KTH Information and
Communication Technology

Designs and Analyses in Structured Peer-to-Peer Systems

Sameh El-Ansary

A Dissertation submitted to
the Royal Institute of Technology (KTH)
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

June 2005

The Royal Institute of Technology (KTH)
School of Information and Communication Technology
Department of Microelectronics and Information Technology
Stockholm, Sweden

TRITA-IMIT-LECS AVH 05:02
ISSN 1651-4076
ISRN KTH/IMIT/LECS/AVH-05/02–SE
and
SICS Dissertation Series 38
ISSN 1101-1335
ISRN SICS-D–38–SE

Printed by Universitetsservice US-AB 2005

To My Beloved Wife Mena

Abstract

Peer-to-Peer (P2P) computing is a recent hot topic in the areas of networking and distributed systems. Work on P2P computing was triggered by a number of ad-hoc systems that made the concept popular. Later, academic research efforts started to investigate P2P computing issues based on scientific principles. Some of that research produced a number of structured P2P systems that were collectively referred to by the term "Distributed Hash Tables" (DHTs). However, the research occurred in a diversified way leading to the appearance of similar concepts yet lacking a common perspective and not heavily analyzed. In this thesis we present a number of papers representing our research results in the area of structured P2P systems grouped as two sets labeled respectively "Designs" and "Analyses".

The contribution of the first set of papers is as follows. First, we present the principle of distributed k-ary search and argue that it serves as a framework for most of the recent P2P systems known as DHTs. That is, given this framework, understanding existing DHT systems is done simply by seeing how they are instances of that framework. We argue that by perceiving systems as instances of that framework, one can optimize some of them. We illustrate that by applying the framework to the Chord system, one of the most established DHT systems. Second, we show how the framework helps in the design of P2P algorithms by two examples: (a) The $\mathcal{DKS}(n;k;f)$ system which is a system designed from the beginning on the principles of distributed k-ary search. (b) Two broadcast algorithms that take advantage of the distributed k-ary search tree.

The contribution of the second set of papers is as follows. We account for two approaches that we used to evaluate the performance of a particular class of DHTs, namely the one adopting periodic stabilization for topology maintenance. The first approach was of an intrinsic empirical nature. In this approach, we tried to perceive a DHT as a physical system and account for its properties in a size-independent manner. The second approach was of a more analytical nature. In this approach, we applied the technique of Master Equations, which is a widely used technique in the analysis of natural systems. The application of the technique lead to a highly accurate description of the behavior of structured overlays.

Additionally, the thesis contains a primer on structured P2P systems that tries to capture the main ideas prevailing in the field and enumerates a subset of the current hot and open research issues.

Acknowledgments

I had the privilege to work with many experienced senior persons during my research and to whom I would like to offer my thanks.

First, I would like to thank my supervisor Prof. Seif Haridi for offering me the chance of being a member of his distinguished research team. Seif's enthusiasm was a strong source of encouragement. He was always keen to share his long experience with me and to teach me new things. He continuously and successfully exerts lots of effort to provide the best research environment and to open new horizons for myself and to all of his students.

Second, I would like to express my gratitude to Dr. Luc Onana Alima. Luc introduced me to the field of distributed algorithms as a teacher. He, then, worked with me as a partner and co-supervised the first part of this thesis. Luc has been a serious working partner who offered maximum moral support and pushed me to the limit whenever needed.

Third, during the second part of the thesis I was co-supervised by Prof. Erik Aurell and Dr. Supriya Krishnamurthy. Erik showed me how physicists do dimensional analysis for systems. He was a constant source of support and inspiration and acted as an example in efficient and hard work. Finally, the supervision of Supriya at the end of the thesis opened to me a complete new scope of thinking by introducing me to the technique of Master Equations. She showed me how systems of intricate complexity could be analyzed with high accuracy using very basic probabilistic primitives.

I would like also to thank the team leader of the DSL lab at SICS Per Brand. Per taught me distributed programming in Mozart. Moreover, he has always been a very inspiring person. His strong intuition and shrewd remarks have always opened the gate for new ideas.

I would like to thank my colleagues in the DSL lab at SICS, Konstantin Popow, Erik Klintskog, Dragan Havelka, Fredrik Holmgren, Frej Drejhammar, Joe Armstrong for being a constant source of help. Ali Ghodsi was a colleague with whom I had lots of enlightening discussions. We experienced together the stress of weekend- and late-night-working in order to conduct simulations and write papers.

On the personal level, I would like to thank my wife, my mother, my father and Prof. Ahmed Rafea. If I was able to finish my PhD, that is because of them. Dr. Mahmoud Rafea has been the friend who shared with me the joys and the hardships of the first two years of living in Sweden.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Motivation

How can we reason better about structured Peer-to-Peer (P2P) overlay networks? This question was always in the background while our research team was observing the quickly-evolving and diversified results in the emerging field of structured P2P systems. In this thesis, we report two main principles that we followed and that led us to a better reasoning about structured P2P systems. These two principles are:

- Distributed k-ary search as a common foundation of a major class of structured P2P systems.
- The perception of structured P2P systems as physical systems for better analysis.

The thesis is composed of two parts each corresponding to the research results obtained by applying the respective principle. The thesis title (*Designs and Analyses in Structured P2P Networks*) reflects the two different natures of its two parts. "*D*esigns" is the name of the first part since the principle of distributed k-ary search was helpful in the design of the $\mathcal{DKS}$ system as well as additional services such as the efficient and self-correcting broadcast algorithms. "*A*nalyses" is the name of the second part, where the focus was not on designing new systems/algorithms but rather on a meticulous analysis of already-existing structured overlays.

This thesis summarizes the research efforts of its author as a member of a team from SICS and KTH researching structured peer-to-peer systems in the context of the European projects PEPITO (IST-2001-33234) and EVERGROW (IST-2004-001935) as well as the Swedish Vinnova projects PPC and AMRAM.

1.2 Thesis Organization

The thesis is written in the "collection-of-papers" style. Since each paper has a number of authors, we report here the individual contribution of the thesis author in each paper.

Chapter 1: Introduction. In this chapter we explain the motivation of the thesis and its organization.

Chapter 2: Structured P2P Overlay Networks Primer. In this chapter, first, we give an idea about the evolution of P2P systems in general. Second, we focus on structured peer-to-peer systems by enumerating some of the prominent systems in the field and explaining the basic principles of their operation. Finally, we enumerate some of the current hot research topics. The chapter is a version of:

> Sameh El-Ansary and Seif Haridi, *An Overview of Structured P2P Overlay Networks*, Book chapter to appear in the upcoming book: Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks, (Editor: Prof. Jie Wu), CRC Press, July 2005.

Thesis Author Contribution: This chapter required the reading and selection of papers as well as devising comparison criteria and gathering of open issues. The thesis author has performed all of the above activities and written this chapter under the supervision of Prof. Seif Haridi.

Part I: Designs

Chapter 3: A Framework for Peer-To-Peer Lookup Services Based On k-ary Search. This chapter contains the first technical report where we tackle the issue of a common framework for the understanding of DHT systems. We show the importance of the framework by showing that the perception of the Chord system as an instance of the distributed k-ary search framework leads to a substantial optimization in its performance. The work is published in:

> Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. A Framework for Peer-To-Peer Lookup Services Based On k-ary Search. Technical Report TR-2002-06, SICS, May 2002.

Thesis Author Contribution: The thesis author together with Luc Onana co-formulated the common framework based on the idea of distributed k-ary search and co-applied the framework to optimize the Chord system. The work was done under the supervision of Luc Onana, Per Brand and Seif Haridi.

Chapter 4: The $\mathcal{DKS}(N, k, f)$ Infrastructure for P2P Applications. In this paper, we show the design of the $\mathcal{DKS}$ system which is a DHT system designed from the beginning on the principles of distributed k-ary search. We also show by means of simulation that the "correction-on-use" technique that is introduced in this paper is feasible provided that enough lookups are taking place in the overlay. The work is published in:

> Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. $\mathcal{DKS}(N, k, f)$: A Family of Low Communication, Scalable and Fault-tolerant Infrastructures for P2P Applications. In *the 3rd International Workshop on Global and Peer-To-Peer Computing on Large-scale Distributed Systems - CCGRID2003*, Tokyo, Japan, May 2003.

Thesis Author Contribution: The $\mathcal{DKS}$ system is designed by Luc Onana. The role of the thesis author in this paper was to im-

plement the $\mathcal{DKS}$ system using a component-based simulation environment and to design simulations to show the validity of the various properties offered by its design; most importantly the ability of correction-on-use to act as the sole correction technique. The work was done under the supervision of Luc Onana, Per Brand and Seif Haridi.

Chapter 5: Efficient Broadcast in Structured P2P networks. This chapter contains a paper that, first, emphasizes the perception of a class of DHT systems as an instance of the distributed k-ary search framework. Second, shows that this perception can be used to build an efficient broadcast algorithm with optimal messaging cost by traversing the distributed k-ary search tree. The work published in:

Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient broadcast in structured P2P networks. In *the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

Thesis Author Contribution: The initial idea of performing broadcasts in a structured network is of Luc Onana. The thesis author contributed with the following: i) Suggested the exploitation of the structured topology for minimizing the number of messages, ii) Co-designed with Luc the broadcast algorithm, iii) Implemented the algorithm and designed the simulation experiments required for the evaluation of the algorithm. The work was done under the supervision of Luc Onana, Per Brand and Seif Haridi.

Chapter 6: This chapter contains a second paper on efficient broadcasting where we combine broadcasting with the correction-on-use technique from chapter 4 to make the broadcast correct the overlay. The work is published in:

Ali Ghodsi, Luc Onana Alima, Sameh El-Ansary, Per Brand and Seif Haridi, Self-Correcting Broadcast in Distributed Hash Tables, In *the 15th International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, Marina del Rey, CA, USA, November 3-5, 2003.

Thesis Author Contribution: This paper involved algorithm design, metric design and implementation. The algorithm design is mainly that of Luc Onana. The thesis author: i) Designed the "Distance-from-optimal-Network" metric used for observing the convergence of the network. ii) Co-simulated with Ali Ghodsi the algorithm on the simulation environment designed by the thesis author. The work was done under the supervision of Luc Onana, Per Brand and Seif Haridi.

Chapter 7: A Component-based Simulation environment. In this chapter, we describe the architecture of the simulation environment designed by the thesis author and that was used throughout the above papers. The environment adopts a component-based architecture building on previous experiences available at the DSL lab at SICS.

Part II: Analyses

Chapter 8: Physics-inspired Performance Evaluation of DHTs. This chapter combines three publications in which we perceive a structured overlay as a physical system and try to find intensive (size-independent) variables that describe its behavior. The main result of this work is the size-independent description of the performance of a structured overlay as a function of the ratio of perturbation (joins/failures) to stabilization. The description is obtained empirically from extensive simulation. The three summarized publications are:

Erik Aurell and Sameh El-Ansary, A Physics-Style Approach to Scalability of Distributed Systems. In *the LNCS Post-Proceedings of the Global Computing 2004 Workshop*, Rovereto, Italy, March 2004.

Sameh El-Ansary, Erik Aurell, Per Brand and Seif Haridi, Experience with a physics-style approach for the study of self properties in structured overlay networks, In *the International Workshop on Self-* Properties in Complex Information Systems*, Bertinoro, Italy, May 2004.

Sameh El-Ansary, Erik Aurell and Seif Haridi, A Physics-inspired Performance Evaluation of a Structured Peer-to-Peer Overlay Network, In *the International Conference on Parallel and Distributed Computing and Networks (PDCN 2005)*, Innsbruck, Austria, February 2005.

Thesis Author Contribution: Erik Aurell suggested the idea of searching for intensive variables in structured overlays. The thesis author suggested using the population density and the ratio of perturbation to stabilization as candidates for intensive variables. All the simulation and analysis activities were performed by the thesis author. The work was done under the supervision of Erik Aurell, Per Brand and Seif Haridi.

Chapter 9: Analytical Study of DHTs under Churn. In this chapter, we take the performance analysis of structured overlays beyond empirical observations. We present a complete analytical study of a structured overlay undergoing perturbation using a Master Equations -based approach. The technique of Master Equations is traditionally used in non-equilibrium statistical mechanics to describe steady-state or transient phenomena. Simulations are used to verify all theoretical predictions instead of being the primary investigation tool as is the case in chapter 7. We also discuss briefly how churn may actually be of different types and the implications this will have on the functioning of DHTs in general. The work was reported in the following

two publications where the paper is a small of version of the technical report:

Sameh El-Ansary, Supriya Krishnamurthy, Erik Aurell and Seif Haridi, An Analytical Study of Consistency and Performance of DHTs under Churn. Technical Report TR-2004-12, SICS, October 2004.

Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell and Seif Haridi, A Statistical Theory of Chord under Churn. In *the 4th Annual International Workshop on Peer-To-Peer Systems (IPTPS 05)*, Ithaca, NY, USA, February 2005.

Thesis Author Contribution: The idea of trying to analytically derive the functional form of the cost-performance trade-off curve is that of the thesis author. The actual derivation of the functional form using Master Equations was *entirely* done by Supriya Krishnamurthy. The thesis author's role was to: i) Decide the quantities that are interesting to analyze, ii) Come up with a chord implementation that is capable of validating the model suggested by Supriya and making sure that the simplifications of the model are not too unrealistic, ii) Guide the presentation of the results to make it palatable to a computer science audience. Finally, Erik Aurell and Sameh cooperated to independently validate and slightly refine the analytical results and situated the work by comparing it to related research results. The work was done under the supervision of Supriya Krishnamurthy, Erik Aurell and Seif Haridi.

Chapter 2

A Structured P2P Overlay Networks Primer

2.1 What is P2P?

Like any new trend that is undergoing evolution, Peer-To-Peer systems do not have a precise definition, instead, many definitions were developed trying to reflect the new features of some phase in the evolution process. The following are some definitions presented in the P2P literature:

> **Oram:** P2P is a class of applications that takes advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, P2P nodes must operate outside the DNS system and have significant or total autonomy from central servers [47, 46].
>
> **Miller:** P2P is a network architecture in which each computer has equivalent capability and responsibility. This is in contrast to the traditional client/server network architecture, in which one or more computers are dedicated to serving the others. However, we need a more complex definition: P2P has five

> key characteristics. (i) The network facilitates real-time transmission of data or messages between the peers. (ii) Peers can function as both client and server. (iii) The primary content of the network is provided by the peers. (iv) The network gives control and autonomy to the peers. (v) The network accommodates peers that are not always connected and that might not have permanent Internet Protocol (IP) addresses [41].

> **P2P Working Group**: P2P computing is the sharing of computer resources and services by direct exchange between systems. These resources and services include the exchange of information, processing cycles, cache storage, and disk storage for files. Peer-to-peer computing takes advantage of existing desktop computing power and networking connectivity, allowing economical clients to leverage their collective power to benefit the entire enterprise. [24]

As one can observe from the different definitions, there is a strong consensus on some concepts such as: Resource sharing, autonomy/decentralization, dynamic IP addresses, and a client-and-server dual role.

2.2 Evolution of P2P Systems

The term Peer-to-Peer is a relatively new term in the areas of networking and distributed systems. According to Oram, P2P computing started to be a hot topic by the middle of the year 2000 [47]. During its short history, P2P passed through several generations. Transitions through generations were motivated by different goals. While most surveys merge what we present as first and second generations, we distinguish them to highlight different transition motives.

2.2.1 First Generation

Basic Idea

The first generation of P2P systems started with the appearance of the file-sharing application Napster [47, 44, 45]. The main contribution of Napster was the introduction of a network architecture where machines are not categorized as client and server but rather as machines that offer and consume resources. Consequently, the term "Peer" was a suitable term for a participant in that system as all participants are more or less of equal functionality. However, in order for machines to locate files in the shared space, Napster's solution was to provide a central directory. That is, the Napster system was composed of two services, a storage service and a directory service. The storage was decentralized and functioning in a Peer-to-Peer style while the directory service was centralized. A participant in a Napster network also had two main characteristics: i) A dynamic Internet address, and ii) Freedom to join and leave the network at any time.

Discussion

The Napster system faced problems that led to its decay as a mainstream P2P system. The main problem was a political problem, due to the copyrighted music files that were illegally shared among participants of the system. Legal problems hindered Napster from continuing to offer its services. Differently said, the central coordination represented by the Napster directory was a single point of failure with the special case that the failure is a political/legal failure and not a technical one. From a technical point of view, the centralized directory service offers a low messaging cost for locating items in the storage space but the load on the directory increases linearly with the number of participants which, anyhow, makes it unscalable.

2.2.2 Second Generation

Basic Idea

The central coordination in the first generation led to the transition to a new genre of P2P systems where the focus is on the elimination of the central coordination. The second generation started with applications like Gnutella [23] and Freenet [19]. A new participant in such systems must know an already-participating member and then uses a flooding algorithm to gain knowledge about other participants. Similarly, a participant performs a flooding algorithm by asking all of his neighbors about a given query. His neighbors act similarly and the process is stopped by a query embedded Time-To-Live value that prevents further forwarding of queries.

Discussion

Second generation systems solved the problem of central coordination. However, the problem of scalability became more severe because of high network traffic induced by the flooding algorithms as shown in studies such as [37, 55]. Moreover, there are no guarantees of finding a data item or a resource that exists in a Gnutella network because the search scope is limited. Freenet follows a slightly better approach which is the document routing model through which a data item d is inserted in a node with an identifier that is most similar to the identifier of d. During search, a query is forwarded guided by the identifier of the data item. Due to the random nature of the Freenet network, guarantees on finding items are low.

An optimization to the flooding/gossiping approach adopted in second generation systems was the introduction of the notion of super-peers that was initially adopted in the Kazaa [30] system and later in the Gnutella system as well. The optimization allows for some nodes to act as directory services and thus reduces the amount of flooding needed to locate data.

2.2.3 Third Generation

Basic Idea

The simultaneous "beauty" and "ugliness" of second generation overlay networks attracted academic researchers from the networking and the distributed systems communities. The "beauty" lies in the simplicity of the solution and its ability to completely diffuse central authority and legal liability. From a computer science point of view, this elimination of central control is very attractive for - among other things - eliminating single points of failure and building large-scale distributed systems. The "ugliness" lies in the huge amount of induced traffic that renders the solution unscalable [37, 55]. The problem of having a scalable P2P overlay network with no central control became a scientifically challenging problem and the efforts to solve it resulted in the emergence of what is known as "structured P2P overlay networks".

The third generation of P2P systems was initiated by research projects such as Chord [61, 62], CAN [53], Pastry [56], Tapestry [70] and P-Grid [1]. Those projects aim at providing what is known as a Distributed Hash Table (DHT) abstraction. A node (Peer) in such systems, acquires an identifier based on a cryptographic hash of some unique attribute such as its IP address or its public key. An identifier for a data item is also obtained through hashing. The hash table actually stores data items as values indexed by their corresponding keys. That is, node identifiers and key-value pairs are both hashed to one identifier space. The nodes are then connected to each other in a certain predefined topology, e.g. a circular space in Chord, a d-dimensional Cartesian space in CAN and a mesh in Tapestry and key-value pairs are stored at nodes according to the given structure. Thanks to the structured topology, data lookup becomes a routing process with low (typically logarithmic) routing table size and maximum path length. Unlike second generation systems, DHTs provide high data location guarantees.

Discussion

DHTs were introduced to let a set of cooperating peers act as a distributed data structure with well-defined operations, namely a distributed hash table with the two primitive operations `Put(key,value)` and `Get(Key)`. The `Put` operation should result in the storage of the value at one of the peers such that any of the peers can perform the `Get` operation and reach the peer that has the value. More importantly, both operations need to take a "small" number of hops. A first naive solution would be that every peer knows all other peers, and then every `Get` operation would be resolved in one hop. Apparently, that is not scalable. Therefore, a second constraint is needed. Each node should know a "small" number of other peers. From a graph-theory point of view, this means that a directed graph of a certain known "structure" rather than a random graph needs to be constructed with scalable sizes of both the outgoing degree of each node and the diameter of the graph.

Given the desirable properties of scalability and high guarantees while meeting the requirements of full decentralization, DHTs are currently considered in research communities as a reasonable approach to routing and location in P2P systems. While having a common principle, each system has some relative advantages. e.g., The Chord system has the property of simple design. Tapestry and Pastry address the issue of proximity routing. P-Grid excels in dealing with unbalanced distributions of identifiers. The most attractive property in all current DHT systems is self-organization. Due to the focus on the absence of central authority, DHTs provide mechanisms by which the structural properties of the network are maintained while the peers are continuously joining and leaving it.

2.3 Definitions and Assumptions

Values. The set of values $\mathcal{V}$ such as files, directory entries etc.. Each value has a corresponding key from the set $Keys(\mathcal{V})$. If a value is a file, the key could be, for instance, its checksum, a combination of owner, creation date and name or any such unique attribute.

Nodes. The set $\mathcal{P}$ of machines/processes also referred to as nodes or

peers. $Keys(\mathcal{P})$ is the set of unique keys for members of $\mathcal{P}$, usually the IP addresses or public keys of the nodes.

The Identifier Space. A common and fundamental assumption of all DHTs is that the keys of the values and the keys of the nodes are mapped into one range using a hashing function. For instance, the IP addresses of the nodes and the checksums of files are hashed using SHA-1 [18] to obtain 128-bit identifiers. The term "identifier" is used to refer to hashed keys of items and of nodes. The term "identifier space" refers to the range of possible values of identifiers and its size is usually referred to by N. We use id as an abbreviation for identifier most of the time.

Items. When a new value is inserted in the hash table, its key is saved with it. We use the term "item" to refer to a key-value pair.

Equivalence of Nodes. The operations of adding a value, looking up a value, adding a new node (join), removing an existing node (leave) are all possible through any node $p \in \mathcal{P}$.

Autonomy of Nodes. The addition or removal of any node is a decision taken locally at that node and there is a distinction between graceful removals of nodes (leaves) and ungraceful removals (failures).

The first contact. Another fundamental assumption in all DHTs is that to join an existing set of peers who already formed an overlay network, a new peer must know some peer in that network. This knowledge in many systems is assumed to be acquired by some out-of-band method. Some systems discuss the possibility of obtaining the first contact through IP multicast, however, it is an orthogonal issue to the operation of any DHT.

Ambiguous terms. Since we are forced to use different terminology to refer to the same logical entities in different contexts, we try to resolve those ambiguities early by introducing the following equalities. Nodes = peer = contact= reference, overlay network = overlay graph, identifier=id, edge = pointer, "point to"= "be aware of" = "keep track of", routing table = outgoing edges, diameter = lookup path length, lookup = query. routing table size = outgoing arity. Also some times, letters like n, s, t, x are used to refer to nodes and values as well as their identifiers but the meaning should be clear from the context.

2.4 Comparison Criteria

The Overlay Graph. This is the main criteria that distinguishes systems from each other. For each overlay graph, we want to know how the graph looks like and what is the outgoing arity of each node in the graph.

Mapping Items Onto Nodes. For a given overlay graph, we want to know the relation between node ids and item ids, i.e. at which node should an item be stored?

The Lookup Process. A tightly coupled property with the overlay graph is how lookups are performed and what is the typical performance.

Joins, Leaves and Maintenance. How a new node is added to the graph and how a node is gracefully deleted from the graph? Joins and leaves make the graph change constantly and some maintenance process is usually required to cope with such changes, so how does this process take place and what is its cost?

Replication and Fault Tolerance. In addition to graceful removal of nodes, failures are usually harder to deal with. Replication is a tightly coupled property since it can be a technique to overcome failures effect or a method of improving efficiency.

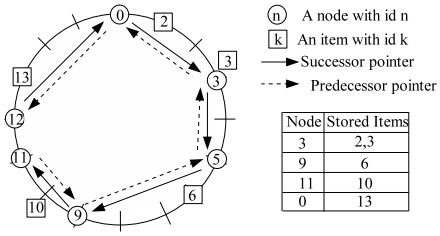
Upper Services and Applications. When applicable, we enumerate some of the applications and services developed using a certain system.

Implementation. Since many systems are of a completely theoretical nature even for their services and applications, we try to give and idea about any available implementations of a system.

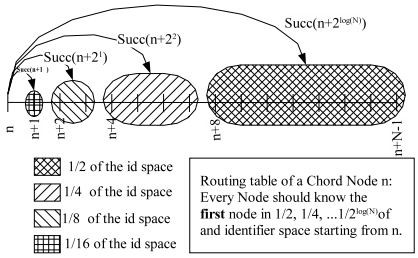
2.5 DHT Systems

2.5.1 Chord

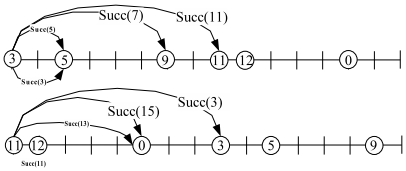
The Overlay Graph. Chord [61, 62] assumes a circular identifier space of size N. A Chord node with identifier u has a pointer to the first node following it clockwise on the identifier space ($Succ(u)$) as well as the first node preceding it ($Pred(u)$). The nodes therefore form a doubly linked list. In addition to those, a node keeps $M = \log_2(N)$ pointers called fingers. The set of fingers of node u is $F_u = \{(u, Succ(u + 2^{i-1}))\}, 1 \leq i \leq M$, where the arithmetic is modulo N.



(a)



(b)



(c)

Figure 2.1: (a) A chord network with $N = 16$ populated with 6 nodes and 5 items. (b) The general policy for Chord's routing tables. (c) Example routing tables for nodes 3 and 11.

The intuition of that choice of edges is that a node perceives the circular identifier space as if it starts from its id. The edges are, then, chosen such as to be able to partition the space into two halves, partition one of the halves into two quarters, and so forth.

In Figure 2.1(a), we show a network with an id space $N = 16$. Each node has $M = \log_2(N) = 4$ edges. The network contains nodes with ids $0, 3, 5, 9, 11, 12$. The general policy for constructing routing tables is shown in figure 2.1(b). Node n chooses its pointers by positioning itself at the start of the identifier space. It chooses to have the pointers to the successors of the ids $n + 2^0$, $n + 2^1$, $n + 2^2$, and $n + 2^3$. The last pointer $n + 2^3$, divides the space into two halves. The one before it $n + 2^2$ divides the first half into two quarters and so forth. However, there may not exist a node at the desired position so its successor is taken instead. Figure 2.1(c) shows the routing entries of node 3 and 11.

Mapping Items Onto Nodes. As shown in figure 2.1(a), an item is stored at the first node that follows clockwise on the circular identifier space. If items with ids 2, 3, 6, 10,13 are to be stored in the network given above, then {2,3} will be stored at 3; $\{6\}$ at 9; $\{10\}$ at 11; and $\{13\}$ at 0.

The Lookup Process. The lookup process comes as a natural result of how the id space is partitioned. Both the insertion and querying of items depend on finding the successor of an id. For example, assume that node 11 wants to insert a new item with id 8, the lookup is forwarded to node 3, which is the closest preceding finger - from the point of view of 11 - to the id 8. Node 3 will act similarly and forward the query to node 5 because 5 is the closest preceding finger for 8 from the point of view of 5. Node 5 finds that 8 is between itself and its successor 9. And therefore, returns 9 as an answer to the query through the reverse path[1]. In all cases, upon getting the answer, node 11's application layer should contact node 9's application layer and ask for the storage of some value under the key 8. Any node looking for the key 8 can act similarly and in no more than

[1]This is known as the recursive method. Another suggested approach in the Chord papers is an iterative method where all the answers path by the node at which the lookup originated, i.e. instead of the path being $11 \rightarrow 3 \rightarrow 5 \rightarrow 3 \rightarrow 11$, in an iterative lookup the path will be $11 \rightarrow 3 \rightarrow 11 \rightarrow 5 \rightarrow 11$. A third approach adopted in other systems like e.g. [4] would be to continue to the destination and send the result to the origin of the lookup, i.e. $11 \rightarrow 3 \rightarrow 5 \rightarrow 9 \rightarrow 11$.

M hops[2], a node will discover the node at which 8 is stored. In general, under normal conditions a lookup takes $O(\log_2(N))$ hops.

Joins, Leaves and Maintenance. To join the network, a node n performs a lookup for its own id through some first contact in the network and inserts itself in the ring between its successor s and the predecessor of s using a periodic stabilization algorithm. Initialization of n's routing table is done by copying the routing table of s or letting s lookup each required edge of n. The subset of nodes that need to adjust their tables to reflect the presence of n, will eventually do that because all nodes run a stabilization algorithm that periodically goes through the routing table and looks up the value of each edge. The last task is transfer part of the items stored at s, namely items with id less than or equal to n need to be transferred to n and that is also handled by the application layers of n and s.

Graceful removals (leaves) are done by first transferring all items to the successor and informing the predecessor and successor. The rest of the fingers are corrected by the virtue of the stabilization algorithm.

Replication and Fault Tolerance. Ungraceful failures have two negative effects. First, ungraceful failures of nodes cause loss of items. Second, part of the ring is disconnected leading to the inability of looking up certain identifiers. Let alone if a set of adjacent nodes fail simultaneously. Chord tackles this problem by letting each node keep a list of the $\log_2(N)$ nodes that follow it on the circle. The list serves two purposes. First, if a node detects that its successor is dead, it replaces it with the next entry in its successor list. Second, all the items stored at a certain node are also replicated on the nodes in the successor list. For an item to be lost or the ring to be disconnected, $\log_2(N) + 1$ successive nodes have to fail simultaneously.

Upper Services and Applications. A couple of applications such as a cooperative file-system [14], a read/write file system [42] and a DNS directory [13] were built on top of chord. As a general purpose service, a broadcast algorithm was also developed for Chord [16].

Implementation. The main implementation of Chord is that by its authors in C++ at [64] where a C++ discrete-event simulator is also available.

[2]Chord counts a remote procedure call and the response to it as one hop.

Naanou [27] is a C$^{\#}$ implementation of Chord with a file-sharing application on top of it.

2.5.2 Pastry

The Overlay Graph. The overlay graph design of Pastry [56] in addition to aiming to achieving logarithmic diameter with a logarithmic node state, also tries to target the issue of locality. In general, as a result of obtaining the node ids by hashing IP numbers/Public Keys, nodes with adjacent node ids may be farther apart geographically. Differently said, two machines in one country, would communicate through a machine in another continent just because the hash of their ids will be far apart in the id space.

Pastry assumes a circular identifier space and each node has a list containing $\frac{L}{2}$ successors and $\frac{L}{2}$ predecessors known as the leaf set. A node also keeps track of M nodes that are close according to another metric other than the id space like, for instance, network delay. This set is known as the neighborhood set and is not used during routing but used for maintaining locality properties. The third type of node state is the main routing table. It contains $\lceil \log_{2^b}(N) \rceil$ rows and $2^b - 1$ columns. L, M and b are system parameters.

Node ids are represented as string of digits of base 2^b. In the first row, the routing table of a node contains node ids that have a distinct first digit. Since the digits are of base 2^b, a node needs to know $2^b - 1$ nodes for each possible digit except its own.

The second row of a node with id n contains $2^b - 1$ nodes that share the first digit with n but differ in the second digit. The third row contains nodes that share the first and second digit of n but differ in the third and so forth. We stress that -1 in $2^b - 1$ is because in each row the node itself would be the best match for one of the columns, therefore we do not need to keep an address of it. Figure 2.2 illustrates how the the id space is partitioned using this prefix matching scheme.

As one can observe, for each of the constraints about the node ids contained in a routing table, there exists many satisfying nodes. Therefore the node with the lowest network delay or the best according to some other criteria is included in the routing table.

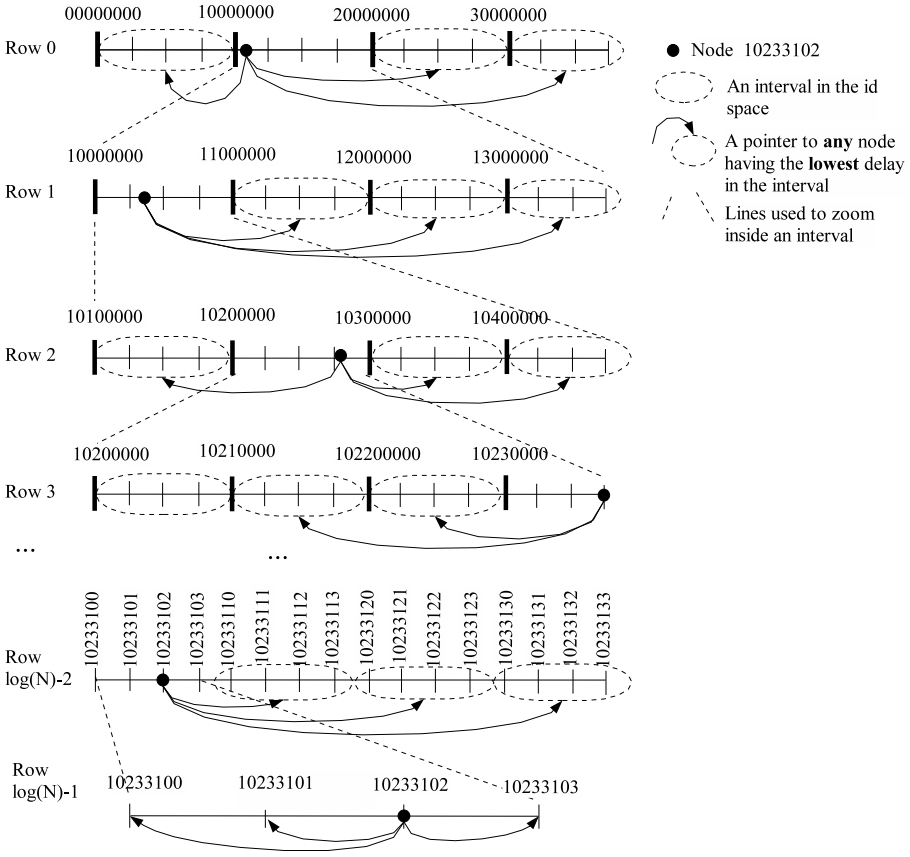


Figure 2.2: Illustration of how the Pastry node 10233102 chooses its routing edges in an identifier space of size $N = 2^{128}$ and encoding base $2^b = 4$.

Mapping Items Onto Nodes. An item in Pastry is stored at the node that is numerically closest to the id of the item. Such a node will have the longest matching prefix.

The Lookup Process. To locate the closest node to an id x, a node n checks first if x falls within the range of node ids covered by its leaf set. If so, it is forwarded to such node. Otherwise, the lookup is forwarded to the node in the interval that x belongs to, that is to a node that shares more digits than the shared prefix between n and x. If no such node is found in n's routing table, the lookup is forwarded to the numerically closest node to x. The later case does not happen so often provided that the ids are uniformly distributed. With the matching of one digit of the sought id in each hop, after $\log_{2^b}(N)$ hops a lookup is resolved.

Joins, Leaves and Maintenance. When a node n joins the network through a node t, then t is usually in the proximity of n and thus the neighborhood set of t is suitable for n. Due to the construction of the routing tables in Pastry, n performs a lookup for its own id to figure out the numerically closest node s to n. It can take the i^{th} row from the i^{th} node on the path from t to s and use those rows in initializing its routing table. Moreover, the leaf set of s is a good initialization for the leaf set of n. Finally, n informs every node in its neighborhood set, leaf set and routing table of its presence. The cost is about $3 \times 2^b \log_{2^b} N$.

Node departures are detected as failures and repaired in a routing table by asking a node in the same row of the failed node for its entry on the failed position.

Replication and Fault Tolerance. Pastry replicates an item on the k closest nodes in its leaf set. This serves in saving an item after a node loss and in the mean time, the replicas act as cached copies that can contribute in finding an item more quickly.

Upper Services and Applications. A number of applications and services were developed on top of Pastry such as, SCRIBE [11] for multicasting and broadcasting. PAST [57], an archival storage system. SQUIRREL [28], a co-operative web caching system. SplitStream [10], a high-bandwidth content distribution .

Implementation. FreePastry [20] is an open-source Java implementation of the Pastry system.

2.5.3 Tapestry

Tapestry [69] is one of the earliest and largest efforts on structured P2P overlay networks. Like Pastry, it is based on the earlier work of a Plaxton [52] mesh. We will not describe the details of Tapestry due to the large similarity with Pastry. However, we have to point out that as a software, it is probably one of the most mature implementations of a structured overlay network. In addition to network simulation, Tapestry has been evaluated using a more realistic environment, namely PlanetLab [51], a globally distributed platform with machines all over the world that is used for testing large-scale systems.

Tapestry is a corner-stone project in the larger Oceanstore [31] project for global-scale persistent storage. Other applications based on Tapestry include the steganographic file system Mnemosyne [25], Bayeux [72] an efficient self-organizing application-level multicast system, and SpamWatch [71] a decentralized spam-filtering system.

2.5.4 Kademlia

The Overlay Graph. The Kademlia [40] graph partitions the identifier space exactly like Pastry. However, it is presented in a different way where node ids are leafs of a binary tree with each node's position is determined by the shortest unique prefix of its id. Each node divides the binary tree into a series of successively lower subtrees that don't contain the node id and keeps at least one contact in each of those subtrees. For instance, a node with id 3 has the binary representation 0011 in an identifier space of size $N = 16$. Since its prefix of length 1 is the digit 0 then it needs to know a node whose first digit is 1. Since its prefix of length 2 is 00, then it needs to know a node with prefix 01. Since its prefix of length 3 is 001, then it needs to know a node with prefix 000. Finally, since its prefix of length four is 0011, then it needs to know a node with a prefix 0010. This policy is illustrated in figure 2.3 which results in a space division exactly like Pastry with the special case of a binary encoding of the digits.

Kademlia does not keep a list of nodes close in the identifier space like the leaf set or the successor list in Chord. However, for every subtree/interval in the identifier space it keeps k contacts rather than one con-

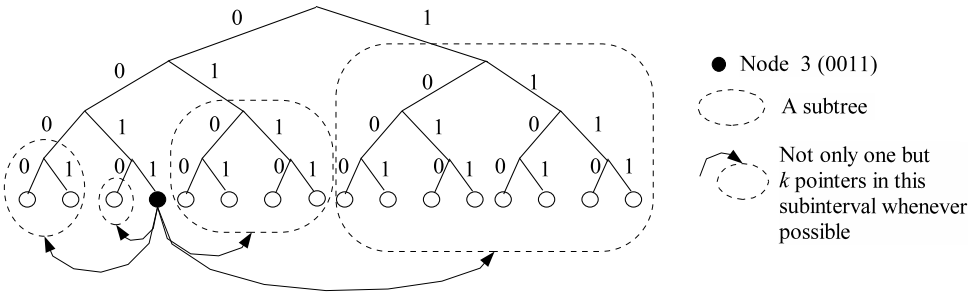


Figure 2.3: The pointers of node 3 (0011) in Kademlia. The same partitioning of the identifier space as in Pastry with binary-encoded digits.

tact if possible, and calls a group of no more than k contacts in a subtree a k-bucket.

Mapping Items Onto Nodes. Kademlia defines the notion of distance between two identifiers to be the value of the bitwise exclusive or (XOR) of the two identifiers. An item is stored at the node whose XOR difference between the node id and the item id is minimal.

The Lookup Process. To increase robustness and decrease response time, Kademlia performs lookups in a concurrent and iterative manner. When a node looks up an id, it checks to which subtree does the id belong and forwards the query to α randomly selected nodes from the k-bucket of that subtree. Each node possibly returns back a k-bucket of a smaller subtree closer to the id. From the returned bucket, another α randomly selected nodes are contacted and the process is repeated until the id is found. When an item is inserted, it is also stored at the k closet nodes to its id. Because of the prefix matching scheme, similar to Pastry, a lookup is also resolved in $O(\log(N))$ hops.

