contain the information and directories give the structure.

Although the grid is built upon collaboration, it is important for many practical use cases that the collaboration can be controlled to protect the virtual organisations from each other. We implement this in the design by structuring the file system with multiple roots. Each grid participant and each group of participants get their own directory root, in which they have the full control.

A virtual organisation is then a separate directory root in which there is a list of associated users and resources and a job queue with scheduling information. Jobs are submitted in the virtual organisation by placing them in the job queue.

The following sections explain in detail how users and groups are realised, how the file system is mapped to a distributed hash table, the means of access control, security considerations for the distributed hash table and how notification and mutual exclusion are grafted onto the decentralised overlay network. Note that in the context of the information service, a user denotes any party with access to the service and could be a resource as well as a human job submitter.

The information system has a high-level interface. For the file system *create directory*, *remove directory*, *list directory*, *create file*, *delete file*, *write file*, *read file*, *subscript to notification*, *cancel notification subscription*, *lock* and *unlock* are supported. For administrating users the system provides *create user*, *delete user*, *create group*, *delete group*. For rights on individual files or directories *grant right*, *deny right* and *list rights* are supported.

2.2.1 Users and Groups

A cornerstone in access control is that the parties have some means of proving their identity. Cryptography enables this with an asymmetric key pair with a public and a private key. The private key can be used to produce signatures or respond to authentication messages in such a way that the authenticity can be verified with the public key.

Hence, each user of the information service, both grid users, resources and the fabric, must be equipped with a private key and the corresponding public

key must be made available to all participants, in a scalable and robust manner to fulfil the design goals.

The distribution of the public keys can be done within the information service itself, however. Each user is given a separate directory root where the public key is placed in a special file. To make it impossible to tamper with the key, we employ a trick, self-certifying pathnames, inspired from the Self-certifying File System (SFS) [32]. The idea is that the name of the directory root of a user is the hash of the user's public key. The public key can then be checked for alterations by hashing it and comparing it with its path.

With this simple idea, it is possible to interact with the information service only if one has a private key corresponding to an existing directory root with an public key file. Requests that cannot be verified with a public key in the information service should be denied.

Adding users

It is still necessary with some procedure for adding new users. They should not be allowed – and are not allowed, due to the above scheme – to add themselves. Instead, a prospective user generates a new key pair and sends the public key to someone with access to the information service. The latter can then sign the key and write a new key file on behalf of the prospective user. When the key file is in the information service, the user is effectively added and can start creating files in the directory root – all files under the root of the user belongs to that user and can only be accessed by others if the user grants the access.

There still seems to be a chicken-and-egg problem with adding the first user. The solution is to generate a key pair when the grid is started and distribute the public key of that key pair with the software. The private key of the key pair is then used to sign the public key of the first real user and thrown away afterwards.

The key signing step is important for two reasons. It enables a node that receives a key file to check that the key is indeed added by someone with access to the information service and not just somehow smuggled in. It also provides traceability – it is possible to find out who is authorising someone to enter the grid. Since the authorising user may leave the system at some point, another user may have to sign the key to avoid that it is invalidated.

The traceability yields several possibilities for trust models. By defining a global policy that only allows a well-defined group to sign keys, the system can be tightly controlled, similar to a UNIX system where by default only the root user can add users. But allowing everyone with access to the information service to sign keys is much more flexible and relatively safe. If a user A adds a malicious user B, B cannot access any of the other data in the information service unless specifically given access to it. B can only add data or new users – and once he is revealed, the data can be removed.

One thing that is lacking from the above description is how to determine who may remove a user and how a user is removed. We cannot allow the user himself to change who has the control over him, and at the same time it is difficult to support changes in this at all while still keeping the key file self-contained. So we opt for a write-once solution where the list of authorised removers is appended to the key file and covered by the signing by the user who authorises the new user. The hash of the key file should include this list to ensure that it is invariant. When a user is deleted, his public key is marked as being invalid. If the key was just removed from the system, he could insert it again and regain his former status since it appears to be valid.

Groups

We design groups in a manner similar to the users by giving each group a public key file. However, this administrative key is only used to define group membership since each group should have two types of users associated, ordinary group members and group administrators, where only the latter have access to the private administrative key of the group.

Then a list of special group keys is placed in the root directory of the group with the last of the keys being the active one. The active key is used for reading and writing files and in general authenticating actions done by someone in the group. The reason there needs to be more than one of these keys is that a new key must be generated every time someone is removed from the group – hence only the latest key is active. The old keys are kept to not have to rewrite all data when a user is removed.

The private key of the active key must somehow be distributed to the members of the groups. This is possible by making a copy of the private key for each member and encrypting that copy with the public key of the member – the technique is further discussed under the topic of access control in Section 2.2.3.

The administrative key is separate from the group keys since it is not possible to change this key without creating a new group because of file of the key is using a self-certifying pathname. Thus it is possible to add new administrators to a group simply by giving them the private administrative key, but it is not possible to remove administrators again because they cannot be forced to forget the private key. For the same reason, the administrative key should be handled with care.

Note that users may still be able to write new data with an old group key as it may not be possible to determine whether a file is just an old replica or a completely new file. But it is not possible to overwrite data written with a later key, so if necessary this can be combated by rewriting all information written by a group.

To sum up, the user and group design just described supports that:

- Everyone can add a new user or group.
- New users and groups get their own directory root where they have the full control.
- Users and groups can be deleted only by a predefined list of users that the authorising user has agreed to.
- Groups have administrators that fully control the group, and ordinary members that can be added and removed at will.
- The members of a group can be chosen arbitrarily, so that it is possible to form groups of groups.

2.2.2 Files and Directories

To store the file system in the distributed overlay network, the files and directories must be mapped to data blocks referenced by keys, as outlined in Section 1.4. Our mapping is simple: each file is mapped to a single block with the key being the hash of the file path (which is plainly unique). The key and some other metadata are stored along with the contents of the file in the block. A directory is mapped as an ordinary file that contains a list of the files and subdirectories in the directory.

In general, files could be so large that storing them in a single block would hurt the load balancing of the overlay network. To alleviate this, it would be possible to split a file into several blocks. But this requires some extra work to ensure that file changes are atomic to avoid that files end up in an inconsistent state if a file writer crashes half-way through a write operation. Since we expect only small amounts of data to be stored in the information service, we decide against the added complexity of splitting files.

Large directories could possibly be a performance issue. An informal test on a desktop Linux machine listing 140175 files, stored in a single file would consume 1912 kilobytes, with an average file name length of ∼13.6 characters. To reduce the size large directory files could be compressed. The mentioned file list could be reduced to 476 kilobytes of `gzip`-compressed data. Another possibility would be to use a more advanced indexing structure, as is used in databases, instead of a flat list.

Block Structure

A file block consists of a header, a body and a signature. The header contains the path of the file, a version number to detect and prevent old versions of a block to overwrite a new version, an access control list to handle access control on the block (further described in Section 2.2.3) and a number of copies of a read key that enables the users who are authorised to read the contents of the file to decrypt the block body. There is one copy of the key for each authorised

reader, encrypted with the public key of that reader. The body is simply the contents of the file, encrypted with the read key.

Finally, a file block is signed by the last user who changed it. This along with the access control list makes it possible for a node receiving the block to determine its authenticity and legality. If the block is not authentic or violates the access control restrictions, the receiving node should discard it.

Public key files are a bit different. As mentioned, they consist of a public key and a list of users and groups that may delete the key, plus a signature by the party who authorised the key file, i.e. someone else than the key owner. Furthermore, the position of key files in the file system is fixed and determined by their contents. This simple structure makes them self-contained and easy to check.

File Operations

The overlay network consists of client and fabric nodes, as discussed in Section 2.1. The clients do not participate in the data storing and routing of search requests, but instead just contact the fabric nodes. All clients must, however, support the file operations mentioned in the beginning of this section, since the network communication is on the block level.

A file can be read by retrieving the block of the file, decrypting the read key and then decrypting the body using the read key. To write a file, a node retrieves the block of the file, reads it, updates the contents, filling in some of the meta data, and finally stores the block in the overlay network again. Creating a file or directory involves creating the block and storing it in the overlay network, after which an entry with the name is added to the parent directory. Deleting a file or directory is done by deleting the entry in the parent directory and then deleting the blocks from the network.

The operations that change files may result in race conditions if multiple users are updating the same file. In this case, it is necessary to obtain a mutex on the changed file.

2.2.3 Access Control

Access control in the information system must follow a radically different approach than with a centralised system. There is no central place to ask for authorisation, which means that each node in the network has to be able to verify incoming blocks by itself. We operate with two types of restrictions: only read and both read and write. The permission to add and delete files follows from the write permissions of the parent directory.

Enforcing Read and Write Restrictions

Read restrictions are enforced by encrypting the contents of each block with a randomly generated key that is distributed with the block, encrypted so that only the authorised readers can read the key, as explained in the previous section. Since blocks are available everywhere in the network, the contents *must* be encrypted to be kept secret.

Distributing the read keys with the blocks makes the blocks quite self-contained and only adds little overhead since symmetric encryption keys are quite small (128-256 bits or only 16-32 bytes). An alternative is to use static read keys that are common to all blocks and store them in the information service. But the number of possible keys grows exponentially in the number of users and groups since any subset of them may need to be given read access, and static long-term keys are more susceptible to attacks.

Write restrictions are enforced by verifying incoming blocks. When a node receives an incoming block, it has two ways to check it depending on what kind of block it is. A key block is verified by checking that the hash of the contents matches the root part of the file path, and checking that the signature is correct.

The signature check may require that the key used for signing is fetched and checked, and so forth, but the procedure should stop at some point with the built-in root key that is distributed with the software. Otherwise there is a signing loop, and the key should be rejected. Since the path of signed keys to the root key may be quite long, the nodes should cache verified keys. This cache must expire at some point, though, otherwise it would be impossible to delete users.

An ordinary data block is verified by checking that the block signature is correct – to verify the authenticity – and checking that the signing user actually is authorised to write the block by scanning the access control list included with the block.

Structure of Access Control Lists

Distributing the access control list for a block with the block is convenient and helps making the block self-contained. But with a naive implementation, it would be possible for a malicious user to rewrite a block, insert himself in the access control list and sign it with his own key. It must be possible for the receiving node to detect that this has happened.

A solution for this problem is to have the users granting rights sign the grants. At first, only the user or group that owns the directory root under which the block is placed can grant rights by adding them to the access control list and signing it. The rights can either be the right to read the block or the right to both read and write the block. The users and groups that have been granted write permissions can afterwards also add more entries, signing the list as they go along. The idea is illustrated in Figure 2.2.

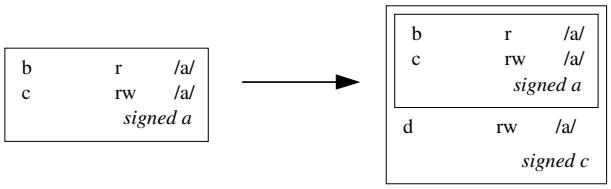


Figure 2.2: An example of granting permissions. The user name is shown to the left, in the middle is the permission type (read or read-write) and to the right is the path. User a has initially granted b read-only and c read-write access to the directory root of a. User c then grants d read-write access by adding the required permission and signing the whole list.

When a new block is created, the access control list is initially just the access control list of the parent directory. Each entry in the list should include the path of the file or directory that the grant is covering. This ensures that it is not possible to copy part of the list and reuse it at another place in the file system, and also makes it possible to let a permission extend recursively to a directory with all its subdirectories. If a new block is a root directory, the list will initially be empty – only the owner of the directory root has access to it.

It is necessary to check the access control list before it can be trusted. Each signature in the list should be verified, and the file paths in each grant should be checked to ensure that the access restrictions are enforced.

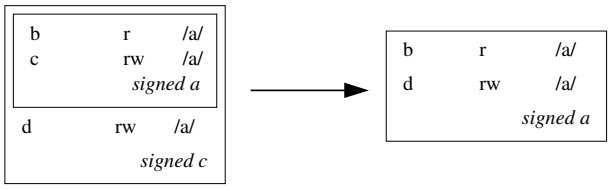


Figure 2.3: An example of removing permissions. User a removes c's read-write permission by removing c from the list and re-signing the remaining grants after c.

Removing a permission is a bit complicated because it might break the signing chain. For instance, if a has authorised c who in turn has authorised d, then if a removes c, the authorisation of d will no longer be valid. The solution is to let a re-sign any permissions that was granted after c, as illustrated in Figure 2.3. Note that when a user is removed from the access control list, a new read key for the file must be generated and the contents re-encrypted to prevent the user from being able to read it.

