

Reliable Peer to Peer Grid Middleware

Matthew Leslie

Green College, University of Oxford

Thesis submitted in fulfilment of the requirements for the degree of Doctor
of Philosophy at the University of Oxford

Trinity Term, 2006

ABSTRACT

Grid computing systems are suffering from reliability and scalability problems caused by their reliance on centralised middleware. In this thesis, we argue that peer to peer middleware could help alleviate these problems.

We show that peer to peer techniques can be used to provide reliable storage systems, which can be used as the basis for peer to peer grid middleware. We examine and develop new methods of providing reliable peer to peer storage, giving a new algorithm for this purpose, and assessing its performance through a combination of analysis and simulation. We then give an architecture for a peer to peer grid information system based on this work. Performance evaluation of this information system shows that it improves scalability when compared to the original centralised system, and that it withstands the failure of participant nodes without a significant reduction in quality of service.

New contributions include dynamic replication, a new method for maintaining reliable storage in a Distributed Hash Table, which we show allows for the creation of more reliable, higher performance systems with lower bandwidth usage than current techniques. A new analysis of the reliability of distributed storage systems is also presented, which shows for the first time that replica placement has a significant effect on reliability. A simulation of the performance of distributed storage systems provides for the first time a quantitative performance comparison between different placement patterns. Finally, we show how these reliable storage techniques can be applied to grid computing systems, giving a new architecture for a peer to peer grid information service for the SAM-Grid system. We present a thorough performance evaluation of a prototype implementation of this architecture. Many of these contributions have been published at peer reviewed conferences [1, 2].

ACKNOWLEDGEMENTS

I am thankful to my supervisors, Todd Huffman and Jim Davies, for their patience and support. I am also grateful to the SAM-Grid team, the Oxford University Software Engineering Group and the Oxford Physics Grid Computing Group for providing an inspiring and stimulating environment in which to work. I am particularly grateful to Sinisa Veseli and Adam Lyon, both of whom provided invaluable help and support for my work on SAM-Grid. Thanks are also due to Timothy Bagot, whose advice has helped me out of tight mathematical spots on more than one occasion.

I owe my sanity to my friends, and this thesis might well never have been completed without the support of the fermiposse and the three musketeers. Thanks to Marilyn Audsley, Farrukh Azfar, Lucio Cerrito, Paul Coe, Ben Cooper, Katherine Copic, Dan Cyr, Toby Davies, Tamsin Edwards, Sinead Farrington, Anant Gajjar, Nathan Goldschmidt, Dean Hidas, Cigdem Issever, Martin Griffiths, Helen Hayward, Bo Jayatilaka, Michael Kirby, Marcus Lewin, Dustin McGivern, Jolie De Miranda, James Monk, Emily Nurse, Philip Perea, Andy Pilko, Nicola Pounder, Jonas Rademacker, Stephen Robinson, Aidan Robson, Laura Somerville, Paul Telford, Muge Karagoz Unel and Simon Waschke for making being a DPhil student more fun than it rightly ought to be.

This work was made possible by funding from the Particle Physics and Astrophysics Research Council, to whom I am grateful.

CONTENTS

1. <i>Introduction and Motivation</i>	1
1.1 Introduction	1
1.2 What is a Grid Computing System?	2
1.3 Why Do we Need Grid Computing Systems?	3
1.4 Grid Middleware	3
1.5 Case Study: The SAM-Grid system	5
1.5.1 Central Systems in SAM-Grid As A Load Bottleneck	6
1.5.2 Central Systems in SAM-Grid As A Single Point Of Failure	7
1.6 The Need for Peer To Peer Grid Middleware	8
1.7 Summary	9
2. <i>Peer to Peer Computing</i>	10
2.1 Introduction	10
2.2 Definition	10
2.2.1 Properties of Peer to Peer systems	11
2.3 A Taxonomy of Peer to Peer Systems	12
2.3.1 Centralised Peer To Peer Systems	12
2.3.2 Unstructured Peer To Peer Systems	14
2.3.3 Structured Peer To Peer Systems	16
2.4 Security in Structured Peer To Peer Systems	26
2.4.1 Routing Attacks	26
2.4.2 Storage, Denial of Service, and Join/Leave attacks	27
2.4.3 Sybil Attacks	28
2.5 Hetrogeneous and Unreliable Physical Networks	31
2.5.1 Non-uniform Network Latency	31
2.5.2 Non-uniform Network Connectivity	31
2.6 Summary	32
3. <i>Reliable Storage in Distributed Hash Tables</i>	33
3.1 Introduction	33
3.2 Replication Algorithms	34
3.3 Design space of reliable peer to peer storage	34
3.4 Unstructured/Soft State Replication	35
3.5 Proactive Replication	35
3.6 Metadata based replication	36
3.6.1 Indexed Replication	36
3.6.2 Version ID	37

3.7	DHash replication	37
3.7.1	DHash Data Fetch Algorithm	39
3.7.2	DHash Updates	39
3.8	Dynamic Replication	40
3.8.1	Replica Enumeration	40
3.8.2	Dynamic Replication: Improving Replica Enumeration	42
3.8.3	Dynamic Allocation Functions	44
3.8.4	Dynamic Data Fetch and Update Algorithms	49
3.9	Summary	51
4.	<i>Configuration of Reliable Storage Algorithms</i>	52
4.1	Introduction	52
4.2	Assessing Reliability	52
4.2.1	Modelling Reliability	53
4.2.2	Validating Our Model	55
4.3	Achieving Reliability	56
4.3.1	Comparing Block Placement to Successor Class Placement	57
4.3.2	Finger Placement	58
4.3.3	Setting Maintenance Frequency	59
4.4	Setting R_{MAX} and R_{MIN} To Allow Recovery From Allocation Collisions	61
4.4.1	Successor, Block, and Predecessor Allocation	61
4.4.2	Finger Allocation	63
4.4.3	Symmetric Allocation	64
4.5	Keyspace Smoothing	64
4.6	Summary	68
5.	<i>Performance Evaluation of Reliable Storage Techniques</i>	69
5.1	Introduction	69
5.2	Our Simulation	69
5.3	Simulation Results	71
5.3.1	Fetch Performance	71
5.3.2	Communication Costs	75
5.3.3	Load Balancing	79
5.3.4	Write Performance	81
5.4	Summary	83
6.	<i>A Peer To Peer Grid Information Service</i>	85
6.1	Introduction	85
6.2	Suitability Of A Peer to Peer Architecture	85
6.3	An Architecture for a Peer to Peer Information System	88
6.3.1	System Capabilities	88
6.3.2	Reading Data	90
6.3.3	Writing Data	90
6.4	Prototype Implementation and Performance	92
6.4.1	Prototype Implementation	92

6.4.2	Experimental Testbed	94
6.4.3	Performance	94
6.4.4	Fetch Latency	95
6.4.5	Impact of System Load on Fetch Latency	95
6.4.6	Impact of Data Access Pattern on Fetch Latency	96
6.4.7	Impact of System Instability on Fetch Latency	98
6.4.8	Impact of System Size on Fetch Latency	98
6.4.9	Impact of Physical Network Latency on Fetch Latency	100
6.4.10	Network Communication	103
6.4.11	Impact of System Size on Network Communication	103
6.4.12	Impact of System Stability on Bandwidth Consumption	104
6.4.13	Impact of Quantity of Data Stored on Bandwidth Consumption	106
6.4.14	Impact of System Load on Bandwidth Consumption	107
6.5	Summary	108
6.6	Related Work	108
7.	<i>Conclusions and Further Work</i>	110
7.1	Conclusions	110
7.2	Areas for Further Work	111
7.2.1	Applying Dynamic Replication To Erasure Coding	111
7.2.2	Allowing for Diurnal Churn	112
	<i>Bibliography</i>	115

LIST OF FIGURES

1.1	Time taken by SAM-Grid to run a complex metadata search (seconds) plotted against time of day. Database backups are running between 2:30 and 9:30 a.m.	7
2.1	Overview Of The Napster Architecture.	13
2.2	Overview Of The Gnutella Architecture.	15
2.3	An example two dimensional CAN keyspace partitioned between 6 nodes.	18
2.4	Kademlia binary tree. The black dot indicates the node, and the dashed ovals indicate the areas from which nodes may be selected for the largest three k-buckets	20
2.5	A Chord ring containing 7 nodes, in a keyspace between 0 and 2^7 showing finger table entries for node 64	21
2.6	A Chord lookup for item 16 is routed from node 30 to item 16's successor	22
3.1	Local Maintenance runs on node 10, synchronising the databases of that node and a number (here three) of its successors. Arrows indicate database synchronisation traffic.	38
3.2	Global Maintenance runs on node 16. A new node (15) has joined since maintenance last ran, and node 16 is no longer within three hops of node 10. In this example, we store 4 copies of each object, and so node 16 is no longer responsible for storing replicas of node 10's data. It sends that data back, and deletes it from its local database.	38
3.3	Illustrative example of successor allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.	45
3.4	Illustrative example of predecessor allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.	46
3.5	Illustrative example of block allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.	47
3.6	Illustrative example of finger allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.	48
3.7	Illustrative example of symmetric allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.	49
4.1	Reliability of simulated Chord systems as compared to our models. Models are shown as solid lines, simulations are dashed, with error bars giving 95% confidence intervals. For the simulations, the x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.99$ column in Table 4.4.1. For comparison, a simulation in which data items are placed according to a pseudo-random distribution function is also shown.	56

4.2	Maximum system size for a system failure probability of $<10^{-6}$, for varying node failure probabilities. These figures are for a replication factor of 8. . . .	57
4.3	Minimum replication factor for a system failure probability of $<10^{-6}$ against node failure probability. These figures are for a 1000 node system.	58
4.4	Probability of system failure for three allocation functions in a 500 node system where 50% of nodes fail, for various replication factors.	59
4.5	Minimum maintenance frequency(in maintenance runs per half-life) necessary to maintain a 500 node system with $FAIL(N, R, S) < 10^{-6}$, versus replication factor.	60
4.6	Effect of smoothing on keyspace distribution, with various l	66
4.7	The average number of join retries necessary to achieve a given smoothing factor. Error bars show one standard deviation	67
4.8	The standard deviation of keyspace sizes versus smoothing factor. Join Period shows s.d. after 50,000 nodes join an initially empty ring of size 2^{32} . Steady state shows the same system after 100,000 churn events (node join followed by node departure).	67
4.9	The values of R_{MAX} necessary to achieve $\pi = 0.99$ for various smoothing factors and values of R	68
5.1	Fetch latency of Chord systems with various placement functions plotted against system size. Error bars show 95% confidence intervals. A logarithmic fit between the data points is shown. These results are for $R = 3$ and 80 maintenance calls per half-life. For dynamic replication, $R_{MIN} = 3$ and $R_{MAX} = 8$	72
5.2	Fetch latency of Chord systems with various placement functions plotted against maintenance frequency. Error bars show 95% confidence intervals. These results are obtained with $N=300$, $R_{MIN}=6$ and $R_{MAX} = 14$	73
5.3	Fetch latency of Chord systems with various placement functions plotted against replication factor. Error bars show 95% confidence intervals. These results are obtained with $N=300$ and 80 maintenance calls per half-life. The x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.95$ column in Table 4.4.1	74
5.4	Parallel fetch latency of Chord systems with various placement functions plotted against replication factor. Error bars show 95% confidence intervals. These results are obtained with $N=200$. The x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.95$ column in Table 4.4.1.	76
5.5	Proportion of all stored data moved in one half-life by each algorithm with various maintenance frequencies. Error bars show 95% confidence intervals. These results are obtained with $N = 300$ and $R_{MIN} = 6$ and $R_{MAX} = 14$. . .	77
5.6	Figure illustrating impact of new node joins on replica groups of size 3 under DHash	78
5.7	Overhead bandwidth of Chord systems with various placement functions plotted against maintenance frequency. Error bars show 95% confidence intervals. These results are obtained with $N=300$, $R_{MIN}=6$ and $R_{MAX} = 14$	79

5.8	Number of nodes communicated with by core maintenance against number of replicas. Error bars show 95% confidence intervals, lines indicate a linear fit to the data. The x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.95$ column in Table 4.4.1.	80
5.9	Proportion of 5000 fetch requests for a single item answered by each replica holder for that item.	81
5.10	Time taken to locate all replica holders of an item, for various replication factors. These values are for a system size of 200 Nodes. Error bars indicate 95% confidence interval. The x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.95$ column in Table 4.4.1.	82
5.11	When using symmetric placement with a replication factor of four, we find the distance between a node and its nearest replica (marked x) is the same as the distance between that node's final and penultimate finger pointer, and their nearest replica.	83
6.1	Proportion of DBServer requests requiring database Read, Write, or Search.	86
6.2	Information Service system architecture. Local IS services run at each station, queries they cannot satisfy are passed to the central DBServer then stored in the ring.	88
6.3	UML Class Diagram showing data storage methods implemented by an Information Service Node.	89
6.4	Fetch Performance Against System Load for Centralised and Peer to Peer Systems. Values are mean average for 2000 requests, error bars show one standard deviation.	96
6.5	Fetch Performance By Cache Status	97
6.6	Relationship between system size and <i>Chord-only</i> fetch times on a low latency network.	100
6.7	Fetch Time Histograms for various system sizes	101
6.8	Deployment Scenarios	102
6.9	Fetch Performance By Deployment Scenario	102
6.10	Evolution of system state following a failure.	104
6.11	Effect of node failure on bandwidth usage	105
6.12	Effect of data storage size on bandwidth usage	106
6.13	Effect of system load on bandwidth usage	107
7.1	Reliability of a 500 node system, in which each node fails with probability 0.5; In the Erasure Coded system, data is stored as a number of erasure coded fragments, 5 of which are needed to reconstruct the item. The system is considered to have failed if any item cannot be reconstructed.	113

LIST OF TABLES

2.1	Index to symbols used in the description of CAN.	17
2.2	Index to symbols used in the description of Kademlia.	19
2.3	Index to symbols used in the description of Chord.	20
4.1	Index to symbols used in section 4.2	53
4.2	Index to symbols used in section 4.4	61
4.3	Values of R_{MAX} required so that R nodes are found in a keyspace interval of length $R_{MAX} \frac{K}{N}$ with probability π , for $\pi = (0.95, 0.99, 0.9999)$ and $R \in [2, 15]$	63
4.4	Values of R_{MIN} that may be placed in an N node system using Finger Allocation with a 99.9% confidence level of recovering from allocation collisions .	64
4.5	Index to symbols used in section 4.5	64
6.1	Locations and specifications of machines used in test deployment	94

1. INTRODUCTION AND MOTIVATION

1.1 Introduction

This thesis examines how we might create reliable peer to peer grid middleware. We begin, in the first two chapters, by defining the terms Peer to Peer, and Grid Middleware. In these chapters we also highlight areas in which grid middleware can be improved using ideas, such as the distributed hash table, taken from recent peer to peer research.

In the third chapter, we concentrate specifically on distributed hash tables, and examine various methods which allow them to be used as the basis for building reliable distributed data stores for grid computing. In chapter three, we introduce one of the original contributions of this thesis — a substantially new replication algorithm, which we call dynamic replication.

The reliable storage techniques introduced in the third chapter involve a number of configuration parameters. In the fourth chapter we present methods that can be used to set these parameters to achieve a desired level of reliability. This chapter is largely original work, and provides new insights into the factors affecting the reliability of a peer to peer data store.

In the fifth chapter, we use simulation to compare the performance of the reliable storage algorithms we introduce and discuss in the previous two chapters. This quantitatively shows how the choice of storage algorithm affects the read latency, write latency, and communication costs of our storage systems.

Finally, the sixth chapter shows how the ideas we have developed could be applied to a real world system. In this chapter we present a case-study in which we build a prototype implementation of a reliable storage system based on a distributed hash table. Benchmarks and performance tests run on this system show that our prototype is effective in improving both reliability and performance.

It is the first of the definitions of grid middleware that we will be chiefly concerned with in this chapter. We will begin by defining what we mean by grid computing, and explain why there is a need for it. We will show the role middleware plays in creating grid computing systems. We present a case study of a grid computing system, and identify the areas in which the performance of this system might be seen as less than satisfactory. Finally, we will cite examples from several sources which suggest that the application of peer to peer techniques would improve performance.

1.2 What is a Grid Computing System?

Grid computing is a paradigm for distributed computing that aims to simplify access to distributed resources. There is no single widely accepted definition of the term, and so two definitions are presented here. The first is from Dr Ian Foster, who is often credited with popularising the concept of grid computing, and who has edited a popular book on the subject [5]. He states that a grid is a system that

“coordinates resources that are not subject to centralised control, using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of service.” [6]

To provide an industrial perspective, the second definition is taken from an IBM marketing document published in 2002. Their definition is similar, but perhaps more specific, and less concise. It states that grid computing is

“the ability, using a set of open standards and protocols, to gain access to applications and data, processing power, storage capacity and a vast array of other computing resources over the Internet. A Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of resources distributed across multiple administrative domains based on their availability, capability, performance, cost, and users’ quality-of-service requirements.”

These two definitions capture the concepts that underpin grid computing — the sharing of computing resources between institutions, and the allocation of those resources to consumers in an automated fashion. These definitions also stress the importance of open, standardised protocols. At the time of writing, a single set of accepted protocols for grid computing has yet to emerge, though the Global Grid Forum is actively engaged in establishing suitable standards.

Should such standards emerge, grid computing could in theory be used to connect and manage computing resources globally, creating “*The Grid*” — a single entity from which computing power could be provided on-demand, much as electrical power is currently provided by the National Grid. At present, The Grid remains a dream; current grid computing systems are usually not inter-operable.

It is worth noting that although the term ‘grid’ is relatively recent, the underlying aims are not. The term *Metacomputing*, coined in 1988 by Larry Smarr, describes essentially the same ideas for coordinating access to distributed resources [7]. The idea of computing power as a utility dates back even further, and was considered as long ago as 1964 by Alan Greenberger [8]. Recent advances in both computing power and network capacity have made these ideas more practical, and so more popular. This resurgence has also been driven by new applications for such systems, which we will examine in the next section.

1.3 Why Do we Need Grid Computing Systems?

The recent surge in the popularity of the Grid Computing concept has been driven by the needs of large scientific collaborations involving the analysis of large quantities of data at different institutions. Applications have been diverse, and include climate modelling [9], aircraft engine design [10], and modelling the causes of heart disease [11]. Industrial users are also interested in grids, as the model of computing power as a utility offers the possibility of increased efficiency. One of the biggest users of grid computing so far has been the High Energy Physics community.

High Energy Physics (HEP) has a particularly strong need for grid computing. Modern particle accelerators, such as the Tevatron in Chicago, and the LHC in Geneva [12], house detectors that produce enormous volumes of information. The CDF [13] and D0 [14] experiments at the Tevatron, for instance are producing data at a rate of approximately one petabyte¹ a year [15]. It is anticipated that experiments at the LHC will generate far more data — approximately fifteen petabytes of data a year [16]. These experiments are international collaborations — for instance the CDF and D0 collaborations incorporate 80 institutions, on four continents.

No single institution has the computing resources to process all of this data. This has created a need to combine computing resources at multiple institutions. Each institution can allow others to use their local resources when they are idle, in return for reciprocal agreements from other institutions. This allows physicists access to larger quantities of computing power and storage than would otherwise be available, ensuring that local resources are always efficiently utilised, but allowing extra resources to be used when demand dictates. This approach may be politically popular, as institutions prefer to retain ownership of their computing resources, rather than entrust them to a single central laboratory.

1.4 Grid Middleware

In order to co-ordinate the resources in a grid, a number of *middleware* components are used. Middleware is software that serves as an intermediary between the raw resources on the grid and the applications that seek to use those resources. Some of the services that are typically provided by Grid computing middleware are listed below:

Information Services: Information services keep track of the resources available, allowing other applications and services to find these resources; They may also monitor the state of these resources, and aggregate this state information.

Security: Security middleware authenticates users, and determines their access rights to resources.

Job Management: Resource management middleware finds appropriate resources for a task, for instance selecting a free CPU to run a job on. It may make use of the

¹ A Petabyte is 2^{50} bytes, or 1,048,576 Gigabytes

monitoring and discovery services provided by the information service to perform this task. The resource management framework may then send the job to the execution site.

Storage Management: Storage middleware manages the storage resources on the grid, providing a *replica catalogue* listing which resources have copies of a given file. There may also be a *metadata catalogue*, which stores information about the files stored in the system. A reliable transfer mechanism for moving and creating replicas of files may also be provided.

The de-facto standard set of open source grid middleware is the Globus Toolkit [17], which offers software that provides most of these services. The Globus tools are widely used, and form the basis for a number of important Grid projects, including the LHC Computing Grid. Other toolkits exist, such as BOINC [18], Sun's Grid Engine [19], and the Open Middleware Infrastructure Institute's Grid software distribution [20] but are less widely used in HEP. The Globus Alliance, who produce the toolkit, are also heavily involved in the standardisation process at the Global Grid Forum, and so it seems likely that future standards will be based around the toolkit components. We list the major components of the Globus toolkit below, providing very brief explanations of their function.

MDS: MDS is Globus's implementation of an information system. It provides an integrated **M**onitoring and **D**iscovery **S**ervice, which aggregates information on monitored resources.

GSI: GSI is Globus's **G**rid **S**ecurity **I**nfrastructure. It makes use of X.509 certificates [21] for authentication.

GridFTP: GridFTP is the file transfer protocol used by Globus's storage middleware. It provides high performance secure data transfers, using GSI for security, and techniques such as multiple TCP streams to increase performance.

RFT: RFT is Globus's implementation of **R**eliable **F**ile **T**ransfer middleware. It monitors and manages GridFTP transfers between grid storage sites, allowing these transfers to be recovered should the transfer be interrupted.

RLS: RLS is Globus's implementation of a **R**eplica **L**ocation **S**ervice, mapping filenames to physical file locations.

GRAM: Globus **R**esource **A**llocation **M**anager forms part of a job management framework, creating a uniform interface for the remote submission and monitoring of jobs to a variety of computer farms.

These middleware components provide only the basic building blocks for grid computing systems, and various grid projects have built higher level services upon them. One such project, in which the author has been involved, is the SAM-Grid system [22, 23, 24]. The author has been responsible for designing and implementing services including a distributed monitoring framework [25], and a distributed stress and unit testing framework [26] for SAM-Grid. The SAM-Grid system will be used as a case study of a Grid computing system.

1.5 Case Study: The SAM-Grid system

The SAM-Grid system is used by physicists to analyse the data from various experiments at the Fermi national accelerator laboratory (Fermilab) in Batavia, Illinois. These experiments are detectors attached to the *Tevatron* particle accelerator, which accelerates protons and antiprotons to enormous energies. The protons and antiprotons collide within the detectors, producing a shower of secondary particles. Many of these secondary particles will interact with the detector, producing a *signature* that can be recorded and analysed to determine the properties of these secondary particles.

Collision *events* take place within the Tevatron at a rate of 7.6MHz². The CDF detector records approximately 250kB of data about each event — meaning it is not feasible to store data on every collision. Instead, very fast hardware *triggers* select only the most interesting events. In the case of CDF, these triggers accept approximately one event in 25,000. Selected events are then scrutinised by a slower, software based trigger which accepts approximately 10% of the incoming events [27]. The final data rate is approximately 10Mbs⁻¹. All these events are written to tape, using the *Enstore* robotic tape library.

The purpose of SAM-Grid is to provide access to this vast library of data to the physicists who are collaborating on these experiments. SAM-Grid keeps track of all files produced by the detector, as well as the files produced by further analysis of this data. All data is categorised by metadata, and physicists can select files of interest by specifying appropriate metadata; the SAM component of SAM-Grid is an abbreviation of *Sequential Access to data via Metadata*.

The original 1997 incarnation of SAM-Grid (simply called SAM) provides a storage management system. SAM keeps track of which files are stored in disk caches, automatically copying requested files from tape, freeing space by evicting less frequently used data if necessary. The system can copy data between disk caches, or from tape storage to disk. The SAM infrastructure was extended in 2002 to include a **J**ob Management Infrastructure, and an **I**nformation and **M**onitoring System (JIM) based on Globus tools.

The original SAM system predates the Globus tools, and consists of custom written code in both C++ and Python. Services communicate using CORBA (Common Object Request Broker Architecture [28]), and an Oracle database hosted at Fermilab is used to store all metadata. We will now give a brief overview of the main architectural components of the SAM-Grid system.

SAM Station The SAM Station software manages a group of storage and processing resources at a single site. Analysis projects running at that site request files from the station. The station then chooses a location from which to fetch the file, clears space on local disks if necessary, and transfers (stages) the file to local disks. The file will remain on local disks until the analysis project indicates the file is no longer needed (the file is *consumed*). The station may use a variety of transfer protocols to copy a file,

² The figures presented here for the Tevatron collision rate, the CDF trigger selection rates, and event sizes are taken from the original design specifications [13]. The actual settings have differed over the course of the experiment, though the final data rates remain of the same order of magnitude.

including GridFTP for transfers between stations, and various protocols for copying files from mass storage systems such as Enstore.

Oracle Database The Oracle Database stores all information and state for the entire SAM system. This includes both global state information, such as the metadata on each file, and all local state, such as which files a given station is currently storing. Stations do not store any of this information locally, and so the database is under heavy demand.

SAM DBServer Oracle licensing conditions limit the number of connections the database can accept simultaneously, and so stations do not connect to the database directly, but instead route their queries through a DBServer. The DBServer receives requests from the SAM stations in the form of CORBA method calls. The DBServer then communicates with the Oracle database to update or request the appropriate information. The response from the Oracle database is marshalled into a CORBA structure, and returned to the station.

CORBA Nameservice The CORBA nameservice functions as the discovery service for SAM-Grid. Stations and DBServers are registered in the name service, allowing other stations to discover their IP addresses.

The JIM component of SAM-Grid uses the globus GRAM and MDS middleware to provide a job management component for analysis jobs using SAM. Jobs are allocated to processors using the Condor-G [29] system, which matches jobs to available resources using 'Class-Ads'. Class-Ads can describe the capabilities of resources, or the resources required for the job. Once a matchmaker has selected a resource for a job to run on, the GRAM system is used to submit the job, and it is monitored using MDS.

This SAM-Grid architecture has proven very successful over the last nine years, enabling distributed computing and analysis which has provided a basis for over 50 scientific papers. However, although performance has been more than adequate, over the long lifetime of the system it has become apparent that the centralised nature of the middleware is causing problems — particularly the reliance on a single database acting as a replica catalogue, a metadata catalogue, and a store for all other state information. We will now give concrete examples of how centralised design has reduced performance, creating a load bottleneck and a single point of failure.

1.5.1 Central Systems in SAM-Grid As A Load Bottleneck

The reliance on the database means that any increase in database load results in a decrease in performance of the entire SAM-Grid system. The SAM-Grid database is equipped with multiple processors and fast I/O hardware. Despite this, periods of peak load cause it to become unresponsive. This effect is illustrated during the daily database backups. Database backups are an essential part of maintaining a reliable database service, but place an increased load on the system being backed up. During these backups, response times increase dramatically.

The effect is illustrated in figure 1.1. This plot shows the time taken to execute a complex metadata search sampled at regular intervals throughout the day. Normal execution time is around one minute. Between 2:30 and 9:30 a.m., the database is backed up to tape, which involves a lot of I/O activity. During this period, execution time rises by up to a factor of ten.

Similar effects may also be observed when a large number of analysis jobs are submitted to an idle computing farm. These jobs then start at approximately the same time, and the database load increases as a result. Response times rise to several minutes per query, and jobs may fail as communication timeouts occur [30].

The degraded level of performance during periods of peak load makes interactive use of SAM-Grid unwieldy, and illustrates how the centralised database creates a load bottleneck for the SAM-Grid system.

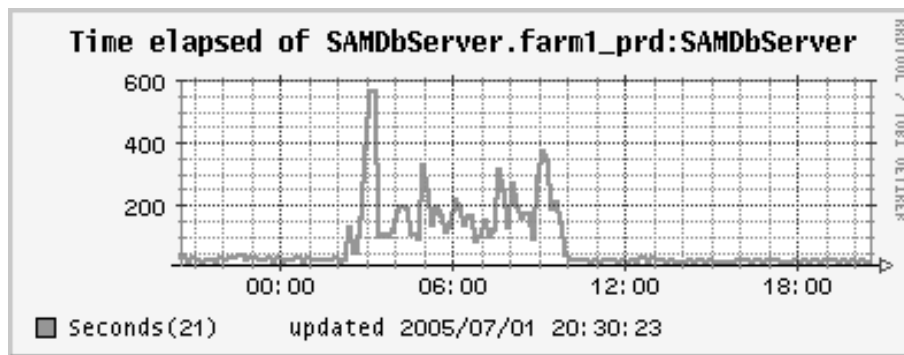


Fig. 1.1: Time taken by SAM-Grid to run a complex metadata search (seconds) plotted against time of day. Database backups are running between 2:30 and 9:30 a.m.

1.5.2 Central Systems in SAM-Grid As A Single Point Of Failure

SAM-Grid relies on many services. The database, the DBServers, and the CORBA naming service are relied upon by all components in the system. While services which rely on the CORBA naming service and DBServers can easily be configured to failover to backups, no such possibility currently exists for the database, due to the much larger quantity of state information it stores.

The SAM database runs on enterprise class SUN hardware. Technicians are automatically alerted to any failures, and respond rapidly and attempt to limit any resulting downtime. The reliability of the database was assessed using data from maintenance email list between the 1st of February and 31st of August, 2005. During this interval, the database was unavailable on 14 separate occasions.

Of these 14 periods of downtime, 6 were due to operating system or database security patches and upgrades, and were scheduled in advance, allowing SAM Users to prepare. The remainder of the downtime was unscheduled — caused by a failed memory module, which led to a period of instability until it was diagnosed and replaced.

The total period during which the database was unavailable over the seven month period

was approximately 12 hours. This corresponds to a reliability of 99.7% — a level of reliability that would generally be considered excellent. However, the impact of these periods of downtime on SAM-Grid operations is greater than simply the downtime experienced while the database is unavailable.

Scheduled downtimes require that the CPU farms used to run analysis jobs are *drained*. This means no new jobs are submitted for 12 hours prior to the downtime, so that no jobs are running when the database is down. Toward the end of the draining process, many cpus are idle. This results in a great deal of potential compute time being wasted. Once the database is back online, jobs may be submitted again.

Unscheduled downtime wastes CPU time, and requires user intervention in identifying and resubmitting affected analysis projects. Analysis projects are generally not robust to database failure, and database downtime will cause them to crash. When an analysis job crashes in this manner, it will in most cases need to be restarted from the beginning. Globally, the analysis farms comprise over seven thousand CPUs, and analysis jobs typically run for 8 hours. Typically, each unscheduled database downtime causes thousands of CPU hours of computing to be wasted as jobs fail. These jobs must then be identified and resubmitted by physicists, causing further delays. Finally, users must wait for some time to be given new slots on the CPU farm, and jobs resubmitted after such a crash will need to queue for the resources again.

After each period of unavailability, the DBServers must be restarted manually, and this also prolongs the length of time during which information is unavailable.

Such delays illustrate how the reliance on the database reduces the effectiveness of the SAM-Grid system. Increasing the robustness of the SAM software and analysis jobs, or deploying a more reliable information system, would help avert this loss of effectiveness.

1.6 The Need for Peer To Peer Grid Middleware

The problems associated with using centralised systems are not unique to SAM-Grid. Other large grid computing systems are suffering from similar problems: A particular example is the European Data Grid, a system for examining the data from the LHC particle accelerator. This accelerator will produce more data than the Tevatron, and so the Grid system is larger than SAM-Grid. The European Data Grid (EDG) uses centralised middleware, often built using Globus Tools. However, performance of prototypes has often been poor — for instance, it takes 5 seconds to match a job to a free resource, meaning a maximum of 17,280 jobs can be submitted a day³. Since the EDG has 10,000 CPUs, this is barely adequate, and suggests the system will not scale to larger grids. Similar bottlenecks arise in the EDG storage system, where the services involved can become unresponsive or fail when heavily loaded [31].

To avoid such issues we would prefer that no single machine be responsible for the operation of an entire grid. This would mean that the failure of an individual machine would

³ These figures were obtained from the 2005 ARDA public document: “WMS Performance Test Plan”, by Hurng-Chun Lee.

have less of an impact on the grid. Additionally, if the responsibility for running the system can be divided equally among a number of machines, then no single machine should become a load bottleneck. Grid middleware which had these properties could be classified as peer to peer middleware, as it would inevitably consist of a group of machines (peers) with equal responsibility.

We are not alone in reaching the conclusion that peer to peer techniques could be employed to improving grid computing systems. Ian Foster and Adriana Iamnitchi have published a paper on the subject, entitled “On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing” [32]. In this paper, they point out synergies between peer to peer and grid computing which they believe will inevitably lead to a convergence between the two areas. In their 2003 paper, ‘Scooped Again’, Ledlie *et al.* point out that the peer to peer and Grid Computing communities have failed to work together so far, despite the fact that “The Grid infrastructure is a great customer waiting for future P2P product”, and that this is harming both communities [33]. In this thesis, we seek to address this issue by exploring how we can use work from the peer to peer community to create reliable and scalable grid middleware.

1.7 Summary

We have explained what grid computing systems are, why they are needed, and how middleware is used to create them. We have explained, using a specific example, that centralised middleware can lead to problems in large grid computing systems, and we have suggested that peer to peer systems may provide a solution to these problems. In the next chapter, we examine peer to peer systems in more detail.

2. PEER TO PEER COMPUTING

2.1 Introduction

In this chapter, we will introduce the concept of peer to peer computing. We begin in section 2.2 by attempting to define this sometimes misunderstood term. We give reasons why the scalability and reliability properties we desire often emerge from such systems. In section 2.3, we categorise and describe common peer to peer architectures, giving particular attention to Distributed Hash Tables in section 2.3.3. Finally, we summarise relevant areas of peer to peer security (Section 2.4) and robust networking (Section 2.5).

2.2 Definition

The term *peer to peer* is used to describe a class of system architectures for distributed computing. The term is perhaps overused, and has certainly been applied to a wide range of very different systems. There is currently no general agreement about precisely what defines a peer to peer system. A popular definition is that of Shirky:

“Peer-to-peer is a class of applications that take advantage of resources storage, cycles, content, human presence available at the edges of the Internet.” [37]

By the resources on the ‘edges of the internet’, Shirky means the resources of client systems — as opposed to servers. While this definition encompasses all the systems we would wish it to, Shirky’s definition is perhaps too general; it also admits many systems we would prefer it did not. SAM-Grid, for example, could certainly be called a peer to peer system under such a definition, despite its reliance on centralised services.

Looking elsewhere, Stoica *et al.* give a definition that prohibits any kind of centralised control. This expresses the common opinion that, in a true peer to peer architecture, all nodes must play equal roles in providing the service, both issuing and servicing requests.

“Peer to peer systems and applications are distributed systems without any centralised control or hierarchical organisation, in which each node runs software of equivalent functionality.” [38]

While such definitions are useful, in reality few complete systems are able to meet such requirements. Almost all peer to peer architectures use some element of centralisation, for

instance to discover other nodes, to lookup information, or to provide security. We feel such systems are correctly described as peer to peer architectures, despite the fact that they make limited use of centralised services.

Schollmeier [39] has dealt with this problem in his definition of peer to peer by dividing peer to peer systems into two categories. Those that use no centralised services, he describes as *pure* peer to peer networks. Those which incorporate some level of centralisation, he describes as *hybrid* systems.

This correctly acknowledges that centralised services may play a part in a peer to peer system. However, to classify systems which are entirely reliant on centralised services as ‘peer to peer’ would seem unhelpful. We therefore offer the following definition of peer to peer computing, which seeks to offer a compromise between Shirky’s broad definition, and Stoica’s more narrow decentralised one.

Definition 1:

A peer to peer system or architecture is a distributed system in which a number of peers cooperate to provide a service. A majority of these peers run software of equivalent functionality, and this majority is capable of offering some level of service independently of any others.

2.2.1 Properties of Peer to Peer systems

Because peer to peer systems are distributed, they must be deployed in a networked environment. Such environments are often unreliable, and both node and communication failures are to be expected. This causes the set of nodes actively participating in the system to change as nodes fail, and leave the system, possibly rejoining later. We call this change in participation *churn*.

Well engineered peer to peer systems should be able to cope with such an environment, and so certain properties are likely to emerge:

Fault Tolerance: Schollmeier [39] notes that an emergent property of most peer to peer systems, and particularly of pure peer to peer systems, is that since all nodes play an equivalent role in the system, the loss of any single node is rarely catastrophic. His definition of a pure peer to per system requires that “any single, arbitrary chosen Terminal Entity can be removed from the network without having the network suffering any loss of network service.”

Because there is a functional equivalence between nodes, replacing a failed node with another is simpler than under client server systems. This, and the inevitability of node churn, means successful peer to peer architectures tolerate the failure of peers well.

Scalability: Node churn makes it difficult to keep track of which nodes are currently participating in the system. This causes the prudent system designers to avoid attempting to maintain *global knowledge* — that is knowledge of all active participants — on any

given node. Instead, wherever possible, *local knowledge* is maintained. Each node typically communicates with only a subset of the nodes in the system. Global connectivity is maintained by allowing requests to be routed through more than one node. This property allows, in some cases, for the system to scale to a large number of participant nodes, and consequently to a large quantity of shared resources.

Automatic Configuration: Changes in the set of participant nodes may occur at any time. Because the majority of nodes are involved in providing the service, the system must respond to these changes as they occur.

This means that a successful peer to peer system must be able to reconfigure itself without user intervention, in order to restore service following a node failure.

Because there is no central authority to decide upon new configurations, nodes must co-operate to reorganise themselves consistently. This property is desirable as it reduces the scope for administrator error, and increases failure tolerance.

While these properties are not an inevitable consequence of a peer to peer architecture, they are achieved by a number of different peer to peer designs. In the next section, we look at these designs in more detail.

2.3 A Taxonomy of Peer to Peer Systems

Peer to peer systems are not a new phenomenon. System designs including the Border Gateway Protocol for internet routing¹ (standardised in 1989) [40], and the Usenet news network (designed in 1979) [41], would fall under our definition of peer to peer architectures. As with Grid computing, recent increases in the processing power, storage, and network bandwidth available on the average home computer means that the benefits of combining these resources have become increasingly attractive. From the late 90s onward, a large number of peer to peer systems have gained widespread popularity.

The architectures of these systems vary, but can be divided into three broad categories: centralised, structured and unstructured systems. These categories are orthogonal to our previous categorisation of pure and hybrid peer to peer systems. A hybrid system might fall into any of these categories, a pure peer to peer system must either be structured or unstructured.

We will now define each of these categories, providing an illustrative example of each.

2.3.1 Centralised Peer To Peer Systems

From the definition we gave in section 2.2, it should be clear that despite appearances, the term ‘centralised peer to peer system’ is not an oxymoron. Centralised peer to peer systems

¹ Although we have generally adopted British spellings elsewhere in this thesis, the American spelling of ‘routeing’ will be used, for consistency with other literature in this area.

use a central index server, or index servers, to store information about all peers. Peers consult the central server to find resources, then communicate directly with the relevant peers to access those resources.

Because of a reliance on a central server, centralised systems do not inherit the failure-tolerance that a pure peer to peer system might. The loss of the index server will mean that resources cannot be found, although previously discovered resources may not be affected. The scalability of a centralised peer to peer system may also be limited by the load placed on the central server by large numbers of peers.

A notorious example of this architecture is Napster, a file-sharing application developed by Shawn Fanning in 1999. Napster used a central server to index the music collections of involved peers. A user could search this index to find a given song, then connect to a peer to download it. This system was immensely popular, attracting over 65 million registered users [42], and helped to draw attention to the power that peer to peer systems can harness. The original system was shut down in 2002, following litigation from the Recording Industry Association of America.

