Distributed Data Structures for Peer-to-Peer Systems

A Dissertation

Presented to the Faculty of the Graduate School

of

Yale University

in Candidacy for the Degree of

Doctor of Philosophy

By

Gauri Shah

Dissertation Director: James Aspnes

December 2003

Keywords: peer-to-peer, distributed algorithms, skip graphs, distributed hash tables, fault-tolerant routing

To my parents, Rajni and Sudhir Shah, for a thinking life.

Abstract

Distributed Data Structures for Peer-to-Peer Systems

Gauri Shah
2003

Peer-to-peer systems are distributed systems of heterogeneous machines with no central authority, that are used for efficient management and location of shared resources. Such systems have become very popular for Internet applications in a short period of time because they provide inherent scalability by distributing the load of maintaining the system across all the participants. At the same time, they present new challenges in designing distributed data structures that can provide the desired functionality such as data availability, dynamic system maintenance, and support for complex queries using untrusted and unreliable components.

We present two complementary distributed data structures that can be used to implement efficient peer-to-peer systems. The first data structure is an abstract model of a **distributed hash table**, which is used as an overlay network by many contemporary peer-to-peer systems. This data structure provides inherent load balancing, and the number of routing hops for a search is logarithmic in the number of resources in the system. We study greedy routing in this model, and prove lower bounds and upper bounds on routing in the presence of failures in the network. We present some heuristics for constructing the network and give experimental results on the performance in practice.

We also present a novel distributed data structure called a **skip graph**, which is a trie of skip lists that share lower layers. A skip graph provides most of the functionality of a distributed hash table. In addition, it supports spatial locality and complex searches such as near matches to a key, keys within a specified range, or approximate queries. We give simple and straightforward algorithms to search and insert a new resource in a skip graph. We also give a repair mechanism that can be combined with the fault-tolerance properties to give a strong foundation for a highly resilient network.

Acknowledgments

I still find it hard to believe that this is the end of my graduate school days, and that the time has come to express my thanks for the wonderful time I have spent here at Yale. It well may be because I never quite realized that it *would* happen, let alone so soon.

Let me start by thanking the one person who has been most responsible for this dissertation seeing the light of day - my advisor, James Aspnes. I can easily imagine myself leaving Yale with a Masters degree in hand, had it not been for Jim. With his endless encouragement, patience, and enthusiasm, he has been a *terrific* advisor. He has taught me many things, right from simple mathematical tricks to the importance of cartoons in talks. Most importantly, he always had the time for me whether I was asking him a fair price to repair my broken car axle or discussing the proof of `LazyClock`. I will miss his delightful humor and all the random conversations on topics such as the incorrect design of Viking helmets. With due apologies to Jim, I cannot thank him enough for the opportunity to bask in the light of his sheer brilliance for the last three years.

My appreciation also goes to all the other members of my committee for their valuable feedback and time. I thank Joan Feigenbaum for much-needed advice on career choices. Arvind Krishnamurthy was always ready to discuss practical issues in networking, and spare cookies for me at tea-time. I am delighted that Antony Rowstron was enthusiastic enough to come all the way across the Atlantic to attend my defense.

Dana Angluin has been a constant source of inspiration and a role model for me. She was always encouraging when I talked to her about anything that bothered me, and I am grateful to her for her invaluable advice on several occasions. René Peralta lightened the atmosphere with his incredible sense of humor and a friendly smile next door at all times.

I thank the department administrative staff, especially Linda Dobb for taking care of the inevitable university formalities, and Amelia Toensmeier and Clara Garguilo for helping out with so many details like laptop reservations and expense reimbursements.

These acknowledgments would be incomplete without thanking my fellow graduate students. My office-mate, Rahul Sami, has helped to build my character over the last year by sharing his immense knowledge of system hacks and teaching me the importance of Zen sayings. Writing my dissertation was easier and a lot of fun because he was writing his thesis at the same time. I have also shared some wonderful times with Karhan Akcoglu, my office-mate for the first three years in this department. My heartfelt thanks to my other friends Gabriel Loh, Patrick Huggins, Jatin Shah, Vijay Ramachandran, and Aleksandr Yampolskiy, not only for the discussions related to research and otherwise, but for all the fun times that we have shared together as well. Thanks are also due to Neha Menon for proof reading my thesis.

Mayur has steadfastly stood by me through all my ups and downs in these last four years. Life would have been meaningless without all the songs he has sung for me, the frequent trips to California, and the long conversations about everything and nothing. And this thesis would have been chock-full of errors without his careful proof reading. I find it impossible to thank my best friend so I will just say that this dissertation would have remained incomplete without him.

Finally, as much as I would like to, I cannot thank my family enough in words, for their unconditional love and support at all times. My sister Uma and my brother-in-law Manoj have always encouraged me, and I have never had to worry about my familial duties while I was studying due to their steadfast support. My parents have given me far more than I ever dreamt of. I am forever indebted to them for having given me the choice and the chance to pursue what I desired, and to be what I am today. To them, I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"The power of the network increases exponentially by the number of computers connected to it. Therefore, every computer added to the network both uses it as a resource while adding resources in a spiral of increasing value and choice."*
>
> -Robert M. Metcalfe (1946-)

In the last decade, we have witnessed a boom in the use of the Internet for distributed applications. One of the popular applications is the use of **peer-to-peer systems** for music file sharing between communities of end users. In January 1999, Shawn Fanning left Northwestern University in his freshman year to develop the software for Napster [85], which was probably the first publicly-deployed system for music sharing. Napster was an instant success, used by millions of people all over the world soon after its inception. However, it was sued by the Record Industry Association of America (RIAA) for copyright infringement in December 1999. In spite of a partnership with the German media company Bertelsmann AG, to develop a membership-based distribution system that would guarantee payments to artists, Napster was eventually shut down by legal authorities in 2001. This, however, did not put an end to peer-to-peer file sharing, and we saw the emergence of many decentralized Napster-clones including Gnutella [43], MojoNation [81], KaZaA [61], AudioGalaxy [7], and Morpheus [82]. As of May 2003, there are more than 220 million downloads of KaZaA and 31 million of Morpheus*. Peer-to-peer systems continue to thrive; in fact, they are no longer restricted to sharing music files, videos, software, and other documents, but have moved beyond the realms of file sharing to projects such as SETI@home [107] which pools spare processing power of participants rather than their music file collections. Today, we see the use of these systems for a multitude of new applications such as data storage [69, 16, 100, 98], group communication [19, 101], web publishing and caching [118, 54, 55], and messaging [114].

This thesis explores the technical aspects of peer-to-peer systems and introduces new data structures to implement such systems efficiently. While the use of peer-to-peer systems raises a plethora of social, legal, and moral issues, we restrict our focus in this dissertation to the technical problems that arise in their design and deployment.

1.1 The peer-to-peer paradigm

The natural first question to ask is: what exactly are peer-to-peer systems? Peer-to-peer systems are large-scale, distributed networks of computers with no central authority†. Each computer or machine in a peer-to-peer system, also known as a **peer**, can be heterogeneous with possibly varying computational power, and is subject to crash failures. Peer-to-peer systems can be used for various different purposes such as: distributed content management, distribution of computing cycles and people-to-people collaboration. For the rest of this dissertation, we concentrate on the issues involved in content distribution, although the use of peer-to-peer systems is by no means restricted to this area of focus.

In a typical content-distribution peer-to-peer system, each peer may have some **resources** to share: these resources can be files, documents, web pages, temporary access to local applications, connections to physical devices etc. Each resource is uniquely identified by a **key**; for example, it could be a URL for a web page or the filename for a document. The primary goal is to share the available resources *efficiently* among all the peers. Given the key of a resource, or a set of keywords that describe a group of acceptable resources,

*Detailed statistics of download information can be found at `http://www.download.com`.

†There can be some central agency that issues machine identifiers, for example IP addresses, but the organization of the peer-to-peer system itself is decentralized.

we would like to design a peer-to-peer system that will locate resources in the system, and enable sharing with minimum communication, storage, and maintenance. In a traditional client-server model, the responsibility to maintain such a system falls on a small cluster of server machines. As opposed to that, peer-to-peer systems distribute the load of maintenance among all the participating peers, thus providing inherent scalability and load balancing. At the same time, new challenges have to be met to build efficient and robust distributed data structures to implement such systems using individual, possibly faulty components.

Scalable Distributed Data Structures was a term first coined by Litwin *et al.*[76] in the context of distributed database systems. We seek to design similar large-scale, distributed data structures that easily support location and maintenance of shared resources, and have the following desirable properties:

Decentralization: The data structure should be distributed among all the participants of the system. A central server, or even a cluster of such servers, may prove to be intolerant to faults, and will require considerable investment for high-performance hardware and high bandwidth.

Scalability: The Internet user community has grown to be so large that distributed systems need to cope with millions of users. In an ideal peer-to-peer system, the cost borne by each participant should not depend too much on the size of the entire system.

Load balancing: We would like the cost of maintaining the system to be uniformly shared between all the peers. Similarly, the system should be able to manage **flash crowds** i.e., high data request volume due to **temporal locality**, when a particular resource becomes extremely popular for a short period of time. For example, the popular web site CNN (`http://www.cnn.com`) saw record traffic, hitting nine million page views an hour during the terrorist attacks on September 11, 2001 in New York, USA, compared to an ordinary volume of eleven million page views a day [39].

Dynamic maintenance: The massive parallelism in peer-to-peer systems, due to high rate of machine arrivals and departures, presents some very challenging issues that are trivially solved in a system with fixed membership. The system should be self-configuring, and machines and resources should be added and deleted from the system quickly without manual intervention or oversight.

Fault tolerance: The data structure should be resilient to both machine and link failures in the system. Even if a part of the system has failed, the data available in the surviving machines should still be accessible, as long as it is located in the same connected component as the requesting peer. Further, the system should gracefully degrade with increasing failures.

Self-stabilization: Not only should the system survive disruptions due to failures, but it should also heal automatically to restore ideal performance. The system should have a repair mechanism that detects local inconsistencies such as machine failures or link outages, and triggers maintenance operations with minimal overhead in terms of network traffic.

Efficient searching: The primary goal of a peer-to-peer system is to locate resources efficiently, and hence support for searching using a variety of specifications is a very desirable property. Complex queries to locate resources such as range queries, near matches to a key, and keyword matches should be supported by a rich query language.

Security: The system should be secure against attacks such as a **denial-of-service** attack, where some miscreant participants may "flood" the system, thereby preventing legitimate traffic. In some applications, it may also be desirable to maintain anonymity of the users, or provide resistance to censorship by preventing certain data items to be deleted from the system.

Topologically-sensitive construction: Routing should be sensitive to network locality such as distance traveled or latency along transmission paths. Two possible approaches are: (i) **Proximity routing** where machines are placed in the network to exploit the underlying topology, and (ii) **Proximity neighbor selection** where the closest neighbors (as per the proximity metric) are chosen among the set of potential neighbors.

In this dissertation, we are primarily interested in developing approaches to design peer-to-peer systems that focus on the features listed above.

1.2 Evolution of peer-to-peer systems

Before we look at contemporary peer-to-peer systems, we discuss the need for a new model to implement peer-to-peer systems. The web has traditionally been used for content distribution. Was it feasible to use the web to implement peer-to-peer systems? There are two important reasons that make the web unsuitable for our purposes. Firstly, the web is based on a traditional client-server model where there are relatively few, more powerful, and reliable server machines that are responsible for serving the content. A peer-to-peer system, on the other hand, is a very dynamic system with peers continuously entering and leaving the system. Traditional solutions to locate resources that can be used in a system with fixed membership like the web, are not viable for such dynamic systems. Secondly, resource location on the web is done using *domain name servers* that are organized hierarchically in a tree. Each node of the tree stores some well-defined subset of the IP keyspace and services requests for it. This tree approach is not very fault-tolerant; there is far too much load on the nodes closer to the root, and even a single failure can partition the entire data structure. Replicating nodes may alleviate the problem but better solutions have been proposed that retain the benefits of trees. The large-scale, distributed nature of the Internet and the growing size of the consumer base have created the need for a new model for peer-to-peer systems. However, we note that by no means are we claiming that the peer-to-peer model will replace the client-server model in the near future; both models will co-exist depending on the applications that they are being used for.

We now explore some of the designs that have been proposed for implementing peer-to-peer systems.

1.2.1 First-generation systems

Napster [85] was the first publicly-deployed peer-to-peer system. It used a central sever to maintain an index of (*filename, IP address*) pairs, and kept track of the locations of all the music files in the system. When a user wanted a particular music file, she would query this server, and upon receiving the IP address(es) of the machine(s) that hosted the requested resource, she would download the file from that location. This file transfer was decentralized and independent of the central index server. One potential argument against this simple approach is the lack of scalability, but the existence of large-scale server clusters, such as those used by popular web sites Google [46] and Yahoo [120], prove otherwise. However, the set-up of such systems may require considerable investment for high-performance machines, high bandwidth etc. which makes it unsuitable for implementing a peer-to-peer system. Additionally, the approach is not fault tolerant; the central server provides a single vulnerable point of failure. This fact led to the ultimate demise of Napster as the legal authorities tracked, and eventually shut down, the single entity that was responsible for the management of Napster.

The fall of Napster saw the emergence of several other systems such as Gnutella [43], Freenet [21][‡], Morpheus [82], and KaZaA [61]. Unlike Napster, Gnutella decentralizes the initial resource discovery in addition to the file transfer, by using **flooding** to locate resources [44]. Gnutella builds a network in which each machine is connected to a few other machines in the network which are called its *neighbors*, i.e., each machine knows the IP addresses of its neighbors and can communicate with them. A machine requests its neighbors for a desired resource; these neighbors in turn ask their neighbors, and so on, until the resource is found. No doubt, the motivation for this design was to make no single participant indispensable, and thus make it impossible for any authority to control or censor the network. From a technical standpoint however, this is not a scalable solution as it creates an enormous volume of traffic in the network for each request [97]. In addition, flooding creates a trade-off between overloading every machine in the network and cutting off searches before completion. If the time-to-live on a message expires, a resource may not be found even if it is only one hop away from the point of cut-off. So even if the resource is actually present in the network, it may not be accessible if it is just *out of reach* of the requesting peer.

Systems such as Morpheus and KaZaA alleviate the problem of flooding traffic somewhat

[‡]We discuss Freenet in further detail in Chapter 2.

in practice by using **super-peers**[§]. Peers form local groups and elect a super-peer to participate on their behalf in the network. All the resource requests are sent only to the super-peers, each of which maintains an index of the resources hosted by its own group. If a resource is not found in its own group, a super-peer will communicate with other super-peers to locate the requested resource. Although it amends some of the problems of flooding, this approach is still inherently unscalable. Flooding can be avoided, but with the additional cost of maintaining synchronized, distributed indexes among the super-peers.

1.2.2 Contemporary systems

The first-generation commercial systems have spawned a lot of interesting research in the academic community, the details of which are covered in Chapter 2. Here, we just give a brief overview of some of the well-known systems to present a context for our contributions.

Several recent systems like CAN [95], Chord [112], Pastry [99], and Tapestry [125] use a **distributed hash table (DHT)** as the basic data structure for a peer-to-peer system. The main operation of a DHT is to retrieve the identity of a node that hosts a particular resource, starting from any other node in the network. DHTs allow location of resources using a directory-like interface that supports storing and fetching data indexed by keys. The underlying theme of all these systems is that they build an overlay network on top of the physical network, and embed the machines in the overlay network by *hashing* their identifiers. Resource keys are distributed, either randomly or by hashing, among the nodes to facilitate uniform load distribution. As both machines and resources are embedded in the overlay network using hashing, these systems are called distributed hash tables. Each node is responsible for the resources that hash to locations near itself. A node in the overlay network can either store pointers to the addresses of the resources that it is responsible for, or it may actually store the resources themselves, depending on the specific application. Details of a common API which is being developed for such structured overlay networks can be seen in [27].

The nodes are linked in the overlay network using a specific link distribution depending on the design. Resource location, using the overlay network, is done in these various systems

[§] A recent version of Gnutella also supports the use of super-peers.

by using different routing algorithms. Each node maintains some information about its neighbors, and routing is done greedily by forwarding messages to the neighbor *closest* to the target node. This inherent common structure leads to similar results for the performance of such networks; with m nodes in the network, most of these systems use $O(\log m)$ space at each node, and take $O(\log m)$ time for routing messages.