Joins, Leaves and Maintenance. A new node finds the closest node to it through any initial contact and uses it to fill its routing table by querying about nodes in different subtrees. If it happens that a k-bucket is filled due to exposure to lots of nodes in a particular subtree, a least-recently-used replacement policy is applied. However, Kademlia makes use of statis-

tics taken from existing peer-to-peer measurements studies which indicate that a node which stayed for a longer time in the past will probably stay connected longer in the future. Therefore, Kademlia can discard the knowledge of new nodes if it knew many other stable nodes in a given subtree.

Maintenance of the routing tables after joins and leaves depends on a technique that is different from the stabilization in Chord or the deterministic update of Pastry. Kademlia maintains the routing tables by using the lookup traffic. The XOR metric results in every node receiving queries from the nodes contained in its routing table (Which is not the case in a system like Chord). Consequently, the reception of any message from a certain node in a certain subtree is essentially an update of the k-bucket for that subtree. This approach clearly minimizes the maintenance cost. However, it is not deeply analyzed.

Another maintenance task is that upon receiving multiple queries from the same subtree, Kademlia updates the latencies of the nodes in a particular k-bucket. This improves the choice of the nodes used for doing lookups and one could say that by doing that, Kademlia also takes into consideration network delay and locality.

Replication and Fault Tolerance. Since leaves are not deeply discussed, we assume that they are treated as failures. Kademlia fault tolerance depends mainly on the strong connectivity since it keeps k contacts per subtrees and not only one and this makes the probability of a disconnected graph low.

Also as mentioned above, Kademlia stores k copies of an item on the k closest nodes to its id. The nodes are also republished periodically. The policy for republishing is that any node that sees itself closer to an item id than all the nodes it knows about, gives it to $k - 1$ other nodes.

Applications and Implementation. Kademlia is probably the one DHT that got a relatively wider non-academic adoption by being used in two file-sharing applications, namely Overnet [48] and Emule [17].

2.5.5 HyperCup

While it has been mentioned many times in the literature that systems like Chord and Pastry, for instance, are approximations of Hypercubes, those works were not presented that way by their authors. HyperCup [58] is a system that presents a way to construct and maintain Hypercubes in a dynamic setting. The performance of HyperCup is similar to the many other DHTs with logarithmic order for both the routing table size and the lookup path length under particular uniformity assumptions. HyperCup also defines a broadcast algorithm based on the concept of a spanning tree of all nodes. A distinguished feature of HyperCup is that it addresses semantic search based on ontological terms. Nodes with similar ontologies are clustered together such that a search by a certain ontological term is achieved as a localized broadcast within a cluster.

2.5.6 DKS

The Overlay Graph. DKS [4] could be perceived as an optimal generalization of Chord to provide shorter diameter with larger routing tables. In the mean time, DKS could be perceived as a meta-system from which other systems could be instantiated. DKS stands for Distributed k-ary Search and it was designed after perceiving that many DHT systems are instances of a form of k-ary search. Figure 2.4 shows the division of the space done in DKS. You can see that it has in common with Chord that each node perceives itself as the start of the space. In the mean time, like Pastry each interval is divided into k rather than 2 intervals.

Mapping Items Onto Nodes. Along with the goal of DKS to act as a meta-system, mapping items onto nodes is also left as a design choice. A Chord like mapping is a valid as a simple first choice. However, different mappings are possible as well.

The Lookup Process. A query arriving at a node is forwarded to the first node in the interval to which the id of the node belongs. Therefore, a lookup is resolved in $\log_k(N)$ hops.

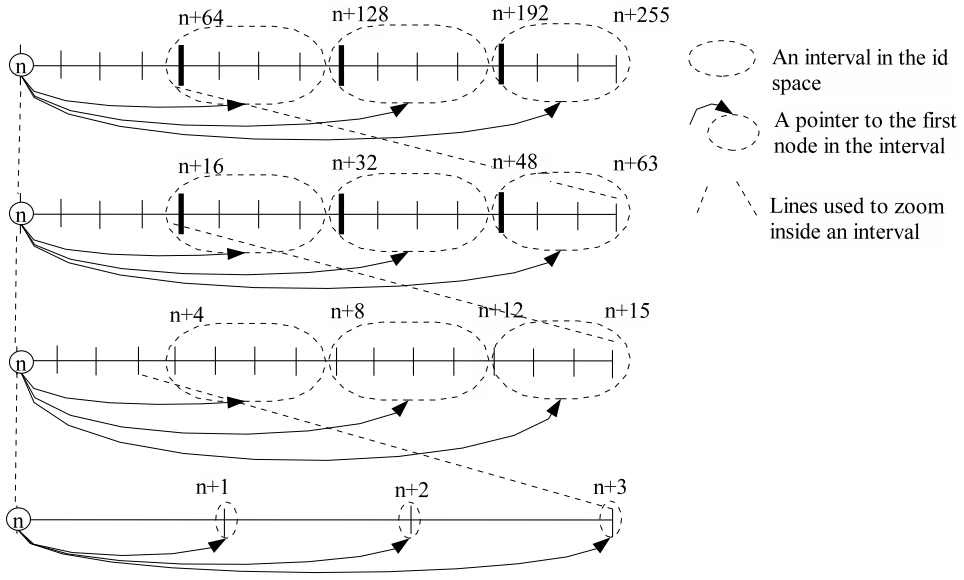


Figure 2.4: Illustration of how a DKS node divides the space in an identifier space of size $N = 2^8 = 256$.

Joins, Leaves and Maintenance. Unlike Chord, DKS avoids any kind of periodic stabilization both for the maintenance of the successors, the predecessor and the routing table. Instead, it relies on three principles, local atomic actions, correction-on-use and correction-on-change. When a node joins, a form of an atomic distributed transaction is performed to insert it on the ring. Routing tables are then maintained using the correction-on-use technique, an approach introduced in DKS. Every lookup message contains information about the position of the receiver in the routing table of the sender. Upon receiving that information, the receiver can judge whether the sender has an updated routing table. If correct, the receiver continues the lookup, otherwise the receiver notifies the sender of the corruption of his routing table and advises him about a better candidate for the lookup according to the receiver's knowledge. The sender then con-

tacts the candidate and the process is repeated until the correct node for the routing table of the sender is used for the lookup.

By applying the correction-on-use technique, a routing table entry is not corrected until there is a need to use it in some lookup. This approach reduces the maintenance cost significantly. However, the number of joins and leaves are assumed to be reasonably less than the number of lookup messages. In cases where this assumption does not hold, DKS combines it with the correction-on-change technique [6]. Correction-on-change notifies all nodes that need to be updated upon the occurrence of a join, leave or failure.

Replication and Fault Tolerance. In early versions of DKS, fault tolerance was handled similar to Chord where replicas of an item are placed on the successor pointers. In later developments [22], DKS tries to address replication more on the DHT level rather than delegating most of the work to the application layer. Additionally, to avoid congestion in a particular segment of the ring, replicas are placed in dispersed well-chosen positions and not on the successor list. In general, for the correction-on-use technique to work, an invariant is maintained where the predecessor pointer has always to be correct and that is provided by the atomic actions on the circle.

Upper Services and Applications. General purpose broadcast [21] and multicast [5] algorithms were developed for DKS.

2.5.7 P-Grid

The P-Grid system has a relatively large set of publications introducing various intricate features. We will try here to account for a subset of its basic notions.

The Overlay Graph. The basic structure of the P-Grid [1] graph mostly resembles Kademlia/Pastry, with some differences. The nodes are regarded as leaves in a binary tree. A node n keeps references to nodes in other subtrees of incremental heights not including n. P-Grid, however, is distinguished by a very unique way of assigning node ids as we will explain shortly.

In addition to references to other nodes in the tree, each node maintains

a set of random peers known as the "fidget list". Each node is assumed to have random frequent interactions with members of its fidget list.

Mapping Items Onto Nodes. Unlike most other DHT systems where a unique attribute (e.g. IP address or public key) governs the position of a node in the id space, in P-Grid, this position ("path" in P-Grid terminology) is determined from the id distribution of the items. This decoupling of a node's identity from its position in the id space is used to provide many unique features.

Joins and Leaves. Initially, a node joins P-Grid with items in its local storage and an empty path. Random interactions with other nodes from the fidget list help the new node to have a path in the search tree. When two peers interact, a number of issues need to be resolved such as: will their paths remain the same? will they give data to each other? could they know better fidgets through this interaction? how will the references be affected if a path changes? The answers to those issues depend on the state of the interacting nodes and are managed in one elegant algorithm called the "Exchange" algorithm. The complexity of the algorithm prevents us from giving a detailed description of it here. Instead, we use an example given in [2] and illustrate it in figure 2.5. In this example, when we say: "two peers P_i and P_j interact", this means that this is a random choice based on the fidget list of one of them. The example shows some cases such as: nodes changing their paths from empty to one bit or from one bit to two bits (specializing on a path) and nodes giving each other data based on the path they are specialized on. Notice that because of the random interactions, two networks can merge very easily which is also a distinguished feature of P-Grid.

In the final state of the example, all nodes have the same storage load. Nevertheless, some nodes have the same path and the same data, which means that for certain paths there are many replicas. That is, for this example, while storage load balancing is achieved, replication load balancing is not achieved. Therefore, P-Grid introduces an extra mechanism for replication load balancing.

Since replication is offered, and nodes have more than one reference in each subtree, a node can leave without notifying any other node. If the leaving node wants to rejoin the network, it searches for its path before

leaving the network and retrieves any missing data for the path it was specialized on.

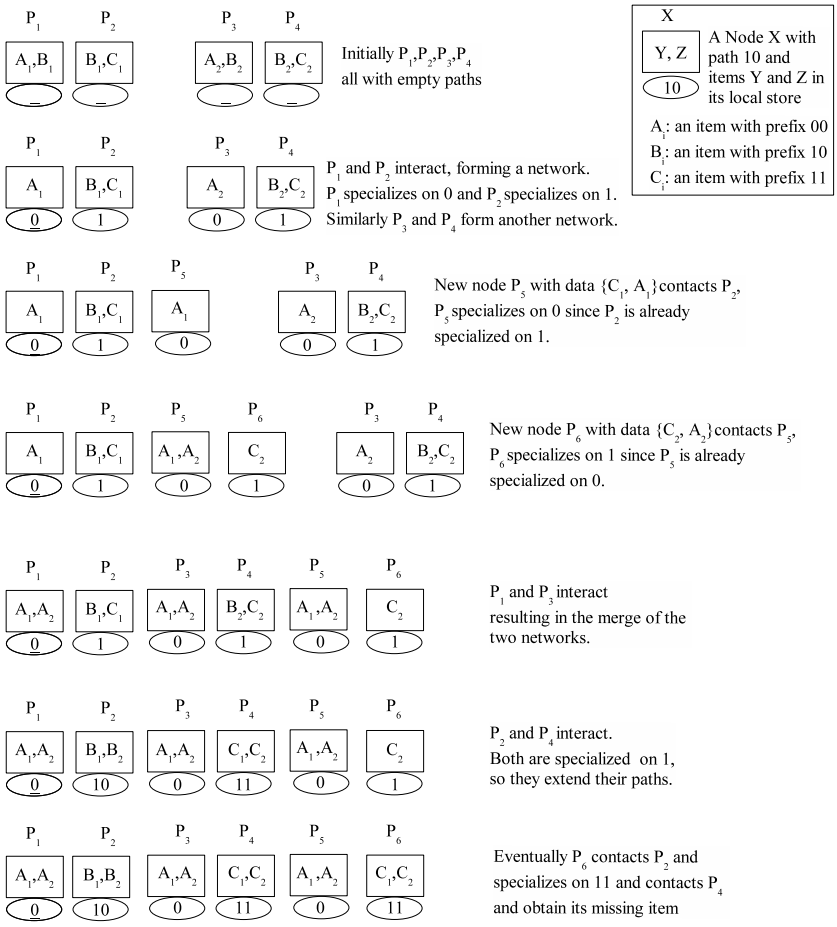


Figure 2.5: Illustration of some interactions of P-Grid nodes.

The Lookup Process and Maintenance. In its most basic form, the lookup process follows a prefix routing scheme through which a node forwards the query to a node with at least one more matching bit of the sought id. The lookup ends when any one of the sought replicas is located.

In [3], the issue of dynamic IP addresses is elaborated on. Many DHTs assume that when a node rejoins with a new IP, it is assigned a new identity. In P-Grid, however, there is a deeper treatment of this issue since there is a complete separation between the identity, the path and the IP address. Thus, a node can have a correct reference to a given path, however, the node specialized on that path could change its IP address. Therefore, lookups which can correct the stale routing tables are introduced in two variants. The first variant is an eager variant in which a discovery of a stale reference upon a lookup triggers the immediate correction of all references. The second variant is a lazy one where a node tries to route through alternate references. Correction is triggered only if no alternate reference was found.

Replication. Most DHTs assume some constant number of replicas for each data item. In P-Grid, this is not the case. The network tries to dynamically balance the replication load as well as the storage load. Therefore, a node continuously collects statistics about the number of other nodes with same/share-prefix of their paths. From those statistics a local approximation to the global replication loads of items is obtained and is used when nodes interact to judge whether more/less replicas are needed.

Implementation. A file-sharing application with the same name is implemented in Java and available at [49].

2.5.8 Koorde

The Overlay Graph. Koorde [29] is based on the DeBruijn graph [39]. Koorde stresses the point that a constant number of outgoing edges per node is enough for having a logarithmic lookup length. The DeBruijn graph is an example capable of doing that. The significance of a constant number of edges is that the maintenance overhead is lower compared to a logarithmic number as is the case in all the previous DHTs we have shown so far. In figure 2.6(a) we show the pointers of all the nodes of a Koorde graph of

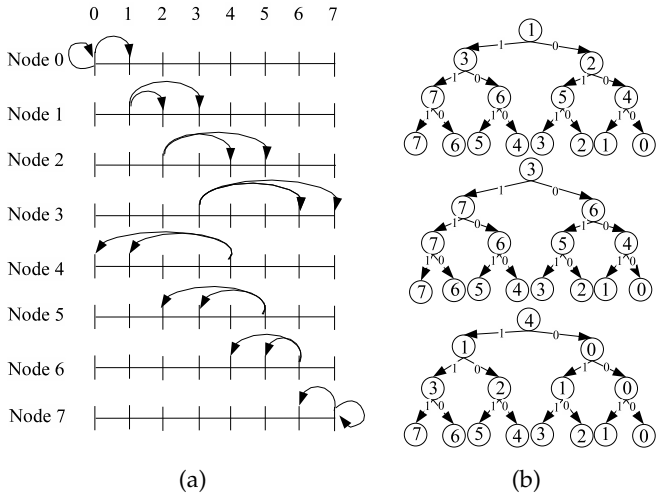


Figure 2.6: (a) The pointers of all the nodes in a complete Koorde network where $N = 8$. Every node n points to nodes of ids $2n$ and $2n + 1$. (b) Examples of how nodes 1, 3 and 4 reach other nodes by matching the destination id digit by digit starting from the most significant bit.

eight nodes. A node with id n has edges to nodes $2n$ and $2n + 1$ in a circular identifier space like Chord. We denote the first and the second edge of node n $E_n \circ 0$ and $E_n \circ 1$ respectively.

Mapping Items Onto Nodes. Exactly like Chord.

The Lookup Process. When a node n needs to lookup an id x represented as a string of binary digits $d_1 d_2 ... d_{\log_2(N)}$, it takes the top bit d_1, if it is a 0, it forwards the query to $E_n \circ 0$ otherwise to $E_n \circ 1$. The second node looks at the remaining string $d_2 ... d_{\log_2(N)}$ and acts similarly. After, at most, $\log_2(N)$ hops a query is resolved. Figure 2.6(b) shows what paths nodes $1, 3$ and 4 take to reach any node in the network. The Koorde paper also elaborates on an algorithm to handle networks where not all the nodes are present in the id space. Each node tries to locally traverse imaginary hops for nodes that do not exist.

Joins, Leaves and Maintenance. Exactly like Chord. In fact, the authors say that Koorde could be perceived as a Chord system with a con-

stant instead of a logarithmic number of fingers. Stabilization is also the basic mechanism for maintenance.

Replication and Fault Tolerance. For fault tolerance to be realized, an out-degree less than $\log(N)$ nodes has to be maintained, otherwise a node will loose all its contacts very easily. This makes the advantage of a constant node state invalid. However, since with k edges, Koorde provides $\log_k(N)$ diameter. Then with $\log_k(N)$ edges, it provides $\log_{\log_k(N)}(N) = \frac{\log(N)}{\log(\log(N))}$ diameter, which is an advantage over the logarithmic class of DHTs.

Load Balancing. The load balancing of items onto nodes will depend on the uniform distribution exactly like Chord. However, another load-balancing issue arises which is the load of message passing on each node. In a DeBruijn graph, some nodes will have more traffic than others by a factor of $O(\log(N))$ of the average traffic load. For example, in the network illustrated in figure 2.6, if every node would send a message to every other node in the network, not all the nodes will endure the same number of messages; 12 messages will be routed via a node like 7 while 21 messages will be routed via a node like 3.

2.5.9 Distance Halving

The Overlay Graph. The Distance Halving (DH[3]) [43] distributed hash table is another system based on the DeBruijn graph like Koorde. However, the way of building the graph is somewhat different. The DH is based on an approach called the continuous-discrete approach for building graphs. To build a DeBruijn graph with this approach, the identifier space is normalized into a continuous space represented by the interval $[0, 1[$. Nodes are points in that interval. Each node y has two edges, a left edge and a right edge denoted $\ell(y)$ and $r(y)$ respectively where $\ell(y) = \frac{y}{2}$ and $r(y) = \frac{y}{2} + \frac{1}{2}$. Given the set of points and their edges, a discretization step is done to build the graph. The set of points are denoted by $\vec{x}$. The points of $\vec{x}$ divide the space into n segments. The segment of a point x_i,

[3]Please do not confuse this abbreviation with the abbreviation of a Distributed Hash Table (DHT).

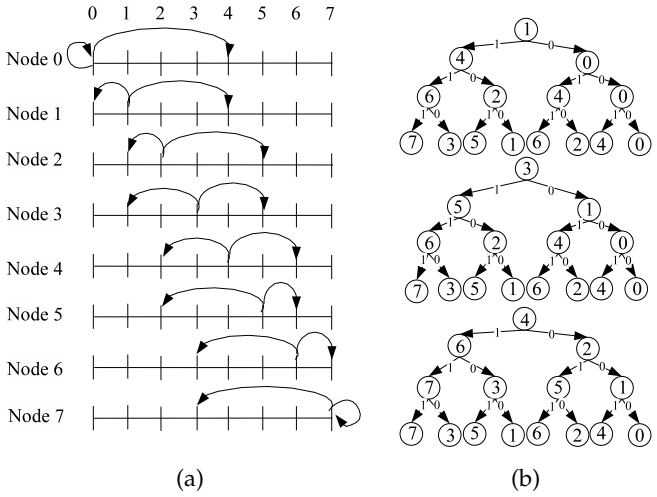


Figure 2.7: (a) The pointers of all the nodes in a complete Distance-Halving network where $N = 8$. (b) Examples of how nodes 1 reaches other nodes by matching the destination id digit by digit starting from the least significant bit.

$S(x_i) = [x_i, x_{i+1})$, $(i = 1...n-1)$ and $S(x_n) = [x_{n-1}, 1) \cup [0, x_1)$. If a node y has an edge that belongs to the segment of some node z, then there is an edge in the discrete graph between y and z. One can also notice that the segments are defined in a way that realizes a circular identifier space.

The intuition behind that graph is that every node divides the space into two intervals and keeps a pointer to a node that is in the middle of the left interval and a pointer in the middle of the right interval. Figure 2.7(a) shows the pointers of all the nodes in a DH network of size $N = 8$. Figure 2.7(b) shows the paths to all possible destinations starting from node 1.

Mapping Items Onto Nodes. Exactly like Chord, Koorde. In DH terminology, an item is stored at node y where the id of the item belongs to $S(y)$.

The Lookup Process. The lookup process, similar to many other DHTs, is done by the prefix matching of the sought id digit by digit. The lookup

is forwarded to the node pointed to by the left edge for matching a 0 digit and to the right edge for matching a 1 digit. The lookup path length is thus $O(\log_2(N))$.

Joins, Leaves and Maintenance. A new node n joins a DH network by looking up the node s such that n belongs to $S(s)$. n then uses s to lookup its left and right edges. By the construction of DH, a node can easily know the nodes that are pointing to it. Therefore a node can easily compute the nodes that needs to be updated and notifies them of n's existence. Updating of others upon a leave is done in the same way. The transfer of the items upon a join or a leave is also similar to Chord.

Replication and Fault Tolerance. DH recognizes the problem of failures that can lead to a disconnected graph and advocates an additional state of $O(\log(N))$ pointers. That comes in agreement with Koorde's reasoning and emphasizes that the main advantage of having a constant degree graph will be compromised if fault tolerance is to be considered. However, with a logarithmic degree, those types of graphs can offer a diameter of $\frac{\log(N)}{\log(\log(N))}$.

Other Comments. The formal analysis of the DH graph and the continuous discrete approach are both useful tools that help gaining more understanding of the properties of a DHT system. The discussion of the smoothness of the graph which is a term used by the authors to quantify the uniformity of distribution of the ids is quite unique. It was noted in other DHT systems that uniform distribution could affect the performance but in DH, an analysis of the magnitude of that effect is provided.

2.5.10 Viceroy

The Overlay Graph. Viceroy [36] is based on the Butterfly [38] network. Like many other systems, it organizes nodes into a circular identifier space and each node has successor and predecessor pointers. Moreover, nodes are arranged in $\log_2(N)$ levels numbered from 1 to $\log_2(N)$. Each node apart from nodes at level 1 have an "up" pointer and every node apart from the nodes at the last level have 2 "down" pointers. There is one short and one long "down" pointers. Those three pointers are called the Butterfly pointers. All nodes also have pointers to successors and predecessors

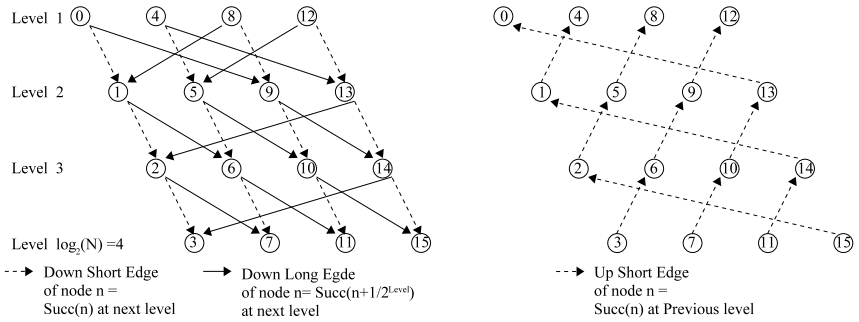


Figure 2.8: The Butterfly edges of a complete Viceroy network with $N = 16$ nodes. (a) The down edges. (b) The up edges.

pointers on the same level. In that way, each node has a total of 7 outgoing pointers.

Figure 2.8(a) shows the down pointers of a network of $N = 16$ nodes where all nodes are present. Figure 2.8(b) shows the up pointers of all nodes. For simplicity, the successor pointers of the ring and the levels are not illustrated[4].

Mapping Items Onto Nodes. Exactly like Chord.

The Lookup Process. To lookup an item x, a node n follows its up pointer until it reaches level 1. From there, it starts going down using the down links. In each hop, it should traverse a pointer that does not exceed the target x. For example, if node 1 is looking up the id 10, first it will follow its up pointer and reach 4 which is at level 1. At node 4 there are two choices either to use the short pointer to 5 or the long pointer to 13, since 5 precedes the target 10, the pointer to 5 is followed. At node 5, there is a direct pointer to 10. In another example, for reaching id 15 from node 3, the path will be $3 \stackrel{up}{\rightarrow} 6 \stackrel{up}{\rightarrow} 9 \stackrel{up}{\rightarrow} 12 \stackrel{down}{\rightarrow} 13 \stackrel{down}{\rightarrow} 14 \stackrel{down}{\rightarrow} 15$. From the last example, we can see that in a worst case, we can traverse all the levels up and down, i.e. $2 \times log(N)$ hops. Needless, to say that the example includes

[4]In fact, some of them coincide with the Butterfly pointers.

a simplified network where all the nodes are present. When the graph is sparse, the reasoning is slightly more complicated, however the expected lookup path length is still $O(\log(N))$.

Joins, Leaves and Maintenance. To join, a node looks up its successor s, fixes the ring pointers and takes the required items from s. After that, it selects a level based on the estimation of the number of nodes. It finds, by a combination of lookups and stepping on the ring, the rest of the pointers (successor and predecessor at the selected level, up and down pointers).

To leave, a node disconnects all its pointers, the concerned nodes consequently are aware and lookup for replacements. Additionally, the stored items are transferred to the successor.

Replication and Fault Tolerance. Viceroy does not deeply discuss ungraceful failures nor replication but refers to Lynch et al.'s paper [34] for a general approach in handling failures in DHTs.

Implementation. There exists a Java implementation of Viceroy at [66]. This homepage includes also a visualization applet that can illustrate the main topology, lookups, joins and leaves in Viceroy.

Other Comments. While the intuitive analysis might lead to thinking that nodes at higher levels endure more lookup traffic, Viceroy's analysis shows that the congestion is not that bad, however such a proof is beyond the scope of that chapter.

2.5.11 Ulysses

Ulysses is another system based on the Butterfly graph. It achieves the known limits of routing table and lookup path length, $O(\log(N))$ and $\frac{\log(N)}{\log(\log(N))}$ while accounting for joins, leaves and failures. In that sense, it agrees with the conclusions of Koorde, Distance-Halving and Viceroy and shows a second way of building a Butterfly network. Ulysses also depends on periodic stabilization for maintenance of the graph. Like Distance-Halving, it discusses the elimination of congestion. Ulysses also has an interesting discussion on the optimization of the average lookup path length.

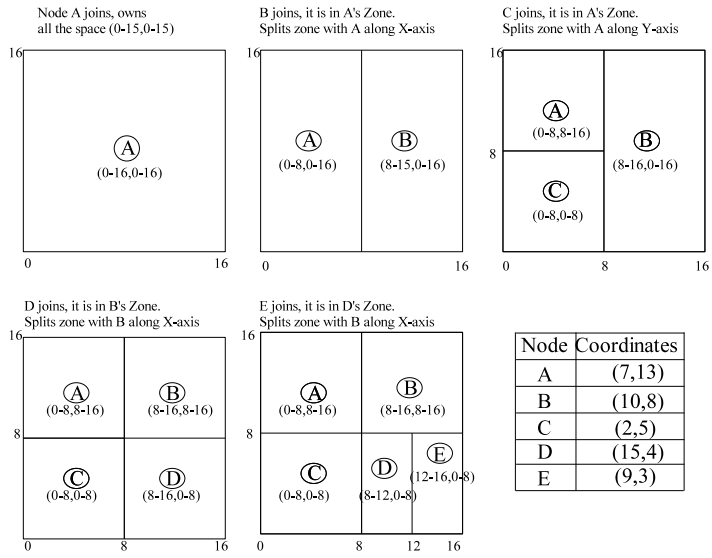


Figure 2.9: The process of 5 nodes joining a CAN network.

2.5.12 CAN

The Overlay Graph. CAN [53] is in a class of its own. The design of the graph is based on a d-dimensional coordinate space. Like all other systems, the nodes and items are mapped onto a virtual space using a uniform hashing function, but the hashing is applied d times to get the d coordinates. For instance, in a 2-dimensional discrete coordinate space, an IP address or key of a file would be hashed once to obtain an x value and another time to obtain a y value. The coordinate space is dynamically partitioned among all the nodes in the system such that every node"owns" its distinct zone within the overall space. Figure 2.9 shows a discrete coordinate space of 16×16 partitioned among 5 nodes.

Mapping Items Onto Nodes. An item with key k is stored at the node that owns the zone onto which k is mapped. Two nodes are neighbors, i.e. have pointers to each other if their zones have common sides.

The Lookup Process. A lookup is achieved by using the straight line path through the Cartesian space from source to destination.

Joins, Leaves and Maintenance. A new node w joins by selecting a random point P, it sends to its initial contact in the network u a JOIN message containing P. Node u consequently routes the message to the node v that owns the zone in which P lies. The zone of v is then split between v and w. Zones are split along the x axis first then along the y axis. Upon a split, the new node learns its neighbors from the previous owner. Neighbors of a new node are neighbors of the previous owner plus the previous owner itself. The new node informs its neighbors of the change. The cost of join in that way is $O(d)$. Finally, items that belong to the new node are obtained from the previous owner.

The leave process is the reverse, a node informs its neighbors of its leaving and merges its zone with a neighbor to produce a valid zone. If no valid zone could be formed, the items are transferred to a neighbor owning the smallest zone.

Under normal conditions, a node sends periodic updates to each of its neighbors given them its zone coordinates. Additionally, there is a background zone-balancing process that tries to reconfigure zones after a series of joins and leaves.

Replication and Fault Tolerance. There are two ways of detecting failures in CAN, the first if a node tries to communicate with a neighbor and fails, it takes over that neighbor's zone. The second way of detecting a failure is by not receiving the periodic update message after a long time. In the second case, the failure would probably be detected by all the neighbors, and all of them would try to take over the zone of the failed node, to resolve this, all nodes send to all other neighbors the size of their zone, and the node with the smallest zone takes over.

Replication in CAN is achieved in two ways. The first way is to use α hashing functions to map an item to α points. When retrieving an item, α queries are sent and α responses are received. The second way is to create multiple instances of the coordinate space. Each instance is called a "reality". If a node storing an item is dead in one reality, the item can be retrieved from one of the other realities because the item would be stored at other nodes in the other realities.

Latency. Every node in CAN keeps round-trip-time (RTT) of its neighbors. When selecting a path for a lookup, a CAN node forwards to the

neighbor with maximum ratio of progress to RTT. CAN also has a mechanism for nodes to choose their points so as to make points near in the IP network also near in the Cartesian space, the technique uses root DNS servers as landmarks from which a node can approximate to which other nodes it is near in the IP network.

Upper Services. A multicast protocol is available for CAN [54]. Some work has also been done on richer queries such as range queries in [7].

2.6 Summary

2.6.1 The Overlay Graph

We summarize the different overlay graphs by providing a classification based on the size of the node state as shown in figure 2.10. The first category is for systems that keep a logarithmic number of routing entries. Most DHT systems are in that category. A common property in that category is the logarithmic order lookup path length. The second category includes systems that use a constant number of routing entries. CAN is in a class of its own as it provides a polynomial order lookup path length. Other systems in the same category include the DeBruijn-based and the Butterfly-based DHTs and such systems offer a logarithmic path length. Naturally, one can instead set the constant of the constant-state systems to a value logarithmic in the number of nodes and get a shorter lookup length. Table 2.1 summarizes those performance trade-offs.

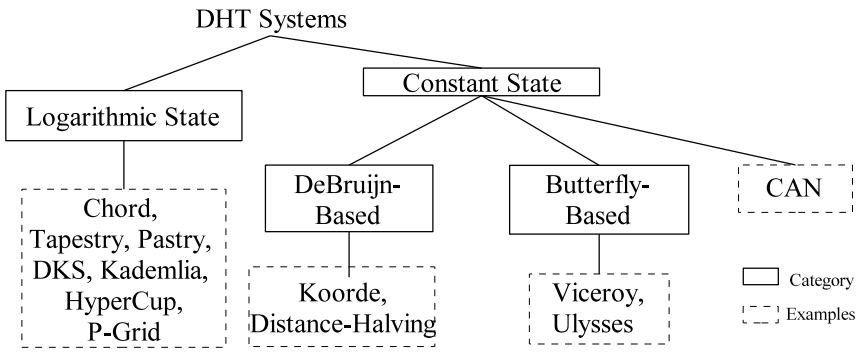


Figure 2.10: A classification of DHT systems based on the size of the node state and underlying graph.

Category	Node State	Lookup Path Length
Logarithmic sate	$O(\log(N))$	$O(\log(N))$
DeBruijn & Butterfly (per se)	$O(k)$	$O(\log(N))$
DeBruijn & Butterfly ($k = O(\log(N))$)	$O(\log(N))$	$\frac{O(\log(N))}{O(\log(\log(N)))}$
CAN (per se)	$O(k)$	$O(kN^{1/k}))$
CAN ($k = O(\log(N))$)	$O(\log(N))$	$O(\log(N)N^{1/\log(N)}))$ $= O(\log(N))^a$

[a] Since $N^{1/\log(N)}$ is a constant factor.

Table 2.1: Summary of Node State and Lookup Path Length for the different categories of systems.

2.6.2 Mapping Items Onto Nodes

Four ways of assigning items to ids are identified and summarized in table 2.2.

Assignment policy	Example Systems
Item assigned to successor on the ring	Chord, DKS, Koorde, Viceroy, DH, Ulysses
Item assigned to numerically closet node	Pastry, Tapestry
Item assigned to XOR closest node	Kademlia
Item assigned to zone owner	CAN

Table 2.2: The different policies for mapping items onto nodes.

In all those scenarios, the fair (load-balanced) assignment of items onto nodes relies on the uniform distribution of the hashing function. This is apart from P-Grid, where the network is in constant trial to load balance the items between nodes irrespective of the distribution of identifiers.

2.6.3 The Lookup Process

The lookup process is a direct result of the node state. Increasing more node decreases the lookup path length but increases the maintenance cost.

In some systems like Pastry, Tapestry, Kademlia and CAN, overlay hops are not the sole optimized metric, additionally network latency is addressed.

Congestion is a tricky issue related to the lookup process. Not only the lookup path length should be optimized, but it should not be the case that some nodes endure more traffic than others which is the case in the DeBruijn and Butterfly graphs. However, authors of systems that suffer from congestion try to adapt those graphs to eliminate congestion.

2.6.4 Joins, Leaves and Maintenance

Joins and leaves jeopardize the desired properties of any good graph, different systems have adopted different techniques to bring back the overlay

graph to its ideal state. Table 2.3 enumerates those techniques.

Maintenance policy	Example Systems
Stabilization	Chord, Koorde, Viceroy, CAN
Use of Traffic	Kademlia
Determinism+ Stabilization	Pastry, Tapestry
Correction on-use + Correction on-change	DKS
Lazy+Randomized	P-Grid

Table 2.3: The different policies overlay graph maintenance policies.

Stabilization is the most common technique where routing table entries are periodically looked up and corrected. The use of traffic is adopted in Kademlia where the graph structure makes a node receive lookups from the same nodes it is pointing to. Pastry also depends on the structure of the graph where a new node can inform all the other nodes that need to be informed about it. Periodic activity is still needed though for collecting latency information. The correction-on-use introduced in DKS, relies on the presence of traffic as well but a receiving node can correct a sending node and no periodic activity is used. Where not sufficient alone, correction-on-use is complemented with a more deterministic technique, namely, correction-on-change. P-Grid has a unique correction mechanism, where the random interaction between peers can lead to the change of their ids in a way that causes eventual optimality of the graph.

2.6.5 Replication and Fault Tolerance

Replication is an essential tool for recovering items stored at failed nodes. Choice of nodes for replication is tightly coupled with the policy for mapping items to nodes. Local vicinity is mostly chosen, for example, the successors on the circle or the k numerically closest nodes.

Fault tolerance is one of the most challenging and open areas in structured overlay networks. Some systems can cope with the failure of a small number of nodes at a time. However, dealing with a large number of simultaneous failures is harder. A constant- state routing table is an advantage that has to be given up if a large number of simultaneous failures

is to be tolerated. Nodes will have to keep their node state to at least logarithmic order to be able to cope with $N/2$ randomly-distributed nodes failing simultaneously or the failure of $O(\log(n))$ adjacent node ids simultaneously.

2.7 Hot and Open Research Issues

DHTs as the state-of-the-art systems, made a remarkable progress in solving the issue of scalability and decentralization while providing determinism and high guarantees. However, they opened the way for new research issues. We briefly enumerate some of those issues.

State-Performance Trade-off. The trade-off between node-state and lookup path length is fundamental. The current known limit is that a constant node state can provide logarithmic path length. However, if fault-tolerance is to be addressed, more state is required. It is still an open question whether a constant state suffices for a fault-tolerant system while preserving the logarithmic path length and without introducing congestion. A good overview of this issue is covered in [67] and [29].

Cost of Maintaining the Structure. While we have seen throughout this chapter different techniques for dealing with the maintenance of the overlay structure, any optimization in that aspect is important for the overall performance of a DHT. For the class of DHTs that depend on the periodic checking and correction (aka stabilization), this periodic activity costs a high number of messages and sometimes unnecessarily in the case of checking stable sections of a routing table. The awareness about this problem motivated research such as, e.g., [35] where a network tries to "self-tune" the rate at which it performs periodic stabilization.

Complex Queries. DHTs assume that for each item, there is a unique key and to retrieve the item one must know the key. That is, one can not search for items matching a certain criteria like a keyword or a regular-expression-specified query. The feasibility of the task is

questionable [32]. Some approaches include the insertion of indices [26] for general queries or using some geometrical constructs that make use of the DHT structure such as space-filling curves [7]. Another approach is to let the hashing be based on keywords or semantic information and not on unique keys [59]. However, this approach destroys one of the basic assumptions of DHTs, namely uniformity of identifiers. In P-Grid, however because of the ability to deal with a non-uniform set of identifiers, efficient range queries could be provided as in [15].

Locality. Though accounted for in systems like Pastry and Tapestry, locality remains to be an open research issue. Additionally, the loss of locality due to hashing is not always considered a disadvantage. The Oceanstore system [31] which depends on Tapestry for location and routing, considers loss of locality favorable because replicas of items would be stored at physically apart nodes which renders a system resistant to denial of service attacks.

Heterogeneity. While all DHT systems aim at letting all nodes have equal duties and responsibilities, the heterogeneity in physical connectivity makes them unequal. Consequently, nodes with higher latencies constitute bottlenecks for the operation of structured P2P systems. Some of the approaches that were suggested to cope with those problems: i) Cloning: The more powerful nodes are cloned so they can act as multiple nodes and receive higher percentage of the uniformly distributed traffic [14] ii) Clustering: Nodes of similar latency behavior are clustered together [68] iii) A third approach is to build an "express way", i.e. an auxiliary overlay that contains only nodes with large forwarding capacity. Each node in the overlay is connected to a variable number of nodes in the auxiliary overlay [68].

Group Communication. Since structured P2P systems offer graphs of known topologies to connect peers, it is natural to start exploiting the structural properties in group communication. The main focus in P2P Group communication is on multicasting. Extensions like [60, 54, 11, 5] aim at providing multicast layers to existing DHT systems.

Publish-subscribe communication [63] is also another form of group communication that was researched in P2P systems such as [8].

Grid Integration. P2P and the Grid are two fields that share key properties such as being large scale distributed systems and the goal of sharing networked resources. The properties of scalability and self organization provided by recent P2P infrastructures are interesting properties for Grid applications. Actually, both research communities are starting to merge, we can observe that from conferences like the international conference on peer-To-peer computing[50] and the international conference on cluster computing and the Grid (CCGRID) [12]. Additionally, the P2P working group [24] and The Global Grid Forum [65], two respective standardization efforts, started to merge their efforts [9].