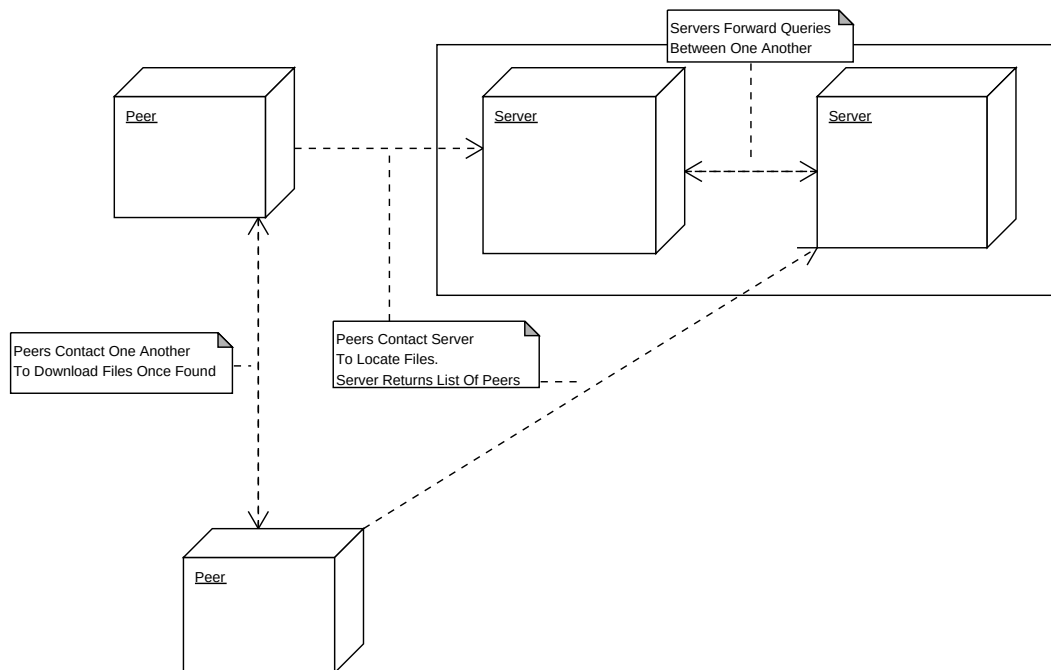


Fig. 2.1: Overview Of The Napster Architecture.

Napster users are divided among a number of central servers, each of which maintains a local database of the files stored on those peers connected to them. The server initially answers queries by searching its own database, but may also forward the query to other servers and collate the results. A large number of central servers were needed; in 2002, Napster was using approximately 160 separate index servers [43].

That the system did scale to such a large number of users is impressive, though it was at the expense of query performance. New users would also take some time to join the system, as the servers are often slow to index their list of files. Additionally, the number of additional users that may be served by adding an additional server drops as the number of

servers increases [44]. This suggests that the system would eventually encounter scalability limits.

Napster illustrates both the strengths and weaknesses of a centralised peer to peer architecture. The central index servers are able to rapidly search a large resource index. However, the centralised systems must be powerful and reliable, and so are expensive, and scalability problems may arise. The failure of a central service causes the failure of the entire system, as illustrated by the ease with which the RIAA was able to shut down the Napster system by closing down the central servers. Unstructured systems, which we will discuss in the next section, avoid these issues by decentralising the resource index.

2.3.2 Unstructured Peer To Peer Systems

An unstructured peer to peer system is one in which there are no rules governing what resources a given node can be responsible for. In an unstructured peer to peer system, the only way to locate a resource is to send a query to all nodes in the system — a technique called query flooding.

The Gnutella system provides an excellent example of an unstructured peer to peer system. Gnutella is a fully decentralised peer to peer file-sharing system, and was first released in 2000. A broad architectural overview of the original version of Gnutella is given in figure 2.2. A peer joins the system by establishing a connection to a bootstrap node, the address of which is learned using an out of band mechanism — for instance from a web page listing. The bootstrap node then provides the addresses of a small subset of the other nodes in the system. The new node will collect addresses from a number of these nodes, before choosing a number of them as its neighbours. It establishes TCP connections with these neighbours. The other addresses it has collected are stored, and are used to form new connections should any of the originally chosen set of peers fail. The neighbour links form a connected graph, in which each node attempts to maintain a user configurable degree.

When a user wishes to locate a file, the node he or she controls broadcasts a query to each of its neighbours. The query may be complex, specifying complex constraints on any metadata the file may have. The neighbours check if they have any data that matches the query, and if so they inform the querying node of this. They may then forward the query to each of their neighbours, which repeat the process. This forwarding process is terminated once the query has been forwarded a specific number of times. To avoid routing loops, query identifiers are used to avoid forwarding the same query more than once. The number of times a query is allowed to be forwarded is called the query's *time to live*, or TTL. Choosing an appropriate TTL value is difficult: a low TTL value may mean not all peers are searched, whereas high TTL values will cause each query to require a great deal of network communication.

Measurement studies have found that the neighbour graph formed by Gnutella hosts exhibits power law properties [47, 48]; a small number of nodes have a much higher degree than others, and these disseminate queries rapidly through the network.

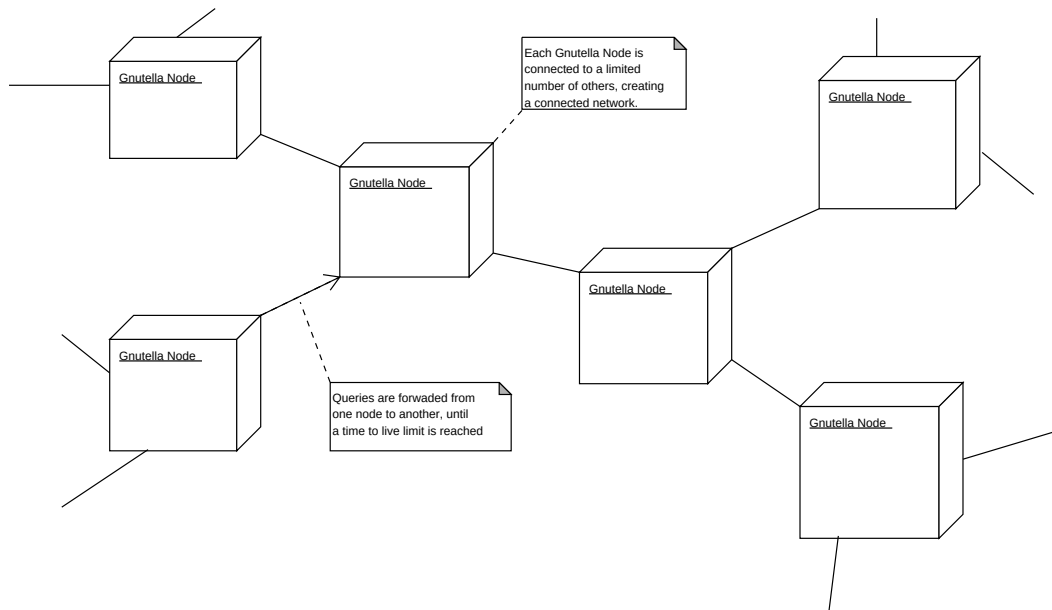


Fig. 2.2: Overview Of The Gnutella Architecture.

This property gives the network excellent resilience to churn, allowing it to remain connected despite node failures [49].

The Gnutella architecture suffers from scalability problems as the number of hosts increases. Since all queries may eventually reach all nodes, and since almost all nodes issue queries, the traffic reaching a node grows rapidly with the number of participants. Eventually, participants with lower bandwidth connections are overwhelmed by query traffic [50]. More recently, new strategies have been employed to improve the scalability of Gnutella [51]; we present them here because they are broadly applicable optimisations that might benefit any unstructured peer to peer network:

Expanding Ring Search: A request may be initially transmitted with a low TTL, and re-transmitted with a higher TTL value if insufficient results are obtained. This has the effect of avoiding searching the entire network when the data is available nearby. As the popularity of files in Gnutella follows a Zipf distribution [52], the majority of requests are for a small number of popular files likely to be found within a small number of hops [50]; this reduces network usage significantly.

Two Tiered Topology: Recent versions of Gnutella use a two-tiered topology, consisting of *ultrapeers* and *leaf nodes*. Ultrapeers act like the central servers in Napster, caching the list of files available on leaf nodes, returning results on behalf of their leaf nodes.

The ultrapeers forward queries between themselves, whereas the leaf nodes do not receive or forward queries directly, but only store and transmit files. Ultrapeers are elected from nodes with high bandwidth, unfirewalled connections. Such a node is said to be ultrapeer capable. When a ultrapeer capable node joins the Gnutella network, it will be instructed to become either a ultrapeer or a leaf node by the existing ultrapeers [53]. A similar approach is believed to be used by the Kazaa file sharing system, but the details of this closed-source software are not well known.

Another optimisation technique for searching unstructured peer to peer networks is to use a *walker*. Instead of flooding the network with queries, a walker explores the network by forwarding the query to a randomly selected node at each step. This has the effect of reducing the number of query messages sent, but will increase the time taken to find an item. One or more walkers may be used in parallel to reduce the query latency. Simulations show that in most scenarios, this method finds content using significantly fewer messages than expanding ring search [54, 48].

The delay involved in searching the network can be improved by having nodes choose neighbours that are physically close by — that is, those which have low network latencies [55]. This approach leads to faster lookup times.

Further improvements in query performance can be achieved by replicating objects on more than one host, such that one has to query as few hosts as possible to find a given item. The replica placement policy affects query performance, and Cohen et. al give an optimal lookup time replica placement algorithm that places replicas based, in part, on their popularity [48].

In conclusion, unstructured peer to peer systems offer the ability to search for objects meeting complex criteria in a large scale environment. Searching large networks requires a great deal of network communication, and this may limit scalability, though various techniques can be used to reduce this cost of finding a given item.

The major weakness of unstructured networks is that discovering all copies of any given data item is not possible without searching every node in the system — an operation that is likely to be very slow. It is also not possible to verify whether or not a given data item is stored in the system without resorting to the same technique. Though such operations are rare in the file-sharing environments that Napster and Gnutella are designed for, they are essential if we wish to store *mutable* data in our network. Mutable data is data which can be changed or updated after being stored in the network. In order to consistently update mutable data, we need to locate all copies of that data currently in the system, and either replace them with the new copy, or delete them. Such operations are likely to be inefficient in an unstructured network. For a solution to these problems, we must turn to structured peer to peer systems, which we will now describe.

2.3.3 Structured Peer To Peer Systems

In contrast to unstructured peer to peer systems such as Gnutella, in a structured peer to peer system each node has a responsibility for specific objects. Consequently, queries need not be flooded through the overlay network, but can be routed directly to the responsible peer.

Examples of structured peer to peer systems include distributed trees, [56, 57], and layered random graphs, [58]. By far the most popular structured system, however, is the *distributed hash table*. Distributed hash tables provide a solution to the lookup problem in distributed systems. Given the name of a data item stored in the system, we can locate the node on which that data item should be stored.

In a Distributed Hash Table (DHT), each host is given an *ID* chosen from some *keyspace*, over which a *distance metric* is defined. Data objects are also allocated keys from this same keyspace to form (key, object) pairs. A host is responsible for those objects with keys closest to it in the keyspace, and must route requests for other keys to those nodes responsible for them. In order to do this, each host is aware of a small subset of other hosts, which it holds in its local *routing table*. This routing table is usually not sufficient to immediately locate the specific node responsible for any given item, and so nodes must co-operate to locate the node responsible for a given key by forwarding requests between one another, reducing the distance between the query node and the key at each forwarding step. A query may be forwarded directly from one node to the next, a process called *recursive* routing. Alternatively, each node may return the address of the next node in the query path to the original query node, which then sends the query on itself — a process called *iterative* routing. Recursive routing will generally lead to shorter lookup times, whereas iterative routing allows a query's progress to be monitored more closely.

DHTs usually also allow for nodes in the DHT to join and leave at will, and provide a Maintenance Algorithm to keep routing tables up to date.

The structure of the routing table influences the lookup performance of the system, and the quantity of network communication required to keep it up to date. A large routing table will allow a query to quickly converge upon the correct host, but will require more communication to maintain.

Although DHTs allow data to be located by name, they do not support complex search queries. We cannot, for instance, locate a node responsible for data whose size is greater than say, 10MB, and such a query must be broadcast to all nodes, in a manner similar to an unstructured network.

We now present a survey of Distributed Hash Tables to illustrate how different implementations of these concepts can lead to systems with different routing and resilience properties.

Content Addressable Network

Symbol	Definition
d	Dimensionality of the CAN keyspace.
N	The number of nodes in the system.

Tab. 2.1: Index to symbols used in the description of CAN.

CAN, the Content Addressable Network, is a Distributed Hash Table developed at Berkeley [59]. The keyspace in CAN is a d -dimensional cartesian space on a unit d -torus. Each key maps to a point in this keyspace. The keyspace is partitioned between the participant nodes, so that each node owns a d -dimensional *zone* within the space, and is responsible for storing any objects mapped into that zone. Each node maintains a routing table with the address and zone boundaries of all owners of neighbouring zones. Figure 2.3 gives an example of a two dimensional space partitioned between CAN nodes.

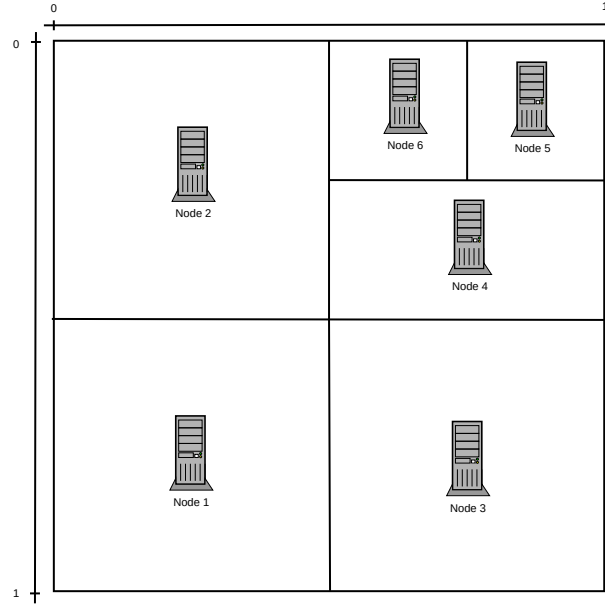


Fig. 2.3: An example two dimensional CAN keyspace partitioned between 6 nodes.

To find the node responsible for a given key, a request is passed from a node to a neighbour which is closer to the requested point, until the request reaches the node whose zone contains that point. For instance, in figure 2.3 a request for a key mapping to the co-ordinates (0.9, 0.1) might be routed from node 1 through node 2, to node 6, and then finally to node 5. In a d dimensional system, this routing mechanism leads to per-node routing tables of size $O(d)$ and average path lengths of $O(dN^{\frac{1}{d}})$ hops.

When a new node joins the system, it picks a point in the keyspace at random, and looks up the current owner of that point. That zone is then divided equally between the current owner and the new node. (Key, object) pairs are transferred from the old owner of the zone to the new node.

When nodes leave the system, their departure is detected by periodic ping messages from nodes in neighbouring zones. These nodes then negotiate so that the neighbour currently responsible for the smallest zone gets assigned the space vacated by the departed node. In the case of multiple failures, it may be necessary for nodes to gather additional information about nodes neighbouring the failure region in order to achieve consensus on how the vacated area is divided between survivors. Because this process can cause a node to become responsible for more than one zone, a background defragmentation algorithm is also necessary, details of which are given in [59], but not reproduced here.

Several techniques may be used to improve the performance of CAN, including using caching and replication to balance the load for popular keys (hotspots) between several nodes. More uniform zone partitioning may be achieved by modifying the join algorithm to select a large or overloaded zone to join, rather than simply joining at random. Network latency may also be taken into account when choosing where to join the network, so that communication between neighbours is fast.

Kademlia

Symbol	Definition
k	Number of nodes in a k -bucket.
α	The level of lookup parallelism.

Tab. 2.2: Index to symbols used in the description of Kademlia.

The Kademlia hash table uses 160-bit binary keys for its keyspace. The distance between two identifiers is defined as the exclusive or (XOR) of the two identifiers, expressed as an integer. This means that keys which differ in the first binary digit are interpreted as being very distant, whereas keys which differ only in the last binary digit are interpreted as being adjacent.

Each node has a routing table consisting of up to 160 k -buckets, numbered from 0 to 160. For each k -bucket, numbered i , the node stores the addresses of up to k nodes at a distance in the range $[2^i, 2^{i+1})$ from it. The larger numbered k -buckets will comprise the largest proportion of the system, whereas the smaller numbered buckets are likely to be empty, as they only admit nodes from very small ranges of IDs, so that it is unlikely node with such an ID will be in the system. The k -bucket system is illustrated in figure 2.4.

Kademlia exploits the fact that nodes that have been available for longer are less likely to fail in order to provide reliability. Each k -bucket is sorted by the time each node was last seen, with the most recently seen at the end of the list. If a new node is encountered, and there is room in the appropriate k -bucket, then that node is simply added to the routing table. If the bucket is already full, the least recently seen node is checked for liveness. If the least recently seen node is still in the system, it remains in the routing table and the newly discovered node's address is discarded. If the least recently seen node does not respond, the newly discovered node is inserted, and the old node removed. The routing table thus evolves to contain nodes with long uptimes.

A lookup proceeds by having a node send a lookup request to α nodes in the closest non-empty bucket to target key. These nodes may be chosen to be the α with the lowest network latency. Each of these nodes sends back the k closest nodes that it knows of. The requesting node then sends out another α requests to the k -closest nodes it received in response to the previous query. When a round fails to retrieve any new servers, the query is re-sent to the k closest nodes found so far. When responses are received from these, the lookup terminates. The final k nodes found are the k closest in the system.

The XOR distance metric has two properties that make it useful for a distributed hash table. Firstly, XOR is unidirectional — that is, given a key, and a distance, there is only one key that is that distance from that key. This means that as requests for a given key are forwarded from node to node, they always converge upon the same lookup path, and this allows the object associated with a key to be cached along the lookup path for that key.

The second useful property is that XOR is symmetric, so that the distance from x to y is always the distance from y to x . This means that if node A is in node B 's routing table,

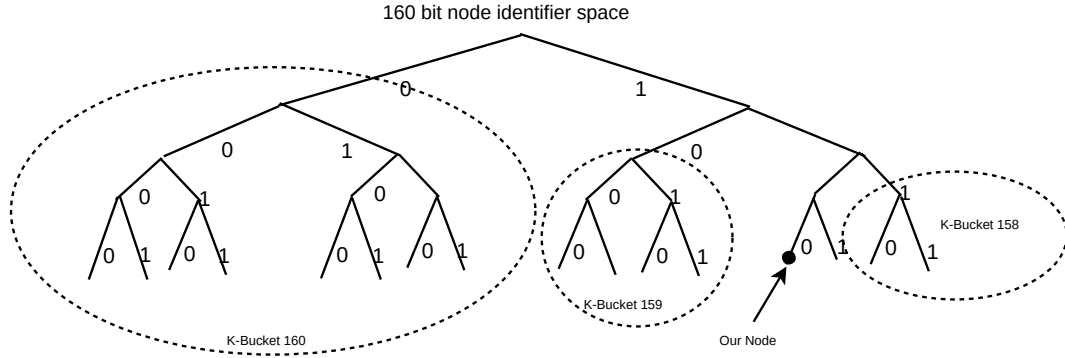


Fig. 2.4: Kademlia binary tree. The black dot indicates the node, and the dashed ovals indicate the areas from which nodes may be selected for the largest three k-buckets

node B can also be used in node A's routing table. This property allows routing traffic to be used to update routing tables. If lookup frequency is sufficient, this alone should maintain up-to-date routing tables. If lookups are infrequent, all nodes in each bucket are contacted hourly if no lookup traffic is seen, and any departed nodes are replaced with new ones.

Caching can be used to provide load balancing. The length of time a key remains in a node's cache is inversely proportional to the distance between the node caching the key, and the key. Thus the closest nodes are most likely to have a cache. The unidirectional nature of lookups means encountering a cached copy of a popular key is likely.

Chord

Symbol	Definition
N	Number of nodes in the system
2^k	The key space size
k	The number of <i>finger</i> nodes in a node's routing table.

Tab. 2.3: Index to symbols used in the description of Chord.

The Chord [38] Distributed Hash Table was developed at MIT, and first published in 2001. The Chord key space is one dimensional, and can be thought of as being arranged in a ring. Chord nodes and data objects are given identifiers between 0 and 2^k , which map to positions on the ring. The distance function between identifiers is the clockwise distance around the ring between them. A node is responsible for data that it is the first clockwise successor of.

Each node's routing table consists of a number of its immediate clockwise neighbours, called its *successors*, its immediate anticlockwise neighbour, called its *predecessor*, and several other nodes at fractional distances around the ring from it, called its *fingers*. The finger table of a given node consists of k fingers, such that for $1 \leq i \leq k$, entry i is the first node on the ring that succeeds that node by 2^{i-1} keys. An example illustrating finger pointers is given in

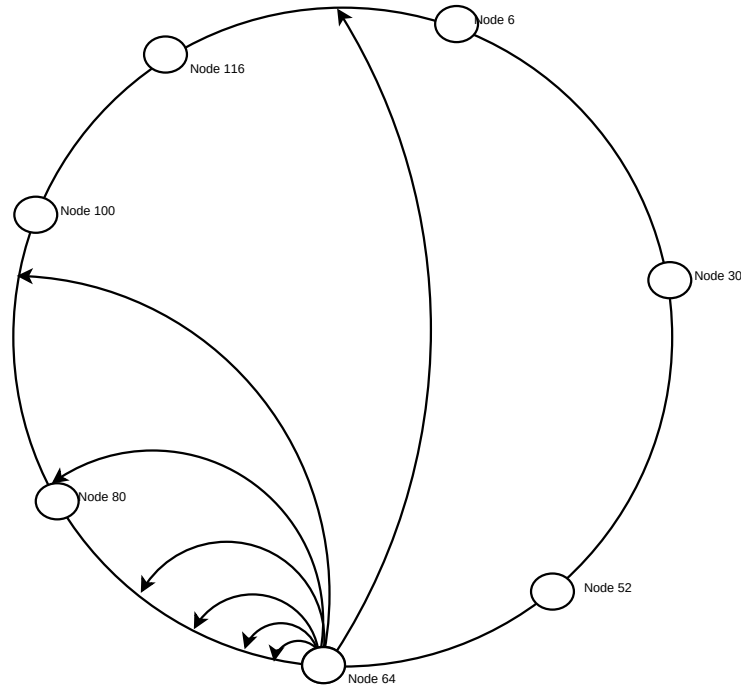


Fig. 2.5: A Chord ring containing 7 nodes, in a keyspace between 0 and 2^7 showing finger table entries for node 64

figure 2.5. In this figure, node 64 would have the node 80 in the first four slots in its finger table, node 100 would fill the next two entries, and the final entry would be node 6.

Requests are routed using a greedy routing algorithm, so that at each step, a request for a key is forwarded to the closest known node to that key, as illustrated in figure 2.6. Because of the structure of the finger table, a key can be located in an N node network in only $\log(N)$ steps [38]. The (recursive) algorithms for finding the successor node of a given key are given as pseudocode in algorithms 1 and 2.

Algorithm 1 $n.find_successor(key)$

```
//n.find_successor(key) - locate immediate successor of key
if  $id \in (n.id, successorList[0].key)$  then
    return  $successorList[0]$ 
else
     $n' = closest\_preceding\_node(key)$ 
    return  $n'.find\_successor(key)$ 
end if
```

In order to join the Chord network, a node chooses a random identifier, and looks up the current successor of that identifier. It then sets that node as its successor. Finally, it starts to run normal Chord maintenance. This maintenance will then eventually result in the node being incorporated into the ring, as other nodes update their routing tables.

Chord maintenance, which restores routing tables following node churn, involves running three algorithms — **fix_fingers**, **stabilize**, and **check_predecessor** — at regular intervals.

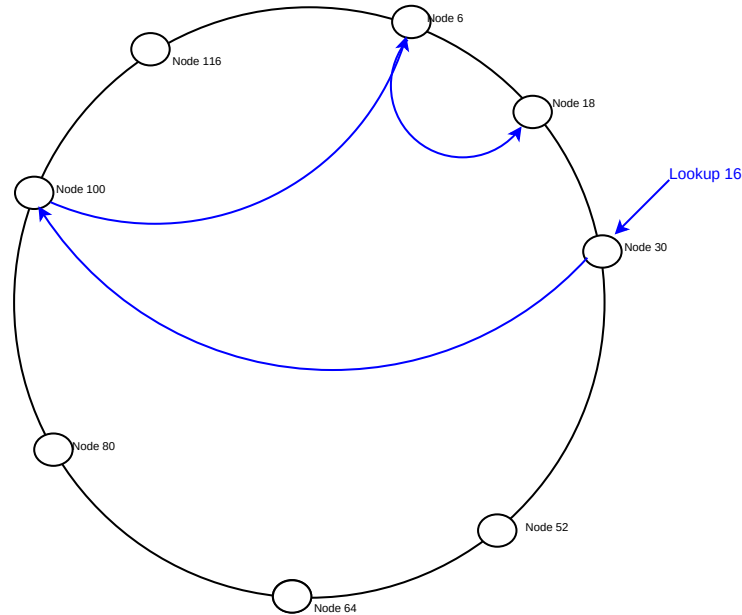


Fig. 2.6: A Chord lookup for item 16 is routed from node 30 to item 16's successor

Algorithm 2 $n.\text{closest_preceding_node}(\text{key})$

// $n.\text{closest_preceding_node}(\text{key})$ - find the closest preceding node to key in n 's routing table

```

for  $i = m$  downto 1 do
  if  $\text{finger}[i] \in (n.\text{id}, \text{key})$  then
    return  $\text{finger}[i]$ 
  end if
end for
return  $n$ 

```

We reproduce the pseudocode for these algorithms here².

Stabilize and notify, algorithms 3 and 4, work together to ensure that the successor lists and predecessor pointers remain up to date. The successor list is copied from each node's immediate successor, so that any changes further around the ring are propagated to the local successor list.

Algorithm 3 $n.\text{stabilize}()$

```

// Stabilize - called at regular intervals to verify  $n$ 's immediate successor
// and inform  $n$ 's successor of its presence
while  $\text{successorList}[0]$  has failed do
     $\text{successorList} \leftarrow \text{successorList}[1\dots]$ 
end while
 $x \leftarrow \text{successorList}[0].\text{predecessor}$ 
if  $x \in (n.\text{id}, \text{successorList}[0].\text{id})$  then
     $\text{successorList}[0] \leftarrow x$ 
end if
 $\text{successorList} = \text{successorList}[0].\text{append}(\text{successor}.\text{successorList})$ 
// Make sure successor list does not grow too long
 $\text{successorList}.\text{takeElements}(\text{MaxSuccessorListLength})$ 
 $\text{successorList}[0].\text{notify}(n)$ 

```

Algorithm 4 $n.\text{notify}(n')$

```

// Notify -  $n'$  believes it is the predecessor of  $n$ 
if  $\text{predecessor} == \text{nil}$  or  $n' \in (\text{predecessor}.\text{id}, n.\text{id})$  then
     $\text{predecessor} \leftarrow n'$ 
end if

```

The predecessor pointer is also checked periodically by the check predecessor algorithm. If it is noticed that the predecessor has failed, no attempt is made to replace it. Instead, it is assumed that when another node wishes to become our predecessor, it will notify us with a call to *notify* (Algorithm 4).

Algorithm 5 $n.\text{check_predecessor}()$

```

// check whether predecessor has failed
if  $\text{predecessor}$  has failed then
     $\text{predecessor} \leftarrow \text{nil}$ 
end if

```

The finger pointers are less vital to correct routing than the successor and predecessor pointers. As long as successor and predecessor pointers are accurate, a query will eventually reach its destination. Inaccurate finger pointers will reduce the efficiency of a lookup, but will not cause it to fail. Thus, we can repair the finger pointers less frequently — Algorithm 6 repairs one finger pointer each time it is called. The frequency with which it is called

² The algorithm given for stabilize differs slightly from that in [38], as it incorporates support for successor lists

will determine the maintenance bandwidth required, as well as the probability that a given pointer will be inaccurate.

Algorithm 6 `n.fix_fingers()`

```

// refreshes finger table entries
next ← next + 1
if next > m then
    next ← 1
end if
finger[next] ← find_successor(n + 2next-1)

```

Other Hash Tables

The three hash table systems we have described illustrate the range of approaches that can be taken in designing these systems. Besides those we have described in detail, many others deserve to be mentioned. In this section, we present what we feel are the salient details of six other hash table systems.

The Tapestry [60] system, designed at Berkeley, uses Plaxton routing to locate keys. Plaxton uses prefix based routing, which incrementally routes messages toward their destination, so that an increasingly long prefix of the query node's identifier is the same as the identifier we are looking for. Routing tables are designed to facilitate this process, and so must contain $b \cdot \log_b(N)$ entries, where b is the base of the identifiers (i.e. 10 for decimal, 2 for binary), and N is the size of the namespace. A given node will be found within $\log_b(N)$ hops.

Pastry [61], designed by a team at Microsoft research, uses a two step routing algorithm. Nodes use an increasing-prefix method similar to that employed by Tapestry, until they reach a node whose routing table does not contain any node with a longer matching prefix. When this happens, the node simply forwards the query to a node with an ID which is numerically closer to the requested item. The average case performance lookup performance is $\log_b(N)$ hops. Methods for dealing with popular keys (hotspots) and for taking advantage of physical network locality are also proposed.

The Koorde [62] system combines Chord with de Bruijn graphs - each node with identifier m is connected to nodes with ID $2m$ and $2m + 1^3$. De Bruijn graphs can be generalised to have degree k by adding connections from m to $[km, km+1, km+2, \dots, km+k]$. By adjusting the size of the routing table, k , from 2 to $O(\log(n))$, Koorde can achieve lookup complexity of between $O(\log(N))$ and $O(\log N / \log \log(N))$ hops.

Kelips [63] divides the identifier space into groups, such that the number of groups is approximately $\sqrt{N}$. Each node has a routing table that contains every other node in its group, and a few other nodes in each contact group. With this large routing table, it is able to achieve $O(1)$ hop lookups. Routing tables are kept up to date both by lookup traffic and by a gossip protocol, which informs current members of new nodes.

³ All arithmetic is modulo some maximum identifier

Viceroy [64] is a distributed hash table that achieves $O(\log(N))$ lookup times with a constant number of outward connections from each node, as opposed to the $O(\log(N))$ required by Chord. It achieves this arranging nodes in $\log(N)$ ‘levels’, with nodes in each level being linked in a ring, and also to nodes in both higher and lower levels. The number of inbound connections to each node can also be limited, leading to an architecture in which the failure of any given node has minimal impact on the rest of the system.

Symphony [65] also uses a constant k links per node, and achieves $O(\frac{1}{k}\log^2(N))$ average lookup hop complexity. The system is flexible, in that it allows for k to be varied while the system is running. The routing protocol is similar to that employed by Chord, but allows lookups to be routed both clockwise and anti-clockwise around the ring. Connections are chosen by estimating the size of the network, and using this to produce a probability distribution function. The identifiers are then chosen at random from this probability distribution function, so that the resulting network has small-world properties [66]. (Small-world networks are characterised by having a low network diameter — that is, the number of hops between two nodes is likely to be small.)

Choosing a Distributed Hash Table

We have now described nine different distributed hash table algorithms. All provide similar APIs, allowing data to be inserted ($put(key, object)$), and looked up ($object = lookup(key)$) in a distributed setting. It may also be possible to offer a uniform API to the routing information and key replication used by the various hash tables [67].

Despite the uniformity in their interfaces, the hash tables each have subtly differing performance characteristics. Simulation studies have been done to compare the performance of various hash tables; in [68], Kelips, Chord, Kademlia and Tapestry are simulated under conditions of node churn. It is shown that, with appropriate parameter choices, most hash tables perform similarly, in terms of the average lookup latency they can achieve given a certain quantity of maintenance bandwidth.

Jain et al. [69], have compared CAN, Chord, and Pastry in a real-world deployment, and also note that the lookup performances of the hash-tables are broadly similar when out-degrees are similar — they note that Chord slightly outperformed the other two systems at large system sizes because of its $O(\log(N))$ routing tables. Rhea et al. [70] have also compared Chord and Tapestry in a real-world setting, and note that Chord slightly outperforms Tapestry for lookups, but that Tapestry generally returns a node with a lower latency than the one found by Chord. As well as these small scale deployments, a real world performance tests of distributed hash tables has been conducted at a very large scale, using the Kad network — a variation of Kademlia. The Kad network forms a part of the file-sharing network e-Mule, and is currently running on approximately 980,000 nodes [71]. The network performs well, satisfying lookups with on average 3.8 network hops. This gives an important indication that DHTs can work in practical systems.

Because the implementation of routing and neighbour sets differs significantly from one hash table to another, a higher layer service may need to be optimised to take advantage of

the particular hash table it employs. Thus, despite the performance similarity among these hash tables, and the possibility of a common API, knowledge of which particular hash table will be used is necessary in designing these higher level services. Much of the further work in this thesis involves the use of a distributed hash table, and so it will become necessary to frame that work in the context of a particular DHT implementation. The fact that there is not a single hash table that is clearly superior to the others makes this choice somewhat arbitrary. The hash table we will be using throughout the remainder of this thesis is Chord, which we choose for the simplicity and elegance of its routing and maintenance algorithms, and its popular acceptance as a base for other work in this area. This does not preclude the use of our work with other DHT algorithms, though some adaptation is likely to be necessary.

2.4 Security in Structured Peer To Peer Systems

The security of any distributed system is a difficult and important issue. A distributed hash table that allows untrusted peers to become involved in the system must accept that some may behave maliciously.

Sit and Morris. have classified the various ways in which malicious peer could interfere with a distributed hash table [72] — dividing them into four separate classes - Routing, Storage, Denial Of Service, and Join/Leave attacks. Another class of attack, named the Sybil Attack, was identified by Douceur in [73].

We begin by examining routing attacks, giving four possible methods of limiting their effectiveness. We then examine the other attack classes Sit and Morris identify. Finally, we will explain Douceur's Sybil Attack and summarize recent work on protecting hash tables from such attacks.

2.4.1 Routing Attacks

When a lookup query is routed towards its destination, multiple nodes are involved in deciding upon the route it will take. Any of these nodes may provide inaccurate information in an attempt to disrupt system operation or cause a lookup to be routed to a node under malicious control.

Avoiding Routing Attacks with Iterative Routing

Sit suggests using iterative routing to allow the querying node to observe lookup progress. Ideally, at each routing step, the query node may request the entire routing table held by the closest known node, and independently choose which node to query next. The entries in each node's finger table are constrained to be the successors of particular points on the ring, and so nodes further from these points than expected are suspicious. Analysis of these routing tables may allow a falsified routing table to be detected. However, this method is rejected by Castro et al. [74] who claim that a malicious node can easily mix false and

correct routing table entries to make reliable automated detection of faulty routing tables difficult.

Avoiding Routing Attacks with Automated Detection of Faulty Routing

Castro et al. use an automated technique to reject lookup responses that are unlikely to be correct, based upon the distribution of nodes throughout the keyspace [74]. Keys are stored both on the node responsible for them, and on a number of that node's successors. A request is routed in the normal way, and the set of nodes returned as the successors of a key is checked. The checks are based on analysing the distribution of identifiers returned. If an attacker controls only a small proportion of the network, and is unable to control the identifiers allocated to its nodes, then the distance between k nodes controlled by the attacker is likely to be larger than the average distance between k nodes. Using this knowledge, we may check the successor list returned by a lookup, and reject it if it is significantly larger than expected. If a lookup is rejected, a more secure routing mechanism is used, sending each request multiple times using multiple channels. This mechanism is able to achieve a routing reliability of 99.9% if the proportion of malicious nodes is less than 30%, but rapidly deteriorates with larger proportions of malicious nodes [74].

Avoiding Routing Attacks with Consensus based Routing

S-Chord [75] proposes a routing system which is able to route lookups correctly when up to 25% of the system is made up of malicious nodes by dividing nodes into swarms, and routing requests from one swarm to another. Each swarm uses majority filtering to decide upon which messages to propagate to the next swarm. This method achieves $O(\log(N))$ lookup times, but each message requires $O(\log^2(N))$ messages to be sent. S-Chord also requires each node to maintain $O(\log^2(N))$ links.

Avoiding False Routing Updates with Sanity Checks

False routing updates may also be issued — in Chord, this could take the form of a malicious node incorrectly calling notify on a node that is not its successor. False routing updates may be avoided to some extent by having every node verify any updates it is sent — a technique that also helps avoid problems caused by network non-transitivity, which we will discuss later.

2.4.2 Storage, Denial of Service, and Join/Leave attacks

Other forms of disruptive behaviour identified by Sit and Morris in [72] include nodes not returning the data they are asked to store (Storage Attacks), deliberately overloading targeted nodes (Denial Of Service Attacks), and nodes joining and rapidly leaving the system (Join/Leave attacks).

Storage attacks involve a node that is given responsibility for a given key choosing to return inaccurate data when that key is requested. Such attacks may be made more difficult by using replication, and avoiding giving a single node the responsibility for all replicas of any given key.

The same tactic also makes overloading a target node less effective. Distributing responsibility for any given key across physically disparate nodes makes overloading any particular node ineffective. Deliberately overloading a target node is, as Sit and Morris point out, similar to a traditional Distributed Denial Of Service attack, and in many ways may be seen as a networking problem rather than a hash table problem.

Rapid joins and leaves are likely to cause significant disruption to the system, as objects stored with keys in their keyrange are copied to newly joined nodes, consuming considerable resources. The scope of such attacks may be limited by ensuring that the resources devoted to any new node are not so great as to degrade the services offered to the rest of the system. The rate at which malicious nodes may join can also be limited to some extent by the same measures we will suggest in the next section as countermeasures for the Sybil attack.

2.4.3 Sybil Attacks

Sybil attacks are a class of attack in which a malicious party poses as more than one node in order to gain control of a majority of either the entire network or of some portion of it. The name comes from the 1976 film *Sybil*, in which the eponymous heroine develops multiple personalities. As we have seen in the previous section, secure routing and storage often depend on limiting the proportion of malicious peers in the network. Once a malicious node controls a majority of the nodes in some region of the network, the methods we have identified for avoiding routing and storage attacks become ineffective in that region. Douceur [73] argues that under certain very general assumptions, it is not possible to prevent a Sybil attack without some form of centralised control. It is, however, possible to exploit scarce resources, such as the limited number of IP addresses an adversary may control, or the limits on computational power available to an adversary, to make Sybil attacks more difficult. In this section, we explore various techniques that aim to make Sybil attacks impractical for real-world adversaries.

Avoiding Sybil Attacks with Central ID Assignment

A centralised, trusted, certificate authority may use public key cryptography to securely allocate node identifiers to IP addresses, signing those identifiers so that they can be verified. Nodes would verify that a node had a signed identifier before incorporating them into their routing tables. This central authority could then control the number of identifiers a single IP-address or subnet could control, limiting the scope for Sybil attacks.

This method, while simple and effective, creates an element of centralisation, and a single point of failure for new nodes joining the system, and so is not ideal.

Avoiding Sybil Attacks with IP Address based Identifiers

Another possible way of limiting the scope for Sybil attacks is to limit the number of identifiers a malicious node may take. For instance, we may use a hash of a node's IP address as a node identifier, so that a single IP address cannot pose as more than one Chord node.