With this design, it is not possible for a user with read-write access to remove permissions that have been granted before that user was granted the read-write access. In the previous example, d cannot remove c since that would require unfolding the permission list to a point where d does not have the read-write permission anymore.

Another problem is that the design is susceptible to replay attacks. If a user

at some point is granted read-write access to a file, it is always possible for that user to save the access control list and later rewrite the block with that access control if he is removed from the list. This attack does require that the user saves the access control list, though. One way to counter the attack would be to include a time limit in the granted permissions; this of course requires one to grant the permissions again periodically.

2.2.4 Preventing Intruders in the Distributed Hash Table

Although the described access control means should be able to protect the blocks themselves from being tampered with or read by unauthorised parties, it is still easy to break down the information service by attacking the distributed hash table.

One such an attack that can cause data loss is if a malicious host starts many fabric nodes on the same host and let them connect to the information service. After a while, some of other fabric nodes will send block replicas to these malicious nodes, and eventually the malicious host may end up with all the replicas of some blocks. Then it can cause data loss simply by discarding them.

To combat this attack, the fabric nodes should not insert other nodes into their routing tables until they have been able to authenticate them. For this reason, each fabric node must have a public key in the information service, just like the users. The authentication procedure need not add to the latency of the operations and can be done in the background because the fabric nodes are free to respond to requests without authentication; the blocks are sufficiently protected in themselves.

A related problem is denial-of-service attacks. These are difficult to deal with, but it may be possible to counter most attempts by enforcing sensible limits to what a fabric node will do for a client simultaneously, including limiting the number of request served per second. Another possibility is to force requesters to do some work by incorporating a hash cash scheme [6].

2.2.5 Notification of File Changes

When a file is changed, there are initially only two parties that are aware of the changes. The entity who changed the file and the r nodes in the overlay network which the block of the file is distributed to. Since the changes are not persistent in the information service before the block reaches at least one of these r nodes, we can conveniently let them keep track of interested parties and see to that they are notified.

Hence, an interested participant should send a subscription message to each of the r nodes storing the block and send a cancellation message when the subscription should be stopped. This means that when an incoming block has been verified and accepted, it is necessary to check whether anyone is sub-

scribing to changes on that block, and if so send a notification message to them with the version number of the new block.

In order to be able to tolerate failures, the list of subscribers should be considered soft state so that the subscribers have to renew their subscription periodically to avoid losing it. This means that a failed subscribed node will be removed from the subscription list after some time, and that the subscribers will detect when a notifying node has failed.

Note that in the case without failures, r nodes will send notification messages, which means that the subscribers will receive $r - 1$ redundant messages, see Figure 2.4. The subscribers must be able to cope with this, but since the notification messages contain the version number of the block, they can just discard redundant and old messages.

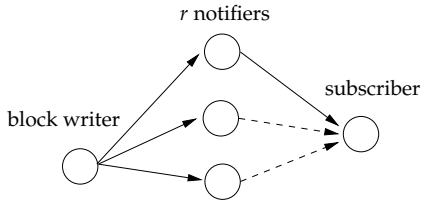


Figure 2.4: A block is written and the r nodes responsible for it notifies the subscriber. The stippled arrows are redundant messages.

With a slightly more complicated protocol, we can also optimise most of the redundant messages away, at least as long as there are no failures. The basic idea is that the nodes that are farthest away from the block give the nodes that are closest to the block some time to send notification messages. The node that is closest to the block should send notifications immediately, the second-closest node should wait one predefined period of time, the third-closest node should wait two periods of time, etc.

Then when a node is about to begin sending notification messages, it should first inform the other $r - 1$ nodes that it is sending notifications, as illustrated in Figure 2.5. This makes them wait some extra time. It then sends the notification messages to the subscribers, and when it is done and has received acknowledgements from the subscribers, it can inform the $r - 1$ other nodes that it is done. The other nodes then do not need to send notifications themselves.

Assuming there are no failures, this results in $2(r - 1) + s$ messages instead of rs messages, where s is the number of subscribers. For $s = 2$ this implies the same number of messages, for $s > 2$ the result is fewer messages. However, the messages that are sent to inform the other $r - 1$ nodes before and after notifying the subscribers need to include the addresses of the subscribers to guard against incomplete knowledge. Since the other nodes will continue if the delay is too long, the algorithm can still sustain $r - 1$ node failures without malfunction.

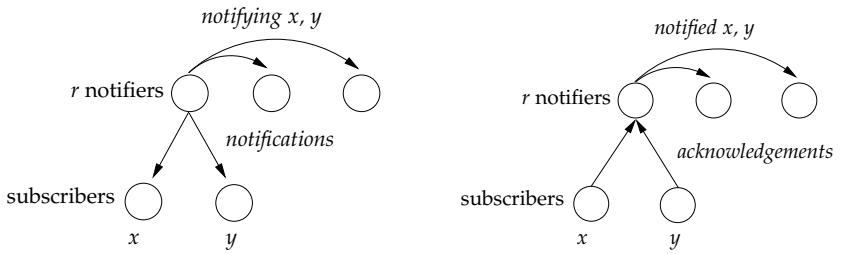


Figure 2.5: A more elaborate notification algorithm. To the left the closest node to a block informs the two other nodes that is notifying and then sends notification messages. To the right it receives acknowledgements and informs the other nodes that the notification is done.

An important use case for notifications that is not discussed in the above is when a participant retrieves a file, reads it and then wish to subscribe to future changes. Before the subscription message arrives at the notifying nodes, another node may already have changed the file. This change would not be noticed by the new subscriber until the next change arrives after the subscription has succeeded.

The participant can alleviate this problem by retrieving the file again after having subscribed to changes. But a more elegant solution is to have the participant send the last version number seen with the subscription message. If the version number is older than the version currently stored at the notifying node, that node can then immediately send back a notification.

2.2.6 Distributed Mutual Exclusion

A distributed algorithm for mutual exclusion requires reaching consensus among the participants over who has the right to a resource. In our case, a resource is simply a name in the form of a file path which may coincide with the name of a real file or directory, but not necessarily. Using this path as the key for mutual exclusion on a resource gives r nodes that can be made responsible for that mutex.

Hence, we construct an algorithm that by contacting these r nodes can ensure that an agreement over who has obtained the mutex will eventually be reached. One possibility is to ask the nodes sequentially for the mutex – but then the latency for obtaining a mutex will be the sum of the latencies for the r nodes. Instead, we first optimistically try to obtain the mutex by sending requests in parallel, which reduces the latency to the response time of the slowest node in case there are no conflicts.

Thus, in the first phase of our algorithm the requesting node asks all r nodes in parallel for the mutex. If all r nodes grant the mutex, it can go on since the r nodes will henceforth claim that it has the mutex. Otherwise another node is holding the mutex, or at least trying to obtain it, and the requesting node cancels the requests for the mutex it has already sent and enters the second

phase of the algorithm.

In the second phase, the requesting node orders the r nodes by their distance to the key and ask each of them in turn for the mutex, starting from the node closest to the key. If one of the nodes does not grant the mutex, the requesting node cancels its requests and waits a random period of time before restarting the second phase. An example is shown in Figure 2.6. The maximum value of the random period of time should be increased for each denial of the mutex to avoid overloading the granting nodes if the mutex is very popular.

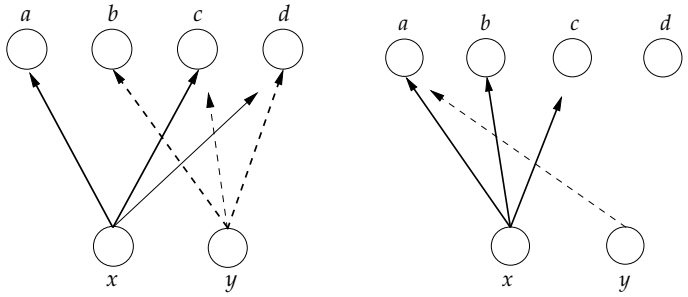


Figure 2.6: To the left, two nodes have simultanously tried to obtain a mutex using the first phase of the algorithm. To the right, they proceed with the second phase. Since x was granted the mutex by the node that is closest to the mutex key, the request of y will fail whereas x continues undisturbed.

Since the requests are sent in a predefined order in the second phase, simultaneously requesting nodes cannot end up in a deadlock. If two nodes were requesting the mutex at the same time, one of them would discover that the other has already been granted the mutex by one of the nodes and hence stop. This works even in the case where the two requesting nodes do not have exactly the same view of which nodes to contact, since they still agree on the order.

The algorithm thus ensures mutual exclusion in case of failures among the mutex granters as long as just one of the nodes that have granted the mutex is still up and among the r closest to the key, because another node can only claim to have the mutex when it has received r grants, and the remaining node will deny it.

To avoid that a failed requester holds a mutex forever, the mutexes must have a limited lifetime. If a requester fails, then the grants of that requester eventually expire so that the algorithm can continue. This works regardless of whether the failed requester was in the midst of obtaining the mutex (received less than r grants) or had already obtained it (received r grants).

Like other mutual exclusion algorithms, the algorithm does not in general avoid deadlocks since a wait-for cycle with two mutexes may appear. The limited lifetime of the mutexes may, however, be able to clear a deadlock with some luck. Another problem with the above algorithm is that it makes it possible for a requesting node to starve indefinitely if the mutex is very popular.

One benefit of the algorithm is that it is quite simple.

2.3 Data Service

The simple data service we describe uses the information service for structuring the data. Any user or group can create a data directory in their root and use the information service to setup the structure, browse the available data and grant or deny rights. It is only when the actual data needs to be transferred or removed that the data service must to be invoked.

Hence it only needs a simple user interface. For transferring data, it should support *store* that given an information service path, a data chunk and a list of replication hosts stores the data at those hosts and *retrieve* that given a path retrieves a data chunk. Since the *store* operation adds replicas, it is also necessary with a *delete* operation to remove replicas.

There must also be a network interface for communication between two data service hosts but for simplicity we assume this to be handled by another protocol, such as HTTP, FTP, scp or SFS.

The following sections discuss the interaction with the information service, where replicas may be located and how to transfer data between the data service hosts through the network interface.

2.3.1 Structure in Information Service

A convenient way to represent the data in the information service is to have a direct correspondence between a file in a data directory and an actual data chunk. Then the users can manipulate the directory structure and the permissions of the files, whereas the data service manipulates the actual contents of the files. The contents of each file is some coordination information for the data chunk it corresponds to. This consists of a list of replica locations and a check sum of the data to be able to detect corrupted chunks; an example is shown in Table 2.1.

chunk path	check-sum	locations
`/a/data/lhc/chunk1`	`a314b3...`	x, y, z
`/a/data/lhc/chunk2`	`ef2f34...`	x
`/a/data/uppaal/mdl`	`ef2f34...`	y

Table 2.1: An example of a data directory under the root of a. The first chunk is replicated three times (at the hosts x, y and z), the two others have only one replica.

The data service must authenticate all request to ensure that it does not send or receive data from a party who is not authorised. It can do that by retrieving the public key of that party from the information service, for authentication,

and checking the permissions of the file corresponding to the data chunk, for authorisation.

The data chunks are not encrypted, however, so if a user wants to ensure that the data is kept secret from the storage resources storing them, it is necessary to encrypt them beforehand.

2.3.2 Replica Locations

Data sets can be located at basically two kinds of places: storage resources, which act as long-term storage and have very large data capacities, and user and resource hosts with smaller capacities where data is kept only temporarily.

The storage resources need to register themselves in the virtual organisations they are providing storage to, so that it is possible for other participants to find out where to place input and output data. The registered information should include the available free space which must be updated whenever data is stored or deleted. Otherwise, the storage resources are simple file servers.

For the temporary data locations, the situation is more complex. The data may be stored at a grid user who needs to submit it with a job, or the data may reside at a grid resource, either because it is the input for a job or because it is the output from a job.

The users have three possible ways of submitting the data. A stable user can setup a data service on his host and write in the information service that the data is available from that host. This allows the resource that eventually receives the job to retrieve the data directly from the user. Another possibility is that the user picks a specific resource, transfers the data to that resource and specifies in the job description that this particular resource is preferred.

Both of these ideas avoids transferring the data more than once, but are not fault-tolerant since the user or the resource may fail. The safe possibility is to transfer the data to at least one storage resource and let the resource running the job retrieve the data from there.

Since the computational resources need to have enough space available for processing jobs, it is likely that there will be excess free space, at least temporarily. This extra space can be used in several ways.

First, by letting retrieved input data reside at the resource even after it has been used, it can be used as a cache for that resource in case other jobs need the same data. This is most effective if the job scheduler is made aware of it and capable of taking it into account.