1.3 Model

Before we move on to our contributions in the next section, we briefly describe the model used for the results. The interested reader is referred to the books by Lynch [77] and Attiya and Welch [6] for further details of the model.

Environment: We use the term **node** to represent a process that is running on a particular machine. We assume a **message-passing** environment in which all processes communicate with each other by sending messages over a communication channel, where each channel provides a bidirectional connection between two specified processes. An algorithm in a message-passing system consists of a local program for each process in the system. This local program provides the ability for the process to perform local computations, and to send messages to and receive messages from each of its neighbors. Occurrences that can take place in the system are modeled as **events**. There are two kinds of events: A **computation** event which represents a computation performed locally by some process, and a **delivery** event that represents the delivery of a message from one process to another. In a system represented by this model, several events may occur concurrently.

Time and asynchrony: A system is said to be totally asynchronous if there is no fixed upper bound on how long it takes for a message to be delivered, or how much time elapses between consecutive steps of a process. In practical systems, however, we would like to assume some upper bound on the message transmission delay so that failures in the system can be detected. We assume that the system is **partially synchronous**, i.e., there is a fixed upper bound (time-out) on the transmission delay of a message. We assume that each message takes at most unit time to be delivered, and any internal processing at a machine takes no time. We also assume that a message which has been sent will eventually get delivered.

Failure model: Processes can **crash**, i.e., halt prematurely, and crashes are permanent. We consider a **failstop** model where processes have some method for correctly detecting whether another process has failed. As messages are always reliably delivered and there is a fixed time-out on every message, a lack of response for a message is one way for the failed processes to be identified, although it may be impossible to distinguish between crashed processes and extremely slow ones. We do not account for **byzantine** failures where a process remains in the system, but it behaves arbitrarily and sends faulty, possibly malicious messages to other processes. We defer the issue of byzantine failures to future work.

Complexity Measures: We are mainly interested in three performance measures for our algorithms: (i) the total amount of time, (ii) the storage space at each process, and (iii) the total number of messages for any algorithm. In a partially synchronous model, as there is a fixed upper bound on the transmission delay of each message, the time taken for an event is of little value. Also given the low cost of storage nowadays, we are not as interested in minimizing the storage for each process as we are in keeping the number of messages low. However, as the storage for each process will mainly consist of information about its neighbors, the higher the number of neighbors, the greater will be the number of messages sent as a part of the repair mechanism to fix any broken links to neighbors. If each node periodically initiates actions to repair links to failed neighbors, it will create a deluge of messages for the repair mechanism. Thus, in order to minimize the number of messages, it would be beneficial to minimize the number of neighbors per process, and in turn reduce the storage space per process. We concentrate on measuring the number of messages for our algorithms, and the space used at each process to evaluate the performance of our data structures. We note that there is a trade-off between storage requirements and number of routing hops for a search query, and we try to maintain a balance between the two parameters. We use the terms space, storage, and state interchangeably to denote the storage space at each process to maintain information about the state of the system.

1.4 Our contributions

In joint work with James Aspnes and Zoë Diamadi [4], I considered the problem of designing an overlay network and a routing mechanism that enables finding resources efficiently in a peer-to-peer system. We argue that many existing approaches to this problem that use DHTs, can be modeled as the construction of a random graph embedded in a metric space whose points represent resource identifiers, and the probability of a connection between two points depends only on the distance between them in the metric space. We study the performance of a peer-to-peer network where resources are embedded at grid points in a simple metric space: a one-dimensional real line. We prove upper and lower bounds on the message complexity of locating particular resources in such a network, under a variety of assumptions about failures of either nodes or the connections between them. Our lower bounds in particular show that the use of inverse power-law distributions in routing, as suggested by Kleinberg [66], is close to optimal.

We propose an abstract model for a peer-to-peer network as follows: (1) embed resources as points in a metric space, (2) construct an appropriate random graph on these points, and (3) efficiently locate resources by greedy routing. We consider a simple one-dimensional metric space with machines and resources embedded at grid points of a real line. Each machine is embedded at a different point for every resource that it hosts. Thus, a single machine may have multiple locations in the overlay network, depending on the number of resources that it hosts. Each point in the metric space is connected to its immediate neighbors and ℓ long-distance neighbors chosen with probability inversely proportional to the distance between them.

Our main contributions are as follows:

- We prove that with n points in the metric space and $\ell \in [1 \ldots \log n]$ long-distance links per point, routing takes $O((\log^2 n)/p\ell)$ time, if each point or link fails with probability $(1-p)$ [Theorems 3.7 and 3.10].

- We prove that **one-sided** greedy routing, where the routing algorithm never traverses a link past the target point, takes $\Omega(\log^2 n/\ell \log\log n)$ time with *any* link distribution. Similarly, with **two-sided** greedy routing, where the link closest to the target point is taken, we get a lower bound of $\Omega(\log^2 n/\ell^2 \log\log n)$ [Theorem 3.3].

- We present some heuristics for constructing and maintaining the network dynamically, and experimental evaluations that show that the heuristics give a resulting link distribution that is very close to the optimal inverse power-law distribution with exponent 1 [Section 3.6].

- We also present some interesting experimental results on the number of routing hops for a search operation using different routing strategies such as backtracking and random rerouting [Section 3.7].

The underlying structure of DHTs resembles a balanced tree in which balancing depends on the near-uniform distribution of the output of the hash function. Hashing the machine identifiers uniformly distributes the machines in the metric space, and thus naturally provides load balancing in the network. However, as the resource keys are also hashed, DHTs can provide only hash-table functionality i.e., they only support point queries or exact match queries. Because hashing destroys the ordering on keys, DHT systems do not support queries that seek near matches to a key or keys within a given range. Designing a system that supports range queries had been posed as an open question by Harren *et al.*[50]. DHT systems also destroy the property of **spatial locality** where related resources are present near each other in the overlay network. So, having explored DHTs, we modified our approach to exploit the underlying tree structure to give tree functionality, while applying a simple distributed balancing scheme to preserve balance and distribute load.

In joint work with James Aspnes [5], I described a new model for a peer-to-peer network based on a distributed data structure that we call a **skip graph**. Skip graphs are a novel distributed data structure, based on skip lists [93], that provide the full functionality of a balanced tree in a distributed system, where elements are stored in separate nodes that may fail at any time. They are designed for use in searching peer-to-peer networks, and by providing the ability to perform queries based on key ordering, they improve on existing search tools that provide only hash-table functionality. Unlike skip lists or other tree data structures, skip graphs are highly resilient, tolerating a large fraction of failed elements without losing connectivity. In addition, constructing, inserting new elements into, searching a skip graph and detecting and repairing errors in the data structure introduced by node failures can be done using simple and straightforward algorithms.

Our main results are summarized as follows:

- We introduce a new distributed data structure called a skip graph for a peer-to-peer network. We show that in a skip graph with n elements, searches, insertions, and deletions take $O(\log n)$ time [Lemmas 4.2, 4.3 and 4.4].

- We prove that a skip graph with n elements has an expansion ratio of $\Omega(1/\log n)$ with high probability, thus proving that it is resilient to adversarial failures [Theorem 4.10]. We also give experimental results to show that a skip graph performs well even in the presence of random failures [Section 4.7.1].

- We give a repair mechanism that repairs any connected component of a defective skip graph in the absence of new failures, inserts, and deletes [Theorem 4.22].

- We prove that the congestion at a particular element x in a skip graph is inversely proportional to the number of elements between x and the target element to which the message is sent, and that the distribution of this congestion decreases exponentially beyond this point [Theorems 4.24 and 4.25].

1.5 Summary

The rest of this dissertation is structured as follows: Chapter 2 covers related research in the area of peer-to-peer systems and other contemporary systems being used commercially. The main technical results are in Chapters 3 and 4. In Chapter 3, we discuss the details of the results on fault-tolerant routing in our abstract model of an overlay network as a metric-space embedding. Most of the material in this chapter is adapted from [4]. Details of skip graphs and their properties are given in Chapter 4, and much of this work appears in [5]. Finally, we conclude with directions for future work in Chapter 5.

Chapter 2

Related Work

There has been a slew of research in the area of peer-to-peer systems in the last couple of years. In this chapter, we summarize some of the well-known contemporary peer-to-peer systems and related research, although this brief survey is by no means complete or exhaustive. In particular, we do not cover the papers related to security in peer-to-peer systems as they are beyond the scope and the focus of this dissertation.

2.1 First-generation systems

In Chapter 1, we discussed some of the first-generation peer-to-peer systems which we mention here for the sake of completeness.

Napster [85] used a central server to index the resources in the system, and peers used this index to locate files in the system. The actual file transfer was decentralized and occurred between the peer that requested the resource and the peer that hosted it. Gnutella [43] decentralizes the initial resource discovery as well by flooding the network with a request. It works like a social network, where to get some piece of information, a person will first ask her friends, and they will in turn ask their friends, and so on. KaZaA [61] and Morpheus [82] ameliorate the problems of flooding by using super-peers, that participate in the network on the behalf of smaller groups of end users, but they do not solve them in the limit. Some rules of thumb for the design of super-peer networks, such as redundancy and cluster size, have been formulated by Yang *et al.*[122] based on a study of the performance of Gnutella.

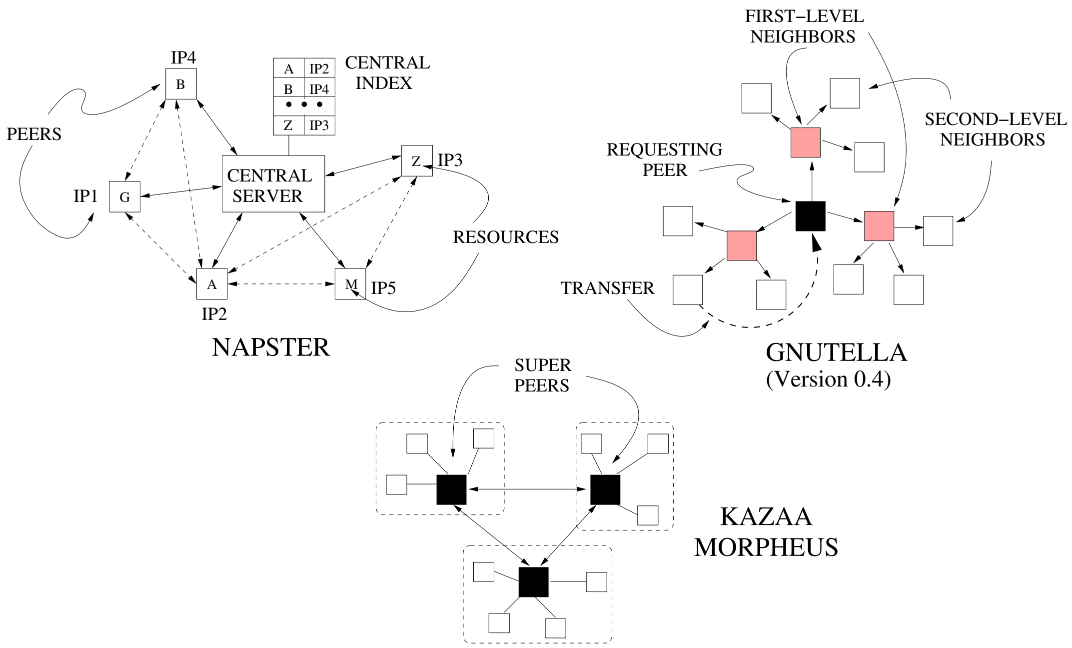


Figure 2.1: First generation systems: Napster, Gnutella, KaZaA, and Morpheus.

Freenet: The main goal of Freenet [21] is to enable publishing, retrieving and replicating documents while protecting the *anonymity* of both the publishers as well as the readers. Each file is inserted into the network using a key which is a hash of its content. Every node maintains a cache (not a persistent store) of the last few keys it has seen, and keeps track of the node that it forwarded the request for a particular key to. A node compares any incoming request to the entries in the cache, finds the closest matching key to it and forwards the message to the corresponding node. Files are inserted and searched in the same way, thus ensuring that the nodes cooperate to route requests to the most likely physical location of the data. If a node fails to locate a resource, it returns the original message downstream along the same path that it came from, and the request is forwarded to the next best neighbor. We can think of the Gnutella search algorithm as a *breadth-first search* as opposed to a *depth-first search* in Freenet. Routing is expected to improve over a period of time because nodes will specialize in locating sets of similar keys as well as storing clusters of files with similar keys. Thus, files will be dynamically replicated in locations near requesters and deleted from locations where there is no interest. The performance of Freenet is difficult to evaluate and it provides no provable guarantee on the search latency. Another drawback is that it does not guarantee to find accessible content.

2.2 Before Distributed Hash Tables

The first-generation peer-to-peer systems inspired the development of more sophisticated second-generation ones like CAN [95], Chord [112], Pastry [99], and Tapestry [125]. Although these systems appear vastly different, there is a recurrent underlying theme: they all use some variant of an overlay metric space in which the machines and resources are embedded using *hashing*. As explained earlier, each machine can either maintain the addresses of the resources that it is responsible for, or it can actually host the resources. Routing is done *greedily* by forwarding packets to neighbors *closest* to the target node with respect to the metric distance. This inherent common structure leads to similar results for the performance of such networks. Before we explain DHTs in detail, in this section, we look at some of the precursors that led to their development.

2.2.1 Hashing

Hashing has been used earlier for several applications such as distributed databases [75, 70, 76] and distributed caching protocols [57, 3]. LH [75] is a hashing method used for extensible disk files that grow or shrink dynamically, with no deterioration in space utilization or access time. Files are organized into buckets on disk that can be accessed using a pair of hashing functions h_i and h_{i+1}. Function h_i is linearly replaced by h_{i+1} as the bucket capacities are exceeded, and few objects need to be transferred each time a new bucket is added. Records can be located in a constant number of hops. LH* [76] generalizes LH to distributed disk files without using a central coordinator and maintains an average load factor of $80\% - 95\%$ for each bucket.

Consistent hashing was introduced by Karger *et al.*[57] for caching protocols that can be used to relieve **hot spots** in distributed systems. Hot spots occur when a sudden increase in the demand for a particular resource swamps the server that hosts it. To avoid hot spots, the server evenly distributes the resources across the set of available caches such that (i) the expected number of resource transfers when a cache leaves, is small, (ii) the users get a mostly consistent view of all the caches as each resource hashes to a small set of caches, and (iii) each cache is responsible only for its fair share of resources. This scheme is similar to LH* but it allows buckets to be added in any arbitrary order. Consistent hashing has been used by Akamai [3] for reliable, global, content delivery on the web. It is also used as the core hashing algorithm in Chord [112] for load balancing and near-uniform distribution of machine identifiers in the metric space to eliminate the need for balancing the underlying tree structure. While the other DHT systems such as CAN [95], Pastry [99], and Tapestry [125], do not specify the exact hash function to be used, they can also use consistent hashing for mapping machines on to the overlay network.

2.2.2 Plaxton/Rajaraman/Richa [PRR] algorithm

Plaxton *et al.*[92] gave a distributed algorithm for accessing shared resources in an overlay network, while using small-sized routing tables at each node in the network. This routing algorithm is used in both Tapestry [125] and Pastry [99] with a few modifications. The algorithm is based on **suffix-based hypercube routing**. Each node has a unique t-bit identifier divided into r levels of $w = t/r$ bits each; let w bits represent a *digit*. A node x will have r sets of 2^w neighbors each, such that each set i, $0 \leq i < r$, will have nodes with identifiers as follows: i common digits with x's identifier, followed by all possible 2^w values for the $(i+1)$st digit, and any of the 2^w possible values for each of the remaining digits. An example routing table is given in Table 2.1. Routing is done by greedily forwarding the message to the node in the routing table, which has the longest common suffix with the target identifier. For example, a message from 1234 to 3433 may be routed as follows: $1234 \rightarrow 121\mathbf{3} \rightarrow 22\mathbf{33} \rightarrow 1\mathbf{433} \rightarrow \mathbf{3433}$.