This approach has several flaws, however. Because of the addressing restrictions imposed by IPv4, it is not uncommon for many machines to share a single public IP address, using network address translation. We may wish to allow those machines sharing an IP to run their own nodes. Even when each machine has its own IP address, limiting a single IP to a single node may not be acceptable, as several DHT storage mechanisms call for one physical node to run multiple virtual nodes in order to facilitate load balancing. Finally, this method also depends on an adversary having access to only a small number of IP addresses, which may not be the case.

Avoiding Sybil Attacks with Bootstrap Graphs

Danezis et al. [76] propose a scheme in which any node can introduce any other into the network, but must only introduce nodes they trust. A malicious peer may fool an honest one into introducing it to the network, and thus introduce many more malicious peers itself. By keeping track of who was introduced to the system by who, a bootstrap graph can be constructed. This bootstrap graph can be used to prefer using nodes introduced with different bootstrap paths, meaning a malicious user would have to trick a large number of honest peers into introducing them to the system if it were to take control of query routing.

The main disadvantage here is that it is not specified how any node is supposed to decide whether another node is malicious before it joins. Without any such mechanism, it seems likely that large numbers of honest peers may well be 'tricked' into introducing malicious nodes to the system.

Avoiding Sybil Attacks with Computational Bottlenecks

Another method of limiting the number of computationally limited malicious users which can join the system is to use cryptographically hard problems (e.g. finding prime factors of a large number), which must be solved before a node can join the system. Rowaihy et al. propose a system where a node wishing to join the system must solve several puzzles [77]. To join the system, a node A contacts any node B already in the system, and is issued with a puzzle. Upon correctly solving the puzzle, node A is introduced to another node, which also issues a puzzle. The request travels up a hierarchy until it reaches the bootstrap node, the first node in the system. The bootstrap node issues a signed token which allows node A to enter the system. Node A is also assigned an identifier. This alone only limits the rate at which malicious nodes may join the system, however we also require some nodes to regularly re-solve puzzles and reacquire a signature from the bootstrap node.

Using this system, N attackers with average speed nodes, solving puzzles which take an average node l minutes to solve, which are required to refresh their keys every W minutes, the number of attacker that can enter the system is given by:

$$N_{attacker} = \frac{NW}{l} \quad (2.1)$$

By adjusting W and l , we can provide a hard bound on $N_{attacker}$, and so on the proportion of malicious nodes that may join the system.

This approach is robust to any adversary who controls a limited quantity of computational resources. One of its main strengths is that it does not depend on the adversary controlling a small range of IP addresses. The main drawback is that solving the computational puzzles may discourage nodes from participating in the network, as it is costly in terms of CPU usage even for well-behaved nodes.

Avoiding Sybil Attacks with Distributed Self-Registration

A distributed method of limiting the number of node identifiers any given IP address can take control of is described by Dinger and Hartenstein, who propose a system of self-registration. Each node may control a number of identifiers. To join the system, a node computes a hash of its IP address and a port number, using some well known hash function. This gives the node identifier.

Each node identifier must be registered by communicating with r registration nodes. Each node first calculates its registration identifiers, which are a hash function of the node's IP address and a registration index. The registration index is a number between one and r , and the owner of these registration indices are the registration nodes. Before joining the system, the joining node must contact each registration node, and inform it of the ID it wishes to use. These registration nodes will accept the registration of a new node only if the number of nodes already controlled by that IP address is less than some constant, a .

When a node attempts to join the system, current participants will first verify that its ID corresponds to its IP address, then verify that its registration has been accepted by at least half of the registration nodes.

This system can ensure that no more than the desired number of nodes can register from any IP address while the number of non-malicious participants remains below 50%. The number of registration nodes required rises with the proportion of malicious peers; if 50% of the peers are malicious, 50 registration nodes are required to ensure the probability of a node being accepted into the system falsely is less than 0.1%.

Of all the anti-Sybil system discussed, this is in some ways the most appealing, as it does not rely on a central authority, or on our adversary being computationally limited.

2.5 Hetrogeneous and Unreliable Physical Networks

Though the description of the distributed hash table algorithms we have given so far have been elegant, there are practical considerations that they fail to address. Network connectivity may not be as symmetric as we would like, and communication overheads between nodes are non-uniform. Using reliable communication protocols such as TCP cannot always compensate for these issues. Here we summarise how the routing and maintenance algorithms can be modified to allow hash tables to operate efficiently and reliably in these circumstances.

2.5.1 Non-uniform Network Latency

The simplistic version of the Chord protocol we gave in earlier sections does not take network latency into account when choosing which nodes are in a given node's routing table. In an internet setting, latency may vary by several orders of magnitude between machines on the same network, and between machines on opposite sides of the Atlantic. It is preferable that in being routed from the query node to the destination node, as few high latency hops are used as possible.

To avoid high latency routing hops in Chord, a method known as proximity neighbour selection can be used. This changes the way in which a finger node is chosen — a node with id a chooses as finger i , any node in the region $a + 2^i \dots a + 2^{i+1}$. Up to x nodes in this region may be considered, and the node with the lowest round trip time is chosen. Simulations have shown that increasing the value of x above 16 leads to diminishing returns. In real-world network topologies, sampling 16 nodes for each finger reduces average routing latency to less than half that achieved by naive Chord [78].

Another method of reducing the impact of high latency links is to issue multiple queries in parallel. EpiChord [79] sends multiple queries in parallel to find the lowest latency route to a key.

2.5.2 Non-uniform Network Connectivity

The network model used in most DHT papers assumes a network in which any node can contact any other. This is not the case in general, due to the wide-scale adoption of firewalls and network address translation to connect home networks to broadband internet connections. Network address translation (NAT) allows multiple machines to share a single IP address, but does not necessarily allow incoming connections to those machines

One possible solution to the NAT problem would be to move to IPv6, which provides a larger address space, and would allow sufficient address space for all devices to be assigned globally unique IP addresses. IPv4 is likely to remain in use for some time, however, and so methods of working with the limitations imposed by network address translation are necessary. In the meantime, a variety of techniques may be used to allow incoming connections, including relaying connections between hosts behind NAT through a third party who is not

behind network address translation. [80]

Freedman et al. [81] have observed another problem that large-scale peer to peer systems can encounter. Network connections between three nodes, A, B and C are non-transitive if A can communicate with B, and B can communicate with C, but C and A are not able to communicate. Non-transitivity is frequently observed in large-scale deployments of distributed hash tables, and may occur because of link failures, or incorrect router configurations. Freedman et al. identify four problems caused by non-transitivity, including routing loops, inconsistent ownership of the keyspace, invisible nodes, and broken return paths. Invisible node problems are caused when non-transitivity means that two nodes are unable to communicate, and so each assumes the other has failed. They update their local routing state appropriately, removing the node they are unable to communicate with. These updates may then be propagated to other nodes, removing both nodes from the system entirely. They suggest that invisible node problems may be avoided by having each node verify all routing updates, so that a node which is unable to connect to another because of non-transitivity cannot declare that node as having departed. Broken return paths occur when the owner of an item is unable to communicate directly with the requesting node, and may be avoided by returning objects along the same path they were looked up with. Inconsistent ownership of the keyspace, where more than one node believes it is responsible for the same key, may be remedied by using either a distributed consensus algorithm, or by replicating a key on multiple hosts.

2.6 Summary

We have introduced the concept of peer to peer computing, and given a taxonomy of peer to peer computing techniques. Considerable attention has been given to the design space of structured peer to peer systems, particularly distributed hash tables, and nine different DHT designs have been discussed. We have then gone on to survey some recent work on the practical adaptations necessary to make a DHT useful in real-world settings, where faulty networks and malicious peers become issues. The area of DHT research is constantly evolving, and some of the papers we have referenced have been published only months before this thesis. It is likely that the DHT techniques will continue to evolve to deal with these problems.

One important aspect of distributed hash table usage should be conspicuous by its absence. As we have observed, reliably and securely operating a distributed hash table requires that keys are reproduced on multiple nodes. These replicas need to be maintained as nodes join and leave the system, and co-ordination is needed to decide upon where they must be stored. It is this area that forms the focus for the next chapter.

3. RELIABLE STORAGE IN DISTRIBUTED HASH TABLES

3.1 Introduction

We have seen how distributed hash tables can provide a reliable routing layer, which can nominate a node as responsible for a given key. Through the use of a maintenance algorithm, which repairs routing tables, it can continue to offer this routing service despite node churn.

For the purposes of providing grid middleware, however, we wish to go beyond simply nominating a node as responsible for a key — we also wish to associate data with that key. If we wish to store an object in our hash table, we must first name that object, so that we know which key to store it under. This might be done by running some hashing algorithm, such as MD5 [82] or SHA[83], on the object. We can then use the hash value as the key under which that object will be stored in the hash table.

Each node then maintains a database of (key, object) pairs for those keys it is responsible for. As nodes join and leave, however, the set of keys a given node is responsible for may change. When a node fails, the key-value pairs it was responsible for become inaccessible, and must be recovered from elsewhere. When a node joins, it becomes responsible for keys that were previously owned by another node, but needs to be sent the values associated with those keys. To do this, a *replication algorithm* must store more than one copy of any data, on a set of nodes; the *replica group* for that key. The replication algorithm must ensure that a consistent set of values continues to be stored on nodes in the replica group. It may also provide a mechanism for discovering all nodes in the replica group for an item so that they may be updated.

In this chapter, we will begin by looking at the aims and the design space of replication algorithms. We then present several specific algorithms, including soft-state, metadata based, DHash, and dynamic replication. The contributions in this chapter include dynamic replication, which offers considerable improvements to the Replica Enumeration algorithm it is based upon. We provide entirely new maintenance, fetch, placement and update algorithms that allow the techniques proposed by replica enumeration to be practically useful.

For the sake of increased clarity, we have separated the introduction and explanation of the algorithms from either any analysis of how they should be configured, or any quantitative evaluation or comparison of their performance. This chapter is dedicated to the replication algorithms themselves. We will analyse how we can configure them so that they achieve the desired level of reliability in the next chapter, and a comparative analysis of their performance will be presented in Chapter 5.

3.2 Replication Algorithms

A replication algorithm not only allows us to store data in a hash table, it also affects other aspects of the system, including how reliable, how scalable, and how secure it is. We will now examine the various ways a replication algorithm can affect system performance.

Reliability: The replication algorithm should not rely on any single node, and must recover from churn without user intervention. The algorithm should seek to place replicas so as to maximise reliability.

Security: The replication algorithm should avoid giving a single node control over storing replicas of any object, so that a single malicious user cannot interfere with data storage.

Scalability: The replication algorithm should scale to storing large quantities of data on a large number of nodes. If the replication algorithm is not to limit the scalability of Chord, per node state and bandwidth usage should scale as $O(\log(N))$.

Addressability and Consistency: The replication algorithm should provide clear guarantees about the consistency of replicas. If we wish to consistently update data, we must first find all the replicas of that data — the replicas need to be easily *addressable*. A distributed commit protocol can then be used to update all replicas consistently. If consistent updates are required, addressing all replicas should be efficient.

Lookup latency: The replication algorithm may reduce the time taken to look up information by placing replicas of data in a manner that allows network locality to be exploited.

Load balancing: The replication algorithm may provide caches of more popular objects in order that the load is evenly balanced among all the nodes in the network.

An ideal replication algorithm would achieve all of these aims, but real world algorithms must compromise. The choice of replication strategy may depend on which goals are more important to the task being considered.

3.3 Design space of reliable peer to peer storage

A replication algorithm can be characterised by the way in which it solves four key problems. The method used to address these issues will decide the extent to which the algorithm reaches the aims we have set out in the previous section.

Replica Maintenance: Node churn will cause replicas to be lost. The replication algorithm must detect and repair these lost replicas without using excessive bandwidth.

Replica Addressability: In order to update an object, we need to locate all replicas of that object. Ways of doing this include limiting replica placement to a fixed number of nodes, searching for replicas, and periodically deleting old replicas. Replica addressability is required if updates are to be made consistently across all replicas of an object.

Replica Cardinality: The number of replicas we keep of a given key may either be fixed in advance, or allowed to vary according to the key's popularity. Variable cardinality often provides superior load balancing, but makes addressability more difficult to achieve.

Replica Placement: The replica placement strategy determines which nodes replicas should be stored on. This can have an impact on both performance and reliability.

Each of the approaches to replication we will now explore provides different solutions to these key problems, and this affects their overall performance. We will begin by describing soft-state replication.

3.4 Unstructured/Soft State Replication

Soft-state storage is a very simple approach to replication. It does not offer a truly reliable storage system unless used with some external reliable store, and so is unlikely to be of general use in removing bottlenecks from grid computing systems. Nevertheless, it can help alleviate bottlenecks for read only data, and is a widely adopted approach.

Soft-state replication makes no attempt to maintain replicas as nodes join or leave. Instead, replica maintenance is performed by periodically reinserting all data objects into the network. Data objects which have been in the network for more than some timeout period expire and are deleted.

Updates are possible only when old data expires and is deleted from the system. New versions may then be inserted whilst maintaining consistency, since any older versions will have expired.

Soft state replication can provide excellent load balancing, and rapid data lookups. As the maintenance and update algorithms do not require replica addressability, we may place replicas wherever demand dictates. This provides opportunities to exploit network locality and use caching to provide fast data lookups.

However, soft state maintenance requires both a reliable storage system outside of the DHT, and a great deal of network communication to reinsert the data on the correct nodes. Because maintenance is expensive, long expiry times are necessary, which prohibits frequent updates. This limits the practical usefulness of soft-state replication in providing mutable storage.

Soft state expiry may be more useful when used in combination with another algorithm. Data expiry can be useful for garbage collection, deleting objects that are not frequently used, or which are no longer needed, while some other technique provides reliability.

3.5 Proactive Replication

Replica maintenance strategies can be either proactive or reactive. A reactive strategy will replace lost replicas as a reaction to node failure, maintaining a target level of replication.

Proactive strategies may also be used. A proactive strategy does not react to node failures, but creates replicas to meet some performance target. An example of a proactive replication scheme is Beehive [84], which creates replicas so as to achieve a target level of lookup performance. Beehive will create large numbers of replicas of popular items, and smaller numbers of replicas of less popular items, so that the average lookup time meets a user-defined target. Beehive works well with power-law type query distributions, where a small minority of items are the target of the majority of requests. In systems where all items are of uniform popularity, it will need to create a much larger number of replicas to achieve a given performance target.

Another proactive replication strategy is used by Tempo [85]. Tempo attempts to avoid the bandwidth spikes which tend to follow a node failure in systems that employ reactive maintenance. Bandwidth spikes are avoided by using a small constant bandwidth budget to create new replicas of objects at all times. This approach is particularly useful for wireless networks, where the bandwidth available may be limited.

Proactive replication could be used in combination with many of the other techniques considered in this chapter. In grid environments however, bandwidth is likely to be plentiful, and it is not clear that key popularity would follow a power-law distribution. Accordingly, we do not feel Beehive or Tempo style replication is likely to be beneficial, and have concentrated on reactive techniques in this thesis.

3.6 Metadata based replication

Metadata based replication algorithms use two classes of data — objects, and metadata. There are two classes of metadata based replication — indexed replication, and version ID replication, which we will now describe. With either algorithm, the metadata allows us to locate a replica of the current version of an object.

While the metadata items manage the replication of the data items, the metadata items themselves must also be replicated. These cannot be replicated using metadata based replication, without introducing a new class of meta-metadata, which must also be replicated, introducing the same problem. Both the DHash and Dynamic Replication schemes, which we will describe later in this chapter, are a suitable choice for the replication of metadata.

3.6.1 Indexed Replication

Indexed replication uses the DHT as a replica location service. Instead of the replication algorithm being responsible for storing data, it instead stores and maintains an index. This index contains a list of nodes at which the actual data is stored.

Data items may then be stored on any node, as long as the location is catalogued in the index. This allows completely free replica cardinality and placement, so that popular keys can be heavily replicated.

Fetch latency may be increased since the index must be located before the data item can

be retrieved. Some mechanism will also be required to maintain both the replicas in the index, and the index items themselves.

3.6.2 Version ID

The Version ID system is used in the Tapestry[60] based storage system OceanStore[86].

For a given object, we store two pieces of data in the DHT. The *version ID* object is stored at a location which can be calculated from the name of that object. The data item itself is stored at another location. The version ID is the metadata item, and contains a pointer to the latest version of the data item. Data items themselves are immutable; only version ID objects can be changed.

If we wish to insert a new version of an object, we first store the new version in the DHT under some key, then update the version ID object so that it points at the new key. Since we can issue updates without finding all copies of the old object, there is no need for replica addressability. This means we can cache a data item at any node. This may increase lookup performance significantly, especially for popular items.

As with replica indexing, the requirement to look up metadata before getting an item may increase latency. Garbage collection is also a concern. If a popular data item is updated, numerous caches of the old version are likely to exist and these are unlikely to be of further use. Some kind of soft-state mechanism for caches is likely to be useful.

3.7 DHash replication

The DHash approach [87] to replication combines the replica placement strategy originally proposed by the authors of Chord with a maintenance algorithm suggested by Cates [88]; this combination is used by the DHash storage system and the Ivy File System [89].

An object is stored at the node responsible for that object's key - this node is the *root node* for that object. In addition to this, replicas of an object are placed on the $r - 1$ successors of the node responsible for that object's key, to achieve a total replica cardinality of r . These nodes make up the *replica group* for that key. To maintain this placement pattern in the face of node churn, there are two maintenance protocols: the *local* and *global* algorithms; these prevent the number of replicas of any object from either dropping too low or rising too high.

Local Maintenance Each node sends a message to its $r - 1$ successors, specifying the key range it is responsible for. These nodes then synchronise their databases so that all objects with keys in this range are stored on both the root node and its successors. (Methods for database synchronisation, such as Merkle Tree hashing [90], are discussed in [88].) The process is illustrated in figure 3.1.

To repair the overlay, the local maintenance algorithm runs twice: in the first pass, the objects whose keys lie in the key range of the root node are identified and gathered at

the root note; in the second, replicas of these objects are distributed to the successor nodes.

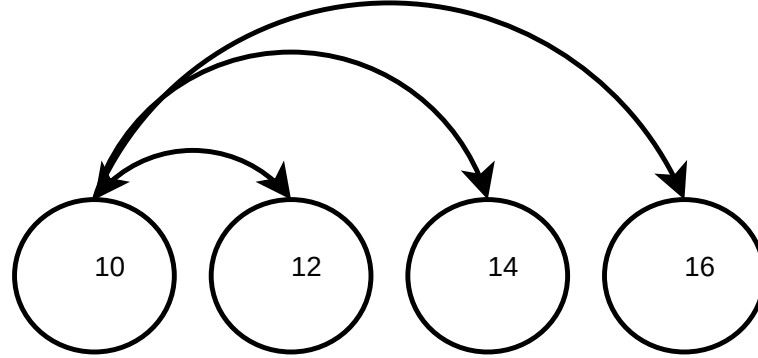


Fig. 3.1: Local Maintenance runs on node 10, synchronising the databases of that node and a number (here three) of its successors. Arrows indicate database synchronisation traffic.

Global Maintenance A node periodically checks the keys of the objects in its database to see if it stores any object that it is no longer responsible for. To do this, it looks up the root node for each object it stores, and checks the successor list of that owner. If it is within $r - 1$ hops of the node, then it will be one of the first $r - 1$ nodes in the successor list. If is not in this list, the node is no longer in the replica group, and so is no longer responsible for keeping a replica of this object. In this case, the object is offered to the current owner, after which it may safely be deleted.

For clarity, we have described global maintenance as checking each object individually. However, a performance optimisation makes it possible to confirm a node is in the replica group for a large number of items with a single lookup. Once a node is found to within $r - 1$ hops, we know that we are responsible for replicas of objects in that node's keyrange. This allows us to verify we should still store all of these objects, and we do not need to look up their owner. The number of lookups required for global maintenance thus falls from $O(\#ObjectsInDatabase)$ to $O(r)$.

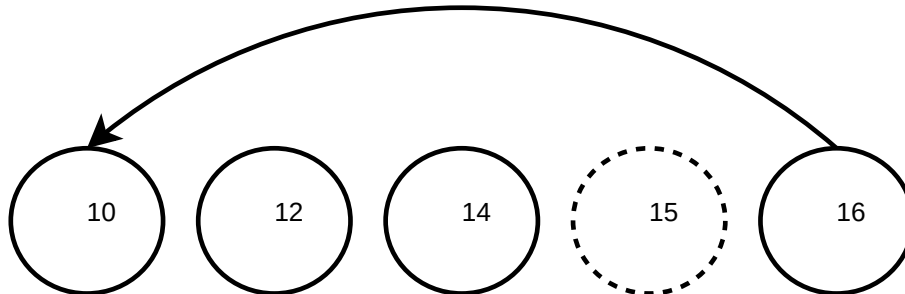


Fig. 3.2: Global Maintenance runs on node 16. A new node (15) has joined since maintenance last ran, and node 16 is no longer within three hops of node 10. In this example, we store 4 copies of each object, and so node 16 is no longer responsible for storing replicas of node 10's data. It sends that data back, and deletes it from its local database.

These algorithms ensure data is placed on the correct nodes, and on the correct nodes only. Immediately after running this maintenance algorithm, the system will be in the ideal state, with replicas stored in the correct placement pattern. After maintenance runs, the system will start to deteriorate again, as churn damages the system. We must run the replication algorithm with sufficient frequency that no data is lost, a problem that we examine in the next chapter.

3.7.1 DHash Data Fetch Algorithm

Although the maintenance algorithm restores the system to its ideal state each time it is run, the system may not be in its ideal state during the periods in-between maintenance algorithm runs. Unrepaired damage caused by node churn will cause some replica locations not to contain the information we seek. When we contact a replica holder to *get* the data, we must anticipate that this may fail.

In order to handle this possibility of failure, we use a *fetch* algorithm, that hides the complexity of searching multiple replica locations. The pseudocode for this algorithm is given as Algorithm 7. The algorithm looks up the responsible nodes using the `findSuccessor`¹ algorithm specified in the previous chapter. It tries each successor in a random order, sharing the load of providing an object between all the replica holders.

Algorithm 7 Fetch for *key* under DHASH Replication

```

successors  $\leftarrow$  findSuccessors(key)
while  $\neg$ item and  $\neg$ successors.isEmpty() do
    node  $\leftarrow$  successors.popRandom()
    item  $\leftarrow$  node.get(key)
end while

```

3.7.2 DHash Updates

Because DHash provides replica addressability, we can locate all replicas of any object, and update these replicas in a consistent manner. Some form of transaction protocol, such as two-phase commit, is necessary if we wish these updates to be guaranteed consistent. Once the set of replicas to be updated has been identified, the exact approach taken to updating those replicas is orthogonal to the design of the rest of the system, and so we will not prescribe any particular update method.

DHash consistency issues

DHash addressability dictates that replicas are only stored on the $r - 1$ successors of the key's owner. The global maintenance algorithm prevents replicas being stored on nodes other

¹ We actually use a `findSuccessors` method, which simply returns a node's entire successor list, rather than a single successor.

than these successors for more than one maintenance interval. However, during the periods between maintenance intervals, it is possible that new nodes may join, causing a node to become more than $r - 1$ hops from the owner. This means any updates will not reach that replica, and this replica will become stale. Because nodes keep track of their immediate predecessor only, they will not be aware of the fact they are no longer in the replica group, and will return this stale information if they are asked for it. To avoid returning stale data, each node would need to constantly keep track of its $r - 1$ predecessors, so that it can verify which keys it is in the replica group for. This may increase maintenance traffic significantly.

3.8 Dynamic Replication

We now present an alternative to DHash replication which we will call dynamic replication. Dynamic replication inherits some properties from a technique called Replica Enumeration [91]. In this section, we begin by describing the original replica enumeration approach, and some of the problems it causes. We give our solutions to these problems, and show how we can configure our dynamic replication function with appropriate placement and cardinality strategies.

3.8.1 Replica Enumeration

Replica Enumeration, as proposed by Waldvogel et al. in [91], aims to remove some of the placement and cardinality restrictions imposed by DHash replication, whilst preserving addressability and the ability to make consistent updates.

The essential difference between DHash and replica enumeration is the placement strategy. The placement strategy for replica enumeration is based around an *allocation function*, $h(m, d)$. For each object with key d , the replicas are placed at replica locations determined by $h(m, d)$ where $m \geq 0$ is the index of that replica. Waldvogel suggests that the allocation function is pseudo-random², so that the replicas are evenly distributed about the address space.

The replication cardinality is variable in a fixed range $1 \leq r \leq R_{MAX}$, allowing greater replication for objects in greater demand. The mechanism used to decide the exact number of replicas is not specified, but might be designed to adapt to both the reliability of the network, and the load on those nodes storing each object.

To provide addressability, the following invariants are maintained:

1. Replicas of an object with key d are only placed at addresses given by $h(m, d)$, where $0 \leq m < R_{MAX}$.
2. For any object d in the system, there always exists an initial replica at $h(0, d)$. The node responsible for this key is that object's *root node*.

² We will show that using pseudo-random placement is impractical if we wish to provide reliable storage.

3. Any further replica with ($m > 1$) can only exist if a replica currently exists for $m - 1$.
4. No resource exists for more than R_{MAX} instances.

Various strategies for finding data are possible in this scheme. One that is generally efficient is to do a binary search over the range $[1..R_{MAX}]$ starting from a randomly selected index in that range. If the data is not replicated at a given location, we use invariant 3 to reduce the search range accordingly.

Advantages of Replica Enumeration

Dynamic replication can help alleviate the lookup bottleneck that affects DHash replication. DHash replication requires that all lookups for a popular key are directed to that key's owner. With an appropriate allocation function, dynamic replication can place replicas at evenly spread well known locations around the ring. Lookup queries for a popular object are then distributed more evenly, and do not have to pass through any single node. We may also select replicas based on latency, calculating the cost for each potential source of an object, then requesting that object from the lowest cost source.

The use of a hash function to place replicas offers improved security over DHash. The DHash replication algorithm entrusts a single node with providing the list of nodes (its successors) onto which a replica is stored. Dynamic replication allows the replica holders to be looked up and verified independently, avoiding the trust bottleneck caused by DHash.

Using a hash function can also help alleviate the consistency issues that DHash can face between global maintenance runs. We have already noted that because Chord nodes are not aware of more than one of their predecessors, they cannot verify they are still responsible for a key using local state. With dynamic replication, it is simple to verify whether a node is still in the replica group for a key by verifying the hash function still maps that key into that node's keyspace.

Replica enumeration then, is potentially a useful improvement over DHash replication. However, the algorithm as presented by Waldvogel also has several problems we need to overcome.

Problems with Replica Enumeration

Allocation Collisions The allocation function may map the same object into the keyspace of a single node under two distinct replica indexes. We call this an *allocation collision*. The effect of a collision is to reduce the number of distinct nodes in the replica group, and hence to reduce reliability. Allocation collisions may occur even if replica locations are well spaced, if the number of participating nodes is relatively small.

Lack of Churn Tolerance The four system invariants we have listed must be maintained if the system is to operate correctly. Unfortunately, maintaining Waldvogel's invariants in a system with a high churn rate is very difficult. Node departures and arrivals could

cause any of the invariants to be violated. It can be shown that lookups will proceed correctly as long as at least invariant two holds. However, for replica addressability, invariants 1 and 4 must also hold, and these may be violated by a single node joining or leaving the system.

Communication costs If we wish to maintain replicas, and thus add churn tolerance, we will need to periodically run a maintenance process. Because Waldvogel suggests each key is mapped by the allocation function to a number of pseudo-random replica locations, it is likely that each key a node owns will map onto the ownership space of a distinct set of nodes. When we maintain data, the node will need to communicate with each of these nodes for each key it is responsible for, in order to collect and distribute replicas. Communication with such a large number of nodes is likely to be impractical. To keep communication costs low, we need to provide alternative placement functions, so that the number of nodes responsible for holding replicas of a single node's keys is as small as possible.

We will now present an improved version of Replica Enumeration, that attempts to address some of these failings.

3.8.2 Dynamic Replication: Improving Replica Enumeration

We will now modify the replica enumeration method with a new maintenance algorithm to allow it to operate correctly and reliably in a system with high churn rates.

The original system invariants given by Waldvogel et al. guarantee that a replica is always available at $h(0, d)$. In a dynamic system, we cannot rely on any single node being available, so we cannot maintain this invariant. We modify our invariants so that we aim to maintain at least one replica in a *core group* of replicas, with high probability. We introduce a new variable, R_{MIN} , for the number of nodes in the core group.

We will use the following definitions to refer to the various roles nodes play in holding replicas.

- A *replica holder* for an object with key d is a node which owns the key $h(m, d)$ for some value of m .
- The *core group* for an object with key d is the set of replica holders for which $m \leq R_{MIN}$.
- The *peripheral group* consists of those replica holders for which $R_{MIN} < m \leq R_{MAX}$.

We can now express our new invariants ³:

³ We use the term invariant to remain consistent with Waldvogel's paper; strictly speaking, these are not invariants, but a set of 'ideal' conditions that we can maintain with some high probability.

1. Replicas of an object with key d can only be retrieved from addresses given by $h(m, d)$ where $\{0 \leq m < R_{MAX}\}$.
2. For any object in the system, there exists with some (configurable, and usually high) probability a replica in the core group.
3. Any peripheral replica with $(m > R_{MIN})$ can only be retrieved for a single maintenance interval if no replica currently exists for $m - 1$.

When we store an item, we try to store it at the first R_{MIN} locations. If we encounter no allocation collisions, we are finished, and all replicas of the object reside in the core group. If, however, we encounter allocation collisions, we must recover from these by trying extra locations, which we will select from the peripheral group. We will continue to try increasingly large replica indices until we have stored a total of R_{MIN} replicas without allocation collisions, or there are no more indices available.

Thus, we have divided the range of replica locations into two parts. The first R_{MIN} locations are the core group, and we will always place replicas at these locations. The second part, the peripheral group, may also contain replicas, but is not guaranteed to.

With these modified invariants, we can give three maintenance protocols which maintain the system invariants under churn.

Dynamic Replication Maintenance Protocols

Core Maintenance Each node first uses the hash function to calculate the keys at which replicas of its objects might be stored. It then looks up the owners of these keys. It then synchronises the appropriate part of its database of objects with the owners of these keys — the appropriate part being those keys which are mapped by the allocation function into that core replica holder's keyspace. This will restore the second invariant.

Core maintenance also deals with allocation collisions, by keeping track of which nodes store replicas from which key ranges, and placing additional replicas on peripheral replica holders if any given keyrange is mapped to the same node more than once. If some part of the keyrange is involved in an allocation collision, an additional copy of that keyrange is placed at the first peripheral replica location. If the same keyrange is involved in another allocation collision, a second additional copy is placed at the second peripheral location, and so on.

Peripheral Maintenance To maintain the third invariant, any node that stores a replica with index $m > R_{MIN}$ must check that a replica of that object is held also on the replica predecessor, the owner of the location with index $m - 1$. If a replica is not present on the previous node, the replica is *orphaned*.

For each peripheral replica a node holds, it must obtain a summary of the objects with the previous index on the replica predecessor. Bloom filters [92] can be used to reduce the cost of these exchanges.

These summaries can be used to remove orphaned peripheral replicas from the system (after offering them to their root node); these orphaned replicas should not be used

to answer fetch requests, but still be stored for at least one maintenance interval—maintenance will often replace the missing replica parent.

Global Maintenance Each node calculates the replica locations for each object it stores. If it stores any object for which no replica location exists within its ownership space, it offers the object to its root node, then deletes it. This restores the first invariant.

Running these algorithms will restore the system to its ideal state, and maintain our invariants, as long as at least one replica of every object remains in the system. As with the DHash maintenance algorithm, we must run the algorithm frequently enough that node churn is unlikely to remove all replicas of any object. We will show in the next chapter that differences in the placement strategy may mean that we need to run maintenance less frequently than would be necessary under DHash.

With these maintenance functions, we have addressed the lack of churn tolerance, and provided a mechanism to deal with allocation collisions. However, we still need to deal with the high bandwidth usage that a pseudo-random replica placement algorithm would result in.

The choice of allocation function is critical to maintenance performance. For each object with key d that a node owns, it must lookup and contact every node in the core replica group for d in order to run the core maintenance protocol. In order that this is scalable, we must ensure that as many of these lookups as possible can be satisfied with little network communication. This requires that the allocation function maps one node's data onto a small number of replica holders. Maintenance performance can be further improved by using functions which exploit the local routing state already stored on each node: for example, we might place replicas on the finger or successor nodes of each node.

We have noted that pseudo-random placement is not a realistic option. Using a pseudo-random function to determine replica locations would lead to high maintenance costs. We suggest that for a given m , $h(m, d)$ is a *translation* in d . This means that for a given value of m , the image of one node's key-range is another continuous range of the same size, which is likely to be divided between a small number of nodes. The total number of nodes that must be communicated with to perform maintenance will be much smaller than with a pseudo-random placement function.

An allocation function which is a translation allows for optimisations in the core maintenance algorithm, as we map entire ranges of keys at once, removing the need to lookup each key individually.

3.8.3 Dynamic Allocation Functions

We now describe five allocation functions we believe might offer good performance. In defining the allocation functions, we use two variables, the number of nodes in the system N , and the maximum key value K . The Chord paper suggests a value of $K = 2^{160}$ [38]. The value of N need not be completely accurate, and may be specified by the user, or estimated by the nodes themselves.

Various techniques can be used to estimate system size without the use of a central server. The simplest method of estimating the number of nodes in the ring is to lookup a number of successors of a given key. The average keyspace owned by each of the nodes in the successor list can then be computed, and used as an estimate of the average keyspace owned by every node in the system. We can then divide the total keyspace size by the keyspace owned by an average node, and obtain an estimate of the number of nodes in the system. The more nodes in the successor list, the more accurate this estimate is likely to be. For more details on estimating the size of a Chord ring, see [93].

For each allocation function, we give the function itself, a brief description of its properties, and a diagram illustrating the layout it aims to achieve.

Successor Allocation

$$h(m, d) = (d + (m \cdot \frac{K}{N})) \bmod K$$

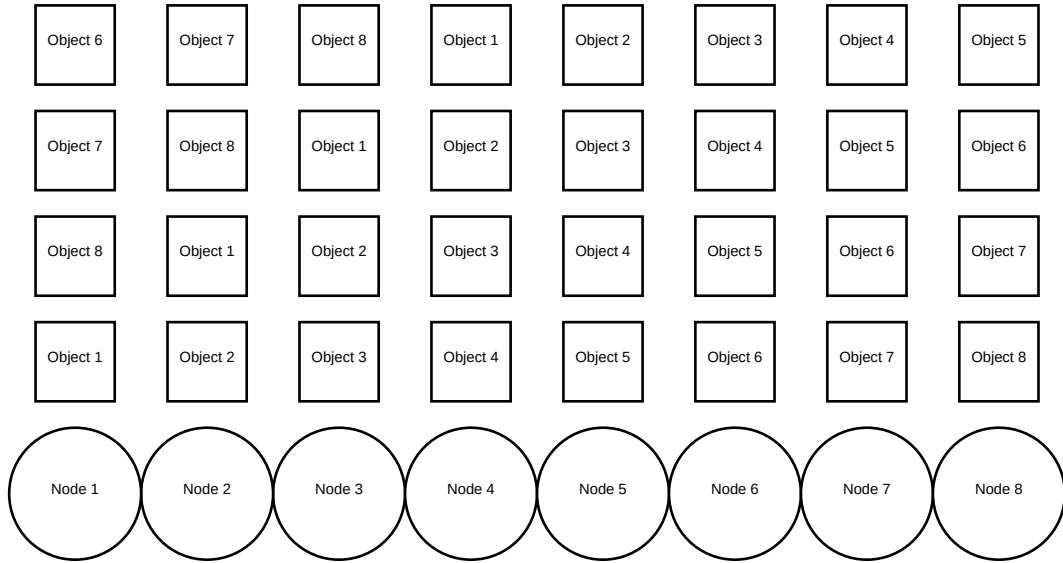


Fig. 3.3: Illustrative example of successor allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.

The successor placement pattern attempts to map replicas of a key onto the successors of the owner of that key. Attempting to map replicas onto successors is efficient, as the Chord Protocol maintains a list of each node's successors as local routing state, so we can locate core replica holders using local routing information, and no network traffic is required. Note that unlike DHash replication, the allocation function defines which nodes store replicas, and not the successor list.

Predecessor Allocation

$$h(m, d) = (d - (m \cdot \frac{K}{N})) \bmod K$$

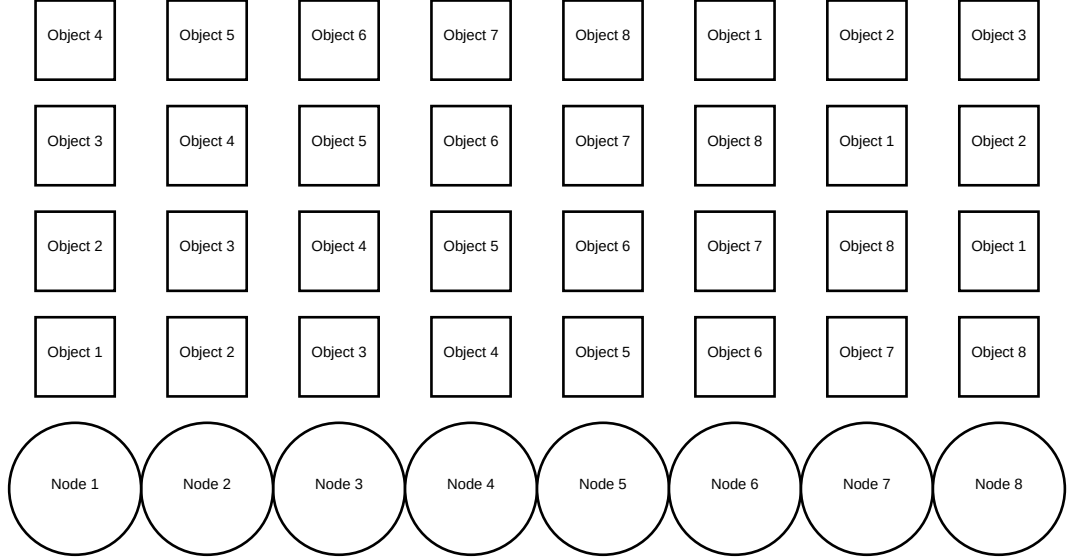


Fig. 3.4: Illustrative example of predecessor allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.

The predecessor placement pattern attempts to map replicas of a key onto the predecessors of the owner of that key. Because queries are routed around the ring clockwise towards the node responsible for them, a lookup for one node is frequently routed through one of its predecessors. Placing replicas on the lookup path for a node can help reduce fetch latency if used with the recursive fetch algorithm - which we will introduce later in this chapter.

Block Allocation

$$h(m, d) = (d - ((d \bmod \frac{K \cdot R_{MAX}}{N}) + (d \bmod \frac{K}{N}) + (m \cdot \frac{K}{N})) \bmod K$$

Block allocation attempts to establish a single core replica group for a set of nodes. If an object is stored on nodes A and B, then all objects on node B will be stored on node A, and vice versa. These set of nodes form an *equivalence class*, as they all store replicas of the same data. As will be seen in the next chapter, this policy provides a lower probability of data loss than other placement functions.

Block allocation achieves this equivalence class property by dividing the ring up into a number of ‘blocks’, each of size $\frac{K \cdot R_{MAX}}{N}$ keys. All nodes in this range of keys are in the same equivalence class. Block allocation provides most of the benefits of both successor and predecessor replication, since most nodes will have replicas placed on both successors and predecessors.

It is worth noting that the block allocation function is discontinuous in d , and the maintenance algorithm must be able to deal with this when mapping ranges of keys onto other nodes.