But the cache can also act as an extra repository for the input data that other computational resources in the vicinity can exploit. This helps avoid overloading storage resources with popular input data. The cached replicas must be registered in the information service by the data service running on the computational resource. A cache-clearing scheme such as evicting the least recently used data must be employed when the available space is low.

The extra space can also be used to store the output data over longer periods of time to avoid transferring them to a storage resource. If the computational resource fails, the data will, however, be unavailable, and at some point the data must be forwarded to a storage resource anyway to make room for other data if they are not retrieved and deleted by the user. But in cases where the computational resources are reasonably stable, this way of handling output data may be preferable.

2.3.3 Data Transfers

Except for the coordination via the information service, the communication in the data service is done in a point-to-point manner where one data service host directly contacts the destination of a *store* operation or the source of a *retrieve* operation. The underlying transfer method, such as HTTP, FTP, scp or SFS, is used to connect to the specified host, authenticate and transfer the data.

When a data set is replicated, there are many possible strategies for optimising the time it takes to retrieve it. One possibility is to test the bandwidth to all replicating hosts just before the transfer is initiated and then choose the fastest host. Unfortunately, small test packages may not be representative of the speed at which large amounts of data can be transferred.

Another strategy is to retrieve the data set from all hosts in parallel. If the data is divided into more chunks than there are hosts, the hosts that turn out to have the fastest connection can be assigned most of the chunks. A simple policy that achieves this is to request a chunk from each of the hosts at first and then request another chunk from a given host when the transfer of the previous chunk from that host is completed. Of course a more sophisticated algorithm is needed to prevent the transfer to be bounded by the latency.

2.4 Grid Resources

In this section we explain how the resources that the grid participants choose to make available can be connected to the grid with the design outlined in the previous sections.

We first describe some general issues: how to register the resources in the information service so that it is possible to manage them, how to submit and schedule jobs, how to cope with failures, how to control the submitted jobs and how to log the major events so that the execution trail is recorded for later examination. The design emphasises flexibility so it is possible to use it with many possible types of resources. Afterwards, we outline the design of a batch cluster and storage resource component.

2.4.1 Resource Registration

The resources need to register themselves on the grid so that it is possible to get an overview of the available resources. For instance, the storage resources need to announce their available free space so that it is possible to determine where to send data. Similarly, a scheduler for the computational resources needs to know what resources are available, unless the scheduling algorithm is very simple.

Each virtual organisation creates a resource directory in which the resources write their registration information. One way to manage the information is to create a subdirectory for each category of resource (e.g. storage, clusters, massively parallel supercomputers), and assign a directory in this subdirectory to a given resource. The resource must be given write permission to the assigned directory and can then fill in the information on its own.

Exactly what information is registered depends on the type of resource; in general there is a fairly static part that describes the capabilities of the resource and a more dynamic part that describes the current status of the resource.

For a storage resource, in addition to the name and contact information intended for human administration the static part would be a network address for establishing a connection to the resource when retrieving or sending data and the maximum amount of space available. The dynamic information would be the amount of free disk space and a time stamp that is updated periodically so that other grid participants can discover if the resource is down without having to contact it.

For a computational resource such as a cluster the static information would be the available processors, memory and disk space and the system architecture in addition to the administration information, and the dynamic information would be the current and possibly scheduled load.

2.4.2 Job Scheduling

Some resources are active resources in the sense that it should be possible to submit jobs that they will then start executing on their own at some point in the future, while other resources are passive, e.g. the storage resources which simply accept and deliver data when requested. For the active resources, it is necessary with a scheduling scheme for deciding where and when jobs are executed.

As mentioned in Section 2.2, our basic idea is that job submitters place their job in the information service. As a consequence, our design is flexible enough to accommodate several scheduling models, although the particular model chosen affects where submitted jobs should be placed in the information service.

The model that NorduGrid is following makes the users responsible for scheduling their jobs at resources – this corresponds to each resource having its

own job queue where users place their jobs. However, the problems associated with this scheduling model, as discussed in Chapter 1, prompts us to consider alternative models. If each virtual organisation has a queue of incoming jobs for each resource category, i.e. common to the resources belonging to that category, then the grid itself can take care of the scheduling.

It is in principle possible to deploy any scheduling algorithm with the information about resources that is available through their registration and the job information available in the incoming queue. It is also likely that different algorithms will be preferable in different cases. The choice of algorithm affects which grid participants do the scheduling.

If the grid is employing desktop machines with unpredictable uptimes but an abundance of resources, then it may make sense to choose a simple algorithm such as first-in, first-out and let the resources themselves select jobs from the job queue when they are idle. Selecting jobs in this manner rather than scheduling them only requires little effort from the resources.

It is also simple to implement. A resource that thinks it will be idle shortly first reads the job queue to see whether there is a suitable job. If not it subscribes to notification of changes to the queue and remains idle. Otherwise it locks the job queue, marks the job as being taken if another resource did not get to it first, timestamps the mark and removes the lock again. Resource failures can be handled by ignoring old marks when considering whether a job is free. There is a risk that the job queue in this simple implementation becomes a bottleneck, though, in which case it must somehow be splitted.

For a grid like NorduGrid where the number of resources is limited, a more advanced scheduling algorithm may be able to give higher throughput and better response times than simple job selection. There are plenty of things that can be taken into account – the architecture of the clusters, e.g. number of processors, processor performance for various tasks, memory bandwidth, node interconnections; the load and failure rates of the clusters; input and output data placement and transfer cost; and so forth.

Because of the information service it should be possible for a scheduler to collect the necessary information and the most difficult task is perhaps to come up with an algorithm that actually makes use of all this information in a scalable manner.

In any case, the most flexible approach is to have the scheduler be a grid fabric component separate from the resource components. Then for each virtual organisation with a job queue, a few selected sites should run an instance of the scheduler that subscribes to changes in the job queue. It is necessary with multiple schedulers to avoid paralysing the grid when one of them fails. The schedulers can coordinate through the information service and use it to ensure that only one scheduler is active at a time.

Apart from flexibility, separating the scheduling task from the grid users and resources also makes it easier to enforce optimisation of the overall perfor-

mance. When individual users and resources are trusted to do the scheduling, they have the possibility of choosing schedules that benefit themselves instead of the schedules that are best for the virtual organisation as a whole.

2.4.3 Job and Resource Failures

Both resources and jobs can fail, and it is important that the grid itself tries to cope with these failures on its own as much as possible. The risk of failures increases in a large distributed system such as a grid, and we want to avoid that this increased risk becomes a burden for the users. One of the most time-consuming problems with NorduGrid in practice was found to be that failures had to be resolved manually [25].

When a job has been picked up by a resources, there are three types of failures that can occur:

1. The resource that is running the job can crash or lose its connection to the rest of the grid, which means that it is unable finish the job and submit the result. Unless the resource can recover from the situation within a relatively short period of time, the job is lost in some sense and should be restarted elsewhere by the grid.

2. The job can fail because of temporary erroneous conditions in its environment, e.g. an I/O error. Many grid applications requires much of their environment in terms of CPU and disk resources and may push the host systems to their limits where failures are more likely to occur. When this kind of failure occurs, simply rerunning the job may be enough to be able to finish it.

3. The job can fail because it contains an error itself. The only way to deal with this kind of error is to finish the job with an error report to the job submitter.

The resource failures can be detected by letting a schedule monitoring component see to, in cooperation with the scheduler, that everything is progressing according to the schedule. The progress can be followed by reading the job status and checking time stamps – a failed resource will not have updated its status for some time. If a resource is determined to have failed, its jobs are then rescheduled elsewhere.

Job failures are more local in nature and can be handled within the resource itself. However, a major problem is how to distinguish between intermittent job failures and deterministic failures that will occur each time the job is run. The difference may not even be obvious to the job submitter himself without a proper examination of the job source code.

One way to resolve this problem is to supply the maximum number of retries upon failure when the job is submitted. If the resources are scarce or if the

job submitter is not confident that the job is correct, then retries can be disallowed. But if the job submitter is confident that the job should work, the job can be retried a couple of times before giving up.

A related problem is that intermittent job failures may be due to resource contention. Although this is difficult to deal with in general, it might be beneficial to wait some time before restarting a failed job.

2.4.4 Controlling Submitted Jobs

When a job has been submitted to the grid, it will at some point be processed and the result made available to the user. But meanwhile the user might discover that the job is faulty or otherwise change his mind about it. To avoid wasting resources and provide a responsive interface to the user, the grid should support changing the job parameters and cancelling the job, as was also required in Section 1.5.

A general solution to this problem is to simply change the parameters in the job description. Any entity that is depending on the information should be subscribed to be notified of changes to the information and can hence act accordingly. The scheduling monitoring component would possibly need to react to such changes, as would a resource that is in the midst processing the job.

An alternative solution in the situation where a job is being processed by a resource is to have a protocol for contacting the resource directly. Such a protocol is likely to be needed anyway if one wants interactive job sessions. But a direct protocol is not enough in itself since the information must be entered into the information service anyway if the resource should fail. Furthermore, there would be an added complexity from such as protocol which for instance also would have to address access control and security. And even though an interactive protocol is devised – our project is not addressing it – it seems unlikely that all resources would support it.

2.4.5 Logging

With the data service in place, it is easy to make the logs of the job execution trails available. Each job has an associated log file which is placed in the data service. The grid participants that process a given job should append data to the log at appropriate places in the processing, e.g. when a job is scheduled to run and when it has finished executing. The details depends on particular participants that handle the job.

The logging facility also provides a foundation for billing the grid users for their usage. With precise and authenticated logs about what happened to jobs after they were submitted, a resource owner can collect the data on a monthly basis and send out bills. Of course, to have a real grid economy, it would be necessary with a more elaborate accounting system – for example, in general

a resource owner would probably prefer that the user proved that he has the money to pay for processing a job before the job was started.

2.4.6 Batch Cluster Resource

Having discussed some general issues, we will now outline the design of a batch cluster resource. It consists of an information service component for communicating with the grid, a data service component for retrieving input data and saving output data, a batch cluster interface component for interacting with the local batch queueing system and a manager component that represents the cluster on the grid and sees to that grid jobs are processed, see Figure 2.7.

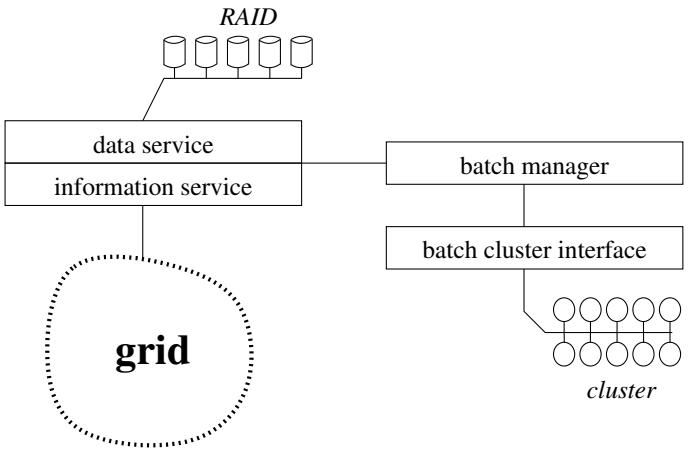


Figure 2.7: A conceptual view of a batch cluster resource.

The information and data service components have been discussed in Section 2.2 and 2.3 so the following sections discuss the batch manager and the batch cluster interface components.

Batch Manager Component

The responsibility of the manager is to keep the status of the cluster known through the information service and run the appropriate jobs on the cluster it represents. To keep the status updated it polls the batch cluster interface at certain intervals and updates the information service entry for the batch cluster resource.

To run jobs on the cluster, the manager interacts with the information service to find appropriate jobs according to the scheduling model, i.e. either by finding jobs itself or by accepting jobs from a scheduling entity. When a job has been selected, the manager retrieves the needed input data through the data service. It then starts the job on the cluster through the batch cluster interface.

Two possible scenarios follow, either the job has completed successfully or the job has failed. In case of a successful completion the output data is written with the data service and the job status is updated in the information service. In case of a job failure the job is optionally restarted depending on the number of retries specified in the job specification. If the job has no more retries left, the job output data is written to the data service and the failure is signalled by updating the status in the information service.

Batch Cluster Interface

The batch cluster interface is an abstraction over the details of different batch cluster queueing systems such as OpenPBS [36] (and relatives like PBS Pro [38], TORQUE [48]) and Condor [47]. The architecture of the component should be flexible enough to allow different back-ends for the different queueing systems. The following basic operations are needed from the interface:

- *Submit job.* This consists of putting the job into the queue, and making sure that data and program files are available while avoiding that security is compromised. After the job has terminated, the execution environment must be cleaned up and the output prepared to the job submitter.
- *Remove job.* The job must be removed and the execution environment cleaned up.
- *Status of jobs.* For instance, whether they have started, are still running, etc.