Performance of Existing DHTs Under Churn. At the time of writing of this thesis, while many systems have been introduced, more work is needed to measure their performance in different aspects and for different applications under different tuning parameters, especially under heavy churn. In an extensive study of various DHT systems under churn, Li et. al conclude the work in [33] by saying: "Evaluating DHT protocols in the presence of churn is a challenge."

Bibliography

[1] Karl Aberer. P-Grid: A self-organizing access structure for p2p information systems. In *InProceedings of the Sixth International Conference on Cooperative Information Systems (CoopIS 2001)*, Trento, Italy, 2001.

[2] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. The quest for balancing peer load in structured peer-to-peer systems. Technical Report EPFL Technical Report IC/2003/32, Ecole Polytechnique Federale de Lausanne (EPFL), 2003. http:// www.p-grid.org / Papers / TR-IC-2003-32.pdf.

[3] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. Efficient, self-contained handling of identity in peer-to-peer systems. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):858–869, 2004.

[4] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. DKS(N; k; f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *The 3rd International Workshop On Global and Peer-To-Peer Computing on Large Scale Distributed Systems (CCGRID 2003)*, Tokyo, Japan, May 2003. http://www.ccgrid.org/ccgrid2003.

[5] Luc Onana Alima, Ali Ghodsi, Per Brand, and Seif Haridi. Multicast in dks(n, k, f) overlay networks. In *The 7th International Conference on Principles of Distributed Systems (OPODIS'2003)*. Springer-Verlag, 2004.

[6] Luc Onana Alima, Ali Ghodsi, and Seif Haridi. A framework for structured peer-to-peer overlay networks. In *LNCS volume of the post-proceedings of the Global Computing 2004 workshop*. Springer-Verlag, 2004.

[7] Artur Andrzejak and Zhichen Xu. Scalable, Efficient Range Queires for Grid Information Services. In *2nd International Conference on Peer-To-Peer Computing*, pages 33–40, Linkping, Sweden, September 2002. IEEE Computer Scociety. ISBN-0-7695-1810-9.

[8] S. Baehni, P. Eugster, and R. Guerraoui. OS Support For P2P Programming: A Case For TPS. In *22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 355–362, Washington - Brussels - Tokyo, July 2002. IEEE.

[9] Per Brand and Karan Bhatia. Relation of OGSA/Globus and Peer2Peer, 2003. http://www.gridforum.org/4_GP/ogsap2p.htm.

[10] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[11] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[12] CCGRID. The IEEE International Symposium on Cluster Computing and the Grid. http://www.ccgrid.org.

[13] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving dns using chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

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

[15] Anwitaman Datta, Manfred Hauswirth, Renault John, Roman Schmidt, and Karl Aberer. Range queries in trie structured overlays. Technical Report EFPL Technical Report IC/2004/111, Ecole Polytechnique Federale de Lausanne (EPFL), 2004. http:// www.p-grid.org / Papers / TR-IC-2004-111.pdf.

[16] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient Broadcast in Structured P2P Netwoks. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[17] Emule. The emule file-sharing application homepage, 2004. http://www.emule-project.net/.

[18] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST., National Technical Information Service, Springfield, VA, April 1995.

[19] FreeNet, 2003. http://freenet.sourceforge.net.

[20] FreePastry. The freepastry homepage, 2004. http://www.cs.rice.edu / CS / Systems / Pastry / FreePastry.

[21] Ali Ghodsi, Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. Self-correcting broadcast in distributed hash tables. In *In Series on Parallel and Distributed Computing and Systems (PDCS'2003)*, Calgary, 2003. ACTA Press.

[22] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. A novel replication scheme for load-balancing and increased security. Technical Report TR-2004-11, SICS, June 2004.

[23] Gnutella, 2003. http://www.gnutella.com.

[24] Peer-To-Peer Working Group. What is Peer-To-Peer?, 2001. http://www.peer-to-peerwg.org/whatis/index.html.

[25] Steven Hand and Timothy Roscoe. Mnemosyne: Peer-to-peer steganographic storage. *Lecture Notes In Computer Science (IPTPS '02)*, pages 130–140, 2002.

[26] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[27] Clint Heyer. Naanou home page, 2004. http://naanou.sourceforge.net.

[28] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, ?-? 2002.

[29] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[30] Kazaa Home Page, 2003. http://www.kazaa.com.

[31] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon,

Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*. ACM, Nov 2000.

[32] Jinyang Li, Boon Thau Loo, Joe Hellerstein, Frans Kaashoek, David R. Karger, and Robert Morris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[33] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, and Thomer M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proc. of the 24th Infocom*, March 2005.

[34] Nancy Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in content addressable networks. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[35] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlayss. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[36] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *InProceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, August 2002.

[37] E. P. Markatos. Tracing a Large-Scale Peer to Peer System: An Hour in the Life of Gnutella. In *The Second International Symposium on Cluster Computing and the Grid*, 2002. http://www.ccgrid.org/ccgrid2002.

[38] MathWorld. The butterfly graph, 2004. http:// mathworld.wolfram.com / ButterflyGraph.html.

[39] MathWorld. The de bruijn graph, 2004. http:// mathworld.wolfram.com / deBruijnGraph.html.

[40] Petar Maymounkov and David Mazires. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[41] Mike Miller. *Discovering P2P*. Sybex International, November 2001. ISBN-0782140181.

[42] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, December 2002. USENIX Association.

[43] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *InProceedings of SPAA 2003*, 2003.

[44] Napster. Open source napster server, 2002. http://opennap.sourceforge.net/.

[45] OpenNap, 2001. http://opennap.sourceforge.net/.

[46] Andy Oram. What is P2P... And What isn't?, November 2000. http://www.oreillynet.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html.

[47] Andy Oram. *Peer-To-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, first edition, March 2001. ISBN:0-596-00110-X.

[48] Overnet. The overnet file-sharing application homepage, 2004. http://www.overnet.com.

[49] P-Grid. The P-Grid homepage, 2004. http://www.p-grid.org.

[50] P2P Conference. The IEEE International Conference on Peer-To-Peer Computing, Use of Computers at the Edge of Networks P2P, Grid, Clusters. http://www.ida.liu.se/conferences/p2p/p2p2002/.

[51] Planet-Lab. The planet-lab homepage. http://www.planet-lab.org.

[52] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[53] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content Addressable Network. In *Proceedings of the ACM SIGCOMM '01 Conference*, Berkeley, CA, August 2001.

[54] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Third International Workshop on Networked Group Communication (NGC '01)*, 2001. http://www-mice.cs.ucl.ac.uk/ngc2001/.

[55] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping The Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems And Implications For System Design. *IEEE Internet Computing Journal*, 6(1), 2002.

[56] 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)*, 329-350 2001.

[57] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 188-201 2001.

[58] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. HyperCuP – hypercubes, ontologies and efficient search on peer-to-peer networks, May 2003. http://www-db.stanford.edu/ schloss/docs/HyperCuP-LNCS2530.ps.

[59] Mario Schlosser, Michael Stinek, Stefan Decker, and Wolfgang Nejdl. A Scalable and Ontology-Based P2P Infrastructure for Semantic Web Services. In *2nd International Conference on Peer-To-Peer Computing*, pages 104–111, Linkping, Sweden, September 2002. IEEE Computer Scociety. ISBN-0-7695-1810-9.

[60] Ion Stoica, Dan Adkins, Sylvia Ratnasamy, Scott Shenker, Sonesh Surana, and Shelley Zhuang. Internet Indirection Infrastructure. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[61] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, August 2001.

[62] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, 2003.

[63] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Inc., 2002. ISBN-0-13-088893-1.

[64] The Chord Project Home Page, 2003. http://www.pdos.lcs.mit.edu/chord/.

[65] The Global Grid Forum. The Global Grid Forum Home Page, 2003. http://www.gridforum.org.

[66] Viceroy. Java implementation and visualization applet, 2004. http://www.ece.cmu.edu/ atalmy/viceroy.

[67] J. Xu, A. Kumar, and X. Yu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. *IEEE Journal on Selected Areas in Communications*, 22(1):151–163, January 2004. Preliminary version appeared in Proc. IEEE INFOCOM 2003.

[68] Zhichen Xu, Mahalingam Mallik, and Magnus Karlsson. Turning Heterogeneity into an Advantage in Overlay Routing. Technical Report HPL-2002-126R2, Hewlett-Packard Labs, July 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-126R2.html.

[69] Ben Y. Zhao, Ling Huang, Sean C. Rhea, Jeremy Stribling, Anthony D Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE J-SAC*, 22(1):41–53, January 2004.

[70] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2000.

[71] Feng Zhou, Li Zhuang, Ben Y. Zhao, Ling Huang, Anthony D. Joseph, and John D. Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems. In *Proc. of Middleware*, pages 1–20, Rio de Janeiro, Brazil, June 2003. ACM.

[72] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of NOSSDAV*, pages 11–20. ACM, June 2001.

Part I

Designs

Chapter 3

A Framework for P2P Lookup Services Based on k-ary Search

A Framework for Peer-To-Peer Lookup Services based on k-ary search

Sameh El-Ansary

Swedish Institute of Computer Science
Kista, Sweden

Luc Onana Alima

Department of Microelectronics and Information Technology
Royal Institute of Technology
Kista, Sweden

Per Brand

Swedish Institute of Computer Science
Kista, Sweden

Seif Haridi

Department of Microelectronics and Information Technology
Royal Institute of Technology
Kista, Sweden

Abstract

Locating entities in peer-to-peer environments is a fundamental operation. Recent studies show that the concept of distributed hash table can be used to design scalable lookup schemes with good performance (i.e. small routing table and lookup length). In this paper, we propose a simple framework for deriving decentralized lookup algorithms. The proposed framework is simple in that it is based on the well-known concept of k-ary search. To demonstrate the applicability of our framework, we show how it can be used to instantiate Chord. When deriving a generalized Chord from our framework, we obtain better performance in terms of the routing table size (38% smaller than the generalization suggested by the Chord authors).

Keywords: Lookup, peer-to-peer, distributed hash table, k-ary search.

3.1 Introduction

Peer-to-peer systems emerged as a special field of distributed systems where the lack of centralized control is a key requirement. Lookup services is one area in the peer-to-peer field that deserves a particular attention as a lookup service is a core requirement in peer-to-peer systems and applications. Given a certain key, the main task of a lookup service is to locate a network node that is responsible for that key.

The lookup problem in peer-to-peer systems has been approached in several ways. In our view, existing lookup services could be categorized based on two main properties: $i)$ scalability, $ii)$ hit guarantee, i.e., possibility of locating an entity in the system given that it is present. Depending on the application, other properties such as security and anonymity may be of interest.

In most of the early peer-to-peer systems such as Napster [3], Gnutella [2] and FreeNet [1], the hit guarantee and the scalability properties are either missing or not simultaneously satisfied. For example, the centralized directory in Napster offers the hit guarantee property while it renders the system unscalable. In Gnutella, the flooding approach prevents it from being scalable [5]. Furthermore, the hit guarantee is limited to the scope of the flooding. Similarly, in FreeNet the search scope is bounded and the use of caching can lead to inconsistent views of the network. The scalability of FreeNet is still to be evaluated.

Later approaches to the lookup problem are based on the concept of *Distributed Hash Table* (DHT). This approach is represented, for example, by systems such as Chord [6], Tapestry [8] and CAN [4]. The idea behind this approach is to let all the names of the different entities in the system be mapped to a single search space by using a certain hashing function and thus all the entities in the system have a consistent view of that mapping. Given that consistent view, various structures of the search space are used for locating entities. For example, in Chord, the search space is structured as a ring. In Tapestry, it is structured as a mesh. In CAN, it is structured as a d-dimensional coordinate space.

The hit guarantee property is well-addressed in the three above-mentioned systems as the whole search space is considered by the indexing structures

	Lookup length	Routing entries	Comments
Chord	$log_2(N)$	$log_2(N)$	N, system size
Tapestry	$log_b(N)$	$blog_b(N)$	b, search space encoding base
CAN	$\frac{d}{4}n^{\frac{1}{d}}$	$2d$	d, some constant

Table 3.1: Lookup length and routing information required in three DHT-based lookup services

in the three cases of ring, mesh and d-dimensional space and is no longer limited to the scope of a certain query. The different indexing structure are realized by means of routing tables. The hit guarantee is offered under normal failure conditions as the three algorithms provide fault-handling mechanisms to repair outdated routing tables. Scalability is also well-addressed because of the fact that a reasonable amount of routing information is required in order to offer an acceptable *lookup length* (i.e., number of hops to resolve a query). Table 3.1 shows that Chord and Tapestry both offer a lookup length and a number of routing table entries that grow logarithmically with the system size. CAN offers a lookup length that grows with the system size as a polynomial with order $1/d$, for some constant d and requires a constant amount of routing information.

3.1.1 Motivation and contribution

After exploration of some of the DHT-based lookup services, we were interested to answer the following question: *Is there a general abstraction that can be used to derive most of the existing DHT lookup services?*

By investigating the question, we observed that the idea of k-ary search seems to be general enough to derive several DHT-based lookup algorithms.

In this paper we show that:

- The lookup problem in peer-to-peer networks could be perceived as k-ary search.
- The DHT-based lookup service, Chord, is a special case of k-ary search where $k = 2$, i.e. performing binary search.

- This line of thinking can improve the lookup length of Chord and the number of routing table entries.

In general, DHT-based lookup services have three basic operations: Insertion, deletion and lookup. The scope of this paper will cover only the lookup operation. In a future paper, we will show how the k-ary search framework can simplify the insertion and deletion operations.

To present the suggested framework, in section 3.2, we introduce the Chord algorithm. In section 3.3, we show how the Chord algorithm can be perceived as an algorithm that mimics binary search. In section 3.4, we show how to perceive the lookup problem as k-ary search. Based on this result, in section 3.5, we show how the k-ary search framework can improve Chord lookup algorithm and the number of routing table entries. Finally, we conclude our work and present future directions in section 3.6.

3.2 The Chord lookup algorithm

In this section, we review the Chord system without considering the aspects of node joins and failures. We only focus on the lookup functionality.

Assuming a network of nodes where each node is assigned a number of keys, the Chord system provides a lookup service. That is, given a key K, a node running the chord algorithm will be able to determine the node to which K is assigned.

3.2.1 The Chord identifier/search space

The nodes' addresses and the keys of data items are both hashed to form a single identifier space. Where each identifier is encoded using m-bits. The identifiers are ordered in an identifier circle modulo 2^m.

3.2.2 Key assignment

Each identifier in the circle corresponds either to a node address or a key of a data item. Let *ID* be the function that maps nodes and keys to the identifier space. We say that a key K is assigned to node n iff

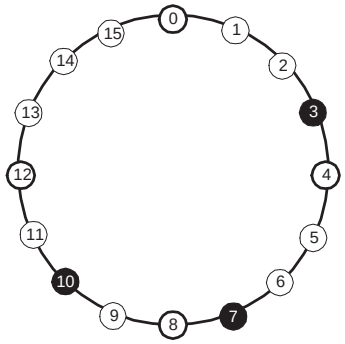


Figure 3.1: An example Chord network with 16 identifiers.

- $ID(K) = ID(n)$ or
- $ID(n)$ is the first identifier that corresponds to a node in the clockwise traversal of the identifier circle, starting from $ID(K)$.

When a key K is assigned to a node n, we say that node n is the successor of K. From now on, we do not make a distinction between a key and its identifier. The same applies for the nodes. Therefore, for an identifier k, we write *successor*(k) to denote the node to which, the key that maps to k is assigned.

Using the system depicted in Figure 3.1, which has three nodes, namely node 3, 7 and 10, the idea of key assignment is as follows. All identifiers from 11 to 3 are assigned to node 3; all identifiers from 4 to 7 are assigned to node 7 and all identifiers from 8 to 10 are assigned to node 10.

3.2.3 The routing table

Each node in the Chord network maintains a routing table of m entries called the *finger table*. At a given node n, the i-th entry of this table, stores

the node s such that s is the successor of $n \oplus_{2^m} 2^{i-1}$[1].

3.2.4 Key location

In this subsection, we briefly describe how to find the location of keys in a Chord network. When a node n receives a query for a key k, n will use its fingers as follows:

- If $k \in]n$,*successor*$(n)]$ then n returns successor of n and we say the query is resolved.
- If $k \notin]n$,*successor*$(n)]$ then, node n forwards the query to the node n', which is the closest preceding node of k according to n's finger table. When n' receives the forwarded query, it acts like node n.

3.2.5 Complexity

The m-th entry of each finger table contains the address of the node f_m, where $f_m =$ *successor*$(n \oplus_{2^m} 2^{m-1})$. Thus, if a query cannot be resolved at a node n, the node n will forward the query to f_m, which is at least half way between n and the target. Using this argument, it is proven in [7] that $log_2(N)$ hops are sufficient to resolve a given query with a routing table of $log_2(N)$ entries.

3.3 Chord as binary-search

Although not explicitly stated in [6, 7], we can see that the Chord lookup algorithm mimics binary search. Seeing the Chord lookup as a binary search simplifies its understanding. In this section, we show how this is the case. Before explaining, we introduce the following definition:

Definition 3.3.1 *Let $I =]x, y]$ be an interval of identifiers. We call the node with identifier x, the responsible for I.*

[1]The notation $x \oplus_z y$ is used to denote $(x + y)\ mod\ z$.

The definition above deserves a comment. The responsible for a given interval I, is the node to which a query for a key k is forwarded once it is determined that k belongs to I.

To show how the Chord lookup algorithm can be perceived as binary search, we consider a fully populated Chord network with 16 nodes. We say that a Chord network is fully populated when there is a node at each identifier of the identifier space.

In order to determine the location of a key, a query is introduced to the Chord network. A query arrives at a node either as an *original* query or as a *forwarded* query. Therefore, a precise characterization of a query Q at an arbitrary node n can be given in terms of the number of hops that Q made in order to reach node n. Hence, an original query made zero hops while a forwarded query made one or more hops. We will denote a query that made i hops, $i \geq 0$, an i-hop query.

Let us see how the Chord lookup algorithm determines the location of key k assuming that the original query for k arrives at node 0.

When the original (or the 0-hop) query for k arrives at node 0, node 0 determines the search space for k, which for node 0, is the whole identifier space, denoted $]0, 0]$, traversing the ring clockwise. Then, node 0 performs the following steps:

1. Using its 4-th entry of the finger table, node 0 divides the search space into the two intervals $]0, 8]$ and $]8, 0]$.

2. Determines the interval to which k belongs.

3. Forwards the query to the node responsible for the interval to which k belongs. Given the two intervals above, the query is forwarded either to node 0 itself or to node 8.

At this point, two cases are to be considered depending on which node the query is forwarded to.

Case 1: the query was forwarded to node 0 itself. In this case, node 0 receives the query after *one* hop and performs the following steps:

1. Using its 3-rd entry of the finger table, node 0 divides the new search space (i.e. $]0, 8]$) into the two intervals $]0, 4]$ and $]4, 8]$.

2. Determines the interval to which k belongs.

3. Forwards the query to the node responsible for the interval to which k belongs. That is, the query is forwarded either to node 0 itself or to node 4.

Case 2: the query was forwarded to node 8. The characteristic of the forwarded query when it arrives at node 8 is that it made *one* hop. Thus, when node 8 receives this "one hop" query for k, node 8 determines that the search space for this query is $]8, 8 \oplus_{16} \frac{16}{2^1}]$ and 8 performs the following steps:

1. Using its 3-rd entry of the finger table, node 8 divides the search space for k into the two intervals $]8, 12]$ and $]12, 0]$.

2. Determines the interval to which k belongs.

3. Forwards the query to the node responsible for the interval to which k belongs. At this point, the query is forwarded to either node 8 itself or to node 12.

By continuing the above startegy of processing forwarded queries, we can observe that each node that receives an x-hop query, $0 \leq x \leq 3$, has only two forwarding alternatives, which means that the search process follows a path of a binary search tree. Figure 3.2 illustrates this behavior.

As illustrated in figure 3.2, after each hop, the search space is halved into two intervals. Therefore, any other node in the network is reachable from the originating node of the query, within 4 hops.

In general, if N and H are respectively the system size and the maximum number of hops, then when a node n receives a i-hop query, $0 \leq i \leq H - 1$, the node n does the following:

1. Determines the search space for the query. This search space is given by the interval
$$]n, n \oplus_N \frac{N}{2^i}] \tag{3.1}$$

2. Using the $(H-i)$-th entry of its finger table, node n divides the search space for k into two intervals: $]n, n \oplus_N \frac{N}{2^{i+1}}]$ and $]n \oplus_N \frac{N}{2^{i+1}}, n \oplus_N \frac{N}{2^i}]$

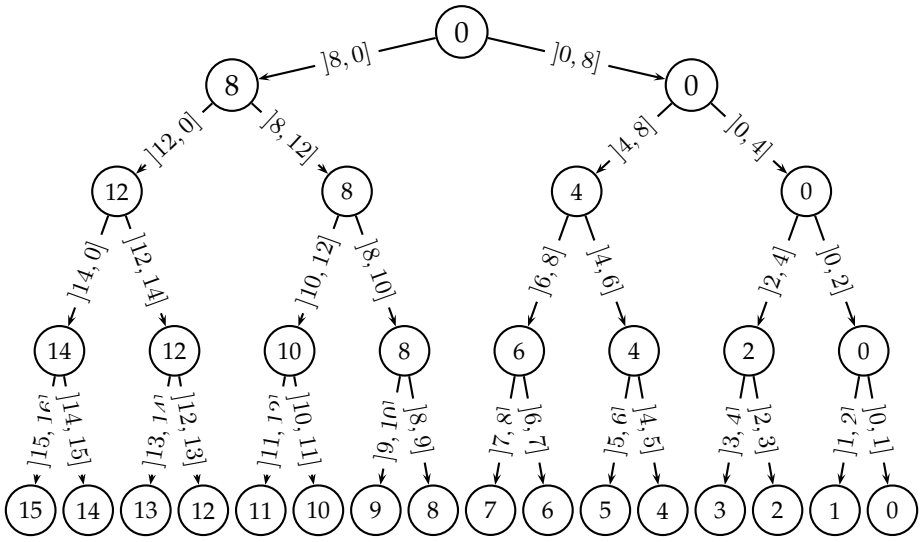


Figure 3.2: Decision tree for a query originating at node 0 in a 16-node network applying binary search

3. Determines the interval to which k belongs.

4. Forwards the query to the node responsible for the interval to which k belongs. More precisely, node n forwards the query to either node n itself or to node $n \oplus_N \frac{N}{2^{i+1}}$.

3.3.1 Complexity

Given that, for each query, the Chord lookup algorithm follows a path of a binary search tree rooted at the node where the query originated, the following results follow.

Theorem 3.3.1 (Lookup length) *The maximum number of hops for any query to be resolved, is $log_2(N)$.*

Proof : Follows from the fact that the height of a binary tree of N nodes is $log_2(N)$. ∎

Theorem 3.3.2 (Routing Table Entries) *The maximum number of routing table entries at each node is $log_2(N)$.*

Proof : Let n be an arbitrary node. From the above algorithm, node n must be able to forward any x-hop query, $0 \leq x \leq H - 1$, where H is the maximum number of hops required to resolve any query.

In order for the node n to route an x-hop query, the node n must select *exactly one* destination between *two* possible forwarding alternatives. But, as the node n does not need an entry for routing to itself, only one entry is needed.

Overall, since x varies from 0 to $H - 1$, and one entry is needed for each x-hop query, therefore, H entries in the routing table are needed.

Since n is an arbitrary node, the theorem follows. ∎

3.4 Lookup services as k-ary search

Having observed that the Chord algorithm mimics binary search, we generalize this idea to develop a modified algorithm that rather mimics k-ary

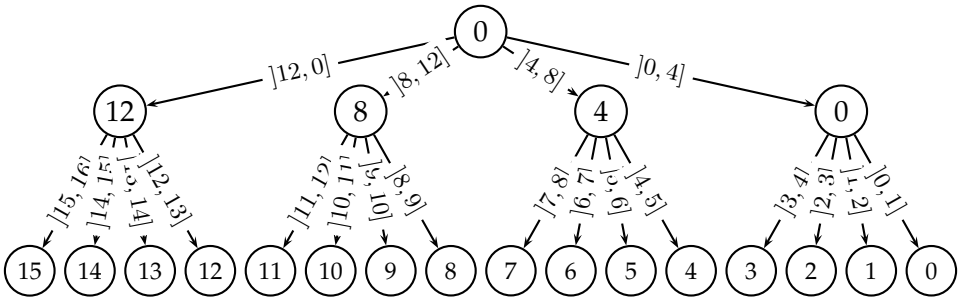


Figure 3.3: Decision tree for a query originating at node 0 in a 16-node network applying 4-ary search.

search , $k \geq 2$. We consider a fully populated system that consists of N nodes and assume that the maximum number of hops required to resolve any query is H. In addition, we assume that the identifier space is organized as a circle modulo N.

When a node n receives a i-hop query for key y, $0 \leq i \leq H-1$, the node n does the following:

1. Determines the search space for the for key y. This search space is given by the interval
$$]n, n \oplus_N \frac{N}{k^i}] \tag{3.2}$$

2. Using the $(H-i)$-th entry of its finger table, node n divides the search space for y into k intervals:
$]n \oplus_N \frac{N}{k^{i+1}} j, n \oplus_N \frac{N}{k^{i+1}}(j+1)]$, $0 \leq i \leq H-1$, $0 \leq j \leq k-1$.

3. Determines the interval to which y belongs.

4. Forwards the query to the node responsible for the interval to which y belongs. More precisely, node n forwards the query to one of the nodes:
$n \oplus_N \frac{N}{k^{i+1}} j$, $0 \leq j \leq k-1$.

Figure 3.3 illustrates the behavior of this algorithm in the case of $k = 4$ in a 16 node system.

3.4.1 Complexity

Given that for each query, the general algorithm presented in the above section, follows a path of a k-ary search tree rooted at the node where the query originated, the following results follow.

Theorem 3.4.1 (Lookup length) *The maximum number of hops for any query to be resolved, is $log_k(N)$.*

Proof : Follows from the fact that the height of a k-ary tree of N nodes is $log_k(N)$. ■

Theorem 3.4.2 (Routing Table Entries) *The maximum number of routing table entries at each node is (k-1)$log_k(N)$.*

Proof : Let n be an arbitrary node. From the above algorithm, node n must be able to forward any x-hop query, $0 \leq x \leq H-1$, where H is the maximum number of hops required to resolve any query.

In order for the node n to route an x-hop query, the node n must select *exactly one* destination between k possible forwarding alternatives. One of these destinations is the node n itself and as the node n does not need an entry for routing to itself, only $k-1$ entries are needed.

Overall, since x varies from 0 to $H-1$, and $k-1$ entries are needed for each x-hop query. Therefore, $(k-1)H$ entries in the routing table are needed.

Since n is an arbitrary node, the theorem follows. ■

	Chord(d)	k-ary Chord
H	$log_x(N)$	$log_k(N)$
R	$\frac{log(N)}{log(\frac{x}{x-1})}$	$(k-1)log_k(N)$

Table 3.2: Chord(d) vs. k-ary Chord

3.5 k-ary search for improving Chord

Having perceived chord as a special case of k-ary search, where $k = 2$, we can observe that if we need to improve the lookup length of Chord to a desired value H, we can choose a suitable k to achieve that value based on the following formula:

$$H = log_k(N)$$

The number of routing table entries, R will be:

$$R = (k-1)log_k(N)$$

We refer to our generalization of Chord by k-ary Chord.

The authors of the Chord system suggested as a future work in [7], a modification of the Chord lookup algorithm if a certain number of hops was desired. The modification suggested the placement of the fingers at intervals that are integer powers of $(1 + \frac{1}{d})$ instead of powers of 2, for some constant d. In that case, the lookup length is $log_{1+d}(N)$. The cost of the modification is an increase in the number of routing table entries to $\frac{log(N)}{log(1+\frac{1}{d})}$. We refer to this generalization by Chord(d).

In order to compare our result with the suggested generalization of the Chord authors, we take $x = 1 + d$ and we obtain table 3.2.

If we let $k = x = 4$, we can see that the number of routing table entries of the k-ary Chord is 38% smaller than Chord(d) as shown in table 3.3.

A more elaborate analysis of the size of the routing table as a function of the system size is shown in Figure 3.4.

N	$R_{\text{Chord}(d)}$	$R_{k-\text{ary Chord}}$
2^4	9.637683359	6
2^8	19.27536672	12
2^{16}	38.55073343	24
2^{32}	77.10146687	48
2^{64}	154.2029337	96

Table 3.3: Number of routing entries for different system sizes with $k = x = 4$

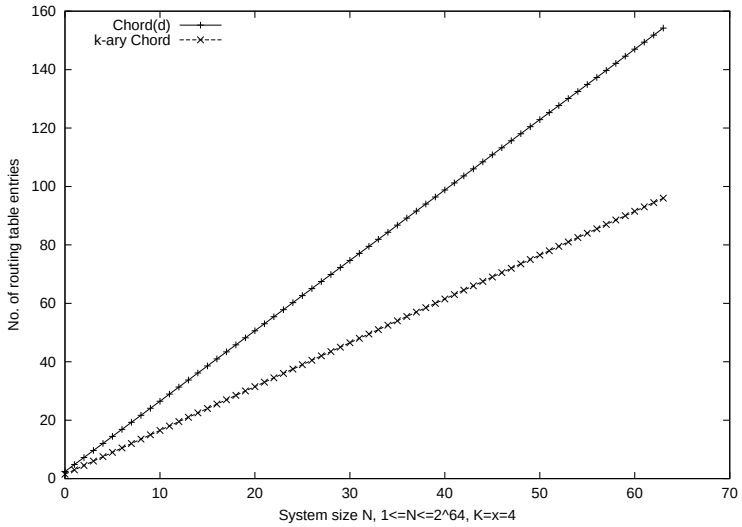


Figure 3.4: Evolution of routing table entries as a function of the system size.

3.6 Conclusion and future work

In this paper, we have presented a simple framework for designing distributed hash table based lookup services. The proposed framework is simple in that it is based on a well-known technique, that of k-ary search.

The paper shows how the idea of k-ary search can be used to derive the Chord lookup algorithm. More importantly, the generalization of the Chord lookup algorithm based on the k-ary search requires, for the same system size and the same lookup length, a routing table which is 38% smaller than the one required in the generalization suggested by the Chord authors.

As future work, we plan to show how this framework can be used to instantiate other distributed hash table based lookup algorithms. In addition, we show in a future paper how our framework simplifies the handling of node joins and failures.

Bibliography

[1] FreeNet. http://freenet.sourceforge.net.

[2] Gnutella. http://www.gnutella.com.

[3] Napster. http://freenet.sourceforge.net.

[4] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[5] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design, 2002.

[6] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001*, pages 149–160, San Deigo, CA, August 2001.

[7] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report TR-819, MIT, January 2002.

[8] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000.

Chapter 4

The $\mathcal{DKS}(N, k, f)$ Infrastructure for P2P Applications

$\mathcal{DKS}(N, k, f)$: A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications[1]

Luc Onana Alima[1], Sameh El-Ansary[2], Per Brand[2] and Seif Haridi[1]
[1]IMIT-Royal Institute of Technology, Kista, Sweden
[2]Swedish Institute of Computer Science, Kista, Sweden
{onana, seif}@it.kth.se, {sameh, perbrand}@sics.se

Abstract

In this paper, we present $\mathcal{DKS}(N, k, f)$, a family of infrastructures for building Peer-To-Peer applications. Each instance of $\mathcal{DKS}(N, k, f)$ is a fully decentralized overlay network characterized by three parameters: N the maximum number of nodes that can be in the network; k the search arity within the network and f the degree of fault-tolerance. Once these parameters are instantiated, the resulting network has several desirable properties. The first property, which is the main contribution of this paper, is that there is no separate procedure for maintaining routing tables; instead, any out-of-date or erroneous routing entry is eventually corrected on-the-fly thereby, eliminating unnecessary bandwidth consumption. The second property is that each lookup request is resolved in at most $\log_k(N)$ overlay hops under normal operations. Third, each node maintains only $(k-1)\log_k(N)+1$ addresses of other nodes for routing purposes. Fourth, new nodes can join and existing nodes can leave at will with a negligible disturbance to the ability to resolve lookups in $\log_k(N)$ hops in average. Fifth, any pair key/value that is inserted into the system is guaranteed to be located even in the presence of concurrent joins. Sixth, even if f consecutive nodes fail simultaneously, correct lookup is still guaranteed.

[1]This work is funded by the European IST-FET PEPITO project, the PIRATES project of the "Region Wallone" in Belgium and Vinnova PPC project in Sweden.

4.1 Introduction

The need for sharing information and computing resources in large scale networks is motivating a significant amount of research in the area of Peer-To-Peer (P2P) computing. The majority of recent research in this area focuses on providing Distributed Hash Tables (DHT) as infrastructures for building large scale P2P applications. The basic functionality of a DHT is to map *keys* to *values*. Most of the existing DHTs aim at achieving logarithmic routing table size and lookup length [6, 7, 4, 5].

A common characteristic of existing DHTs is the use of what we call *active correction* for maintaining routing tables. Typically, the active correction consists in running, periodically, specific routines whose sole purpose is to correct routing table entries [6, 7, 4, 5]. This results in additional bandwidth consumption, which in practical systems, constitutes a drawback.

To overcome the above-mentioned drawback, we propose an alternative approach in which there is no active correction. Instead, we use a technique we call *correction-on-use* by which, any out-of-date routing information is corrected on-the-fly while performing lookups and insertions of key/value pairs. Our design still provides logarithmic complexity as we demonstrate by simulation in the sequel.

4.1.1 Motivations and contributions

The motivation to our work is threefold. The first is the following observation.

> In P2P systems in which at any time, the number of lookups and key/value (or document) insertions is significantly higher than the number of joins, leaves and failures, the cost incurred by active correction is unnecessary.

The second is the need to build tunable (or flexible) infrastructures for building P2P applications. The rational behind flexibility is that for a given desirable system maximum size, N, one should be able to tune the infrastructure to achieve good balance between routing table size and lookup length.

The third is the *lookup reliability*. That is, we target infrastructures that guarantee to find the data associated to a key provided that the corresponding key/value pair were inserted into the system. Systems utilizing active correction usually do not meet this requirement.

The main contributions of this paper can be put as follows. First, we demonstrate the viability of the observation stated above. Second, we provide a design that is tunable and guarantees lookup reliability.

The result presented in this paper can be seen to some extent as a generalization of the Chord system [6]. Letting k be equal to 2 leads to a $\mathcal{DKS}(N, k, f)$ with the same routing table size and the same lookup length as in Chord. However, Chord uses active correction, which has a high and unnecessary communication cost.

The technique we use to correct routing tables is general enough to be applied to other systems. We demonstrate the validity of our approach by means of simulation.

4.1.2 An overview of our approach

Our approach builds on two main ideas. First, we use a form of interval routing that we call *distributed k-ary search* [3] (Hence, the name $\mathcal{DKS}$). Second, we use a novel technique that we call *correction-on-use* for maintaining routing tables. In this section, we briefly introduce these two ideas.

The distributed k-ary search principle is as follows. Given a key identifier t, in order to resolve the key t (i.e. find the data associated to t), the distributed k-ary search proceeds in $\log_k(N)$ steps. At the beginning of the search, the search space is equal to the whole identifier space. At each step of the (distributed) search, the current search space is divided into k equal parts. Each part is under the responsibility of a well chosen node. This partitioning of the search space is repeated until we reach k equal parts containing each only one element. At this point, the part that contains t is the one that is used to resolve the key t. Provided that the identifier space is a power of k, which we assume, it is easy to see that for an identifier space of size N, each lookup will require at most $\log_k(N)$ hops under normal operation. In addition, each node needs a routing table of only $(k-1)\log_k(N)$ entries as we demonstrate in the rest of this paper.

The correction-on-use technique is based on the fact that in P2P systems, having out-of-date routing entries should be considered as the normal situation rather than an exception due to frequent changes. Therefore, a challenging task is to provide efficient integrated mechanisms for handling out-of-date routing entries on-the-fly while providing logarithmic lookup length and reliable lookup.

Intuitively, the idea in the correction-on-use technique consists in embedding technical information into messages such that when a peer n' receives a message MSG from another peer n, peer n' can determine whether the routing information used by peer n when sending the message MSG was correct or not. Hence, if peer n' finds out that the routing information used by n was wrong, peer n' immediately informs peer n. In addition, peer n' tells peer n which peer it knows is a possible candidate for the routing entry used by n. When peer n receives such a notification, it updates the erroneous routing entry and repeats the operation that led it to contact n'. In this manner, any out-of-date routing information is eventually corrected; and if ever the system enters some steady period, no additional bandwidth consumption takes place in contrast to systems that use active correction (or periodical stabilization).

How fast the correction-on-use technique corrects routing entries depends on how frequently each routing entry is used. Intuitively, the more useful traffic (i.e. traffic related to lookup and key/value insertion), the higher is the correction rate of routing table entries.

To ensure lookup reliability, we let peers join and leave the system in an *atomic manner*. Fault-tolerance is achieved by means of replication as it is the case in traditional fault-tolerant systems [2].

4.1.3 Paper organization

The remaining of this paper is organized as follows. In section 4.2, we present the concepts used in the design of the $\mathcal{DKS}(N, k, f)$. Section 4.3 is devoted to the construction of a $\mathcal{DKS}(N, k, f)$ network. Section 4.4 describes how the *correction-on-use* is achieved. In Section 4.5 we briefly describe the protocol used for resolving any key identifier and correcting out-of-date (or erroneous) routing entries. Section 4.6 describes how

nodes leave a $\mathcal{DKS}(N, k, f)$ network. Section 4.7 explains how failures are handled. Section 4.8 presents experimental results and finally, Section 4.9 concludes.

4.2 The concepts in the design of the $\mathcal{DKS}(N, k, f)$

In this section, we present the concepts behind the design of the $\mathcal{DKS}(N, k, f)$ systems. We begin by giving the underlying assumptions.

4.2.1 Underlying assumptions

For the design of the $\mathcal{DKS}(N, k, f)$, we model a distributed system as a set of *processes* linked together through a *communication network*. Processes communicate by message passing. The communication network is assumed to be *connected*, *asynchronous*, *reliable* and *FIFO*.

To set up a $\mathcal{DKS}(N, k, f)$ system, it is assumed that: k is an integer greater or equal 2. The maximum number of nodes that can be in the system is $N = k^L$ where L is assumed to be large enough to achieve very large distributed systems. f is the fault-tolerance parameter. Each peer knows the parameters N, k and f, thus can compute L.

4.2.2 The identifier space and notations

In designing $\mathcal{DKS}(N, k, f)$ systems, we assume, like in most P2P infrastructures [6, 5], that nodes of a $\mathcal{DKS}(N, k, f)$ and objects managed by these nodes are uniquely identified by identifiers taken from the same logical space.

In this paper, we assume that the identifier space, denoted $\mathcal{I} = \{0, 1, \cdots, N-1\}$ is organized as a ring. It is worth pointing out that this choice is arbitrary, because the design principle shown in this paper can be applied for other organizations of the identifier space.