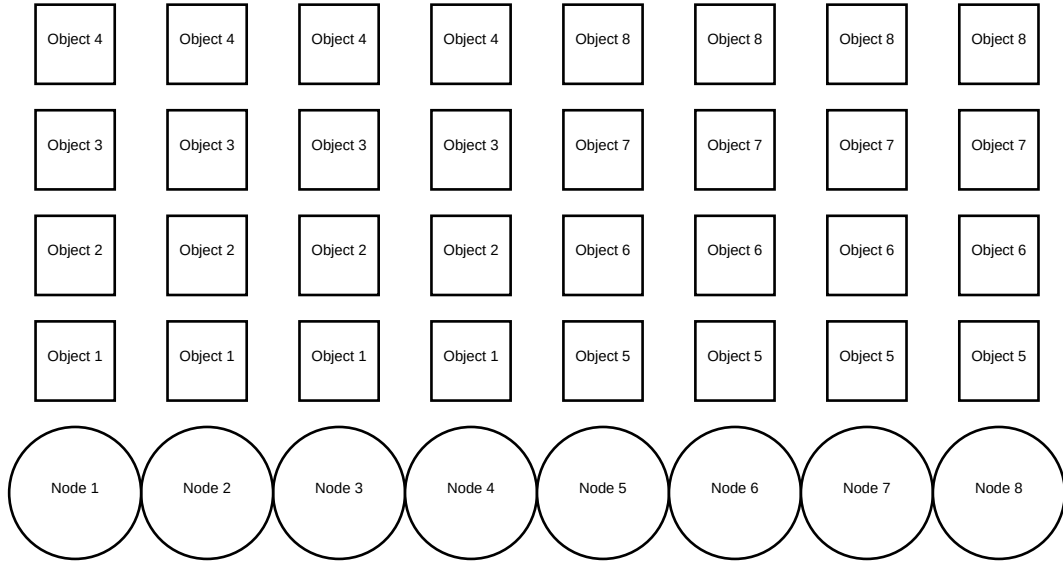


Fig. 3.5: Illustrative example of block allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.

Finger Allocation

$$h(m, d) = (d + (K/2^m)) \bmod K$$

Finger allocation, like successor allocation, takes advantage of the local routing information already maintained by the Chord algorithm. Chord maintains routing information about nodes at fractional distances around the ring, called fingers. Placing replicas on these finger nodes reduces the number of lookups that must be made to perform replica maintenance, and distributes replicas more evenly around the ring than successor, predecessor or block placement.

The distance between replica locations decreases exponentially with the replica index (m). Because of this, all replica indexes beyond a certain value are likely to collide. When using finger replication, the maximum number of replicas that can be placed is limited by the number of nodes in the system — we discuss this in more detail in Section 4.4.2.

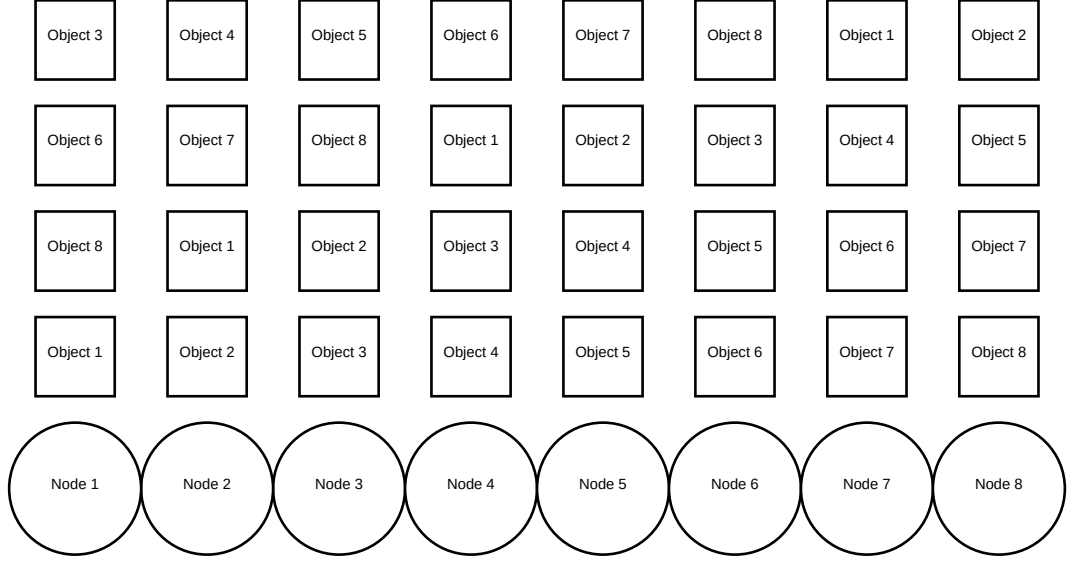


Fig. 3.6: Illustrative example of finger allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.

Symmetric Allocation

$$hopsiz = \frac{K}{R_{MIN}}$$

$$h(m, d) = (d + (m \cdot hopsiz)) \bmod K$$

Symmetric allocation is an allocation function proposed by Ghodsi et al. in [94]. It divides the keys in the system into $\frac{N}{R_{MIN}}$ equivalence classes, such that if key A is mapped to key B by the allocation function, then key B will also be mapped to key A. If, by dividing keys into equivalence classes, we can also divide nodes into equivalence classes, symmetric allocation will achieve the same high level of reliability as block allocation. However, although the nodes in figure 3.7 are divided into equivalence classes by symmetric allocation, we shall see that this is not the case in general.

Symmetric allocation places replicas at intervals of $\frac{K}{R_{MIN}}$ keys — the largest possible interval that allows R_{MIN} equally spaced replica locations. This makes allocation collisions very unlikely, but means that local routing state is not sufficient to find the core replica holders.

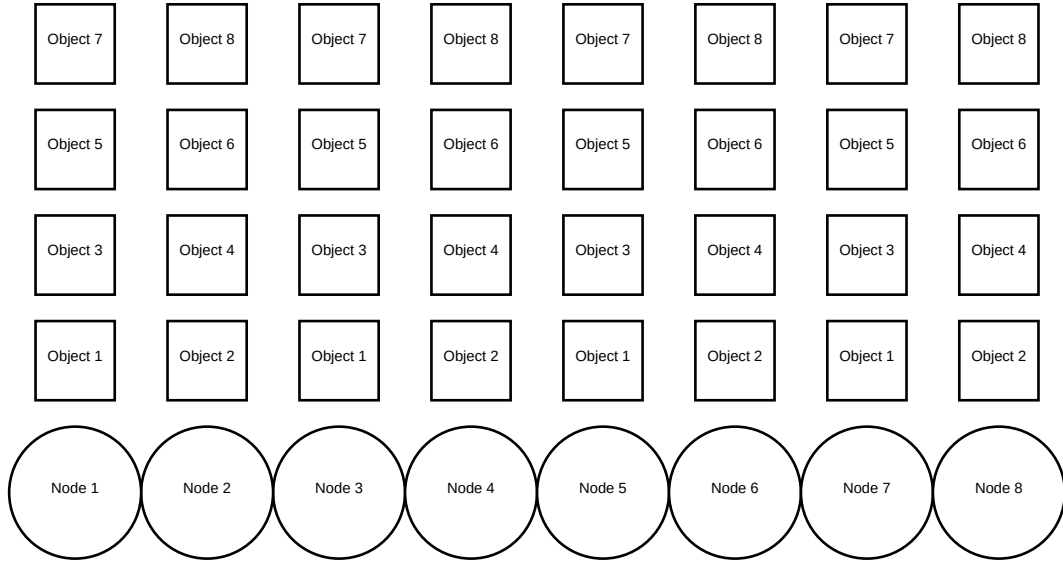


Fig. 3.7: Illustrative example of symmetric allocation. A system with 8 objects, stored on 8 nodes with a replication factor of four.

3.8.4 Dynamic Data Fetch and Update Algorithms

As with DHash, our maintenance algorithm leaves the system in its ideal state only immediately after it has run. Node churn quickly causes the system to become imperfect again, and some replica holders may not hold the replicas we seek. Once again we use a *fetch* algorithm to mask the complexity of dealing with these failures. The dynamic fetch algorithm needs to choose which replica indexes to lookup and in which order. In many cases, algorithm 8 will give good performance.

Algorithm 8 Dynamic Fetch for *key*

```

indexes  $\leftarrow [0 \dots R_{MAX}]$ 
item  $\leftarrow NULL$ 
while  $\neg item$  do
  index  $\leftarrow indexes.popRandom()$ 
  item  $\leftarrow recursiveGet(key, h(index, key))$ 
  if  $\neg item$  and index  $> R_{MIN}$  then
    indexes.removeRange(index, R_{MAX})
  end if
  if indexes = [] then
    indexes  $\leftarrow [0 \dots R_{MAX}]$ 
  end if
end while

```

This algorithm picks identifiers from the entire range of replica indexes (core and peripheral addresses). It picks the index at random, so that the load of providing an object is spread across all the replica holders. Once it has picked a location, it uses the recursive get procedure we will describe in the next section to collect the data. If none of the indices provides the object successfully, it simply tries the indices again. If a replica of the data

survives, maintenance will eventually restore it to the correct locations.

A potential issue with this algorithm is that we always include peripheral replica locations in our search. Peripheral replicas may not contain a replica, even when the system is in its ideal state, and so searching the peripheral range is less likely to result in success than searching the core range. We might instead choose to search the core range first, but this will place an increased load on the core range, and any replicas placed to avoid allocation collisions will be underused. Nevertheless, in situations where load balancing is not critical, shorter fetch times can be attained by searching the core replica locations before trying peripheral ones.

Another optimisation might be made if maintenance is infrequent. By Invariant 3, when the system is in its ideal state, if we find a peripheral replica is empty, there can be no replicas with higher indices. Algorithm 8 exploits this, eliminating the entire range of peripheral replicas with higher indexes if one peripheral replica is found empty. If churn has caused Invariant 3 to be violated, this may eliminate valid replicas, which can then only be found when the entire range of locations is considered again. This could result in poor performance, and simply removing the replica index known to be empty from the list of indices for consideration may be preferable.

Recursive data lookup

In order to increase performance when looking up data, we use algorithm 9 to perform *recursive gets*. This combines the Chord lookup and get messages; a node can either respond to the get message with the requested data, or forward the query towards the intended recipient. This allows any node on the lookup path of a request to return the data *preemptively*. Preemptively returning data avoids further lookup hops, and so reduces fetch latency. Predecessor placement specifically seeks to place replicas of a key on the lookup path for that key, and it is by allowing preemptive returns that this is able to reduce fetch latency.

Algorithm 9 Recursive get for *key* at *location* = $h(m, key)$ for some *m*

```

if self.keyrange.containsReplica(key) and self.store.contains(key) then
    return (self.store.get(key))
end if
if self.keyrange.contains(location) then
    return NULL
end if
next = self.closestPrecedingNode2(location)
return next.recursiveGet(key, location)

```

Recursive data lookup can interfere with load balancing, since some replicas holders may receive queries for a key more often than others. To prevent this, an overloaded node may choose to forward a recursive get request to a closer node, even if it holds a replica of the key being requested.

Dynamic Update Algorithm

When writing information into the system for the first time, we need to discover how many replica locations we need to use. The fastest way to do this is to look up the owners of all replica indices from 0 to R_{MAX} simultaneously. When the lookups complete, we select the R_{MIN} distinct nodes with the lowest replica indices, and place replicas of the data on these nodes.

When we wish to update information, we must contact the owners of all R_{MAX} replica locations. Thus to remove the chance that orphaned peripheral replica holders continue to serve stale data. Once all replica holders are discovered, consistent updates might be possible using some commit mechanism. We do not prescribe the method used to make the updates here.

Whether performing writes or updates, we may save bandwidth at the expense of increased latency by looking up locations sequentially until we have found the first R_{MIN} distinct nodes.

If a higher probability of a temporary inconsistency among peripheral replica holders is acceptable, we can improve performance by offering updates only until we encounter a certain number of empty peripheral replica locations, since, by invariant 3, it becomes increasingly unlikely that any further locations are occupied. This could dramatically improve performance if $R_{MAX} \gg R_{MIN}$.

3.9 Summary

We have introduced a variety of approaches to creating reliable data stores from distributed hash tables. Of the algorithms we have considered, the DHash and dynamic algorithms are perhaps the most interesting. They form a necessary part of any metadata based replication algorithm, and offer a superior service to soft-state based replication.

Many questions have intentionally been left unanswered in this chapter. For instance, how often should maintenance run in order to maintain invariants with high probability? Which placement function for dynamic replication should be used? How large does R_{MIN} need to be to achieve reliability? What values should be used for R_{MAX} ? How does dynamic replication perform in relation to DHash replication? In the next two chapters, we will use a combination of mathematical analysis and simulation to answer these questions.

4. CONFIGURATION OF RELIABLE STORAGE ALGORITHMS

4.1 Introduction

The DHash and dynamic replication algorithms we have described present a range of configuration parameters that we must set. These parameters include the placement function used, the number of replica locations available, and the frequency with which to run the maintenance algorithm. Additionally, dynamic replication requires that we set the R_{MIN} and R_{MAX} parameters appropriately so that we can recover from allocation collisions. In this chapter, we will attempt to gain an insight into how we should set these configuration parameters to achieve our goal of creating a reliable storage system in an efficient manner.

We will begin by exploring the effect of the replica placement pattern on system reliability. We will show that some placement patterns result in a more reliable system than other placement patterns. This is one of the most important and wide ranging contributions of this thesis, as it applies to any storage system in which multiple replicas of data are stored. We will then show how we can use this analysis to calculate how many replicas we need store to achieve reliability with a given placement pattern, and how system size affects reliability. We will suggest the frequency with which the replica maintenance algorithm should run. Finally, we present and analyse a new technique called keyspace smoothing that helps ensure the load on each node is evenly balanced, improving the performance of dynamic replication.

4.2 Assessing Reliability

Both the DHash and the dynamic replication algorithms require that we store replicas of data, and that we run a maintenance algorithm to replace any missing replicas. How many replicas are kept, and how often we run maintenance, determine the reliability of the system. The more replicas we keep, the less frequently maintenance is required to achieve a given level of reliability. This means that there is a trade-off between the bandwidth used by maintenance algorithm runs and the disk space used for storing replicas. In order to understand this trade-off, we need to understand how *reliable* a given system is under a given set of conditions.

The first step is to define exactly what we mean by reliability. One might define a system's reliability as the probability that all replicas of a given object are lost over some period of time. For a given object to be lost, all r of the nodes holding replicas of it must fail. If the probability of a node failing in a given time period is given by a uniform independent probability p , the probability of a given object being lost is simply p^r . We say an object is

Symbol	Definition
N	Number of nodes in the system.
r	Number of replicas of a data item stored.
p	The probability a given node will fail during one maintenance interval.
$Reliability(p, r, N)$	The probability a system of N nodes with failure probability p , storing r replicas of every object will not lose any data during a maintenance interval.
S	The number of maintenance intervals per system half-life.
R_{MIN}	Minimum permissable number of replica indices.
R_{MAX}	Maximum permissable number of replica indices.

Tab. 4.1: Index to symbols used in section 4.2

lost if all nodes storing replicas of that object fail.

For the purposes of providing reliable storage, however, we are concerned not with the probability of a particular object being lost, but with the probability that **any** of the objects stored in the system are lost. Under our definition, the reliability of a system will mean the probability that no objects stored in that system are lost.

Our choice to model node failures as independent will be valid in large decentralized systems where few machines share a common source of failure such as a power supply or network connection. In more centralized systems, where all nodes share a common hub or power source, correlated failures are more likely. Under such conditions, the single point of failure is likely to be the least reliable part of the system, and becomes a reliability bottleneck. Our analysis will provide an insight into the reliability that might be achieved in a system with no other reliability bottlenecks.

4.2.1 Modelling Reliability

To assess the reliability of a storage system, we model it as a sequence of N nodes, each of which has a uniform, independent failure probability p over the period of time between maintenance runs.

A failure is considered to be any event which means that a node is no longer able to participate fully in the system. The assumption of independence is valid in systems that are widely distributed, where resources such as network connectivity and power supply are not shared between nodes.

In Chord, each node is responsible for a key-range. We abstract this to arrive at a simplified model. We model the Chord ring as a sequence of N nodes with identifiers $[0, N)$. We assume that every node's key-range contains one or more objects. We model those objects which are under the control of a single node as a single data object, assigning it the same identifier as the node. We then use a hash function to place replicas, calculating the index at which replicas are located from that index. We also assume that exactly r copies of every object are stored.

Reliability with DHash, Successor, or Predecessor Placement

Under our model, DHash, successor and predecessor placement are equivalent. We will refer to this class of placement functions as the *successor class*. When data is arranged in this way, system failure corresponds to at least one sequence of r nodes failing before a data maintenance run can repair the data. As each node's failure probability over this period is p , we can represent a node's failure as a simple Bernoulli trial. The probability of a system failure during a single maintenance interval is thus the probability of finding a sequence of r failures in these Bernoulli trials.

This corresponds to the *Run Problem*, the probability of obtaining a sequence of r successful outcomes in N Bernoulli trials with probability of success p . The general solution to this problem is known, and can be expressed in terms of a generating function in s [95]. Note that although s appears in the generating function, it is a free variable. The solution to the run problem is expressed in terms of the first N coefficients of s^i in the polynomial expansion of the generating function, s does not appear in these coefficients.

$$F(p, r, s) = \frac{p^r s^r (1 - ps)}{1 - s + (1 - p)p^r s^{r+1}} \equiv \sum_{i=r}^{\infty} c_i^p s^i \quad (4.1)$$

$$RUN(p, r, N) = \sum_{i=r}^N c_i^p \quad (4.2)$$

$$Reliability(p, r, N) = 1 - RUN(p, r, N) \quad (4.3)$$

By summing the first N coefficients of s^i in the polynomial expansion of $F(p, r, s)$, we obtain the probability that the system fails. The complimentary probability is the reliability of the system.

Reliability with Block or Symmetric Placement

Under our reliability model, block and symmetric placement are also equivalent. We will call this class of placement functions the *block class*. For a system using block class placement to fail, all r nodes in some equivalence class must fail. As equivalence classes correspond to disjoint sets of nodes, the failure of one equivalence class is independent of the failure of any of the others. As each equivalence class fails with probability p^r , the probability that at least one of the $\frac{N}{r}$, distinct equivalence classes will fail¹ is given by:

$$Reliability(p, r, n) = (1 - p^r)^{N/r} \quad (4.4)$$

¹ We assume here $n \bmod r = 0$. This means there are a whole number of equivalence classes.

4.2.2 Validating Our Model

Our model made several assumptions – namely that nodes are equally spaced, and that the same number of replicas are stored of every object. For block class systems, we also assumed that the allocation function divides the whole system into a number of equally sized equivalence classes – that $n \bmod r = 0$. These assumptions allowed us to develop simple mathematical models, but are not true of Chord systems in general, where nodes are randomly spread throughout the keyspace, and allocation collisions can occur. There is clearly a need to validate our model works in the environment of real Chord systems.

To do this, we ran a simulation of a realistic Chord ring, where nodes are given random identifiers. We chose to simulate a ring of 500 nodes, into which we placed objects such that every node owns at least one object. We then placed replicas of these objects using the various allocation schemes we have proposed. We vary R_{MIN} between 3 and 20. The value of R_{MAX} was set for a probability of allocation-collisions of 0.01 using the method we will give in 4.4.1.

We ran 1000 such simulations for every permutation of replication factor and placement algorithm. We then mark each node in each system as failed with $p = 0.5$ and record the proportion of the 1000 systems in which no object is lost.

In figure 4.1, the results are compared with our model’s predictions for $N = 500$, $p = 0.5$, and the appropriate value of r . We can see that our successor model fits both successor and DHash placement well. The block class reliability predictions are also a good match for actual block placement reliability.

Block placement, in situations where $n \bmod r \neq 0$, maps more replicas to some nodes than others. If a ‘block’ happens to contain more than R_{MIN} nodes, these extra nodes are excluded from the block, and no replicas are placed on them. This effectively reduces the size of the system, and explains why the simulated reliability is, in some cases, better than our model – the failure of these empty nodes does not cause the loss of any data. While this is clearly not optimal from a load balancing point of view, it does mean the system achieves a very high reliability.

Symmetric placement, however, does not achieve the level of reliability our model predicts. This is because symmetric placement does not divide the nodes in real Chord systems into equivalence classes. Recall that an equivalence class is a set of nodes such that if node A and B are in the same equivalence class, then they store exactly the same data. Symmetric placement appears to divide nodes into equivalence classes in our simplified model where every node owns only a single key. This apparent division into equivalence classes is shown in examples in [94], and in figure 3.7. In real systems, however, a Chord node owns not a single key, but a range of keys. When this key-range is mapped to its image under symmetric allocation, the image range may be split between several nodes. Because of this, the equivalence class property is not established, and performance does not follow that predicted by our model. The actual reliability appears to be slightly better than that of successor replication – suggesting we might use this model to provide conservative estimates of the reliability of symmetric placement. A more accurate model for the performance of

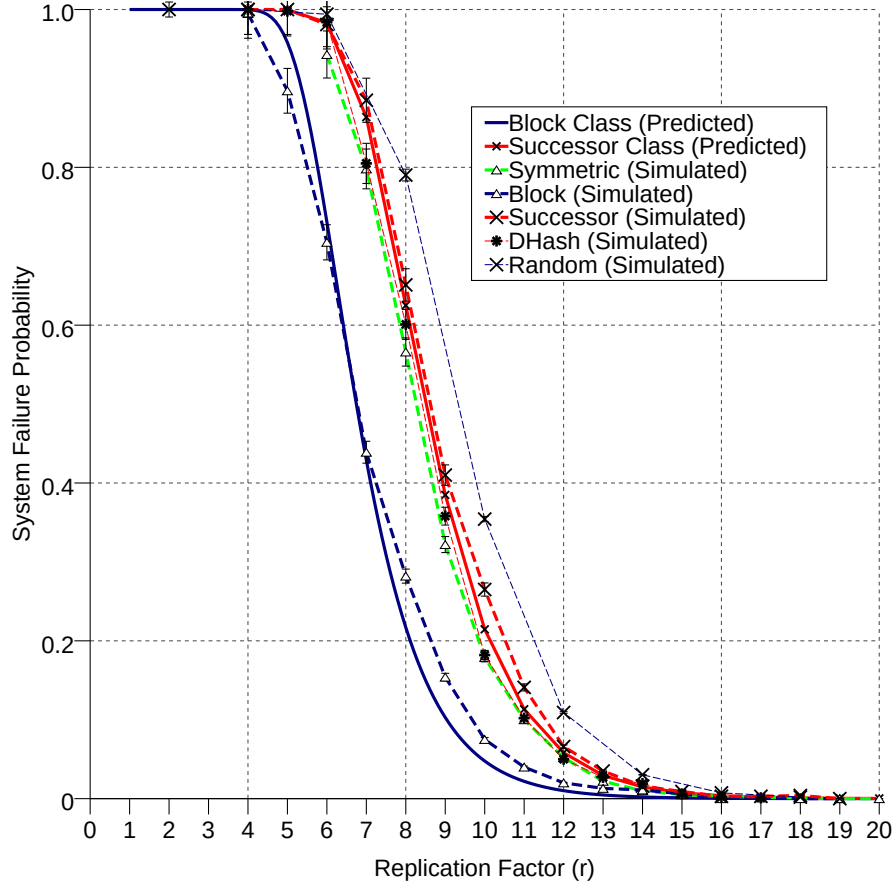


Fig. 4.1: Reliability of simulated Chord systems as compared to our models. Models are shown as solid lines, simulations are dashed, with error bars giving 95% confidence intervals. For the simulations, the x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.99$ column in Table 4.4.1. For comparison, a simulation in which data items are placed according to a pseudo-random distribution function is also shown.

symmetric placement might be an interesting area for further research.

4.3 Achieving Reliability

We wish to investigate how many replicas r we need store in a system with size N and with node failure probability p , in order that the system is reliable. By using the models from the previous section, we can calculate how reliable a block or successor class system with any given parameter set is. To solve equation 4.1, a computer algebra system (Mathematica), was used.

In this section, we will show which values of N , r , and p lead to reliable systems. We have defined a reliable system as a system which has a failure probability of less than 10^{-6} between maintenance intervals. Although this definition may seem somewhat arbitrary, it suffices to reveal the relationship between the number of replicas, the size of the system, and

the difficulty of achieving reliability in that system

4.3.1 Comparing Block Placement to Successor Class Placement

Having developed models for the reliability of block and successor class systems, we can compare the reliability these systems obtain with a given number of nodes and replicas. To do this, we vary the number of replicas and nodes, and find the *maximum acceptable node failure probability* p , that, with a given number of nodes and replicas, provides a system with overall failure probability 10^{-6} . If the node failure probability is higher than this value, the system will not be reliable.

We will first examine the impact of increasing system size on the critical node failure probability. We set the replication factor at eight, and calculate the critical node failure probability for various system sizes. The results are presented in figure 4.2. It is immediately clear that the block algorithm is more reliable. For a given level of per-node reliability, a larger reliable storage system can be created when using block placement. The impact of the difference in reliability on system scalability is pronounced. For instance, with a node reliability of $p = 0.1$, block placement allows us to scale to a system of up to 800 nodes, whereas successor systems become unreliable with more than 100 nodes. We can also see that, as we might have expected, a larger system (with an increased number of points of failure) requires more reliable nodes in order to offer the same level of system reliability.

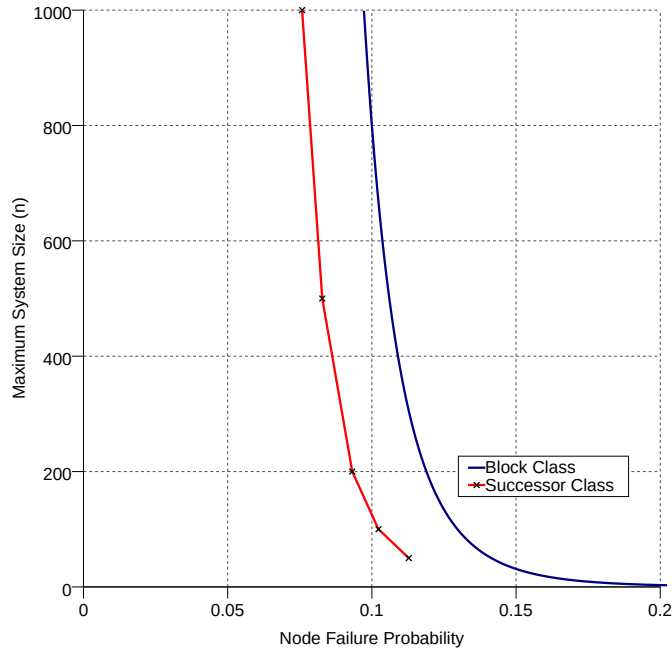


Fig. 4.2: Maximum system size for a system failure probability of $< 10^{-6}$, for varying node failure probabilities. These figures are for a replication factor of 8.

We can also examine the impact of node failure probability on the number of replicas required to achieve a reliable system. From figure 4.3, it is clear that the block algorithm

is more reliable. It is also clear that node reliability has a significant effect on the number of replicas we need to store. If we store 14 replicas using block placement, we can tolerate node failure probabilities of 28%, compared to less 5% if we use only 6 replicas. The results confirm what we might intuitively suspect – that by increasing the replication factor, we build a reliable system from less reliable nodes. Over this range of replication factors, there is an almost linear relationship between critical node failure probability, and replication factor. If we can run maintenance often enough that node failure probability between maintenance is less than 0.25, we will be able to calculate the necessary replication factor to provide reliable storage.

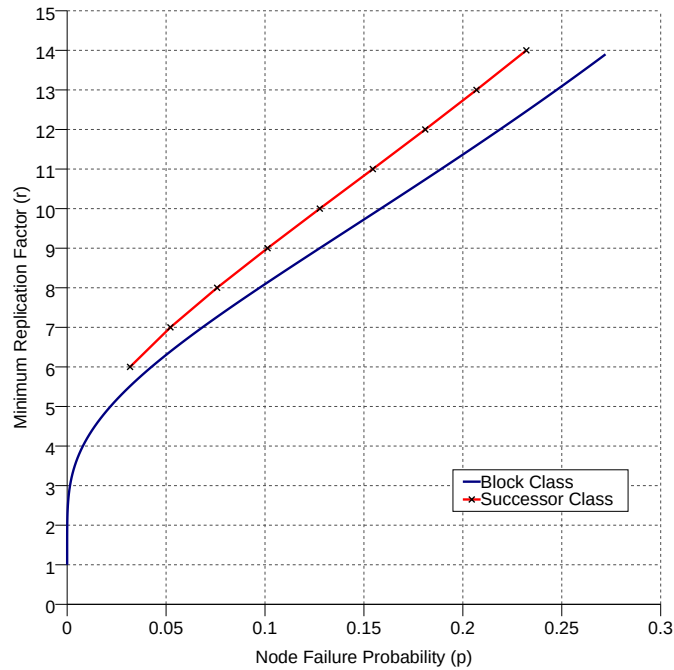


Fig. 4.3: Minimum replication factor for a system failure probability of $< 10^{-6}$ against node failure probability. These figures are for a 1000 node system.

4.3.2 Finger Placement

We have so far compared the successor class algorithms to block placement, and shown that symmetric replication results in reliability similar to that of successor class placement, but have not considered finger placement. As we do not have a closed form for the reliability of a system using finger placement, we will use simulation to compare this to other patterns. Our model consists of a 500-node network, in which 250 nodes are marked as failing. For a given value of N or r , we produced 1000 sample networks, placed data appropriately, and recorded the proportion of these networks which have failed (that is, in which data loss occurred). This is then used as an estimate of the failure probability. Figure 4.4 shows the probability of data loss for finger, block, and successor allocation.

From this plot, it is clear that finger placement is significantly less reliable than other data placement algorithms.

Interestingly, when we included a pseudo-random placement function in these simulations, it produced similar system failure rates to the finger placement system. We speculate that this is because they each exhibit the property that if two nodes both store a replica of a given object (they *share* that object), it is unlikely that they share any other object. This is in contrast to the equivalence classes established by block allocation, where two nodes which share one object share all other objects.

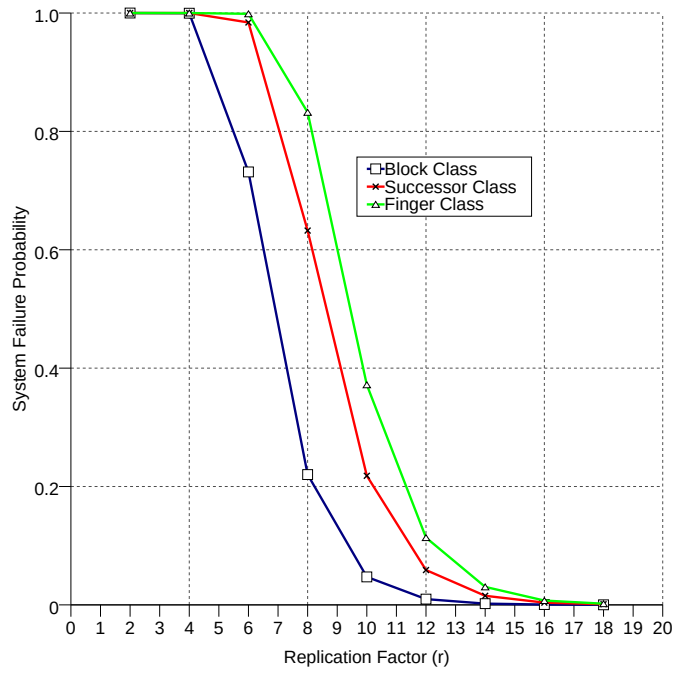


Fig. 4.4: Probability of system failure for three allocation functions in a 500 node system where 50% of nodes fail, for various replication factors.

4.3.3 Setting Maintenance Frequency

In the sections above, we have presented results in terms of the probability of node failure between maintenance algorithm runs. If we reduce the interval between maintenance algorithm runs, we reduce the probability that a failure occurs during this interval. As long as there is sufficient bandwidth available to perform maintenance at the required frequency, this may be a viable way to increase the reliability of the system.

We will model the rate at which maintenance runs independently of the rate at which failures occur by considering the number of times we run our maintenance algorithm in one *half-life*. The general definition of the half-life of a distributed system is the minimum of either the time taken for half the nodes in the system to fail, or for the time taken for new nodes joining the system to cause size to double. We will consider steady state systems,

where nodes join and leave at the same rate. In such systems, the half-life is the time period after which we expect half the nodes in the system to have joined since the beginning of that period.

In order to reduce the node failure probability over one maintenance interval, we increase the number of maintenance intervals per half-life (S). In our previous equations, p becomes $\frac{1}{2S}$. For instance if $S = 2$, then $p = 0.25$, and we will run two maintenance intervals per half-life. However, the system may fail during either of these intervals. In general, if a system with N nodes, which fail with probability p storing r replicas of each object survives one maintenance interval with probability $Reliability(p, r, N)$, it will survive the entire half life with probability $\left[Reliability(\frac{1}{2S}, r, N)\right]^S$.

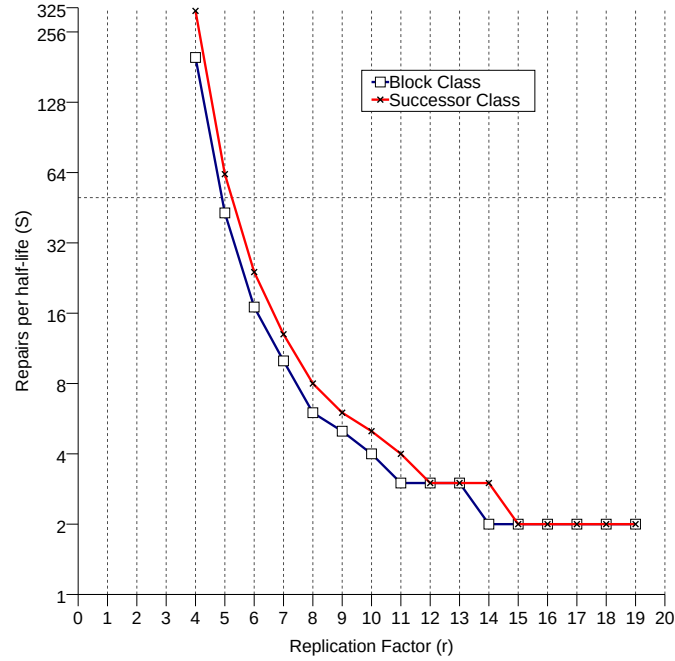


Fig. 4.5: Minimum maintenance frequency(in maintenance runs per half-life) necessary to maintain a 500 node system with $FAIL(N, R, S) < 10^{-6}$, versus replication factor.

We are interested in obtaining settings for S and r such that we obtain a system which is reliable over the entire half-life. In figure 4.5, we show values of S and r that lead to a 500 node system enjoying a failure probability of 10^{-6} over an entire half-life. We can see that if we store a very low number of replicas, we must run maintenance very frequently, but that as we increase the number of replicas stored, this frequency drops off sharply. There is little benefit to increasing the number of replicas beyond 15 (14 for block placement), as by this point neither class of placement pattern requires more than two maintenance calls per half-life.

4.4 Setting R_{MAX} and R_{MIN} To Allow Recovery From Allocation Collisions

We have noted that when using dynamic placement, allocation collisions can occur when the allocation function maps two replicas into the keyspace of the same node. Each allocation collision reduces the number of nodes in the core replica group, reducing reliability.

The core maintenance algorithm keeps track of which nodes have been allocated replicas from which key ranges. If a node is allocated more than one replica of a given object, core maintenance instead places a replica of that object on a node in the peripheral group. The more allocation collisions occur, the more objects will be placed on peripheral nodes. The number of peripheral replica locations available is limited by R_{MAX} , and so we must choose a value of R_{MAX} so that sufficient peripheral locations are available. We could simply select a very large value of R_{MAX} , ensuring sufficient peripheral replica locations are available. However, this has undesirable side-effects – when updating objects, all peripheral locations must be checked for old copies, and this operation will be very slow if R_{MAX} is large. To choose the lowest appropriate value for R_{MAX} , we need to analyse the probability that allocation collisions occur.

4.4.1 Successor, Block, and Predecessor Allocation

Symbol	Definition
K	Keyspace size.
h	Distance between adjacent replica locations (in multiples of $\frac{K}{N}$).
x	Random Variable: Distance between two adjacent nodes.
X	Distribution of x .
c	Random Variable: Number of distinct nodes found when searching R_{MAX} adjacent replica locations.
C	Distribution of c .
$h(m, d)$	An allocation function that maps a data item with key d and replica index d to a replica location $h(m, d)$.

Tab. 4.2: Index to symbols used in section 4.4

The choice of R_{MAX} will depend on the allocation function being used. Under successor, block, and predecessor placement algorithms, different replica locations are closely spaced in the keyspace. This means allocation collisions are more likely than under symmetric and finger replication, where replica locations are well-spaced throughout the keyspace, and two different replica locations are very unlikely to be owned by the same node.

Chord specifies that nodes are given randomly chosen identifiers. If N nodes are uniformly distributed throughout a keyspace of size K , the probability of a node having a given ID is $\frac{N}{K}$. We will refer to the numerical difference between one node's ID and the ID of its immediate successor as the *distance*, x , between those two nodes. We can represent the

distribution of distances, between two adjacent nodes as a geometrically distributed random variable X with parameter $\frac{N}{K}$. Thus, the probability of observing a distance x between two adjacent nodes is given by:

$$P(X = x) = \left(1 - \frac{N}{K}\right) \left(\frac{N}{K}\right)^x \quad (4.5)$$

We can use this to find the probability that an allocation collision occurs. An allocation collision occurs if two adjacent replica locations are owned by the same node. For this to happen, no nodes must exist in the x keys between the replica locations. The expression for this probability is given below – because the geometric distribution is memoryless, this is the same for all keys:

$$P(X > x) = 1 - P(X \leq x) = \left(1 - \frac{N}{K}\right)^{x+1} \quad (4.6)$$

As we would expect, the probability that an allocation collision occurs drops as x , the distance between replica locations, rises. If we express the distance between replica locations as a proportion (h) of average key space size, $x = \frac{hK}{N}$, the probability an allocation collision occurs is:

$$P(\text{collision}) = \left(1 - \frac{N}{K}\right)^{\frac{hK}{N}+1} \quad (4.7)$$

Since, for all practical systems, $K \gg N$, a good approximation of the probability of a collision in any system is given by taking limit of this equation as $\frac{N}{K} \rightarrow 0$:

$$P(\text{collision}) \approx \lim_{\frac{N}{K} \rightarrow 0} \left(1 - \frac{N}{K}\right)^{\frac{hK}{N}+1} = \frac{1}{e^h} \quad (4.8)$$

When we use successor class or block allocation functions, the distance between replica locations is $\frac{K}{N}$, i.e. $h = 1$, so $P(\text{collision}) = e^{-1}$. Now that we know the probability that a replica location ‘collides’ with an adjacent location, we can find the distribution of the number of non-colliding replica locations (i.e. locations belonging to distinct nodes) we expect to find when we try a total of R_{MAX} locations (i.e. $h(1, d) \dots h(R_{MAX}, d)$). This is a binomial distribution, with $n = R_{MAX}$, and $p = e^{-1}$. Let the random variable C be the number of collisions with R_{MAX} locations, then, using the following equation for the CDF of a binomial distribution, we have:

$$P(C \leq c) = I_{(1-e^{-1})}(R_{MAX} - k, k + 1)$$

R	R_{MAX} for $\pi = 0.95$	R_{MAX} for $\pi = 0.99$	R_{MAX} for $\pi = 0.9999$
1	3	5	10
2	6	8	13
3	8	10	16
4	10	12	18
5	12	14	21
6	14	16	23
7	16	18	25
8	18	20	27
9	19	22	30
10	21	24	32
11	23	26	34
12	25	28	36
13	27	30	38
14	29	32	40
15	31	34	42

Tab. 4.3: Values of R_{MAX} required so that R nodes are found in a keyspace interval of length $R_{MAX} \frac{K}{N}$ with probability π , for $\pi = (0.95, 0.99, 0.9999)$ and $R \in [2, 15]$

Here I is the regularised incomplete beta function. We would like to use this to find the value of R_{MAX} such that, even with collisions, we can store at least R replicas on distinct nodes. The actual number of allocation collisions that can occur is unbounded, but large numbers of allocation collisions are unlikely. We can choose values of R_{MAX} such that we have an arbitrary confidence level that less than $R_{MAX} - R$ collisions occur, and we are able to place replicas on R distinct nodes. We will call this confidence level π . For given values of R_{MAX} and R , π is the probability that we have more than $R_{MAX} - R$ collisions, that is, $P(C > R_{MAX} - R)$.