2.4.7 Storage Resource

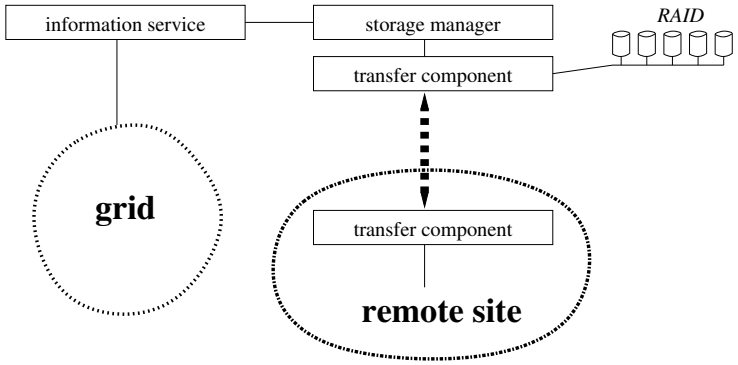


Figure 2.8: A storage resource with an ongoing data transfer.

A storage resource consists of three components. An information service component for authentication and authorisation before transfers. A transfer component, e.g. based on HTTP, FTP, scp or SFS, for the actual data transfers.

And a manager component which glues the two other components together, handles configuration such as the level of participation in the grid and where to place the data, and publishes the resource status in the information service.

Figure 2.8 shows a conceptual view of a data transfer between a local storage resource and a remote data service.

Chapter 3

Empirical Evaluation of Information Service

Since the information service is the foundation which the rest of the design is built upon, it is important to evaluate whether our proposed information service design is feasible. Hence, we have implemented the access control, mutual exclusion (file locking) and notification features on top of a distributed hash table implementation and arranged a number of experiments. This chapter describes the test implementation and setup and our experiments with the file system with access control, the mutual exclusion algorithm and the notification algorithm.

3.1 Test Implementation

The implementation is an enhancement and adaptation of Heurika [24], a prototype of a distributed file system for local-area networks implemented with a distributed hash table design that is a slight adaptation of Kademlia [31]. Heurika uses UDP for communication and lacks support for access control, mutual exclusion and notification.

Heurika is a multi-threaded design implemented in C++. It utilises the Boost [7] and GNU Common C++ [10] libraries that provide access to portable threads, the network stack, function pointers and smart pointers. The design is based on the concepts of tasks, task queues, workers and callbacks. Tasks compromise the basic file system operations, e.g. requesting a mutex, getting and putting a block, making look ups, etc. The execution of a task can result in the creation of new tasks, e.g. requesting a mutex will result in a new lookup task to find the closest nodes to the mutex. Tasks are executed by worker threads that repeatedly request tasks to execute from a task queue. Callback functions are used to signal the completion of tasks.

The adaptations for this project includes implementing the lacking features

and changing the communication protocol to TCP. The move to TCP was motivated by the need for a grid to operate over the Internet as opposed to a local-area network, and to support arbitrary message sizes since the UDP implementation was limited in practise to message sizes of about 40 KB.

The implementation of the access control features on top of the Kademlia blocks and the mutual exclusion and notification protocols was relatively straight-forward given the existing Heurika design. OpenSSL [37] was used for the cryptographic operations. Note that the authentication procedure mentioned in Section 2.2.4 was not implemented for the tests.

The TCP design maintains a pool of open connections so that there are at most one connection open between two nodes. The connection is closed after 30 seconds of inactivity. Each connection has an associated receiver thread that receives the incoming messages and put them in a work queue for processing by a worker thread. The reason we implemented this connection caching scheme is that many of the communication patterns involve multiple messages back and forth, and there is a considerable latency overhead in the connection-establishment handshake that TCP performs.

The number of lines of reused source code from the Heurika implementation at time of import was 3289. Our extensions and adaptation have added 5357 lines of code for a total of 8646 lines.

A last point to note about the implementation is that since Heurika was intended to operate on local-area networks, its implementation of Kademlia is not optimised to work on the less homogeneous and stable Internet. For instance, it does not try to avoid communication with high-latency nodes. Unfortunately, we did not have time to amend this problem. So the time-dependent results from our prototype may be considerably worse than what is possible to achieve. For example, the simulations in SkipNet [23] (Figure 12) show that on average the latencies for the latency-optimised implementations is less than half the latencies of the unoptimised implementations.

3.2 Test Setup

The tests were performed on 7 dual-processor Intel Pentium III 733 MHz computers with 2 GB memory running the Sarge release of Debian GNU/Linux, connected via 100 Mb Ethernet. Each computer hosted a number of nodes in the system, e.g. 142 each for a total of 994 nodes.

In order to simulate Internet conditions, delay on the network connections should be introduced. The first attempt at this was to use the Dummynet [41] feature of FreeBSD that can modify a flow of IP packets, e.g. to introduce delay, limit bandwidth or simulate packet loss. The solution had the kernel of the Linux machines change the destination addresses to the FreeBSD router which was supposed to reset the destinations, pass the packets through Dummynet and send them back to the Linux machines. Unfortunately, the FreeBSD router

could not handle the load so this solution was abandoned.

Instead we modified our implementation so that the receiver threads wait just after having received a message and before inserting it in the work queue for further processing. The delay used is based on a simple model that corresponds to each node being connected to a large switch with a high-latency link. Hence, the reception delay is calculated as the latency between the sender and the switch plus the latency between the switch and the receiver.

The latencies of the links to this hypothetical switch were randomly chosen in the range $[7; 55]$ ms for each node before the tests began. This corresponds to the round-trip times being twice the common round-trip times in NorduGrid, as measured from the cluster at Aalborg University.

To avoid having to wait for the overlay network in the information service to stabilise at the beginning of each test, the addresses of all other nodes were inserted into the routing tables of each node before the tests began. This is not an unrealistic premise, as most nodes in a real setting would know each other for an established overlay network of the size the tests were conducted with. And the nodes still had to sent lookup messages before most operations as specified by the Kademlia protocol. Furthermore, the periodic block republishing and routing table maintenance was disabled to avoid having them interfere with the results.

A last point that must be kept in mind when evaluating the tests is that each test machine was loaded with many information service nodes (142 nodes per machine for most tests). During peek periods of activity, this may have had an effect on the results.

3.3 File System Tests

The file system tests explores some aspects of the performance of the two block types described in Section 2.2.2. In the first test the overhead of authenticating key blocks are measured. The purpose of the second test is to measure the overhead of authenticating data blocks, and in the third test concludes by measuring the time it takes to do a complete read and write of a data block.

The test setup in all three tests consisted of a network with 994 nodes where one node was writing blocks (and reading them again in the third test) while the other nodes were passive storage nodes. The replication constant was set to 5. The tests was run five times with similar overall results, so the graphs are representative examples.

3.3.1 Adding Users

The purpose of this test is to examine the number of keys that must be retrieved to authenticate a key block as the number of users in the system grows. This is interesting partly because of latency issues, partly to keep the network and

processor usage to an acceptable level. Key blocks must be authenticated on many occasions, not only when new users join the system but also when a key is retrieved, e.g. to authenticate a data blocks.

It is difficult to predict the result of this test. The average path from a key to the root node is prolonged as the number of user keys in the system grows. This means that we would expect that more keys need to be fetched to authenticate a key block as time goes. On the other hand, when more users are added, there will be more activity on the nodes and hence a greater probability that the key caches (mentioned in Section 2.2.3) contain the needed keys already.

Description

The writer node constructs 2000 new public user keys and writes them to the network one at a time as key blocks, signing them with a random user key from the users it has already added (it keeps the private keys of the users it has added to be able to this). This means that the first new user must be signed by the root key, whereas the second user can be signed with equal probability by either the root key or the key of the first user, etc. Initially the other nodes only knows of the root key, but later insert all keys they have verified themselves in their local key cache as the test proceeds.

Results

Figure 3.1 illustrates that the number of keys fetched when receiving a key block is on average constant as the number of users in the system grows.

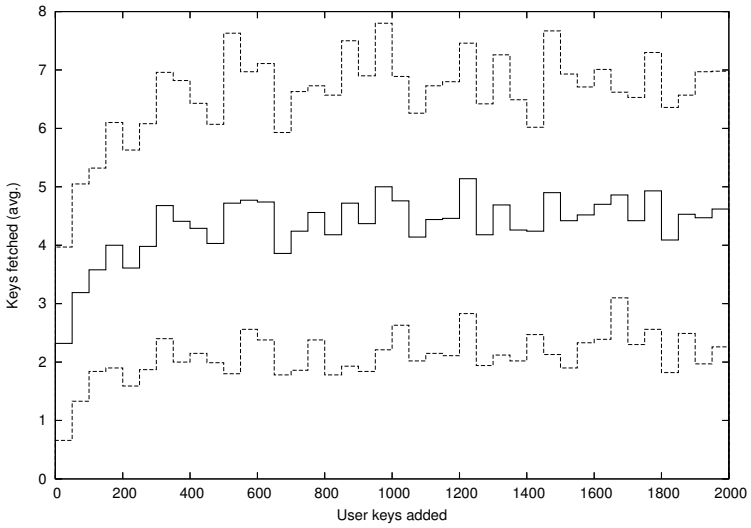


Figure 3.1: The middle graph shows the number of keys fetched to check a key block as more users are added. Each data point is an average of the 50 keys. The standard deviation is also plotted as the line above and below.

Between 0 to 200 added users, only few keys are cached but the average path length from the root key to the added keys are small, and only few keys need to be fetched. After 200-300 key blocks, the test enters a phase where each node has received one key on average (since the replication constant is 5) and the caching mechanism begins to have an effect. About this point in time, the curve stabilises. We attribute this to the effectiveness of the caching scheme.

Figure 3.2 illustrates the signing structure for the first 100 keys and can give an insight into why the caching scheme is effective. The tree is smaller near the root than at the leaves, and at the same time all signing paths must pass through the part near the root since they originate from it. Hence, caching the small bottom part of the tree can decrease the number of keys that must be retrieved substantially. For instance, caching the node labeled 1 will decrease the number of keys that must be retrieved by one for about 50% of the nodes in the tree.

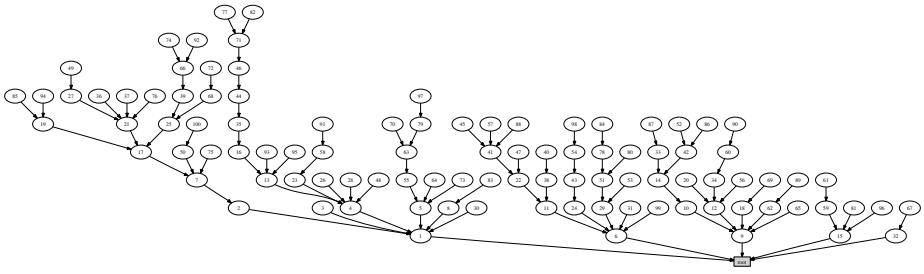


Figure 3.2: The graph shows which key was signed by which key for the first 100 keys. The root node is at the bottom.

Another thing that contributes to keeping the number of retrieved keys constant is that as the tree grows in size, the height is likely to only grow about logarithmically. Figure 3.3 shows the raw data that Figure 3.1 is based on, and evidently there are minor variations from the constant tendency from long, but not very long, signing paths.

A test with 5000 key blocks was also conducted to see if the trend continues for more users. The result is shown in Figure 3.4 and confirm what has been observed.

3.3.2 Overhead of Data Block Access Control

The purpose of this test is to examine the number of keys that must be retrieved to authenticate a data block over time when more and more data blocks are written. The keys that must be retrieved stem from the signatures on the access control lists and the signature on the block itself, as well as the keys that must be retrieved recursively to authenticate the keys used for the signatures.