Let $b = 2^w$. Then with m nodes in the network, the size of the routing table is $(b-1) \cdot \lceil \log_b m \rceil$ and routing takes $\lceil \log_b m \rceil$ time. This was probably the first routing algorithm introduced that was scalable for peer-to-peer systems. However, in the original scheme, each resource was associated with a unique **root** node that maintained the address of the resource. This is disadvantageous as it requires global knowledge to map resources to their

Suffix Length	Neighbors			
0	1131	3422	1213	-
1	3214	1224	-	334**4**
2	21**34**	-	43**34**	14**34**
3	-	2**234**	3**234**	4**234**

Table 2.1: An example of the routing table of node 1234 using the PRR routing scheme. Here $t = 8, r = 4, w = 2$.

root nodes.

2.3 Distributed Hash Tables

In this section, we give a brief overview of the distributed hash table systems, highlighting some of their major design aspects.

2.3.1 Content-addressable network

CAN [95] partitions a d-dimensional coordinate space into **zones** that are owned by nodes. Resource keys are mapped to points in the coordinate space using a uniform hash function, and then stored at the node which owns the zone in which the point is located. Each node maintains an $O(d)$ state about its neighbors which abut with the node only in one dimension. Greedy routing, by forwarding messages to the neighbor closest to the target zone, takes $O(dm^{1/d})$ time with m nodes in the network. With $d = \log m$, each node uses $O(\log m)$ space and routing takes $O(\log m)$ time. Figure 2.2 shows an example of a two-dimensional CAN with a route from zone 3 to zone 8 using greedy routing.

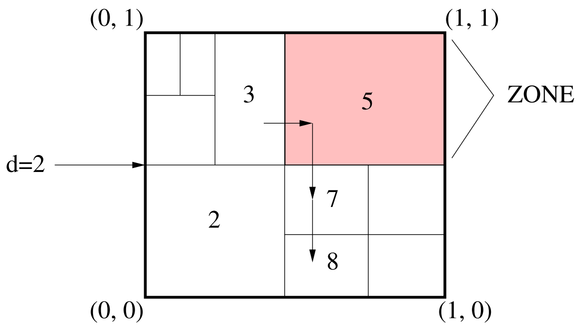


Figure 2.2: A two-dimensional content-addressable network (CAN).

To join the network, a new node picks a random point in the coordinate space, and the existing node that owns the zone in which the point lies, splits the zone in half and assigns one half to the new node. The dimension along which the split occurs is chosen in a fixed round-robin fashion among all possible dimensions. Node failures are handled in the same way by merging zones of failed neighbors. Various additional design improvements include multiple coordinate spaces to improve data availability, better routing metrics such as forwarding to the neighbor with the smallest round-trip latency, multiple peers per zone and multiple hash functions for fault tolerance etc.

For proximity routing, a topologically-sensitive construction of the CAN was presented in [96], which is similar to the ideas of *Landmark Hierarchy* of Tsuchiya [115] and *Global Network Positioning* proposed by Ng *et al.*[86]. Each node in CAN measures its round-trip delay to a set of pre-determined landmarks, and accordingly places itself in the coordinate space. With t such landmarks, $t!$ orderings are possible, so the coordinate space is divided into $t!$ equal-sized portions, and each node will join the portion that matches its landmark ordering. This method has two drawbacks: It is not self-organizing because of the need to fix the landmarks, and it can cause a non-uniform distribution of nodes in the coordinate space leading to hot spots.

2.3.2 Tapestry and Pastry

Tapestry [125] and Pastry [99] use the PRR algorithm [92] explained in Section 2.2.2 as the core routing algorithm, with modifications to the other features of the design to make it suitable for use in a distributed environment. To incorporate fault tolerance, Tapestry maintains multiple neighbors per entry in the routing table to route around failures. It eliminates the need for a central entity for mapping resources to root nodes, by assigning a resource to the node whose identifier matches the hash value of the resource key in the maximum number of trailing bits. **Surrogate routing** is used to locate a resource by also storing it at a node closest to its chosen root node, if the latter is absent. There are many more optimizations such as dynamic node addition and deletion, soft-state publishing and supporting mobile resources, which are too involved to describe here, and the interested reader is referred to [125].

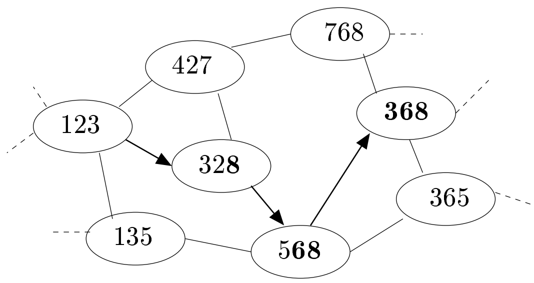


Figure 2.3: Tapestry, based on the PRR routing algorithm. Routing from node 123 to node 368 is shown here. Pastry is a similar system that uses prefix-based routing.

Pastry uses prefix-based routing instead of suffix-based as in Tapestry. Each node maintains a **neighborhood** set and a **leaf** set in addition to the routing table as per the PRR algorithm. The neighborhood set of a node consists of a group of nodes that are closest to it as per the proximity metric; they help the node to choose the closest nodes that satisfy the routing table criteria. The leaf set L is the set of nodes with the $|L|/2$ numerically-closest larger identifiers, and the $|L|/2$ numerically-closest smaller identifiers relative to the node's own identifier. If the routing table node is not accessible, the current node will forward the message to one of the leaf set nodes whose identifier prefix matches the target in the same number of bits as the current node, but whose identifier is numerically closer to that of the target. This procedure ensures that the routing always converges, though not necessarily efficiently.

Both Pastry and Tapestry can exploit topology by proximity neighbor selection i.e., choosing the *closest* node (as per the proximity metric), out of all the possible neighbors with appropriate matching identifiers. If the entire m^2 matrix with the latency or distance values between all m nodes is available, then the best neighbor can easily be chosen to support proximity routing; efficient heuristics can also be used for neighbor selection in the absence of this matrix. From their experimental results, Pastry does particularly well with paths that are only $30\% - 40\%$ longer than the ideal routes. However, both schemes weigh progress in the identifier space versus the progress in the proximity space, thus making it unclear if this is the best approach to achieve *global* proximity routing.

2.3.3 Chord

Chord [112] provides a hash-table functionality by mapping m machines (using consistent hashing [57]) to identities of $\log m$ bits placed around a *identifier circle*. Each node x stores a pointer to its immediate **successor** i.e., the closest node in the clockwise direction along the circle. In addition, it also maintains a **finger table** with $\log m$ entries such that the i-th entry stores the identity of the successor of $x + 2^{i-1}$ on the identifier circle. An example of these pointers is shown in Figure 2.4. Each resource is also mapped using hashing onto the identifier circle and stored at the first node succeeding the location that it maps to. Routing is done greedily to the farthest possible node in the routing table, and it is not hard to see that this gives a total of $O(\log m)$ routing hops per search. Further, consistent hashing ensures that no node is responsible for maintaining much more than its fair share of resources.

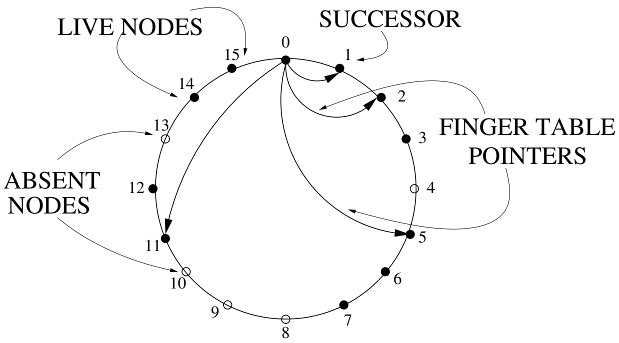


Figure 2.4: The finger table pointers for node 0 in Chord.

For fault tolerance, each node maintains a *successor list* which consists of the next several successor nodes, instead of a single one. Theoretical results about the performance of Chord in the presence of concurrent joins and involuntary departures can also be found in [112].

Variants Some systems use consistent hashing like Chord for load balancing resources across an active set of nodes; however they use different connections between nodes for routing.

Kademlia [80] has the same design as Chord, except for the distance metric; it uses the XOR value of the two node identifiers in the overlay network. Unlike Chord, this makes the distance between any two nodes symmetric. An advantage of this network is that each node can choose its neighbors from a range of possible choices, which makes the network fault tolerant, and allows it to adapt to changes in the network conditions such as latency, by choosing different nodes to route to.

Symphony [79] uses a probabilistic scheme for maintaining the links in a Chord-like base system: The probability of a link between two nodes u and v is inversely proportional to the distance between them. This scheme is similar to the one we explain in Chapter 3, but unlike us, they do not give analytical results for fault tolerance.

Viceroy [78] maintains a constant-degree, logarithmic-diameter approximation to a butterfly network for routing. It reduces the space per node to an expected $O(1)$ links by the clever use of butterfly routing in the last stages of a search operation. However, improving the fault tolerance of such a network while maintaining the low storage is still an open question.

Koorde [56] embeds a de Bruijn graph on the Chord identifier circle for routing. Each node with a t-bit identifier $b_t b_{t-1} \ldots b_1$ is connected to nodes with identifiers $b_{t-1} b_{t-2} \ldots b_1 0$ and $b_{t-1} b_{t-2} \ldots b_1 1$. This gives a similar, constant-degree design like Viceroy but it is relatively easier to construct and the fault tolerance can be improved by increasing the degree to t.

2.4 Censorship-resistant networks

Some of the DHT systems are partly resilient to random node failures, but their performance may be badly impaired by adversarial deletion of nodes. Fiat *et al.*[35] introduced censorship-resistant networks which are resilient to adversarial deletion of a constant fraction of the nodes*, such that most of the remaining live nodes can access most of the remaining data items. With m nodes in the network, the network topology is based on a butterfly network of depth $(\log m - \log\log m)$, where each node represents a set of peers, and each peer can be a member of multiple nodes. An expander graph is maintained between any two connected nodes of the network as shown in Figure 2.5.

*The paper describes a network that is robust up to deletion of half the nodes, but it can be generalized to any arbitrary fraction of deleted nodes.

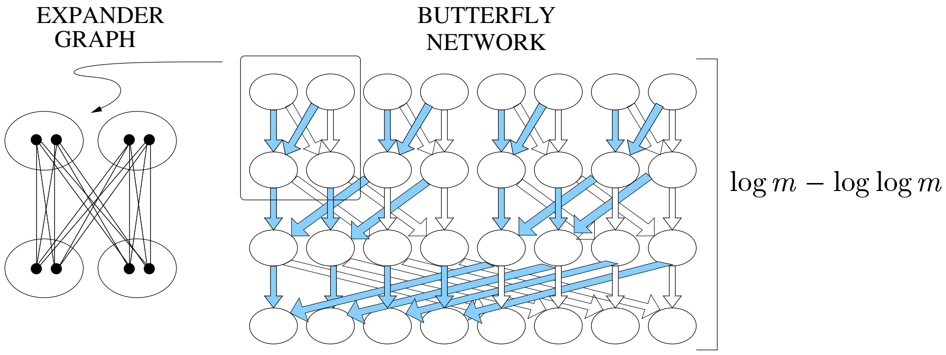


Figure 2.5: Censorship-resistant network: butterfly network of depth $\log m - \log\log m$ with expander graphs between nodes.

Multiple copies of the data items are stored in the nodes in the bottom-most level and searches are run in parallel for any copy of a data item. Each search takes $O(\log m)$ time, $O(\log^2 m)$ messages and $O(\log m)$ storage per node. The authors also give a *spam-resistant* variant of this scheme which requires additional storage space and messages, where the adversary can take control of up to half of the nodes, and yet be unable to generate false data items. The main drawback of this design is that it can survive only a static attack, and cannot be dynamically maintained as more nodes join the network. Some extensions of this result for dynamic maintenance can be seen in [102] and [28]. However, the construction of such networks is very complex, and it is still an open question to dynamically maintain such a network when the size changes by an order of magnitude to need a new level of butterfly linkage. A simpler design of a DHT which is provably fault-tolerant to *random faults* was recently given by Naor *et al.*[84].

2.5 Other systems

There are some other peer-to-peer systems that either eliminate namespace virtualization, or use hybrid systems which have a less rigid structure compared to DHTs, but provide better search efficiency compared to the first-generation systems such as Gnutella. We explore some of these designs in this sections.

2.5.1 Non-virtualized systems

TerraDir: TerraDir [109, 14, 62] is a recent system that provides a directory lookup for *hierarchical namespaces*, such as DNS names or Unix file system names, as opposed to the flat namespaces in DHTs. Caching and replication are heavily used for both fault tolerance and reduction of query latencies. Nodes at height i in the hierarchical namespace tree are replicated $O(i)$ times, so that the root is replicated $O(\log m)$ times (with m nodes in the network), while the leaves have just one replica. Each node is replicated randomly at other nodes, and each parent maintains information about all the replicas of each of its children. Routing is performed by going "up" in the tree until the longest common prefix of the source and destination is reached, and then going "down" until the destination is reached. For example, as shown in Figure 2.6, a search from `/J/D/A` to `/J/D/F` goes up till `/J/D` and then down to the destination.

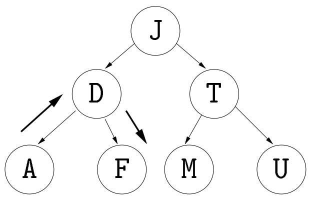


Figure 2.6: Routing in TerraDir from `/J/D/A` to `/J/D/F`.

Each node also calculates the distance to the target from all the possible replicas that it maintains and chooses the closest, live one to improve routing. As the namespace is not virtualized, spatial locality is maintained, which supports non-point queries such as `/J/*`. But this support is only partial because if a request is serviced from some replica and not the original node, then it may be possible that the address for the next data item is not known. Further, there are as yet no provable guarantees on load balancing and fault tolerance for this network.

Distributed trie: Freedman *et al.*[38] use a distributed trie [29, 36, 68] to support non-point queries; the keys are maintained in a trie which is distributed among the peers depending on their access locality. Updates are done lazily by piggybacking information on query traffic. This algorithm reduces the message traffic compared to flooding but can degenerate to a broadcast for all nodes with stale views of the network.

SkipNet: SkipNet is another non-virtualized system, very similar in spirit to skip graphs that we describe in Chapter 4, which was developed independently by Harvey *et al.*[51]. We elaborate on the difference between the two systems in Section 4.4.

2.5.2 Hybrid systems

Yappers: Yappers [42] has an unstructured network like Gnutella but introduces hashing for storing keys. Each machine x is assigned one of b *colors* based on its IP address: color(x) = hash(IP(x)) mod b. Each node maintains information about an *immediate neighborhood* which comprises of all nodes within a constant number of routing hops from itself. Resources are stored at nodes whose color matches the hash value of the resource key, so searches are only forwarded to nodes of the same color. While this improves the performance of the Gnutella search, it does not give any provable guarantees on routing time or load balancing.

Kelips*: Kelips* [48] is a hybrid system between a DHT and a super-peer network, where m nodes are clustered into $O(\sqrt{m})$ **affinity groups** using hashing. Each node resembles a super-peer by maintaining information about all the nodes and resources in its own group, and one contact from every other group. Space requirement per node is $O(\sqrt{m})$ and routing takes $O(1)$ time. Updated network information is propagated using gossip protocols given by Kempe *et al.*[63]. For larger peer-to-peer systems, the space and maintenance costs are prohibitively high, and routing may fail due to stale routing tables.

2.6 Improved features

So far we have seen designs of peer-to-peer systems as a whole but research has also been done on specific features of these systems such as improved search techniques and load balancing.

2.6.1 Search techniques

Intelligent flooding: In [121], the authors propose three different search techniques for Gnutella instead of naïve flooding on the network: (i) *Iterative deepening* where multiple bread-first searches are initiated with successively larger depth limits, (ii) *Directed BFS* where the breadth-first search includes only those neighbors which have performed well in the past, and (iii) Maintaining *local indexes* at each node about all the other nodes within a certain fixed radius to process the query on the behalf of all nodes within that radius. A technique similar to (iii) i.e., maintaining **routing indices** which take into account information about near neighbors and the number of routing hops to reach them, can be seen in [25]. While these schemes perform much better than the basic flooding technique of Gnutella, they are not very applicable to more sophisticated systems that eliminate flooding altogether.