Given, the ring of at most N identifiers, some definitions are in order. First, we note that the whole identifier space can be represented by an interval of the form $[x, x[$ or $]x, x]$ for an arbitrary $x \in \mathcal{I}$. For any $x \in \mathcal{I}$, we

note that $[x, x] = \{x\}$ and $]x, x[= \mathcal{I}\backslash\{x\}$. From now on, we use for $a, b \in \mathcal{I}$, $a \ominus b$ for $(a - b)$ modulo N and $a \oplus b$ for $(a + b)$ modulo N.

Next in the paper, we shall need to determine whether a given identifier belongs to a part (or interval) of the identifier space. For this reason, we use an appropriate boolean function $\hat{\in}$, which will serve that purpose. For simplicity of notation, we shall use infix notation for the function $\hat{\in}$.

4.2.3 Key/value pairs management

Let (t, v) be a key/value pair, where t is a key identifier and v is the value associated with t. When inserted in a $\mathcal{DKS}(N, k, f)$ network, this pair is stored at the first node met moving on the identifier space starting from t, in the clockwise direction. When a pair (t, v) is stored at a node n, we say that node n manages the key t.

4.2.4 Levels and views

Any $\mathcal{DKS}(N, k, f)$ network is built in a manner that allows each lookup to be performed as a *distributed k-ary search* [3] to ensure that each lookup is resolved in at most $\log_k(N)$ hops. To achieve this, each node in a $\mathcal{DKS}(N, k, f)$ network has $\log_k(N)$ *levels* numbered from 1 to L, where $L = \log_k(N)$. In the sequel, we shall use $\mathcal{L}$ for $\{1, 2, \cdots, L\}$.

When at level $l \in \mathcal{L}$, a node n has a *view* V^l of the identifier space. The view V^l consists of k equal parts, denoted I_i^l, $0 \leq i \leq k - 1$, and defined below level by level. Next, we use $\mathcal{K}$ for $\{0, 1, \cdots, k - 1\}$.

- At level 1:

$$\begin{array}{l} V^1 = I_0^1 \cup I_1^1 \cup I_2^1 \cup \cdots \cup I_{k-1}^1 \text{ where} \\ I_0^1 = [x_0^1, x_1^1[,\ I_1^1 = [x_1^1, x_2^1[, \cdots,\ I_{k-1}^1 = [x_{k-1}^1, x_0^1[\\ x_i^1 = n \oplus i\frac{N}{k}, \text{ for } 0 \leq i \leq k - 1 \end{array}$$

- At level $2 \leq l \leq L$:

$$\begin{array}{l} V^l = I_0^l \cup I_1^l \cup I_2^l \cup \cdots \cup I_{k-1}^l \text{ where} \\ I_0^l = [x_0^l, x_1^l[,\ I_1^l = [x_1^l, x_2^l[, \cdots,\ I_{k-1}^l = [x_{k-1}^l, x_1^{l-1}[\\ x_i^l = n \oplus i\frac{N}{k^l}, \text{ for } 0 \leq i \leq k - 1 \end{array}$$

Notice that the end of I^l_{k-1} is equal to the end of I^{l-1}_0.

4.2.5 Responsibilities

Let n be an arbitrary $\mathcal{DKS}(N, k, f)$ node. Let l, $1 \leq l \leq L$, and let $V^l = \cup_{0 \leq j \leq k-1} I^l_j$ be the view that node n has at level l. With respect to node n, each I^l_j, $0 \leq j \leq k-1$ has associated to it a node, $R(I^l_j)$, that node n considers *responsible* (or *representative*) for I^l_j. Intuitively, the responsible for I^l_j represents the node that n will contact for example, when trying to resolve a key identifier that belongs to I^l_j.

The choice of representative nodes for the views determine the routing mechanism used for resolving keys. In this paper we show one possible way of selecting representatives. The choice presented here is similar to the one made in the Chord system. However, there are several other possibilities.

Let x be an identifier. We denote by $S(x)$ the first node encountered in the interval $[x, x[$ moving in the clockwise direction.

For an arbitrary node n and an arbitrary level $l \in \mathcal{L}$, the representatives for I^l_i, $i \in \mathcal{K}$ are given by Table 4.1, where the responsible for an interval $I^l_i = [x^l_i, x^l_{i+1}[$ is taken to be $S(x^l_i)$. Notice that each node is itself responsible for I^l_0 for any $1 \leq l \leq \log_k(N)$.

I^l_i	$R(I^l_i)$
I^l_0	n
I^l_1	$S(x^l_1)$
$\vdots$	$\vdots$
I^l_{k-1}	$S(x^l_{k-1})$

Table 4.1: Responsibilities at node n.

4.2.6 Routing information

To ensure that each lookup has a path length of at most $\log_k(N)$, each node n of a $\mathcal{DKS}(N, k, f)$ system maintains a routing table that is organized as shown in Table 4.2.

Level	Intervals	Responsible
1	I_0^1	n
	I_1^1	$S(x_1^1)$
	I_2^1	$S(x_2^1)$
	$\cdots$	$\cdots$
	I_{k-1}^1	$S(x_{k-1}^1)$
$\cdots$		
$L-1$	I_0^{L-1}	n
	I_1^{L-1}	$S(x_1^{L-1})$
	I_2^L	$S(x_2^{L-1})$
	$\cdots$	$\cdots$
	I_{k-1}^L	$S(x_{k-1}^{L-1})$
L	I_0^L	n
	I_1^L	$S(n \oplus 1)$
	$\cdots$	$\cdots$
	I_{k-1}^L	$S(n \oplus (k-1))$

Table 4.2: Routing table of the $\mathcal{DKS}(N, k, f)$ node n.

Notice that at each level, an arbitrary node n has an entry for itself and only $k-1$ entries for other nodes. Hence, the total number of entries in each routing table is $(k-1)\log_k(N)$ (without counting n).

From now on, we model each routing table as a mapping *RT*, which is of type

$$RT \quad : \quad \mathcal{L} \rightarrow (\mathcal{K} \rightarrow \mathcal{I})$$

Hence, $(RT\,(l))\,(i)$ denotes the responsible for the interval I_i^l. Sometimes, for the sake of clarity, we write RT_n, to emphasize that the routing table under consideration is that of node n.

In addition to the above routing table, each node n maintains a pointer denoted p, to its predecessor on the ring. The predecessor of a node n is the first node met when moving in the counterclockwise direction starting from n. So, in total, each node needs only $(k-1)\log_k(N)+1$ for the purpose of the lookup.

4.3 $\mathcal{DKS}(N, k, f)$ networks construction

Let $\mathcal{N}$ be a $\mathcal{DKS}(N, k, f)$ network. Note that by definition, $\mathcal{N}$ is either an empty set or a non-empty set of nodes. Let n_j be a $\mathcal{DKS}(N, k, f)$ node that wants to join $\mathcal{N}$. Two situations are to be considered depending on whether or not $\mathcal{N}$ is empty.

To handle insertion of new peers (or nodes), we add a component denoted pP to each node in a $\mathcal{DKS}(N, k, f)$ network. The component pP at any node is used only for atomic insertion of new nodes. At any point in time, the pP of a node n contains either the address of the new node that node n is currently inserting or the value **nil**. The pointer pP of a node n is **nil** if node n is not inserting a new node.

Joining an empty $\mathcal{DKS}(N, k, f)$. In this case, node n_j is the first node of $\mathcal{N}$. The insertion of n_j simply amounts to perform the following:

- for each $1 \leq l \leq \log_k(N)$ and $i \in \mathcal{K}$, set $(RT(l))(i)$ to n_j
- set p to n_j and pP, to **nil**.

Joining a non-empty $\mathcal{DKS}(N, k, f)$. To join a non-empty $\mathcal{DKS}(N, k, f)$ network $\mathcal{N}$, the joining node n_j sends a join request message to a known node that is currently in $\mathcal{N}$. The join request by n_j is possibly forwarded until it reaches the node n that is currently the successor of n_j. To simplify the understanding of the insertion of n_j in the system, we consider two cases. (i) The first case is when n is currently the only node of the system. (ii) The case where n is not the only node in the system, thus has a predecessor p, which is different from n.

The intuition in both cases is that node n will compute according to appropriate invariants, an approximate routing table for the new node n_j.

Case (i)**:** The insertion of n_j in this case is made as follows. Node n computes the routing table of n_j according to the following formula.

$$\left(\forall l \in \mathcal{L} : \left(RT_{n_j}(l)\right)(0) = n_j\right) \tag{4.1}$$

$$\begin{gathered}\left(\forall l \in \mathcal{L}, i \in \mathcal{K}\backslash\{0\} : \left(n_j \oplus i\tfrac{N}{k^l}\right) \hat{\in}]n_j, n] : \right. \\ \left.\left(RT_{n_j}(l)\right)(i) = n\right)\end{gathered} \tag{4.2}$$

$$\begin{gathered}\left(\forall l \in \mathcal{L}, i \in \mathcal{K}\backslash\{0\} : \left(n_j \oplus i\tfrac{N}{k^l}\right) \hat{\in}]n, n_j[: \right. \\ \left.\left(RT_{n_j}(l)\right)(i) = n_j\right)\end{gathered} \tag{4.3}$$

Naturally, node n adapts its routing table to account the arrival of the new node n_j. The predecessor pointers of n_j and n are properly set.

Case (ii)**:** In this case, node n has a predecessor p, which is different from n. Node n computes an approximate routing table for n_j according to the following formula.

$$\left(\forall l \in \mathcal{L} : \left(RT_{n_j}(l)\right)(0) = n_j\right) \tag{4.4}$$

$$\begin{gathered}\left(\forall l \in \mathcal{L}, i \in \mathcal{K}\backslash\{0\} : \left(n_j \oplus i\tfrac{N}{k^l}\right) \hat{\in}]n_j, n] : \right. \\ \left.\left(RT_{n_j}(l)\right)(i) = n\right)\end{gathered} \tag{4.5}$$

$$\begin{gathered}\left(\forall l \in \mathcal{L}, i \in \mathcal{K}\backslash\{0\} : \left(n_j \oplus i\tfrac{N}{k^l}\right) \hat{\in}]p, n_j] : \right. \\ \left.\left(RT_{n_j}(l)\right)(i) = n_j\right)\end{gathered} \tag{4.6}$$

$$\begin{gathered}\left(\forall l \in \mathcal{L}, i \in \mathcal{K}\backslash\{0\} : \left(n_j \oplus i\tfrac{N}{k^l}\right) \hat{\in}]n, p] : \right. \\ \left.\left(RT_{n_j}(l)\right)(i) = as\right)\end{gathered} \tag{4.7}$$

In (4.7), as is the node that is, according to the current knowledge of n, the successor of $\left(n_j \oplus i\frac{N}{k^l}\right)$. When the RT_{n_j} is computed, node n sends it to n_j. In addition, the message that carries RT_{n_j} contains also the information that the current predecessor of n will become the predecessor of n_j. Node n updates its predecessor from p to n_j; also, RT_n is revised to account the arrival of n_j.

More important to note at this point is that at the end of an insertion of a new node, the new node receives an approximate routing table that is computed without any lookup and satisfies the core invariant given in

(4.8) and that captures the idea that when a node n inserts a new node n_j, node n gives to n_j a routing table RT_{n_j} such that for any $l \in \mathcal{L}$ any $i \in \mathcal{K}$, $\left(RT_{n_j}(l)\right)(i)$ is equal to the node as_n that node n currently thinks is the successor of $(n_j \oplus i\frac{N}{k^l})$.

$$\left(\forall l \in \mathcal{L}, i \in \mathcal{K} : \left(RT_{n_j}(l)\right)(i) = as_n\right) \tag{4.8}$$

To ensure proper insertion of new nodes between n and its predecessor, we use *local atomic* insertion. More precisely, if concurrent joins happen to be between n and its predecessor, node n serializes them. Notice that the atomic insertion of a new node n_j by node n involves only n, the predecessor of n and n_j (when we do not consider fault-tolerance).

Given that concurrent joins at different parts of the circle can take place by the same time, and that when a node joins it obtains an approximate routing table, it is possible that routing entries can be out of date.

Handling such out of date information is one of the key contributions of this paper. Indeed, rather than using separate stabilization mechanism to be run periodically, we adopt another approach in which erroneous or out-of-date routing entries are detected and corrected on-the-fly. We intuitively present the technique used in the next subsection.

4.4 Correction-on-use

The correction-on-use technique builds on two simple observations.

Observation 1: By piggybacking the level and the interval information in lookup/insert messages, a remote node n' can determine upon receiving a message from n, whether the routing entry used by n to send the received message was correct.

Observation 2: A node n can determine upon receiving a message from a remote node n', whether it has an erroneous routing entry.

To exploit the first observation, we recall that by definition of responsibilities, at a node n, for any l $(1 \leq l \leq \log_k(N))$, and any $i \in \mathcal{K}$, we should have that $(RT_n(l))(i) = S(n \oplus i\frac{N}{k^l})$. This invariant will hold in any configuration of the system where each node has correct routing information.

Therefore, if a node n sends the level (i.e. l) and the interval information (i.e. i) while sending message to a node n', then node n' upon receiving the message can determine whether or not the entry used by n was correct. With respect to node n', the entry used by n to send a message carrying l and i, is correct *only if* node n' is the successor of $(n \oplus i\frac{N}{k^l})$. So, n' can accurately detect erroneous routing entries when receiving a message carrying the level and the interval used by the sender node.

When, a node n' detects that a node n contacted it using a level and an interval that shoud not have been used, node n' sends an error message to node n. This error message serves to inform node n that the entry it used to contact n' is erroneous. In addition, this error message carries the address of the predecessor of n' as the candidate node for correcting the routing entry used by n. When node n receives such an error message, it updates its routing table and repeates the operation (eg. lookup request, pair key/value insertion request) that led it to contact n', but now the message is sent to the predecessor of n'.

The exploitation of the second observation is immediate. Indeed, when a node n receives a message from a remote node n', node n checks its routing table to determine whether it should have n' in its routing table. If this is the case, node n updates its routing table accordingly.

4.5 Lookup in a $\mathcal{DKS}(N, k, f)$

The protocol for resolving keys in a $\mathcal{DKS}(N, k, l)$ serves also for correcting erroneous or out-of-date routing entries. As we have already sketched the idea of detection and correction of routing entries, in this section, we only present briefly how keys are resolved.

When a node n receives a lookup request for key identifier t, from its user, node n checks if t is between its predecessor, p and itself. If this is case, node n does a local lookup to find the value associated to t. The result is returned to the user. Otherwise, node n triggers a forwarding process that goes level by level, and that consists in routing lookup messages to the node that succeeds t on the identifier circle. Each lookup message carries necessary information (level and interval) for detection and correction of routing entries. When the node n' successor of t is reached, n' performs a

local lookup to retrieve the value associated to t. The result is forwarded backward or sent directly to the origin of the lookup.

Inserting key/value pairs in the system is similar to the lookup. In addition, messages for inserting key/value pairs are also used for detection and correction of routing entries.

4.6 Leave

Briefly, let n be a node that wants to leave the system. Let s be the successor of n. To start the leave process, node n asks its application layer to hand over the state to the application layer at s. From this point, node n enqueues all messages related to lookup, join of new nodes and to insertion of key/value pairs that arrive to it. When the state is transfered, node s is notified by its application layer. Upon receiving the notification from its application layer, node s sends a "you can leave" message to n that it can leave the system. Upon receiving the "you can leave" message, node n sends all the work enqueued to node s. Then, node n leaves without any additional message.

The departure of n is detected by any other node m that points to n when node m tries to communicate with n. Upon this detection, node m replaces n by the node it believes is the successor of $n + 1$.

The tricky part of the leave operation is when consecutive nodes attempt to leave by the same time. We manage this situation by a serialization mechanism that is omitted in this paper.

4.7 Failure

In $\mathcal{DKS}(N, k, f)$ we handle two failure situations. The first is the inability for two peers to communicate in a timely manner due to, for example, network congestion. This leads to timeout events. The second case is the *real failure* situation, where a peer fails by stopping and this fact is detected only if the site of the failed peer is still functional.

To achieve fault-tolerance, each node n maintains a list fl_n that contains the first $(f + 1)$ nodes that follows n on the identifier circle. Also,

the application layer replicates state accordingly to ensure that even if f consecutive nodes fail simultaneously, the system can still provide reliable lookup.

When a node n detects the crash of another node n', the behavior of n depends on whether or not $n' \in fl_n$. In the case $n' \in fl_n$, node n replaces node n' by the node n_s in fl_n that follows the failed node n'. In addition, the list fl_n is corrected using fl_{n_s}. If the failed node n' was not in fl_n, then node n replaces n' by the node n_{as}, that node n believes is the first node following n'. Notice that the believe by n might not be correct. However, the node n_{as} used for replacement is chosen such that it respects the invariant given in (4.8). Hence, even though the node used to replace the failed node n' is not correct, subsequent attempts of using the substitute node will eventually correct the corresponding routing entry.

4.8 Experimental results

The $\mathcal{DKS}(N, k, f)$ family is implemented and simulated using a distributed algorithms simulator developed by our team using the Mozart [1] programming platform.

Experiments setting. In order to evaluate the performance of our system, we conducted several experiments where the maximum system size is 2^{20}. Two of these experiments are reported here.

Experiment 1. The goal of this experiment was to observe the evolution of the lookup length while new nodes are joining and the number of lookups increases. To do this, we bootstrapped the system with 500 nodes using a search arity $k = 2$. Then, we inserted 10×2^{12} keys into the system. Afterward, we introduced concurrency by letting 3500 new nodes join while $\alpha \times 2^{12}$ (where $10 \leq \alpha \leq 100$) lookups are taking place. These concurrent events were scheduled at a rate of one event every 3 seconds following a Poisson distribution. The result of this experiment is shown in Figure 4.1. This figure shows that as we increase the number of lookups, the average lookup length tends to $\frac{1}{2}log_2(2^{10})$ and the 99th percentile of the

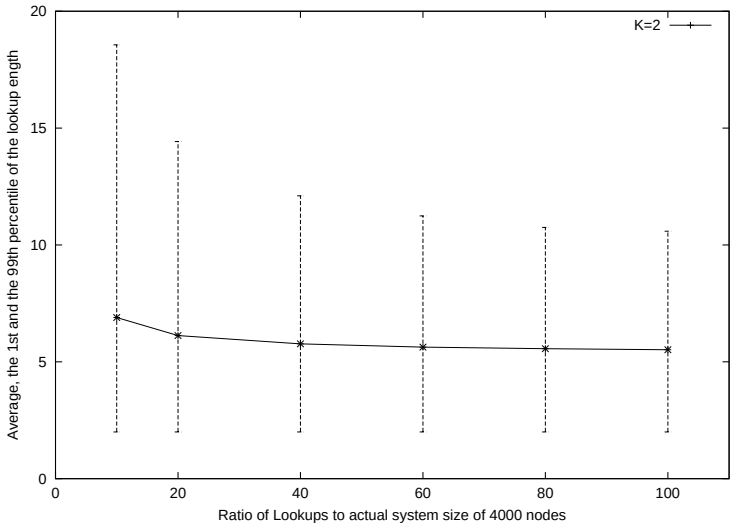


Figure 4.1: The average, the 1st and the 99th percentile of the lookup length as a result of increasing the lookup traffic in a system bootstrapped with 500 nodes and 3500 joins are done concurrently with lookups.

lookup length tends to $\log_2(2^{10})$. A reader familiar with the Chord system can see that those are the typical lookup bounds offered by the Chord system. The $\mathcal{DKS}(N, k, f)$ system offers the same bounds, yet without active stabilization.

Experiment 2. This experiment was conducted to observe the evolution of the lookup length while a proportion of the system is changing with concurrent joins and leaves and the number of lookup increases. For that, we bootstrapped the system with 2^{10} nodes. Then, we kept changing 20% of the system with 10% of joins and 10% of leaves happening concurrently. The experiment was repeated for search arity of 2 and 4. In figure Figure 4.2, one can observe that in the case of $k = 2$, the system is able to achieve the expected logarithmic lookup bounds when the number of lookups is 120 times the system size. However, in the case of $k = 4$,

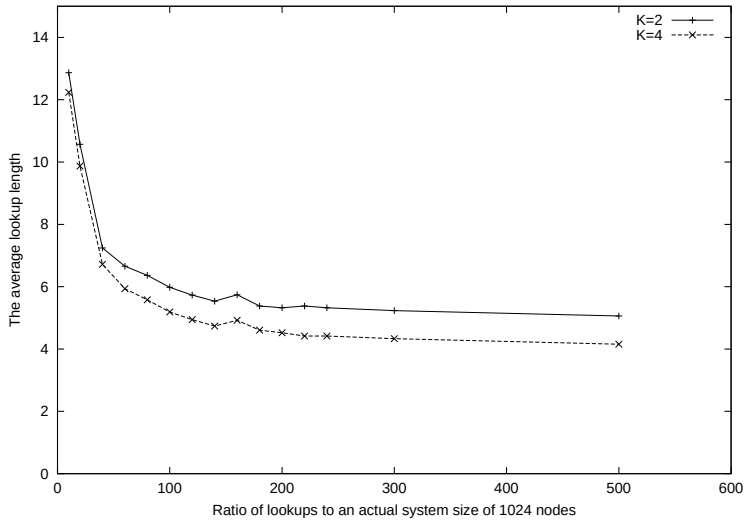


Figure 4.2: The average lookup length as a result of increasing the lookup traffic in a system of actual size 2^{10} while 10% of the nodes leave, and another 10% join concurrently.

the system is slowly converging towards its expected performance. i.e., more lookups are needed in that case. Moreover, in Figure 4.3, the 99th perecentile of the lookup length for the case where $k = 4$ tends to be high when there is no enough lookup traffic which is natural, since the number of out-of-date entries is larger because of the larger routing tables. As the lookup traffic increases, the system with $k = 4$, starts to outperform the system with $k = 2$. In all our experiments, the number of lookup failures observed was negligible with respect to the amount of lookup requests injected.

4.9 Concluding remarks and future work

In this paper, we presented $\mathcal{DKS}(N, k, f)$, a family of low communication, scalable and fault-tolerant infrastructures for building P2P applications.

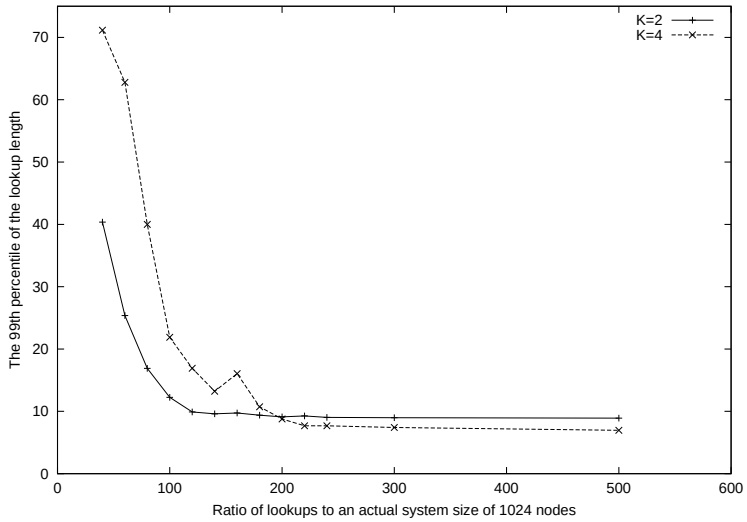


Figure 4.3: The 99th percentile of the lookup length as a result of increasing the lookup traffic in a system of actual size of 2^{10} while 10% of the nodes leave, and another 10% join concurrently.

The low communication of our infrastructures comes from the elimination of active correction, in which separate procedures are periodically run in order to maintain routing tables.

Our design is suitable for P2P systems in which, at any time, the traffic induced by lookup and key/value insertion is significantly higher than that induced by join, leave and failure. Notice that this assumption implies that at any time, the number of lookup and key/value insertion is significantly higher than the system size, because join, leave and failure affect the system size.

A $\mathcal{DKS}(N, k, f)$ can be seen as a generalization of the Chord system. Taking $k = 2$ gives a system with a routing table exactly as in Chord and a lookup length of $\log_2(N)$. However, Chord uses active correction.

The design proposed in this paper is close, to some extent, to systems such as Pastry [5] and Tapestry [7]. The main differences are that in these

systems, prefix routing and active correction are used. Furthermore, in Pastry, locality information is exploited while in our design, it is not. However, it is worth noticing that the idea of distributed k-ary search can be applied for prefix routing by reconsidering the identifier space.

The $\mathcal{DKS}(N, k, f)$ is designed to facilitate the development of P2P applications. This means that any $\mathcal{DKS}(N, k, f)$ should provide an API that supports a variety of operations. In this paper, we have presented only the LOOKUP. Currently, we are considering issues such as multicast/broadcast in $\mathcal{DKS}(N, k, f)$. As future work, we will investigate heterogeneity and locality in $\mathcal{DKS}(N, k, f)$ networks.

Acknowledgments

We would like to thank Peep Kungas, Samer Al-Kassimi for their help in implementing some of the routines of our simulation environment.

Bibliography

[1] Mozart Consortium. http://www.mozart-oz.org.

[2] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[3] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. A framework for peer-to-peer lookup services based on k-ary search. Technical Report TR-2002-06, SICS, May 2002.

[4] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

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

[6] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001*, pages 149–160, San Deigo, CA, August 2001.

[7] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000.

Chapter 5

Efficient Broadcast in Structured P2P Networks

Efficient Broadcast in Structured P2P Networks [1]

Sameh El-Ansary[1], Luc Onana Alima[2], Per Brand[1], and Seif Haridi[2]
[1]Swedish Institute of Computer Science, Kista, Sweden
{sameh,perbrand}@sics.se
[2]IMIT-Royal Institute of Technology, Kista, Sweden
{onana, seif}@imit.kth.se

Abstract

In this paper, we present an efficient algorithm for performing a broadcast operation with minimal cost in structured DHT-based P2P networks. In a system of N nodes, a broadcast message originating at an arbitrary node reaches all other nodes after exactly $N - 1$ messages. We emphasize the perception of a class of DHT systems as a form of distributed k-ary search and we take advantage of that perception in constructing a spanning tree that is utilized for efficient broadcasting. We consider broadcasting as a basic service that adds to existing DHTs the ability to search using arbitrary queries as well as dissiminate/collect global information.

[1]This work is funded by the Swedish funding agency VINNOVA, PPC project, the European IST-FET PEPITO project and by the PIRATES project at UCL, Belgium.

	Flooding	DHT
Queries	Arbitrary	Key Lookup
Query-Induced Traffic	$O(N)$	$O(log(N))$
Hit Guarantees	Low	High
Connectivity Graph	Random	Structured

Table 5.1: Flooding Approach vs. DHT Approach

5.1 Introduction

Research in P2P systems resulted in the creation of many Data/Resource-location systems. Two approaches were used to tackle this problem; the flooding approach and the Distributed Hash Table approach. The common characteristic of both approaches is the construction of an application-level overlay network. Table 5.1 includes some of the major differences between the two approaches.
The DHT approach with a structured overlay network, determinism, relatively low traffic and high guarantees is currently perceived in the P2P research community as the "reasonable" approach. Many systems were constructed based on that approach such as Tapestry [17], Pastry [13], CAN [10], Chord [14], Kademlia [8]. In contrast, the flooding-based approach represented by [5] [4] is mainly considered as unscalable based on a number of traffic analyses such as [7, 12].

A missing feature in most DHTs is the ability to perform search based on an arbitrary query rather than key lookups. Extensions to existing DHTs are needed to supply this feature. Arbitrary querying is realized in flooding-based systems via broadcasting. However, the random nature of the overlay network renders the solution costly and with low guarantees.

In this position paper, we show the status of our work on extending DHTs with an efficient broadcast layer. We are primarily investigating how to take advantage of the structured nature of the DHT overlay network in performing efficient broadcasts. We provide broadcasting as a basic service in DHTs that should be deployed for any kind of global dissemination/collection of data.

In the next section, we describe related work. In section 5.3, we explain our approach based on the perception of a class of DHTs as systems performing *distributed k-ary search*. In section 5.4, we present a broadcast algorithm for one of the DHTs, namely Chord. Some preliminary simulation results are presented in section 5.6. Finally, we conclude and show intended future work in section 5.7.

5.2 Related Work

Our work can be classified as an arbitrary-search-supporting extension to DHTs. From that perspective, the following research shares the same goal: **Complex Queries in DHTs**. In [6], an extension to existing DHT systems was suggested to add the ability of performing complex queries. The approach constructs search indices that enable the performance of database-like queries. This approach differs from ours in that we do not add extra indexing to the DHT. The analysis of the cost of construction, maintenance, and performing database-like join operations is not available at the time of writing of this paper.

Multicast. Since broadcast is a special case of multicast, a multicast solution developed for a DHT such as [15, 11, 2] can provide broadcast. Nevertheless, a multicast solution would require the additional maintenance of a multicast group which is, in the case of broadcast, a large group containing all the nodes of the network. In our approach, we do not require such an additional cost, we depend on the routing information of the already-maintained overlay network.

5.3 Our Approach

5.3.1 DHTs as Distributed k-ary Search

By looking at the class of DHT systems that have logarithmic performance bounds such as Chord, Tapestry, Pastry, and Kademlia, one can observe that the basic principle behind their operation is performing a form of distributed k-ary search. In the case of Chord, a binary search is performed. For other systems like, e.g., Tapestry and Pastry, the search arity is higher.

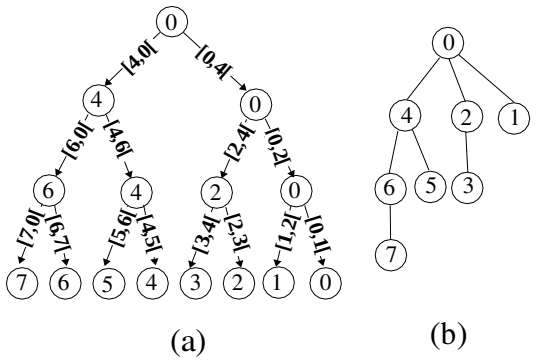


Figure 5.1: (a) Decision tree for a query originating at node 0 in a fully-populated 8-node Chord network. (b) The spanning tree derived from the decision tree by removing the virtual hops.

In this paper, we explain the perception of the Chord system as a special case of distributed k-ary search. The arguments apply to higher search arities as well.

The familiarity of the reader to the Chord system and its terminology is assumed. However, we restate the structure of the routing tables. Every Chord node has an identifier that represents its position in a circular identifier space of size N. Each Chord node maintains a table of $M = \log_2(N)$ routing entries, called the fingers. We denote the table of fingers at a node by *Finger*. At a node n, *Finger*$[i]$ contains the address of the successor of $n + 2^{i-1}$, $1 \leq i \leq M$.

To illustrate the idea of the distributed k-ary search, without loss of generality, we assume a Chord system with identifier space of size $N = 8$. The system is fully populated, i.e. a node is present for every identifier in the space. In Figure 5.1 (a), we show the decision tree of a lookup query originating at node 0. Given a query for a key whose identifier is x, node 0, starts to lookup for the node responsible for x by considering the whole identifier space as the search space. Based on the interval to which x belongs (arc labels in figure 1 (a), the query is forwarded and the process is repeated with the search space reduced to a half of the previous search

space. Hence, all nodes are reachable by a query-guided path of at most $H = \log_2(N)$ hops.

Notice that some of the hops are made from one node to itself. We call such hops, virtual hops. An important observation to be made from the decision tree shown in Figure 5.1 (a) is that a spanning tree can easily be derived by removing virtual hops. Figure 5.1 (b) shows a spanning tree derived from the decision tree by removing virtual hops. A more elaborate explanation on the idea of distributed k-ary search is presented in [1, 3].

5.3.2 Problem Definition

Having highlighted the idea of distributed k-ary search, we can now state the problem we solve in this paper.

Problem. *Given an overlay network constructed by a P2P DHT system, find an efficient algorithm for broadcasting messages. The algorithm should not depend on global knowledge of membership and should be of equal cost for any member in the system.*

Note that in the problem definition, we emphasize the P2P assumptions, i.e. the absence of central coordination and where every peer pays the same cost for running the algorithm.

5.3.3 Solutions

Efficient Broadcast. We base our solution on the fact that from the decision tree of the distributed k-ary search, a spanning tree can be derived by removing virtual hops. Figure 5.1 (b) shows a spanning tree derived from the binary decision tree for the 8-node Chord system. In section 5.4, we show how to construct this tree in a distributed fashion.

Gnutella-like Broadcast. A simple solution for the above-mentioned problem is to apply a Gnutella-like algorithm, where every node forwards a received query to its neighbors. This approach has an extra advantage when applied in a structured overlay network compared to a random network, namely, the ability to determine the diameter of the network. Speaking of the class of DHTs with logarithmic performance, one can set the Time-To-Live (TTL) parameter of the queries to the logarithm of the total number of nodes and be sure that the flooding process covers the whole

network instead of using a heuristic TTL that results in unknown guarantees. However, this solution retains the main property of non-scalability. In section 5.6, we compare Gnutella-like broadcasting to efficient broadcasting.

Ring Traversal. As the overlay network of a system like Chord is organized in a ring, traversing that ring by following the successor pointers is also a possible solution. The solution differs from our solution in execution time. That solution requires the sequential traversal of the ring while our algorithm reaches different parts of the network in parallel.

5.4 The Broadcast Algorithm

5.4.1 System Model & Notation

We assume a distributed system modeled by a set of nodes communicating by message passing through a communication network that is: (i) Connected, (ii) Asynchronous, (iii) Reliable, and (iv) providing FIFO communication.

A distributed algorithm running on a node of the system is described using rules of the form:

$$\frac{\textbf{receive}(\textit{Sender} : \textit{Receiver} : \text{MESSAGE}(arg_1, .., arg_n))}{\text{Action(s) ....}}$$

The rule describes the event of receiving a message MESSAGE at the *Receiver* node and the action(s) taken to handle that event. A *Sender* of a message executes the statement **send**(*Sender* : *Receiver* : MESSAGE($arg_1, .., arg_n$)) to send a message to *Receiver*.

5.4.2 Rules

Initiating a Broadcast. A broadcast is initiated at any node as a result of a user-level request. That is, a user-level layer entity P can send to a node Q a message INITBROADCAST(*Info*) where *Info* is a piece of information that must be broadcast e.g. an arbitrary search query, a statistics gathering query, a notification, etc.

```
receive(P : Q : INITBROADCAST(Info))
for i in 1 to M − 1 do
  //Skip a redundant finger
  if Finger[i] ≠ Finger[i + 1] then
    R := Finger[i]
    Limit := Finger[i + 1]
    send(Q:R:BROADCAST(Info, Limit))
  fi
od
//Process the M^th finger
send(Q:Finger[M]:BROADCAST(Info, Q))
```

Figure 5.2: Initiating a Broadcast Message

The role of the node Q is to act as a root for a spanning tree. As shown in the rule in Figure 5.2, Q does that by sending a BROADCAST message to all its neighbors. Note that, unless the identifier space is fully populated, the table *Finger* of a node contains many redundant fingers. For a sequence of redundant fingers, the last one is used for forwarding while the others are skipped.

A BROADCAST message contains the *Info* to be broadcast and a *Limit* argument. A *Limit* is used to restrict the forwarding space of a receiving node. The *Limit* of a *Finger*$[i]$ is *Finger*$[i+1]$, $(1 \leq i \leq M-1)$ where M is the number of entries of the routing table. The M^{th} finger's limit is a special case where the *Limit* is set to the sender's identifier. As an example, we use the sample Chord network given in section 5.3.1. When node 0 initiates a broadcast, it sends to nodes 1, 2, and 4. Giving them the limits of 2, 4, and 0 respectively. By doing that it is actually telling node 4 to cover the interval $[4, 0[$, i.e. half of the space. It is telling node 2 to cover the interval $[2, 4[$, i.e. quarter of space and finally, telling node 1 to cover the interval $[1, 2[$, i.e. an eighth of the space.

Processing a Broadcast. A node Q receiving a BROADCAST(*Info*, *Limit*) message delivers it to its application layer and continues the broadcast in a subtree confined in the interval $]Q, Limit[$. In addition to skipping the

receive($P : Q :$ BROADCAST(*Info*, *Limit*))

```
//Take some action to deliver to application layer ...
for i in 1 to M − 1 do
  //Skip a redundant finger
  if Finger[i] ≠ Finger[i + 1] then
    //Forward while within "Limit"
    if Finger[i] ∈]Q, Limit[ then
      R := Finger[i]
      //NewLimit must not exceed Limit
      if Finger[i + 1] ∈]Q, Limit[ then
        NewLimit := Finger[i + 1]
      else
        NewLimit := Limit
      fi
      send(Q:R:BROADCAST(Info, NewLimit))
    else
      exit for
    fi
  fi
od
```

Figure 5.3: Processing a Broadcast Message

redundant fingers, Q forwards to every finger whose identifier is before the *Limit*. Moreover, when forwarding to any finger, it supplies it with a *NewLimit*, defining a smaller subtree. Note that, this will only happen if *NewLimit* $\in]Q, Limit[$, i.e. the *NewLimit* is not exceeding the *Limit* given by the parent. Figure 5.3 contains the rule for processing a broadcast message.

Note that for any node other than the initiating node, the M^{th} finger will never be used, so we do not try to forward to it. In general, after h hops, the (M-h)th finger at most is used in forwarding. An additional invariant of the two rules that is not shown in the figures, for the simplicity of presentation, is that a node never sends a BROADCAST message to itself. A finger of a node n can point to n only in the rare case that half or more

of the identifier space does not contain any nodes which is most unlikely given the assumption of a uniform distribution of node identifiers.

Replies. We are considering the issue of replying to the broadcast source to be an orthogonal issue that depends on the *Info* argument of the BROADCAST message. Several strategies could be considered for replying, for example : (i) Sending the broadcast source with every broadcast message and it is contacted directly by a node willing to reply (ii) The reply is propagated to the root over the same spanning tree.

5.4.3 Correctness Argument

Coverage of All Nodes. As a DHT system constructs a *connected* graph of nodes and as every node that receives a broadcast message forwards it to all of its neighbors (except those it knows by DHT construction properties that they are going to be contacted by other nodes), therefore, *eventually* every node in the system receives the broadcast message.

No Redundancy. The algorithm ensures that disjoint (non-overlapping) intervals are considered for forwarding. Consequently every node receives the broadcast message exactly once.

5.5 Cost Versus Guarantees

While presenting an efficient algorithm for broadcast in DHT-based P2P networks, we are aware that the cost of N-1 messages, especially in large P2P systems can be prohibitive for many applications. The point is that we offer broadcasting as a basic service available for a system that is willing to pay its cost. Our algorithm offers strong guarantees and utilization of traffic for that endured cost. In order to offer the same guarantees on a network, of the same size, in a Gnutella-like broadcast, a substantially higher cost is paid. The next section elaborates more on this comparison.

Predictable Guarantees. The broadcast as presented in section 5.4, offers strong guarantees as it explores every node in the network. Minor modifications to the algorithm could be applied to, deterministically, reduce the scope of the broadcast and thus offer weaker, yet predictable guarantees. For example, by sending only to the M^{th} (or all but the M^{th})

finger while initiating a broadcast, only 50% of the network is covered in the broadcast. Similar pruning policies could be applied to achieve different coverage percentages.

Different Traversal Policies. The algorithm could also be modified to support an iterative deepening policy. This policy was suggested in [16] for use in unstructured overlay networks. We believe that combining this policy with our algorithm can decrease the messaging cost, especially, when one query hit suffices as a result.