The value of R_{MAX} we choose depends on the confidence level we are willing to accept. The table below shows values of R and R_{MAX} such that π is 0.95, 0.99, and 0.9999, respectively. Note that a value of R means a total of $R + 1$ copies of an object – one original, at the location $h(0, d)$ and R replicas, at R locations from the range $h(1, d) \dots h(R_{MAX}, d)$.

4.4.2 Finger Allocation

When using finger allocation, the distance between replica locations decreases exponentially with the replica index, and allocation collisions are very likely to occur for large replica indexes. This means that the number of replicas that can be stored is limited by the number of nodes in the system – large numbers of nodes are required so that a node is likely to exist in the small intervals between replica locations.

We recommend setting R_{MAX} to $2\lceil \log_2(N) \rceil$, as larger replica indices give a distance between replica locations of less than K/N^2 , and so the probability of allocation collision is greater than $e^{-1/N}$. With such low odds of successfully finding an extra replica holder, searching further locations would be inefficient.

N	50	100	200	500	1000
R_{MIN}	3	4	5	6	7

Tab. 4.4: Values of R_{MIN} that may be placed in an N node system using Finger Allocation with a 99.9% confidence level of recovering from allocation collisions

When using this value of R_{MAX} , we can calculate the probability of an allocation collision occurring for each replica location, and so obtain the probability that some combination of at least R locations do not collide. In table 4.4.2 we show the maximum values of R_{MIN} that can be placed in an N node system such that the probability we are able to recover from allocation collisions is 99.9%. Based on this, we would suggest that R_{MIN} must be set to less than $\lceil \log_2(N) - 3 \rceil$.

4.4.3 Symmetric Allocation

When using symmetric allocation, all replicas are spaced by $\frac{K}{R}$ keys. Using Equation 4.8, and setting $h = \frac{N}{R}$, the probability of an allocation collision is $e^{-\frac{N}{R}}$.

This means R locations will be collision free with probability $\left(1 - e^{-\frac{N}{R}}\right)^R$. For large N and realistic values of R , this probability is close to one – allocation collisions are very unlikely.

4.5 Keyspace Smoothing

Symbol	Definition
l	The <i>smoothing factor</i> .
Q	The number of distinct identifiers we allow a node to choose from.
q	The index of the identifier assumed by a given node.

Tab. 4.5: Index to symbols used in section 4.5

Chord specifies that nodes are assigned identifiers at random. This can lead to a node being given an identifier very close to that of another node in the system. This is undesirable for two reasons; firstly, that node will not be responsible for storing many keys, and so may be underused. Secondly, it leads to a large variation in keyspace sizes, which leads to the need for large values of R_{MAX} .

By assigning nodes identifiers in a more controlled way, we can reduce the variation in keyspace size. Allowing any node to choose an arbitrary identifier, however, creates security issues – many DHT security mechanisms rely on the ability to verify an ID belongs to a node with a specific IP address.

We can however, allow each IP address to be associated with a small number, Q , of possible Chord identifiers, using a relationship such as: $id = \text{hash}(IPaddress + q)$ where

q may be a value between zero and Q . With such a mechanism, we may allow nodes to reject identifiers which would lead to a small keyspace, whilst also allowing for other nodes to verify that a node holds a valid identifier.

The pseudocode in Algorithm 10 presents an algorithm for choosing a suitable ID from the available set of Q in an efficient manner. A node tries each ID in turn, rejecting any that are too close to an existing node. In order to decide whether a node is too close, we use two pieces of information. Firstly, an estimate of the number of the nodes in the system is obtained – using the estimation mechanism given in Section 3.8.3 – this allows us to estimate the keyspace available for each node. Secondly, a *smoothing factor*, l , is used to decide how large a proportion of the average keyspace size we are willing to accept between the joining node and the surrounding nodes.

A value of $l = 0$ is equivalent to the naive Chord join algorithm. High values of l will give smaller variation in keyspace size, but need large values of Q , as most IDs will be rejected for being too close to other nodes. Lower values of l will accept a larger proportion of IDs, but will lead to a relatively large variation in keyspace size.

Algorithm 10 Join Algorithm with Keyspace Smoothing - Join(bootstrapNode, Q , l)

```

 $q \leftarrow 0$ 
while  $q \leq Q$  do
     $id \leftarrow \text{hash}(IP + q)$ 
     $successors \leftarrow \text{findSuccessors}(id)$ 
     $sizeEstimate \leftarrow \text{estimateSize}(successors)$ 
     $minSpace \leftarrow l \cdot (KeyspaceSize / sizeEstimate)$ 
     $predSpace \leftarrow \text{distance}(successors[0].predecessor.id, id)$ 
     $succSpace \leftarrow \text{distance}(id, successors[0].id)$ 
    if  $predSpace > minSpace$  and  $succSpace > minSpace$  then
         $\text{join}(id)$ 
    return
    end if
     $q \leftarrow q + 1$ 
end while
 $\text{join}(id)$ 

```

In order to assess the performance of the keyspace smoothing algorithm, and find suitable values for the parameters l and Q , we simulated the use of the algorithm on a Chord ring, with keys between 0 and 2^{32} . During the *join phase* we allow the ring size to grow to 50,000 nodes. We then enter a *steady state phase*, during which nodes leave and join at the same rate.

This effect of join smoothing on the distribution of the number of keys between adjacent nodes (*keyspace sizes*) in the steady state phase can be seen in figure 4.6. The probability of a particular identifier having a node associated with it is $\frac{N}{K}$. We can think of the distance between two nodes as the number of consecutive empty keys between two nodes. We can then represent each key succeeding a node as a trial with success probability $\frac{N}{K}$, and the distance between two nodes as the number of trials before we have a success (i.e. find a node). The probability distribution of these distances is thus geometric, with parameter $\frac{N}{K}$.

With smoothing, the distribution of keypace sizes becomes bell-shaped when $l > 0$. The larger the value of l , the smaller the standard deviation of the keypace size distribution.

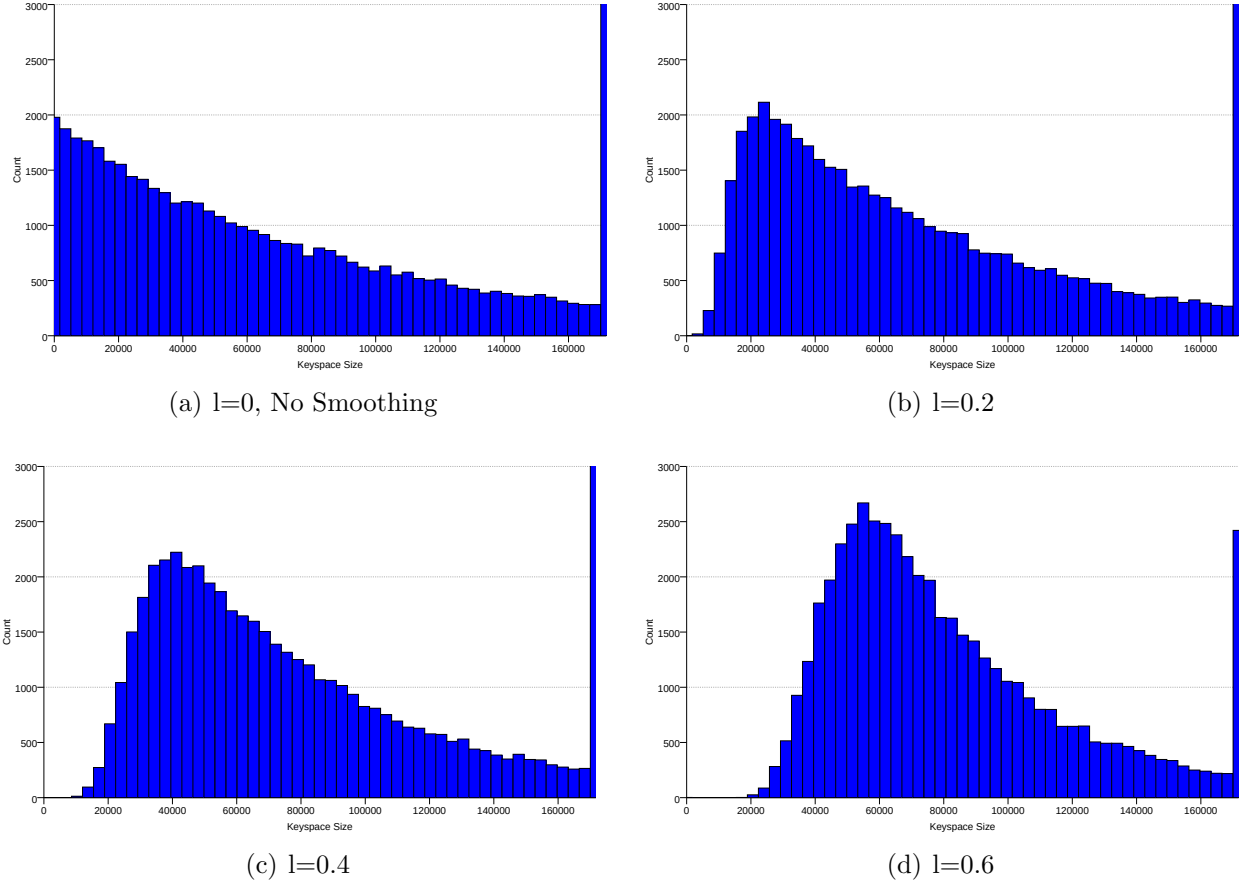


Fig. 4.6: Effect of smoothing on keypace distribution, with various l

With keypace smoothing enabled, it can be seen that some nodes still have keypace sizes far less than l times the average keypace size. This is because those nodes have over-estimated the size of the system, and are this willing to join the system with a small keypace.

The effect of different values of l on the number of different IDs that need to be tried before a node can join the system at an ‘acceptable’ position can be seen in figure 4.7. Smoothing values of 0.6 and less will work well with Q set to less than 20, but the number of IDs that need to be tested grows rapidly with larger smoothing factors, and these are likely to be impractical.

The simulation shows the behaviour of the algorithm is different in each phase, and the distribution of keypace sizes rapidly changes upon entering the steady state phase. Smoothing is more effective when the system is in the join phase than during the steady state phase. The standard deviation of the keypace size is plotted against l for both the steady-state system, and a system of the same size at the end of the join phase, in figure 4.8. It can be seen that during the joining phase, the keypace sizes are less variable than they are under a steady state system. It can also be seen that the effect of increasing l is less pronounced during the join phase.

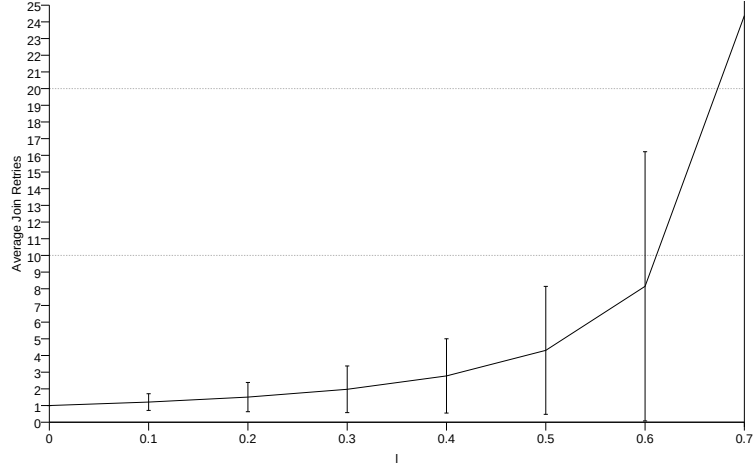


Fig. 4.7: The average number of join retries necessary to achieve a given smoothing factor. Error bars show one standard deviation

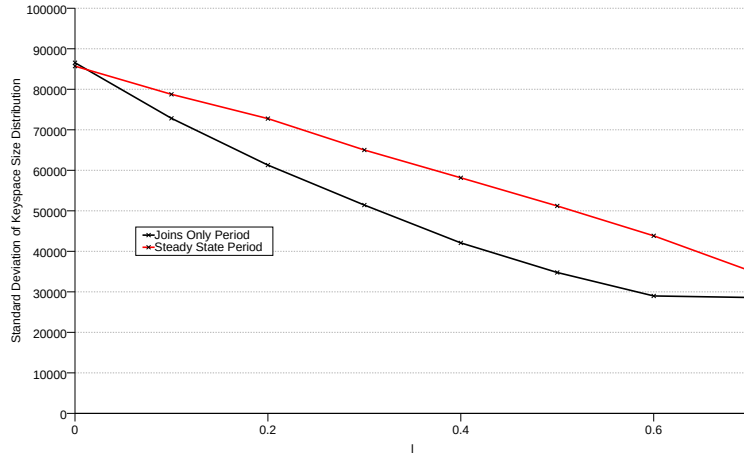


Fig. 4.8: The standard deviation of keyspace sizes versus smoothing factor. Join Period shows s.d. after 50,000 nodes join an initially empty ring of size 2^{32} . Steady state shows the same system after 100,000 churn events (node join followed by node departure).

Analytic calculation of the values of R_{MAX} appropriate for placing R replicas with a given confidence level is difficult when using node smoothing, as keyspace sizes no longer follow a standard distribution. We have used simulation to attempt to place R replicas at locations separated by intervals of K/N . We observe how many replica locations were necessary to place those replicas on R distinct nodes, for various R . In our simulations, we place 100,000 objects into a Chord ring with a smoothing factor $l = 0.5$, during the steady state phase.

We used these observations to pick values of R_{MAX} that would have resulted in 99% of our sample objects being placed on R distinct nodes. These can then be compared to the values of R_{MAX} we obtained for $\pi = 0.99$ in Section 4.4.1. In figure 4.9, we plot the values of R_{MAX} necessary with various smoothing factors and values of R . This shows that keyspace smoothing is an effective way of reducing the number of peripheral replica locations

necessary, and would help improve update performance.

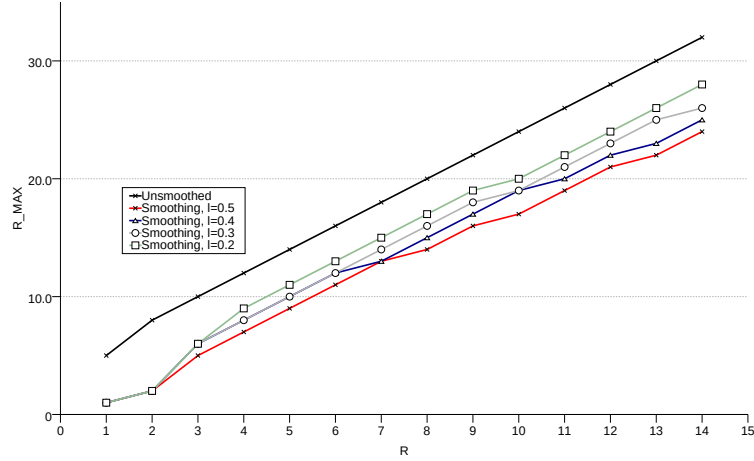


Fig. 4.9: The values of R_{MAX} necessary to achieve $\pi = 0.99$ for various smoothing factors and values of R .

4.6 Summary

In this chapter, we have examined the relationships between the configuration of a system and its reliability. We have shown that the block placement function achieves a higher system reliability than either the successor class or the finger placement strategy.

Relationships between the system size, the number of nodes in the system, and the maximum node failure rate that can be tolerated between maintenance intervals have been illustrated. We have provided quantitative methods that can be used to calculate how often maintenance needs to be run, or how many replicas to store, for block or successor class systems with a given size and half-life.

We have also investigated the settings for the R_{MAX} parameter of dynamic replication, and provided suggested values of that parameter in order to avoid allocation collisions. We have suggested a technique called node smoothing, that reduces the need to allow a wide range of replica locations, and improves the performance of dynamic replication.

5. PERFORMANCE EVALUATION OF RELIABLE STORAGE TECHNIQUES

5.1 *Introduction*

In this chapter, we will use a detailed simulation of Chord to quantify the impact of replication strategy on the performance of a reliable peer to peer storage system. Using results from the previous chapter, we will supply appropriate parameters to configure each replication strategy, and then assess their performance.

The two main metrics we will use to assess performance are fetch latency, and communications costs. Our simulations will show how each of these is influenced by factors including maintenance frequency, systems size, and replication factor.

Our simulation studies of replication algorithms represent a new contribution to the field as the first direct comparison between replication strategies. Quantitative relationships between maintenance costs and performance are observed, and we are able to show that dynamic replication outperforms DHash in most circumstances. We are also able to compare the performance of each placement function, quantifying differences in performance, and so allowing informed decisions on which function might be most appropriate for a given application.

5.2 *Our Simulation*

To assess quantitatively the performance and bandwidth usage of our storage techniques, a simulation was used. This avoids the difficulties implicit in managing large numbers of physical machines [96], as well as simplifying the creation of the reproducible and controlled environment necessary to perform meaningful comparisons. The simulation was written in Python using the open source SimPy[97] discrete-event simulation framework. Using SimPy provides several advantages over multithreaded simulation designs. We wish to simulate hundreds of multi-threaded nodes concurrently. Simulating each of these using full threads would have required a great deal of memory, as each thread requires its own stack. SimPy models concurrent processes using light-weight *generator functions*, rather than full threads, and so allowed the simulation of hundreds of nodes with a memory usage of just 100KB per node.

The simulation implements a message-level model of a Chord network running each of the replication algorithms described in chapter 3. Simulated nodes are able to pass messages

between one another, and there is a simulated delay between sending a message and its receipt. Network latency is assumed to be uniform, and one time unit in the simulation is equivalent to a single network hop. Computation is assumed to take zero time. We believe this is a reasonable simplification, since the lookup algorithms are simple, and likely to execute in less than a millisecond, whereas network delays in distributed grids are of the order of hundreds of milliseconds. Bandwidth is assumed to be unlimited. This is also a fair approximation for the very high speed links between grid data centres – typically of the order 10^9 bits per second.

The lookup and maintenance algorithms are implemented as described in Chapter 3. The Chord algorithm is implemented with a successor list length of ten nodes, and a finger table size of 12. We run the *Stabilization*, *Check-Predecessor* and *Fix-Fingers* Chord maintenance algorithms at regular intervals – 50 times between each node failure. R_{MAX} will be set using the values suggested for $\pi = 0.95$ in Chapter 4. The dynamic fetch algorithm is configured to check core replications before trying any peripheral locations.

The simulations first run for a ‘ramp-up’ phase, during which the nodes join the system. The nodes are given random identifiers throughout the keyspace. During the ramp-up phase, no nodes fail. Once a given number of nodes have joined, the value of N used by the allocation function is set, and the data is then placed onto nodes. One hundred data items per node are placed into the ring, with randomly chosen identifiers.

The simulation then enters a steady state phase where new nodes join and existing nodes fail at regular intervals. The simulation then runs for one half life, during which replica maintenance is also run at regular intervals – unless otherwise specified, maintenance runs eighty times in one half-life. During the steady-state phase, we issue 50,000 requests for items chosen at random from those placed in the ramp-up phase. The time taken to retrieve items is recorded. We also record how much data is sent across the network (excluding overheads such as TCP packet headers). We assigned messages sizes based on their content – an IP address is 4 bytes, a node identifier is 16 bytes, and the type of the message is encoded in one byte. When nodes attempt to synchronise their (key,object) databases, we calculate the bandwidth that would have been used if merkel tree hashing [90] were implemented, though our simulation does not actually implement tree hashing. Network traffic is categorised, so that we can distinguish between traffic generated by the data maintenance algorithm and traffic generated by the Chord maintenance algorithm. We also count the number of times data items are copied across the network.

In a real-world implementation of this system, the bandwidth used will generally be greater than that estimated by our model. Factors such as TCP packet headers, and any service/application layer overheads (such as SOAP headers or conversion to a CORBA standardized messaging format) will increase the size of messages. Our simulation is useful in modeling the trends in bandwidth usage that might be seen in such implementations.

Nodes detect communication failures using timeouts – if a message is not acknowledged within 3 time units (network hops) it is assumed to have been lost, and the recipient is assumed to have failed. When performing recursive lookups or fetches, the querying node will wait for $3\log(N)$ time units before deciding the query has failed somewhere along the

lookup path. A node will continue to retry lookups until they succeed – Chord maintenance will eventually repair the overlay and allow the lookup to succeed.

The identifiers assigned to nodes are randomly selected, as are the IDs of the objects placed, and which nodes fetch which objects. To illustrate how much variation this causes in the results, we run each simulation four times. We will show the averaged results of these four runs, together with the standard deviation of the result between these simulations.

5.3 Simulation Results

The simulation was run over a range of 120 different configurations of placement function, replication factor, and network size. Each of these configurations was run four times with a different random seed. We will now analyse the results of these simulations. We first examine fetch performance, and how it is influenced by each of the configuration parameters. We will then consider communication costs and load balancing.

5.3.1 Fetch Performance

Fetch performance is measured by the average *fetch latency*. Fetch latency is the time that elapses between a node issuing a request for an object, and receiving a replica of that object.

Fetch latency is influenced by the placement pattern, the size of the system, the number of replicas in the system, and how frequently maintenance is performed. We will begin by examining the effect of system size, noting the differences between different placement functions.

Fetch Performance versus System Size

Five different system sizes were tested, and their average fetch latency observed. The results are shown in figure 5.1. The relationship between system size and lookup performance is logarithmic. This result can be obtained from analysis of the Chord algorithm [38], and the fact this property is also displayed by our simulation helps verify its accuracy.

The relationship these simulations show between placement function and fetch latency is interesting. The best performance is offered by predecessor placement. Predecessor placement results in more preemptive returns than other placement patterns. This is because when we fetch an item, a particular replica location index is chosen. The fetch algorithm tries all locations from the core replica group before any locations in the peripheral group. In the case of predecessor placement, the peripheral replica group precedes the core replica group, and so the query is likely to pass through the peripheral group en-route to the node that was selected from the core group.

Under successor replication, the peripheral group follows the core group, and so the query is not likely to pass through the peripheral group enroute. This means that preemptive returns only happen when the replica location index chosen is one of the higher indices. The

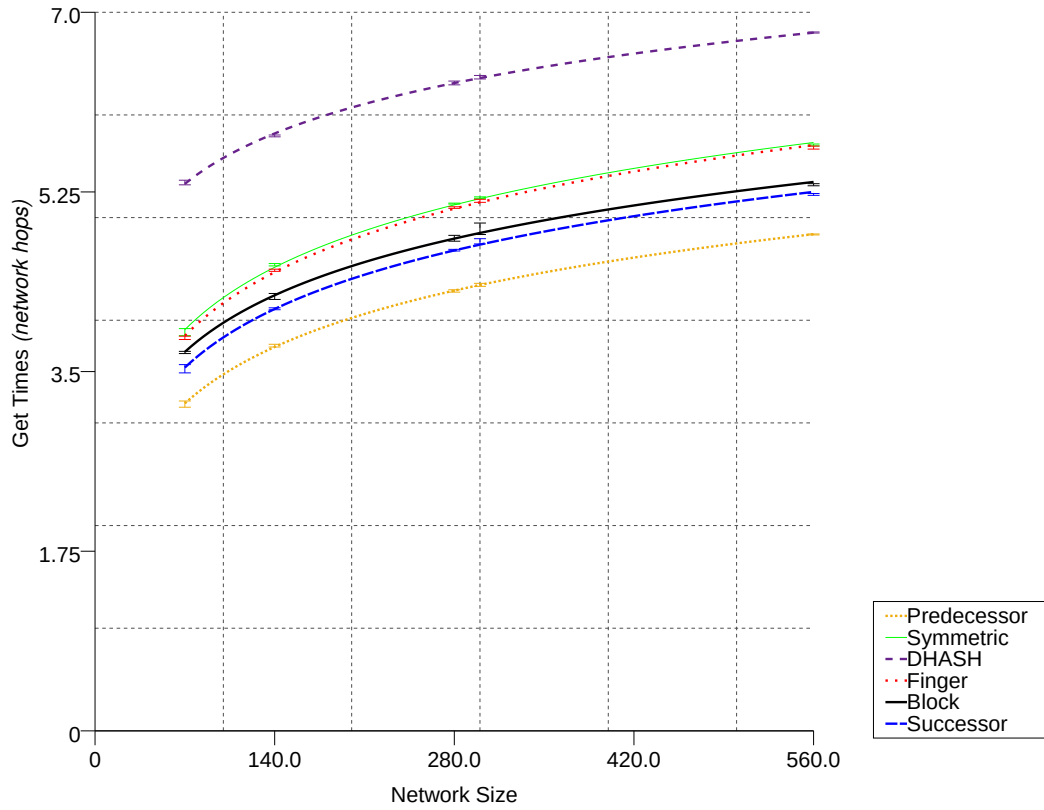


Fig. 5.1: Fetch latency of Chord systems with various placement functions plotted against system size. Error bars show 95% confidence intervals. A logarithmic fit between the data points is shown. These results are for $R = 3$ and 80 maintenance calls per half-life. For dynamic replication, $R_{MIN} = 3$ and $R_{MAX} = 8$.

replica locations with high indices are preceded by the locations with lower indices, and so a query for a high-index replica location is likely to be routed through a lower-index replica location.

Block placement is slower than successor placement because there can be discontinuities in the placement function – if the first replicas of an object are at the end of the block, subsequent replicas are ‘wrapped round’ to the beginning of the block, leaving a gap between the two ranges of locations. This means the chances of a request being routed through other replicas are reduced in comparison to successor replication.

Symmetric replication places replicas at well spaced locations, and a query is very unlikely to pass through another enroute. This means it produces the longest fetch times of all the dynamic functions.

Finger replication places replicas at locations that are closer together for larger replication indexes. This means there is a small chance of a preemptive return when high index locations are fetched, and so finger placement achieves very slightly faster average fetch times than symmetric placement.

DHash replication is slower than any dynamic function. This is because there is no

possibility of a preemptive return. In addition to this, the DHash fetch algorithm cannot choose which replica to fetch until it receives the successor list of the owner of the object it is fetching. Once it has this list, it chooses one of the owner's r successors, and must make an additional network round-trip to that node to fetch the object. This additional round-trip makes up most of the difference between DHash and the dynamic algorithms. We might eliminate it by allowing the owner of the object to select a replica on behalf of the requesting node, but this would pose security problems, as a malicious owner would then easily be able to subvert all requests for replicas of objects it owns by answering them itself. This would be hard to detect without making repeated requests.

Fetch Performance versus Maintenance Frequency

The frequency of maintenance also has an effect on the fetch latency, illustrated in figure 5.2, where we range the number of maintenance calls per half-life between 1 and 80, and record the impact on fetch latency. A low maintenance frequency means that a given replica location is less likely to actually contain the replicas it should. If a replica is *missing* from a given location, the fetch algorithm will try an alternative location, and so fetch time is increased.

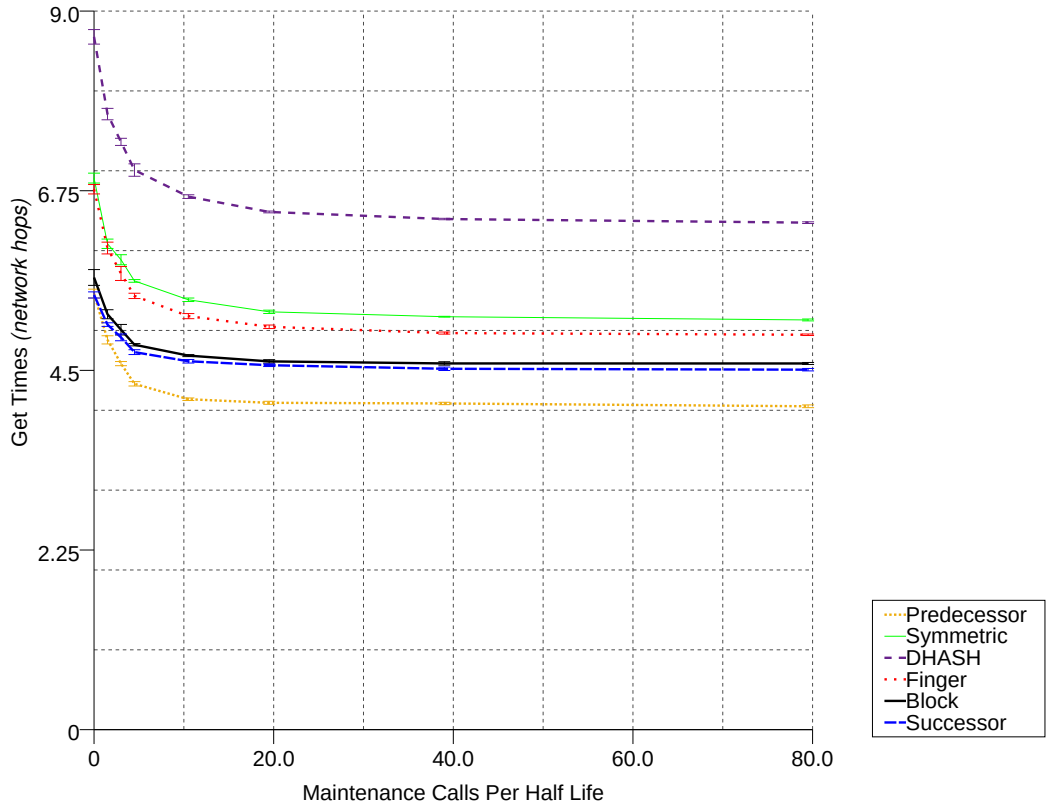


Fig. 5.2: Fetch latency of Chord systems with various placement functions plotted against maintenance frequency. Error bars show 95% confidence intervals. These results are obtained with $N=300$, $R_{MIN}=6$ and $R_{MAX} = 14$

All replication functions have a higher performance as replication frequency increases, and for all functions running maintenance twenty or more times per half-life results in near optimal performance. However, there is a significant difference in how each placement function is affected by low maintenance frequencies. Those maintenance functions in which preemptive returns are common are less affected by missing replicas. A fetch request for a missing replica may succeed if another replica is returned preemptively. For instance, when using predecessor placement, fetch latency increases by only 1.41 hops over the range of maintenance frequencies we simulate, whereas DHash fetch latency increases by 2.31 hops.

Fetch Performance versus Replication Factor

For those dynamic placement systems where preemptive returns are likely, larger replication factors lead to lower lookup times. We can see that the effect of increasing replication factor from two to seven on the fetch latency under symmetric placement is very slight (0.2 hops), whereas predecessor placement's fetch latency falls by nearly one hop.

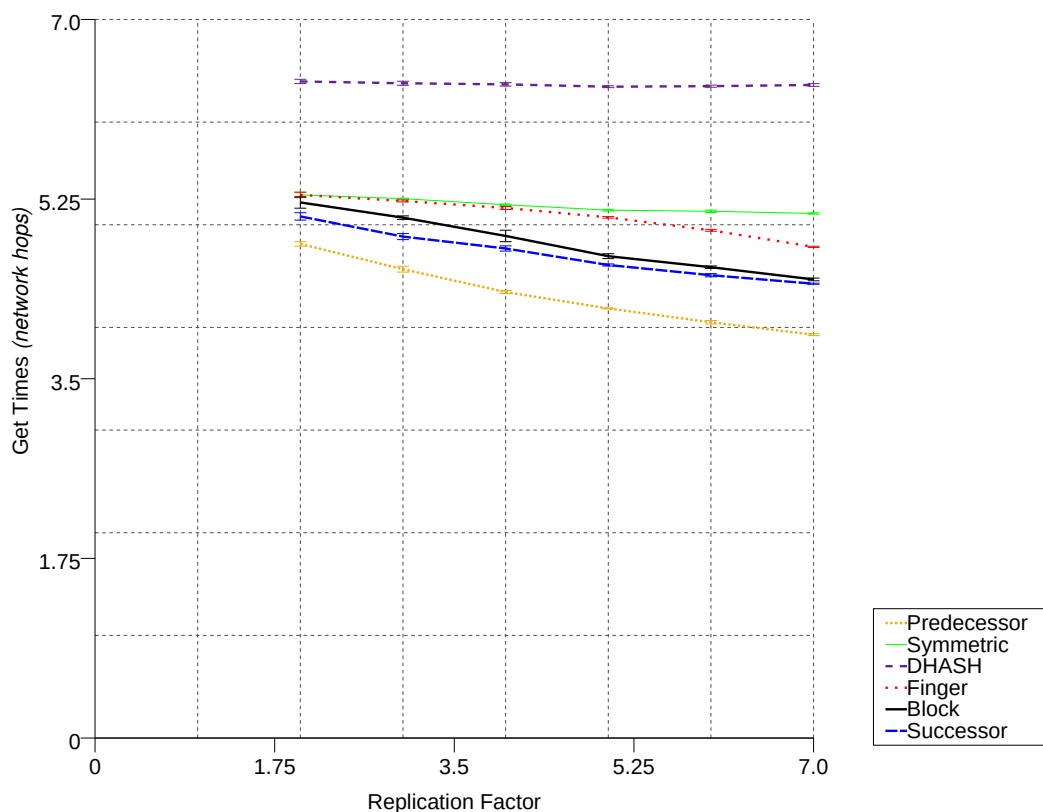


Fig. 5.3: Fetch latency of Chord systems with various placement functions plotted against replication factor. Error bars show 95% confidence intervals. These results are obtained with $N=300$ and 80 maintenance calls per half-life. The x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.95$ column in Table 4.4.1

For successor, predecessor and block placement, the effect of adding an additional replica on lookup latency becomes less pronounced as we increase the total number of replicas.

Finger replication places replicas closer and closer together as the replica index increases. Large replication factors lead to several replicas being placed in close proximity to one another, and so the chances of preemptive return are increased when larger replication factors are used. This is clear in this plot, which shows that for low replication factors, finger placement results in fetch latency similar to symmetric placement, but is significantly faster than symmetric replication when we use large replication factors.

Fetch Performance with Lookup Parallelism

Our fetch performance analysis so far has concentrated on the time taken between issuing a request for and receiving a copy of a single randomly chosen replica. We may also issue requests for all replicas simultaneously, accepting the first replica to be found. Because some replica locations may be found more quickly than others, this will reduce fetch times, at the cost of increasing bandwidth usage. The simulated parallel fetch times for various replication factors are shown in figure 5.4.

Symmetric replication provides excellent performance when used with parallel fetches. This is because it places replicas at well spaced intervals, so that all nodes are likely to have a replica within a few network hops of them. This is also the case, though to a lesser extent, with finger replication, which places all replicas in one half of the ring. The other placement functions cluster all replicas together, so that relatively few nodes are within a small number of hops of these replicas. This means that parallel fetches do not increase lookup performance to the same extent as the other dynamic placement functions. DHash is not able to use parallel lookups, as we must locate the owner of the item before locating any replicas.

5.3.2 Communication Costs

The communication costs incurred by each algorithm are evaluated quantitatively by our simulations. We have divided communications costs into two categories, both over the period of one half-life. The first category is data movement, and is the proportion of the total quantity of data stored in the system that is moved in one half-life. The second is overhead bandwidth, which is the total size of all messages exchanged in deciding where data should be stored (but not the costs of actually moving the data to the correct location). We will now analyse our results in each category.

Data Movement

Data movement is, in most systems, likely to be the chief communication cost in running a reliable peer to peer storage system. While there is a dramatic difference between the DHash and Dynamic algorithms, the quantity of data transmitted across the network over one half-life by either system is of the same order of magnitude of the total quantity of data stored. This is likely to dwarf overhead costs, which are of the order of megabytes per node.

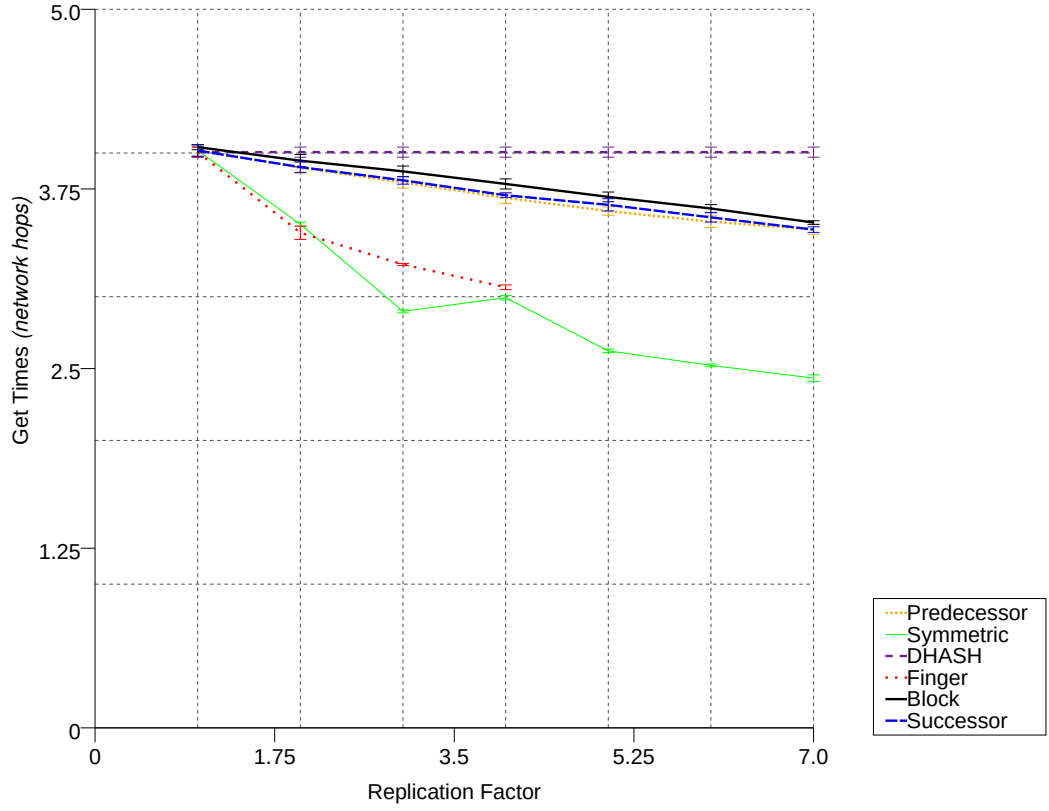


Fig. 5.4: Parallel fetch latency of Chord systems with various placement functions plotted against replication factor. Error bars show 95% confidence intervals. These results are obtained with $N=200$. The x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.95$ column in Table 4.4.1.

Figure 5.5 shows that DHash systems move approximately twice the quantity of data that dynamic systems do. This is because of the disruptive effect that a node joining or leaving the system has on the placement of replicas when using DHash. Consider a node joining halfway between two existing nodes under both systems. In figure 5.6, we illustrate this with a segment of a simple Chord system with a replication factor of 3. A new node with ID 25 arrives, and is inserted between nodes 20 and 30.

Under dynamic replication, this node inherits half of the replicas that had been mapped to node 30. This is approximately half the data stored on one node, on average.