Indexing: Harren *et al.*[50] describe a special indexing scheme, first introduced in Witten *et al.*[119], for pseudo-keyword searching in DHTs. Each key I is split into "t-grams": distinct t-length substrings, and an index of the form $(g_I, I, address)$ is maintained for each t-gram g_I. A search for a resource is also done using the same t-grams, and only those resources which match more than a specified number of t-grams are returned. This helps to find resources with similar names but still does not support more complex queries such as range queries and near matches to a key.

Bloom filters: In **YouSearch** [55, 12], a peer-to-peer system for publishing web documents, each peer uses a bloom filter [15] to create a precise summary of the documents it stores. These bloom filters are periodically exchanged between peers to update content information. Like the earlier scheme, this one also does not support complex queries, and may additionally suffer from the problem of *false positives* i.e., it may appear that a peer stores some resource even though it does not.

Locality-sensitive hashing: Gupta *et al.*[47] use **locality-sensitive hashing** [74, 53] to support answering approximate queries. Their system design substitutes consistent hashing in Chord with locality-sensitive hashing, so that items that are close together in the original space are located close to each other in the hashed space as well. A query for some data item will produce a result that is similar to the requested item. This network not only sacrifices load balancing by using a different hash function but it also does not support answering complex queries *exactly.*

The feasibility of peer-to-peer web indexing and searching is considered in [73]. Each web document is described using a set of keywords and the responsibility for keywords is uniformly distributed among participant peers. Each peer maintains **posting lists** of the addresses of the documents that contain the keywords that it is responsible for. Enhancements to reduce query bandwidth include caching results, pre-computation of posting list intersection for multiple-word queries, bloom filters etc. The authors conclude that while these enhancements are promising, further compromises in result quality and changes in peer-to-peer network structure will be required to make the problem feasible for practical systems.

2.6.2 Load balancing

Even with the use of consistent hashing, a system like Chord could have an imbalance in the load distribution if there is a single peer that is responsible for a large segment of the identifier circle. One solution given in the Chord design is the use of **virtual** peers where each machine is assigned multiple segments to give better load balancing. This increases the space requirement at each node and still does not guarantee perfect load balancing. We look at some of the alternative schemes that have been proposed for better load balancing.

Power of two: Byers *et al.*[17] propose using "the power of two" paradigm for load balancing, where each data item chooses $d \geq 2$ hash functions to determine d candidate nodes on the identifier circle. Prior to insertion, the load on each of those nodes is compared and the data item is inserted at the node with the minimum load. With high probability, this ensures that the maximum load at any of the m nodes with m resources, is at most $\log\log m/\log d + O(1)$. To avoid the extra burden of querying all these possible peers during a search, each of the d initially-chosen peers can store a pointer to the location where the data item is eventually stored.

Sloppy hashing: Coral [37] is yet another variant of Chord that uses "sloppy hashing" for storing data items that may have multiple addresses. Each node maintains a fixed buffer size for storage of the *(key, address)* pairs. When a new pair has to be inserted but the corresponding node is full, the pair is stored at a node one hop away from the original target towards the requesting peer. In addition, Coral tries to maintain clusters of nodes for optimizing network locality: each node is a part of several layers of distributed hash tables each of which consists of other nodes progressively further away from itself in the physical network. Although the sloppy hashing along with cluster management provides limited load balancing, it is not applicable in systems which have one or few copies of each resource because the spilling over technique will not be very effective. In addition, it is expensive to maintain several DHTs for cluster management.

Load transfer: Rao *et al.*[94] explore three different schemes to periodically transfer load between virtual peers: (i) Choose two peers at random and initiate a transfer if one is more overloaded than the other; (ii) Allow an overloaded peers to choose the best of several available under-loaded peers to shed it load; and (iii) Allow several overloaded and under-loaded peers to mutually transfer load among each other. Their simulation results show that it is possible to balance the load within 95% of the optimal value with these schemes.

2.7 Applications

In this section, we give a brief overview of different applications that have been proposed and/or implemented using peer-to-peer systems. We have purposely left out implementation details in this section as the main purpose is to give the reader an idea of the variety of applications that are being built using peer-to-peer systems, rather than the specific details.

2.7.1 Publishing systems

Peer-to-peer systems have been widely used for data storage systems. **Oceanstore** [69] provides a global-scale, persistent data store in which routing to locate files is done using

Tapestry [125], which is described in Section 2.3.2. **Publius** [118] is a censorship-resistant web-publishing system that uses hashing to distribute shares of the encryption key to different peers. **Tangler** [117] is similar to Publius except that it uses a Chord-like base system to distribute actual file data blocks, not just the encryption keys. **CFS** [26] and **PAST** [100, 98] are other data storage systems based on Chord and Pastry respectively.

Mnemosyne [49] is a peer-to-peer **steganographic** storage service based on Tapestry, where files can be stored such that attackers are unable to obtain their contents, but a legitimate user can access them. Each file block is encrypted and written to a randomly chosen location with additional replication to avoid overwriting blocks. Further enhancements to provide fault tolerance include encoding information into t blocks such that any $s \leq t$ blocks are sufficient to retrieve it.

2.7.2 Web publishing and caching

YouServ [55] is a web-publishing system that allows users to collectively use their local machines for web hosting and file serving. Peers are responsible for storing and replicating content, and a coordinator is used to direct queries to the peer that hosts a particular page. Support for hosting web pages for a peer that is currently inaccessible, at other peers makes the content available most of the time.

Squirrel [54] is a decentralized, peer-to-peer web cache developed on top of Pastry [99]. The system pools together local caches of peers to form a global, scalable web cache without the associated overheads such as hardware and maintenance costs. Experimental results show that developing such a system is not only feasible but also compares favorably to a dedicated web server.

2.7.3 Name services

Simple Distributed Security Infrastructure (SDSI) is a proposed public-key infrastructure in which names are defined in local namespaces and longer names can link multiple namespaces to delegate trust using certificates. **ConChord** [2] is a distributed SDSI certificate directory built on Chord, for name resolution over multiple namespaces, and membership checking to see if a certificate is applicable to a particular key.

DHash [24] is a system that uses Chord as an alternative service structure to the DNS; *(host name, address)* pairs are stored in a distributed fashion using hashing, to alleviate some of the drawbacks of using tree data structures for DNS lookups. The authors conclude that in spite of the advantages of a distributed DNS, the high latencies, difficulties in adding new features at all clients, and the lack of incentives to serve unrelated data make the system less attractive than the current DNS. A similar conclusion for using peer-to-peer systems for name services can be seen in Overlook [113] where the authors say that such systems are too sensitive to the transport layer and the client machines. A different perspective that focuses on the advantage of *semantic-free* reference routing by using peer-to-peer for DNS can be seen in [10].

2.7.4 Network performance measurement

Srinivasan *et al.*[110] suggest using a peer-to-peer system for measurement of network performance in **M-coop**. With peers distributed all over the world, such a system would give good a good level of network coverage that would be difficult to achieve by manually selecting endpoints. However, there are several issues such as sufficient participation, usefulness of measurements, and handling collusions which remain to be addressed before implementing such a system in practice.

2.7.5 Managing flash crowds

CoopNet [89] and **Backslash** [111] are systems that address the flash crowd problem, due to a sudden increase in the demand for some data, by proposing that clients that have downloaded the data, serve it to other clients. This is implemented by having the server maintain a list of served clients and redirecting future requests to them. To prevent redirection overload on the server, Backslash further stores resource addresses, either en route to the requesting node in a DHT overlay, or on random peers using hashing.

2.7.6 Group communication

Pastry has been used as the base for **Scribe** [101], an event-notification infrastructure for a topic-based publish/subscribe system. Subscribers register for a topic of their interest and receive events related to that topic irrespective of the publisher. A multicast tree is maintained for each topic which is associated with a **rendez-vous** point i.e., the point in the overlay network that is closest to the topic identifier. Each tree is formed by joining the routes from the subscribers for that particular topic.

Peer-to-peer systems are also used for application-level broadcast [32] as well as multicast [95, 19, 18]. There are two approaches for multicast: *intelligent flooding* used in **M-CAN** [96] based on CAN [95], and building a *multicast tree* as in Scribe [19] and **SplitStream** [18], both based on Pastry [99]. A comparison of the performance of the two approaches under identical workloads [20] shows that the tree approach outperforms the flooding approach.

Chapter 3

Fault-tolerant Routing

In this chapter, we present an abstract model of a distributed hash table, prove our results for fault-tolerant greedy routing in this model, give heuristics for construction and maintenance of such a network, and present experimental results on the performance in practice.

3.1 Overview

Although the various peer-to-peer systems that use distributed hash tables (see Chapter 2 for details) seem vastly different, there is a recurrent underlying theme in the use of some variant of an overlay metric space in which the machines are embedded. The locations of machines and resources in this metric space are determined by their identifiers and keys respectively. Each node maintains some information about its neighbors in the metric space, and routing is then simply done by forwarding each packet to the neighbor *closest* to the target node with respect to the metric. In CAN [95], the metric space is explicitly defined as the coordinate space which is divided into zones and the distance metric used is simply the Euclidean distance. In Chord [112], the nodes are located on grid points on a real circle, with distances measured clockwise along the circumference of the circle. Tapestry [125], and Pastry [99] also place nodes on a real circle although distances can be measured in either direction. This inherent common structure leads to similar results for the performance of such networks. In this chapter, we explain why most of these systems achieve similar performance guarantees by describing a general setting for such overlay metric spaces, although most of our results apply only in one-dimensional spaces.

Further, we are interested in the fault-tolerance properties of such networks. For large systems, where nodes appear and leave frequently, resilience to repeated and concurrent failures is a desirable and important property. We prove that with our overlay space and linking strategies, the network performs reasonably well even with a large number of failures. We also give experimental results that support our theoretical proofs in practice.

Our approach too provides a hash table-like functionality, based on keys that uniquely identify the resources. To accomplish this, we map resources to points in a metric space either directly from their keys or from the keys' hash values. This mapping dictates an assignment of machines to metric-space points. Each machine is assigned a separate location for each resource that it hosts; so a machine can be located at multiple points in the overlay network. We construct and maintain a random graph linking these points and use greedy routing to traverse its edges to find data items. The principle we rely on is that failures leave behind yet another (smaller) random graph, ensuring that the network is robust even in the face of considerable damage. Another compelling advantage of random graphs is that they eliminate the need for global coordination. Thus, we get a fully-distributed, egalitarian, scalable network with no bottlenecks.

We measure performance in terms of the number of messages sent in the network for a search or an insert operation. The repair mechanism may generate additional traffic, but we expect to amortize these costs over the search and insert operations. Given the growing storage capacity of machines, we are less concerned with minimizing the storage at each node; the space requirements are small and the information stored at a node consists only of an IP address for each neighbor. However, as pointed out in Chapter 1, more neighbors lead to an increase in traffic for maintenance of links in the repair mechanism, thus it is beneficial to keep the storage requirement down to a minimum.

The rest of this chapter is organized as follows. Section 3.2 explains our abstract model in detail. We give our lower bounds for greedy routing in Section 3.3. We prove upper bounds on greedy routing without failures in Section 3.4, and in the presence of failures in Section 3.5. In Section 3.6, we present a heuristic method for constructing the random graph and provide experimental results that show its performance in practice. Section 3.7 describes the results of other experiments we conducted to test the performance of greedy routing in our distributed data structure. Conclusions and future work are discussed in Section 3.8.

3.2 Our approach

The idea underlying our approach consists of three basic parts: (1) embed resources as points in a metric space, (2) construct a random graph by appropriately linking these points, and (3) efficiently locate resources by routing greedily along the edges of the graph. Let R be a set of resources spread over a large, heterogeneous network X. For each resource $r \in R$, $owner(r)$ denotes the machine in X that provides r and $key(r)$ denotes the resource's key. Let K be the set of all possible keys. We assume a hash function $h : K \rightarrow V$ such that resource r maps to the point $v = h(key(r))$ in a metric space (V, d), where V is the point set and d is the distance metric as shown in Figure 3.1. We can use a hash function such as SHA-1 [1] to populate the metric space uniformly. Note that via this resource embedding, a machine x is mapped onto the set $V_x = \{v \in V : \exists r \in R,\, v = h(key(r)) \wedge (owner(r) = x)\}$, namely the set of metric-space points assigned to the resources the machine provides. Thus a single machine can have multiple locations in the metric space, one location for each resource that it hosts*.

Our next step is to construct a directed random graph from the points embedded in V. We assume that each newly-arrived machine x is initially connected to some other machine in X. Each machine x generates the outgoing links for each vertex $v \in V_x$ independently. We give details about the number of outgoing links in our model in later sections. A link $(v, u) \in V_x \times V_y$ simply denotes that x knows that y is the machine that provides the resource mapped to u; hence, we can view the graph as a virtual overlay network of information, pieces of which are stored locally at each machine. Machine x constructs each link by executing the search algorithm to locate the end-point of that link. If the metric space is not populated densely enough, the choice of an end-point may result in a vertex

*We note that if there are multiple owners for a resource that has the same key but exists on several machines, they all hash to the same location but each owner creates its own independent links.

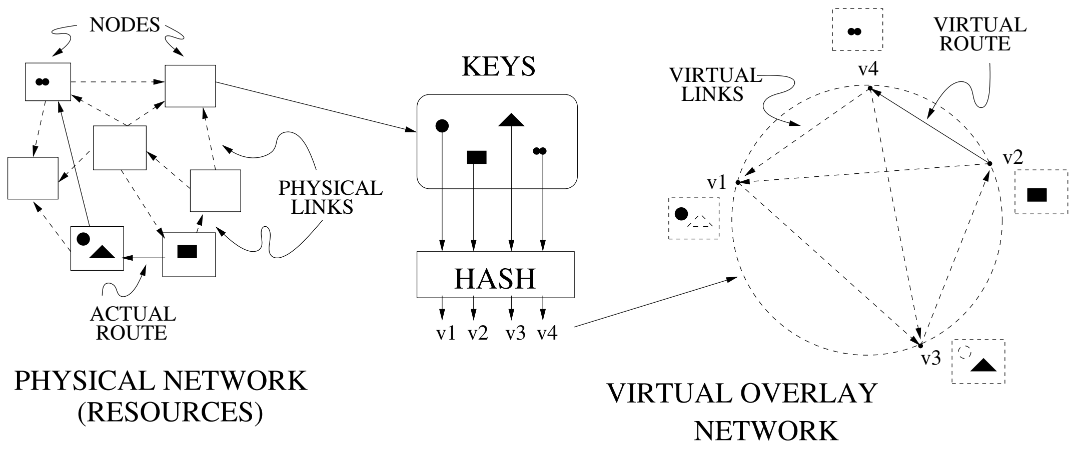


Figure 3.1: An abstract model of a distributed hash table.

corresponding to an absent resource. In that case, x chooses as its neighbor, the point present closest to the original end-point. Moving to nearby vertices will introduce some bias in the link distribution, but the magnitude of error does not appear to be large, as shown in a more detailed description of the graph construction given in Section 3.6.

Having constructed the overlay network of information, we can now use it for resource location. As new machines arrive, old machines depart, and existing ones alter the set of resources they provide or even crash, the resources available in the system change. At any time t, let $R^t \subseteq R$ be the set of available resources and I^t be the corresponding overlay network. A request by machine x to locate resource r at time t is served in a simple, localized manner: x calculates the metric-space point v that corresponds to r, and a request message is then routed over I^t to v from the vertex in V_x that is closest to $v^\dagger$. Each machine needs only local information, namely its set of neighbors in I^t, to participate in the resource location. Routing is done greedily by forwarding the message to the machine mapped to a metric-space point as close to v as possible. The problem of resource location is thus translated into routing on random graphs embedded in a metric space.

To a first approximation, our approach is similar to the "small-world" routing work by Kleinberg [66], in which points in a two-dimensional grid are connected by links drawn from a normalized power-law distribution (with exponent 2), and routing is done by having each node forward each packet to the neighbor closest to the packet's destination. Kleinberg's

† Note that since R^t generally changes with time, and may specifically change while the request is being served, the request message may be routed over a series of different overlay networks I^{t_1}, I^{t_2}, $\ldots$, I^{t_k}.

approach is somewhat brittle because it assumes a constant number of links leaving each node. Getting good performance using his technique also depends on having a complete two-dimensional grid of points. We are not as interested in keeping the degree down and accept a larger degree to get more robustness. Also, we cannot assume a complete grid, because fault tolerance is one of our main objectives, and since machines are mapped to points in the metric space based on what resources they provide, there may be missing points.