5.6 Simulation Results

In this section, we show preliminary simulation results for the presented broadcast algorithm. We are primarily interested to see that all nodes are covered in the broadcast process and that no redundant messages are sent. Additionally, we want to compare the messaging cost of the efficient broadcast algorithm with that of the Gnutella broadcast algorithm over the same size of the network and with the same guarantees offered. The experiments were conducted on a distributed algorithms simulator developed by our team and using the Mozart [9] programming platform.

Experiments Setting. To study the messaging cost, we create an identifier space of size 2^{16} and we vary the number of nodes in the space, from 2^3 up to 2^{14} with increasing powers of 2. For each network size, after all the nodes join the system, we initiate a broadcast process starting at a randomly-chosen node. We wait until the broadcast process ends and, then, analyze the messages to see if all the nodes are covered and count the amount of redundant messages. We repeat the same experiment a number of times, initiating the broadcast from different sources.

Both the efficient and the Gnutella algorithms are evaluated in the same way. We use the basic Gnutella algorithm except that we deploy it on a structured rather than a randomly-connected overlay network. That is, the unique fingers of the Chord nodes are used as neighbors. Moreover, we set the Time-To-Live (TTL) parameter of the Gnutella broadcast to the diameter of the network , i.e. $\log_2(N)$ which should be just enough to guarantee that all the nodes of the network are covered.

Results. For the number of messages, the efficient broadcast algorithm

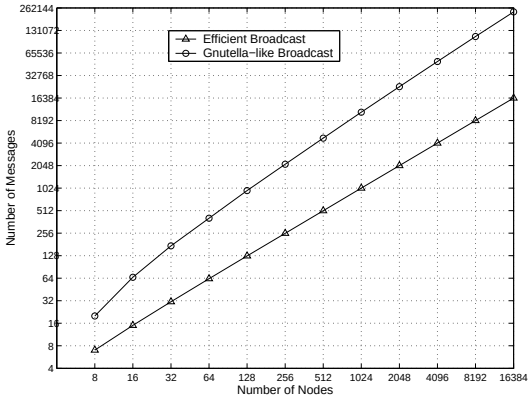


Figure 5.4: Comparison of number of messages needed to cover all nodes using efficient broadcast and Gnutella-like flooding in a structured network.

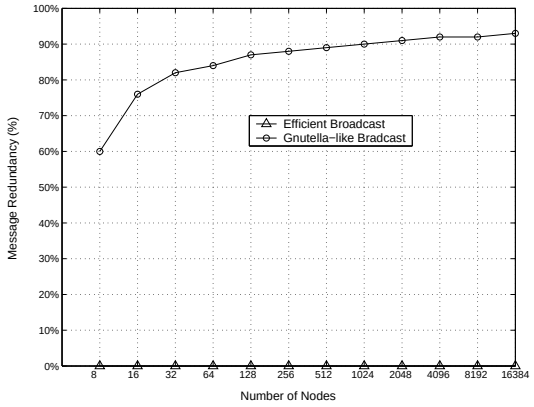


Figure 5.5: Comparison of percentage of redundant messages generated by efficient broadcast and Gnutella-like flooding in a structured network.

constantly produces N-1 messages for the different network sizes. The Gnutella algorithm succeeds to cover all the nodes, thanks, to the TTL parameter, but does that with a substantially larger amount of messages. The comparison is shown in figure 4. The reason for that difference is the redundant messages that are sent in the Gnutella case and are eliminated in the efficient broadcast case. It is worth noting that the amount of redundancy increases with system size, strongly affecting scalability if the strong guarantees are to be maintained. Figure 5 shows the percentage of redundant messages from the total number of messages generated by both algorithms.

5.7 Conclusion and Future Work

In this paper, we showed the status of our work in extending the functionality of DHTs with the ability to perform efficient broadcasts. Our approach depended mainly on the perception of systems such as Chord, Tapestry, Pastry, and Kademlia as implementations of distributed k-ary search. We gave an algorithm for traversing the k-ary search tree and thus, constructing a spanning tree of an overlay network formed by a DHT.

We based all our explanation on Chord as a simple system implementing binary search. In future papers, we intend to elaborate more on how to construct a spanning tree in systems with higher arities.

We suggested a number of strategies by which a peer deploying the efficient broadcast algorithm can reduce its scope by pruning a spanning tree in order to generate less traffic, yet with the ability to deterministically decide the percentage of network members that are covered in the broadcast and thus offering predictable guarantees. More experiments need to be done for the evaluation of those strategies.

For the issue of dynamic network (joins/leaves), more experimental results are needed to: (i) Quantify the effect of outdated routing tables on the properties offered by the efficient broadcast algorithm. (ii) Guide the design of a more fault-tolerant version of the algorithm. In its current state, our algorithm, depends heavily on the ability of the underlying DHT system to cope quickly with the dynamic nature of the network.

Bibliography

[1] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. Dks(n; k; f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *To appear in the 3rd International workshop on Global and Peer-To-Peer Computing on large scale distributed systems*, Tokyo, Japan, May 2003.

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

[3] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. A framework for peer-to-peer lookup services based on k-ary search. Technical Report TR-2002-06, SICS, May 2002.

[4] FreeNet. http://freenet.sourceforge.net, 2003.

[5] Gnutella. http://www.gnutella.com, 2003.

[6] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in dht-based peer-to-peer networks. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.

[7] E. P. Markatos. Tracing a large-scale peer to peer system: An hour in the life of gnutella. In *Second International Symposium on Cluster Computing and the Grid*, 2002.

[8] Petar Maymounkov and David Mazires. Kademlia: A peer-to-peer information system based on the xor metric. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.

[9] Mozart Consortium. http://www.mozart-oz.org, 2003.

[10] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[11] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Third International Workshop on Networked Group Communication (NGC '01)*, 2001.

[12] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design, 2002.

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

[14] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report TR-819, MIT, January 2002.

[15] Ion Stoica, Dan Adkins, Sylvia Ratnasamy, Scott Shenker, Sonesh Surana, and Shelley Zhuang. Internet indirection infrastructure. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.

[16] Beverly Yang and Hector Garcia-Molina. Efficient search in peer-to-peer networks. In *The 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, 2001.

[17] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000.

Chapter 6

Self-Correcting Broadcast in Distributed Hash Tables

Self-Correcting Broadcast in Distributed Hash Tables [1]

Ali Ghodsi[1],Luc Onana Alima[1], Sameh El-Ansary[2],
Per Brand[2] and Seif Haridi[1]
[1]IMIT-Royal Institute of Technology, Kista, Sweden
[2]Swedish Institute of Computer Science, Kista, Sweden
{aligh, onana, seif}@it.kth.se, {sameh, perbrand}@sics.se

Abstract

We present two broadcast algorithms that can be used on top of distributed hash tables (DHTs) to perform group communication and arbitrary queries. Unlike other P2P group communication mechanisms, which either embed extra information in the DHTs or use random overlay networks, our algorithms take advantage of the structured DHT overlay networks without maintaining additional information. The proposed algorithms do not send any redundant messages. Furthermore the two algorithms ensure 100% coverage of the nodes in the system even when routing information is outdated as a result of dynamism in the network. The first algorithm performs some correction of outdated routing table entries with a low cost of correction traffic. The second algorithm exploits the nature of the broadcasts to extensively update erroneous routing information at the cost of higher correction traffic. The algorithms are validated and evaluated in our stochastic distributed-algorithms simulator.

[1]This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-33234 PEPITO project and partially by the Vinnova PPC project in Sweden

6.1 Introduction

The need for making effective use of the huge amount of computing resources attached to large scale networks, such as the Internet, has established a new field within the distributed computing area, namely, Peer-to-Peer (P2P) computing.

The current trend in this new field builds on the idea of distributed hash tables (DHT) that provide infrastructures for scalable P2P systems [12, 13, 2, 7]. The infrastructure is a logical network, called an *overlay network*, within which key/value pairs are stored. The main operation offered by DHT-based overlay networks is the lookup operation, that is finding a value associated with a given key. However, the lookup operation itself is not enough to perform arbitrary queries such as context dependent searches. Furthermore, it is difficult, in large DHT systems, to collect statistical information about the system, such as the overall system usage for billing purposes.

In this paper we present two broadcast algorithms for the distributed k-ary system ($\mathcal{DKS}$) [2] that can be used to solve the above mentioned problems. The choice of $\mathcal{DKS}$ is motivated by two reasons. First, the $\mathcal{DKS}$ systems, in contrast to all other systems [9, 14, 6], avoid the use of periodic stabilization protocols for maintaining routing information. Instead, a novel technique called *correction-on-use* serves to correct outdated routing information on-the-fly. Network bandwidth is thus saved during periods when activity is low. Second, $\mathcal{DKS}$ provides the ability to tune the ratio between routing table size and maximum lookup length. E.g. a system can be configured with large routing tables and a low maximum lookup length, consequently, making broadcasts faster.

6.1.1 Contribution

The work in [4] paved the way for doing broadcasts on top of structured P2P networks such as the Chord system [12, 11]. However, the algorithm in [4] fails to cover all nodes when the routing information is inconsistent, which is the natural case in dynamic P2P networks as a consequence of nodes joining or leaving.

In this paper we present two broadcast algorithms that deal with routing table inconsistencies. The new broadcast algorithms guarantee 100% coverage even in the presence of frequent network changes and outdated routing information. Furthermore, unlike other similar attempts[8], nodes do not receive any redundant messages.

Furthermore, we extend the *DKS* philosophy of avoiding the use of periodic stabilization. The second broadcast algorithm exploits the nature of a broadcast to effectively correct outdated routing information at the cost of extra local computation and network traffic.

The proposed algorithms can be used to perform multicast. Each multicast group is then represented by an instance of *DKS* within which the proposed broadcast algorithms can be used to disseminate multicast messages.

6.1.2 Related work

Our work can be classified as extending DHTs to support arbitrary-searches. From that perspective, the research in complex queries shares the same goal. In [5] the idea is to construct search indices that enable the performance of database-like queries. This approach differs from ours in that we do not add extra indexing to the DHT. The analysis of the cost of construction, maintenance, and performing database-like join operations is not available at the time of writing of this paper.

Since broadcast is a special case of multicast, a multicast solution developed for a DHT such as [10, 8, 3] can provide broadcast functionality. Nevertheless, a multicast solution would require the additional maintenance of a multicast group which, in the case of broadcast, is a large group containing all the nodes in the network. For example [3] uses one rendezvous node per group, that disseminates messages with the help of potential non-members called forwarders by using multicast trees. In [8], a bootstrap node stores information about a group, in which it is not necessarily a member. Additionally, there is a inherent redundancy of messages when the coordinate space is not perfectly partitioned. In our approach, these two drawbacks are avoided.

6.1.3 Outline

The remaining of this paper is organized as follows. In section 6.2 we give an overview of $\mathcal{DKS}$ systems. Section 6.3 provides informal and formal descriptions of the proposed algorithms. Section 6.4 is devoted to the validation and the evaluation of the two algorithms. Finally, section 6.5 concludes.

6.2 $\mathcal{DKS}$ overview

In the following sub-sections we present the $\mathcal{DKS}$ systems. We focus on its two main contributions, a generalization to tune the lookup length, and a correction-on-use technique used to avoid periodic stabilization protocols for maintaining routing information.

6.2.1 Structure of the $\mathcal{DKS}$

$\mathcal{DKS}$ systems are configured with the parameters, N, and $k \geq 2$, such that the lookup length is guaranteed to take at most $log_k(N)$ hops for a network of maximum size N. With k defined, the maximum number of nodes that can be simultaneously in a $\mathcal{DKS}$ network is chosen to be $N = k^L$ for some large L. Every node knows k and N, and can therefore compute L.

Once N has been defined, all nodes and keys in the system are deterministically mapped onto the identifier space, $\mathcal{I} = \{0, 1, .., N-1\}$, by using a globally known hash function, H. The identifier space is a circular space modulo N.

Each $key/value$ pair is physically stored at the first node encountered in the ring, moving in clockwise direction, starting at $H(key)$.

We shall use the notation $a \oplus b$ for $(a + b)$ modulo N for all $a, b \in \mathcal{I}$. The whole identifier space can be represented by an interval of the form $[x, x[$ or $]x, x]$ for an arbitrary $x \in \mathcal{I}$. For any $x \in \mathcal{I}$, we note that $[x, x] = \{x\}$ and $]x, x[= \mathcal{I} \backslash \{x\}$.

6.2.2 Routing tables

Each node, in addition to storing key/value pairs, maintains a routing table. The routing table consists of $log_k(N)$ *levels*. Let $\mathcal{L} = \{1, 2, .., log_k(N)\}$ be the set of levels.

At each level, $l \in \mathcal{L}$, a node n has a view of the identifier space defined as:

$$V_l = [n, n \oplus \frac{N}{k^{l-1}}[$$

This means that for level one, the view consists of the whole identifier space, and at any other level $l > 1$, one k:th of V_{l-1} is considered.

At any level $l \in \mathcal{L}$, the view is partitioned into k equally-sized intervals denoted I_i^l for $0 \leq i \leq k-1$. At a node n, I_i^l is defined as:

$$I_i^l = [n \oplus i\frac{N}{k^l}, n \oplus (i+1)\frac{N}{k^l}[, \;\; i \in \{0, 1, .., k-1\}, \; l \in \mathcal{L}$$

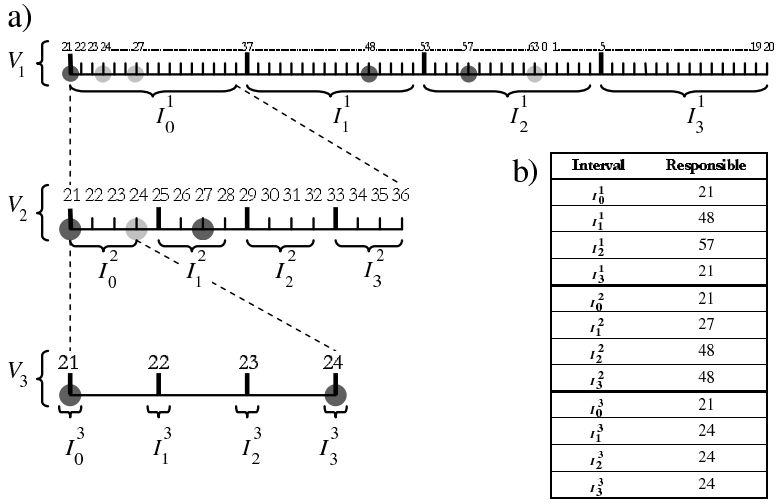
Each node, n, maintains a responsible node for every interval in its routing table. For any level, $l \in \mathcal{L}$ the responsible for interval I_0^l is always n itself. [2] For all other intervals $j \in \{1, 2, .., k-1\}$, the responsible for interval I_j^l is chosen to be the first node encountered, moving in clockwise direction, starting at the beginning of the interval. We shall use the function $R(I)$ to denote the id of the responsible node for interval I.

In addition to storing a routing table, each node, n, maintains a *predecessor* pointer, that is the first node encountered, moving in counter-clockwise direction, starting at n.

An important property of a $\mathcal{DKS}$ system is that when a node n joins or leaves the system, only n's predecessor and successor are explicitly updated in a fault-free context. The rest of the nodes in the system will find out about n existence or departure by the correction-on-use technique described in section 6.2.4.

Figure 6.1 shows an example of a $\mathcal{DKS}$ network from one node's point of view. Note that in figure 6.1 we have mapped the modulo N circle onto a line from node 21's view.

[2] The responsible node's identifier and network address is stored such that communication can be established with it.



Interval	Responsible
i_0^1	21
i_1^1	48
i_2^1	57
i_3^1	21
i_0^2	21
i_1^2	27
i_2^2	48
i_3^2	48
i_0^3	21
i_1^3	24
i_2^3	24
i_3^3	24

Figure 6.1: a) A *DKS* network with $k = 4$ and $N = 64$, with the nodes 21, 24, 27, 48, 57, and 63 present. The figure shows node 21's views, V_1, V_2 and V_3, and how each view is partitioned into $k = 4$ equally sized intervals. The dark nodes represent the responsible nodes from node 21's view. b) Node 21's routing table showing each interval and its responsible node.

6.2.3 Lookups

To initiate a search for a key identifier id at a node n the distributed lookup is performed as follows. If id is between n's predecessor and n, the key/value pair is stored at n itself and can be resolved locally at n.

Otherwise, n searches its routing table at level $l = 1$, for an interval I_i^l in V_l such that $id \in I_i^l$, for $0 \leq i \leq k-1$. The lookup request is thereafter forwarded to the responsible node for interval I_i^l with the parameters l and i piggybacked.

A node n' upon receipt of the forwarded request checks if the key identifier id is between its predecessor and itself. If so, then n' returns the value associated with id to n. Otherwise, it searches its routing table at level $l+1$ for an interval that contains id. Then a lookup request is forwarded to the responsible for that interval. The current level and interval are again pig-

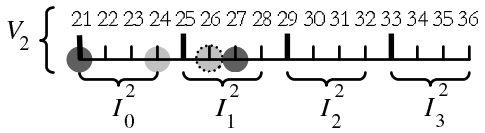


Figure 6.2: A node with identifier 26 joins the network depicted in figure 6.1. As node 21 is not the predecessor of node 26, it will not immediately be informed about node 26's existence. Hence it will continue to, erroneously, consider node 27 as responsible for I_1^2. If node 21 sends a lookup message to node 27, node 21 will find out about node 26's existence by correction-on-use. Alternatively, node 21 will become aware of node 26's existence if node 26 sends a lookup message to node 21.

gybacked in the forwarded request. This process repeats until the node storing the key id is found, in which case the value associated with id is recursively sent back to n.

6.2.4 Correction-on-use

In a *DKS* network, routing information can become outdated as a result of joining or leaving nodes. Figure 6.2 shows how routing entries become outdated as a result of a join operation. The outdated routing entries are corrected only when they are used. As long as the ratio of lookups to joins, leaves, and failures is high, the routing information are eventually corrected. This is the essential assumption in *DKS*, which is validated in [2].

Correction-on-use is based on two ideas. The first idea is to embed the level, l, and the interval, i, parameters with every lookup or insertion message. A node n receiving a lookup or insertion message from a node n' can then calculate the start of the interval, I_i^l at node n', for which n is responsible according to the node n'. If n's predecessor is in the interval $[n' \oplus i\frac{N}{k^l}, n[$, then node n notifies the node n' of the existence of n's predecessor. Node n' can then update its erroneous routing entry.

The second idea is that a message sent by a node p to another node n is an indication that p exists and is thus part of the *DKS* network. Hence,

node n examines all of its intervals to determine if p should be responsible for any of the intervals, in which case routing information is updated.

6.3 The broadcast algorithms

6.3.1 Desired properties

The broadcast algorithms should have the following desirable properties:

- *Coverage*. All the nodes present in the system, at the time a broadcast operation starts, receive the broadcast message as long as they remain in the system.
- *Redundancy*. Any node that receives a broadcast message receives it once, disregarding messages sent trough erroneous pointers as they will trigger correction-on-use.
- *Correction of routing information*. The broadcast algorithms should contribute to the correction of outdated routing information.

6.3.2 Informal description

The basic principle of the two broadcast algorithms is as follows. A node starting the broadcast iterates through all levels in $\mathcal{L}$ starting at the first level. At each level, the node moves in counter-clockwise direction through all of its intervals, broadcasting a message to each responsible node. Each broadcast message, sent by a node n, carries with it the parameters *l*, *i* and *limit*. The message's purpose is twofold. First, it delivers the intended data to the receiving node. Second, it serves as a request to a receiving node to cover all nodes in the interval $]n \oplus i * \frac{N}{k^l}, limit[$. Each node, receiving the broadcast message, repeats the mentioned process, but makes certain not to broadcast to a node beyond the $limit$ given to it.

To illustrate the principle of the proposed algorithms, a fully populated $\mathcal{DKS}$ network with $N = 16$ and $k = 4$ is considered. A broadcast initiated at node 0 proceeds level by level. Beginning at level one, node 0 sends a broadcast message to node 12 giving it responsibility to cover the interval

$]12, 0[$. Thereafter it repeats the same procedure for I_2^1 giving node 8 responsibility for the interval $]8, 12[$. After sending a broadcast to interval I_1^1 the algorithm moves to level two, repeating the process for the intervals I_3^2, I_2^2, and I_1^2. Each of the responsible nodes receiving the message from node 0 will repeat a similar process except they will not go beyond the limits assigned to them. For example node 12 will not send, at level one, to its intervals I_3^1, I_2^1, I_1^1 as they are beyond the given limit 0. Instead, it will move to level two, sending a broadcast to the nodes responsible for intervals I_3^2, I_2^2, and I_1^2.

6.3.3 Formal description

In both algorithms we assume a distributed system modeled by a set of nodes communicating by message passing through a communication network that is: (i) Connected, (ii) Asynchronous, (iii) Reliable, and (iv) providing FIFO communication.

A distributed algorithm running on a node of the system is described using rules of the form:

$$R \;::\; \frac{\textbf{receive}(\textit{Sender}, \textit{Receiver}, \text{MESSAGE}(arg_1, .., arg_n))}{\text{Action}}$$

The rule R describes the event of receiving a message MESSAGE at the *Receiver* node and the action taken to handle that event. A *Sender* of a message executes the statement **send**(*Sender*, *Receiver*, MESSAGE($arg_1, .., arg_n$)) to send a message to *Receiver*.

The first algorithm The first broadcast algorithm is given by figure 6.3. Rule $R1_1$ describes the reaction of a $\mathcal{DKS}$ node upon receipt of a BCASTREQUEST($data$) from the application layer. Rule $R1_1$ triggers rule $R2_1$ with the parameters $l = 1$, $k = 0$, and $limit$ set to the initiating node's id, giving the initiating node responsibility to cover all nodes in the system.

When a broadcast is initiated, the algorithm proceeds level by level. At each level, the node iterates all intervals from $k - 1$ down to 1 and sends a message to the responsible node for each of the intervals. To avoid sending duplicate messages to nodes responsible for several intervals, a message

is only sent when the id of the responsible node is not beyond the end of the interval checked for.

Due to outdated routing table entries some intervals might not seem to have any nodes even though they are populated. The responsibility of covering those intervals is delegated to the next interval in the iteration. This is done by not changing the $limit$ parameter when an interval seem to be unpopulated.

Improving the correction of the routing information In order to improve the correction of outdated routing information, we extend Algorithm 1 with *self-correction*. The idea consists of extending the responsibility assigned to a node n', by a node n, to cover other preceding intervals that n' is responsible for according to n. Hence, if other nodes exist in n''s preceding intervals, which n is not aware of, n' will trigger correction-on-use and the routing information will be corrected at n. The subroutine FINDLOWEST is used for this purpose.

The second broadcast algorithm is the same as the first algorithm, except that rule $R2_1$ is replaced by rule $R2_2$ as shown in figure 6.4.

$R1_1$ :: **receive**(u, n, **BCASTREQUEST**($data$))
send($n : n$: **BCAST**($data, 1, 0, n$)

$R2_1$ :: **receive**(n', n, **BCAST**($data, l, i, limit$))
if $n' \oplus i\frac{N}{k^l} \in]predecessor, n]$ **then**
%% Deliver the message to the application layer
for $\lambda := 1$ **to** $log_k(N)$ **do**
for $\tau := k-1$ **downto** 1 **do**
if $R(I_\tau^\lambda) \in]n, limit[$ **then**
send(n, R(I_τ^λ), **BCAST**($data, \lambda, \tau, limit$))
$limit := n \oplus \tau\frac{N}{k^\lambda}$
fi
od
od
else
send(n, n', **BADPOINTER**(**BCAST**($data, l, i, limit$), $predecessor$))
fi

$R3_1$ :: **receive**(n', n, **BADPOINTER**(**BCAST**($data, l, i, limit$), $candidate$)
for $\lambda := 1$ **to** $log_k(N)$ **do**
for $\tau := k-1$ **downto** 1 **do**
if $n \oplus \tau\frac{N}{k^\lambda} \in]n, candidate]$ **and** $R(I_\tau^\lambda) \in]candidate, n]$ **then**
$R(I_\tau^\lambda) = candidate$
fi
od
od
send(n, $candidate$, **BCAST**($data, l, i, limit$))

Figure 6.3: Algorithm 1

```
R2_2 :: receive(n',n,BCAST(data, l, i, limit))
        if n' ⊕ i N/k^l ∈ ]predecessor, n] then
          %% Deliver the message to the application layer
          for λ := 1 to log_k(N) do
              for τ := k − 1 downto 1 do
                  if R(I_τ^λ) ∈ ]n, limit[ then
                    (i', l') := FINDLOWEST(n, R(I_τ^λ))
                    send(n, R(I_τ^λ), BCAST(data, i', l', limit))
                    limit := n ⊕ i' N/k^l'
                  fi
              od
          od
        else
          send(n,n',BADPOINTER(BCAST(data, l, i, limit), predecessor))
        fi

Subroutine :: FINDLOWEST(n', r)
              for λ := 1 to log_k(N) do
                  for τ := k − 1 downto 1 do
                      if R(I_τ^λ) = r then
                        (l', i') := (λ, τ)
                      fi
                  od
              od
              return (l', i')
```

Figure 6.4: Algorithm 2. The rules $R1_2$ are $R3_2$ are the same as rules $R1_1$ and $R3_1$ in figure 6.3.

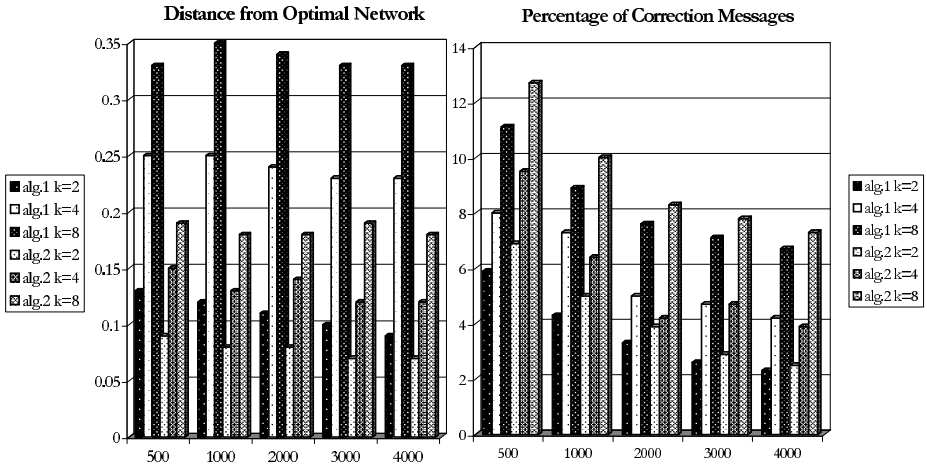


Figure 6.5: Experiment 1: a) Shows the distance from the optimal network b) Shows the percentage of correction messages

6.4 Simulation Results

In this section we show preliminary simulation results for the broadcast algorithms. We use the following four metrics for evaluation. *Coverage*, *Redundancy*, *Correction Cost* and *Distance from Optimal Network*.

The *Coverage* and the *Redundancy* metrics are calculated by taking a snapshot of all the nodes present in the overlay network at the initiation time of each broadcast. The simulator then maintains a counter for each node receiving the broadcast message. The coverage is calculated by counting the percentage of nodes in the snapshot that received the broadcast message by the end of the simulation. The redundancy is computed by counting the number of covered nodes that received the message more than once. *Correction Cost* is defined as the percentage of messages used for correction of routing entries out of the total number of messages generated by a broadcast. *Distance from Optimal Network* is the ratio of the number of erroneous routing entries in all nodes to the total number of routing entries in the system. When this ratio is equal to 0 the routing information is said to be optimal.

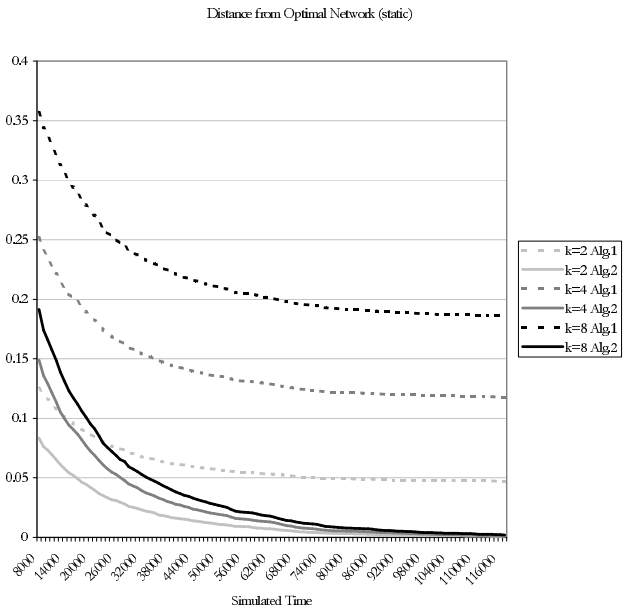


Figure 6.6: Experiment 2: Shows the convergence to a maximally optimal network while performing broadcasts with algorithm 1, 2.

The experiments were conducted on a stochastic discrete distributed-algorithms simulator developed by our team and using the Mozart [1] programming platform. In this paper we present the results of two experiments. The purpose of the first experiment was to test the system in a dynamic setting and evaluate the performance of our algorithms using the mentioned metrics. The second experiment focused on the convergence towards a minimal distance from the optimal network.

Experiment 1. A *DKS* network of size $N = 2^{12}$ was created. The population of nodes in the system was considered a variable P that took values from $\{500, 1000, 2000, 3000, 4000\}$. For each value of P, we proceeded in two steps. First, we initialized the system with 10% of P. Second, 90% of P nodes joined while P broadcasts were initiated. The experiment was repeated for the values of $k = 2, 4, 8$. That is, with a high probability, each node initiated one broadcast while the overlay network was growing.

Experiment 2. A $\mathcal{DKS}$ network of size $N = 2^{12}$ was created. The system was initialized with 1500 nodes. Thereafter an arbitrary number of broadcasts were initiated. The experiment was repeated for the values of $k = 2, 4, 8$.

Results. In all our experiments, the *Coverage* and *Redundancy* were 100% and 0 respectively as expected from the design.

Distance from Optimal Network. Two observations can be made from Figure 6.5 a). First, for all values of k, Algorithm 2 corrects routing information more effectively than Algorithm 1. Second, the final distance from the optimal network is mainly affected by the search arity, k, and not the population size. From Figure 6.6 we can see that algorithm 2, in contrast to algorithm 1, effectively converges to the optimal network for all search arities.

Correction Cost. As shown in Figure 6.5 b), the correction cost is in general higher for Algorithm 2. This was expected as the correction requires some additional overhead.

6.5 Conclusion

In this paper we presented two algorithms for broadcasting on structured peer-to-peer networks. Our work was motivated by two reasons. First, the need to extend distributed hash tables to perform arbitrary queries and retrieval of global statistical information about the DHTs. Second, to provide robust algorithms that can be used for multicasting within groups in the context of $\mathcal{DKS}$ overlay networks. Each group is formed by creating a specific $\mathcal{DKS}$ instance for it.

The proposed algorithms use the $\mathcal{DKS}$ philosophy of avoiding periodic stabilization to maintain routing information. The second algorithm extends the philosophy by heavily correcting incorrect routing information.

In addition, the broadcast algorithms provide full coverage even if nodes have erroneous routing information. Furthermore, each broadcast message is received once even if new nodes join while the broadcasts are taking place.

The proposed algorithms have been validated and evaluated in a dynamic network through simulations and the obtained results confirm our

expectations. More precisely, Algorithm 1 gives less correction overhead and larger distance from the optimal network compared to Algorithm 2.

Bibliography

[1] Mozart Consortium, 2003. http://www.mozart-oz.org.

[2] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. DKS(N; k; f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *The 3rd International Workshop On Global and Peer-To-Peer Computing on Large Scale Distributed Systems (CCGRID 2003)*, Tokyo, Japan, May 2003. http://www.ccgrid.org/ccgrid2003.

[3] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[4] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient Broadcast in Structured P2P Netwoks. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[5] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[6] Petar Maymounkov and David Mazires. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[7] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content Addressable Network. In *Proceedings of the ACM SIGCOMM '01 Conference*, Berkeley, CA, August 2001.

[8] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Third International Workshop on Networked Group Communication (NGC '01)*, 2001. http://www-mice.cs.ucl.ac.uk/ngc2001/.

[9] 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)*, 329-350 2001.

[10] Ion Stoica, Dan Adkins, Sylvia Ratnasamy, Scott Shenker, Sonesh Surana, and Shelley Zhuang. Internet Indirection Infrastructure. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[11] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, August 2001.

[12] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, 2003.

[13] Ben Y. Zhao, Ling Huang, Sean C. Rhea, Jeremy Stribling, Anthony D Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE J-SAC*, 22(1):41–53, January 2004.

[14] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2000.

Chapter 7

A Component-based P2P Simulation Environment

7.1 Motivation

Due to the complex nature of P2P algorithms, i.e. the large number of nodes and the many possible scenarios of peer interactions under different operating conditions, formal reasoning about such algorithms is challenging. One of the goals of the PEPITO project [2], in which this work is done, is to start tackling this issue. However, in our work on studying and designing P2P algorithms, we needed a rather practical tool that can quickly show the validity of new ideas and that is capable of measuring the performance of the various algorithms in different operating conditions.

To achieve our goal, we developed a simulation environment for the validation and evaluation of P2P algorithms. We aimed at providing an environment that allows us to focus on interesting properties of the P2P algorithms currently under study/design. Meanwhile we wanted to allow enough expressive power to accommodate for upcoming algorithms.

The design of our simulation environment was based on two previous works in the field of simulation. The first is the Chord simulator [5] which is a discrete-event simulator designed for simulating the Chord system. The second, is the experience acquired by our colleagues at the DSL lab at SICS during their work on component-based large-scale simulation in the

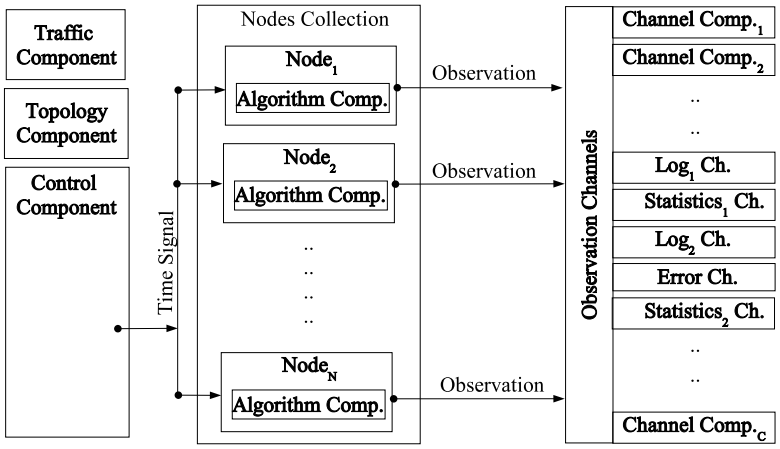


Figure 7.1: Simulator Architechture

iCities project [3] and which depends on favorable features of the Mozart programming system [1]. While the Chord simulator provided us with a good starting point, we needed to have an environment in which we can compare several algorithms easily and that is where we needed to make use of a component-based architecture according to the methodology presented in [4].

This chapter aims at describing the simulator from a software engineering point of view and not as an explanation of a certain simulation model. Instead, we view the architecture presented as generic architecture capable of realizing different simulation models. The rest of this chapter is dedicated to the description of the simulation environment architecture.

7.2 Architecture

7.2.1 Overview

The component-based simulation architecture suggested in [4] assumes the availability of a library of different component types from which an experiment is composed. Following this strategy, we have designed a number of components that suit our requirements for simulating P2P algorithms. To perform a certain experiment, the designer of the experiment should choose a controller component, a topology component, a traffic component, an algorithm component, and a number of observation channel components. The composition of the simulation environment, thus, is done, by configuring a parameters module that specifies the different component types and their parameters in addition to global simulation parameters.

7.2.2 The Traffic Component

The traffic component is responsible for generating traffic, i.e. different events that can happen during the simulation such as new peers joining a network, existing peers leaving the network gracefully or due to a failure, a peer inserting, looking up or deleting a key-value pair.

So far, we have worked with two types of traffic components: The first type is adopting the traffic model of the Chord simulator where: (i) There are five types of events: join, leave, fail, insert, lookup, (ii) All the traffic is generated before the simulation begins, i.e. with a predetermined maximum simulation time, and (iii) Lottery scheduling is used for generating the different events. The second type of traffic components we use for working with predefined topologies, especially when we want to define non-probabilistic traffic pattern, e.g. letting every node search for each possible item. In that case, we do not need probabilistic scheduling and we do not require a predefined maximum simulation time.

New traffic component types could be authored and plugged easily in case one would want to adopt a different scheduling policy like using, for instance, Poisson processes for node arrivals and exponential distribution for node failure.

Irrespective of the traffic component used, events that are generated are saved in time queues. The structure of the time queue is defined as follows:

Definition 7.1 *A time queue is defined as the mapping* $TimeQueue : Time \rightarrow 2^E$*, where* $Time \in Z^+$ *is a simulation time step and* E *is the set of all possible events. We write* $TimeQueue_p(t)$ *to denote all the events that are scheduled at time step* t *at process* p*.*

7.2.3 The Topology Component

The topology component is an optional component. In some simulations, we are interested to start with a set of nodes connected in a predefined topology. In other cases, we want the algorithms under evaluation or design to construct the network topology themselves. Therefore, in most of the simulations related to studying performance measures of core DHT functionalities, we do not use the topology component for constructing the overlay network.

An example simulation where we need to use a topology component is validation experiments. Mainly, to check that a certain property holds, given that the topology is optimally structured, e.g., checking that a broadcast algorithm will cover all the nodes of a correctly structured Chord or DKS system. In that case we use a topology component to generate that optimal structure. Another typical case is that after letting a large number of nodes join a P2P network, we need to validate that they were able to self-organize as an optimally structured network.

7.2.4 The Controller Component

A controller component serves as the main program of the simulation environment. Different controller components types are used depending on what other types of components are needed and how they are initialized. The behavior of an example controller component is given in the algorithm below.

As shown in algorithm 1, a controller component, starts by initializing other components such as the traffic component, the topology component,

Algorithm 1 Controller Behavior

```
1: INITTHETRAFFICCOMPONENT()
2: INITTHETOPOLOGYCOMPONENT() //Optional
3: INITTHEOBSERVATIONCHANNELS()
4: for t = 1 to MaxTime
5:    forall p : p ∈ P ∪ {Controller}
6:       p.Signal(t)
7:    End for
8: End for
9: CONCLUDETHEOBSERVATIONS() //Optional
```

and the observation channels components [lines $1-3$]. The time queue of the controller is usually used to save the output of the traffic component. The second task of the controller is to advance the simulation time. In each time step, the controller as well as the set $\mathcal{P}$ of all nodes being simulated are signaled. Consequently, the set of events $TimeQueue_{\mathcal{P}\cup\{Controller\}}(t)$ are executed [lines $4-8$]. Finally, the controller concludes the observations, i.e. computes any statistical or validation information that are needed [line 9].

7.2.5 The Node Abstraction

The core of the simulator lies in the behavior of the collection of nodes. As illustrated in figure 7.1, the nodes interact with two main entities of the simulator, the controller and the observation channels. The controller advances the time and signals all the nodes with the current time step. The nodes use the observation channels to log different kinds of information about the simulation state.

Each node hosts an algorithm component. Figure 7.2 shows a more detailed view of the architecture of each individual node. The node is abstracted in three different layers; a *TimedNode* layer, a *DistributedNode* layer and an *AlgorithmNode* layer.