Under DHash placement, node 25 becomes a part of node 10 and node 20's replica group, and receives a replica of all their data. It also receives half the data that was stored on node 30. In a system with $r = 3$, a node stores replicas for two other nodes, and the data it owns, and so node 25 receives, on average, $\frac{5}{6}$ ths of the data stored on an average node. In general, a newly joined DHash node inherits $\frac{2r-1}{2r}$ of the data stored on an average node. The net result is that nearly twice the volume of data is moved to restore the correct placement under DHash than under dynamic placement.

For both dynamic and DHash maintenance, the quantity of data moved increases with the maintenance frequency. This is because the maintenance calls are equally spaced throughout

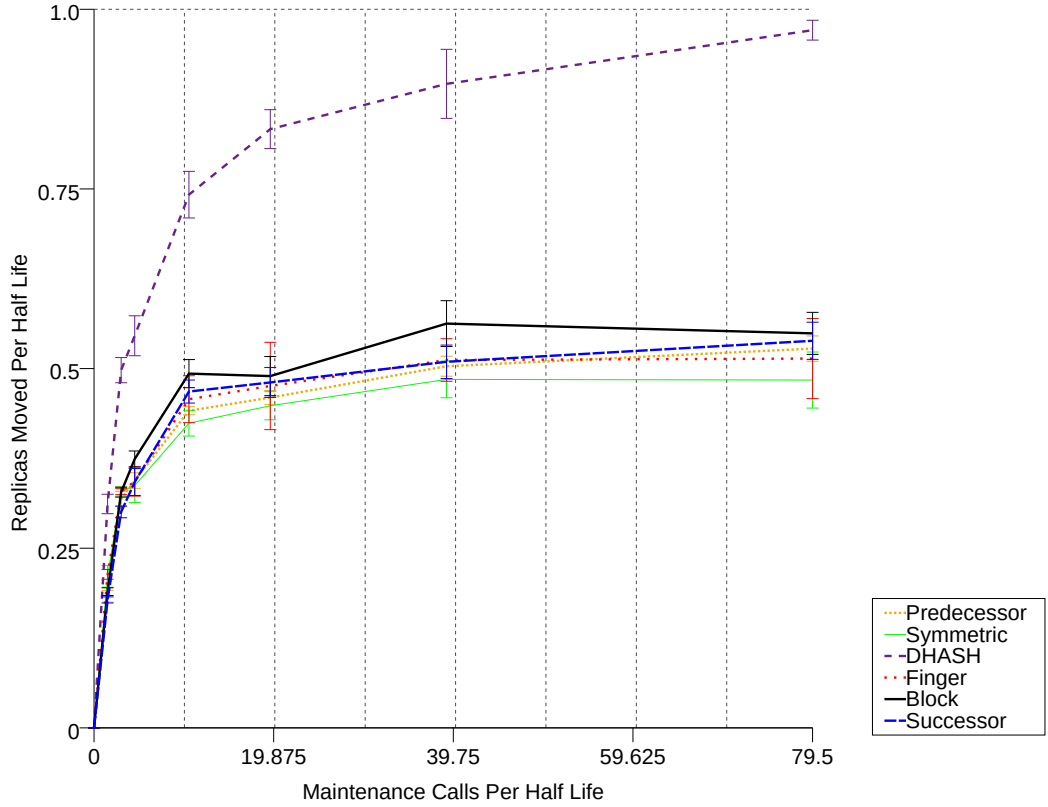


Fig. 5.5: Proportion of all stored data moved in one half-life by each algorithm with various maintenance frequencies. Error bars show 95% confidence intervals. These results are obtained with $N = 300$ and $R_{MIN} = 6$ and $R_{MAX} = 14$.

the half-life. This means that if maintenance runs only once, it runs half-way through the simulation, and failures that happen after this point are not repaired. The more frequently maintenance runs, the fewer failures are left unrepaired.

The sizeable errors (error bars show one standard deviation) on the data moved by dynamic functions makes distinguishing between them difficult. The results do suggest that symmetric replication moves slightly fewer replicas than other placement functions, and this may be due to the fact that since there are few allocation collisions under this placement functions, fewer peripheral replica holders are created, and so fewer replicas are moved due to peripheral maintenance.

Overhead Bandwidth

Figure 5.7 shows how much data is sent by an average node in running the data maintenance algorithm when maintenance is performed with various frequencies. The network communication overhead of the Dynamic Algorithms is significantly greater than DHash. This is as we would suspect, as DHash never needs to lookup other nodes to perform local maintenance – it is conducted using the local successor list. Dynamic algorithms must also perform peripheral maintenance, which DHash does not require.

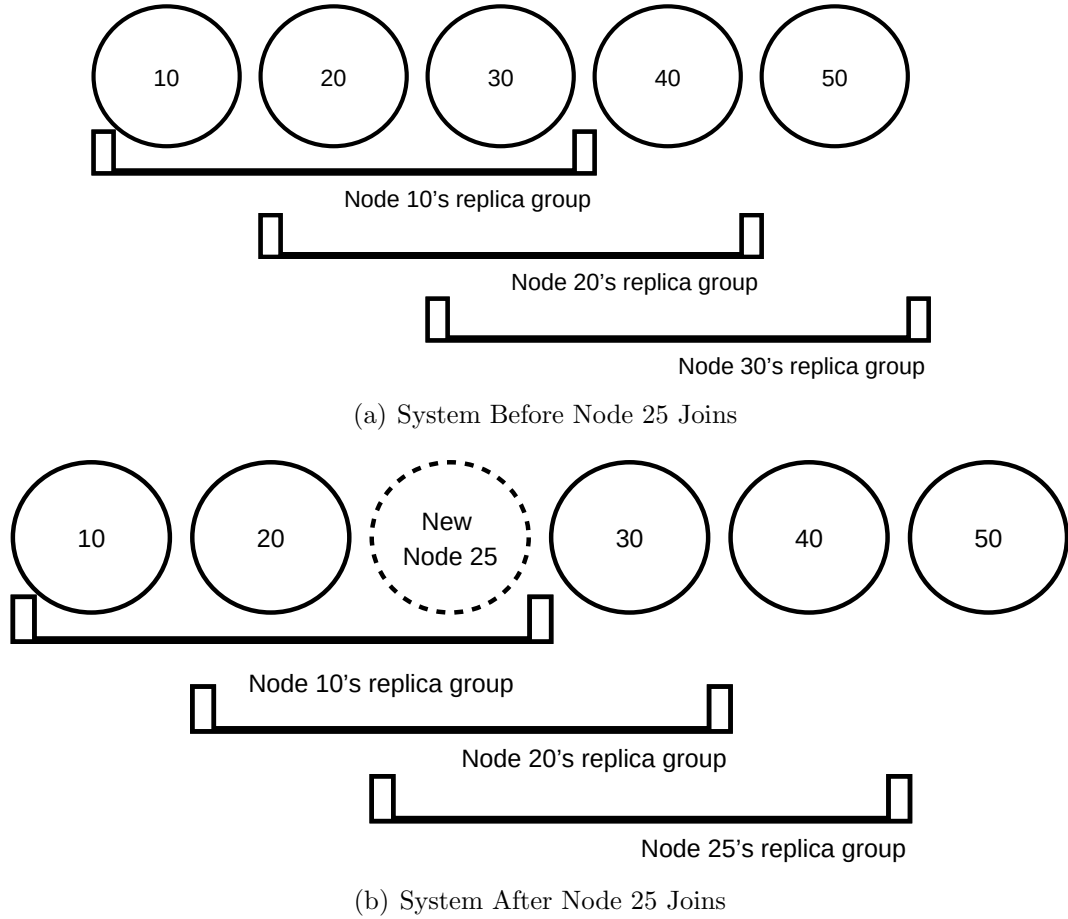


Fig. 5.6: Figure illustrating impact of new node joins on replica groups of size 3 under DHash

If maintenance is performed infrequently (less than 20 times per halflife), the maintenance overhead per call is relatively high – the gradient of the bandwidth/call curve is steep. This is because when maintenance is performed rarely, there are a large number of repairs to perform at each database synchronisation call. Once the number of calls per half life reaches 20, there are very few repairs to perform each call, and the bandwidth used by a maintenance call reaches a minimum – the slope of the graph becomes constant.

The simulation shows that finger placement has the same maintenance costs as symmetric placement, despite the fact that finger nodes are held in local routing tables. This is simply because although the finger nodes ID is known, their predecessors ID is not, and so it is not possible to determine which keys they are responsible for without doing a normal lookup. Placing replicas on finger nodes does not reduce overhead bandwidth.

The ranking of the placement functions suggests that the main influence on maintenance bandwidth is how many nodes we must communicate with to place replicas. Placement functions such as successor, predecessor, and block frequently experience allocation collisions. If an allocation collision occurs, the allocation function must try additional replica locations, and this leads to increased communication. We can see the relationship between the number of replica locations considered, and the replication factor in figure 5.8. The relative lack of allocation collisions explains why finger and symmetric replication have lower overheads than other functions. The discontinuities in block placement mean that it often has to consult an

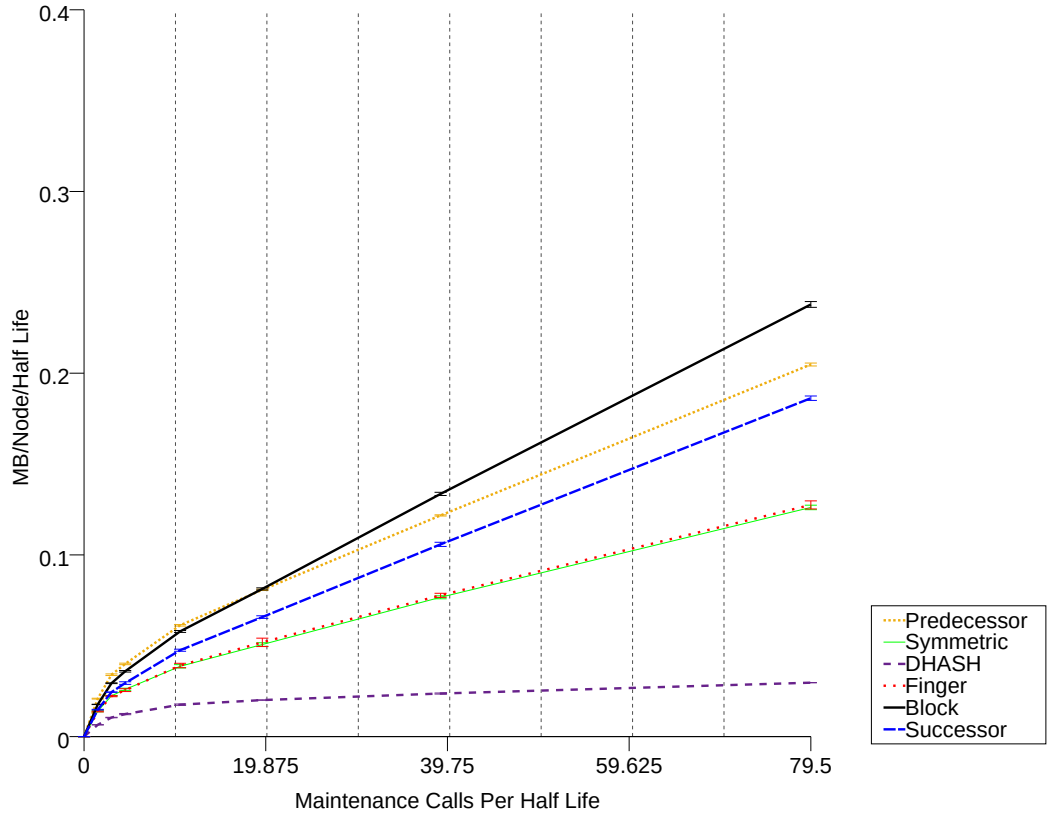


Fig. 5.7: Overhead bandwidth of Chord systems with various placement functions plotted against maintenance frequency. Error bars show 95% confidence intervals. These results are obtained with $N=300$, $R_{MIN}=6$ and $R_{MAX} = 14$

additional node, leading to the difference between it, successor and predecessor placement.

5.3.3 Load Balancing

The preemptive returns used by dynamic placement mean that the burden of answering fetch requests for a key is not equally balanced among the replica holders for that key. In our simulations, a node always returns data preemptively if it can. The resulting distribution of requests satisfied by each replica holder for each replication strategy is shown in figure 5.9.

The error bars on these measures are large, and this reflects the fact that the number of requests routed through a node is dependant on its keyspace size. A node with a large keyspace will receive a large number of incoming lookups, and so will respond to more queries preemptively. We assigned each node a random keyspace size in our simulations, leading to these variations.

Despite the large variations, it is clear that predecessor, successor and block placement do not balance the load equally among replica holders, whilst symmetric and DHash placement do. The finger placement strategy is slightly better balanced than the predecessor, successor and block, but still places more load on some replica holders than others.

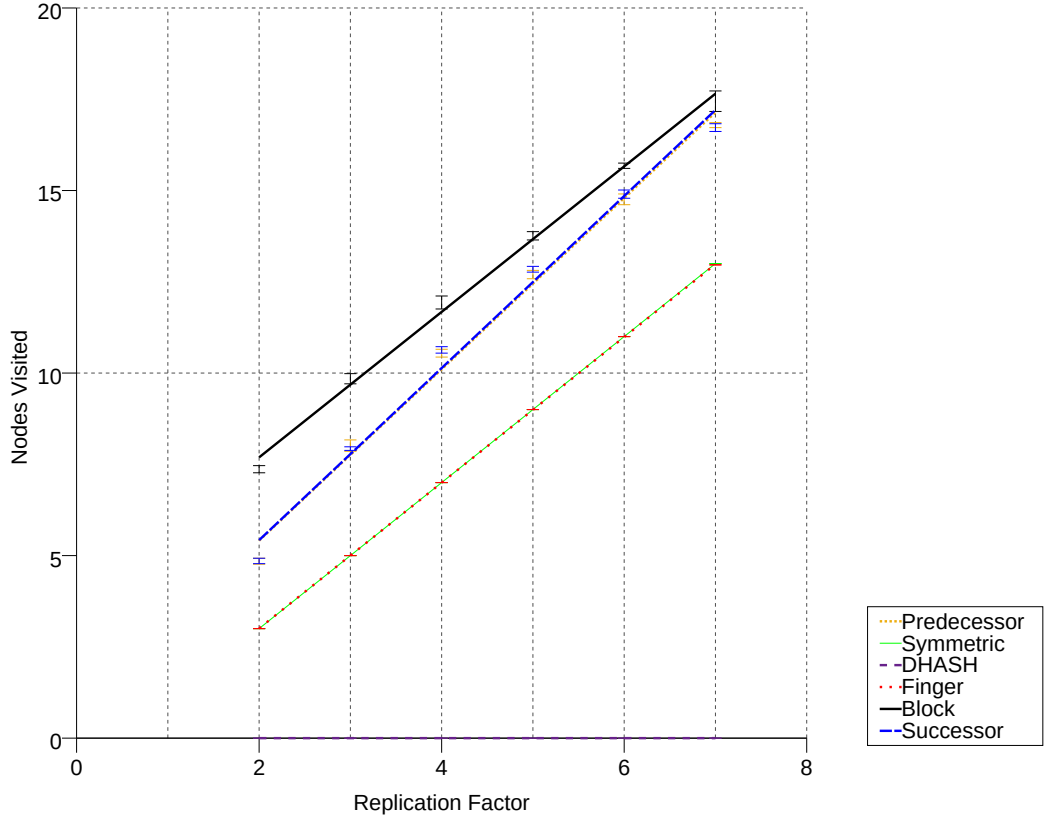


Fig. 5.8: Number of nodes communicated with by core maintenance against number of replicas. Error bars show 95% confidence intervals, lines indicate a linear fit to the data. The x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.95$ column in Table 4.4.1.

The most heavily loaded peer, under successor, block and predecessor placement, was always the peer with the lowest ID. This is a consequence of the way that lookups are routed clockwise around the Chord ring, and so are likely to encounter the replica holders with lower IDs before those with higher IDs.

Despite the fact that some nodes are under higher load than others when we consider a given object, the load should be more evenly balanced when we consider all objects. The objects are spread uniformly around the ring by the hash function, and so if all objects are equally popular, it is unlikely that any single node would have a significantly higher load than average.

If some keys were more popular than others, however, it is likely that the nodes responsible for these objects would become overloaded. Our recursive get algorithm allows for a node that is overloaded to choose not to return a replica, and instead to forward the request. If this was to become the norm, the fetch latency advantage that predecessor, block and successor placement gain due to preemptive returns would be lost, and their fetch latency would become equivalent to that of symmetric replication.

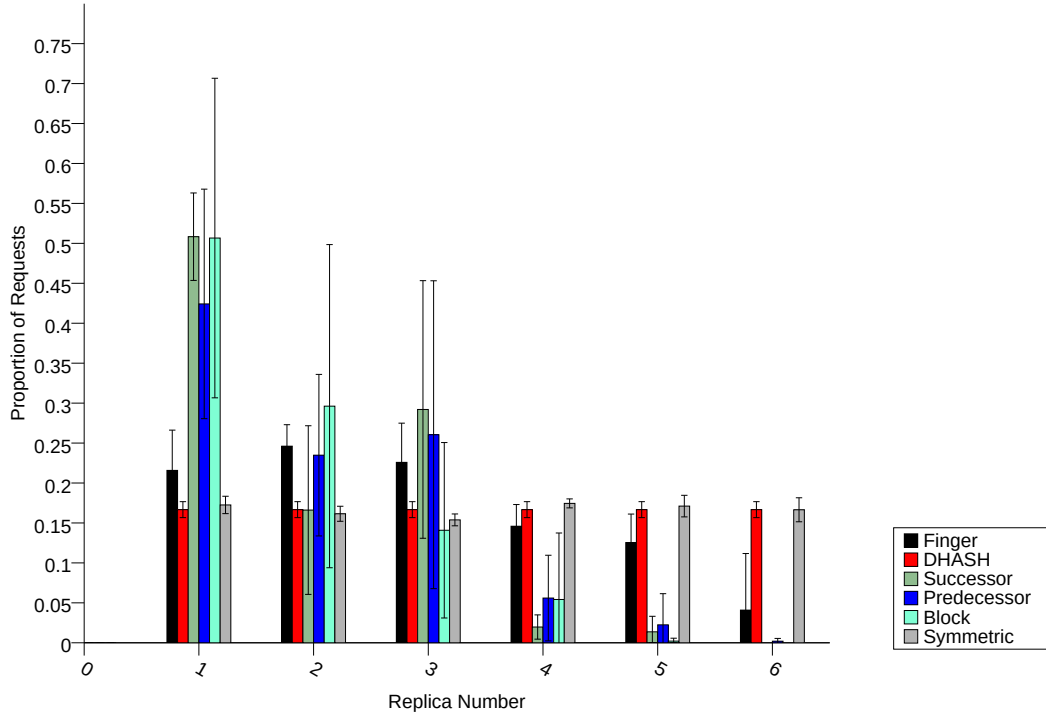


Fig. 5.9: Proportion of 5000 fetch requests for a single item answered by each replica holder for that item.

5.3.4 Write Performance

A write/update to a key begins with finding all replica locations for that key. Once this has been done, a variety of methods might be employed to consistently update that key, and the performance of these is no longer influenced by the replication algorithm. In our simulations, we measured the time taken to find all replica locations for various replication factors, and the results are shown in figure 5.10.

Because our dynamic write algorithm looks up all replica locations (including peripheral locations) in parallel, the time taken to find all replica holders is equivalent to the maximum of the lookup times for each replica location.

The DHash algorithm needs only one lookup – the successor list of the owner of an item includes all of the replica holders. Because of this, DHash write performance was not affected by the replication factor in our simulations.

Of the dynamic placement algorithms, symmetric placement gave the worst performance. This is because it distributes replicas equally around the ring, so that regardless of the location of the writing node, it is very likely that at least one replica holder will be in a position that requires a large number of routing hops to reach. The likelihood that at least one replica is in a position that is slow to access increases with the number of replicas. This means write performance is reduced as the number of replicas increases – the slowest replica is the limit on write performance.

The finger replication pattern places replicas at well spread locations in one half of the

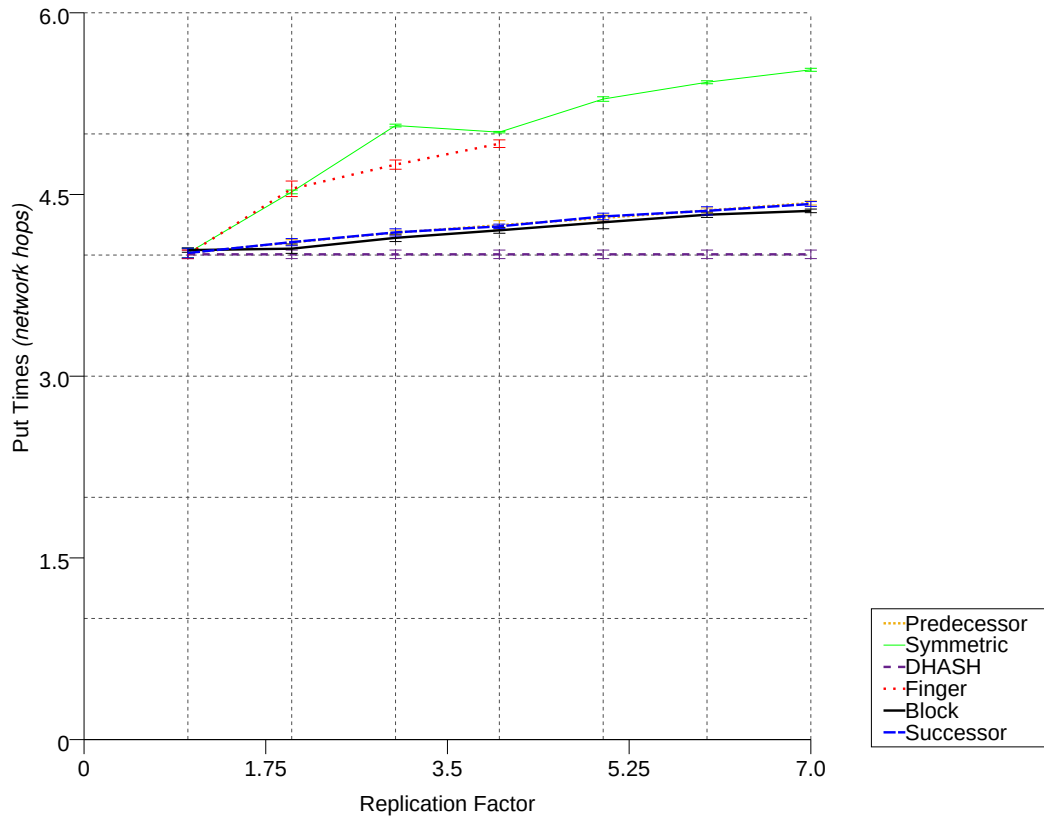


Fig. 5.10: Time taken to locate all replica holders of an item, for various replication factors. These values are for a system size of 200 Nodes. Error bars indicate 95% confidence interval. The x-axis shows R_{MIN} . For R_{MAX} values, see the $\pi = 0.95$ column in Table 4.4.1.

ring. This means some nodes are closer to the replicas than others, and these nodes are able to locate the replica holders more quickly. This results in a lower average write latency than symmetric placement. It is not possible to use finger placement with a replication factor higher than four in systems of 200 nodes, so larger replication factors are not shown.

With successor, predecessor, and block placement, all replicas are placed in close proximity on the ring. Some nodes will be ‘nearby’ to these replicas, and so able to route to these locations in few hops. This means that on average, the write time is lower than with symmetric or finger placement.

An interesting artifact shown in this plot is that the performance of symmetric placement with a replication factor of four is actually better than with a replication factor of three. This surprising result is explained by the structure of the Chord routing tables. When a replication factor of four is used, a node’s nearest replica will lie in the quadrant of the ring between it and its penultimate finger pointer. Let us call the distance between our node and its nearest replica ‘ x ’. The distance from the nearest replica to our node’s penultimate finger pointer is also x , as is the distance from our node’s final finger pointer to the next replica, and the distance between the remaining replica and our node’s final finger’s penultimate finger — see figure 5.11.

Using the finger pointers, all these nodes may be reached in two hops or fewer, and so all that remains to be done is for the query to travel the distance x . Let us call the number of hops it takes for the query to travel this distance t . For some ‘good’ values of x , t will be small. If any replica happens to be in such a position, then all four replicas will be found within $t + 2$ hops. Thus, it is possible that all four replicas may be located very quickly. This is unusual – with other replication factors, it is very unlikely that **all** replicas will be in positions that can be located quickly.

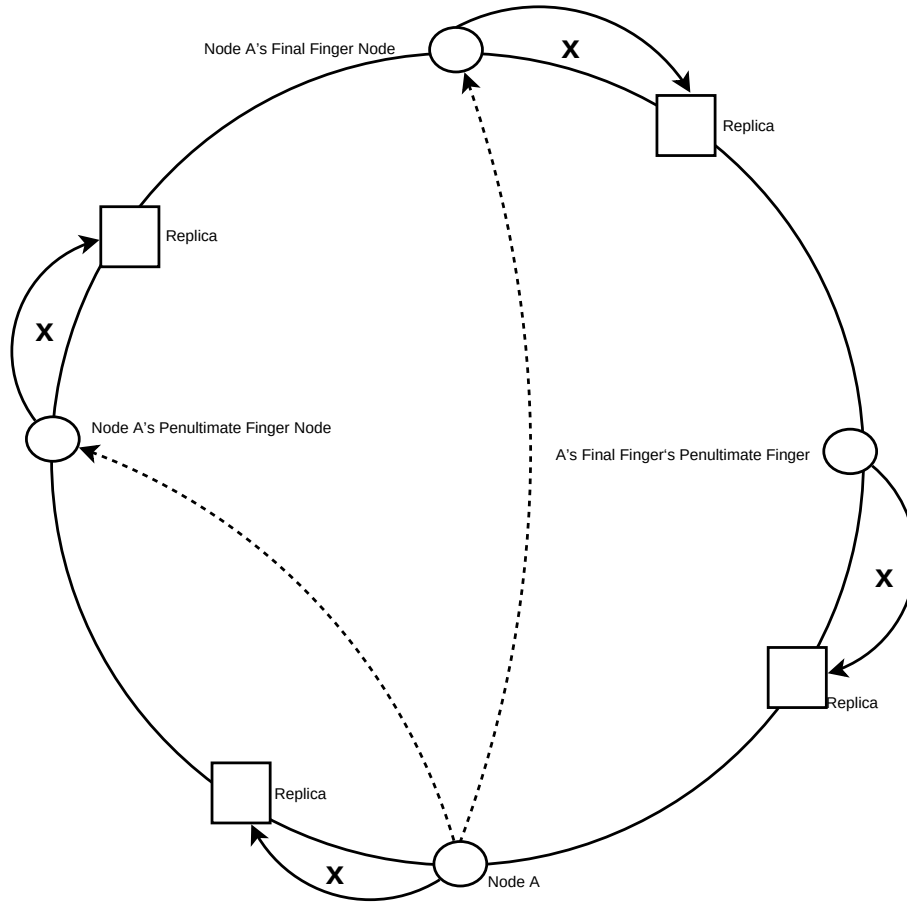


Fig. 5.11: When using symmetric placement with a replication factor of four, we find the distance between a node and its nearest replica (marked x) is the same as the distance between that node's final and penultimate finger pointer, and their nearest replica.

5.4 Summary

This chapter has revealed the complex trends relating maintenance frequency, placement pattern, replication factor and system size on fetch latency and communication costs.

The results have shown that DHash placement moves more data, and achieves slower fetches than any dynamic placement function. The differences between dynamic placement functions are also pronounced. Block placement, while providing the highest reliability, also appears to require the greatest maintenance costs. Predecessor placement results in the

fastest fetches, although its lead over successor and block placement is slight. Finger and symmetric placement incur the lowest communication costs. Finger placement appears to have little advantage over symmetric placement, other than slightly faster fetch times at high replication factors.

No single placement function excels in all areas, and so the choice of placement function must be made based on the priorities of the system designer. *Block allocation* is good in most areas, and offers the highest reliability. It would be a good general use choice, and the best choice for reliability critical applications. *Successor allocation* does not achieve the reliability of block allocation, but offers good fetch performance, and so might be chosen over block allocation in situations where this was considered more important than reliability. *Predecessor allocation* improves further on the fetch latency of Successor allocation, at the cost of increased bandwidth usage — it would be preferable to successor allocation only where bandwidth usage was not an important factor. In situations where bandwidth usage was critical, *Symmetric allocation* might be considered, as it offers the lowest communication costs of all algorithms tested. The read and write latency of symmetric allocation is comparatively high, unless parallel fetches are used — in which case it achieves very low fetch latency. *Finger allocation* offers superior write performance to symmetric placement, although this is at the cost of slightly increased communication costs, increased parallel fetch times, and significantly reduced reliability. Finger placement might be useful where both bandwidth usage and write performance are considered more important than other criteria. *DHash* has the lowest write latency of all algorithms tested, and although read latency is high, and it has by far the largest communication costs, it might be optimal for a system in which data update latency is the critical factor.

6. A PEER TO PEER GRID INFORMATION SERVICE

6.1 *Introduction*

The possibilities for reliable distributed data storage presented in the previous chapters may provide an alternative to the centralised information services currently used in most Grid systems.

In this chapter, we show how SAM-Grid may be adapted to use a peer to peer information system. We demonstrate the practical feasibility of such a system through the implementation, deployment and performance testing of a prototype.

Section 6.2 gives an assessment of the extent to which a peer to peer information service can replace the current centralised solution. Section 6.3 describes an architecture for a distributed information service for SAM-Grid. Section 6.4 describes a prototype implementation, and uses this to give a quantitative assessment of the performance characteristics of the system, in terms of the lookup latencies and bandwidth usage.

6.2 *Suitability Of A Peer to Peer Architecture*

In Chapter 2, we presented scalability, performance, and reliability problems caused by the current implementation of SAM-Grid. In this section, we assess how far a DHT based architecture, with the inherent load balancing and failure tolerance it can provide, can help to solve these problems.

One of the main limitations of a DHT is that it offers only a key based lookup service. Currently, the SAM-Grid information system provides both key based lookup of information and complex searches of metadata. A DHT based architecture may remove the reliance on the database for key based lookup of information, but would continue to rely on the database for searches.

In order to assess whether providing reliable key based lookups alone would help address the issues facing SAM-Grid, it is necessary to examine the proportion of requests which can be satisfied without recourse to the database.

To assess this, the types of requests made by SAM users at the CDF experiment were examined over a four day period. The data were collected from the four main production DBServers. These DBServers are responsible for serving the analysis farms, user requests,

stations and file concatenation requests. These requests are delivered to the DBServer as CORBA method calls, and provide an insight into the requirements SAM-Grid places on its Information Service.

Over the 4 day period, 83,132 requests for the retrieval, storage, or modification of information were recorded. The majority of these (76%) originated from stations, with a further 21% coming from compute farms and non-interactive concatenation jobs. 3% of requests originated from the user DBServer, intended for interactive jobs. This bias towards non-interactive usage reflects the fact that most SAM-Grid requests are generated by batch jobs running on CPU farms.

The requests may be categorised according to the type of database activity required to satisfy them. Those requests which specified as part of the request the unique keys of any rows in the database they read or modified are classified as a read or a write, respectively. Those queries which do not specify the unique key of the rows in the database they read or modify, or which request a metadata search, are classified as searches. Figure 6.1 shows the breakdown of the type of operation.

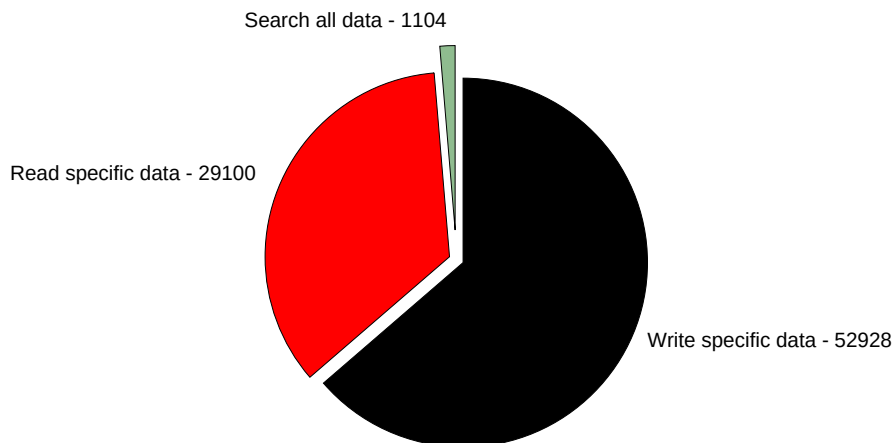


Fig. 6.1: Proportion of DBServer requests requiring database Read, Write, or Search.

The majority of the requests (68%) involve modifying a named project, updating the list of files consumed by a project, or retrieving replica locations for a named file. Over the sample period, just 1104 complex search operations were recorded, accounting for just over 1% of activity.

Analysis of the source of these complex searches shows that the majority are requested by monitoring tools. These tools regularly run queries to fetch information such as the list of all currently running projects. They use the ‘query’ function of the DBServer to pass SQLS

queries to the database. These monitoring functions could not currently function without complex search. However, the temporary loss of monitoring tools would be acceptable during database downtime. Additionally, it would be feasible to redesign the monitoring system to allow it to operate despite the lack of a central database, either by storing lists of running projects as named objects, or by moving toward a more distributed, event based, monitoring architecture.

A minority of the complex searches are caused by users creating or referencing *dataset definitions*. Dataset definitions are a set of criteria for file metadata that define which files a physicist has selected as suitable for a particular analysis project. Once such a list is created, the result may be stored as a *snapshot* under a given name. When the snapshot is referred to in future, the list of files can be retrieved without repeating the search. This reduces the frequency of complex searches across all files, and thus reduces database load. Furthermore, snapshots can be stored in a DHT, allowing requests for the snapshot to succeed without the central database.

During database downtimes, it is not possible to define new snapshots. Being unable to define new snapshots for long periods would constitute a serious loss in functionality for SAM-Grid. Defining new snapshots, however, is not critical to currently running projects, and could be accepted during short periods of database downtime.

The majority of SAM-Grid information requests do not require complex search, and so a name based lookup service would also remove the burden of a significant proportion of queries from the central database. Many read operations will be satisfied by the DHT without needing to consult the database. The analysis given above indicates that this would reduce the number of database queries by 35%. However, since writing information and answering complex search queries requires more computation than simple reads, it is likely that other operations will continue to place considerable load on the database.

In conclusion, a system which provided name based information lookup during database downtime would significantly improve the failure tolerance of the SAM-Grid system. No activities critical to continued operation of running projects require complex search, and so the impact of database downtime on CPU farms would be mitigated. Read scalability could also be improved. This suggests that it is possible to improve the reliability and performance of SAM-Grid through the use of a distributed hash table, though it is neither necessary nor feasible to replace the central database completely.

6.3 An Architecture for a Peer to Peer Information System

6.3.1 System Capabilities

In designing the system architecture, our aim is not to allow all SAM-Grid operations to continue during database downtime, but instead to focus on providing a simple, reliable system that supports the majority of SAM-Grid operations. The system should use the strengths of a distributed hash table to increase reliability, while falling back to the database to provide complex search and arbitrate on the consistency of updates. The architecture described in this section removes reliance on the central database for the most common station operations, and allows limited station autonomy, so that a station may continue to process local files without network access. The database remains an important component of the system, but is no longer a bottleneck for reading data. The ability to read information and modify some information when the database is not available means that brief periods of database downtime are acceptable, and mitigate the catastrophic impact of database downtime on the analysis farms.

In order to integrate a Chord Ring with SAM-Grid, an intermediary Information Service (IS) is introduced. Each station runs its own information service – its *gateway node*, and these form the Chord ring. The IS implements the same interface as a DBServer, and so we simply configure each station to contact the gateway node instead of the central DBServer. The IS takes requests from the station and either passes them to the central DBServer, or uses the Chord ring to look up the information. A broad architectural diagram is shown in figure 6.2.

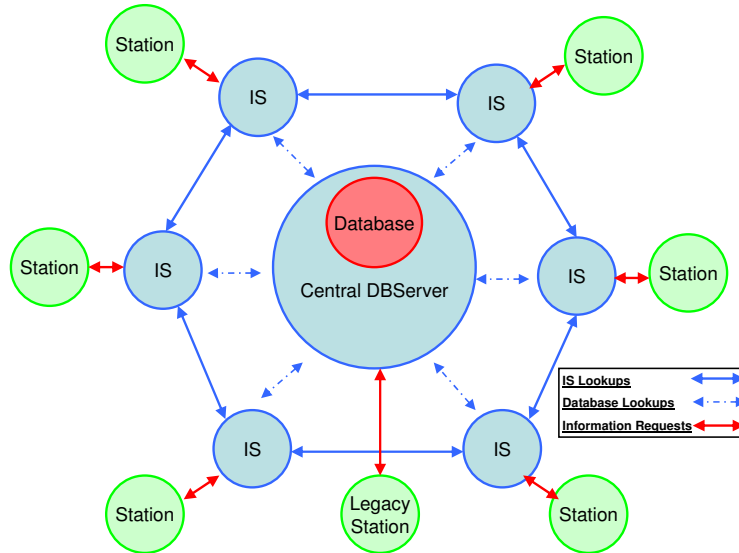


Fig. 6.2: Information Service system architecture. Local IS services run at each station, queries they cannot satisfy are passed to the central DBServer then stored in the ring.

The central database is still used for all complex searches. When a station requests specific information, however, the information service will consult the Chord ring. If the information is not found, the lookup will be passed to the central DBServer, and the results

will be both inserted into the Chord ring and returned to the station. The central DBServer also sends updates to the Chord Ring, and so updates from legacy stations contacting the central DB-Server directly, are also sent to the Chord ring. This ensures the central database and the Chord Ring remain consistent during the deployment of the proposed system – during deployment there will inevitably be periods during which only a fraction of sites are using their own Information Service.

Each node has a local store of information, implemented as a persistent database, which stores Chord (key,value) pairs. When information is stored in the Chord ring, it is given a name based on the type of object it represents, and that object's unique key. These names are then converted by a hash function into a 160 bit Chord ID. The associated data is then serialised and stored on the nodes responsible for that ID. When a request is made for a specific piece of information, the unique key is extracted and used to calculate the relevant Chord ID, which is then fetched from the Chord ring.

Each information service node joins the Chord ring using a bootstrap node, which is selected at random from a list in the configuration file. If no bootstrap node is specified, a new Chord ring is created, initially consisting of just one node. Other nodes may then use that node as a bootstrap.

Security is provided using a host-based mechanism, as used by the rest of the SAM system. The database contains a list of hosts which are authorised to access SAM-Grid, and this list is checked whenever a new node accesses the system. The list of authorised hosts may be cached in the Chord ring, so that it may be accessed when the database is unavailable. Changes may only be made from authorised hosts. This security system reduces the possibility for malicious activity such as Sybil attacks.

As well as implementing the DBServer interface, each Chord node implements a peer to peer storage interface. This is shown in the UML diagram in figure 6.3. Each of these methods will be described in more detail in the following sections. In addition to these data storage and retrieval methods, each information service must implement both the standard Chord protocol, and data maintenance protocols, as described in the previous chapters. The data maintenance protocol must be slightly modified to recover from network partitions. During both local and global maintenance, object modification times are compared. If two nodes store the same object with different modification times, the most recent version is retained.

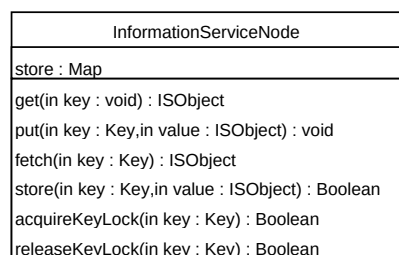


Fig. 6.3: UML Class Diagram showing data storage methods implemented by an Information Service Node.