The use of random graphs is partly motivated by a desire to keep the data structure scalable and the routing algorithm as decentralized as possible, as random graphs can be constructed locally without global coordination. Another important reason is that random graphs are by nature robust against failures: a node-induced subgraph of a random graph is generally still a random graph; therefore, the disappearance of a few vertices, along with all their incident links (due to failure of some machines implementing the data structure) will still allow routing while the repair mechanism is trying to heal the damage. The repair mechanism also benefits from the use of random graphs, since most random structures require less work to maintain their much weaker invariants compared to more organized data structures.

Embedding the graph in a metric space has the very important property that the only information needed to locate a resource is the location of its corresponding metric-space point. That location is permanent, both in the sense of being unaffected by disruption of the data structure, and easily computable by any node that seeks the resource. So, while the pattern of links between nodes may be damaged or destroyed by failure of nodes or of the underlying communication system, the metric space forms an invulnerable foundation over which to build the ephemeral parts of the data structure.

3.2.1 Comparison with DHTs

Our approach is essentially an abstract model of a DHT with a few modifications. DHTs use machine identifiers and resource keys to determine the locations of the machines and resources respectively in the overlay metric space. Each machine is responsible for some subset of the resources and either maintains the addresses of the other machines that host these resources, or maintains the resources themselves; typically this subset consists of resources that are located near the machine's location in the metric space.

While our model is similar to the DHTs, it is *resource-centric* compared to the *machine-centric* approach of the DHTs. As in DHTs, we propose the use of a metric space as an overlay network and embed machines and resources in this overlay network. However, our approach differs in the way this embedding is actually done. We propose that each machine x in the network has multiple locations V_x in the metric space, one for each resource that it hosts. Thus each vertex $v \in V_x$ represents one resource hosted by machine x. This implies that x has to maintain links as per the chosen link distribution for *each* of its resources; if x has k resources and each vertex in the metric space has ℓ links, x has to maintain $(k\ell)$ links.

Our approach gives the benefit of **content locality** i.e., each machine is responsible for the addresses of the resources that it hosts, which increases manageability and security. No machine needs to place its trust in some other machine to maintain the address of its resources. Another small advantage is that since the resource address is served by the same machine that hosts the resource, no extra communication is required to start the resource transfer after its location has been determined. At the same time, it presents certain difficulties. Each machine has to maintain a large number of links; while this is not an issue in terms of space at each machine as each link is nothing but the IP address of another machine in the system, it can result in a lot of overhead for maintenance as a part of the repair mechanism. If each node periodically checks the status of its neighbors and initiates actions to repair links to faulty neighbors, it can cause a flood of messages given the large number of neighbors per node. In addition, load balancing may be skewed because if a few machines own most of the resources, they will be responsible for an unusually large part of the network maintenance.

On the other hand, DHTs naturally provide load balancing and minimize the number of links per machine. However, they do not provide content locality, and thus inherently assume that all the peers selflessly maintain the addresses of the resources, or the resources themselves, even though they are actually provided by other machines in the system.

We note that we can easily adopt the load balancing properties of DHTs in our design.

Instead of multiple locations for a machine as per its resources, it can have a single location based on the hash value of its identifier. Similar to DHTs, we can also hash the resource keys to determine which machine will be responsible for the resources. Each machine will maintain information about the resources that hash to the same location (or nearby locations) as itself. This gives the natural load balancing properties as well as minimization of space at each machine that are key features of a DHT. But it also leads to the loss of the important property of content locality. Depending on the application that the peer-to-peer system is used for, and depending on whether security or load balancing is more important, one of the two approaches can be suitably used.

In the sections that follow, we present our results which can be applied to both scenarios as we refer to the number of points in the overlay metric space for our results. We do not commit to how the resources are distributed, so a point in the metric space can correspond either to a machine or a resource. Note that when we say *node*, we actually refer to a *vertex* in the virtual overlay network, not a *physical* machine in the network.

3.2.2 Tool

Some of our upper bounds will be proved using a well-known upper bound of Karp *et al.*[60] on probabilistic recurrence relations. We will restate this bound as Lemma 3.1, and then show how a similar technique can be used to get *lower bounds* with some additional conditions in Theorem 3.3.

Lemma 3.1 ([60]) *The time $T(X_0)$ needed for a non-increasing real-valued Markov chain $X_0, X_1, X_2, X_3 \ldots$ to drop to 1 is bounded by*

$$T(X_0) \leq \int_1^{X_0} \frac{1}{\mu_z} dz, \tag{3.1}$$

when $\mu_z = \mathrm{E}[X_t - X_{t+1} : X_t = z]$ is a nondecreasing function of z.

This bound has a nice physical interpretation. If it takes one second to jump down μ_x meters from x, then we are traveling at a rate of μ_x meters per second during that interval. When we move past some position z, we are traveling at the average speed μ_x determined by our starting point $x \geq z$ for the interval. Since μ is nondecreasing, using μ_z as our estimated speed underestimates our actual speed when passing z. The integral computes the time to get all the way to zero if we use μ_z as our instantaneous speed when passing position z. Since our estimate of our speed is low (on average), our estimate of our time will be high, giving an upper bound on the actual expected time.

3.3 Lower bounds

In this section, we present our lower bounds on routing. We consider greedy routing in a graph embedded in a line where each node is connected to its immediate neighbors and to multiple long-distance neighbors chosen according to a fixed link distribution. We give lower bounds for greedy routing for *any* link distribution satisfying certain properties (Theorem 3.3). We only give a brief intuition of the proof here as the bulk of the work was done by the co-author and the details are beyond the scope of this dissertation.

We would like to get lower bounds on the process described in Lemma 3.1, in addition to upper bounds, and we will not necessarily be able to guarantee that μ_z, is a nondecreasing function of z. But we will still use the same basic intuition: The average speed at which we pass z is at most the maximum average speed of any jump that takes us past z. We can find this maximum speed by taking the maximum over all $x > z$; unfortunately, this may give us too large an estimate. Instead, we choose a threshold U for "short" jumps, compute the maximum speed of short jumps of at most U for all x between z and $z + U$, and handle the (hopefully rare) long jumps of more than U by conditioning against them. Subject to this conditioning, we can define an upper bound m_z on the average speed passing z, and use essentially the same integral as in (3.1) to get a lower bound on the time.

We now give a lower bound on the expected time taken by greedy routing on a random graph embedded in a line. Each node in the graph has expected out-degree at most ℓ and is also connected to its immediate neighbor on either side. With n nodes in the metric space, we consider two variants of the greedy routing algorithm and derive lower bounds for them equal to $\Omega(\log^2 n/(\ell^2 \log\log n))$ and to $\Omega(\log^2 n/(\ell \log\log n))$, as stated in Theorem 3.3. For large values of ℓ, a lower bound of $\Omega(\frac{\log n}{\log \ell})$ on the worst-case routing time can be derived very simply, as follows.

Lemma 3.2 *Let $\ell \in (\log n, n^c]$, $0 < c < 1$. Then for any link distribution and any routing strategy, the number of routing hops per search, with n nodes is $T = \Omega(\frac{\log n}{\log \ell})$.*

Proof: With ℓ links for each node, we can reach at most ℓ^k nodes at step k. Assuming that the minimum time to reach all n nodes is T, $\ell^T = n$. This gives a lower bound of $\Omega(\frac{\log n}{\log \ell})$ on T. ∎

For constant values of ℓ, the lower bound is $\Omega(\log^2 n / \log \log n)$, which is substantially higher than the time $O(\log n)$ that can be obtained by building a tree (which requires dependence in the distribution on links). Thus the lower bound shows some of the costs inherent of assuming independence and symmetry between all nodes in the graph. The cost of symmetry is reduced, however, for larger ℓ: with $\ell = \Omega(\log n)$, only $\log \log n$ factors separate the costs of symmetric and asymmetric solutions.

We consider two variants of the greedy routing algorithm. Without loss of generality, we assume that the target of the search is labeled 0[‡]. In **one-sided greedy routing**, the algorithm never traverses a link that would take it past its target. So if the algorithm is currently at v ($v < 0$) and is trying to reach 0, it will move to the node $v - \Delta_i$ with the smallest non-positive label. (A similar action is taken for $v > 0$.) In **two-sided greedy routing**, the algorithm chooses a link that minimizes the distance to the target, without regard to which side of the target the other end of the link is. In this case, the algorithm will move to a node $v - \Delta_i$ whose label has the smallest absolute value, with ties broken arbitrarily. One-sided greedy routing can be thought of as modeling algorithms on a graph with a boundary when the target lies on the boundary, or algorithms where all links point in only one direction (as in Chord [112]).

Our results are stronger for the one-sided case than for the two-sided case. With one-sided greedy routing, we show a lower bound of $\Omega(\log^2 n/(\ell \log \log n))$ on the time to reach 0 from a point chosen uniformly from the range 1 to n that applies to any link distribution. For two-sided routing, we show a lower bound of $\Omega(\log^2 n/(\ell^2 \log \log n))$, with some

‡We assume that nodes are labeled by integers and identify each node with its label.

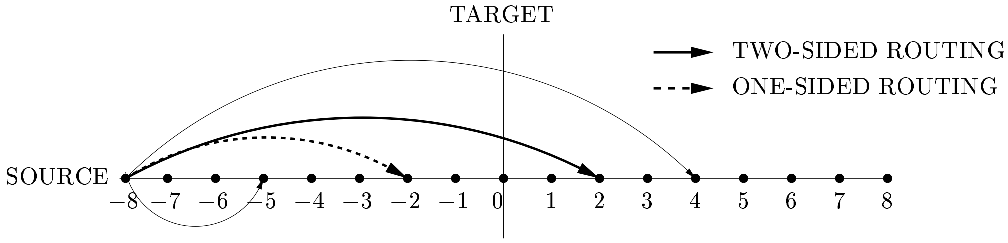


Figure 3.2: Different links are chosen when going from the source node towards the target node, depending on whether the greedy routing used is one-sided or two-sided.

constraints on the distribution. We conjecture that these constraints are unnecessary, and that $\Omega(\log^2 n/(\ell \log\log n))$ is the correct lower bound for both models.

Theorem 3.3 *Let G be a random graph whose nodes are labeled by the integers. Let Δ_v for each v be a set of integer offsets chosen independently from some common distribution, subject to the constraint that -1 and $+1$ are present in every Δ_v, and let node v have an outgoing link to $v - \delta$ for each $\delta \in \Delta_v$. Let $\ell = \mathrm{E}[|\Delta|]$. Consider a greedy routing trajectory in G starting at a point chosen uniformly from $1 \ldots n$ and ending at 0.*

With one-sided routing, the expected time to reach 0 is

$$\Omega\left(\frac{\log^2 n}{\ell \log\log n}\right). \tag{3.2}$$

With two-sided routing, the expected time to reach 0 is

$$\Omega\left(\frac{\log^2 n}{\ell^2 \log\log n}\right), \tag{3.3}$$

provided Δ is generated by including each δ in Δ with probability p_δ, where (a) p is unimodal, (b) p is symmetric about 0, and (c) the choices to include particular δ, δ' are pairwise independent.

We also believe that the bound continues to hold in higher dimensions than 1 but a simple extension of the proof of Theorem 3.3 is not sufficient to prove the same. This lower bound explains why most of the DHT systems give similar performance with $O(\log n)$ links per node using greedy routing[§].

[§]Viceroy [78] is an exception that reduces the space cost to $O(1)$ by the clever use of butterfly routing in the last stages of the search.

3.4 Upper bounds without failures

In this section, we present upper bounds on the number of routing hops for a search operation in a simple metric space: a one-dimensional real line. To simplify theoretical analysis, the network is set up as follows:

- Nodes are located at grid points on the real line. Each node v is connected to its nearest neighbor on either side, and to ℓ long-distance neighbors. Let n be the number of nodes.

- With $\ell \in [1, \log n]$, the long-distance neighbors are chosen as per the inverse power-law distribution with exponent 1, i.e., each long-distance neighbor u is chosen with probability inversely proportional to the distance between v and u. Formally,

  $$\Pr[u \text{ is a neighbor of } v] = \frac{1/d(v,u)}{\sum_{u' \neq v} 1/d(v,u')},$$

  where $d(v,u)$ is the distance between nodes v and u in the metric space. With $\ell \in (\log n, n^c]$, $0 < c < 1$, we use a deterministic linking strategy which is explained in detail later in this section.

- Routing is done greedily by forwarding the message to the neighbor closest to the target node.

We analyze the performance for the cases of a single long-distance link and of multiple links, in a failure-free network in this section, and in a network with link and node failures in the next section.

3.4.1 Single long-distance link

We first analyze the number of routing hops for a search in an idealized model with no failures and with one long-distance link per node. Kleinberg [66] proved that with n^d nodes embedded at grid points in a d-dimensional grid, with each node v connected to its immediate neighbors and one long-distance neighbor u chosen with probability proportional to $1/d(v,u)^d$, any message can be delivered in time polynomial in $\log n$ using greedy routing.

While this result can be directly applied to our model with n nodes, $d = 1$, and $\ell = 1$ to give $O(\log^2 n)$ routing hops per search, we get a much simpler proof using Lemma 3.1. We include the proof below for completeness.

Theorem 3.4 *Let each node be connected to its immediate neighbors (at distance 1) and* 1 *long-distance neighbor chosen with probability inversely proportional to its distance from the node. Then the expected number of routing hops per search, with* n *nodes is* $T(n) = O(H_n^2)$.

Proof: Let μ_k be the expected number of nodes crossed when the message is at a node s that is at a distance k from the destination t. Clearly, μ_k is non-decreasing.

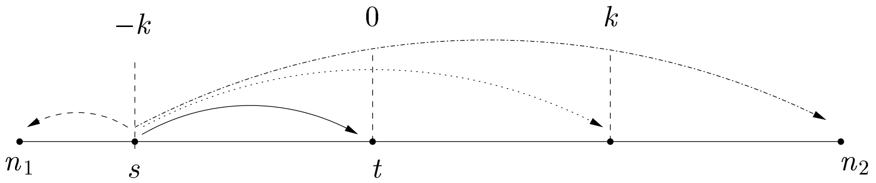


Figure 3.3: All the possible nodes that can be reached from the source node s en route to the destination node t.

From Figure 3.3, we see that the routing algorithm traverses the long-distance link from s if it ends in $(-k, k)$; otherwise the short-hop link to the immediate neighbor closer to the destination t is taken. Thus we get:

$$\mu_k = \frac{\sum_{i=1}^{k} \frac{1}{i} \cdot i}{S} + \frac{\sum_{i=1}^{k-1} \frac{1}{2k-i} \cdot i}{S} + \frac{\sum_{i=1}^{n_1-k} \frac{1}{i} \cdot 1}{S} + \frac{\sum_{i=2k}^{n_2+k} \frac{1}{i} \cdot 1}{S},$$

where

$$\begin{aligned} S &= \sum_{i=1}^{n_1-k} \frac{1}{i} + \sum_{i=1}^{n_2+k} \frac{1}{i} \\ &= H_{n_1-k} + H_{n_2+k} \\ &< 2H_n \end{aligned}$$

Then

$$\mu_k > \frac{1}{S}[k + 0 + H_{n_1-k} + H_{n_2+k} - H_{2k}] > \frac{k}{S} > \frac{k}{2H_n}$$

Clearly, μ_k is non-decreasing, and thus using Lemma 3.1, we get

$$\begin{aligned} T(n) &\leq \sum_{k=1}^{n} \frac{1}{\mu_k} \\ &= \sum_{k=1}^{n} \frac{2H_n}{k} \\ &= O(H_n^2) \end{aligned}$$

Thus we see that with this distribution, the number of routing hops per search is logarithmic in the number of nodes. ∎

3.4.2 Multiple long-distance links

The next interesting question is whether we can reduce the number of routing hops per search by using multiple long-distance links instead of a single one. In addition to improvement in performance, multiple links also give the benefit of robustness in the face of failures. We first look at improvement in performance by using multiple links in the network and then go on to analysis of failures in Section 3.5. We consider different strategies for generating links and routing depending on number of long-distance links ℓ in two ranges: $\ell \in [1, \log n]$ and $\ell \in (\log n, n^c]$, $0 < c < 1$. Let us first consider our randomized strategy for link distribution when $\ell \in [1, \log n]$.