The *TimedNode* layer is concerned with simulation time. It is the layer at which the time queue of a node is dealt with. Its interface provides two functions:

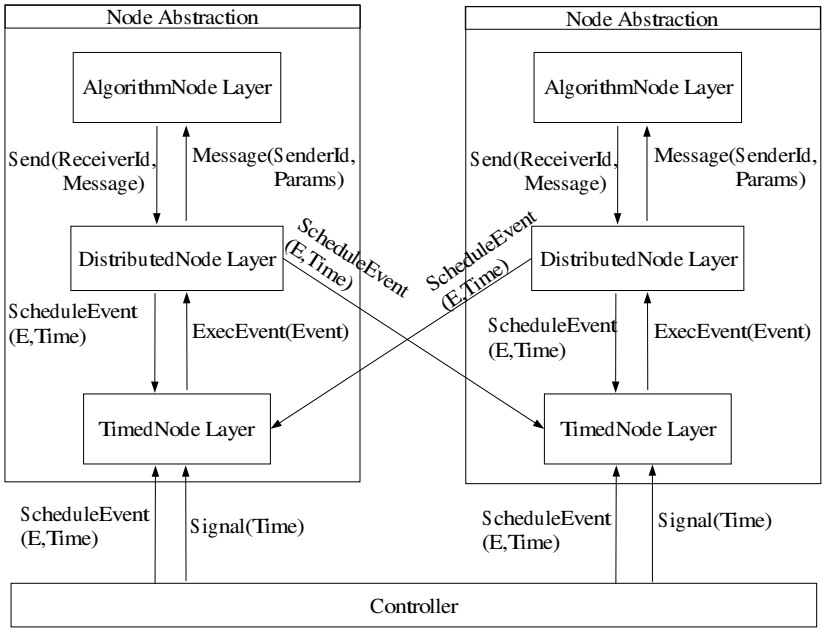


Figure 7.2: The Node Abstraction.

- $ScheduleEvent(E, Time)$
- $Signal(Time)$

A call to $ScheduleEvent(E, Time)$ results in the insertion of a new event in the time queue at the specified time and a call to $Signal(Time)$, leads to the execution of all the events scheduled at the specified time. This layer is a basic layer for any entity that depends on simulation time and at this level, there is no awareness of a network or a distributed algorithm.

The *DistributedNode* layer is concerned with providing the abstraction of a node that uses a network to send and receive messages and hides the notion of simulation time and its interface provides two functions:

- $Send(ReceiverId, Message)$

- $ExecEvent(Event)$

The $Send$ function is used by the $AlgorithmNode$ layer to send messages and the $ExecEvent$ function is used by the $TimedNode$ layer when the time comes for the execution of a certain event. Sending messages is modeled by the scheduling of an event at the receiving node. That is, if a sender node sends a message at time t, an event is scheduled at time $t + AverageTransmissionDelay()$ at the receiving node where *AverageTransmissionDelay()* is a function that provides an average transmission delay using a Poisson distribution. The *DistributedNode* layer can also schedule an event at the local node which is typically used for modeling timeouts. This layer is also the place for modeling any communication-related issues like message loss, netwrok partitioning, firewalls, etc.

The *AlgorithmNode* layer hosts the algorithm component. The receiving of a message is implemented via a polymorphic interface, i.e. instead of providing a message called $Receive$, a message of the form *Message(SenderId, Params)* is used as a polymorphic method call that should be implemented by any algorithm component in the *AlgorithmNode* layer. Since this layer uses the underlying *DistributedNode* layer to interact with other nodes by sending and receiving messages, a real communication layer can replace the *DistributedNode* layer leading to the reuse of the simulated algorithm component as a working piece of software.

7.2.6 The Observation Channels Components

The observation channels offer a very flexible means for gathering information about what happens during simulation. They are configured prior to running a simulation via a specification record described by the following grammar:

```
ObservationChannels ::= channels({<ChannelName>(
                                   output:<OutputTarget>
                                   state:'on' | 'off'
                                   )
                                     ...
                                  }
```

```
                                  )
<ChannelName>  ::= <atom>
<OutputTarget> ::= console
                 | file(<FileName>)
                 | module(<ModuleName>)
<FileName> ::= <string>
<ModuleName> ::= <string>
```

An example of such configuration is as follows:

```
Channels = channels(
            protocolErr(output:console state:on)
            commTrace(output:module(
             "TraceFilter.ozf")
                      state:on)
            stat(output:module("Stat.ozf")
                 state:off)
            sys(output:console state:on)
            debug(output:console state:on)
            join(output:console state:off)
            bstat(output:module("BcastStat.ozf")
                  state:on)
            convergence(output:file("conv.txt")
                        state:on)
          )
```

As shown in the example each observation channel is responsible for tracking a different kind of information. Each channel has an output target. This could be the console, simple dump in a file or to be processed by a module. A module that is to be used as an output target for a channel must implement the interface for observation channels which includes the two methods $write()$ for sending data to the channel and $close()$ to indicate the end of incoming data and perform any finalizing steps if any. Depending on what observations are interesting during a certain experiment, channels could be turned on and off.

Bibliography

[1] Mozart Consortium, 2003. http://www.mozart-oz.org.

[2] Peer-To-Peer Implementation and Theory (PEPITO), EU Project IST-2001-33234 , 2003. http://www.sics.se/pepito.

[3] The iCities project, EU Project IST - 1999 - 11337, 2003. http://icities.csd.uoc.gr/.

[4] M. Rafea, F. Holmgren, K. Popov, S. Haridi, S. Lelis, P. Kavassalis, and J. Sairaamesh. Application architecture of the internet simulation model: Web word of mouth (WoM). In *MS 2002: IASTED International Conference on Modelling and Simulation*, May13–15 2002.

[5] The Chord Project Home Page , 2003. http://www.pdos.lcs.mit.edu/chord/.

Part II

Analyses

Chapter 8

Physics-inspired Performance Evaluation of DHTs.

Physics-inspired Performance Evaluation of DHTs.

Sameh El-Ansary[1], Erik Aurell[1,2] and Seif Haridi[1,3]

[1] Distributed Systems Laboratory
SICS Swedish Institute of Computer Science
[2]Department of Physics, KTH-Royal Institute of Technology
[3]IMIT, KTH-Royal Institute of Technology
{sameh,eaurell,seif}@sics.se

Abstract

In the majority of structured peer-to-peer overlay networks a graph with a desirable topology is constructed. In most cases, the graph is maintained by a periodic activity performed by each node in the graph to preserve the desirable structure in face of the continuous change of the set of nodes. The interaction of the autonomous periodic activities of the nodes renders the performance analysis of such systems complex and simulation of scales of interest can be prohibitive. Physicists, however, are accustomed to dealing with scale by characterizing a system using intensive variables, i.e. variables that are size independent. The approach has proved its usefulness when applied to satisfiability theory. This work is the first attempt to apply it in the area of distributed systems. The contribution of this paper is two-fold. First, we describe a methodology to be used for analyzing the performance of large scale distributed systems. Second, we show how we applied the methodology to find two intensive variables that describe the characteristic behavior of the Chord overlay network, the variables are: 1) *The density of nodes in the identifier space and* 2) *The ratio of the magnitude of perturbation of the network (joins/failures) to the magnitude of periodic stabilization of the network.*

Keywords: DHT performance, Structured Overlay networks, Data Collapse, Complex Systems

8.1 Introduction

A number of structured P2P overlays [16, 14, 17, 8, 2], aka Distributed Hash Tables (DHTs) were recently suggested. In most such systems, nodes self-organize in a graph with a diameter and outgoing arity of nodes that are both of a logarithmic order of the number of nodes. Some systems like [7, 13] can even provide a logarithmic order diameter with a constant order routing table. To maintain the graph in an optimal state despite change of membership (joins/failures), each node needs to follow some self-repair policy to keep its routing table (the outgoing edges) up-to-date.

Our general observation on the literature of structured overlay networks is that the self-organization aspect is dominant compared to self-repair. The typical case for a paper introducing a DHT system would be to show the structure of the routing table, how the overlay graph will be constructed, and protocols for joins and leaves. When it comes to self-repair, the discussion gets relatively superficial. We find arguments on the level of: "*Periodic maintenance of routing tables will ensure its correctness*", "*Data items have to be republished periodically by the upper layer*", etc.. In the best cases, a simulation is given showing that under particular values of maintenance rates, the network can operate.

We attribute the shortage of work on performance analysis of self-repair properties to the following two factors: $i)$ The novelty of such systems and the requirement to establish the new concepts first before deeper analysis is performed. $ii)$ Once we have a system deploying a self-repair policy, an analytical model that describes the behavior of the system can range in degree of difficultly from not-so-clear to very-complex-to-analyze. Therefore, we see the compelling need for more studies whether analytical or simulation-based that can tell us whether those novel techniques are really useful or over-hyped.

In this paper we try a novel approach for analyzing the performance of one of the most well-established overlay networks, namely the Chord system [16, 15]. In the following section we motivate our deployment of a physics inspired approach and we state our methodology. After that, we explain our assumptions in implementing the Chord protocols. We follow that by the core sections of the paper namely the application of our

methodology to find intensive variables. Finally, we conclude the work in the last section.

8.2 The physics-inspired approach

8.2.1 Motivation.

Having observed that analytical models are not always trivial to formulate given a system applying a given self-repair policy, we thought that using simulation seemed to be the practical tool for analyzing such systems. However, we figured out that scales of interest could be prohibitive for simulation purposes. At this point, a physics-style approach started to be of interest since physicists are accustomed to reasoning about large natural systems.

8.2.2 How do physicists deal with scale?

Physics was the first science to encounter problems of this sort. The number N of molecules in a macroscopic body, say a liter of water, is about 10^{27}. On the microscopic level, all substances are made of atoms and molecules of the same basic type – different number of electrons, protons and neutrons – yet large lumps of the same kind of atoms or molecules make up substances with distinct qualities which we can perceive. Those can be density, pressure (of a gas at given density and temperature), viscosity (of a liquid), conductivity (of a metal), hardness (of a solid), how sound and light do or do not propagate, if the material is magnetic, and so on.

The first level of analysis in a physical system of many components, is to try to separate *intensive* and *extensive* variables. Extensive variables are those that eventually become proportional to the size of the system, such as total energy. Intensive variables, such as density, temperature and pressure, on the other hand, become independent of system size. A description in terms of intensive variables only is a great step forward, as it holds regardless of the size of the system, if sufficiently large.

Further steps in a physics-style analysis may include identifying phases, in each of which all intensive variables vary smoothly, and where the char-

acteristics of the system remain quantitatively the same.

8.2.3 Was the approach useful in the computer science arena?

A physics-style approach was carried over to satisfiability theory more than ten years ago. KSAT is the problem to determine if a conjunction of M clauses, each one a disjunction of K literals out of N variables can be satisfied. Both M and N are extensive variables, while $\alpha = M/N$, the average number of clauses per variable, is an intensive variable. For large N, instances of KSAT fall into either the SAT or the UNSAT phase depending on whether α is larger or smaller than a threshold $\alpha_c(K)$ [10, 3]. The order of the phase transition, a statistical mechanics concept roughly describing how abrupt the transition is, has been shown to be closely related to the average computational complexity of large instances of KSAT with given values of K and α [12, 11]. Recent advances include the introduction of techniques borrowed from the physics of disordered systems, leading to an important new class of algorithms, currently by far the best of large and hard SAT problems [9]. Without question, statistical mechanics have been proven to be very useful on very challenging problems in theoretical computer science, and it can be hoped that this will also be the case in the analysis and design of distributed systems.

8.2.4 "Data collapse": the tool for observing intensive variables

Let us assume that we have a system that we evaluate using a function $m(S, P)$ where m is some performance metric, S is the size of the system and P is a set of system parameters. A description of the system in terms of all the possible values of m under all values of S and P is not very transparent, and can be prohibitive to enumerate.

If all the parameters in P are essential, we cannot do better. If on the other hand they only influence m through some combination or functional relationship, it is preferable to use instead $m(g(\psi))$, where $\psi \subset \{S\} \cup P$, and g is some function. We will assume g smooth, as otherwise the practical utility is generally small. The main advantage is that this encodes a definite understanding of what really influences system behavior. Ad-

ditionally, the data can be presented in a systematic and much compact manner.

A discussion of the process of obtaining ψ and g in general is outside this presentation. Suffice it so say that if the relationship is simple, say linear, it can be found by exhaustive search. Once however a putative relationship has been posited, it can be tested for by systematically varying all parameters. If there is indeed a *functional relationship*, the relation $(g(\psi), m)$ has to be injective. In a diagram of m versus g, all data points must then fall on one curve. This phenomenon, if it occurs, is called "data-collapse", and serves to prove that the posited relationship correctly describes the system. If we were to apply that to the K-SAT case mentioned in section 8.2.3, m is the fraction of unsatisfiable instances, and $\psi = \{M, N\}$ and $\alpha = g = \frac{M}{N}$.

8.2.5 Application of the approach in distributed systems

Having explained the importance of identifying intensive variables in describing characteristics that hold irrespective of size and shown the pragmatic tool to find such variables, namely, data collapse, we state the main question that we try to answer in this work and summarize the methodology we use to answer that question:

> *"Is it possible to find intensive variables to describe the characteristics of structured peer-to-peer overlay networks that deploy periodic maintenance of the overlay graph?"*

The Methodology

Step 1: Nomination of intensive variables. This step is a speculative step where we try to nominate a subset of the parameters ($\psi \subset \{S\} \cup P$)that affect the performance and speculate that a function of them ($g(\psi)$) can be an intensive variable.

Step 2: Looking for a performance metric. In this step a performance metric $m(g(\psi))$ is identified and is usually an easier step for systems where a performance metric is already agreed upon in the community but the step can also involve the introduction of new metrics.

Step 3: Simulation. With plotting m versus g in mind, we chose a range of values of ψ to explore a wide spectrum of parameter interaction. Finally, if a data collapse is observed, then an intensive variable is identified.

8.3 Background & assumptions about Chord

Despite the fact that the Chord system is one of the most clearly explained DHT systems, when implementing it there are some nuances in the implementation details that can affect the performance of the system. We provide here a semi-formal definition of Chord and state our assumptions when needed.

- **The Chord Graph**: $G = (V, E)$ where V is the set of nodes (machines/processes) and E is a set of directed edges.
- **Identifier** Id_u: Each node in the set V has a unique identifier obtained by hashing a unique property such as its IP address or public key. We write Id_u to refer to the Id of node $u \in V$ and we label the edges using the corresponding identifiers of the nodes.
- **Size of the identifier space** N: The number of possible identifiers. The Chord system assumes that all the nodes are totally ordered in a circle that has N positions.
- **Population Size** P: The number of nodes $P = |V|$, $1 \leq |V| \leq N$).
- **Successor of node** u**,** $Succ(Id_u)$: The successor of an identifier Id_u is the first node following u in the circular identifier space. e.g if $N =$

16, $V = \{3, 11\}$, then we have two nodes whose identifiers are 3 and 11. Therefore the successor of any of the identifiers $4, 5, ...10, 11$ is node 11, similarly the successor of any of the nodes $12, 13, 14, 15, 1, 2, 3$ is node 3.

- **Routing table of node** u**:** $RT(Id_u) = \{(Id_u, Id_v) : Id_v = Succ(Id_u + 2^{i-1})\}, (1 \leq i \leq \log_2 N)$

- **Succesor List of node** u**:** $SL(Id_u) = \{(Id_u, Id_v) : Id_v = Succ(Id_u)$, $Succ(Succ(Id_u))$, $Succ(Succ(Succ(Id_u)))$, ... upto $\log_2 N$ successors $\}$

- **Lookup process:** A lookup of Id_q at a node u results in u forwarding the lookup to a node v pointed to by the highest edge of u preceding Id_q. v acts similarly. With each crossing of an edge (hop), the distance to Id_q decreases by a factor of a half at least and thus after $O(\log_2 P)$ hops the lookup reaches its goal.

- **Joins:** When a new node joins the system through any other node, it performs a lookup to find out its successor and then uses a periodic stabilization algorithm to correct all its edges. During that process, both its edges, and the edges of other nodes that should point to it are out-of-date and this results in lookups taking more hops to be resolved.

- **Failures:** When a node fails ungracefully - that is, without notifying the nodes pointing to it of its failure - lookups from nodes who have edges pointing to the failed node will fail and will be retried with lower edges. This also results in the lookups taking more hops to be resolved.

- **Stabilization:** There has been a couple of stabilization algorithms suggested for reparing the routing table. One iterates through the edges in a round-robin fashion and another picks randomly one of the edges. In our implementation we use the second. For the stabilization of the successor list, there was a non-formal description in the Chord paper, and we adopted a naive interpretation where each

nodes asks its successor for its successor list and blindly replaces its own dropping the last entry. We also assume for simplicity that the stabilization period for the routing table and the successor list is the same.

8.4 Intensive variable A: Density (ρ)

8.4.1 Application of the Methodology

Our first application of the methodology is to study a variable related to the overlay in an optimal static setting which might be of more theoretical interest, however it outlines the application of our methodology on a simple example first before moving to a more complex example that will be discussed in 8.5.

Step 1: Nomination of intensive variables. Let N be the size of the identifier space and P be the population as defined in section 8.3. We define the density (ρ) to be the ratio $\frac{P}{N}$ with a maximum value of 1 for a fully populated system. Our question is: "Is ρ an intensive variable?"

Step 2: Looking for a performance metric. A key quantity of interest in a DHT system is the average lookup path length. It indicates how well an overlay network structure can minimize the number of hops/edges traversed before a query is resolved and is the most-widely used metric for evaluating the performance of overlay networks.

Step 3: Simulation. Let CHORD(P,N) be an optimal Chord graph, that is all the edges of all nodes are correctly assigned according to the definitions in section 8.3. For all $N \in \{2^7, 2^8, .., 2^{14}\}$, for all $P \in \{0.1 \times N, 0.2 \times N, .., 1.0 \times N\}$, we generate CHORD($P$,$N$), inject uniformly distributed P^2 lookups, and record the average lookup length over the P^2 lookups denoted $\langle L(P,N)\rangle$ or equivalently $\langle L(\rho,N)\rangle$. This procedure is repeated 10 times, with different random seeds, and the results are averaged.

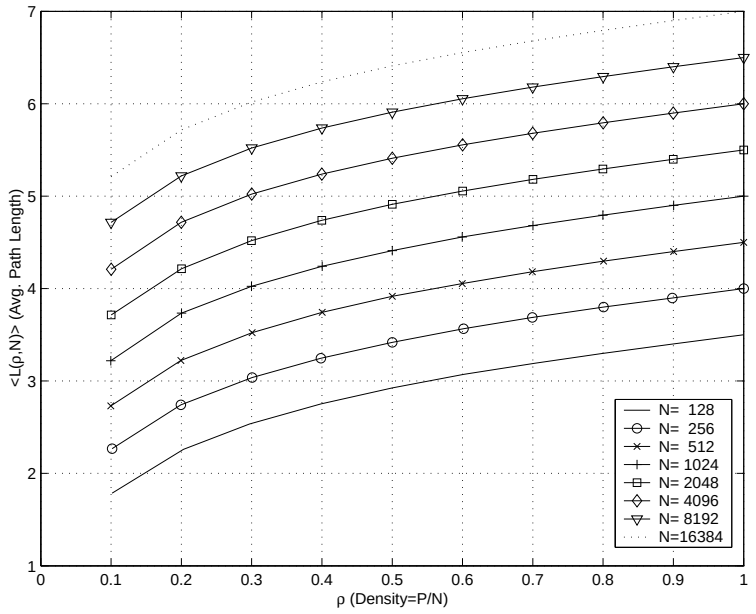


Figure 8.1: The average lookup length as a function of ρ and N.

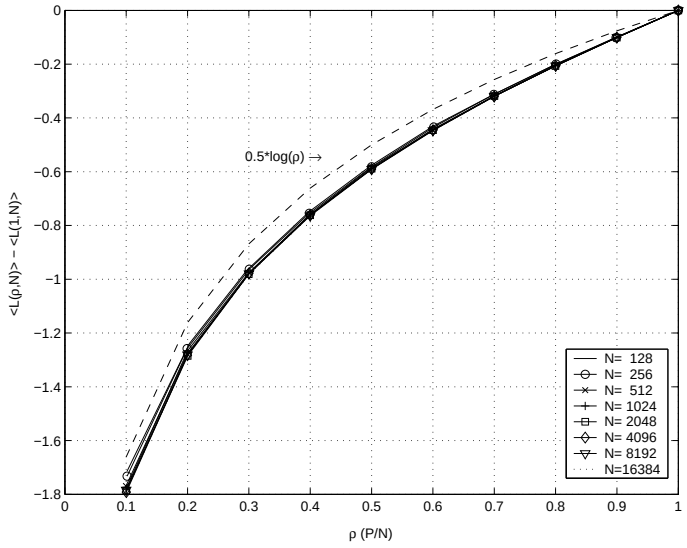


Figure 8.2: Data collapse of the average lookup length as a function of ρ and N compared to $0.5 \log_2 \rho$.

8.4.2 Results.

Figure 8.1 shows the behavior of the path length as a function of the density and the size of the identifier space. The curves are, to first approximation, vertically shifted by the same distance, while the values of N used are exponentially spaced. This means that the dependence on N alone (constant P) is logarithmic. Indeed, it was noted in the Chord papers that the average path length is $0.5 \times \log_2 P$. However, we can see an additional observation by looking at the data collapse obtained in figure 8.2 by subtracting $\langle L(1, N)\rangle$ from every respective curve $\langle L(\rho, N)\rangle$ compared to $0.5 \log_2 \rho$. From the data collapse, we can clearly see that $\langle L(\rho, N)\rangle = 0.5 \log_2 \rho + f(\rho)$ where the function f is a decreasing function. That is for any given number of nodes, the average lookup length increases when they are placed in a smaller identifier space.

It is a curious fact that the function $f(\rho)$ alluded to above is decreasing. We call this curious, because if the P populated nodes are regularly spaced in the circular geometry of the address space of Chord, the average path length is exactly $0.5 \log_2 P$, in other words larger. Hence, we have as a result that randomization improves the performance of P2P system built on DHT, even in a static situation, with no peers leaving or joining the system. We believe this may be of some conceptual importance, even if the effect is small. Additionally, it shows that a characteristic of a self property such as self-organization of nodes into a structured graph can be described irrespective of the size of the system.

> **Conclusion 1.** *The density (ρ) of nodes in an identifier space of a Chord system is an intensive variable.*

8.5 Intensive variable B: Ratio of Perturbation to Stabilization (β)

We like to perceive a structured overlay network as a system operating under two competing forces: perturbation and stabilization. Perturbation is the change in the set of nodes by adding and removing nodes, aka "churn". Stabilization is the periodic maintenance of edges performed

by each node. Perturbation pulls the network towards suboptimal performance because whenever a node is added or removed, some outgoing edges of some other nodes need to be changed. Stabilization brings it back to optimal state by reassigning a sub-optimally-assigned edges. The competition between those two forces governs the performance of the system. So far, there is no research results in the DHT community, that we are aware of, that can answer the following question:

> **Question** 1**.** *If a system of size S is operating under a magnitude of perturbation x and a magnitude of stabilization y, what is the performance of the system according to some performance metric $m(x, y)$?*

A comprehensive simulation-based answer to question 1 can not be obtained without an exhaustive exploration of all the parameters which is prohibitive. Instead, we try to answer it by posing the two following simpler questions which in essence are attempts to find data collapse:

> **Question** 2 **(Perturbation-Stabilization Equilibrium).** *Assume that we know $m(x_1, y_1)$ where x_1 and y_1 are some values of perturbation and stabilization respectively, is $m(x_1, y_1) = m(\alpha \times x_1, \alpha \times y_1)$ where α is a constant? Differently said, is there an equilibrium of those two competing forces?*

> **Question** 3 **(Scalability of the equilibrium).** *Whatever the answer to question 2 is, how does it hold for larger system sizes?*

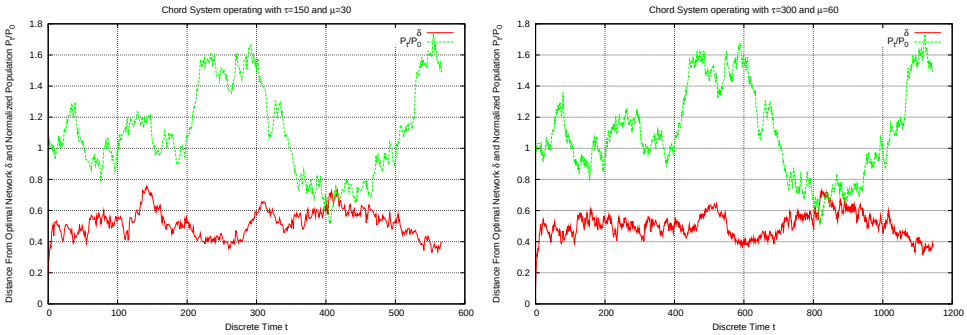
8.5.1 Application of the Methodology

To answer the above two questions we apply our methodology once more as follows:
Step 1: Nomination of intensive variables. Let τ be the time between two stabilization actions of a certain node. Let μ be the average time between two perturbation events (joins or failures) while the network is in a stable state. That is, the number of nodes is varying around a certain average population P_0. Taking μ as the magnitude of perturbation and τ as the magnitude of stabilization, we need to answer the following question: "Is $\beta = \frac{\mu}{\tau}$ an intensive variable?".

Step 2: Looking for characteristic behavior. In this investigation, in addition to the average lookup path length as the indicator of characteristic behavior, we needed a metric that gives more insight about the state of the graph and thus we used what we call the "distance from optimal network" δ which is computed as follows:

$$\delta = \frac{\sum_{i \in P} \sum_{j=1..\log N} \text{Edge}_j^i \neq \text{OptimalEdge}_j^i}{P \log N} \tag{8.1}$$

Where Edge_j^i is the $j^{th}(1 \leq j \leq \log N)$ outgoing edge of a node $i \in P$ and OptimalEdge_j^i is the optimal value for that edge. $P \log N$ is the total number of edges (P nodes, $\log N$ edges per node, where N is the size of the identifier space). Informally, δ is the number of "wrong" (outdated) edges in the overlay graph over the total number of edges. It varies between 0 and 1 where a value of 0 means that the graph is optimal.



(a) (b)

Figure 8.3: Example experiments showing the average normalized population size of 128 nodes under perturbation (joins/failure) and the average distance from optimal network under two different rates of perturbation (μ) and stabilization (τ) but the same $\beta = \frac{\mu}{\tau}$

Step 3: Simulation Setup.
We let P_0 nodes form a network and we wait until δ is equal to 0 i.e. the

network graph is optimal. We then let the network operate under specified values of μ and τ for 50 turnovers (A turnover is the replacement of P nodes with another P). During this experiment, we record δ frequently and average it over the whole experiment. That is in addition to the average lookup path length L over the whole experiment as well. We write $\langle \delta(P, \tau, \mu) \rangle$ and $\langle L(P, \tau, \mu) \rangle$ to denote the average δ and L obtained by running this experiment setup under given values of P, τ and μ.

Figure 8.3 shows example experiments. On the x-axis two values are plotted at each discrete time t: $i)$ δ at time t, $ii)$ $\frac{P_t}{P_0}$, the population at time t divided by the initial population P_0. During the experiment, the average population is P_0 (That is $\left\langle \frac{P_t}{P_0} \right\rangle = 1.0$). In figure 8.3(a) the values of τ and μ were 150 and 30 respectively, therefore $\beta = 0.2$. In figure 8.3(b) both values of τ and μ are doubled while keeping beta constant. Interestingly, the average distance in both examples is almost the same, that is $\langle \delta(128, 150, 30) \rangle = 0.518$ and $\langle \delta(128, 300, 60) \rangle = 0.504$. While those examples are promising for showing that an equilibrium of perturbation and stabilization exists, we show the investigation of a broad range of values of β in the next section.

8.5.2 Results

Perturbation-Stabilization Equilibrium

For one value of P_0, we examine various values of β by fixing τ and varying μ. We, then, try a different τ and vary μ such that the same values of β are conserved. The target of this procedure is to understand whether δ is dependent solely on β or is it one of the components of β (τ or μ) that controls $\langle \delta \rangle$ and $\langle L \rangle$.

In more details, we try values of $\tau = \{150, 300, 600, 1200\}$. For each value of τ we try different values of μ. For instance, for $\tau = 150$, values of $\beta = \{25, 30, 45, ..., 195, 200\}$. For $\tau = 150$, values of $\beta = \{50, 60, 70, ..., 390, 400\}$. The same procedure is repeated for values of $P_0 = \{64, 128, 512, 1024\}$.

The results are shown in figures 8.5 and 8.4. Observe that as β increases the time between two perturbation events increase, i.e. the network has

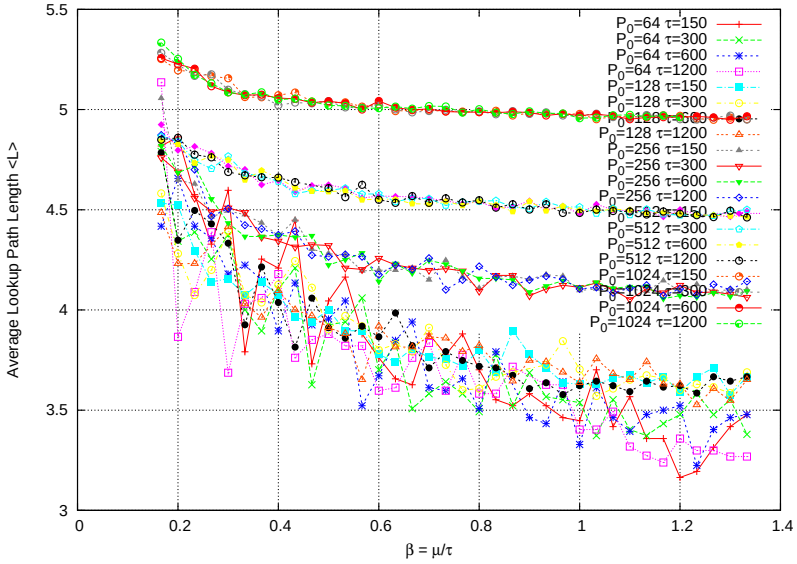


Figure 8.4: The average lookup path length $\langle L \rangle$ as a function of the speculated intensive variable β (the ratio of average time between perturbation events μ and average time between stabilization events τ).

more chance to heal itself. Therefore, it is expected to see that both $\langle L \rangle$ and $\langle \delta \rangle$ decrease indicating a better performance.

In figure 8.5 we find that for smaller network sizes, the average lookup length oscillates strongly and it is not clear that β alone controls the behavior. For larger sizes, this oscillation disappears and we clearly see that the curves superimpose nicely and a data collapse is obtained. Another way to see the behavior of the lookup length is to plot its deviation from the optimal value ($\langle L_{opt} \rangle = 0.5 \times \log \langle P_0 \rangle$) as shown in figure 8.6. We ignore the density effect as we are working with an identifier space of size $N = 4096$ and therefore $f(\rho)$ is negligible.

Unlike the average lookup path length, the distance from optimal network gives a much more consistent view of the network behavior. As shown in figure 8.5, for each given size all the curves superimpose nicely.

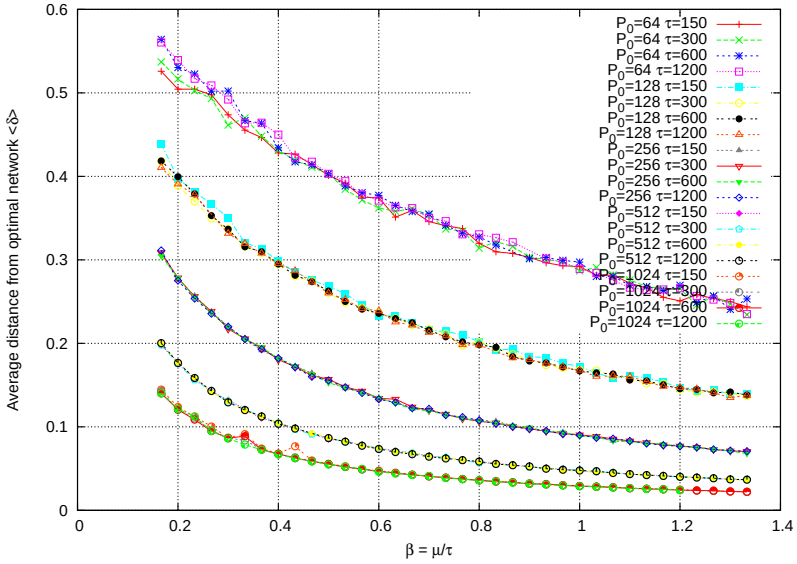


Figure 8.5: The average distance from optimal network $\langle\delta\rangle$ as a function of the speculated intensive variable β.

That is, we have a much more vivid data collapse. A discussion of a data collapse for all the sizes is provided in the next section. However, at this point, we can see that the network exhibits a perturbation-stabilization equilibrium.

Scalability of the equilibrium

If we want to discuss the scalability of the equilibrium, we need to perceive the obtained results differently. The stabilization as defined in section 8.5.1 is a "node-level" event while the perturbation is a "whole-graph-level" event. That is, β is defined as $\frac{Perturbation\ of\ the\ system\ (\mu)}{Stabilization\ of\ each\ node\ (\tau)}$. Therefore, if we were to compare the behavior of two network sizes under the same values of τ and μ, the network with larger size will have the same perturbation but higher stabilization since the number of nodes is larger. Therefore, we

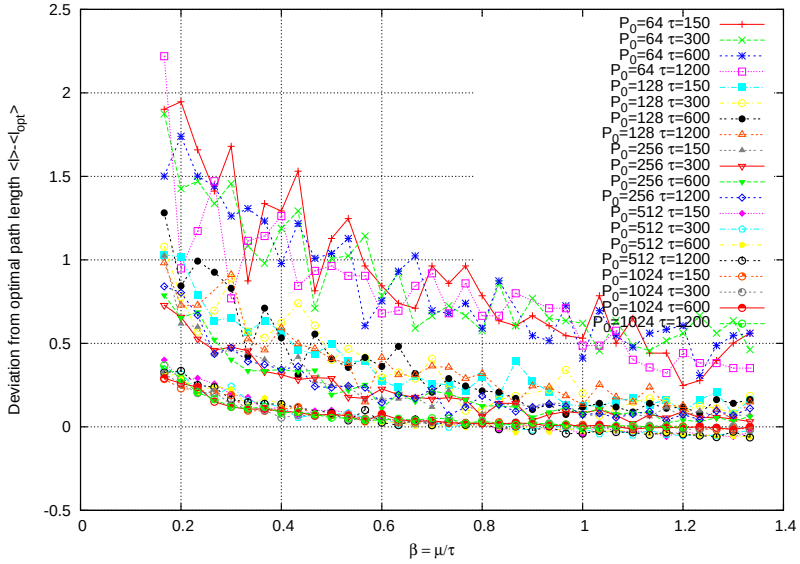


Figure 8.6: The deviation from optimal average lookup path length $\langle L \rangle - \langle L_{opt} \rangle$ as a function of the speculated intensive variable β.

define β' to be $\frac{Perturbation\ of\ the\ system\ (\mu)}{Stabilization\ of\ the\ system\ (\frac{\tau}{\langle P_0 \rangle})}$ to have a more fair comparison. The β' re-plots of figure 8.6 and 8.5 are shown in figures 8.7 and 8.8 respectively.

In figure 8.7 we can see that we have a noisy data-collapse due to the oscillations of the lookup length of the small networks. Nevertheless, a clear common trend is obvious and we believe that enhancing it more is an exercise in statistical methods. However, judging it by using the distance from optimal network (figure 8.8), we can see that we have a clear data collapse.

Conclusion 2. The ratio of system perturbation to system stabilization β' in a Chord system is an intensive variable.

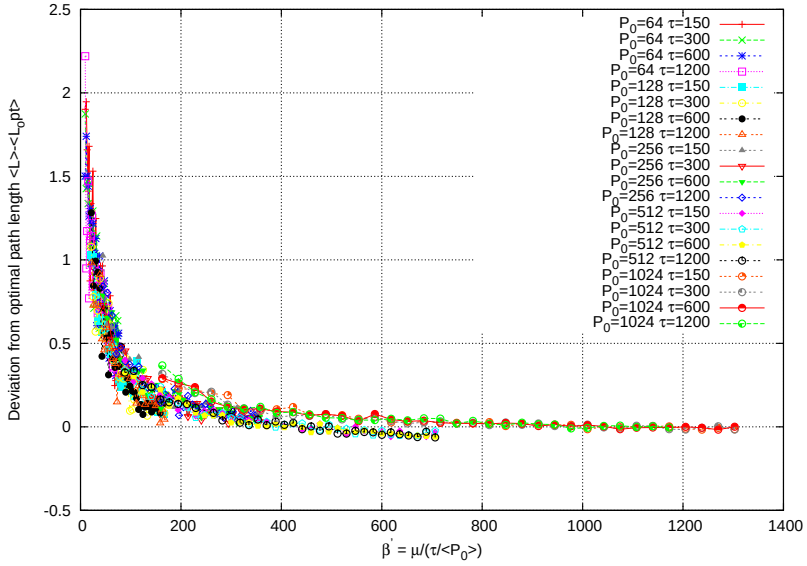


Figure 8.7: Data collapse of figure 8.6 obtained by using β'.

8.6 Related Work

We consider in this section the research in the structured overlay networks community, that we are aware of, which addressed self-repair on deep levels.

A lower bound on stabilization rate. In [5], the Chord research team gives the first theoretical analysis on the lower bound of stabilization rate required for a network to remain connected in face of continuous joins and failures. A lower bound for the stabilization rate is definitely a substantial result, however, we are interested to answer more questions regarding the stabilization rates such as: If we consider the range of stabilization rates where the overlay graph won't be disconnected, the guarantee of graph disconnection is not sufficient for practical purposes, how is the network performance affected by each such rate? Differently said, if we have an application that demands a limit on the latency in terms of the number of hops, what is the required stabilization rate?

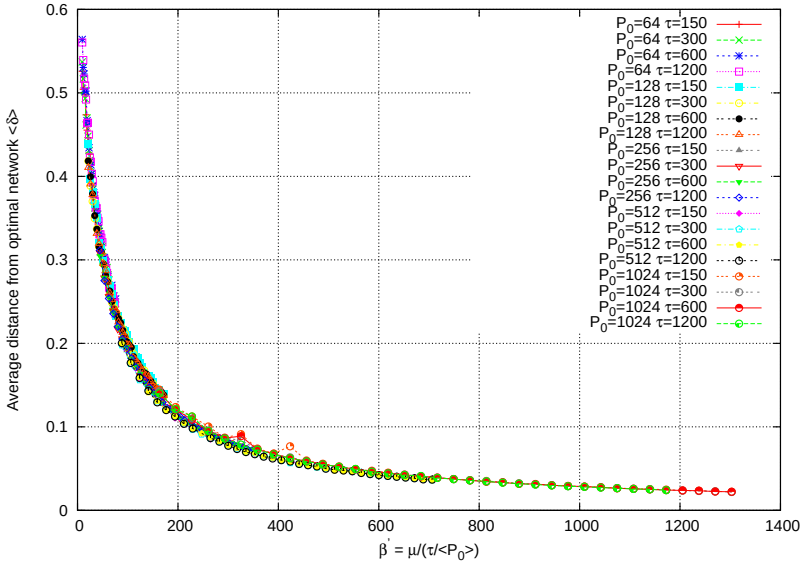


Figure 8.8: Data collapse of figure 8.5 obtained by using β'.