We expect that the number of keys retrieved will decrease as more data

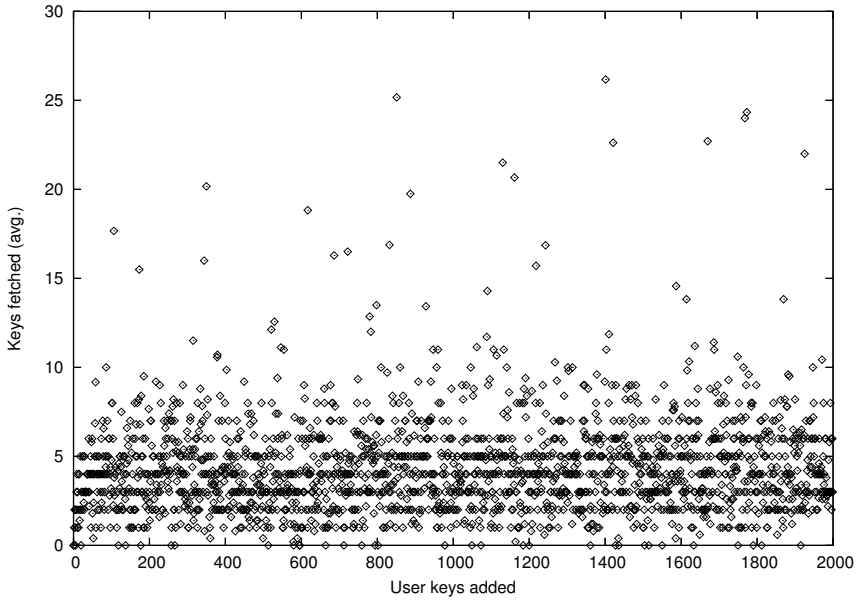


Figure 3.3: The number of keys fetched in average among the five nodes that receive a key, for each key written; compare with Figure 3.1.

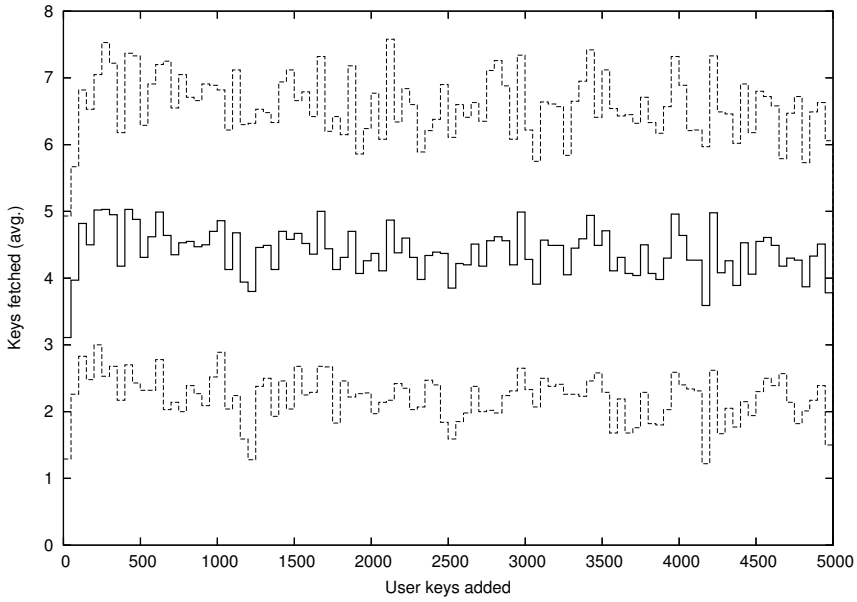


Figure 3.4: A graph similar to that of Figure 3.1, but with 5000 users in total.

blocks are written because of the caching. It is, however, not clear by how much it decreases.

Description

The test first creates 1000 key blocks signed with random other users in the system as in the previous test. Afterwards, 5000 data blocks are created and written one by one. The data blocks contain an access control list with between one and ten signatures and between one and ten rights for each of these signatures with uniform probability.

Results

The results of the test are shown in Figure 3.5. As the number of data blocks in the system increases, the number of average key fetches decreases. After 5000 data blocks have been distributed, the average number of fetches is only 1/3 of what it was at the beginning of the data block distribution. Clearly, the key caches can be very effective at decreasing the overhead of the access control features of the file system.

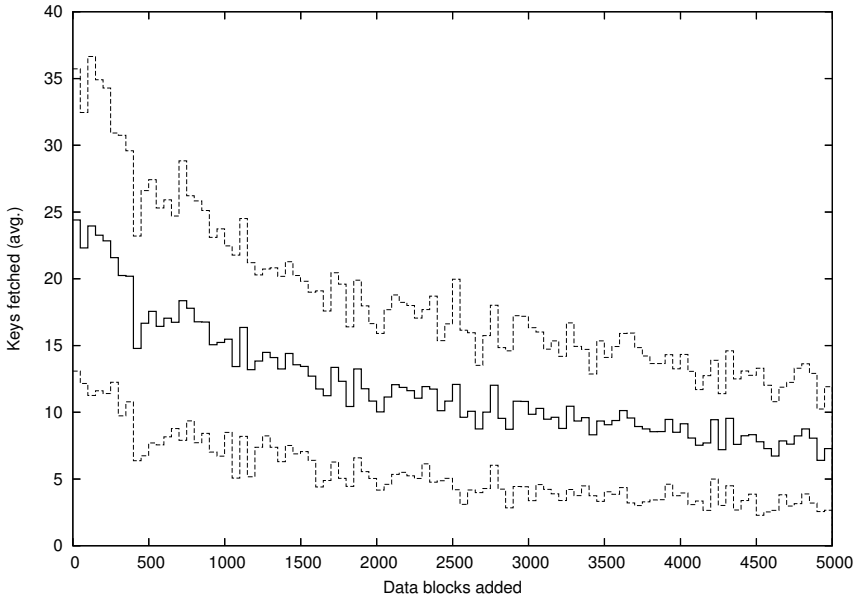


Figure 3.5: The middle graph shows the average number of keys fetched to authenticate data blocks as more data blocks are written to the system. The standard deviation is also plotted.

But the test also shows that there is a considerable overhead on average when receiving a data block, and the question is whether the overhead is still reasonable in practise. To answer this question, it is necessary with more information about the actual usage patterns. Fortunately, it seems likely that many

blocks will be modified multiple times or likewise read multiple times by the same node so that there can be almost 100% cache hits after the first read or write.

3.3.3 Read and Write Latency

The purpose of this test is to get an overview of the overall performance of the file system design. We measure the time it takes to read and write data blocks. For reading, this entails how long it takes to retrieve the block, authenticate it and then decrypt the contents. Writing measures how long it takes to send a data block to each of the r replicating nodes and get an acknowledgement from each.

Description

The test first creates 1000 new users one by one, signed with a random other user in the system as in the previous tests. Then 5000 data blocks are created and distributed one by one by. As in the previous test, the data blocks contains an access control list with between one and ten signatures and between one and ten rights for each of these signatures. The data part of the block is filled with 1-30 kilobytes of random data. After the data blocks have been written to the network, the writing node clears its local key cache and starts reading all the blocks again.

The latency of writing each data block and reading each data block is observed.

Results

Figure 3.6 shows the output of the write latency test. On the graph one can see seven different layers, the first layer, and 83% of the total amount of block writes, are data block writes that completed normally, the layers just above 10, 20, 30, 40 and 50 seconds are caused by the occurrence of 10 and 20 second network timeouts. It is not clear why the layer around 8-9 seconds occurred, but we think it is related to the high load caused by the cryptographic operations. We also believe this caused the network timeouts since the nodes should be able to contact each other.

Figure 3.7 illustrates the first layer and shows that the write latency slightly decreases as the number of data blocks in the system increases. We attribute this to the key caches, as explored in the two previous tests, since the keys that need to be retrieved to verify a block decreases. The write latency is, however, generally high. This might not be a problem in practise, though, since the write operations in some cases can be done in the background.

Figure 3.8 shows the result of the read test. The read latency decreases as the number of data blocks read increases. The test starts by clearing the key

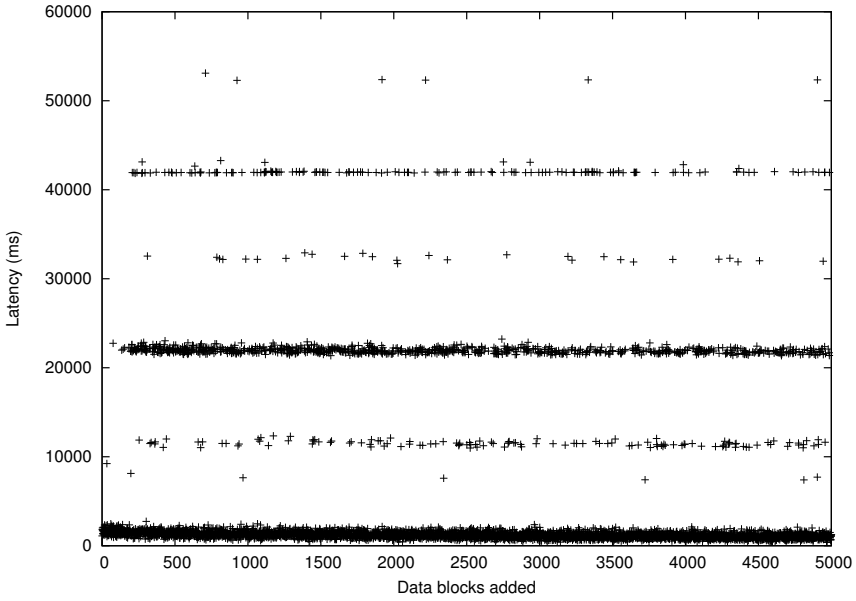


Figure 3.6: The time it takes to write a data block, for each data block written; the bottom layer is shown in Figure 3.7.

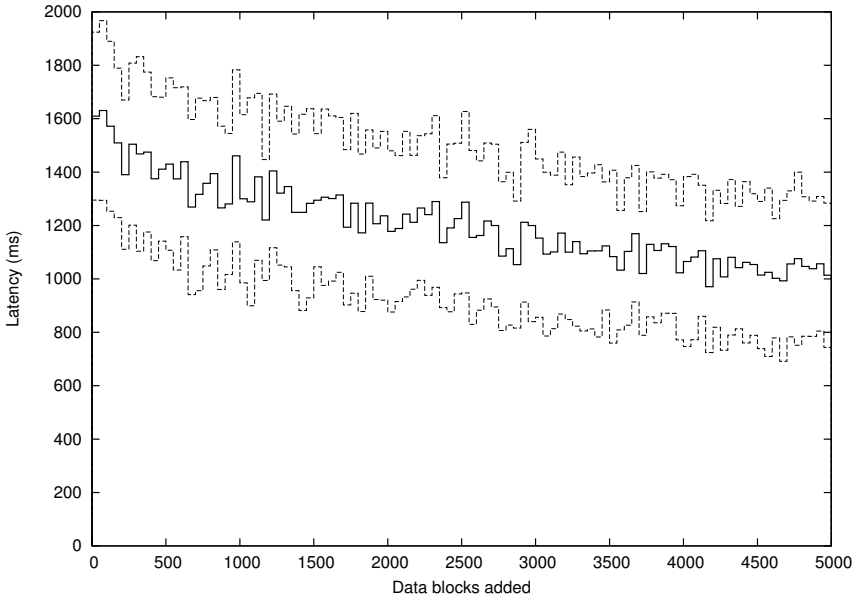


Figure 3.7: The middle graph shows the time it takes to write a block to the file system as more data blocks have been added. Each data point is an average of the 50 keys. The standard deviation is also plotted as the line above and below.

cache to simulate a new node joining the system. After 50 data blocks read the latency is halved, and the test settles on 200 ms latency after 500 read blocks. The test shows that the overhead of reading a block will be low for a reader that has been in the network for some time, whereas a new node has a notable latency.

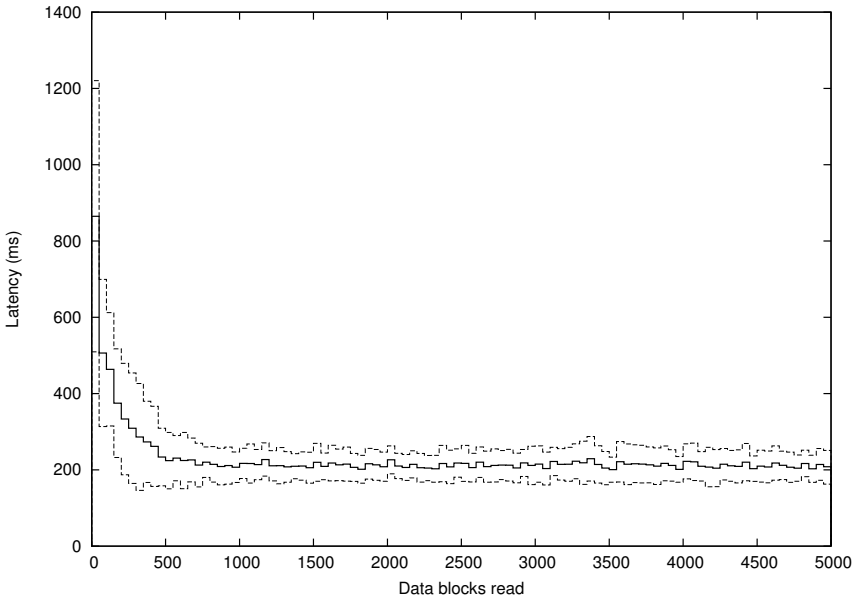


Figure 3.8: The middle graph shows the time it takes to read a data block as more data blocks have been read. The standard deviation is also plotted.