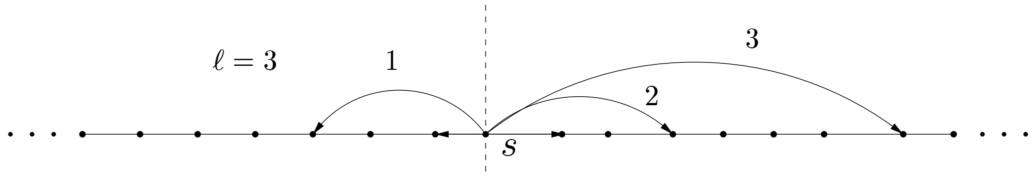


Figure 3.4: Multiple (ℓ) long-distance links for each node s.

In [67], Kleinberg uses a group structure to get $O(\log n)$ routing hops for the case of a polylogarithmic number of links. However, he uses a more complicated algorithm for routing while we obtain the same bound (for the case of a line) using only greedy routing. We prove that with $\ell \in [1, \log n]$ long-distance links, the expected number of routing hops per search is $O(\log^2 n/\ell)$. The basic idea for the proof comes from Kleinberg's model [66].

Kleinberg considers a two-dimensional grid with nodes at every grid point. The routing of the message is divided into phases. A message is said to be in phase j if the distance from the current node to the destination node is between 2^j and 2^{j+1}. There are at most $(\log n + 1)$ such phases. He proves that the expected time spent in each phase is at most $O(\log n)$, thus giving a total upper bound of $O(\log^2 n)$ on the number of routing hops for routing. We use the same phase structure in our model, and our proof is along similar lines.

Theorem 3.5 *Let each node be connected to its immediate neighbors (at distance 1) and ℓ long-distance neighbors chosen independently with replacement, with probability inversely proportional to their distances from the node. Let $\ell \in [1, \log n]$. Then the expected number of routing hops per search, with n nodes is $T(n) = O(\log^2 n/\ell)$.*

Proof: In our multiple-link model, each node has ℓ long-distance neighbors chosen with replacement. The probability that v chooses a node u as its long-distance neighbor is $1 - (1-q)^\ell$, where $q = \frac{d(v,u)^{-1}}{\sum_{u' \neq v} d(v,u')^{-1}}$. We can get a lower bound on this probability as follows:

$$\begin{aligned}
1-(1-q)^\ell &> 1-(1-q\ell+\frac{\ell(\ell-1)}{2}q^2) \\
&= q\ell - \frac{\ell(\ell-1)}{2}q^2 \\
&= q\ell\left[1-\frac{(\ell-1)q}{2}\right] \\
&= q\ell\left[1-\frac{\ell q}{2}+\frac{q}{2}\right] \\
&\geq q\ell\left[1-\frac{\ell q}{2}\right].
\end{aligned}$$

Observe that $\ell q < 1$, because $q < \frac{1}{\log n}$ and $\ell \leq \log n$. So, the probability that v chooses u as its long-distance neighbor is at least

$$\begin{aligned}
q\ell\left[1-\frac{\ell q}{2}\right] &\geq q\ell\left[1-\frac{1}{2}\right] \\
&= \frac{q\ell}{2} \\
&= \ell[2d(v,u)H_n]^{-1}
\end{aligned}$$

Now suppose that the message is currently in phase j. To end phase j at this step, the message should enter a set of nodes B_j at a distance $\leq 2^j$ of the destination node t. As the nodes are embedded on a real line, there are at least 2^j nodes in B_j, each within distance $2^{j+1} + 2^j < 2^{j+2}$ of v. So the message enters B_j with probability $\geq 2^j \ell \frac{1}{2H_n 2^{j+2}} = \frac{\ell}{8H_n}$.

Let X_j be the total number of steps spent in phase j. Then

$$\begin{aligned} EX_j &= \sum_{i=1}^{\infty} Pr[X_j \geq i] \\ &\leq \sum_{i=1}^{\infty} \left(1 - \frac{\ell}{8H_n}\right)^{i-1} \\ &= \frac{8H_n}{\ell}. \end{aligned}$$

Now if X denotes the total number of steps, then $X = \sum_{j=0}^{\log n} X_j$, and by linearity of expectation, we get $EX \leq (1 + \log n)(8H_n/\ell) = O(\log^2 n/\ell)$. ∎

For $\ell \in (\log n, n^c]$, , $0 < c < 1$, we use a deterministic strategy. We represent the location of each node as a number to a base $b \geq 2$, and generate links to nodes at distances $1x, 2x, 3x, \ldots, (b-1)x$, for each $x \in \{b^0, b^1, \ldots, b^{\lceil \log_b n \rceil - 1}\}$. Thus the number of links $\ell = (b-1)\lceil log_b n \rceil$. Routing is done by eliminating the most significant digit of the distance at each step. As the maximum distance can be at most $b^{\lceil \log_b n \rceil}$, we get $T(n) = O(\log_b n)$. This strategy is similar in spirit to the PRR algorithm [92] explained in Section 2.2.2.

Some special cases are instructive. Let $\ell = O(\log n)$, and let each node link to nodes in both directions at distances $2^i, 1 \leq i \leq 2^{\log n - 1}$, provided nodes are present at those distances. This gives $T(n) = O(\log n)$. Similarly, let $\ell = O(\sqrt{n})$. Links are established in both directions to existing nodes at distances $1, 2, 3, \ldots, \sqrt{n}, 2\sqrt{n}, 3\sqrt{n}, \ldots, \sqrt{n}(\sqrt{n} - 1)$, giving $T(n) = O(1)$. In fact, $T(n) = O(1)$ when $b = n^c$, for any fixed $c > 0$.

Theorem 3.6 *Choose an integer $b > 1$. With $\ell = (b-1)\lceil \log_b n \rceil$, let each node link to ℓ nodes at distances $1x, 2x, 3x, \ldots, (b-1)x$, for each $x \in \{b^0, b^1, \ldots, b^{\lceil \log_b n \rceil - 1}\}$. Then the number of routing hops per search, with n nodes is $T(n) = O(\log_b n)$.*

Proof: Let $d_1, d_2, \ldots d_t$ be the distances of the successive nodes in the routing path from the target t, where d_1 is the distance of the source node and $d_t = 0$. For each

$d_i, \exists k_i \in \{0, 1, \ldots, \lfloor \log_b n \rfloor\}$ such that $b^{k_i} \leq d_i < b^{k_i+1}$. Hence $1 \leq \lfloor \frac{d_i}{b^{k_i}} \rfloor < b$. Now each node is connected to the node at distance $b^{k_i} \lfloor \frac{d_i}{b^{k_i}} \rfloor$. We get

$$d_{i+1} = d_i - b^{k_i} \lfloor \frac{d_i}{b^{k_i}} \rfloor = d_i \mod b^{k_i} < b^{k_i}.$$

Thus k_i drops by at least 1 at every step. As $k_1 \leq \lceil \log_b n \rceil$, we get $T(n) = O(\log_b n)$. ∎

3.5 Upper bounds with failures

Failures in the system can be of two types: *machine* failures, where a peer can crash, or *link* failures which are transient system outages. We do not make any guarantees of accessing the data which is stored on a machine that fails. However, we are interested in seeing how the network performs in the presence of these failures. Each search follows a trail of machines until it reaches the destination. If any of these intermediate machines have failed, or connecting links are broken, the search can fail. If the number of such failed searches is small, then we can restart the failed searches. If the number is large, then we can use some smart searching techniques, for example, re-routing the search to another random machine in the network or backtracking, or we can even start the initial search in parallel in the hope that one of the searches will succeed. In this section, we analyze the performance of the searches in the presence of such failures.

One solution is to provide fault tolerance by **replication**. Maintaining several copies provides survivability, allows load balancing over different replicas, and helps eliminates untrustworthy peers by validation. Suitable placement of replicas can also help to eliminate co-related failures such as system outages or natural disasters. However, it requires additional maintenance for replica management and load balancing between replicas. We focus on the inherent fault tolerance of the data structure to see how it performs in the presence of faults without additional mechanisms to repair these faults.

Although we analyze only crash failures, there are other kinds of failures that can occur in the system. A network partition constitutes a more serious failure because it may completely isolate some part of the network from the rest of it. In such a case, a search may come very close to the destination but it may be unable to reach it because all the neighbors of the target are inaccessible. To detect and repair partitions, each node can keep track of some other random nodes in the network that it knows of, and in the absence of any search activity, ping these nodes to check it if has been partitioned off from the rest of the network. We restrict our focus to performance of searches within a connected component of our data structure.

An even worse type of failure to deal with is a **Byzantine** failure, when a machine continues to participate in the network but its actions becomes untrustworthy. A byzantine participant could intercept searches and send out bogus messages, or worse, lie about its position in the data structure. Douceur [31] showed that if a single faulty entity can represent multiple identities and infiltrate the network, it can control a substantial fraction of the network. We defer the issue of dealing with byzantine failures to future work.

3.5.1 Failure of links

We start by analyzing the performance of our network in the presence of link failures in the overlay network. A link between two nodes u and v in the overlay network could fail if the network connection between the two machines that map to u and v, is faulty. It appears that our linking strategies may fail to give the same number of routing hops per search in case the links fail. However, we show that we get reasonable performance even with link failures. In our model, we assume that each link is present independently with probability p. Let us first look at the randomized strategy with $\ell \in [1, \log n]$ long-distance links.

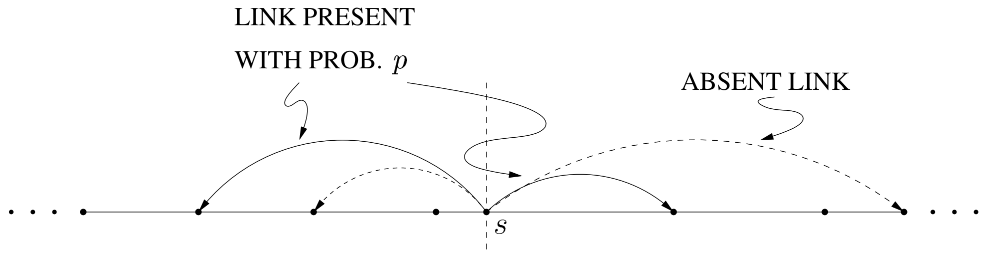


Figure 3.5: Each long-distance link is present with probability p.

Our proof is along similar lines as our proof for the case of no failures. Intuitively, since some of the links fail, we expect to spend more time in each phase and this time should be inversely proportional to the probability with which the links are present. We prove that with each link present with probability p, the expected time spent in each phase is $O(\log n/p\ell)$, which gives a total of $O(\log^2 n/p\ell)$ routing hops per search.

Theorem 3.7 *Let the model be as in Theorem 3.5. If the probability of a long-distance link being present is p, then the expected number of routing hops per search, with n nodes is* $O(\log^2 n/p\ell)$.

Proof: Recall that in case of no link failures, the probability that v chooses a node u as its long-distance neighbor is at least $q\ell/2$ where $q = \frac{d(v,u)^{-1}}{\sum_{u' \neq v} d(v,u')^{-1}}$.

Now when we consider link failures, given that v chose u as its long-distance neighbor, the probability that there is a link present between v and u is p. So, the probability that v chooses a node u as its long-distance neighbor is at least $pq\ell/2 = p\ell[2d(u,v)H_n]^{-1}$.

The rest of the proof is the same as the proof for theorem 3.5. Let X_j be the number of steps spent in phase j. Then

$$\begin{aligned} EX_j &= \sum_{i=1}^{\infty} Pr[X_j \geq i] \\ &= \frac{8H_n}{p\ell} \end{aligned}$$

If X denotes the total number of steps, then by linearity of expectation, we get $EX \leq (1 + \log n)(8H_n/p\ell) = O(\log^2 n/p\ell)$. ∎

We turn to the deterministic strategy with $\ell \in (\log n, n^c]$, $0 < c < 1$. If a link fails, then the node has to take a shorter long-distance link, which will not take the message as close to the target as the initial failed link. Clearly as p decreases, the message has to take shorter and shorter links which increases the number of routing hops.

To make the analysis simpler, we change the link model slightly, and let each node be connected to other nodes at distances $b^0, b^1, b^2, \ldots, b^{\lfloor \log_b n \rfloor}$. Once again, we compute the expected distance covered from the current node and use Lemma 3.1 to get a total of $O(b \log n/p)$ routing hops.

Theorem 3.8 *Let the number of links be $O(\log_b n)$, and let each node have a link to distances $b^0, b^1, b^2, \ldots, b^{\lfloor \log_b n \rfloor}$. If the probability of a link being present is p, then the expected number of routing hops per search, with n nodes is $T(n) = O(bH_n/p)$.*

Proof: Let the distance of the current node from the destination be k. Let μ_k represent the distance covered starting from this node. Then with probability p, there will be a link covering distance $b^{\lfloor \log_b k \rfloor}$. If this link is absent with probability $q = 1 - p$, then we can cover a distance $b^{\lfloor \log_b k \rfloor - 1}$ with a single link with probability pq, and so on. In general, the average distance μ_k covered when the message is at distance k from the destination is

$$
\begin{aligned}
\mu_k &= pb^{\lfloor \log_b k \rfloor} + pqb^{\lfloor \log_b k \rfloor - 1} + \ldots + pq^{\lfloor \log_b k \rfloor - 1} b^1 + q^{\lfloor \log_b k \rfloor} b^0 \\
&\geq \sum_{i=0}^{\lfloor \log_b k \rfloor} pb^{\lfloor \log_b k \rfloor - i} q^i \\
&= pb^{\lfloor \log_b k \rfloor} \sum_{i=0}^{\lfloor \log_b k \rfloor} \left(\frac{q}{b}\right)^i \\
&= pb^{\lfloor \log_b k \rfloor} \frac{1 - (q/b)^{\lfloor \log_b k \rfloor + 1}}{1 - (q/b)} \\
&= \frac{p(b^{\lfloor \log_b k \rfloor + 1} - q^{\lfloor \log_b k \rfloor + 1})}{b - q} \\
&\geq \frac{p(bk/b - 1)}{b - q} \\
&\geq \frac{p(k - 1)}{2(b - q)}.
\end{aligned}
$$

Using Lemma 3.1, we get

$$
\begin{aligned}
T(n) &\leq \sum_{k=1}^{n} \frac{1}{\mu_k} \\
&= 1 + \sum_{k=2}^{n} \frac{2(b - q)}{p(k - 1)} \\
&= 1 + \frac{2(b - q)}{p} \left[\sum_{k=2}^{n} \frac{1}{(k - 1)} \right] \\
&= O(bH_n/p)
\end{aligned}
$$

As p decreases, the number of routing hops increases; whereas as b decreases, the number of routing hops decreases but the information stored at each node increases. ∎

3.5.2 Failure of nodes

A node failure in the overlay network implies that some vertex v has failed. This situation arises if the resource mapped to point v is no longer available, or if the machine x that hosts that resource has crashed. In the latter case, all points $v \in V_x$ that x maps to, will also fail.

We consider two different cases of node failures to study their effect on routing performance. In the first case, all the nodes only link to other nodes that have *not already* failed. In the second case, failures occur after all the links are established.

Binomially distributed nodes Let p be the probability that a node is present. Here also, each node is connected to its nearest neighbors and one long-distance neighbor. In addition, the probability of choosing a particular node as a long-distance neighbor is conditioned on the existence of that node.

Theorem 3.9 *Let the model be as in Theorem 3.4. Let each node be present with probability p and all nodes link only to existing nodes. Then the expected number of routing hops per search, with n nodes is $O(H_n^2)$.*

Proof: We bound the expected drop μ_k from point k as follows:

$$\begin{aligned}
\mu_k &= \frac{\sum_{i=1}^{k} \frac{1}{i} \cdot i \cdot p}{p \cdot S} + \frac{\sum_{i=1}^{k-1} \frac{1}{2k-i} \cdot i \cdot p}{p \cdot S} + \frac{\sum_{i=1}^{n_1-k} \frac{1}{i} \cdot 1 \cdot p}{p \cdot S} + \frac{\sum_{i=2k}^{n_2+k} \frac{1}{i} \cdot 1 \cdot p}{p \cdot S} \\
&> \frac{1}{S}[k + 0 + H_{n_1-k} + H_{n_2+k} - H_{2k}] \\
&> \frac{k}{S} > \frac{k}{2H_n}.
\end{aligned}$$

Using Lemma 3.1, we get $T(n) \leq \sum_{k=1}^{n} 1/\mu_k = O(H_n^2)$. This is exactly the same result that we get in Section 3.4.1 where all the nodes are present. ∎

This result is not surprising because if nodes link only to other existing nodes, the only difference is that we get a smaller random graph. This does not affect the routing algorithm or the number of routing hops per search

General failures We observe that the analysis for node failures is not as simple as that for link failures, because we no longer have the important property of independence that we have in the latter case. In the case of link failures, the nodes first choose their neighbors and then it is possible that some of these links fail; thus, the event that a node is connected to another node is completely independent of the event that, say, its neighbor is connected to the same node. Each link fails independently, and so the accessibility of a target node from any other node depends only on the presence of the link between the two nodes in question.