Self-tuned message failure rate. In [6], the Pastry research team gives an analytical model for the percentage of message loss due to outdated routing tables as a function of the frequency of routing table probes. The model is then used to adaptively "self-tune" the rate at which the network self-repairs its routing entries that are pointing to failed nodes which is another major result because of its practical implications. However, the analysis based on the failed nodes only is not sufficient since a node can have alive but sub-optimally assigned edges.

Comparing performance under churn. The work in [4] gives the first thorough simulation-based analysis for overlays network under churn (perturbation). In that sense it is the most similar research to our work. The simulation model covers physical network proximity and compares several systems and not only Chord. On the other hand, their simulation is based on one churn rate while in ours we tackle several rates and we are interested in finding data-collapse so as to present results that hold irre-

spective of the network size.

8.7 Note on the implementation.

To perform the experiments, we used the Mozart [1] distributed programming platform to implement a discrete event simulator. We wrapped our simulator in a distribution layer to be able to schedule experiments on a cluster of 16 nodes at SICS. Each nodes is an AMD Athlon(tm) XP 1900+ (1.5GHz) with a 512MB of memory.

8.8 Conclusion and future work

We have reported in this paper our progress in investigating whether a physics-style analytical approach can give more understanding to the performance of structured overlay networks. The approach mainly necessitates the description of the characteristics of the system using variables that do not depend on the size, known as intensive variables.

Using this approach, we have shown that: $i)$ The density of nodes in an identifier space is an intensive variable that describes a characteristic behavior of a network irrespective of its size. $ii)$ The ratio of perturbation to stabilization β', variable governs the relative amount of wrong pointers and the average lookup path length of the overlay graph irrespective of its size.

In the continuation of this work, we intend to do the following:

- Perform the same experiments with a wider spectrum of numbers to have more statistically-accurate results.
- Use the characteristic behaviors in providing a more adaptive nature to the current DHT algorithms.
- Search for more intensive variables and possible phase transitions.

Bibliography

[1] Mozart Consortium, 2003. http://www.mozart-oz.org.

[2] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. DKS(N; k; f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *The 3rd International Workshop On Global and Peer-To-Peer Computing on Large Scale Distributed Systems (CCGRID 2003)*, Tokyo, Japan, May 2003. http://www.ccgrid.org/ccgrid2003.

[3] S. Kirkpatrick and B. Selman. Critical behaviour in the satisfiability of random boolean expressions. *Science*, 264:1297–1301, 1994.

[4] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *The 3rd International Workshop on Peer-to-Peer Systems (IPTPS'02)*, San Diego, CA, Feb 2004.

[5] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *ACM Conf. on Principles of Distributed Computing (PODC)*, Monterey, CA, July 2002.

[6] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlayss. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[7] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *InProceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, August 2002.

[8] Petar Maymounkov and David Mazires. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[9] M. Mézard, G. Parisi, and R. Zecchina. Analytic and algorithmic solutions of random satisfiability problems. *Science*, 297:812–815, 2002.

[10] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. *AAAI-92. Proceedings Tenth National Conference on Artificial Intelligence*, pages 873, 459–65, 1992.

[11] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. 2+p-sat: Relation of typical-case complexity to the nature of the phase transition. *Random Structures and Algorithms*, 3:414, 1999.

[12] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic phase transitions. *Nature*, 1999.

[13] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *InProceedings of SPAA 2003*, 2003.

[14] 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)*, 329-350 2001.

[15] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, August 2001.

[16] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, 2003.

[17] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2000.

Chapter 9

Analytical Study of DHTs under Churn

Analytical Study of Consistency and Performance of DHTs under Churn. [1]

Sameh El-Ansary[1], Supriya Krishnamurthy[1], Erik Aurell[1,2] and Seif Haridi[1,3]

[1] Distributed Systems Laboratory
SICS Swedish Institute of Computer Science
[2]Department of Physics, KTH-Royal Institute of Technology
[3]IMIT, KTH-Royal Institute of Technology
{sameh,supriya,eaurell,seif}@sics.se

Abstract

In this paper, we present a complete analytical study of dynamic membership (aka churn) in structured peer-to-peer networks. We use a master-equation-based approach, which is used traditionally in non-equilibrium statistical mechanics to describe steady-state or transient phenomena. We demonstrate that this methodology is infact also well suited to describing structured overlay networks by an application to the Chord system. For any rate of churn and stabilization rates, and any system size, we accurately account for the functional form of: the distribution of inter-node distances, the probability of network disconnection, the fraction of failed or incorrect successor and finger pointers and show how we can use these quantities to predict both the performance and consistency of lookups under churn. Additionally, we also discuss how churn may actually be of different 'types' and the implications this will have for structured overlays in general. All theoretical predictions match simulation results to a high extent. The analysis includes details that are applicable to a generic structured overlay deploying a ring as well as Chord-specific details that can act as guidelines for analyzing other systems.

Keywords: Peer-To-Peer, Structured Overlays, Distributed Hash Tables, Dynamic Membership in Large- scale Distributed Systems, Analytical Modeling, Master Equations.

[1]This work is funded by the Swedish VINNOVA AMRAM and PPC projects, the European IST-FET PEPITO and 6th FP EVERGROW projects.

9.1 Introduction

An intrinsic property of Peer-to-Peer systems is the process of never-ceasing dynamic membership. Structured Peer-to-Peer Networks (aka Distributed Hash Tables (DHTs)) have the underlying principle of arranging nodes in an overlay graph of known topology and diameter. This knowledge results in the provision of performance guarantees. However, dynamic membership continuously "corrupts/churns" the overlay graph and every DHT strives to provide a technique to "correct/maintain" the graph in the face of this perturbation.

Both theoretical and empirical studies have been conducted to analyze the performance of DHTs undergoing "churn" and simultaneously performing "maintenance'. Liben-Nowell et. al [6] prove a lower bound on the maintenance rate required for a network to remain connected in the face of a given dynamic membership rate. Aspnes et. al [2] give upper and lower bounds on the number of messages needed to locate a node/data item in a DHT in the presence of node or link failures. The value of such theoretical studies is that they provide insights neutral to the details of any particular DHT. Empirical studies have also been conducted to complement theory by showing how within the asymptotic bounds, the performance of a DHT may vary substantially depending on different DHT designs and implementation decisions. Examples include the work of: Li et. al [5], Rhea et.al [8] and Rowstron et.al [3].

In this paper, we present a new approach to studying churn, based on working with master equations, a widely used tool wherever the mathematical theory of stochastic processes is applied to real-world phenomena [7]. We demonstrate the applicability of this approach to one specific DHT: Chord [9].

9.2 Related Work

Closest in spirit to our work is the informal derivation in the original Chord paper [9] of the average number of timeouts encountered by a lookup. This quantity was approximated there by the product of the average number of fingers used in a lookup times the probability that a given

finger points to a departed node. Our methodology not only allows us to derive the latter quantity rigorously but also demonstrates how this probability depends on which finger (or successor) is involved. Further we are able to derive an exact relation relating this probability to lookup performance and consistency accurately at any value of the system parameters.

In the works of Aberer et.al [1] and Wang et.al [10], DHTs are analyzed under churn and the results are compared with simulations. However, the main parameter of the analysis is the probability that a random selected entry of a routing table is stale. In our analysis, we determine this quantity from system details and churn rates.

A brief announcement of the results presented in this paper, has appeared earlier in [4].

9.3 Assumptions & Definitions

Basic Notation. In the sequel, we assume that the reader is familiar with Chord. However we introduce the notation used below. We use $\mathcal{K}$ to mean the size of the Chord key space and N the number of nodes. Let $\mathcal{M} = \log_2 \mathcal{K}$ be the number of fingers of a node and $\mathcal{S}$ the length of the immediate successor list, usually set to a value $= O(\log(N))$. We refer to nodes by their keys, so a node n implies a node with key $n \in 0 \cdots \mathcal{K} - 1$. We use p to refer to the predecessor, s for referring to the successor list as a whole, and s_i for the i^{th} successor. Data structures of different nodes are distinguished by prefixing them with a node key e.g. $n'.s_1$, etc. Let fin_i*.start* denote the start of the i^{th} finger (where for a node n, $\forall i \in 1..\mathcal{M}$, $n.fin_i.start = n + 2^{i-1}$) and fin_i*.node* denote the node pointed to by that finger (which is the closest successor of $n.fin_i.start$ on the ring).

Steady State Assumption. λ_j is the rate of joins per node, λ_f the rate of failures per node and λ_s the rate of stabilizations per node. We carry out our analysis for the general case when the rate of doing successor stabilizations $\alpha\lambda_s$, is not necessarily the same as the rate at which finger stabilizations $(1 - \alpha)\lambda_s$ are performed. In all that follows, we impose the steady state condition $\lambda_j = \lambda_f$ unless otherwise stated. Further it is useful to define $r \equiv \frac{\lambda_s}{\lambda_f}$ which is the relevant ratio on which all the quantities we are interested in will depend, e.g, $r = 50$ means that a join/fail event takes

place every half an hour for a stabilization which takes place once every 36 seconds. Throughout the paper we will use the terms $\lambda_j \Delta t$, $\lambda_f \Delta t$, $\alpha\lambda_s \Delta t$ and $(1-\alpha)\lambda_s \Delta t$ to denote the respective probabilities that a join, failure, a successor stabilization, or a finger stabilization take place during a micro period of time of length Δt.

Parameters. The parameters of the problem are hence: $\mathcal{K}$, N, α and r. All relevant measurable quantities should be entirely expressible in terms of these parameters.

Chord Algorithms & Simulation A detailed description of the algorithms used is provided in Appendix A. Since we are collecting statistics like the probability of a particular finger pointer to be wrong, we need to repeat each experiment 100 times before obtaining well-averaged results. The total simulation sequential real time for obtaining the results of this paper was about 1800 hours that was parallelized on a cluster of 14 nodes where we had $N = 1000$, $\mathcal{K} = 2^{20}$, $\mathcal{S} = 6$, $200 \leq r \leq 2000$ and $0.25 \leq \alpha \leq 0.75$.

9.4 The Analysis

9.4.1 Distributional Properties of Inter-Node Distances

During churn, the average inter-node distance is a fluctuating quantity whose distribution is used throughout our analysis. The derivation we present here of this distribution is independent of any details of the DHT implementation and depends solely on the dynamics of the join and leave process. It is hence applicable to any DHT that deploys a circular key space.

Definition 9.4.1 *Given two keys $u, v \in \{0...\mathcal{K}-1\}$, the "distance" between them is $u - v$ (with modulo-$\mathcal{K}$ arithmetic). We interchangeably say that u and v form an "interval" of length $u - v$. Hence the number of keys inside an interval of length ℓ is $\ell - 1$ keys.*

Definition 9.4.2 *Let Int_x be the number of intervals of length x, i.e. the number of pairs of consecutive nodes which are separated by a distance of x keys on the ring.*

$Int_x(t+\Delta t)$	Rate of Change
$= Int_x(t) - 1$	$c_{1.1} = (\lambda_f \Delta t) 2P(x)$
$= Int_x(t) - 1$	$c_{1.2} = (\lambda_j \Delta t) \frac{N(x-1)P(x)}{\mathcal{K}-N}$
$= Int_x(t) + 1$	$c_{1.3} = (\lambda_f \Delta t) \sum_{x_1=1}^{x-1} P(x_1)P(x-x_1)$
$= Int_x(t) + 1$	$c_{1.4} = (\lambda_j \Delta t) \frac{2N}{\mathcal{K}-N} \sum_{x1>x} P(x_1)$
$= Int_x(t)$	$1 - (c_{1.1} + c_{1.2} + c_{1.3} + c_{1.4})$

Table 9.1: Gain and loss terms for $Int(x)$ the number of intervals of length x.

Theorem 9.4.1 *For a process in which nodes join or leave with equal rates independently of each other and uniformly on the ring, and the number of nodes N in the network is almost constant with $N << K$, the probability ($P(x) \equiv \frac{Int_x}{N}$) of finding an interval of length x is: $P(x) = \rho^{x-1}(1-\rho)$ where $\rho = \frac{\mathcal{K}-N}{\mathcal{K}}$.*

Proof : By definition $\sum P(x) = 1$ and $\sum x\, P(x) = \mathcal{K}/N$. Further, for the mean number of peers, the join-leave process we consider, simply implies that $\frac{dN}{dt} = \lambda_j - \lambda_f$ We will need to check that an equation for $Int_(x)$ does indeed satisfy the above constraints.

We now write an equation for Int_x by considering all the processes which lead to its gain or loss. These are summarized in table 9.1

First, a failure of either of the boundary nodes of an interval of size x leads to its loss at rate $c_{1.1}$. That is, since the node killed is randomly picked amongst all the nodes in the interval, the probability that it was participating on either side of an interval of length x is $2P(x)$.

Second, an interval of size x can be lost at rate $c_{1.2}$ if a joining node splits it. Only joining with keys that belong to one of the Int_x intervals can lead to the loss of an interval of length x and in each one of these, there are $x-1$ ways (available keys) for splitting. Therefore $(x-1) \times Int_x$ positions out of the $\mathcal{K} - N$ available keys can destroy an interval of length x. That is, the probability that one of the intervals of length x is destroyed is $\frac{(x-1)Int_x}{\mathcal{K}-N}$ which can be rewritten as $\frac{N(x-1)P(x)}{\mathcal{K}-N}$.

Third, the number of intervals of size x can increase by 1 at rate $c_{1.3}$ if a failure of a boundary node results in the aggregation of two adjacent intervals. To clarify that, we give the following examples. An interval of

length 1 cannot be formed by such a process. An interval of length 2 can be formed by the failure of a node if the node that failed was shared between two adjacent intervals of length 1. We are assuming here that the probability of picking two adjacent intervals of length 1 is $P(1)^2$. This is in effect assuming that the probability of having two adjacent intervals of size 1, factorises. However for this system, this is an accurate estimation. Thus, in general, the probability of forming an interval of length x is $\sum_{x1=1}^{x-1} P(x_1)P(x-x_1)$.

Fourth, an increase can happen at rate $c_{1.4}$ if a join event splits a larger interval into an interval of size x. For a join to form an interval of length x, it must occur in an interval of length greater than x. In each interval of length $x_1 > x$, there are exactly two ways of forming an interval of length x. Therefore, the probability of forming an interval of length x is equal to $\frac{2\sum_{x_1>x} Int_x}{\mathcal{K}-N}$, which can be rewritten as $\frac{2N\sum_{x_1>x} P(x)}{\mathcal{K}-N}$

Finally, Int_x remains the same if none of the above happens.

Therefore the equation for Int_x for $x > 1$ is:

$$\begin{aligned}\frac{dInt_x}{dt} =& -P(x)\left[2\lambda_f + \frac{N\lambda_j(x-1)}{\mathcal{K}-N}\right] \\ &+ \lambda_f \sum_{x_1=1}^{x-1} P(x_1)P(x-x_1) \\ &+ 2\lambda_j \frac{N}{\mathcal{K}-N} \sum_{x_1>x} P(x_1).\end{aligned} \tag{9.1}$$

We can check that :

$$\frac{d}{dt}\sum Int_x = \frac{dN}{dt} = \lambda_j - \lambda_f \tag{9.2}$$

as required.

Further we can check that the constraint:

$$\frac{d}{dt}\sum xInt_x = \frac{d\mathcal{K}}{dt} = 0$$

is also obeyed. Equation 9.1 can be readily solved leading to the solution:

$$P(x) = \rho^{x-1}(1-\rho) \tag{9.3}$$

where $\rho = \frac{\mathcal{K}-N}{\mathcal{K}-N(1-\frac{\lambda_j}{\lambda_f})}$. In the special case we are interested in here where $\lambda_j = \lambda_f$, we have $\rho = \frac{\mathcal{K}-N}{\mathcal{K}}$. Note that if $\lambda_j \neq \lambda_f$, then N is actually an increasing/decreasing function of time. ■

Given the above term for ρ we can state the following corollary that gives an intuitive meaning for ρ in the case $\lambda_j = \lambda_f$.

Corollary 9.4.1.1 *Given a ring of $\mathcal{K}$ keys populated by N nodes, $\rho \equiv \frac{\mathcal{K}-N}{\mathcal{K}}$ is the ratio of the unpopulated keys to the total number of keys, i.e. the probability of picking a key at random and finding it empty is ρ.*

The proof of the above theorem does assume that (in the case $\lambda_j = \lambda_f$) the number of nodes N is fairly constant. Indeed at first sight this seems to be strictly true from Eq. 9.2. However, just as in a random walk, the variance in this case increases with time. We will comment more on the properties of the variance later. For the moment, we note that the above result can be generalised to also include the case when N is a largely fluctuating quantity. In this case we only need to multiply the N dependent terms in Eq. 9.1 with $Prob(N,t)$: the probability that there are N nodes in the system at time t, and average over N.

We now derive some Chord-specifig properties of this distribution which will be used in the ensuing analysis.

Property 9.4.1 *For any two keys u and v, where $v = u + x$, let b_i be the probability that the first node encountered inbetween these two keys is at $u + i$ (where $0 \leq i < x$). Then $b_i \equiv \rho^i(1-\rho)$. The probability that there is definitely at least one node between u and v is: $a(x) \equiv 1 - \rho^x$. Hence the conditional probability that the first node is at a distance i given that there is at least one node in the interval is $bc(i,x) \equiv b(i)/a(x)$.*

Explanation : Consider b_i first. For any key u, the probability that the first node encountered is at u itself (b_0) is $1-\rho$ from Corollary 9.4.1.1. Similarly the probability that the first node encountered is at $u+1$ (b_1) is $\rho(1-\rho)$. In general, the probability that the first populated node starting from u is at $u+i$ is $b(i) \equiv (\rho)^i(1-\rho)$. Given this, the probability that there is at least one node between u and $v = u + x$ (not including the case when the node is at v) is $\sum_{i=0}^{x-1} b_i = 1 - \rho^x \equiv a(x)$. ■

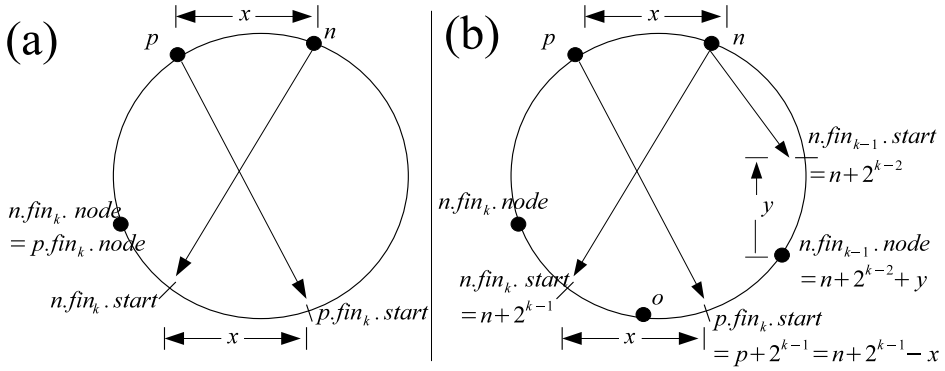


Figure 9.1: (a) Case when n and p have the same value of $fin_k.node$. (b) Case where a newly joined node p copies the k^{th} entry of its successor node n as the best approximation for its own k^{th} entry (by the join protocol). In this case, there could be a node o which is the 'correct' entry for $p.fin_k.node$. However, since p is newly joined, the only information it has access to is the finger table of n.

Property 9.4.2 *The probability that a node and at least one of its immediate predecessors share the same k^{th} finger is $p_1(k) \equiv \frac{\rho}{1+\rho}(1-\rho^{2^k-2})$. This is $\sim 1/2$ for $\mathcal{K} >> 1$ and $N << \mathcal{K}$.Clearly $p_1 = 0$ for $k = 1$. It is straightforward (though tedious) to derive similar expressions for $p_2(k)$ the probability that a node and at least two of its immediate predecessors share the same k^{th} finger, $p_3(k)$ and so on.*

Explanation : If the distance between node n and its predecessor p is x, the distance between $n.fin_k$.*start* and $p.fin_k$.*start* is also x (see Fig. 9.1(a)). If there is no node inbetween $n.fin_k$.*start* and $p.fin_k$.*start* then $n.fin_k$.*node* and $p.fin_k$.*node* will share the same value. From Eq. 9.3, the probability that the distance between n and p is x is $\rho^{x-1}(1-\rho)$. However, x has to be less than 2^{k-1}, otherwise $p.fin_k$.*node* will be equal to n. The probability that no node exists between $n.fin_k$.*start* and $p.fin_k$.*start* is ρ^x (by Property 9.4.1). Therefore the probability that the $n.fin_k$.*node* and $p.fin_k$.*node* share the same value is: $\sum_{x=1}^{2^{k-1}-1} \rho^{x-1}(1-\rho)\rho^x = \frac{\rho}{1+\rho}(1-\rho^{2^k-2})$ ■

Property 9.4.3 *We can similarly assess the probability that the join protocol results in further replication of the k^{th} pointer. Let us define the probability $p_{join}(i,k)$ as the probability that a newly joined node, chooses the i^{th} entry of its successor's finger table for its own k^{th} entry. Note that this is unambiguous even in the case that the successor's i^{th} entry is repeated. All we are asking is, when is the k^{th} entry of the new joinee the same as the i^{th} entry of the successor? Clearly $i \leq k$. Infact for the larger fingers, we need only consider $p_{join}(k,k)$, since $p_{join}(i,k) \sim 0$ for $i < k$. Using the interval distribution we find, for large k, $p_{join}(k,k) \sim \rho(1-\rho^{2^{k-2}-2})+(1-\rho)(1-\rho^{2^{k-2}-2})-(1-\rho)\rho(2^{k-2}-2)\rho^{2^{k-2}-3}$. This function goes to 1 for large k.*

Explanation : A newly joined node p, tries to assign $p.fin_k.node$ to the best approximate value from the finger table of its successor n. This approximate value might turn out to be $n.fin_k.node$, especially for the larger fingers. If p chooses the k^{th} entry of n as its own k^{th} entry, it must be because the $k-1^{th}$ entry of n (if distinct, as is always the case for large k) does not afford it a better choice. The condition for this is : $p.fin_k.start > n.fin_{k-1}.node$. If the distance between $n.fin_k.start$ and $p.fin_k.start$ is x, and the distance between $n.fin_{k-1}.start$ and $n.fin_{k-1}.node$ is y (see Fig. 9.1 (b)), then the constraint on x and y is $n + 2^{k-1} - x > n + 2^{k-2} + y$ or $x + y < 2^{k-2}$. We also have the added constraint that $x < 2^{k-1}$, since otherwise $p.fin_k.node$ would simply be n. Thus the probability $p_{join}(k,k)$ is:

$$\sum_{x=1}^{2^{k-1}-1} \sum_{y=1}^{2^{k-2}-x} P(x)P(y) = \sum_{z=2}^{2^{k-2}-1} \rho^{z-2}(1-\rho)^2(z-1) \tag{9.4}$$

■

where we have put in the expressions for $P(x)$ and $P(y)$ from Eq. 9.3 and converted the double summation to a single one. This expression can be summed easily to obtain the result quoted above.

We can also analogously compute $p_{join}(i,k)$ for any i. The only trick here is to estimate the probability that starting from i, the last *distinct* entry of n's finger table *does not* give p a better choice for its k_{th} entry. This can again readily be computed using property 9.4.1.

9.4.2 Successor Pointers

We now turn to estimating various quantities of interest for Chord. In all that follows we will evaluate various *average* quantities, as a function of the parameters. However this same formalism can also be used for evaluating higher moments like the variance.

In the case of Chord, we need consider only one of three kinds of events happening at any micro-instant: a join, a failure or a stabilization. One assumption made in the following is that such a micro-instant of time exists, or in other words, that we can divide time till we have an interval small enough that in this interval, only any one of these three processes occur. We also effectively assume that the time scales on which stabilizations occur is much faster that a join or a failure event. Another assumption is that the state of the system is a *product* of the state of all the nodes. Nodes are hence assumed to have, for the most part, states independent of each other , *i.e.* the probability of two adjacent nodes having a wrong successor pointer is taken to be the product of the individual nodes having wrong successor pointers (though as we will see, in the case of finger pointers, we do also consider the case when adjacent nodes might have correlated fingers). However, this ansatz works very well.

Consider first the successor pointers. Let $w_k(r, \alpha)$, $d_k(r, \alpha)$ denote the fraction of nodes having a *wrong* k^{th} successor pointer or a *failed* one respectively and $W_k(r, \alpha)$, $D_k(r, \alpha)$ be the respective *numbers* . A *failed* pointer is one which points to a departed node and a *wrong* pointer points either to an incorrect node (alive but not correct) or a dead one. As we will see, both these quantities play a role in predicting lookup consistency and lookup length.

By the protocol for stabilizing successors in Chord, a node periodically contacts its first successor, possibly correcting it and reconciling with its successor list. Therefore, the number of wrong k^{th} successor pointers are not independent quantities but depend on the number of wrong first successor pointers. We first consider s_1 here, and then briefly discuss the other cases towards the end of this section.

We write an equation for $W_1(r, \alpha)$ by accounting for all the events that can change it in a micro event of time Δt. An illustration of the different cases in which changes in W_1 take place due to joins, failures and stabi-

Change in $W_1(r, \alpha)$	Rate of Change
$W_1(t + \Delta t) = W_1(t) + 1$	$c_{2.1} = (\lambda_j \Delta t)(1 - w_1)$
$W_1(t + \Delta t) = W_1(t) + 1$	$c_{2.2} = \lambda_f (1 - w_1)^2 \Delta t$
$W_1(t + \Delta t) = W_1(t) - 1$	$c_{2.3} = \lambda_f w_1^2 \Delta t$
$W_1(t + \Delta t) = W_1(t) - 1$	$c_{2.4} = \alpha \lambda_s w_1 \Delta t$
$W_1(t + \Delta t) = W_1(t)$	$1 - (c_{2.1} + c_{2.2} + c_{2.3} + c_{2.4})$

Table 9.2: Gain and loss terms for $W_1(r, \alpha)$: the number of wrong first successors as a function of r and α.

lizations is provided in Fig. 9.2. In some cases W_1 increases/decreases while in others it stays unchanged. For each increase/decrease, table 9.2 provides the corresponding probability.

By our implementation of the join protocol, a new node n_y, joining between two nodes n_x and n_z, has its s_1 pointer always correct after the join. However the state of $n_x.s_1$ before the join makes a difference. If $n_x.s_1$ was correct (pointing to n_z) before the join, then after the join it will be wrong and therefore W_1 increases by 1. If $n_x.s_1$ was wrong before the join, then it will remain wrong after the join and W_1 is unaffected. Thus, we need to account for the former case only. The probability that $n_x.s_1$ is correct is $1 - w_1$ and from that follows the term $c_{2.1}$.

For failures, we have 4 cases. To illustrate them we use nodes n_x, n_y, n_z and assume that n_y is going to fail. First, if both $n_x.s_1$ and $n_y.s_1$ were correct, then the failure of n_y will make $n_x.s_1$ wrong and hence W_1 increases by 1. Second, if $n_x.s_1$ and $n_y.s_1$ were both wrong, then the failure of n_y will decrease W_1 by one, since one wrong pointer disappears. Third, if $n_x.s_1$ was wrong and $n_y.s_1$ was correct, then W_1 is unaffected. Fourth, if $n_x.s_1$ was correct and $n_y.s_1$ was wrong, then the wrong pointer of n_y disappeared and $n_x.s_1$ became wrong, therefore W_1 is unaffected. For the first case to happen, we need to pick two nodes with correct pointers, the probability of this is $(1 - w_1)^2$. For the second case to happen, we need to pick two nodes with wrong pointers, the probability of this is w_1^2. From these probabilities follow the terms $c_{2.2}$ and $c_{2.3}$.

Finally, a successor stabilization does not affect W_1, unless the stabilizing node had a wrong pointer. The probability of picking such a node is

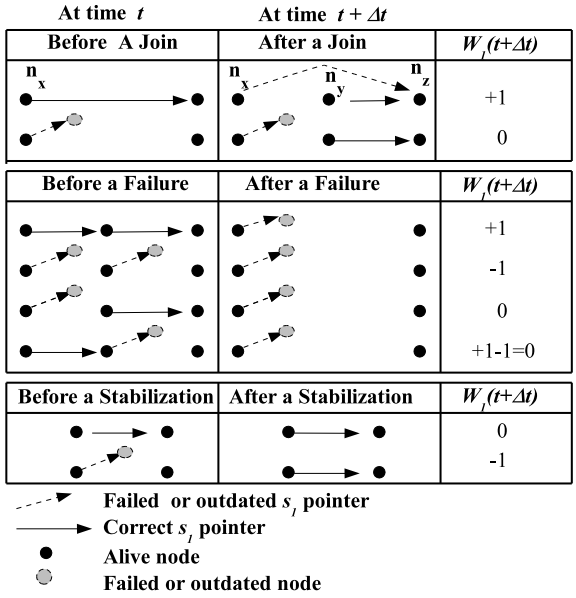


Figure 9.2: Changes in W_1, the number of wrong (failed or outdated) s_1 pointers, due to joins, failures and stabilizations.

w_1. From this follows the term $c_{2.4}$.

Hence the equation for $W_1(r, \alpha)$ is:

$$\frac{dW_1}{dt} = \lambda_j(1 - w_1) + \lambda_f(1 - w_1)^2 - \lambda_f w_1^2 - \alpha\lambda_s w_1$$

Solving for w_1 in the steady state and putting $\lambda_j = \lambda_f$, we get:

$$w_1(r, \alpha) = \frac{2}{3 + r\alpha} \approx \frac{2}{r\alpha} \tag{9.5}$$

This expression matches well with the simulation results as shown in Fig. 9.3. $d_1(r, \alpha)$ is then $\approx \frac{1}{2}w_1(r, \alpha)$ since when $\lambda_j = \lambda_f$, about half the number of wrong pointers are incorrect and about half point to dead nodes. Thus $d_1(r, \alpha) \approx \frac{1}{r\alpha}$ which also matches the simulations well as shown in Fig. 9.3. We can also use the above reasoning to iteratively get $w_k(r, \alpha)$ for any k.

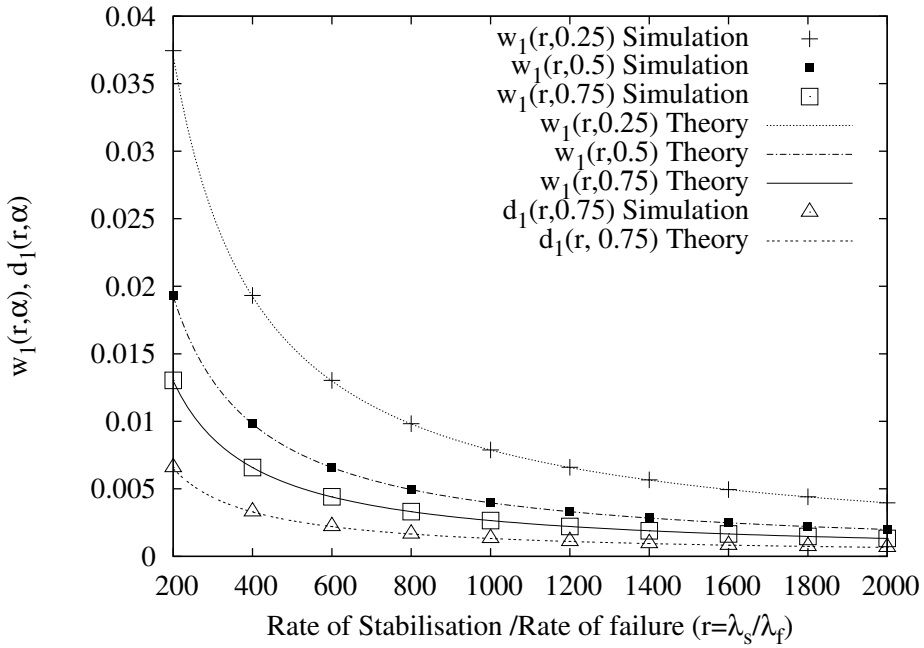


Figure 9.3: Theory and Simulation for $w_1(r, \alpha)$, $d_1(r, \alpha)$

9.4.3 Break-up (Network Disconnection) Probability

We demonstrate below, how calculating $d_k(r, \alpha)$: the fraction of nodes with dead k^{th} pointers, helps in estimating precisely the probability that the network gets disconnected for any value of r and α. Let $P_{bu}(n, r, \alpha)$ be the probability that n consecutive nodes fail. If $n = \mathcal{S}$, the length of the successor list, then clearly the node gets disconnected from the network and the network breaks up. For the range of r considered in Fig. 9.3, $P_{bu}(\mathcal{S}, r, \alpha) \sim 0$. However should we go lower, this starts becoming finite. The master equation analysis introduced here can be used to estimate $P_{bu}(n, r, \alpha)$ for any $1 \leq n \leq \mathcal{S}$. We indicate how this might be done by considering the case $n = 2$. Let $N_{bu}(2, r, \alpha)$ be the number of configurations in which a node has both s_1 and s_2 dead and $P_{bu}(2, r, \alpha)$ be the fraction of such configurations. Table 9.3 indicates how this is estimated

Change in $W_1(r, \alpha)$	Rate of Change
$N_{bu}(t + \Delta t) = N_{bu}(t) + 1$	$c_{3.1} = (\lambda_f \Delta t) d_1(r, \alpha)$
$N_{bu}(t + \Delta t) = N_{bu}(t) + 1$	$c_{3.2} = \lambda_f \Delta t (1 - d_1) d_2$
$N_{bu}(t + \Delta t) = N_{bu}(t) - 1$	$c_{3.3} = \alpha \lambda_s \Delta t P_{bu}(2, r, \alpha)$
$N_{bu}(t + \Delta t) = N_{bu}(t)$	$1 - (c_{3.1} + c_{3.2} + c_{3.3})$

Table 9.3: Gain and loss terms for $N_{bu}(2, r, \alpha)$: the number of nodes with dead first *and* second successors

within the present framework.

A join event does not affect this probability in any way. So we need only consider the effect of failures or stabilization events. The term $c_{3.1}$ accounts for the situation when the *first* successor of a node is dead (which happens with probability $d_1(r, \alpha)$ as explained above). A failure event can then kill its second successor as well and this happens with probability $c_{3.1}$. The second term is the situation that the first successor is alive (with probability $1 - d_1$) but the second successor is dead (with probability d_2). This probability is $\sim 2/\alpha r$ (the second successor of a node being dead either implies that the first successor of *its* first successor is dead with probability d_1, or that it has not stabilized recently, and hence has not corrected its second successor pointer.This happens with probability $\sim 1/\alpha r$. These two terms add up to $2/\alpha r$). A stabilization event reduces the number of such configurations by one, if the node doing the stabilization had such a configuration to begin with.

Solving the equation for $N_{bu}(2, r, \alpha)$, one hence obtains that $P_{bu}(2, r, \alpha) \sim 3/(\alpha r)^2$. As Fig. 9.4 shows, this is a precise estimate.

We can similarly estimate the probabilities for three consecutive nodes failing, *etc*, and hence also the disconnection probability $P_{bu}(\mathcal{S}, r, \alpha)$. This formalism thus affords the possibility of making a precise prediction for when the system runs the danger of getting disconnected as a function of the parameters.

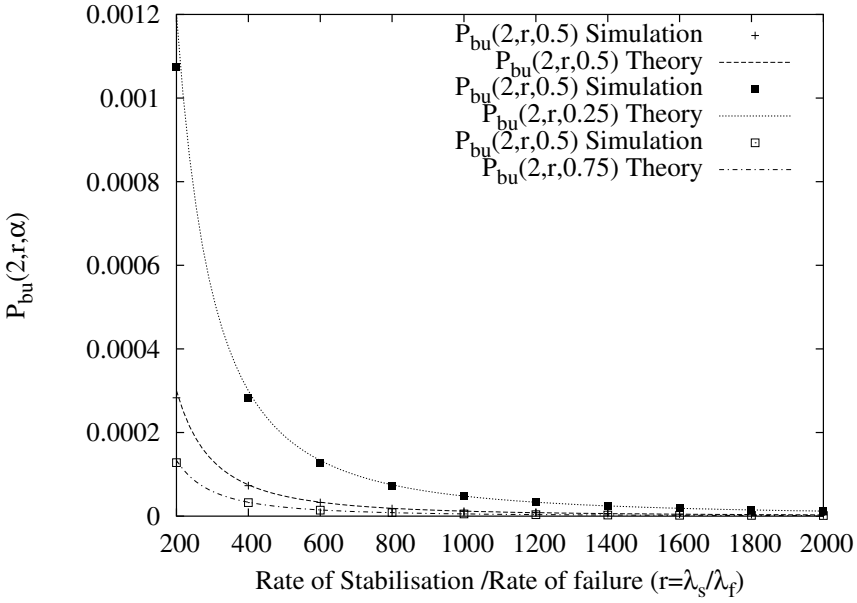


Figure 9.4: Theory and Simulation for $P_{bu}(2, r, \alpha)$

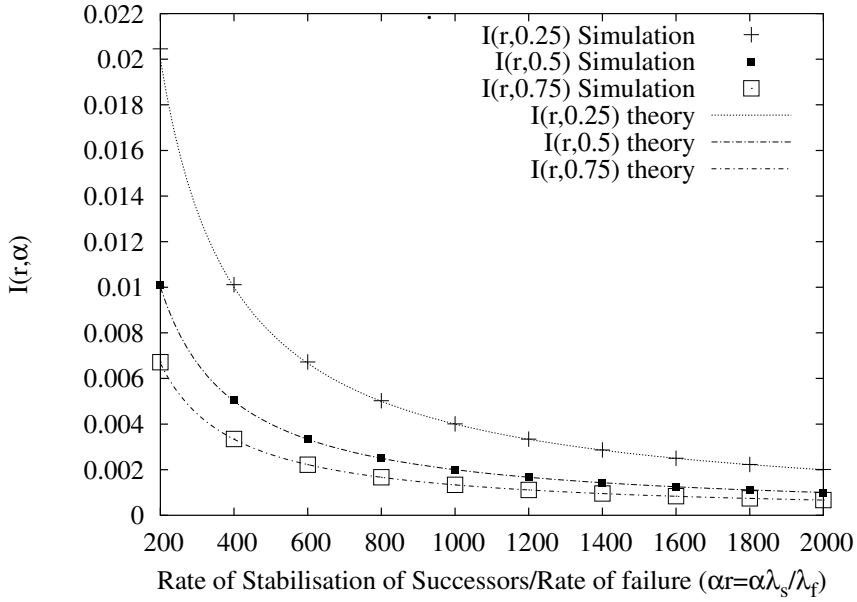


Figure 9.5: Theory and Simulation for $I(r, \alpha)$

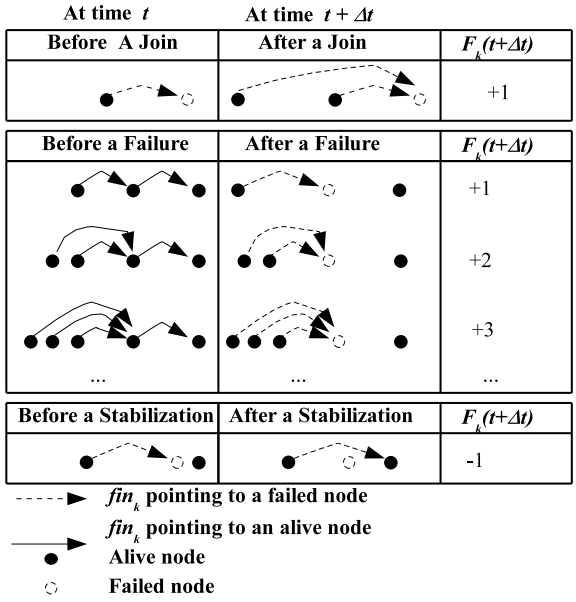


Figure 9.6: Changes in F_k, the number of failed fin_k pointers, due to joins, failures and stabilizations.