For both the read and write test, the considerations from the previous test still apply; when reading or writing the same block, almost all keys would be cached so that the latency is much lower. Furthermore, as explained in the beginning of this chapter, the implementation has not been optimised to prefer low-latency nodes, which might be able to improve the results considerably.

3.4 Distributed Mutual Exclusion Tests

This section documents the testing of the proposed algorithm for distributed mutual exclusion. The essential requirements for any mutual exclusion algorithm is to satisfy two properties, namely safety and liveness. Thus, the purpose of testing is to evaluate whether or not the algorithm satisfies these two properties. Finally, to evaluate the applicability of the algorithm the performance characteristics are also tested.

A common factor in all tests is that the implemented distributed mutual exclusion algorithm is tested under conditions of severe stress. This is to test the algorithm in a worst case scenario. In a realistic setting several conditions

vary arbitrarily. As examples we list a few below;

- A varying number of hosts ranging from a few to several thousands.
- A varying frequency of incoming mutex requests.
- Due to variation in popularity, the majority of mutex requests often pertain to a minimal subset of the entire resources (blocks) in the system.
- Varying network conditions e.g. latency, bandwidth, congestion, etc.
- Heterogeneous nodes in terms of platform and architecture.

To vary the above mentioned conditions by random does not seem as a reasonable solution as it would not simulate realistic conditions: as randomness scarcely resembles reality. Moreover randomness would obfuscate the conditions of the testing environment. Hence, to produce comparable results during worst case testing of the distributed mutual exclusion algorithm, the following assumptions are made, beyond those listed in Section 3.2.

- The frequency of mutex requests is very high. Actually, since peers does not do anything else but request mutexes, they request all the mutexes they need as fast as possible.
- The test environment is always comprised of 993 active hosts that acts as both mutex requesters and mutex granters.

3.4.1 Safety

The most important property of a mutual exclusion algorithm is safety. Safety is concerned with the extent to which an algorithm guarantees mutual exclusion, i.e. that only a single process is allowed to hold a given mutex at any time. Testing whether the algorithm satisfies the safety property should be performed by simulating the failure scenarios listed in Figure 2.2.6, thereby striving to break the algorithm. Unfortunately, this was not possible to achieve within the limited time period of this project. Instead, in the following, we argue that the algorithm indeed satisfies the safety property and elaborate the cases where safety is broken by unexpected behaviour of the underlying DHT network.

As documented in Section 2.2.6 the algorithm ensures safety as long as all of the nodes requesting a mutex for some resource, typically a file or a data block, agrees on at least one of the r-nodes guarding the mutex. In the following, nodes in the process of requesting a mutex is termed client nodes. Client nodes can only disagree on the r-closest nodes to a resource in situations where the arrival or departure of nodes have been detected at some client nodes but not at others.

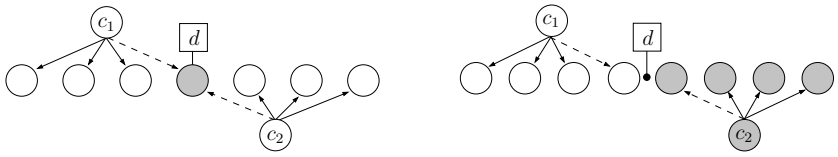


Figure 3.9: To the left, the view, at client nodes, of the r-closest nodes to d, needs only have a single node in common for safety to be satisfied. Conversely, if the client nodes view of the r-closest nodes to d does not have any nodes in common the algorithm cannot satisfy safety.

In Figure 3.9 two cases of disagreement on the world view between two client nodes are illustrated. In the first case, to the left, two clients, c_1 and c_2, are requesting a mutex for a resource d. Each client contacts the $r = 4$ closest nodes to d, according their world view. Thus, to obtain the mutex for d a client node needs to receive 4 grants. As illustrated the clients disagree on three nodes, and agree on a single one. Hence, only one client node can receive the fourth grant as the algorithm permits a node from granting two mutexes on the same resource.

In the second case, to the right, the same two client nodes, c_1 and c_2, again requests a mutex for resource d. They contact the $r = 4$ closest nodes, according to their world view, and both clients are granted a mutex for resource d. As illustrated, the algorithm is not able to coordinate the granting of a mutex when client nodes does not agree on at least one of the r-closest nodes to a resource.

3.4.2 Liveness

Another essential property of a mutual exclusion algorithm is liveness. Liveness describes whether a mutual exclusion algorithm guarantees that any request for a mutex is eventually granted.

The algorithm, as documented in Section 2.2.6, employs no measures to ensure liveness. Thus, the purpose of this test is to observe how this affects the operation of the algorithm in practice. The primary concern is to evaluate whether the algorithm is applicable despite the absence of liveness guarantees.

Description

The test is conducted by having 993 nodes request a mutex for the same resource at the same time. Each node requests a mutex with a lifetime of 1 second. When a client node is granted a mutex and the mutex expires it immediately re-requests the same mutex again. It is of course impossible to synchronise the time perfectly between the participating client nodes of the test. Hence, the results of similar runs of this test can be significantly different, and the test is executed several times. Each test is allowed to run for 3600 seconds or until all client nodes has been granted the mutex at least once.

Results

The results of two specific runs of the liveness test are depicted in Figure 3.10.

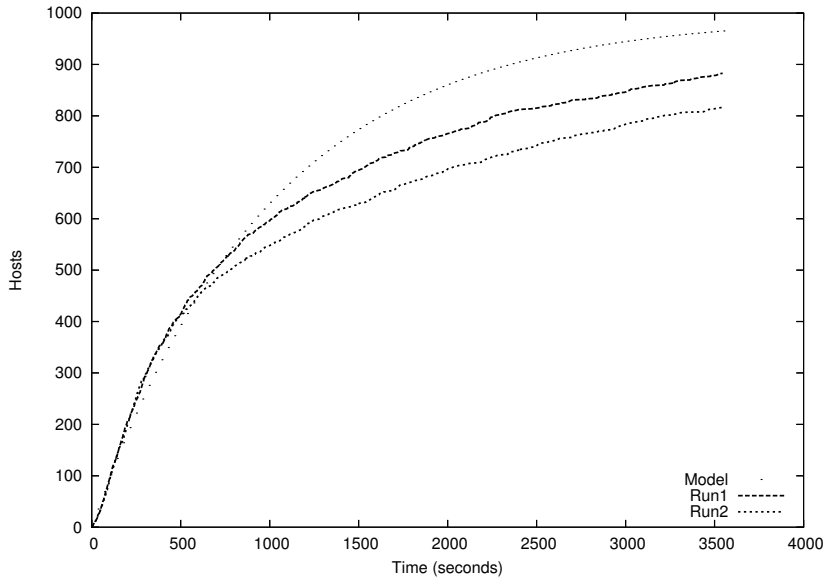


Figure 3.10: A plot of two different runs of the liveness test. The graph depicts how many hosts have aquired a mutex at least once as a function of elapsed time in the each test run.

A simplistic probabilistic model can provide some insight into why the graphs look like they do. If we assume that the mutexes are acquired in rounds of one second and that the probability of obtaining the mutex for each node is independent of the node and of time and hence is fixed at $p = 1/993$, then the probability that a node has not obtained the mutex after x rounds is $(1-p)^x$. Conversely, the probability that a node has obtained the mutex is $1-(1-p)^x$.

On average, we expect that the total number of nodes N that have obtained the mutex after x seconds is

$$N = 993 \cdot \left(1 - \left(1 - \frac{1}{993}\right)^x\right)$$

A graph of N as a function of x is plotted as *model* in Figure 3.10.

Initially, when no nodes have obtained any mutex ($x = 0$ seconds), the probability of any client node obtaining a mutex for the first time is 1, but the probability gradually decreases as the test progresses and most nodes get their first mutex. As the graph indicates both test runs follow the model for approximately the first 500 seconds. Of course, the model is crude and the assumptions only approximately hold.

Figure 3.11 presents an distribution of mutex grants on hosts during two runs of the liveness test. During the first run of the test, 112 clients never obtained a single mutex during the 3600 seconds. 124 client nodes were able to

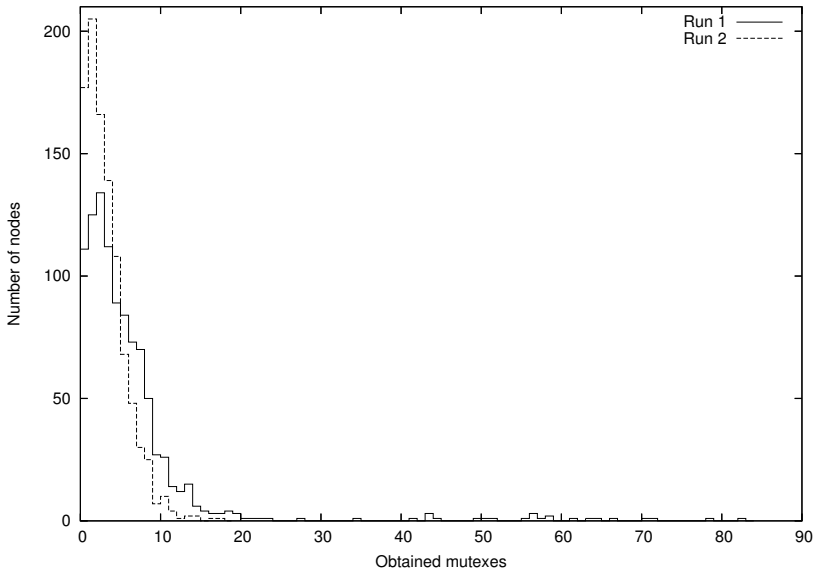


Figure 3.11: A plot of the distribution of mutexes granted during the two runs of the liveness test.

obtain a single mutex, and 133 and 111 client nodes, respectively, were able to obtain two and three mutexes. One client node was able to obtain a total maximum of 82 mutexes during the first run.

During the second run of the test, 177 clients never obtained a single mutex during the 3600 seconds. 205 client nodes were able to obtain one mutex, and 166 and 139 client nodes, respectively, were able to obtain two and three mutexes. During the second run, one node was able to obtain a maximum of 17 mutexes.

Hence, as expected, our proposed algorithm is not capable of ensuring that every node gets the mutex. Although most nodes get the mutex at least once, this still may be a problem in practise. Part of the problem is probably also that the algorithm makes clients that are denied the mutex several times wait even longer before they try to request it again. This favours clients that start afresh.

3.4.3 Performance

The purpose of this test is to evaluate how the proposed algorithm performs under severe stress by a having a large number of client nodes request different mutexes sequentially.

Description

To test the sequential performance of the algorithm, the 993 nodes request a mutex for n different resources. Each node generates the n resource names ran-

domly so that there is no overlap with the resources of the other nodes. The total amount of time needed for all nodes to obtain the n mutexes, as well as the number of mutexes obtained per second, are measured.

In order to evaluate the scalability of the proposed algorithm for distributed mutual exclusion both the sequential and parallel tests are executed for two different values of n, namely 250 and 500. All tests are allowed to run for at most 3600 seconds.

Results

The results of the performance tests with $n = 250$ and $n = 500$ are depicted in Figure 3.12 and Figure 3.13 repectively. A common problem is the occurrence of *sleeping periods*. In both tests the algorithm initially performed well in granting approximately 250 mutexes per second. However, the performance quickly degrades towards a short period with close to zero mutexes granted per second. We have no other explanation for this oddness than the machines in our test environment being overloaded.

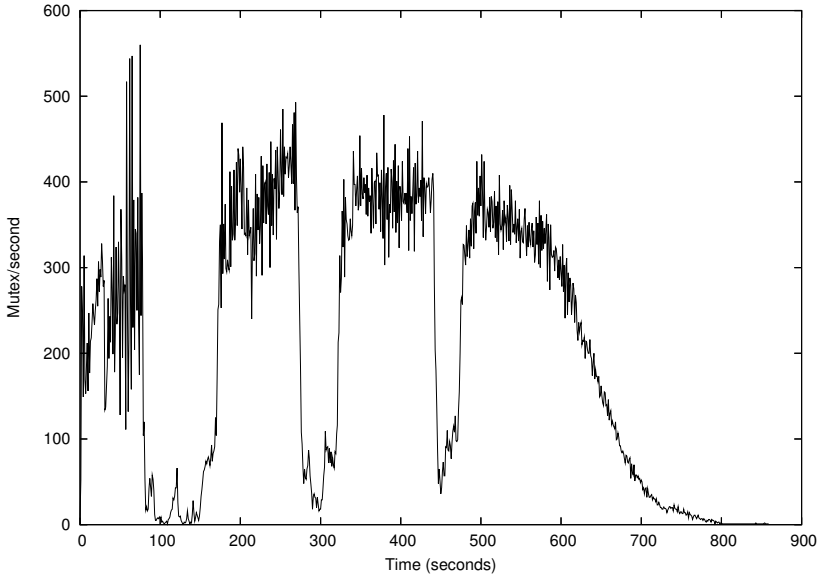


Figure 3.12: The graph shows mutex grants obtained per second. Each client requested 250 mutexes as fast as possible, amounting to a total number of 248.500 mutex requests in approximately 900 seconds.