In case of node failures, this important independence property is no longer true. Suppose that a node v cannot communicate with some other node u (because u failed), even though there may be a functional link between v and u. Now the probability of some other node w being able to communicate with u is not independent of the probability that v can communicate with u, because the probability of u being absent is common for both the cases. This complicates the analysis of the performance because it is no longer the case that if one node cannot communicate with some other node, it has a good chance of doing so by passing the message to its neighbor.

In order to analyze this situation, we consider jumps only to one phase lower rather than jumping over several phases. The idea is that the jumps between phases are independent, so once we move from phase j to phase $j-1$, further routing no longer depends on any nodes in phase j. We can condition on the number of nodes being alive in the lower phase and estimate the time spent in each phase. Intuitively, if a node is present with probability p, we would expect to wait for a time inversely proportional to p in anticipation of finding a node in the lower phase to jump to.

Theorem 3.10 *Let the model be as in Theorem 3.5, and let each node fail with probability p. Then the expected number of routing hops per search, with n nodes is $O(\log^2 n/(1-p)\ell)$.*

Proof: Let T be the time taken to drop down from phase j to phase $j-1$. Let m out of M nodes be alive in phase $j-1$, and let q be the probability that a node in phase j is connected to some node in phase $j-1$. Then the expected time T to drop to phase $j-1$, given that there are m live nodes in it (each of the M nodes are equally likely to have failed), is given by

$$E[T|m] = 1 + \left[(1-q) + \frac{q(M-m)}{M}\right] E[T|m] = \frac{M}{qm}.$$

As m can vary between 1 and M[¶], we get

$$\begin{aligned}
E[T] &= \sum_{m=1}^{M} \frac{M}{qm} \left[p^{M-m}(1-p)^m \binom{M}{m}\right] \\
&= \frac{M}{q} \sum_{m=1}^{M} \frac{1}{m} p^{M-m}(1-p)^m \binom{M}{m} \\
&\leq \frac{M}{q} \sum_{m=1}^{M} \frac{2}{m+1} p^{M-m}(1-p)^m \binom{M}{m} \\
&= \frac{2M}{q(M+1)(1-p)} \sum_{m=1}^{M} p^{M-m}(1-p)^{m+1} \binom{M+1}{m+1} \\
&\leq \frac{2M}{q(M+1)(1-p)} \left[p + (1-p)\right]^{M+1} \\
&= \frac{2M}{q(M+1)(1-p)}.
\end{aligned}$$

Not surprisingly, the expected waiting time in a phase is inversely proportional to the probability of being connected to a node in the lower phase and to the probability of such a node being alive.

For our randomized routing strategy with $\ell \in [1, \log n]$ long-distance links, $q \approx \ell / H_n$. With at most $(\log n + 1)$ phase, we get an expected $O(\log^2 n/(1-p)\ell)$ routing hops. ∎

In contrast, for our deterministic routing strategy, certain carefully chosen node failures can lead to dismal situations where a message can get stuck in a local neighborhood with no hope of getting out of it or eventually reaching the destination node. We conjecture that this should be a low-probability event, so its occurrence will not affect the number of routing hops considerably.

[¶]Note that m cannot be 0 because if there are no live nodes in the lower phase, the routing fails at this point.

3.6 Construction of graphs

As the group of nodes present in the system changes, so does the graph of the virtual overlay network. In order for our routing techniques to be effective, the graph must always exhibit the property that the likelihood of any two vertices v and u being connected is $\Omega(1/d(v,u))$, where $d(v,u)$ is the distance between the two nodes v and u. We describe a heuristic approach to construct and maintain a random graph with such an invariant.

Since the choice of links leaving each vertex is independent of the choices of other vertices, we can assume that points in the metric space are added one at a time. Let v be the k-th point to be added. Point v chooses the end-points of its outgoing links according to the inverse power-law distribution with exponent 1 and connects to them by running the search algorithm. If a desired point u is not present, v connects to u's closest live neighbor. In effect, each of the $k-1$ points already present before v is surrounded by a basin of attraction, collecting probability mass in proportion to its length. However, this creates a skew in the link distribution as links are only established from new nodes to older nodes. So we amend our strategy to allow older nodes also to establish links to new nodes.

Let v be a new point. We give earlier points the opportunity to obtain outgoing links to v by having v:

1. Calculate the number of incoming links it "should" have from points added before it arrived, and
2. Choose such points according to the inverse power-law distribution with exponent 1[‖].

If ℓ is the number of outgoing links for each point, then ℓ will also be the expected number of incoming links that v has to estimate in step (1). We approximate the number of links ending at v by using a Poisson distribution with rate ℓ, that is, the probability that v has k incoming links is $\frac{e^{-\ell}\ell^k}{k!}$, and the expectation of the distribution is ℓ.

After step (2) is completed by v, each chosen point u responds to v's request by choosing one of its existing links to be replaced by a link to v. The choice of the link to replace can vary. We use a strategy that builds on the work of Sarshar *et al.*[105]. In that work,

[‖]All this can be easily calculated by v since the link probabilities are symmetric.

the authors use ideas of Zhang *et al.*[123] to build a graph where each node has a single long-distance link to a node at distance d with probability $1/d$. When a node with a long-distance link at distance d_1 encounters a new node at distance d_2, either due to its arrival or due to a data request, it replaces its existing link with probability $p_2/(p_1 + p_2)$, where $p_i = 1/d_i$, and links to the new node. We extend this idea to our case of multiple long-distance links. Consider a node u with k neighbors at distances $d_1, d_2, \ldots, d_k$. When a new node v at distance d_{k+1} requests an incoming link from u, u replaces one of its existing links with a link to v with probability $p_{k+1}/\sum_{j=1}^{k+1} p_j$. This is a trivial extension of the formula $p_2/(p_1+p_2)$ of [105]. However, this probability must now be distributed among u's k existing long-distance links since u needs to choose one of them to redirect to v. We choose to do that according to the inverse power-law distribution with exponent 1, that is, u chooses to replace its link to the node at distance d_i, $1 \leq i \leq k$, with probability $p_i/\sum_{j=1}^{k} p_j$. Hence, the probability that u decides to link to v and decides to replace its existing link to the node at distance d_i with a link to v is equal to $(p_i/\sum_{j=1}^{k} p_j) \cdot (p_{k+1}/\sum_{j=1}^{k+1} p_j)$. Notice that u may decide not to redirect any of its existing links to v with probability $1 - p_{k+1}/\sum_{j=1}^{k+1} p_j$. The intuition for using such replacement strategy comes from the invariant that we want to maintain dynamically as new nodes arrive: u has a link to a node i at distance d_i with probability inversely proportional to d_i; hence, conditioning on u having k long-distance links, the following equation must hold:

$$
\begin{aligned}
\text{Prob}\,[u \text{ replaces link to } i \text{ with link to } v] &= \text{Prob}\,[u \text{ has a link to } i \text{ before } v \text{ arrives}] \\
&\quad - \text{Prob}\,[u \text{ has a link to } i \text{ after } v \text{ arrives}] \\
&= \frac{p_i}{\sum_{j=1}^{k} p_j} - \frac{p_i}{\sum_{j=1}^{k+1} p_j} \\
&= \frac{p_i}{\sum_{j=1}^{k} p_j} \cdot \frac{p_{k+1}}{\sum_{j=1}^{k+1} p_j}
\end{aligned}
$$

We note that the same heuristic can be used for a repair mechanism. A node can periodically ping its neighbors to check if they are still alive. Suppose a node detects that some its neighbors have failed. Using the above heuristic, it can choose some other nodes in the network to link to. If most of the neighbors have failed, then the node can just re-insert itself in the network.

To analyze the performance of the heuristic in practice, we used it to construct a fully-populated network of $n = 2^{17}$ nodes embedded at grid points on a real line. Each node had $\log n$ (17) links. After averaging the results over the ten networks, we plotted the distribution of long-distance links derived from the heuristic, along with the ideal inverse power-law distribution with exponent 1, as shown in Figure 3.6. We see that the derived distribution tracks the ideal one very closely, with the largest absolute error being roughly equal to 0.023 for links of length 2, as shown in the graph of Figure 3.7.

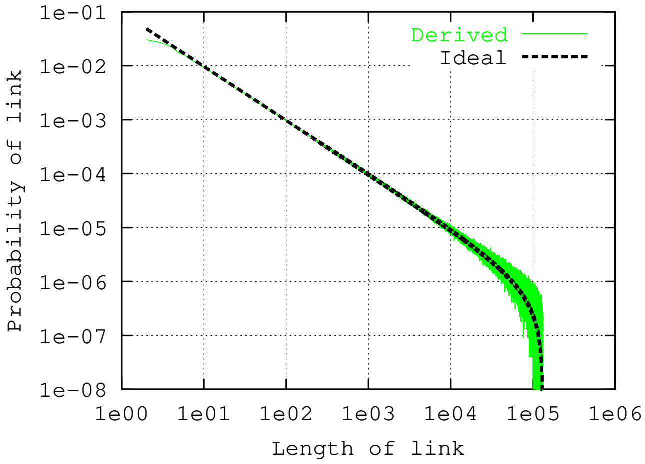


Figure 3.6: The distribution of long-distance links produced by the inverse-distance heuristic (Derived) compared to the ideal inverse power-law distribution with exponent 1 (Ideal).

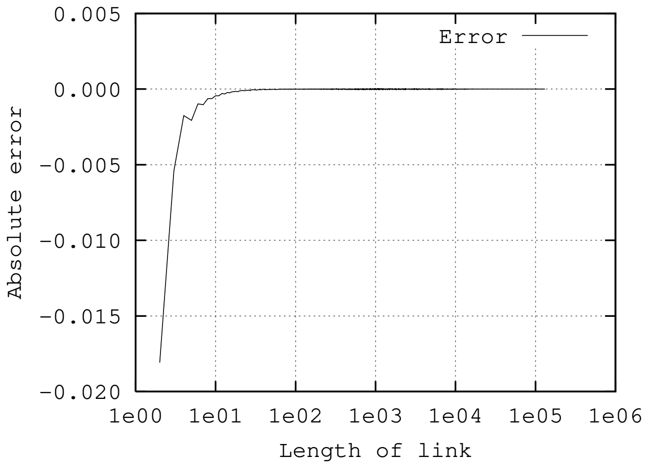


Figure 3.7: The absolute error between the derived distribution and the ideal inverse power-law distribution with exponent 1.

We also performed experiments for an alternative link replacement strategy: a node chooses its *oldest* link to replace with a link to the new node. The performance of this strategy is almost as good as the performance of our replacement strategy described previously. We omit those results because it is difficult to distinguish between the results of the two strategies on the scale used for our graphs.

There has also been other related work by Pandurangan *et al.*[90] on how to construct, with the support of a central server, random graphs with many desirable properties, such as small diameter and guaranteed connectivity with high probability. Although it is not clear what kind of fault-tolerance properties this approach offers if the central server crashes, or how the constructed graph can be used for efficient routing, it is likely that similar techniques could be useful in our setting.

3.7 Experimental results

To measure the performance of our data structure in the presence of failures, we simulated a network of $n = 2^{17}$ nodes at the application level, not the IP level. Each node was connected to its immediate neighbors and had $\log n$ (17) long-distance links chosen as per the inverse power-law distribution with exponent 1 as explained in Section 3.4. Routing was done greedily by forwarding each message to the neighbor closest to the target node. In each simulation, the network was set up afresh, and a fraction p of the nodes failed. We then repeatedly chose random source and destination nodes that had not failed and routed a message between them. For each value of p, we ran 1000 simulations, delivering 100 messages in each simulation, and averaged the number of routing hops for successful searches and the number of failed searches.

With node failures, a node may not be able to find a live neighbor that is closer to the target node than itself. We studied three possible strategies to overcome this problem as follows:

1. Terminate the search.

2. Randomly choose another node, deliver the message to this new node, and then try

to deliver the message from this node to the original destination node (similar to the hypercube routing strategy of Valiant [116]). We use this strategy only once; if the chosen random node has failed, or if there is a dead node on the either the path from the original source to the random node or on the path from the random node to the destination node, the search fails.

3. Keep track of a fixed number (in our simulations, five) of nodes through which the message is last routed and backtrack. When the search reaches a node from where it cannot proceed, it backtracks to the most recently visited node from this list and chooses the next best neighbor to route the message to.

For all these strategies we note that once a node chooses its best neighbor in some step of routing, it does not send the message to any other link if it finds out that the best neighbor has failed, unless it gets another message as a part of the backtracking strategy.

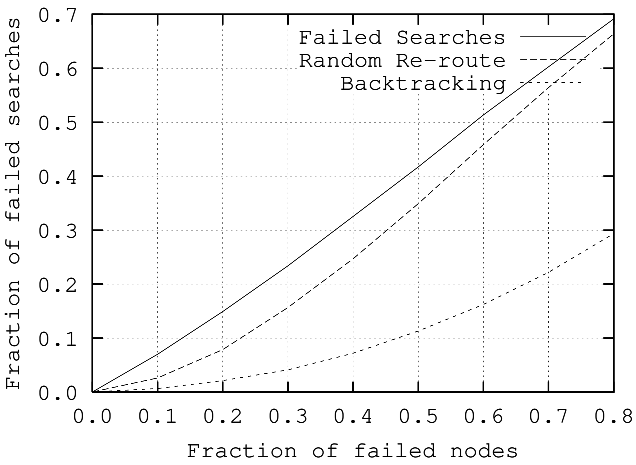


Figure 3.8: The fraction of messages that fail to be delivered as a function of the fraction of failed nodes.

Figure 3.8 shows the fraction of messages that fail to be delivered. We see that the network behaves well even with a large number of failed nodes. Even if we just terminate the search upon reaching a failed node, we get less than p fraction of failed searches with p fraction of failed nodes. In addition, backtracking gives a significant improvement in reducing the number of failures as compared to the other two methods: with 80% failed nodes, we still get less than 30% failed searches. These results are very promising and it would be interesting to study backtracking analytically.

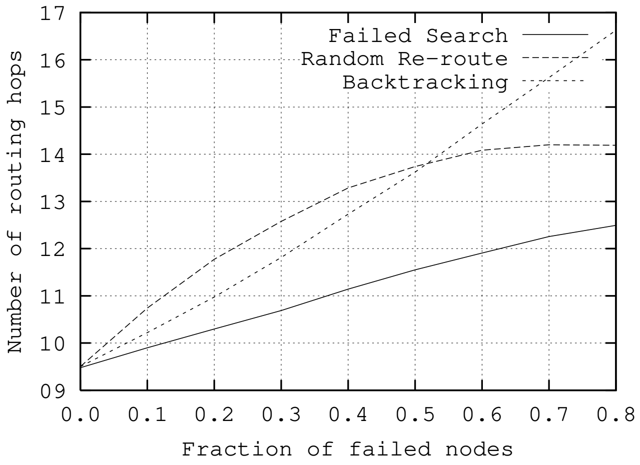


Figure 3.9: The average number of routing hops for successful searches as a function of the fraction of failed nodes.

Figure 3.9 shows the number of routing hops for successful searches versus the fraction of failed nodes. Here too, the network does well with failures; the average number of routing hops is small except with backtracking. We see that in the case of random rerouting, the average number of routing hops does not increase too much as the probability of node failure increases. This happens because quite a few of the searches fail, so the ones that succeed (with a few routing hops) lead to a small average number of routing hops.