9.4.4 Lookup Consistency

By the lookup protocol, a lookup is inconsistent if the immediate predecessor of the sought key has a wrong s_1 pointer. However, we need only consider the case when the s_1 pointer is pointing to an alive (but incorrect) node since our implementation of the protocol always requires the lookup to return an alive node as an answer to the query. The probability that a lookup is inconsistent $I(r, \alpha)$ is hence $w_1(r, \alpha) - d_1(r, \alpha)$. This prediction matches the simulation results very well, as shown in Fig. 9.5.

9.4.5 Failure of Fingers

We now turn to estimating the fraction of finger pointers which point to failed nodes. As we will see this is an important quantity for predicting lookups, since failed fingers cause timeouts and increase the lookup length. We need however only consider fingers pointing to *dead* nodes.

$F_k(t+\Delta t)$	Rate of Change
$= F_k(t) + 1$	$c_{4.1} = (\lambda_j \Delta t) \sum_{i=1}^{k} p_{join}(i,k) f_i$
$= F_k(t) - 1$	$c_{4.2} = (1-\alpha)\frac{1}{\mathcal{M}} f_k(\lambda_s \Delta t)$
$= F_k(t) + 1$	$c_{4.3} = (1-f_k)^2[1-p_1(k)](\lambda_f \Delta t)$
$= F_k(t) + 2$	$c_{4.4} = (1-f_k)^2(p_1(k)-p_2(k))(\lambda_f \Delta t)$
$= F_k(t) + 3$	$c_{4.5} = (1-f_k)^2(p_2(k)-p_3(k))(\lambda_f \Delta t)$
$= F_k(t)$	$1 - (c_{4.1} + c_{4.2} + c_{4.3} + c_{4.4} + c_{4.5})$

Table 9.4: Some of the relevant gain and loss terms for F_k, the number of nodes whose k^{th} fingers are pointing to a failed node for $k > 1$.

Unlike members of the successor list, *alive* fingers even if outdated, always bring a query closer to the destination and do not affect consistency or substantially even the lookup length. Therefore we consider fingers in only two states, alive or dead (failed). By our implementation of the stabilization protocol (see Appendix A), fingers and successors are stabilized entirely independently of each other. Thus even though the first finger is also always the first successor, this information is not used by the node in updating the finger.

Let $f_k(r, \alpha)$ denote the fraction of nodes having their k^{th} finger pointing to a failed node and $F_k(r, \alpha)$ denote the respective number. For notational simplicity, we write these as simply F_k and f_k. We can predict this function for any k by again estimating the gain and loss terms for this quantity, caused by a join, failure or stabilization event, and keeping only the most relevant terms. These are listed in table 9.4 and illustrated in Fig. 9.6

A join event can play a role here by increasing the number of F_k pointers if the successor of the joinee had a failed i^{th} pointer (occurs with probability f_i) and the joinee replicated this from the successor as the joinee's kth pointer (occurs with probability $p_{join}(i,k)$ from property 9.4.3). For large enough k, this probability is one only for $p_{join}(k,k)$, that is the new joinee mostly only replicates the successor's kth pointer as its own k^{th} pointer. This is what we consider here.

A stabilization evicts a failed pointer if there was one to begin with. The stabilization rate is divided by $\mathcal{M}$, since a node stabilizes any one

finger randomly, every time it decides to stabilize a finger at rate $(1-\alpha)\lambda_s$.

Given a node n with an alive k^{th} finger (occurs with probability $1-f_k$), when the node pointed to by that finger fails, the number of failed k^{th} fingers (F_k) increases. The amount of this increase depends on the number of immediate predecessors of n that were pointing to the failed node with their k^{th} finger. That number of predecessors could be 0, 1, 2,.. etc. Using property 9.4.2 the respective probabilities of those cases are: $1 - p_1(k)$, $p_1(k) - p_2(k)$, $p_2(k) - p_3(k)$,... etc.

Solving for f_k in the steady state, we get:

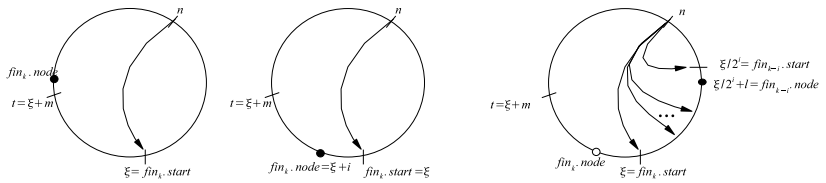
$$f_k = \frac{\left[2\tilde{P}_{rep}(k) + 2 - p_{\text{join}}(k,k) + \frac{r(1-\alpha)}{\mathcal{M}}\right]}{2(1+\tilde{P}_{rep}(k))} - \frac{\sqrt{\left[2\tilde{P}_{rep}(k) + 2 - p_{\text{join}}(k,k) + \frac{r(1-\alpha)}{\mathcal{M}}\right]^2 - 4(1+\tilde{P}_{rep}(k))^2}}{2(1+\tilde{P}_{rep}(k))} \tag{9.6}$$

where $\tilde{P}_{rep}(k) = \Sigma p_i(k)$. In principle its enough to keep even three terms in the sum. The above expressions match very well with the simulation results (Fig. 9.8).

9.4.6 Cost of Finger Stabilizations and Lookups

In this section, we demonstrate how the information about the failed fingers and successors can be used to predict the cost of stabilizations, lookups or in general the cost for reaching any key in the id space. By cost we mean the number of hops needed to reach the destination *including* the number of timeouts encountered en-route. Timeouts occur every time a query is passed to a dead node. The node does not answer and the originator of the query has to use another finger instead. For this analysis, we consider timeouts and hops to add equally to the cost. We can easily generalize this analysis to investigate the case when a timeout costs some factor n times the cost of a hop.

Define $C_t(r,\alpha)$ (also denoted C_t) to be the expected cost for a given node to reach some target key which is t keys away from it (which means reaching the first successor of this key). For example, C_1 would then be the



	(a)	(b)	(c)
Prob.	$1-a(m)$	$a(m)(1-f_k)$	$a(m)f_k h_k(i)bc(l,\xi/2^i)$
Cost $(C_t=C_{\xi+m})$	$C_{\xi+m}=C_\xi$	$C_{\xi+m}=1+bc(i,m)C_{m-i}$ $0\le i<m$	$C_{\xi+m}=1+1+C_{\xi_i-l+m}+(i-1)$ $1\le i\le k-1, 1\le l<\xi/2^i$

Figure 9.7: Cases that a lookup can encounter with the respective probabilities and costs.

cost of looking up the adjacent key (1 key away). Since the adjacent key is always stored at the first alive successor, therefore if the first successor is alive (which occurs with probability $1 - d_1$), the cost will be 1 hop. If the first successor is dead but the second is alive (occurs with probability $d_1(1 - d_2)$), the cost will be 1 hop + 1 timeout = 2 and the *expected* cost is $2 \times d_1(1 - d_2)$ and so forth. Therefore, we have $C_1 = 1 - d_1 + 2 \times d_1(1 - d_2) + 3 \times d_1 d_2(1 - d_3) + \cdots \approx 1 + d_1 = 1 + 1/(\alpha r)$.

For finding the expected cost of reaching a general distance t we need to follow closely the Chord protocol, which would lookup t by first finding the closest preceding finger. For the purposes of the analysis, we will find it easier to think in terms of the closest preceding *start*. Let us hence define ξ to be the *s*tart of the finger (say the k^{th}) that most closely precedes t. Hence $\xi = 2^{k-1} + n$ and $t = \xi + m$, i.e. there are m keys between the sought target t and the start of the most closely preceding finger. With that, we can write a recursion relation for $C_{\xi+m}$ as follows:

$$
\begin{aligned}
C_{\xi+m} &= C_{\xi}\left[1-a(m)\right] \\
&+ (1-f_k)a(m)\left[1+\sum_{i=0}^{m-1} bc(i,m)C_{m-i}\right] \\
&+ f_k a(m)\Bigg[1+\sum_{i=1}^{k-1} h_k(i) \\
&\sum_{l=0}^{\xi/2^i-1} bc(l,\xi/2^i)(1+(i-1)+C_{\xi_i-l+m})+O(h_k(k))\Bigg]
\end{aligned}
\tag{9.7}
$$

where $\xi_i \equiv \sum_{m=1,i} \xi/2^m$ and $h_k(i)$ is the probability that a node is forced to use its $k-i^{th}$ finger owing to the death of its k^{th} finger. The probabilities a, b, bc have already been introduced in section 4, and we define the probability $h_k(i)$ below.

The lookup equation though rather complicated at first sight merely accounts for all the possibilities that a Chord lookup will encounter, and deals with them exactly as the protocol dictates.

The first term (Fig. 9.7 (a)) accounts for the eventuality that there is no node intervening between ξ and $\xi+m$ (occurs with probability $1-a(m)$). In this case, the cost of looking for $\xi+m$ is the same as the cost for looking for ξ.

The second term (Fig. 9.7 (b)) accounts for the situation when a node does intervene inbetween (with probability $a(m)$), and this node is alive (with probability $1-f_k$). Then the query is passed on to this node (with 1 added to register the increase in the number of hops) and then the cost depends on the length of the distance between this node and t.

The third term (Fig. 9.7 (c)) accounts for the case when the intervening node is dead (with probability f_k). Then the cost increases by 1 (for a timeout) and the query needs to find an alternative lower finger that most closely precedes the target. Let the $k-i^{th}$ finger (for some i, $1 \leq i \leq k-1$) be such a finger. This happens with probability $h_k(i)$, i.e., the probability that the lookup is passed back to the $k-i^{th}$ finger either because the intervening fingers are dead or share the same finger table entry as the k^{th} finger is denoted by $h_k(i)$. The start of the $k-i^{th}$ finger is at $\xi/2^i$ and the

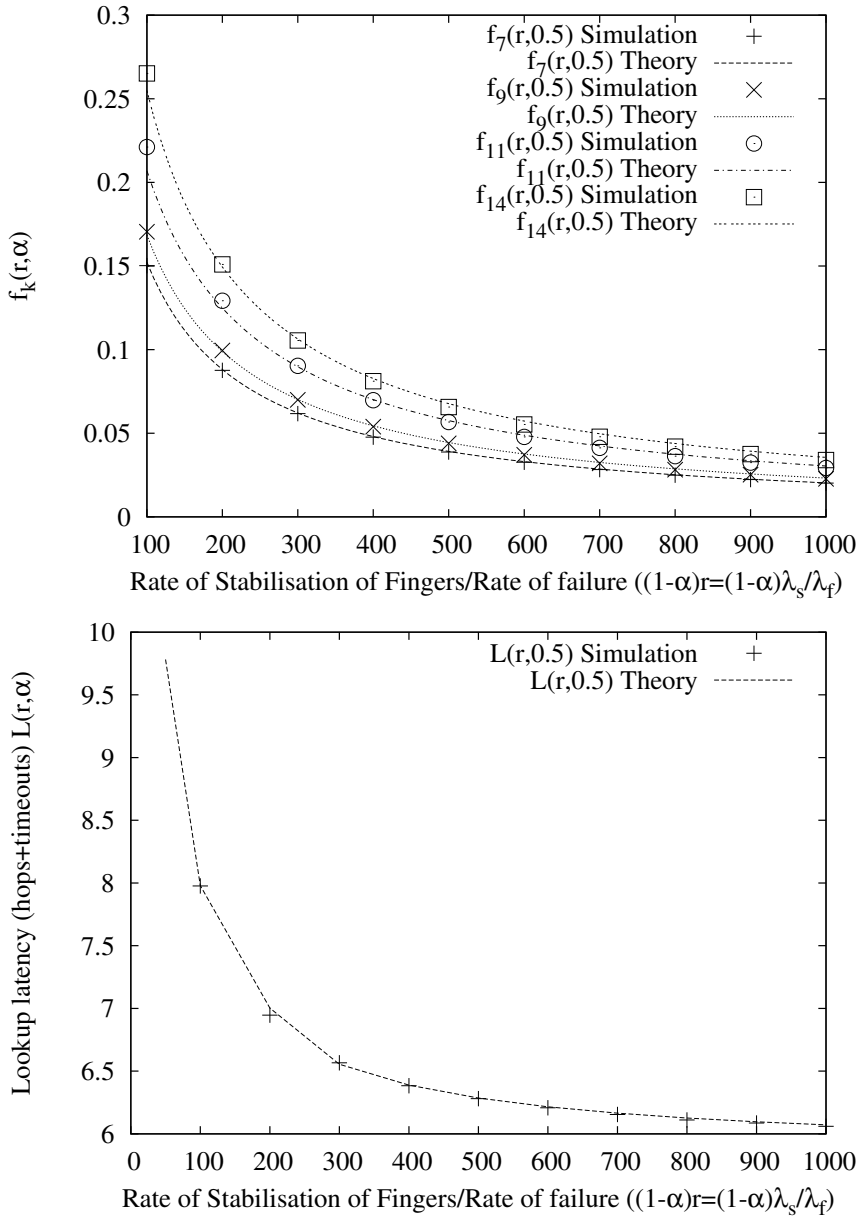


Figure 9.8: Theory and Simulation for $f_k(r,\alpha)$, and $L(r,\alpha)$

distance between $\xi/2^i$ and ξ is equal to $\sum_{m=1,i} \xi/2^m$ which we denote by ξ_i. Therefore, the distance from the *start* of the $k - i^{th}$ to the target is equal to $\xi_i + m$. However, note that $fin_{k-i}.node$ could be l keys away (with probability $bc(l, \xi/2^i)$) from $fin_{k-i}.start$ (for some l, $0 \leq l < \xi/2^i$). Therefore, after making one hop to $fin_{k-i}.node$, the remaining distance to the target is $\xi_i + m - l$. The increase in cost for this operation is $1 + (i - 1)$; the 1 indicates the cost of taking up the query again by $fin_{k-i}.node$, and the $i - 1$ indicates the cost for trying and discarding each of the $i - 1$ intervening nodes. The probability $h_k(i)$ is easy to compute given property 9.4.1 and the expression for the f_k's computed in the previous section.

$$\begin{aligned} h_k(i) =& a(\xi/2^i)(1 - f_{k-i}) \\ & \times \Pi_{s=1,i-1}(1 - a(\xi/2^s) + a(\xi/2^s) f_{k-s}), i < k \\ h_k(k) =& \Pi_{s=1,k-1}(1 - a(\xi/2^s) + a(\xi/2^s) f_{k-s}) \end{aligned} \tag{9.8}$$

Eqn.9.8 accounts for all the reasons that a node may have to use its $k - i^{th}$ finger instead of its k^{th} finger. This could happen because the intervening fingers were either dead or not distinct. The probabilities $h_k(i)$ satisfy the constraint $\sum_{i=1}^{k} h_k(i) = 1$ since clearly, either a node uses any one of its fingers or it doesn't. This latter probability is $h_k(k)$, that is the probability that a node cannot use any earlier entry in its finger table. In this case, n proceeds to its successor list. The query is now passed on to the first alive successor and the new cost is a function of the distance of this node from the target t. We indicate this case by the last term in Eq. 9.7 which is $O(h_k(k))$. This can again be computed from the inter-node distribution and from the functions $d_k(r, \alpha)$ computed earlier. However in practice, the probability for this is extremely small except for targets very close to n. Hence this does not significantly affect the value of general lookups and we ignore it for the moment.

The cost for general lookups is hence

$$L(r, \alpha) = \frac{\Sigma_{i=1}^{\mathcal{K}-1} C_i(r, \alpha)}{\mathcal{K}}$$

The lookup equation is solved recursively, given the coefficients and C_1. We plot the result in Fig 9.8. The theoretical result matches the simulation very well.

9.5 What is Churn?

We now discuss a broader issue, connected with churn, which arises naturally in the context of our analysis. As we mentioned earlier, all our analysis is performed in the steady state where the rate of joins (λ_j) is equal to the rate of failures λ_f. However the rates λ_j and λ_f can themselves each be chosen in one of two different ways. They could either be "per-network" or "per-node". In the former case, the number of joinees (or the number of failures) *does not* depend on the current number of nodes in the network. This is the case when a poisson model is considered either for arrivals or departures. Put in another way, this is like saying that on average, there is always a fixed number of nodes joining or failing per time interval, irrespective of the total number of nodes in the network. In the case when these rates are chosen to be per-node, the number of joinees or failures *does* depend on the current number of occupied nodes). We consider three possibilities here, when λ_j is per-network and λ_f is per-node; both are per-network or (as is the case studied in this paper) both are per-node. In all three cases, since the system is always studied in the steady state where the total number of joinees per unit time is equal to the total number of failures per unit time, the equation for the mean is always $dN/dt = 0$. We hence expect the mean behaviour to be the same, at least in the regime when N is roughly constant. However the behaviour of fluctuations is very different in each of these three cases.

In the first case, the steady state condition is $\lambda_j/N_o = \lambda_f$, where N_o is the initial number of nodes in the system. The equation for the mean is $dN/dt = \lambda_j/N - \lambda_f$, which ensures that N cannot deviate too much from the steady state value. Similarly one can write an equation for the second moment N^2: $dN^2/dt = (\lambda_j/N + \lambda_f) + 2(\lambda_j - N\lambda_f)$. While the first term is a 'noise' term which encourages fluctuations, the second term becomes stronger the larger the deviation from N_o and hence strongly damps out fluctuations. Thus the number of nodes in the system remains close to its initial value.

In the second case, where the join and failure rates are both per-network the equation for the mean is $dN/dt = \lambda_j/N - \lambda_f/N$. Hence putting $\lambda_j = \lambda_f$ ensures the steady state condition. However in this case, the equation for

the second moment is $dN^2/dt = (\lambda_j/N + \lambda_f/N)$. The joins-failures process thus makes the system execute a "random-walk" in N, where the "steps" of the walk depend on N and are smaller if N is larger. For such a system, fluctuations are not bounded and a large deviation can and will take the system to the $N = 0$ state eventually. The time for this to happen scales with N as N^3 for this process.

The third case (which is also the case considered in this paper) is when both rates are per-node. This is very similar to the second case. The equation for the mean is just $dN/dt = \lambda_j - \lambda_f$ as mentioned earlier. Again setting $\lambda_j = \lambda_f$ ensures steady state. The equation for the second moment is now $dN^2/dt = (\lambda_j + \lambda_f)$. There is thus again no "repair" mechanism for large fluctuations, and the system will be eventually driven to extinction. In this case the process on N is just an ordinary random walk and the time taken to hit the $N = 0$ state scales as N^2.

Which of these 'types' of churn is the most relevant? In the real world, the churn felt by a DHT, might possibly be some time-varying mixture of these three, and will also possibly depend on the application. It is hence probably of importance to study all these mechanisms and their implications in detail.

9.6 Discussion and Conclusion

To summarize, in this paper, we have presented a detailed theoretical analysis of a DHT-based P2P system, Chord, using a Master-equation formalism. This analysis differs from existing theoretical work done on DHTs in that it aims not at establishing bounds, but on precise determination of the relevant quantities in this dynamically evolving system. From the match of our theory and the simulations, it can be seen that we can predict with an accuracy of greater than 1% in most cases.

Though this analysis is not *exact* (in the sense that there are approximations made to make the analysis simpler), yet it provides a methodology to keep track of most of the relevant details of the system. We expect that the same analysis can be done for most other DHTs in a similar manner, thus helping to establish quantitative guidelines for their comparison.

Apart from the usefulness of this approach for its own sake, we can

also gain some new insights into the system from it. For example, we see that the fraction of dead finger pointers f_k is an increasing function of the length of the finger. Infact for large enough $\mathcal{K}$, all the long fingers will be dead most of the time, making routing very inefficient. This implies that we need to consider a different stabilization scheme for the fingers (such as, perhaps, stabilizing the longer fingers more often than the smaller ones), in order that the DHT continues to function at high churn rates.

Bibliography

[1] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. Efficient, self-contained handling of identity in peer-to-peer systems. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):858–869, 2004.

[2] James Aspnes, Zoë Diamadi, and Gauri Shah. Fault-tolerant routing in peer-to-peer systems. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 223–232. ACM Press, 2002.

[3] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*. IEEE Computer Society, 2004.

[4] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. A statistical theory of chord under churn. In *The 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Ithaca, New York, February 2005.

[5] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *The 3rd International Workshop on Peer-to-Peer Systems (IPTPS'02)*, San Diego, CA, Feb 2004.

[6] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *ACM Conf. on Principles of Distributed Computing (PODC)*, Monterey, CA, July 2002.

[7] N.G. van Kampen. *Stochastic Processes in Physics and Chemistry*. North-Holland Publishing Company, 1981. ISBN-0-444-86200-5.

[8] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Annual Technical Conference(USENIX '04)*, Boston, Massachusetts, USA, June 2004.

[9] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, 2003.

[10] Shengquan Wang, Dong Xuan, and Wei Zhao. On resilience of structured peer-to-peer systems. In *GLOBECOM 2003 - IEEE Global Telecommunications Conference*, pages 3851–3856, Dec 2003.

A Our Implementation of Chord

A.1 Joins, Failures & Ring Stablization

Initialization. Initially, the predecessor p, successors ($s_{1..\mathcal{S}}$) and fingers ($fin_{1..\mathcal{M}}$) are all assigned to nil.

Joins (Fig. 9.9). A new node n joins by acquiring its successor from an initial random contact node c. It also starts its first stabilization of the successors and initializes its fingers.

Stablization of Sucessors (Fig. 9.9). The function *fixSuccessors* is triggered periodically with rate $\alpha\lambda_s$. A node n tells its first alive successor y that it believes itself to be y's predecessor and expects as an answer y's predecessor $y.p$ and successors $y.s$. The response of y can lead to three actions:
Case A. Some node exists between n and y (i.e. n's belief is wrong), so n prepends $y.p$ it to its successors list as a first successor and retries *fixSuccessors*.
Case B. y confirms n's belief and informs n of y's old predecessor $y.p$. Therefore n considers $y.p$ as an alternative/initial predecessor for n. Finally, n reconciles its successors list with $y.s$.
Case C. y agrees that n is its predecessor and the only task of n is to update its successors list by reconciling it with $y.s$.

By calling *iThinkIamYourPred* (Fig. 9.9), some node x informs n that it believes itself to be n's predecessor. If n's predecessor p is not alive or nil, then n accepts x as a predecessor and informs x about this agreement by returning x. Alternatively, if n's predecessor p is alive (discovering that will be explained shortly in section A.3), then there are two possibilities: The first is that x is in the region between n and its current predecessor p therefore n should accept x as a new predecessor and inform x about its old predecessor. The second is that p is already pointing to x so the state is correct at both parties and n confirms that to x by informing it that x is the predecessor of n. In all cases the function returns a predecessor and a successors list.

The function *firstAliveSuccessor* (Fig. 9.9) iterates through the successors list. In each iteration, if the first successor s_1 is alive, it is returned. Otherwise, the dead successor is dropped from the list and nil is appended

to the end of the list. If the first successor is *nil* this means that all immediate successors are dead and that the ring is disconnected.

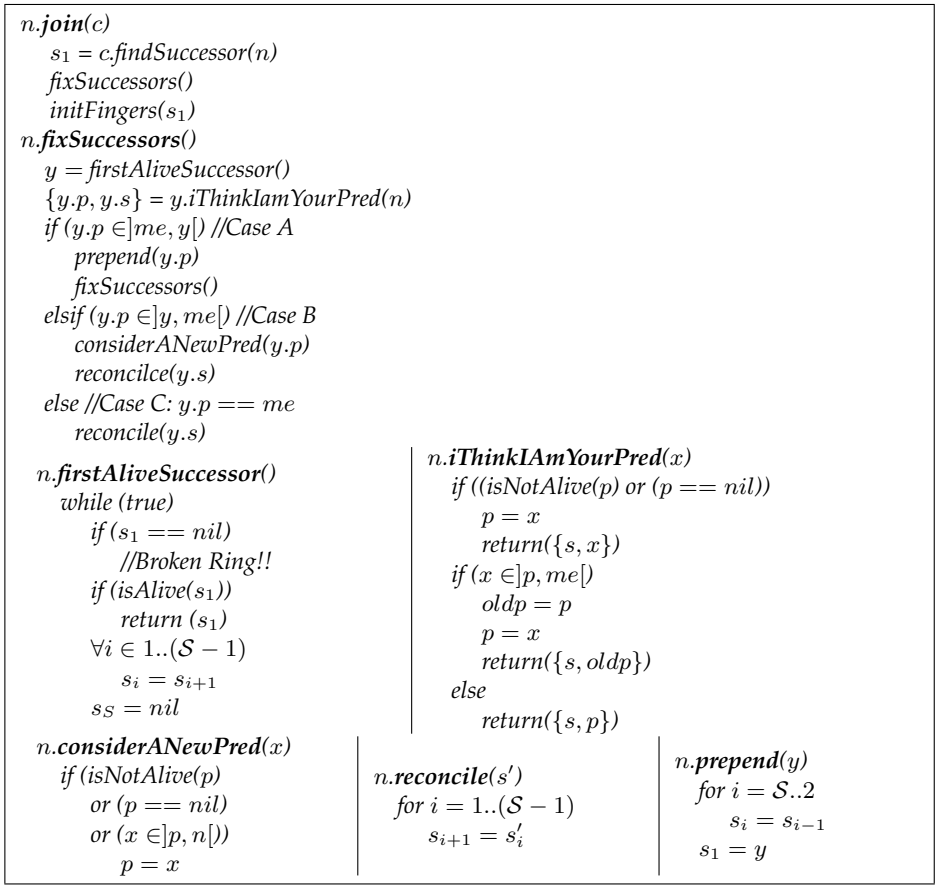


Figure 9.9: Joins and Ring Stabilization Algorithms

```
n.initFingers(s_1)
    f' = s_1.f
    ∀i ∈ 1..M s.th. (fin_i.start ∈]n, s_1]),
        fin_i.node = s_1
    ∀j ∈ 1..M s.th. (fin_j.start ∉]n, s_1]),
        fin_j.node =localSuccessor(f', fin_j.start)
n.localSuccessor(f,k)
    for i = 1..M
        if (k ∈]n, fin_i])
            return(fin_i)
    return(nil)

n.fixFingers(k)
    1 ≤ i = random() ≤ M
    fin_i.node = findSuccessor(fin_i.start)
```

Figure 9.10: Initialization and Stabilization of Fingers

A.2 Lookups and Stablization of Fingers

Stablization of Fingers (Fig. 9.10). Stabilization of fingers occurs at a rate $(1-\alpha)\lambda_s$. Each time the *fixFingers* function is triggered, a random finger fin_i is chosen and a lookup for $fin_i.start$ is performed and the result is used to update $fin_i.node$.

Initialization of Fingers (Fig. 9.10). After having initialized its first successor s_1, a node n sets all fingers with starts between n and s_1 to s_1. The rest of the fingers are initialized by taking a copy of the finger table of s_1 and finding an approximate successor to every finger from that finger table.

Lookups (Fig. 9.11). A lookup operation is a fundamental operation that is used to find the successor of a key. It is used by many other routines and its performance and consistency are the main quantities of interest in the evaluation of any DHT. A node n looking up the successor of k runs the *findSuccessor* algorithm which can lead to the following cases:

Case A. If k is equal to n then n is trivially the successor of k.

Case B. If $k \in]n, s_1]$ then n has found the successor of k, but it could be that s_1 failed and n did not discover that as yet. However, entries in the successors list can act as backups for the first successor. Therefore, the first alive successor of n is the successor of k. Note that, in this case, while we try to find the first alive successor, we do not change the entries in the successors list. This is mainly because, for the sake of the analysis, we

want that the successor list is only changed at rate $\alpha\lambda_s$ by the *fixSuccessors* function and is not affected by any other rate.

```
n.findSuccessor(k)
  //Case A: k is exactly equal to n
  if (k == n)
    return(n)
  //Case B: k is between n and s_1
  if (k ∈]n, s_1])
    return(firstAliveSuccessorNoChange());
  //Case C: Forward to the lookup to
  //the closest preceding alive finger
  cpf = closestAlivePrecedingFinger(k);
  if (cpf == nil)
    y = firstAliveSuccessorNoChange();
    if (k ∈]n, y])
      return(y);
    cpf = closestAlivePrecedingSucc(k);
    return(cpf.findSuccessor(k))
  else
    return (cpf.findSuccessor(k));
n.firstAliveSuccessorNoChange()
  i = 1
  while (true)
    if (s_i == nil)
      //Broken Ring!!
    if (isAlive(s_i))
      return (s_i)
    i++
n.closestAlivePrecedingFinger(k)
  for i = M..1
    if ((fin_i ∈]n, k[)
      and (fin_i ≠ nil)
      and isAlive(fin_i))
        return(fin_i)
  return(nil)
n.closestAlivePrecedingSucc(k)
  for i = S..1
    if ((s_i ∈]n, k[)
      and (s_i ≠ nil)
      and isAlive(s_i))
        return(s_i)
  return(cpf)
```

Figure 9.11: The Lookup Algorithm

Case C. The lookup should be forwarded to a node closer to k, namely the closest alive finger preceding k in n's finger table. The call to the function *closestAlivePrecedingFinger* returns such a node if possible and the lookup is forwarded to it. However, it could be the case that all alive preceding fingers to k are dead. In that case, we need to use the successors list as a last resort for the lookup. Therefore, we locate the first alive successor y and if $k \in]n, y]$ then y is the successor of k. Otherwise, we locate the closest alive preceding successor to k and forward the lookup to it.

A.3 Failures

Throughout the code we use the call $isAlive$ and $isNotAlive$. A simple interpretation of those routines would be to equate them to a performance of a ping. However, a correct implementation for them is that they are discovered by performing the operation required. For instance, a call to $firstAliveSuccesor$ in Fig. 9.9 is performed to retrieve a node y and then call $y.iThinkIamYourPred$, so alternatively the first alive successor could be discovered by iterating on the successor list and calling $iThinkIamYourPred$.

Chapter 10

Conclusions and Future work

Since this thesis is of the "collection-of-papers" type, there is no one single thesis statement that we would like to conclude with. Instead, for each of the two parts of the thesis, we will put the conclusion in the form of an anecdotal summary of how the research questions evolved leading to the results reported in the respective part.

1 Conclusion of Part I : *Designs*

The idea of a common framework for structured overlays started to emerge after a first reading of some of the early systems in the logarithmic class such as Chord, Pastry and Tapestry. It appeared to us that there was some implicit common concept that stands behind those systems. We noticed that the concept of a virtual distributed search tree of arbitrary arity is expressive enough to stand as a common foundation. By applying our observation to Chord, we obtained a more optimal routing table size which is 38% smaller than the generalization suggested by the Chord authors.

This optimization led to the design of the $\mathcal{DKS}(N; k; f)$ system, which is a system designed from first principles based on the Distributed k-ary Search concept. The system acts as a meta-system from which other DHT systems could be instantiated. Additionally, the system introduces the technique of "Correction-on-use" for topology maintenance. Instead of performing periodic checks to make sure that the routing information is

up-to-date, a node utilizes lookup traffic to extract information about the correctness of the routing information and corrects itself and the sender's information accordingly, thus, saving a large amount of unnecessary network traffic. We have shown via simulation that if enough queries were taking place, they could provide enough correction without the need for additional dedicated topology maintenance traffic.

Performing a global operation is another rather difficult task in decentralized systems because each peer in the network knows a small set of neighbors. If a global operation like counting the number of peers, or the dissemination of statistical information is desired, a naive epidemic strategy will end up being very costly. Having perceived a structured overlay as a search tree, traversing it optimally seemed like a feasible approach to perform a global operation. Therefore, we designed an optimal algorithm for constructing a spanning tree for chord/$\mathcal{DKS}$. While the presentation of the algorithm is based on the Chord/$\mathcal{DKS}$ system, the idea is also applicable to other DHTs such as Tapestry and Pastry. The spanning tree algorithm is important for optimal broadcasting which is a basic form of group communication and a useful service for performing complex queries rather than key lookup.

Finally, we combined the broadcast algorithm with the correction-on-use technique. This combination serves two purposes. First, it adds robustness to the broadcast algorithm. Second, it makes the broadcast algorithm itself a tool for mass correction of the overlay graph.

2 Conclusion of Part II : *Analyses*

As we can see from chapter 2, the asymptotic limits of the performance of structured overlays are well-known based on the underlying static graph topology. However, for practical purposes, the deployment of a particular system needs more than the knowledge of the asymptotic limits. For instance, if we were to connect 2^x computers in a structured overlay, and the overlay was structurally perfect, it would not take more than x hops ($O(\log 2^x)$) to reach any other computer. However, with a high rate of dynamic membership (churn), the structure of the overlay is suboptimal. For a certain rate of that churn (say λ_j nodes per minute) and a certain rate of

periodic stabilization (say λ_s per minute per node), the computers might easily take $2x$ or $4x$ hops instead of x on average to communicate. This is still $O(\log 2^x)$, but it is a huge difference in the practicality of the system. More importantly, it is possible to bring back this network to optimal performance if the nodes were stabilizing a little bit more frequently. From that point stemmed the main question of our "Analyses":

> How *much* churn against how *much* topology maintenance, will lead to how *good* system performance?

The above question actually implicitly includes in it the following questions: "*For which system?*" and "*Under which topology maintenance strategy?*". Not only that, but, for the same system and with the same maintenance strategy, there are subtle variations that affect the performance dramatically. Having realized that, we decided to trade-off generality for depth and started to focus on one system: "Chord", (i.e. the special case for the $\mathcal{DKS}$ system with binary arity) and one topology maintenance strategy, namely, periodic stabilization.

We began first with an empirical physics-inspired performance analysis. The physics inspiration lies in trying to describe the behavior using an "intensive variable", i.e. one that does not depend on the size of the system. The result of this analysis was a size-independent curve showing the performance degradation of the system as a function of the ratio of churn to stabilization (β) obtained after extensive simulations of networks of different sizes under many conditions. The size independence of the curve renders the analysis widely-applicable. That is, one can know that the performance will degrade by, for instance, 60% if the ratio of stabilization to churn is for instance 30, (i.e. 30 stabilizations per node for each join/leave leave per node) and that degradation is irrespective of whether the system contains one thousand nodes or one million nodes.

After obtaining results empirically, we were faced with the following theoretical challenge: "*Why does the degradation curve look particularly the way it is?*", i.e. "*What is its functional form?*". Beyond a curve fit, " *Is it possible to analytically derive its functional form.*". At that point, another physics-borrowed technique was useful, namely, the technique of "Master Equations". With that technique, for a given quantity of interest, one needs

to enumerate all the events that lead to its increase and decrease over time. With the knowledge of the probability of occurrence of each such event, the formulation of a differential equation is made possible. For our analysis, we started from the simplest system, a ring of nodes that are periodically stabilizing to keep the ring intact. We incrementally enriched the system until we analyzed a full-version of chord. At that level, a precise functional form of the performance degradation curve was at hand. The output of this theoretical analysis was validated against simulations and was found to match to a very high degree of accuracy. The importance of the result is not limited to the explanation of the performance degradation of Chord/$\mathcal{DKS}$, but the derivation of that curve from first principles acts as a tutorial in how to analyze problems in the computer science domain using the technique of Master Equations.

3 Future Work

At the time of writing of this thesis, our research team has the intention to pursue a number of research points mainly in the line of analysis based on the Master Equations technique including the following:

- Using the functional forms obtained so far in developing adaptive algorithms that can follow the optimal cost-performance curve.
- Analyzing various correction techniques other than periodic stabilization, namely correction-on-use and correction on change.
- Augmenting the cost-performance trade-off analysis with physical link latencies obtained from real traces.
- Analyzing systems outside the logarithmic class of DHTs such as the ones based on the DeBruijn and Butterfly graphs.
- Analyzing the traffic load incurred by the constant relocation of replicated data items due to churn.

Swedish Institute of Computer Science SICS Dissertation Series

1. Bogumil Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, 1990.

2. Mats Carlsson, *Design and Implementation of an OR-Parallel Prolog Engine*, 1990.

3. Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*, 1990.

4. Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*, 1991.

5. Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*, 1991.

6. Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*, 1991.

7. Roland Karlsson, *A High Performance OR-parallel Prolog System*, 1992.

8. Erik Hagersten, *Toward Scalable Cache Only Memory Architectures*, 1992.

9. Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*, 1993.

10. Mats Björkman, *Architectures for High Performance Communication*, 1993.

11. Stephen Pink, *Measurement, Implementation, and Optimization of Internet Protocols*, 1993.

12. Martin Aronsson, *GCLA. The Design, Use, and Implementation of a Program Development System*, 1993.

13. Christer Samuelsson, *Fast Natural-Language Parsing Using Explanation-Based Learning*, 1994.

14. Sverker Jansson, *AKL - - A Multiparadigm Programming Language*, 1994.

15. Fredrik Orava, *On the Formal Analysis of Telecommunication Protocols*, 1994.

16. Torbjörn Keisu, *Tree Constraints*, 1994.

17. Olof Hagsand, *Computer and Communication Support for Interactive Distributed Applications*, 1995.

18. Björn Carlsson, *Compiling and Executing Finite Domain Constraints*, 1995.

19. Per Kreuger, *Computational Issues in Calculi of Partial Inductive Definitions*, 1995.

20. Annika Waern, *Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction*, 1996.

21. Björn Gambck, *Processing Swedish Sentences: A Unification-Based Grammar and Some Applications*, June 1997.

22. Klas Orsvärn, *Knowledge Modelling with Libraries of Task Decomposition Methods*, 1996.

23. Kia Höök, *A Glass Box Approach to Adaptive Hypermedia*, 1996.

24. Bengt Ahlgren, *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption*, 1997.

25. Johan Montelius, *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, May*, 1997.

26. Jussi Karlgren, *Stylistic experiments in information retrieval*, 2000

27. Ashley Saulsbury, *Attacking Latency Bottlenecks in Distributed Shared Memory Systems*, 1999.

28. Kristian Simsarian, *Toward Human Robot Collaboration*, 2000.

29. Lars-Åke Fredlund, *A Framework for Reasoning about Erlang Code*, 2001.

30. Thiemo Voigt, *Architectures for Service Differentiation in Overloaded Internet Servers*, 2002.

31. Fredrik Espinoza, *Individual Service Provisioning*, 2003.

32. Lars Rasmusson, *Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design*, 2002.

33. Martin Svensson, *Defining, Designing and Evaluating Social Navigation*, 2003.

34. Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, 2003.

35. Emmanuel Frecon, *DIVE on the Internet*, 2004.

36. Rickard Cöster, *Algorithms and Representations for Personalised Information Access*, 2005

37. Per Brand, *The Design Philosophy of Distributed Programming Systems: the Mozart Experience*, 2005

38. Sameh El-Ansary, *Designs and Analyses in Structured Peer-to-Peer Systems*, 2005