6.3.2 Reading Data

An information system node implements the *get(key)* method to allow other nodes to request a key from its local store. A pseudocode description of the get algorithm is given as Algorithm 11. Attempts to *get* a key which is locked will, after a short timeout, return an empty value. Data which is locked — perhaps because it is in the process of being changed — can instead be fetched from the database. When the database is available, it provides the definitive current version of any given data item.

Algorithm 11 Pseudocode for Information Service *get(key)* procedure

```

keyLock = self.store.getLock(key)
// Attempt to acquire lock for key. If key is currently locked, timeout and
return None
haveLock = keyLock.acquire(0.1)
if haveLock then
    item = self.store.get(key)
    keyLock.release()
    return item
else
    return None
end if
  
```

As well as the *get* method, which acts directly on a particular node's local store, an information service node implements methods that allow a user to request information from the Chord Ring as a whole. This is the *fetch* method, which we will now examine in more detail. A pseudocode description of the data fetch algorithm used by the IS is given in Algorithm 12.

If the fetch algorithm fails to locate the item in the Chord Ring or in the database, it will return a null value, *None*. In this circumstance, the client must back off and retry after allowing data maintenance to take place.

6.3.3 Writing Data

The self-healing nature of a Chord ring creates new problems for an information system that seeks to maintain consistent data. A network split (for instance a networking failure between the US and Europe) can cause the system to divide into two or more fragments, each of which contains a copy of all the data. If we allow this data to be modified, the update will appear to succeed, but will only reach only one fragment. This will cause serious problems when the fragments are reconnected, since only one version of an object can be kept, and users may find that operations they thought had succeeded now appear to have failed. We solve this problem by only allowing certain fragments to modify certain information, ensuring that a given object can only be modified by a single fragment. When that fragment rejoins the rest of the system, data maintenance updates the stale information stored in other fragments.

Decisions on which data can be changed by a particular fragment are made depending

Algorithm 12 Pseudocode for Information Service *fetch(key)* procedure

```

// First try local store
if self.store.contains(key) then
    item  $\leftarrow$  self.store.get(key)
else
    // Not in local store, try Chord Ring
    successors  $\leftarrow$  chordRing.lookup(key)
    replicaHolders  $\leftarrow$  take(REPLICATION_FACTOR, successors)
    // Try each replica holder in turn, in a random order
    while replicaHolders  $\neq \emptyset$  and item = None do
        replicaHolder  $\leftarrow$  replicaHolders.removeRandom()
        item  $\leftarrow$  replicaHolder.store.get(key)
    end while
end if
if item = None then
    // Not in local store or Chord Ring, Fetch from DBServer
    item  $\leftarrow$  dbServer.get(key)
    // Attempt to store in Chord ring. 'store' will not update/replace existing items.
    chordRing.store(item.key, item)
end if
return item

```

on the type of object being modified. We divide data objects into two categories — *local data*, and *global data*. Local data are objects that a particular station controls exclusively. Examples of a station's local data objects are the station's disks, the lists of files stored on those disks, and the lists of which projects are running on that station. Only the fragment containing the station which owns a local data item may modify that item. This means local data may be modified during database downtime, if the station which owns that data can be contacted.

Objects that cannot be associated exclusively with a particular station are categorised as global data. An example of such an object would be the metadata associated with a file. We allow global data to be modified only if the database can be contacted. During network splits, this means that only the fragment containing the database may continue to update global information.

In order to allow consistent updates to all replicas of an object, keys in the local store may be locked by a remote node using *acquire(key, timeout)* and *release(key)* methods. The *acquire* method will either lock the appropriate key, or, if that key is locked, wait for up-to *timeout* seconds for the key to be unlocked. The *acquire* method returns a value indicating whether the key was successfully locked. The key will then remain locked until it is either unlocked, or the lock times out. A lock timeout is necessary so a node which fails without unlocking the keys it has acquired does not permanently prevent those keys from being changed. If a locked key times out, the associated value must be reverted to its value before the key was locked, in order to avoid possible inconsistency.

Once locks are acquired, information is updated using the low level *put* method. The *put* algorithm simply updates a node's local store with the new value of the supplied key.

When changes are made without the database being available, updates are made in the Chord ring only. These must eventually be committed to the database, and so a node that has become disconnected from the database must monitor database availability, and any changes to items it owns must be committed to the database as soon as it becomes available.

The complexity of writing data into the information system is managed by the *update* algorithm. A pseudocode description of the update algorithm is given in Algorithm 13.

The update algorithm may fail if all existing replicas cannot be locked. This may occur because another user is currently attempting to lock the same key. The client should retry the operation after a random delay, so that each client has a chance to acquire the necessary locks. The update algorithm may also fail if the database is unavailable and the data being modified is global. In these circumstances, the client must wait for the database to become available, or return a failure to the user. A final failure mode occurs if the data is local to a station, and the station associated with the data cannot be contacted. This was not a failure mode in the centralised system, and so technically this adds a new point of failure to the system. It does not decrease system reliability in practice, as although local data is read by other stations, it is not modified except by its owner station. The reliability benefit of allowing local data to be updated without the database therefore outweighs this additional reliability bottleneck.

6.4 *Prototype Implementation and Performance*

In order to test the effectiveness of our architecture, it was decided that a prototype implementation should be built. The aim of the test system is both to illustrate the feasibility of our architecture, to show trends in latency and bandwidth consumption, and to indicate likely performance bottlenecks in a full-scale system.

6.4.1 *Prototype Implementation*

The prototype implements a subset of the full functionality of the proposed architecture. This subset was chosen to minimise development time, whilst allowing performance testing of key parameters such as the fetch latency and bandwidth use of the system.

The prototype implements a Chord Ring running the DHash maintenance algorithm. Full replication and failure recovery features are implemented. The Information Service layer implements a read-only version of the architecture described in the previous section, and supports only one type of object — SAM file metadata. Requests for all other data types are passed through the information service to the central DBServer.

Our prototype is implemented in Python, and uses CORBA for network communications. The implementation is multithreaded, with two threads running Chord and DHash maintenance algorithms, as well as a CORBA Thread Pool handling remote method calls.

Algorithm 13 Pseudocode for Information Service *update(key, newvalue)* procedure

```

// Lookup any replicas of this item in Chord ring
successors ← chordRing.lookup(key)
replicaHolders ← take(REPLICATION_FACTOR, successors)
// Lock all replicas of this item against reading and writing by other nodes
lockSuccess = acquireKeyLockOnNodes(key, replicaHolders)
updateSuccess = False
// For consistency during network partitions, we may modify only global data,
or data for which the owner is present
updatePossible = item.ownerStation = Null or item.ownerStation.isAvailable()
if self.database.isAvailable() and lockSuccess and updatePossible then
    // Update in database
    self.database.put(key, newvalue)
    // Update in Chord Ring
    for all replicaHolder in replicaHolders do
        replicaHolder.put(key, newvalue)
    end for
    updateSuccess = True
else
    // Database unavailable
    // Check item belongs to a station and that station is available
    if item.ownerStation ≠ Null and item.ownerStation.isAvailable() then
        // Update in Chord Ring
        for all replicaHolder in replicaHolders do
            replicaHolder.put(key, newvalue)
        end for
        updateSuccess = True
    end if
end if
// Unlock all replicas
releaseKeyLockOnNodes(key, replicaHolders)
return updateSuccess

```

Node Name	Location	CPU Type	Clock Speed	Physical Memory
tbse01	Oxford, England	Intel Pentium II	266MHz	160MB
tbwn01	Oxford, England	Intel Pentium III	600MHz	128MB
tbwn02	Oxford, England	Dual Intel Pentium II	400MHz	256MB
tbce01	Oxford, England	Intel Pentium II	350MHz	128MB
tblcfg	Oxford, England	Intel Pentium II	350MHz	256MB
tbgen	Oxford, England	Intel Pentium II	200MHz	128MB
cdfsamth	Chicago, USA	Dual Intel Pentium III	800Mhz	1GB
d0sammon	Chicago, USA	Dual Intel Pentium III	1GHz	1GB
cdfsam02	Chicago, USA	Dual Intel Xeon	2.80GHz	4GB

Tab. 6.1: Locations and specifications of machines used in test deployment

6.4.2 Experimental Testbed

In order to test our system in a realistic setting, we used 9 nodes intended to simulate the likely real world deployment of the SAM-Information system. To mimic the international nature of the experimental collaboration, half of the connected Information System nodes were hosted at Fermilab and half of were hosted at Oxford University. At Fermilab, fast dual processor machines were used to host two information service nodes. Due to the limited resources available to us for testing, some nodes were poorly specified by current standards. However, CPU speed appears not to be a performance bottleneck; the CPU usage reported by the system tool *top* whilst running on a 300Mhz Intel Pentium II, ranged between 15 and 20% during the load tests.

Our information service was used in conjunction with the CDF production information system. The central DBServer, *cdfsam01* was used. This machine has four 2.4Ghz Xeon processors, and 4GB of RAM. The database was *fcdfora4* at Fermilab, which has eight 900Mhz SUN processors, and 30GB of RAM.

In section 6.4.9, we also test a lower latency physical network, consisting of 18 1Ghz Pentium 3 nodes on a local area network, in order to provide an indication of how network latency can affect system performance.

In our tests, the Chord and DHash maintenance algorithms run at intervals of every 30 and 45 seconds respectively. Successor lists are configured to have a length of three, up to 5 distinct fingers nodes are maintained, and the replication factor is set to two.

6.4.3 Performance

The performance of the system was assessed using two separate metrics, *fetch latency* and *bandwidth usage*. Fetch latency is the performance of the system as perceived by an interactive user, and measures the latency from the time when a request being sent until the time the relevant information is returned to the client.

The system's bandwidth usage is also an important indication of the systems performance. Nodes in the system are in constant communication with one another. As interna-

tional network links are expensive, the system must use the network efficiently.

6.4.4 Fetch Latency

We first examine factors affecting fetch latency, the latency between requests being issued and information being returned. System performance as measured by data fetch latency depends on several main factors:

System Load Although the simple prototype information system places minimal requirements on a systems CPU, many concurrent clients may place an increased strain on the machines involved, and may affect performance.

Data Access Patterns The proportion of the data which is cached in the information service affects data fetch latency. Data which is not cached must first be searched for in the information system. This data being absent, the system must then fetch the data from the central database and cache it in the ring before returning it to the client.

Data which is frequently requested will benefit from lower fetch latency than items which have not been cached.

System Size Chord lookups require $O(\log(n))$ network hops in an n node system. In practice, however, variations in CPU time and network latency can make the relationship more complex.

System Stability A node failure may cause an increase in fetch latency until the maintenance algorithm redistributes all data to all nodes responsible for it. Once the system has recovered, a decrease in fetch times may be observed due to the reduced number of participants in the system.

Physical Network Latency The way in which the ring is deployed affects fetch latency. Ideally, all machines and the client will be on the same local network. A more distributed information service with longer network latencies between hosts will lead to longer lookup times.

We will now examine each of these factors in turn, using our testbed to give quantitative assessments of how these factors affect system performance.

6.4.5 Impact of System Load on Fetch Latency

We tested the impact of system load on fetch times by running varying numbers of clients against our 12 node test system. Each client ran on a separate CPU, and contacted the Information Service through a different IS node. Each client requested 2000 cached items, and the average fetch time was computed.

For comparison, tests of the fetch latency offered by the current centralised information service were also conducted. The same performance tests were run against the existing CDF

SAM-Grid system during a period where there was little other activity. Again, average fetch latencies were computed from 2000 requests. Both sets of results are shown in figure 6.4.

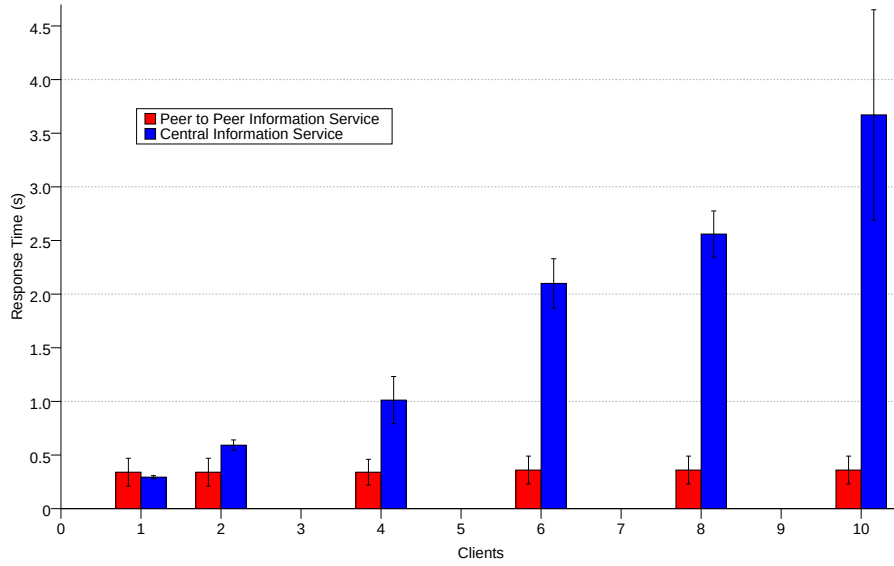


Fig. 6.4: Fetch Performance Against System Load for Centralised and Peer to Peer Systems. Values are mean average for 2000 requests, error bars show one standard deviation.

The centralised implementation exhibits poor scalability in this test. It became significantly slower as the number of clients increased. The bottleneck in these tests was the Oracle database, which must fetch the data to answer each query sequentially from a single disk array.

The peer to peer version showed only very slight variation in fetch times as load increased. Although the peer to peer version was slower than the centralised implementation when serving one client, it was an order of magnitude faster when serving ten clients. This test clearly demonstrates the superior read scalability of the peer to peer information system.

6.4.6 Impact of Data Access Pattern on Fetch Latency

The graph shown in figure 6.9 shows times for fetching data already stored in the distributed hash table element of the information system. When data is not stored in the distributed hash table, the prototype information system will first check the hash table entry is empty, then fetch the information from the database, then insert the data into the distributed hash table, and finally return the data to the user.

This means that the time taken to retrieve data for the first time is significantly longer than subsequent reads. Figure 6.5 shows the difference between fetching uncached and cached data on our testbed. As previously, fetch times shown are the mean average over 2000 requests.

Actual fetch performance would tend to the cached value as the system evolved, bringing a higher proportion of the databases content into the Chord ring. There would be an initial

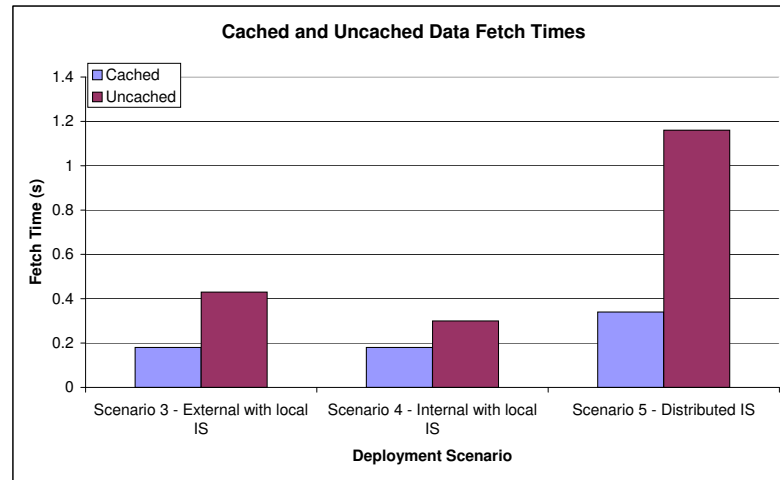


Fig. 6.5: Fetch Performance By Cache Status

period during which few of the requests would be satisfied by cached data, and during this period, average fetch time will be higher.

The distributed information system can take over a second to retrieve uncached data, suggesting a need for improvement. Three optimisations to the architecture could potentially reduce fetch times for uncached data:

Asynchronous Hash Table Insertion. When fetching previously uncached data for the first time, the current architecture first inserts data into the Chord ring, then returns the data to the client.

By returning inserting the data into the hash table in a dedicated control thread running either in parallel to or after returning the data to the client, a significant performance gain should be achievable.

Lazy Replica Checking If the primary replica holder for a given object does not hold a replica of that object, then it is most often the case that this object has not yet been inserted into the information system.

A performance improvement could be achieved by searching only a single replica holder before deciding the data is not in cache and going to the database. Additional replica holders would only be searched if the database was not responsive. In the common case where the data is not cached, and the database is available, this will significantly reduce initial fetch latency.

Pre-caching A simple solution that might aid performance would be to have a dedicated process loading the Chord ring with data preemptively, before any user or station requests it. This would mean that it is already in the cache when a node requests it.

An implementation of the first two techniques would reduce fetch-time for uncached data. Pre-caching could eliminate uncached data entirely.

6.4.7 *Impact of System Instability on Fetch Latency*

To assess the effect of system instability on fetch latency, we simulated the failure of one node from our testbed, and monitored the fetch latency of two clients requesting data at that time. In some cases, the failure had no impact on either client, and no deviation from normal lookup times was observed after the node failure. In others, however, a CORBA communication error occurred, causing a back-off period of 20s, followed by a retry which was always successful. Those fetches not being routed through the failed node at the time of failure, as well as those fetches that were launched after the routing overlay was repaired, proceeded as normal.

Communication errors are made more common because of the recursive nature of the routing algorithm. If any node on a key's lookup path fails while that key is being requested, a communication error occurs. This error is propagated through all other nodes routing the request, and reaches the requestor, triggering the back-off period. The back-off period is set to be 300% of the Chord maintenance interval, to allow all time for repairs to complete.

An alternative to waiting for repairs would be to try an alternative routing path. Chord provides a number of possible routing paths to a given key, and nodes usually choose the one which requires the least network hops, forwarding a request for a key to the closest node in its routing table to that key. If that node has failed, our current implementation propagates the failure to the client. An alternative would be to forward the request to the next closest node in the routing table instead. This would decrease the impact of node failures on fetch latency, at the expense of an increase in bandwidth usage and the complexity of the routing algorithm.

6.4.8 *Impact of System Size on Fetch Latency*

The Chord routing algorithm should show lookup latency logarithmic in the number of participant nodes [38]. Chord's finger table based routing means that the distance remaining to the key is halved at each routing step, so that if the keyspace size is 2^m , then after $2\log(N)$ steps, the keyspace between the current routing node and the key k will be at most $2^m/N^2$. With randomly placed nodes, the probability of another node being in this space (so requiring another routing hop) is $1/N$. For large systems, this probability is insignificant, and so lookups are very likely to terminate after $2\log(N)$ steps [38].

The small numbers of nodes involved in our test systems means the $1/N$ probability of an extra hop being required is not insignificant. The smaller systems could be expected to have a slightly longer fetch times than a purely logarithmic scaling would suggest.

We examined the scaling of fetch latency with system size using our testbed, varying the number of participant nodes while keeping the number hosted on each site equal. The standard 2000 item metadata fetch performance test was used for each system size.

These results of these tests, however, showed no relation between system size and fetch latency. No uniform trend was observed, and there was no significant increase in fetch latency between 4 and 12 node systems. This is explained by the heterogenous nature of the network links between nodes. Depending on the underlying physical network, each routing hop involved in routing a request is either a high latency hop, across the Atlantic, or a low latency hop, between two machines on the same site. As we change the number of machines, the successor and finger pointers of some nodes also change. Nodes communicate only with nodes in their local routing database: if the nodes in this routing table happen to be on the same site, a routing hop will not increase the fetch latency as much as a high-latency link would have. Transatlantic links typically involve latency a factor of 100 greater than same-site communication. Thus, the lookup latency in our testbed depends not only on the total number of routing hops, but on the number of routing hops which cross the Atlantic. In our testbed, these changes in query routing paths outweigh any due to system size.

In order to isolate changes in routing table structure from changes in system size, and so see the relationship between system size and lookup time more clearly, we deployed an information service on a set of machines with homogeneous network links. A cluster of 18 1GHz Pentium III machines connected by low latency ethernet were used. Because network latency between nodes is low, the CPU time involved in marshaling and unmarshaling data became a significant factor in fetch latency when fetching the large SAM-Grid metadata items. To obtain the timing resolution necessary to distinguish changes in lookup time, we used a different test client. Our test client does not use the marshalling code in the DBServer adaptor class, and instead queries the Chord Ring directly. It stores an 8 character string rather than a 2KB SAM metadata record. The client generates 3000 such values at random, stores them, and then subsequently reads them back. The latency of 3000 reads are measured and we use these to assess fetch latency for this test.

This test client provided the necessary timing resolution to observe the effect of system size on fetch latency, as illustrated in figure 6.6. During this test, we occasionally saw fetch times of greater than 100ms. These may have been caused by other tasks sharing the network or the worker nodes. The number of outliers seen shows no correlation with system size, and as they do not reflect the normal performance of the system, the outliers are excluded from the analysis. These outliers account for less than 0.15% of the 21,000 measurements taken.

The results of this test were consistent with a logarithmic relationship between fetch latency and system size. Each point is an average of 3000 fetch times measurements made on a system with a given size. The error on this measurement is very small. Due to time constraints, we test only a single system, with one set of randomly chosen node identifiers, for each system size. This means we record the average fetch time for a particular configuration of node identifiers. It is likely that the average fetch time for several different configurations of node identifiers would have a larger variation, and would agree more closely with the logarithmic fit – as in the simulation results given in figure 5.1.

Figure 6.7 shows the histograms of fetch latencies for four different system sizes. In each, we can see distinct peaks, at separations of approximately 3ms. These peaks correspond to the queries answered after a given number of network hops. We can see that in the 6 node system, there are only three peaks, indicating that all requests are fulfilled with between

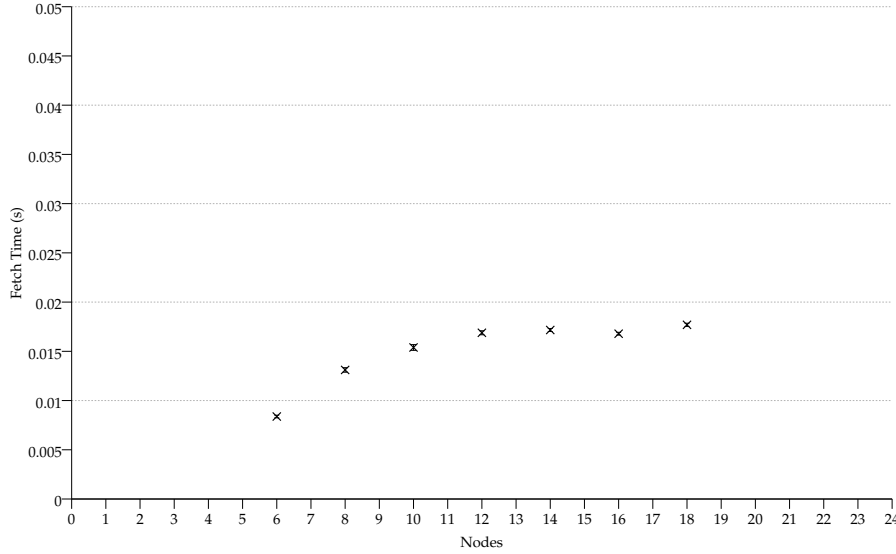


Fig. 6.6: Relationship between system size and *Chord-only* fetch times on a low latency network.

zero and two network hops. Finding data in the 18 node system, however, requires up to 5 network hops. The maximum number of network hops required to find data increases with network size. The small system sizes we used mean that we would not expect to see a perfect logarithmic scaling, and the nature of the trend is unclear from the small range of system sizes we tested.

We can conclude that although the performance of the underlying Chord algorithm is consistent with a logarithmic scaling when deployed in a homogeneous environment, the trend is obscured in small real-world deployments in heterogeneous networks. In low latency settings, lookup times are only a small proportion of metadata fetch time, with the majority of fetch time being the CPU time required to marshal and unmarshal SAM metadata.

In a high latency setting, lookup time is the most significant component of fetch time. In this case, however, the relationship between system size and lookup time is more complex, as the underlying physical network topology determines the number of high latency links involved in a given lookup path. This suggests that minimising the number of high latency hops, using a method such as Proximity Neighbour Selection [98] would be an effective way of reducing fetch times in a large scale deployment.

6.4.9 Impact of Physical Network Latency on Fetch Latency

To illustrate the effect of various likely network settings, we deployed our information service in a variety of different network settings, as illustrated in figure 6.8.

Deployments A and B are given for comparison purposes, and reflect the performance of the centralised system. All information requests are routed to the central DBServers. In Scenario A, the performance of the centralised implementation was tested from the Fermilab site (low latency link). In scenario B, we used a machine in Oxford, connected to the Fermilab site with a high latency network link.

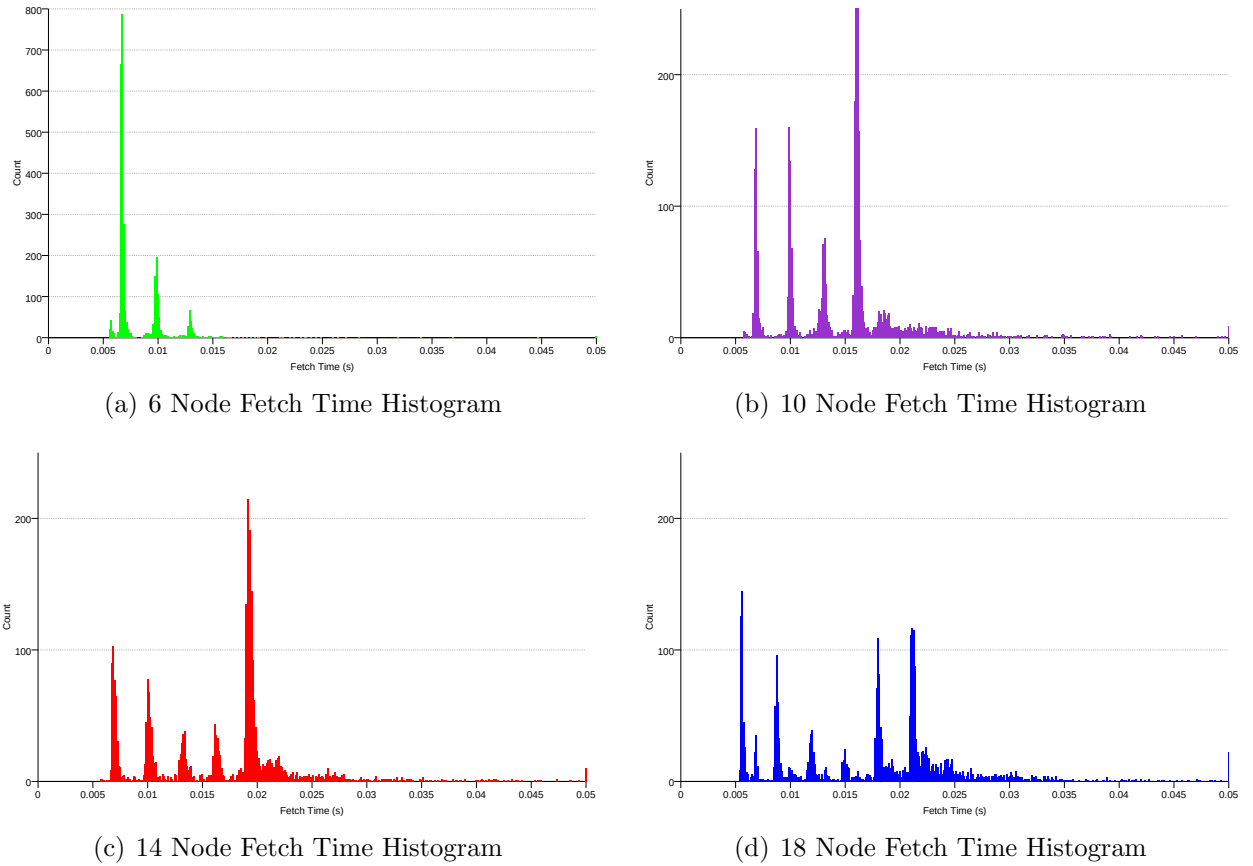


Fig. 6.7: Fetch Time Histograms for various system sizes

Deployment one, two and three use the peer to peer information system architecture. Deployment one represents a system in which data is cached in a single information service node. This pattern represents the peak performance our architecture can achieve, no network communication is required to fetch a data item.

Deployment patterns two and three represent systems in which data is looked up in a Chord Ring. Pattern two represents a single site Chord Ring, with low latency networking. We used the cluster of Pentium III nodes at Oxford to test this scenario. Pattern three is a distributed Chord Ring, involving high latency links. We used our 12 node testbed to simulate this scenario.

These deployment scenarios were tested by using the SAM client to send requests for the metadata for two thousand files, recording the times to retrieve each one, and calculating the mean average fetch time. Each item is requested twice. On the first request, the data is loaded into the initially empty information system. On the second pass, all requests are satisfied by the Information System cache. Results for the second pass are shown in figure 6.9.

We can see that for most users of the current information system, even a peer to peer information service using high latency links would improve fetch latency. The exception to this is that users at Fermilab (deployment A), would see a slight increase in fetch latency if they switched to our 12 node distributed information service.

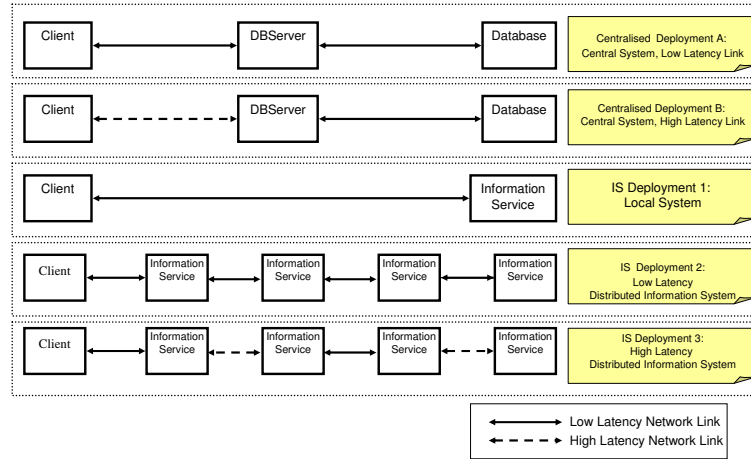


Fig. 6.8: Deployment Scenarios

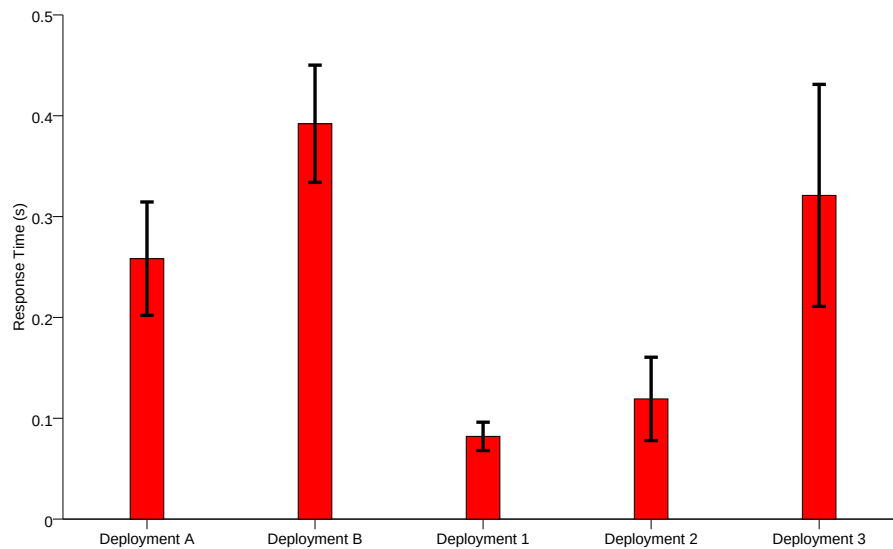


Fig. 6.9: Fetch Performance By Deployment Scenario

The reason that information may be fetched from the distributed information service faster than under the current system lies in the CPU time the centralised system requires to translate a request into SQL, execute that SQL query on the database, translate the result into a Python structure, and then finally marshal that Python structure with CORBA for return to the client. Our prototype is less complex, requiring only that a python structure from serialised key/value pairs stored in memory is retrieved, after which CORBA can marshal the data for return.

Beyond comparing the two system architectures, it is interesting to note that the low latency system is only very slightly slower than the single node solution. Network latency is clearly not the dominant factor in determining the fetch times in small low latency systems, and using an implementation of Chord in a compiled language is likely to be an effective way to improve performance.

The large difference in fetch times between low and high latency networks, however,

confirms that network communication is likely to be the biggest barrier to fetch performance for a global information service, and that reducing the number of high latency network hops, by increasing routing table size, or by using latency aware routing, is likely to be the most effective way of improving performance when the system is deployed in this way.

6.4.10 Network Communication

The system's bandwidth usage is an indication of the system's scalability. It is also important that the system can share the network without disrupting other critical services. We will now examine the factors that influence the bandwidth consumption of the system. In all bandwidth tests, data is taken from a node in our 12 node testbed under deployment 3.

Factors that might affect bandwidth consumption include:

System Size A Chord node communicates primarily with its successor and predecessor to keep the ring stable, and so system size should not affect maintenance bandwidth, we seek to confirm this.

System Stability Recovering from a node loss means sending all data that node was responsible for to another node. Depending on the quantity of data stored, this may require a great deal of communication. We will examine bandwidth usage immediately after a node failure.

Data Storage Size The number of keys stored on each node increases the quantity of data that must be exchanged for data maintenance to synchronise the data on different nodes. We seek to qualitatively assess the communication cost of our data maintenance algorithm.

System Load An increased number of lookups leads directly to increased bandwidth usage. We will investigate how the system performs under load.

6.4.11 Impact of System Size on Network Communication

The impact of the system's size on a node's bandwidth usage is a limit on system scalability. A typical Chord node maintains contact with a fixed number of successor and predecessor nodes, plus a number of finger nodes which should vary according to system size.

However, since Chord routing tolerates incorrect finger pointers, we check the validity of only one finger node per maintenance interval. This is sufficient to maintain the routing layer, and by using such a method, we ensure maintenance bandwidth remains independent of system size.

We confirmed this by tracking the number of bytes sent during Chord maintenance, and found that in a stable system, Chord maintenance required 670 bytes, regardless of system size. Such low communication costs mean that it would be feasible to run the maintenance algorithm frequently, or to have a larger neighbour set, if either of these were desirable.

These figures examine bandwidth usage in a stable system, in which nodes do not leave or join. Changes in a node's neighbour set may increase bandwidth usage, as the Chord maintenance algorithm repairs the neighbour information. The impact of changes in system membership is examined in the next section.

6.4.12 Impact of System Stability on Bandwidth Consumption

The stability of the system will have a dramatic effect on its bandwidth usage. When a node fails, the data maintenance algorithm will attempt to recreate replicas of any data the failed node stored by copying a surviving replica to a new location.

To illustrate the effect of a node failure on bandwidth usage, we use our 12 node testbed, in which 1000 metadata items, which have an average size of 2KB, are stored. As the replication factor was set to two, this gives a total of 2000 items and 4 megabytes of data in the system.

Once the system was stable and all data is stored, two clients request randomly selected metadata items from the system. A single node is then caused to fail, and the effects of this failure are monitored. The measurements taken are annotated in figure 6.11, which shows the average bandwidth consumption over thirty second periods taken throughout the duration of the experiment.

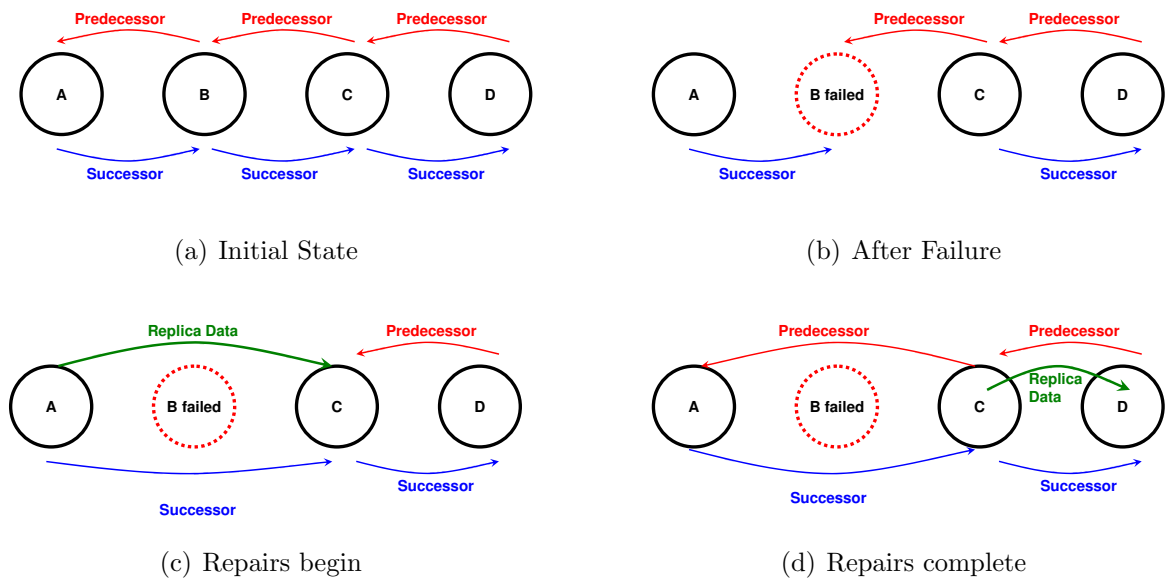


Fig. 6.10: Evolution of system state following a failure.

The initial state of the system surrounding the failed node can be thought of as being arranged as in 6.10(a). Node B was caused to fail, and the effects on the outgoing bandwidth from node C were recorded.

The system was in the state shown in figure 6.10(a) until 18:11, when node B was disconnected from the network. The system was then in the state shown in 6.10(b). This

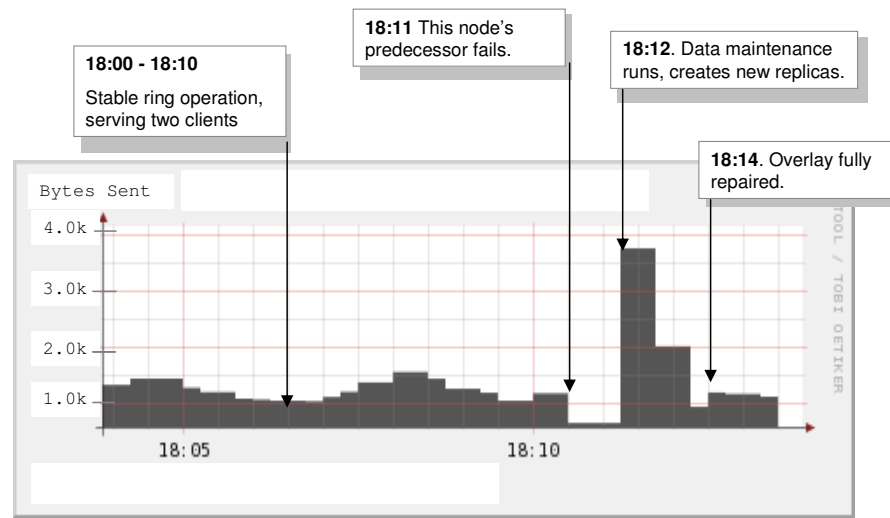


Fig. 6.11: Effect of node failure on bandwidth usage

caused an immediate reduction in the quantity of data being sent by node C, visible in figure 6.11. This is because requests are not routed correctly while the Chord ring is inconsistent.