In both performance tests the algorithm were able to sustain a rate of approximately 350 mutexes granted per second, and a peak performance at approximately 550 mutexes granted per second. As one can see in e.g. Figure 3.13 the number of mutexes granted per second decreased at the end of each test. This is because some of the clients finishes before the others.

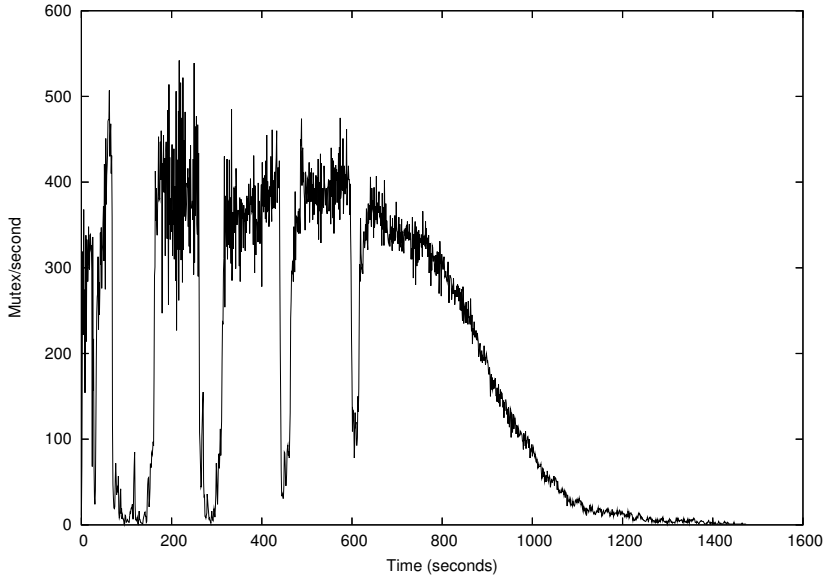


Figure 3.13: The graph shows mutex grants obtained per second. Each client requested 500 mutexes as fast as possible, amounting to a total number of 497.000 mutex requests in approximately 1500 seconds.

The performance tests revealed that even though the number of mutexes requested by the participating client nodes double, from 250 to 500, the amount of time needed for these mutexes to be granted was not. The total amount of time needed to grant 250 mutexes was less than 900 seconds, whereas the total amount of time needed to grant 500 mutexes was less than 1500 seconds.

So in conclusion, the performance of the algorithm seems reasonable when the obtained mutexes are independent.

3.5 Notification Tests

The goal of testing the notification algorithm is to evaluate the performance and fault tolerance of the advanced notification algorithm which uses a wait delay before notifying to reduce redundant messages.

The success criterion for the notification algorithm in each test is that all file subscribers receive a notification every time the file is modified. Furthermore the performance must be reasonable, as of not consuming unreasonable amounts of CPU time and being able to accommodate a large number of subscribers.

All tests were run using the advanced algorithm in a system consisting of 994 nodes and the replication constant set to 5. The system was left for 5 minutes before the test began to ensure that all nodes were ready.

The constants used in the advanced algorithm are kept fixed for all tests.

The initial timeout before the other notifier nodes assumes that a notifier has failed is set to 1 second, while the bump value used to increment the time out when a node sends a start notification message to the other notifiers is set to 5 seconds.

First the performance for large amounts of subscribers is evaluated, followed by a test of how the notifiers and the advanced algorithm copes with frequent file modifications. Lastly the behaviour of the algorithm with faulty notifiers is examined.

3.5.1 Stress Test with Many Subscribers

The purpose of this test is to explore the performance of the notification algorithm with many subscribers to a file. This is important as the notification algorithm must not be a major bottleneck for a node. Furthermore, the performance of the functionalities that rely on the knowing of file changes depends highly on the characteristics of the notification algorithm.

Description

In this test 993 nodes subscribe to a file residing on a specified node (the 994th node) whose behaviour is observed, and four others since $r = 5$. When all nodes have finished subscribing to the file, the closest node waits a while and then modifies the file, invoking the notification procedure. Since the observed node is the closest node, the advanced algorithm chooses this node to be the first to begin the notification procedure.

During notification, the CPU load and number of messages sent per second are observed, in conjunction with the outcome of the process, namely if all nodes have received a notification.

Results

The test was run a number of times with very similar results. All subscribed nodes received a notification message. During the notification process the observed node was able to send the 993 notification messages in an average of 1500 ms (disregarding the acknowledgements from the subscribers), while the CPU load during the process was negligible, below 5%.

The low CPU load hints that the performance is bounded by the network communication. This probably stems from the overhead of creating TCP connections, since the amount of time that the closest node waits before modifying the file is long enough to have closed the connections in the TCP pool.

Since the entire procedure takes some time, the last node that receives a notification may receive it a few seconds later than the first node. For fairness reasons, it might be a good idea to transmit the notifications in a random order so that no node gets placed last every time.

In a setup like NorduGrid with are few and fast hosts, the notification algorithm seems to be able to handle the job quite well. Compared to a polling approach, the notifications save considerable bandwidth.

3.5.2 Stress Test with Frequent File Modifications

The aim of this test is to observe how the system reacts when a file is modified frequently. This should give a feel of how well the algorithm performs in a busy environment with many active participants.

Description

To ensure that the performance measurements are as precise as possible, the test incorporates three different participants, a notifier node which is monitored in regards to CPU load and messages sent, a file modifier node which takes care of all write operations, and the subscriber nodes. This division of labour is intended to ensure that the measurements done on the notifier node corresponds to a real scenario.

The system is left to stabilise before the file modifier begins to do write operations with a constant rate of one operation every 1000 ms. The test was later repeated with write operations done every 500 and 100 ms. Since write operations start by doing a lookup in the overlay network, the round-trip time of the lookups result in a practical problem when using a single node to simulate frequent file modifications. The node doing write operations is not able to send the modifications fast enough. To remedy this problem the test omits the lookups.

Because of the TCP connection pool, one would expect the first notification round to take more time than the following rounds because each TCP connection has to be created in the first round, whereas they can be reused in the following rounds.

The observations to be done in this test are as in the previous test.

Results

The results of the test is shown in Table 3.1 which shows the first five rounds of notifications.

All test runs had a overlapping start phase where two rounds of notifications were being processed at the same time. We attribute this to the connection establishment overhead. Once the notifying node has established all needed connections, the remaining messages can be sent very quickly. For the 1000 ms test the rounds with open connections were executed in 100-130 ms each, well below the 1000 ms between the rounds.

In the test with a write interval of 500 ms the results are very similar. The only difference is that the initial phase had three overlapping rounds instead

	1000 ms			*500 ms*			*100 ms*		
	start	end	time	start	end	time	start	end	time
1	1.651	3.495	1.844	20.193	22.007	1.814	4.746	6.577	1.831
2	2.655	3.496	841	20.699	22.006	1.307	4.854	6.578	1.724
3	3.659	3.783	124	21.201	22.006	805	4.964	6.577	1.613
4	4.664	4.785	121	22.039	22.172	133	5.183	6.578	1.395
5	5.667	5.788	121	22.205	22.312	107	5.401	6.578	1.177

Table 3.1: The results of the tests with 1000 ms, 500 ms and 100 ms write intervals with the start and end times of the rounds and the total time spent sending the notifications.

of two.

For the test with a write interval of 100 ms, the notifying node was not able to keep up with the incoming block changes. The first few notification rounds were taken care of, while the rest drowned in the growing number of pending notifications. The notification rounds which did finish took between 130 ms and 3800 ms.

During the initial phases in the three tests, the overlapping of notification rounds generate a number of obsolete messages since the notifier node sends a notification for all changes. For the 1000 ms test, it was between 60-120 obsolete messages out of the total of 993 messages for the first round. The growing number of obsolete notifications is also what overwhelms the notifier node in the test with 100 ms.

Clearly, the implementation should be modified to avoid sending these messages. If the modification ensures that each notification round does not notify the same nodes first each time but instead notifies the nodes in a round-robin fashion, then even the test with 100 ms write intervals should be able to keep up. Otherwise the algorithm seems to be delivering good performance.

3.5.3 Fault Tolerance of Notifiers

In this test the correctness of the algorithm is evaluated with respect to node failures among the r closest notifier nodes. It is essential that all subscribers receive a notification even in an error-prone grid environment.

Description

The system is set up with one node as a file modifier and the remaining 993 nodes as subscribers. The file modifier writes the file when the system has settled. In the first test run the closest node is killed halfway through its notification procedure such that the second-closest node becomes responsible for the notification round. In the second test run, the four closest of the five nodes are killed halfway through their notification rounds, making the last node responsible.

During these erroneous scenarios, the behaviour of the algorithm is observed together with the latency of the notification process.

Results

Both test runs were run twice. In all cases, all subscribers received a notification which indicates that the algorithm behaves much as expected in an erroneous environment. The maximum latency between the file being written by the file modifier and the node that received a notification as the last was 7114 and 7125 ms respectively when one node was killed, and 21916 and 21699 ms when four nodes were killed. The distribution of the latencies are shown in Figure 3.14 and 3.15.

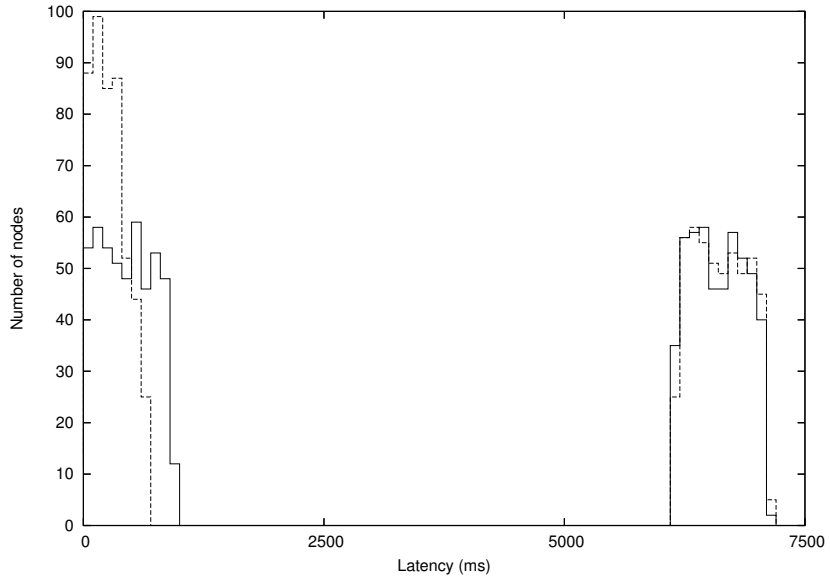


Figure 3.14: The distribution of the latencies in the test where the first notifier was killed.

The delay from notifier to subscriber largely depends on the choice of the constants in the algorithm. Since we chose the initial timeout to be 1 and the bump value to 5 seconds, each time a nodes starts notifying and fails, the procedure takes as a minimum an additional 5 seconds.

When a node begins a notification round, a bump of the timeout is triggered on all the remaining notifiers, which ends up as stagnation in the reception of notifications if that notifier fails. For the test with one notifier killed, Figure 3.14 clearly illustrates this stagnation as a 5 second interval from 1000 ms to 6000 ms with no nodes receiving notifications.

Figure 3.15 shows a similar behaviour for the test with four killed notifiers. The stagnation is longer due to the additional nodes failing half-way through the procedure and thus incrementing the timeouts.

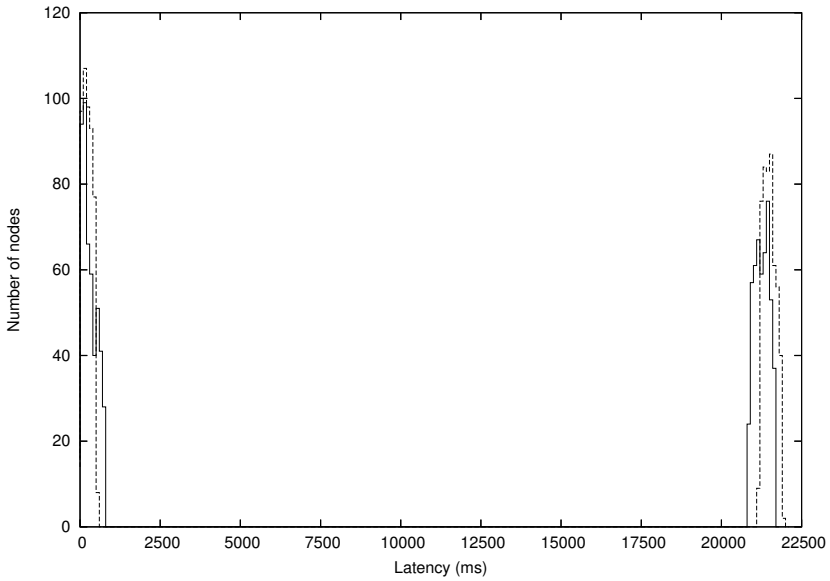


Figure 3.15: The distribution of the latency in the test where the four first notifiers were killed.

The algorithm seems to cope well with failures among notifier nodes. The test also indicates that the constants should perhaps be optimised to the specific setting, due to their large influence on the performance.

Chapter 4

Conclusion

4.1 Summary of Project Results

In Chapter 1 the basic nature of computational grids was studied along with the challenges inherently present when implementing grids. Existing major grid efforts were presented along with their defining characteristics. A thorough presentation of NorduGrid leads to an identification of problems in the current design and implementation. Based on this we outline a simpler grid architecture centered around a distributed file system built on a completely decentralised overlay network.

The outline leads to a proposal for a new design for grid computing environments in Chapter 2. The design is based on a distributed and decentralised file system, the information service, with support for access control through asymmetric cryptography, notifications of file changes and mutual exclusion. The file system is the primary medium for communication between the participating entities in the grid.

A separate data service is also designed for handling the massive amounts of data that current grid environments face. The data service in our design makes use of the information service to store information of the data in the grid, but actual data transfers are performed in a point-to-point manner, preferably from multiple hosts in parallel.

It is essential for an evolving grid environment that it is extensible so that support can be added for various types of resources. We have discussed some general issues for resources, including how to handle and schedule jobs and handle job failures, and outlined a design for two types of resources, namely batch clusters and storage systems, the ones that NorduGrid currently consists of. In our design, resources use the information service to register information about the resource and for retrieving jobs for the resource, making use of the data service for input and output data transfers. This effectively reduces the task of adding new resource types to writing an interface between the infor-

mation service and the actual resource.

In a grid environment user components are also needed for job submission and grid monitoring tools. In our design the information needed for these tools are stored in the information service and the jobs are submitted to it.

The fault-tolerance, performance and scalability of the information system is central to our design. Hence, we decided to test the features that we needed for the information service, access-controlled file system operations, notification of file changes and a mutex algorithm, on top of a decentralised distributed hash table. We did this by enhancing and adapting a Kademlia-based [31] file system implementation.

Then in Chapter 3 we presented various tests to evaluate the information service. The test of the file system showed that checking key blocks is constant in the number of keys that need to be fetched to authenticate the block when adding more users, while checking a data block required fewer key fetches as the number of data blocks already checked grows. Finally the latency of reading a data block is notable for clients nodes, while small for fabric nodes. The latency of writing data blocks is high, ranging from 1.2 seconds to 1.6 seconds.

The tests of the distributed mutual exclusion algorithm showed that the algorithm scales reasonably well and that performance is satisfactory even under conditions of severe stress. The safety property of mutual exclusion was discussed and we implied that that the proposed algorithm guarantees safety, even with up to r nodes failing. However, the test also revealed that the lack of liveness guarantees may be a problem in practise.

The notification tests showed that the algorithm performs well even with many subscribers, but that there is a problem with the implementation with frequent file modifications. The algorithm was found to be fault tolerant when nodes were brought to fail deliberately, although the latency before notifications are received increases greatly. This decrease depends highly on the constants used in the algorithm, which should be tuned for the specific setting.

4.2 Strengths and Weaknesses

Having summed up the achievements of the project, we will now try to stress the strengths and weaknesses of the grid architecture we have designed.

First of all, the information service makes the system very flexible. It relieves the entities using it from having to communicate through custom network protocols and supports group coordination. And it does not impose any particular constraints on the information stored in it except for the hierarchical directory structure.

Then there are two properties that we have explicitly dealt with by designing the system on top of a distributed hash table, fault tolerance and scalability. Both are properties that are important to have from the very first since they are difficult to graft onto an existing design.

The fault tolerance of the information service means that as long as the systems on top of it are designed with some care, it is possible to have a very robust grid where the failure of a participant such as a resource does not affect any other participants including the users. This is an important aspect in the amount of manual maintenance needed and thus the overall user-friendliness of the grid.

Although the information service itself is inherently scalable, also practically speaking since it is easy to add more fabric nodes because of its self-organising properties, the most important reason why a grid based on our system should be able to scale is perhaps that our architecture is flexible enough to support several alternative models of operation, e.g. for scheduling.

Another benefit of our system is that it deals with the problem of user administration and public-key distribution explicitly in a decentralised manner, thus relieving the system from a centralised human bottleneck.

There are also some weaknesses of our design. One that was highlighted under the test of the file system, is that the file system operations may have a high latency because of the security model. This may, however, not be a problem for the currently deployed large grids because they are mostly batch systems where jobs generally take a long time to process. Another issue with the access control system is that due to the use of asymmetric cryptography in a decentralised manner, it is difficult to revoke rights from a truly malicious party. It is unclear whether it is possible to amend this problem without imposing further performance penalties.

We have tried to alleviate other weaknesses during the design phase. But since the foundation of our architecture is quite new compared to the tried and tested client-server architecture, there are likely to be problems that we have not thought of and that will only show up when a complete grid design is implemented and running.

In spite of this uncertainty, we believe that the our design could provide a viable alternative to the architecture of NorduGrid, and that it has the potential to make future grids more robust, less of a burden to maintain and develop, easier to contribute resources to, and easier to extend to new fields.

4.3 Future Work

Since our time was limited we have mostly dealt with the foundations of the architecture. With more time available, there are a lot of possible future directions one could examine:

- Because of the sheer scope of a complete grid system, we have necessarily had to leave some loose ends here and there in the design, in particular in the upper level components. Many details must be carefully examined and decided upon before a complete grid can be implemented.

- However, developing and evaluating a prototype implementation of the entire grid design is an ultimate test of the design. Testing should preferably be performed in a real environment, e.g. using PlanetLab [39]. With a prototype running, it would be possible to compare with other grid solutions and complete evaluation of the design could be made.
- One could experiment with other data service designs than what we have suggested. It is certainly easy to think of more advanced solutions. One thing that is probably important in practise is automatic replica management so that one only has to specify the desired number of replicas and then the system takes care of maintaining them.
- Network partitions where the nodes in the network are separated into two distinct groups are not handled by our system. The problem is that the two distinct groups may continue independently which for example may result in inconsistant job queues. This needs to be investigated to see whether it is possible to mitigate.
- The mutual exclusion algorithm does not guarantee liveness. However, another protocol has recently been proposed for maintaining mutual exclusion in dynamic peer-to-peer systems [30]. Since this protocol does guarantee liveness, it may be preferable.
- A scheduler framework and a scheduler must be designed and tested. Preferably, a scheduler should optimise the overall throughput without too much overhead, produce robust schedules and still be scalable. It is, however, a non-trivial question to what extent these goals can be fulfilled. With a simple algorithm, some files in the information service may end up as bottlenecks so that the result is not scalable enough.
- The design could be extended to explicitly address desktop machines too. Various issues, such as scalability, transient network connections and limited access to the Internet, must be addressed. A solution could be the development of grid hubs that operate as proxies for the desktop machines in an organisation, with the hubs having a proper connection to the grid. This could also handle some of the scalability issues involved.
- We have not addressed accounting and payment for grid services. With the provided log facility, it would be possible to add this on top of the existing design, but it may be preferable to have a more integrated solution.

Bibliography

[1] Alchemi - .NET grid computing framework. http://www.alchemi.net/.

[2] David P. Anderson. Public computing: Reconnecting people to science, March 2004.

[3] Storage Networking Industry Association. Common internet file system technical reference v. 1.0, March 2002.

[4] The ATLAS experiment. http://atlasexperiment.org/.

[5] Paul Avery and Ian Foster. Griphyn annual report for 2003–2004, August 2004.

[6] A. Back. Hash cash - a denial of service counter-measure.

[7] Boost C++ libraries. http://www.boost.org/.

[8] Rajkumar Buyya. Grid computing info centre FAQ. http://www.gridcomputing.com/gridfaq.html.

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

[10] GNU Common C++ Resources. http://www.gnu.org/software/commoncpp/.

[11] George Coulouris, Jen Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design, 3rd edition*. Addison Wesley, 2001.

[12] distributed.net. http://www.distributed.net/.

[13] eDonkey2000 - Overnet. http://edonkey.com.

[14] P. Eerola, B. Konya, and O. Smirnova et. al. ATLAS Data-Challenge 1 on NorduGrid. In *Computing in High Energy and Nuclear Physics*, 2003. http://www.nordugrid.org/documents/MOCT011.pdf.

[15] P. Eerola, B. Konya, and O. Smirnova et. al. The nordugrid architecture and tools. In *Computing in High Energy and Nuclear Physics*, 2003. http://www.nordugrid.org/documents/MOAT003.pdf.

[16] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2. Morgan-Kaufmann, 1998.

[17] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Lecture Notes in Computer Science*, 2150:1–??, 2001.

[18] M. Freedman and D. Mazières. Sloppy hashing and self-organizing clusters, 2003.

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

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

[21] Onne Gorter. Database File System. `http://ozy.student.utwente.nl/projects/dbfs/`.

[22] Richard Grimes. Code Name WinFS. `http://msdn.microsoft.com/longhorn/default.aspx?pull=/msdnmag/issues/04/01/WinFS/default.aspx`.

[23] N. HARVEY, M. JONES, S. SAROIU, M. THEIMER, and A. WOLMAN. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of USITS*. USENIX, 2003.

[24] Anders Rune Jensen, Jasper Kjersgaard Juhl, Lau Bech Lauritzen, Ole Laursen, and Michael Gade Nielsen. Heurika: A decentralised shared file system for local area networks. Student report (3rd year), Aalborg University, December 2003.

[25] Henrik Thostrup Jensen and Jesper Ryge Leth. A Job Manager for the NorduGrid ARC. Master's thesis, Aalborg University, June 2004.

[26] Kazaa. `http://www.kazaa.com/`.

[27] B. Kónya. NorduGrid server installation instructions. `http://www.nordugrid.org/documents/ng-server-install.html`.

[28] Legion: A worldwide computer. `http://legion.virginia.edu/`.

[29] The LHC grid computing project. `http://lcg.web.cern.ch/LCG/`.

[30] Shi-Ding Lin, Qiao Lian, Ming Chen, and Zheng Zhang. A practical distributed mutual exclusion protocol in dynamic peer-to-peer systems. In *The 3rd International Workshop on Peer-to-Peer Systems*, 2004.

[31] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS02*, March 2002.

[32] 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.

[33] Minimum intrusion grid. `http://www.imada.sdu.dk/~anden/MiG/`.

[34] NorduGrid middleware, the advanced resource connector. `http://www.nordugrid.org/middleware/`.

[35] Seth Nickell. GNOME Storage. `http://www.gnome.org/~seth/storage/`.

[36] OpenPBS public home. `http://www-unix.mcs.anl.gov/openpbs/`.

[37] OpenSSL: The Open Source toolkit for SSL/TLS. `http://www.openssl.org/`.

[38] PBS Pro Home. `http://www.pbspro.com/`.

[39] PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org/`.

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

[41] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[42] 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.

[43] SETI@Home: Search for extraterrestrial intelligence at home. `http://setiathome.ssl.berkeley.edu/`.

[44] 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.

[45] Sun Microsystems, Inc. RFC1094: NFS: Network file system protocol specification, March 1989.

[46] Osamu Tatebe, Youhei Morita, Satoshi Matsuoka, Noriyuki Soda, and Satoshi Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 102–110, 2002.

[47] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[48] TORQUE Resource Manager. `http://www.supercluster.org/torque/`.

[49] Andrei Tsaregorodtsev, Vincent Garonne, and Ian Stokes-Rees. Dirac: A scalable lightweight architecture for high throughput computing. In *5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[50] UNICORE. `http://www.unicore.org/`.

[51] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). `http://www.ietf.org/rfc/rfc2251.txt`.

[52] T. Ylonen. SSH - secure login connections over the internet. Proceedings of the 6th Security Symposium) (USENIX Association: Berkeley, CA):37, 1996.

[53] 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.