After the failure, the Chord maintenance protocol ran. Node A becomes aware that its successor has failed, and updates its successor pointer. Node C also notes the failure of its predecessor. The order in which these two happen determines how rapidly the Chord overlay is repaired.

If node A runs Chord maintenance first, it will attempt to notify node C that it is now its predecessor. However, node C will not yet have noticed that node B has failed, and so it will refuse to change its predecessor pointer. If node C runs maintenance first, it will have noticed the failure of node B, and will accept node A as a predecessor.

In this case, node A runs its maintenance protocol first, and the first round of Chord maintenance ends in the situation shown in 6.10(c). If the data maintenance protocol had run at this point, node A would be able to place the replicas of its data that were previously stored on node B onto node C. Node C, however, would be unable to create new replicas of the data it was storing for node B, as it does not yet have the complete routing information necessary to decide which data it is responsible for.

The next time the Chord maintenance protocol ran, node A again attempted to notify node C that it is its predecessor. This time, the attempt is successful, and the Chord overlay was repaired to the state shown in 6.10(d). Now that the Chord ring has been repaired, the data maintenance protocol on node C can synchronise the data it is responsible for with its successor, node D. As node C has become responsible for the data previously stored on node B, it now sends a copy of this data to node D.

In this test, the second Chord maintenance call and the data maintenance call occur between 8:12 and 18:12.30'. The data maintenance then causes the spike in outgoing data seen at 18:12.30' in figure 6.11. After this, the ring is repaired entirely, and the rate of outgoing data is as it was before the failure.

A bandwidth spike is seen following a node failure. In this system, the data maintenance traffic sent by the successor of the failed node was approximately 175KB. This represents the one twelfth of the data in the system that had previously been replicated on nodes B and C, but is now stored on C and D. Node A also sent 160KB of data over the same interval, recreating the data which had previously been stored on nodes A and B, but is now stored on A and C.

From this test, we can conclude that bandwidth usage does spike following a node failure. Bandwidth usage following node failure is higher than at any other point we saw in our tests. The quantity of data sent will depend on the quantity of data stored on an average node in the system.

6.4.13 Impact of Quantity of Data Stored on Bandwidth Consumption

The quantity of data stored in the system affects the quantity of bandwidth needed to perform data maintenance. Our data maintenance algorithm simply sends a list of the keys stored on each node to those other nodes that store replicas for it. To assess the scalability of this approach, we store varying quantities of data in a 12 node information service, and record the outgoing bandwidth from one of the participant nodes. The results were then plotted against the number of data items that node was storing, as shown in figure 6.12.

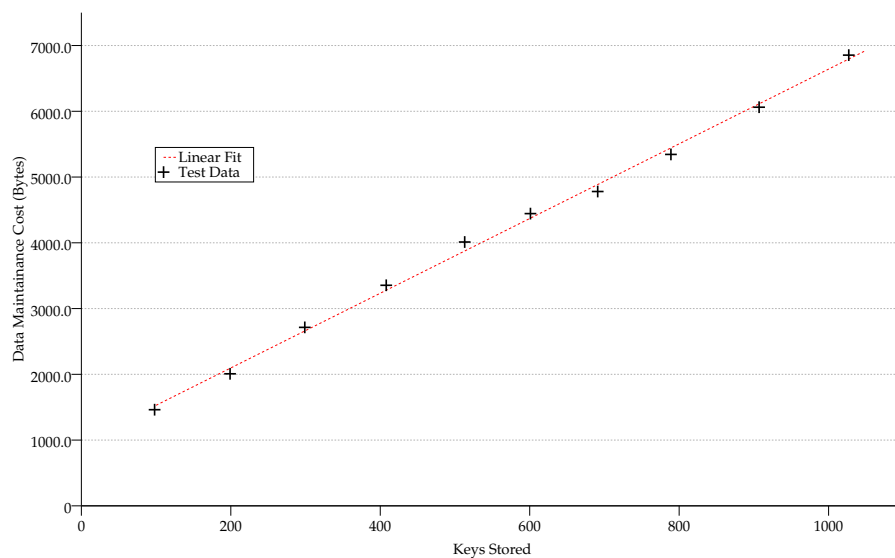


Fig. 6.12: Effect of data storage size on bandwidth usage

Extrapolating from this trend, we can predict that if the current CDF SAM-Grid database, consisting of the order of 1,000,000 items were to be stored on the 20 nodes currently functioning as stations, data maintenance would require in the region of 350KB. If data

maintenance is to run once every 45s, average bandwidth usage would be around 7.7K/s. This would result in a system-wide total of 13GB of data being sent every day.

Using techniques such as Merkle tree hashing, lossless compression or Bloom filter [92] exchange would reduce the size of this list and so the cost of data maintenance. A discussion of efficient data maintenance techniques may be found in [88].

6.4.14 Impact of System Load on Bandwidth Consumption

To test the impact of system load on bandwidth usage, bandwidth usage figures collected during the system load test described in the section 6.4.5 are used. In figure 6.13, we show the relationship between a nodes outgoing bandwidth, and the load on the system, expressed by the number of data items it is sending per second.

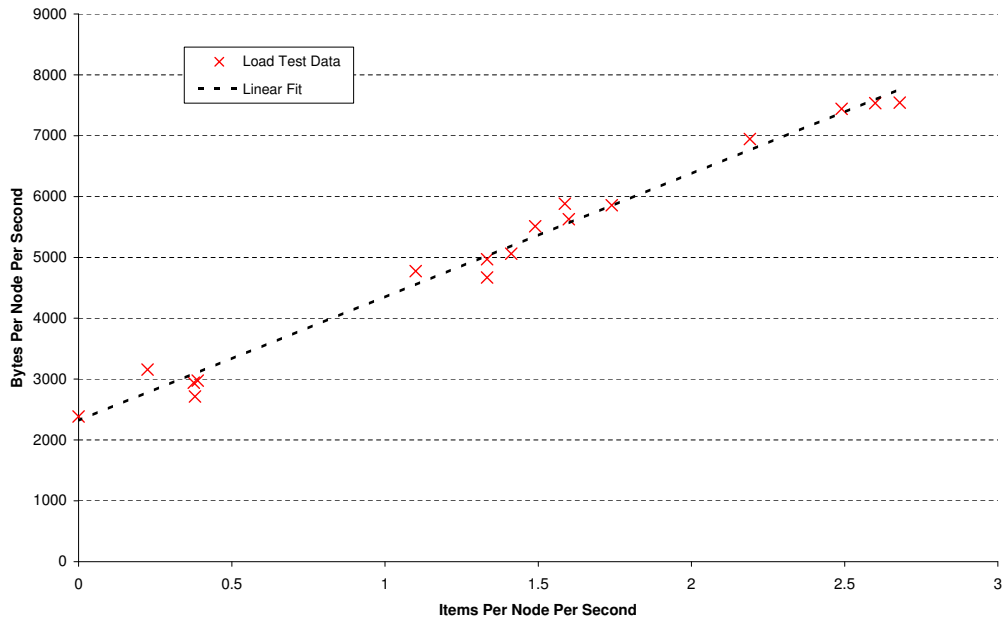


Fig. 6.13: Effect of system load on bandwidth usage

During the load tests, the system is being queried by up to 10 simultaneous clients. The system as a whole is satisfying up to 30 requests per second. This means a large number of lookups are routed through each node, and an increasing quantity of bandwidth must be used for routing purposes.

Figure 6.13 shows bandwidth consumption scales linearly with the load on the system (as indicated by the number of items being served per second). This indicates that communication is dominated by sending metadata items – which are about 2KB each. The quantity of

bandwidth consumed by the overhead of forwarding queries to other nodes remains constant throughout, at just over 2000 bytes/s. This indicates that routing bandwidth is unlikely to be a limiting factor to the number of clients the system may serve.

6.5 Summary

In this chapter, we have concentrated on combining peer to peer technologies with a Grid System. We have shown that although the peer to peer system cannot replace the SAM-Grid database, it can provide a level of service sufficient to enhance the reliability and scalability of the system.

We have given an architecture that allows peer to peer technologies to be successfully combined with an existing centralised Grid System. Our architecture combines a pass-through information system layer with a Chord ring, a data storage layer, and techniques for maintaining system consistency. We have implemented a limited prototype of this architecture, and deployed this prototype on a small testbed.

Using performance tests conducted with the prototype system, we have identified trends in lookup latency and bandwidth consumption. These trends indicate that the system has good performance in most areas, but have also highlighted some areas where performance could be improved. Where performance appeared less satisfactory, we have identified methods that can be used to improve this situation.

Our method of seamlessly combining a peer to peer lookup layer into a centralised architecture may prove useful in a wider context. If peer to peer storage is to be of immediate benefit to current Grid systems, it is important that it can be incorporated into existing designs. Our architecture illustrates how this may be achieved through the use of a pass-through layer. It is likely that this technique will be applicable to systems other than SAM-Grid.

The promising results we have recorded indicate that further work on a complete implementation of our architecture would be worthwhile. An obvious area for future work would be in implementing and testing the write caching mechanisms in order to decide on appropriate lock timeouts mechanisms. Our prototype did not explore write performance, which is likely to be considerably slower than read performance, and the technique of using leased locks to ensure consistency will need to be evaluated and carefully optimised. We would also like to explore the use of bloom filters as a data maintenance technique. Finally, the impact of lazy replica checking and asynchronous hash table insert on lookup times for uncached data need to be investigated.

6.6 Related Work

We believe that this work represents the first architecture which can provide persistent, reliable, peer to peer storage for an existing Grid system.

Cai *et al.* have described a DHT based replica location service [99], and given a practical assessment of its performance. The system they describe relies on soft-state storage and regular replica replacement to maintain reliability. Our work expands on this system by offering reliable self-repairing storage, and by giving a framework for consistent updates. This allows for the storage of mutable data, rather than simply replica locations. Our system also presents an architecture which works in tandem with an existing grid environment, without requiring changes to existing SAM-Grid software.

Andrzejak *et al.* [100] describe a peer to peer grid information system which allows for data to be located using range queries. The system uses a distributed hash table, and constructs order-preserving hash functions that map data items uniformly across the keyspace. Although we did not feel it necessary to provide a decentralised search facility for SAM-Grid, the methods presented here might provide a way of doing so. Further research would be necessary to determine if such an approach could offer good performance when searching with the large quantities of complex metadata used in SAM-Grid.

Cates [88], has given a practical study of the performance of the DHash protocol, and supplies similar figures to ours for bandwidth consumption when idle. Our performance evaluation complements this, studying performance in a more complex, grid based, setting and supplying additional details on the impact of system size and system load on bandwidth consumption, as well as a quantitative analysis of the performance of a naive data synchronisation method.

OpenDHT [101] is a public, Pastry based DHT service, and performance studies, and offers a practical performance assessment of that DHT. It also differs from our work in that it uses no maintenance protocol. Our work also offers a treatment of the affect of system size on latency, and a more thorough review of communication costs.

Our methods of ensuring consistency, uses locking to ensure consistent updates, and deferring to the database or a single owner in the case of updates, is specifically tailored to the SAM-Grid system. Other, more general systems have been proposed.

Etna [102] uses the Paxos algorithm to elect a primary node responsible for a given keyrange, and uses this primary to achieve consistency.

Ivy [89] uses per-participant logs, avoiding the issue of consistency by storing indefinitely, and storing updates as change logs. A user may choose whether or not to apply a given change log to an item.

7. CONCLUSIONS AND FURTHER WORK

7.1 *Conclusions*

This thesis set out to explore how we could create reliable peer to peer grid middleware. Using techniques including probability theory, simulation, and a real-world prototype, we have shown how peer to peer distributed hash tables provide an efficient and scalable substrate for designing middleware which alleviates many of the problems suffered by existing centralised systems.

In order to do this, we introduced a new algorithm for storing data in a distributed hash table, which we call dynamic replication. Our simulations and analysis show that this algorithm can provide both better performance and lower operating costs than similar existing algorithms.

In order to allow users to configure our algorithms appropriately, we introduced several powerful new techniques for analyzing the complex relationships between system size, replication factor, data maintenance frequency, and the reliability of a DHT based storage system. These techniques provide for the first time a systematic way of deciding the replication factor and maintenance frequency necessary to achieve a target level of system reliability.

We also provided detailed simulations which give the first analysis of the impact of replica placement on the reliability, performance, and bandwidth costs of a DHT storage system. Our simulations of the performance of these algorithms under churn indicate that they perform well, even under relatively infrequent maintenance, and that performance differs significantly between placement functions. Excellent scalability is observed. Simulations also suggest overhead bandwidth is unlikely to be significant, and that the main bandwidth cost will result from recovering from node failures. Recovering from node failures uses significantly less bandwidth under our new dynamic replication algorithm than under the DHash algorithm.

Finally, we have demonstrated how the peer to peer storage systems we have developed can be used to create practical information service middleware for grid computing by giving an architecture for, and deploying a prototype of, a SAM-Grid peer to peer grid information service. The prototype demonstrated the scalability and reliability properties we had hoped for, and strongly suggests that such techniques have the potential to be valuable to future grid computing systems.

7.2 Areas for Further Work

The work we have presented in the thesis so far presents numerous opportunities for further development. Some areas that might be particularly interesting include:

Reliability Models: It seems likely that block placement gives the maximum possible system reliability when used in systems where $N \bmod r = 0$. If this is indeed the case, we would like to prove it.

Further, although we have good models for the reliability of successor and block placement, we have had to fall back on simulation to calculate the reliability of finger placement and symmetric placement. It is not certain that closed forms can be found for the reliability of these systems — their identification, or a proof of their non-existence, would be an interesting area for further work.

Reliable Storage Simulations: Our simulations modelled network latency, but assumed a high bandwidth environment, in which the time taken to send the data in any message was assumed to be negligible. In lower bandwidth networks, or networks storing very large quantities of information per object, this is not the case. It should be possible to extend our simulation to model limited bandwidth situations. If this extension were implemented, it would be interesting to find the point at which the communications channels become saturated for various maintenance frequencies and quantities of data stored.

P2P information service: Our prototype information service did not allow for the storage of mutable data. If demand exists, we would like to test the performance and confirm the correctness of the lock based update method we propose.

There are also more open ended questions to be answered about the role peer to peer technology can play in providing middleware other than information services. A key bottleneck in some grid systems is the job management middleware, and it remains to be seen if peer to peer techniques can be applied here. Security and monitoring middleware might also benefit from peer to peer technologies.

In addition to those areas listed above, we have identified two other areas as particularly important areas for further work, and explored them in more detail. These areas are the use of erasure coding, and adaptation for diurnal churn, and make up the next two sections.

7.2.1 Applying Dynamic Replication To Erasure Coding

Two different approaches may be taken toward adding redundancy to our data storage. Simple replication stores multiple complete copies of the item associated with a given key. We refer to this approach as replication. Another approach, *Erasure Coding*, which we will introduce in this section, can also be used.

Erasure coding[103] divides an item into N fragments such that the item can be reconstructed from a subset of M of those fragments, where $M < N$. IDA (Information Dispersal

Algorithm) [103] is an erasure coding algorithm which can be used to generate fragments from the original data in such a way that any M fragments are very likely to contain enough unique information to reconstruct the original data. This property is useful, as if a fragment is lost, a new fragment may be generated from the original data and used without checking whether it is distinct from all existing fragments.

We have concentrated on creating reliable storage through the use of full replicas, for the sake of clarity and simplicity. The same techniques, however, could be applied to erasure coded fragments. To apply replication based techniques to erasure coded data, some modification will be required to allow for the original item to be reconstructed when it is necessary that new fragments are generated to replace those that have been lost.

Erasure coding allows high levels of availability to be achieved with a lower storage cost. For instance, using full replication, storing 10 replicas on nodes with a failure probability of 0.5 leads to an item loss probability of $\frac{1}{2^{10}}$, and a storage overhead of ten. If we use erasure coded fragments, ten of which are needed to reconstitute the original data, we need store only 40^1 fragments to achieve the same item failure probability. Each fragment is one tenth the size of the original, so the storage overhead is four. Erasure coding then is, in this example, 2.5 times as efficient in its use of storage space. [104]

The use of erasure coding can provide increased overall efficiency in some circumstances [104, 105], but at the cost of additional complexity, and an increased recovery cost. In order to recreate a lost fragment, the original data must be reassembled, requiring co-operation among at least M of the systems which stored the data. This means that in systems with high churn rates, erasure coding can increase maintenance bandwidth significantly.

Though our analysis of the reliability of the various placement algorithms cannot be applied to erasure coded data, monte-carlo simulation shows that the same relationships between placement functions apply to both erasure coded data and full replicas. Figure 7.1 shows the output of a monte carlo simulation, which illustrates that block placement still provides superior reliability to other placement functions when used with fragments. A plot showing the reliability of full replication systems with various replication factors is also shown for comparison.

7.2.2 Allowing for Diurnal Churn

We have, in our reliability models, and in our simulations, modelled *steady-state* environments, in which the rate at which nodes leave is equal to the rate at which they join. This is likely to be the case in a Grid environment, where the main cause of node departure is node failure, and the machine is likely to be repaired and replaced. If we wish to exploit user workstations, however, the main cause of node departure is likely to be machines being shut down when they are not in use by their owner. This means the number of machines in

¹ Using the formula: $k_c = (\frac{\sigma_\epsilon \sqrt{\frac{a(1-a)}{m}} + \sqrt{4a + \frac{\sigma_\epsilon^2 a(1-a)}{m}}}{2a})^2$, from [104]. Where m =fragments necessary to reconstruct the item, a =node failure probability, total number of fragments necessary = $k_c \cdot m$, and σ_ϵ is the number of standard deviations from the mean of an ($\bar{x} = 0, \sigma = 1$) normal distribution such that $p(x \leq \sigma_\epsilon)$ is the desired level of reliability.

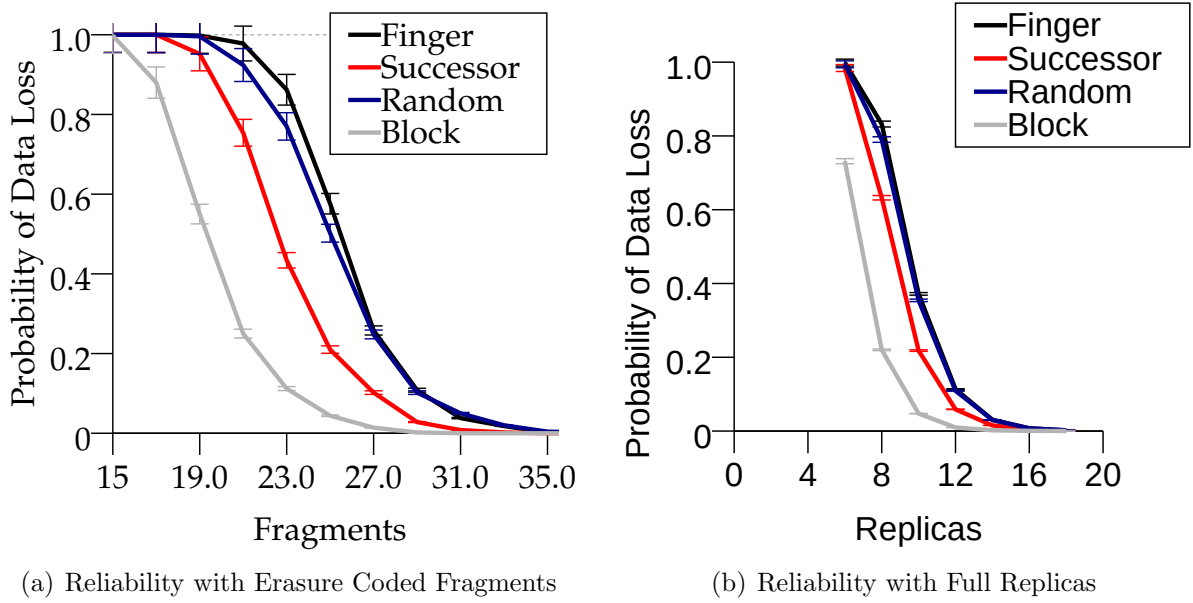


Fig. 7.1: Reliability of a 500 node system, in which each node fails with probability 0.5; In the Erasure Coded system, data is stored as a number of erasure coded fragments, 5 of which are needed to reconstruct the item. The system is considered to have failed if any item cannot be reconstructed.

the network, N , is likely to fluctuate throughout the day, leading to *diurnal churn*.

This fluctuation will cause the block, successor and predecessor algorithms to operate incorrectly, as all these placement patterns make use of the value of N in calculating replica locations, assuming it to be constant. Diurnal churn will mean that a constant value of N is likely to be incorrect. The effect of this varies depending on whether our estimate is too high, or too low:

N too high If our estimate of N is too high, we will underestimate the average keyspace size. This will increase the frequency of allocation collisions, as two adjacent replica locations are more likely to be owned by a single node.

N too low If our estimate of N is too low, we will overestimate the average keyspace size. This may mean that the nodes in the core group are not held in local routing tables, requiring extra routing overhead. It may also mean that the lookup path for an item does not converge onto the replica holders, and this will reduce the effects of preemptive returns.

The effect of our value of N becoming too low is to reduce the efficiency of the system. The effect of N becoming too high, however, is more severe – an increase in allocation collisions may mean that the R_{MAX} available replica locations do not encompass R_{MIN} distinct nodes, and the reliability of the system will decrease.

We envisage two possible methods for avoiding this scenario, which we will now describe briefly. An investigation into which of these is most efficient would be interesting:

Avoid Measurements of N

The positions at which replicas are stored under DHash, finger and symmetric replication does not depend on the value of N , and so these systems could function without modification in a diurnal churn environment – though finger and symmetric replication are likely to experience allocation collisions if N drops below some minimum value.

For other placement patterns, we can select a distance between replica locations, and a value of R_{MAX} , such that sufficient locations are available for the whole range of possible system sizes. However, this is likely to be inefficient when the system is at the lower end of the range of allowable system sizes.

Updating Measurements of N

If a central server can track membership of the network, it can provide up-to-date measurements of N to the current members of the network at regular intervals, allowing the use of block, successor and predecessor placement patterns. However, any central server may become a bottleneck, even when charged with such a simple task. Additionally, as the value of N changes, the nodes responsible for replicas of a given item may also change, leading to increased bandwidth usage as replicas are copied between nodes.

Another option would be to measure N in a distributed fashion, by sampling the distribution of nodes through some portion of the keyspace. All nodes should not take the sample from the same keyspace, as this will place an increased burden on the nodes in that keyspace leading to problems similar to that of using a central server. If we do not have every node sample the same area of the keyspace, each node will have a different measure of N . This will lead to each node mapping the replicas of a given key to different locations. It would therefore be necessary to associate each replica with a range of keys, representing the range of keys it might be mapped to – the *replica range*. More accurate measurements of N would lead to smaller replica ranges.

Whether the additional messaging overheads required to move replicas, and search the replica range would be outweighed by the ability to use block or predecessor placement patterns is unclear. Further study of the impact of such a system on lookup performance is necessary, and would be a good area for further work.

BIBLIOGRAPHY

- [1] Matthew Leslie and Sinisa Veseli. SAMGrid peer-to-peer information service. In *Proceedings of Computing in High Energy Physics 2006 (CHEP06)*, February 2006.
- [2] Matthew Leslie, Jim Davies, and Todd Huffman. Replication strategies for reliable decentralised storage. In *First International Workshop On Dependable and Sustainable Peer To Peer Systems (DASP2P06)*, April 2006.
- [3] Abhishek S. Rana. A globally distributed grid monitoring system to facilitate high-performance computing at D0/SAM-Grid (design, development, implementation and deployment of a prototype). Master's thesis, The University of Texas at Arlington, December 2002.
- [4] Gabriele Garzoglio. *A Globally Distributed System for Job, Data, and Information Handling for High Energy Physics*. PhD thesis, DePaul University Chicago, November 2005.
- [5] Ian T. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kauffman, November 1999.
- [6] Ian T. Foster. What is the grid? A Three Point Checklist. *GridToday*, July 2002.
- [7] Alan Beck. Larry Smarr On The Shape Of The Grid In 2003. *Article in GridToday*, December 2002.
- [8] Alan Greenberger. The computers of tomorrow. *The Atlantic Monthly*, May 1964.
- [9] Ann Chervenak, Ewa Deelman, Carl Kesselman, Bill Allcock, Ian Foster, Veronika Nefedova, Jason Lee, Alex Sim, Arie Shoshani, Bob Drach, Dean Williams, and Don Middleton. High-performance remote access to climate simulation data: a challenge problem for data grid technologies. *Elsevier Parallel Computing*, 29(10):1335–1356, October 2003.
- [10] J. Austin, T. Jackson, and M. Jessop. Diagnostics and prognostics on the grid: the distributed aircraft maintenance environment project (DAME). In *Proceedings of EPSRC All Hands Meeting*, September 2004.
- [11] D. J. Gavaghan, A. C. Simpson, S. Lloyd, D. F. Mac Randal, and D. R. S. Boyd. Towards a grid infrastructure to support integrative approaches to biological research. *Philosophical Transactions of the Royal Society: Mathematical, Physical and Engineering Science*, 363(1883):1829–1841, August 2005.

-
- [12] N. V. Krasnikov and V. A. Matveev. Physics at the Large Hadron Collider. *Physics of Particles and Nuclei*, 28:441–470, September 1997.
 - [13] The CDF II Detector, Technical Design Report, FERMILAB-PUB-96/390-E. Technical report, The CDF II Collaboration, October 1996.
 - [14] Abazov, V. et al. The upgraded DØ detector. *Nuclear Instruments and Methods in Physics Research A*, 565:463–537, September 2006.
 - [15] Gene Oleynik, Bonnie Alcorn, Wayne Baisley, Jon Bakken, David Berg, Eileen Berman, Chih-Hao Huang, Terry Jones, Robert D. Kennedy, Alexander Kulyavtsev, Alexander Moibenko, Timur Perelmutov, Don Petravick, Vladimir Podstavkov, George Szmuksta, and Michael Zalokar. Fermilab’s multi-petabyte scalable mass storage system. *22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST’05)*, 00:73–80, April 2005.
 - [16] LHC Computing Grid Technical Design Report , LCG-TDR-001 CERN-LHCC-2005-024. Technical report, The LCG TDR Editorial Board, June 2005.
 - [17] Ian T. Foster. Globus toolkit version 4: Software for service-oriented systems. In *Networks and Parallel Computing*, pages 2–13. Springer-Verlag, November 2005.
 - [18] David P. Anderson. BOINC: A system for public-resource computing and storage. In *GRID ’04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID’04)*, pages 4–10, November 2004.
 - [19] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *CCGRID ’01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, May 2001.
 - [20] J. Bradley, C. Brown, B. Carpenter, V. Chang, J. Crisp, S. Crouch, D. De Roure, S. Newhouse, G. Li, J. Papay, C. Walker, and A. Wookey. The OMII software distribution. In *Proceedings of the UK e-science All hands meeting*, September 2006.
 - [21] International Telecommunication Union. Recommendation X.509: Information Technology - Open Systems Interconnection - The Directory: Authentication Framework. June 1997.
 - [22] A. Baranovski, I. Bertram, G. Garzoglio, L. Lueking, I. Terekhov, S. Veseli, and R. Walker. The SAM-GRID project: architecture and plan. In *Nuclear Instruments and Methods in Physics Research, Section A (Elsevier Science), Proceedings of ACAT’2002*, June 2002.
 - [23] A. Baranovski, I. Bertram, G. Garzoglio, L. Lueking, I Terekhov, S. Veseli, and R Walker. SAM-grid: Using SAM and Grid middleware to enable full function Grid Computing. In *Proceedings of the 8th International Conference on B-Physics at Hadron Machines (Beauty 02), Santiago de Compostela, Spain*, June 2002.

-
- [24] A. Baranovski, G. Garzoglio, Arthur Kreymer, Lee Lueking, Stefan Stonjek, Igor Terekhov, Frank Wuerthwein, Alain Roy, P. Mhashikar, V. Murthi, Todd Tannenbaum, Rod Walker, Fedor Ratnikov, and T. Rockwell. Management of grid jobs and information within SAMGrid. *Proceedings of Computing in High Energy Physics 2003 (CHEP03)*, March 2003.
- [25] A. Lyon, P. Vokac, M. Zimmerler, A. Baranovski, G. Garzoglio, L. Loebel-Carpenter, R. Herber, R. Illingworth, R. Kennedy, A. Kreymer, A. Kumar, L. Lueking, W. Merritt, I. Terekhov, J. Trumbo, S. White, S. Veseli, M. Burgon-Lyon, R. StDenis, S. Belforte, U. Kerzel, M. Leslie, V. Bartsch, S. Stonjek, F. Ratnikov, and A. Sill. The SAM-Grid monitoring service and its integration with MONALISA. *Proceedings of Computing in High Energy Physics 2003 (CHEP03)*, 2003.
- [26] A. Lyon, A. Baranovski, G. Garzoglio, L. Loebel-Carpenter, R. Herber, R. Illingworth, R. Kennedy, A. Kreymer, A. Kumar, L. Lueking, W. Merritt, I. Terekhov, J. Trumbo, S. White, S. Veseli, M. Burgon-Lyon, R. StDenis, S. Belforte, U. Kerzel, M. Leslie, V. Bartsch, S. Stonjek, F. Ratnikov, and A. Sill. Application of the SAM-Grid test-harness for performance evaluation and tuning of a distributed cluster implementation of data handling services. *Proceedings of Computing in High Energy Physics 2003 (CHEP03)*, 2003.
- [27] Gomez-Ceballos G., Belloni A., Bolshov A., Iyutin B., Kravchenko I., Leonardo N., Knuteson B., Paus C, Tether S., Tseng J., and Wurthwein F. Event builder and level 3 at the cdf experiment. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 518:522–524(3), 1 February 2004.
- [28] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA (Common Object Request Broker Architecture)*. Prentice-Hall, Inc., 1996.
- [29] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, August 2001.
- [30] Gabriele Garzoglio, Andrew Baranovski, Parag Mhashikar, Ljubomir Perkovic, and Anoop Rajendra. A case for application-aware grid services. In *Proceedings of Computing in High Energy Physics 2006 (CHEP06)*, February 2006.
- [31] G.A. Stewart and G. McCance. Grid data management: Reliable file transfer services' performance. In *Proceedings of Computing in High Energy Physics 2006 (CHEP06)*, February 2006.
- [32] Ian T. Foster and Adriana Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS02)*, March 2002.

-
- [33] Jonathan Ledlie, Jeffrey Shneidman, Margo I. Seltzer, and John Huth. Scooped, again. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS03)*, February 2003.
 - [34] Dreamtech Software Team. *Cracking The Code: P2P Application Development*. Published by Hungry Minds, 2002.
 - [35] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.
 - [36] K. Aberer and M. Hauswirth. An overview of peer-to-peer information systems. In *Distributed Data & Structures 4, Records of the 4th International Meeting (WDAS 2002)*, Paris, France, March 2002.
 - [37] Clay Shirky. *Peer-to-Peer: Harnessing the power of disruptive technologies*, chapter 8: Gnutella. O'Reilly and Associates, 2001.
 - [38] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, August 2001.
 - [39] Rüdiger Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the IEEE 2001 International Conference on Peer-to-Peer Computing (P2P2001)*, pages 101–102, August 2001.
 - [40] Pete Loshin. *Big Book of Border Gateway Protocol (BGP) RFCs*. Morgan Kaufmann, 2000.
 - [41] Henry Spencer and David Lawrence. *Managing Usenet*. O'Reilly, 1998.
 - [42] Bo Leuf. *Peer To Peer Collaboration and Sharing Over The Internet*, chapter 7. Addison Welsey, 2002.
 - [43] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems*, 9(2):170–184, August 2003.
 - [44] Beverly Yang and Hector Garcia-Molina. Comparing hybrid peer-to-peer systems. In *The VLDB Journal*, pages 561–570, September 2001.
 - [45] Adriana Iamnitchi, Matei Ripeanu, and Ian Foster. Small-world file-sharing communities. In *The 23rd Conference of the IEEE Communications Society (InfoCom 2004)*, March 2004.
 - [46] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks, 2002. In proceedings of the ACM SIGCOMM'02 Conference, August 2002.

-
- [47] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1):50–57, February 2002.
 - [48] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 258–259, June 2002.
 - [49] R. Albert, H. Jeong, and A. Barabási. The Internet’s Achilles’ heel: Error and attack tolerance of complex networks. *Nature Issue 406, Pages 378-38*, July 2000.
 - [50] Nick Blundell and Laurent Math. An overview of gnutella optimisation techniques. In *Proceeding of the 3rd Annual Postgraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet 2002)*, UK, June 2003.
 - [51] Daniel Stutzbach and Reza Rejaie. Characterizing the two-tier gnutella topology. In *Proceedings of the ACM SIGMETRICS*, June 2005.
 - [52] Alexander Klemm, Christoph Lindemann, Mary K. Vernon, and Oliver P. Waldhorst. Characterizing the query behavior in peer-to-peer file sharing systems. In *4th ACM SIGCOMM conference on Internet measurement (IMC '04)*, pages 55–67, October 2004.
 - [53] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. Technical Report TR-2001-26, University of Chicago, July 2001.
 - [54] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *Proceedings of IEEE INFOCOM, 2004.*, March 2004.
 - [55] Andrea Ceccanti and Gian Paolo Jesi. Building latency-aware overlay topologies with QuickPeer. In *Proceedings of the Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services (ICAS/ICNS 2005)*, pages 24–29, October 2005.
 - [56] Michael J. Freedman and Radek Vingralek. Efficient peer-to-peer lookup based on a distributed trie. In *Proceedings of the 1st International Workshop on Peer-To-Peer Systems (IPTPS)*, March 2002.
 - [57] Chi Zhang, Arvind Krishnamurthy, and Randolph Y. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *Proceedings of the 4th International Workshop on Peer-To-Peer Systems (IPTPS'05)*, pages 47–57, February 2005.
 - [58] C. Law and K. Siu. Distributed construction of random expander networks. In *Proceedings of IEEE Infocom*, April 2003.
 - [59] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference*, August 2001.

-
- [60] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.
 - [61] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.
 - [62] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal hash table. In *Proceedings of the 2nd International Workshop on Peer-To-Peer Systems (IPTPS'03)*, February 2003.
 - [63] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-To-Peer Systems (IPTPS'03)*, February 2003.
 - [64] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, pages 183–192, August 2002.
 - [65] G S Manku, M Bawa, and P Raghavan. Symphony: Distributed hashing in a small world. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems*, pages 127–140, March 2003.
 - [66] Jon Kleinberg. The small-world phenomenon: an algorithm perspective. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 163–170, May 2000.
 - [67] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-To-Peer Systems (IPTPS'03)*, February 2003.
 - [68] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and M. Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the 3rd International Workshop on Peer-To-Peer Systems (IPTPS'04)*, February 2004.
 - [69] Sushant Jain, Ratul Mahajan, and David Wetherall. A study of the performance potential of dht-based overlays. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 11–25, March 2003.
 - [70] S. Rhea, T. Roscoe, and J. Kubiatowicz. Structured Peer-to-Peer overlays need application-driven benchmarks. In *Proceedings of the 2nd International Workshop on Peer-To-Peer Systems (IPTPS'03)*, February 2003.
 - [71] Daniel Stutzbach and Reza Rejaie. Improving lookup performance over a widely-deployed DHT. In *Proceedings of IEEE INFOCOM*, April 2006.

-
- [72] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the 1st International Workshop on Peer-To-Peer Systems (IPTPS)*., March 2002.
 - [73] J. Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-To-Peer Systems (IPTPS)*, March 2002.
 - [74] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI02)*, December 2002.
 - [75] Amos Fiat, Jared Saia, and Maxwell Young. Making chord robust to byzantine attacks. In *Proc. of the European Symposium on Algorithms (ESA), 2005*, pages 803–814, October 2005.
 - [76] George Danezis, Chris Lesniewski-Laas, M. Frans Kaashoek, and Ross Anderson. Sybil-resistant DHT routing. In *Proceedings of the 10th European Symposium On Research In Computer Security*, September 2005.
 - [77] Hosam Rowaihy, William Enck, Patrick McDaniel, and Thomas La Porta. Limiting Sybil Attacks in Structured Peer-to-Peer Networks. Technical Report NAS-TR-0017-2005, Network and Security Research Center, July 2005.
 - [78] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.
 - [79] Ben Leong, Barbara Liskov, and Erik D. Demaine. Epichord: Parallelizing the chord lookup algorithm with reactive routing state management. Technical Report MIT-LCS-TR-963, MIT, August 2004.
 - [80] Bryan Ford. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference*, April 2005.
 - [81] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and DHTs. In *Proc. 2nd Workshop on Real, Large, Distributed Systems (WORLDS 05)*, December 2005.
 - [82] Bruce Schneier. *Applied Cryptography, Section 18.5: MD5*. Wiley, 2nd edition, October 1995.
 - [83] Federal Information Processing Standards (FIPS) Publication 180-1, Secure Hash Standard (SHS). Technical report, U.S. DoC/NIST, April 17 1995.
 - [84] V. Ramasubramanian and E. Sirer. Beehive: Exploiting power law query distributions for $O(1)$ lookup performance in peer to peer overlays. In *Symposium on Networked Systems Design and Implementation, San Francisco CA.*, March 2004.

-
- [85] E. Sit, A. Haeberlen, F. Dabek, B. Chun, H. Weatherspoon, R. Morris, M. Kaashoek, and J. Kubiatowicz. Proactive replication for data durability. In *Proceedings of the 5th International Workshop on Peer-To-Peer Systems (IPTPS'06)*, February 2006.
 - [86] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gum-madi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.
 - [87] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with Chord, a distributed lookup service. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, May 2001.
 - [88] Josh Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.
 - [89] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.
 - [90] Ralph C. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 Symposium on Security and Privacy*. IEEE Computer Society, April 1980.
 - [91] Marcel Waldvogel, Paul Hurley, and Daniel Bauer. Dynamic replica management in distributed hash tables. Research Report RZ-3502, IBM, July 2003.
 - [92] G. L. Peterson. Time-space trade-offs for asynchronous parallel models (reducibilities and equivalences). In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, April 30th - May 2nd 1979.
 - [93] Andreas Binzenhöfer, Dirk Staehle, and Robert Henjes. Estimating the size of a Chord ring. Technical Report 348, University of Würzburg Research Report, November 2004.
 - [94] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In *The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, August 2005.
 - [95] W Feller. *An Introduction to Probability Theory and Its Application, Vol. 1*, page 300. Wiley, 2nd edition, 1968.
 - [96] David Oppenheimer, Jeannie Albrecht, David Patterson, and Amin Vahdat. Distributed resource discovery on PlanetLab with SWORD. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*, December 2004.
 - [97] Klaus Muller. Advanced systems simulation capabilities in SimPy. In *Proceedings of Europython*, 2004.

-
- [98] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.
 - [99] Min Cai, Ann Chervenak, and Martin Frank. A Peer-to-Peer Replica Location Service Based on a Distributed Hash Table. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 56, November 2004.
 - [100] Artur Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. In *P2P '02: Proceedings of the Second International Conference on Peer-to-Peer Computing*, page 33, September 2002.
 - [101] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiawicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A Public DHT Service and Its Uses. In *Proceedings of ACM SIGCOMM*, 2005.
 - [102] Athicha Muthitacharoen, Seth Gilbert, and Robert Morris. Etna: a fault-tolerant algorithm for atomic mutable DHT data. MIT-LCS Technical Report MIT-LCS-TR-993, MIT, June 2005.
 - [103] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.
 - [104] Rodrigo Rodrigues and Barbara Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In *Proceedings of the 4th International Workshop on Peer-To-Peer Systems (IPTPS'05)*, February 2005.
 - [105] Zheng Zhang and Qiao Lian. Reperasure: Replication protocol using erasure-code in peer-to-peer storage. In *21st Symposium on Reliable Distributed Systems*, October 2002.