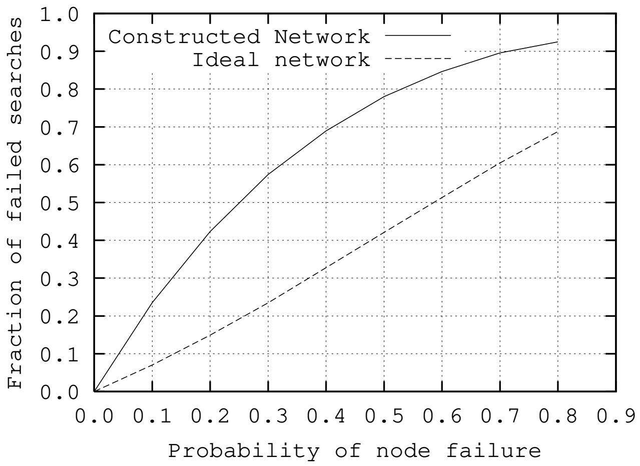


Figure 3.10: A comparison of the fraction of failed searches in the ideal network and the network constructed according to the heuristic given in Section 3.6.

We also compared the search performance of the ideal network and that of the network constructed using the heuristics given in Section 3.6. We ran 10 iterations of constructing a network of 2^{17} nodes, both ideally as well as according to the heuristic, and delivered 10000 messages between randomly chosen nodes. The search was terminated if it reached a failed node, similar to the first strategy in the earlier experiments. Figure 3.10 shows the number of failed searches as the probability of node failure increases. We see that the network constructed using the heuristic does not perform as well as the ideal network even though the magnitude of error between the two distributions is fairly small (Figures 3.6 and 3.7). It appears that the small skew in the link distribution affects the search performance greatly, and it would be interesting to see how this can be further improved.

3.8 Conclusions and future work

Model	Number of Links ℓ	Upper Bound	Lower Bound
No failures	1	$O(\log^2 n)$	$\Omega(\frac{\log^2 n}{\log\log n})$
	$[1, \log n]$	$O(\frac{\log^2 n}{\ell})$	$\Omega(\frac{\log^2 n}{\ell \log\log n})$ [One-sided] $\Omega(\frac{\log^2 n}{\ell^2 \log\log n})$ [Two-sided]
	$(\log n, n^c]$	$O(\frac{\log n}{\log b})$	$\Omega(\frac{\log n}{\log \ell})$
Link present with probability p	$[1, \log n]$	$O(\frac{\log^2 n}{p\ell})$	-
	$(\log n, n^c]$	$O(\frac{b \log n}{p})$	-
Node present with probability p	$[1, \log n]$	$O(\frac{\log^2 n}{p\ell})$	-

Table 3.1: Summary of upper and lower bounds for greedy routing.

Table 3.1 summarizes our upper and lower bounds. We note that with our deterministic strategy with no failures, the number of long-distance links $\ell = O(b \log_b n)$.

We have shown that greedy routing in an overlay network organized as a random graph in a metric space can be a nearly optimal mechanism for searching in a peer-to-peer system, even in the presence of many faults. Our lower bound also shows that our linking strategy i.e., using the inverse power-law distribution with exponent 1, is close to optimal.

We see this as an important first step in the design of efficient algorithms for such networks, but many issues still need to be addressed. Our results mostly apply to one-dimensional metric spaces like the line or the circle. One interesting possibility is whether similar strategies would work for higher-dimensional spaces. We could then use some of the dimensions to represent the actual physical distribution of the machines in real space, which would allow resources requests to be serviced from replicas that are closest in physical space to the requesting machine. Good network-building and search mechanisms for this model might allow efficient location of nearby instances of a resource without having to resort to local flooding (as in [63]). Our experimental results also show that using smart search techniques such as backtracking greatly improve the performance of searches in the presence of failure. It would be interesting to analyze this strategy formally and design other strategies that can reduce the number of failed searches. Another promising direction would be to study the security properties of greedy routing schemes to see how they can be adapted to provide desirable properties like anonymity or robustness against byzantine failures.

Chapter 4

Skip Graphs

In this chapter, we explore some of the limitations of distributed hash tables and propose a new data structure called a skip graph for implementing a peer-to-peer system.

4.1 Drawbacks of DHTs

As explained in Chapter 2, recent peer-to-peer systems like CAN [95], Chord [112], Pastry [99], Tapestry [125], and Viceroy [78] use a distributed hash table approach for a peer-to-peer system. The main operation in these networks is to retrieve the identity of the node which stores the resource, from any other node in the network. To this end, there is an overlay graph in the form of a metric space in which the nodes and resources are embedded. The locations of the nodes and resources in the overlay network are determined by the hash values of their identifiers and keys respectively. In a network like Chord, the purpose of the hash function is two-fold:

1. Hashing maps resources to nodes uniformly so that no node gets much more than its fair share of resources to manage. In Chord with m machines and n resources, using a base hash function such as SHA-1 [1] guarantees with high probability that each node is responsible for at most $(1 + \log m) \cdot n/m$ keys. Thus, hashing provides a natural load balancing among the nodes.

2. In addition, hashing the node identifiers (for example, their IP addresses) distributes

the nodes uniformly in the overlay metric space. As a result of that, it does not need any additional mechanism to balance the tree that is maintained for routing in the network.

However, hashing results in scrambling the resource keys and in turn destroys **spatial locality**. Spatial locality is the property where related resources are located close to each other in the data structure. For example, if the resources are web pages, then if the network maintained spatial locality, `http://www.cnn.com` and `http://www.cnn.com/weather` would be located close to each other in the data structure. If a search for any resource $i+1$ is immediately preceded by a search for resource i, the network would be able to use the information from the first search to improve the performance of the second search. This functionality helps applications such as prefetching of web pages and smart browsing. In a DHT, hashing will map these resources to random locations in the overlay network, and the two searches will be completely independent of each other.

Also, as hashing destroys the ordering on the keys, DHT systems do not easily support complex queries such as near matches to a key, keys within a certain range or approximate queries. Such systems cannot be easily used for applications such as versioning and user-level replication (explained in Section 4.6), without adding another layer of abstraction and its associated maintenance costs. In contrast, a data structure that does not destroy the key ordering can be used to provide all these features allowing for a simple underlying architecture. At the same time, we can continue to use the load balancing properties of the DHTs by using hashing, independent of the mechanism for resource location.

4.2 Our approach

We describe a new model for a peer-to-peer network based on a distributed data structure that we call a **skip graph**. Before we go on to explain skip graphs in detail in Section 4.3, we give an overview of the advantages of this data structure. Resource location and dynamic node addition and deletion can be done in time logarithmic in the size of the graph, and each node in a skip graph requires only logarithmic space to store information about its neighbors. More importantly, there is no hashing of the resource keys, so related resources are present near each other in a skip graph. This may be useful for certain applications such as prefetching of web pages, enhanced browsing, and efficient searching. Skip graphs also support **complex queries** such as range queries, i.e., locating resources whose keys lie within a certain specified range*. There has been interest in supporting complex queries in peer-to-peer-systems, and designing a system that supports range queries was posed as an open question [50]. Skip graphs are resilient to node failures: a skip graph tolerates removal of a large fraction of its nodes chosen at random without becoming disconnected, and even the loss of an $O(1/\log n)$ fraction of the nodes chosen by an adversary still leaves most of the nodes in the largest surviving component. Skip graphs can also be constructed without knowledge of the total number of nodes in advance. In contrast, DHT systems such as Pastry and Chord require *a priori* knowledge about the size of the system or its key space. Although these systems initially choose a very large keyspace which cannot be exhausted easily, for example 2^{128} identifiers, no requirement about the knowledge of the size of the keyspace is still an interesting property.

The rest of this chapter is organized as follows: we describe skip graphs in Sections 4.3. Some very recent related work is covered in Section 4.4. We give the details of the skip graph algorithms in Section 4.5. Some applications of skip graphs are given in Section 4.6. Sections 4.7 and 4.8 describe fault-tolerance properties and the repair mechanism for a skip graph. Congestion analysis and results are described in Section 4.9. Finally, we conclude in Section 4.10.

4.3 Skip graphs

A **skip list**, introduced by Pugh [93], is a randomized balanced tree data structure organized as a tower of increasingly sparse linked lists. Level 0 of a skip list is a linked list of all nodes in increasing order by key. For each i greater than 0, each node in level $i-1$ appears in level i independently with some fixed probability p. In a doubly-linked skip list, each node stores a predecessor pointer and a successor pointer for each list in which it appears, for an average of $\frac{2}{1-p}$ pointers per node. The lists at the higher level act as "express lanes"

*Skip graphs support complex queries along a single dimension i.e., for one attribute of the resource, for example, its name key.

that allow the sequence of nodes to be traversed quickly. Searching for a node with a particular key involves searching first in the highest level, and repeatedly dropping down a level whenever it becomes clear that the node is not in the current level. Considering the search path in reverse shows that no more than $\frac{1}{1-p}$ nodes are searched on average per level, giving an average search time of $O\left(\log n \frac{1}{(1-p)\log\frac{1}{p}}\right)$ with n nodes at level 0. Skip lists have been extensively studied [93, 91, 30, 65, 64], and because they require no global balancing operations are particularly useful in parallel systems [40, 41].

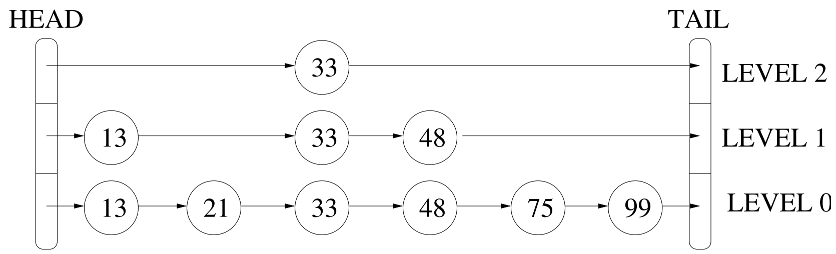


Figure 4.1: A skip list with $n = 6$ nodes and $\lceil \log n \rceil = 3$ levels.

We would like to use a data structure similar to a skip list to support typical binary tree operations on a sequence whose nodes are stored at separate locations in a highly distributed system subject to unpredictable failures. A skip list alone is not enough for our purposes, because it lacks redundancy and is thus vulnerable to both failures and congestion. Since only a few nodes appear in the highest-level list, each such node acts as a single point of failure whose removal partitions the list, and forms a hot spot that must process a constant fraction of all search operations. Skip lists also offer few guarantees that individual nodes are not separated from the rest even with occasional random failures. Since each node is connected on average to only $O(1)$ other nodes, even a constant probability of node failures will isolate a large fraction of the surviving nodes.

Our solution is to define a generalization of a skip list that we call a skip graph. As in a skip list, each of the n nodes in a skip graph is a member of multiple linked lists. The level 0 list consists of all nodes in sequence. Where a skip graph is distinguished from a skip list is that there may be many lists at level i, and every node participates in one of these lists, until the nodes are splintered into singletons after $O(\log n)$ levels on average. A skip graph supports search, insert, and delete operations analogous to the corresponding operations for skip lists; indeed, we show in Lemma 4.1 that algorithms for skip lists can be applied directly to skip graphs, as a skip graph is equivalent to a collection of n skip lists that happen to share some of their lower levels.

Because there are many lists at each level, the chances that any individual node participates in some search is small, eliminating both single points of failure and hot spots. Furthermore, each node has $\Theta(\log n)$ neighbors on average, and with high probability no node is isolated. In Section 4.7 we observe that skip graphs are resilient to node failures and have an expansion ratio of $\Omega(\frac{1}{\log n})$ with n nodes in the graph.

In addition to providing fault tolerance, having an $\Omega(\log n)$ degree to support $O(\log n)$ search time appears to be necessary for distributed data structures based on nodes in a one-dimensional space linked by random connections satisfying certain uniformity conditions [4]. While this lower bound requires some independence assumptions that are not satisfied by skip graphs, there is enough similarity between skip graphs and the class of models considered in the bound that an $\Omega(\log n)$ average degree is not surprising.

We now give a formal definition of a skip graph. Precisely which lists a node x belongs to is controlled by a **membership vector** $m(x)$. We think of $m(x)$ as an infinite random word over some fixed alphabet, although in practice, only an $O(\log n)$ length prefix of $m(x)$ needs to be generated on average. The idea of the membership vector is that every linked list in the skip graph is labeled by some finite word w, and a node x is in the list labeled by w if and only if w is a prefix of $m(x)$.

To reason about this structure formally, we will need some notation. Let Σ be a finite alphabet, let Σ^* be the set of all finite words consisting of characters in Σ, and let Σ^ω consist of all infinite words. We use subscripts to refer to individual characters of a word, starting with subscript 0; a word w is equal to $w_0 w_1 w_2 \ldots$. Let $|w|$ be the length of w, with $|w| = \infty$ if $w \in \Sigma^\omega$. If $|w| \geq i$, write $w \restriction i$ for the prefix of w of length i. Write ϵ for the empty word. If v and w are both words, write $v \preceq w$ if v is a prefix of w, i.e., if $w \restriction |v| = v$. Write w_i for the i-th character of the word w. Write $w_1 \wedge w_2$ for the common prefix (possibly empty) of the words w_1 and w_2.

Returning to skip graphs, the bottom level is always a doubly-linked list S_ϵ consisting of all the nodes in order as shown in Figure 4.2. In general, for each w in Σ^*, the doubly-linked list S_w contains all x for which w is a prefix of $m(x)$, in increasing order. We say that a particular list S_w is part of level i if $|w| = i$. This gives an infinite family of doubly-linked lists; in an actual implementation, only those S_w with at least two nodes are represented. A skip graph is precisely a family $\{S_w\}$ of doubly-linked lists generated in this fashion. Note that because the membership vectors are random variables, each S_w is also a random variable.

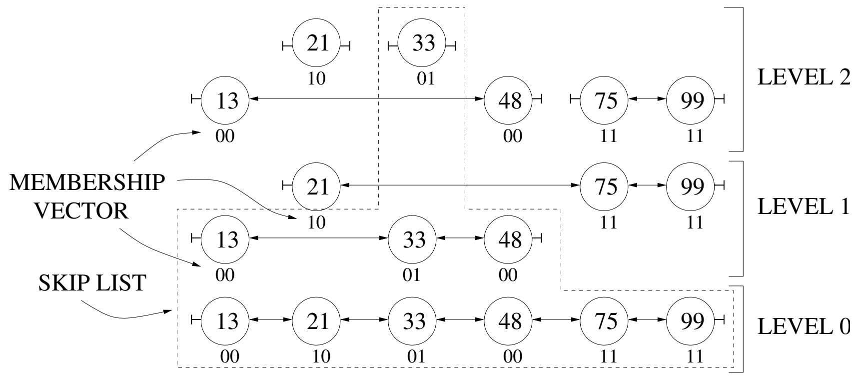


Figure 4.2: A skip graph with $n = 6$ nodes and $\lceil \log n \rceil = 3$ levels.

We can also think of a skip graph as a random graph, where there is an edge between x and y whenever x and y are adjacent in some S_w. Define x's left and right neighbors at level i as its immediate predecessor and successor, respectively, in $S_{m(x)\restriction i}$, or $\perp$ if no such nodes exist. We will write xL_i for x's left neighbor at level i and xR_i for x's right neighbor, and in general will think of the R_i as forming a family of associative composable operators to allow writing expressions like $xR_iR_{i-1}^2$ etc. We write x.maxLevel for the first level ℓ at which x is in a a singleton list, i.e., x has at least one neighbor at level $\ell - 1$.

An alternative view of a skip graph is a **trie** [29, 36, 68] of skip lists that share their lower levels. If we think of a skip list formally as a sequence of random variables $S_0, S_1, S_2, \ldots$, where the value of S_i is the level i list, then we have:

Lemma 4.1 *Let $\{S_w\}$ be a skip graph with alphabet Σ. For any $z \in \Sigma^\omega$, the sequence $S_0, S_1, S_2, \ldots$, where each $S_i = S_{z\restriction i}$, is a skip list with parameter $p = |\Sigma|^{-1}$.*

Proof: By induction on i. The list S_0 equals S_ϵ, which is just the base list of all nodes. A node x appears in S_i if $m(x) \restriction i = z \restriction i$; conditioned on this event occurring, the probability that x also appears in S_{i+1} is just the probability that $m(x)_{i+i} = z_{i+1}$. This event occurs with probability $p = |\Sigma|^{-1}$, and it is easy to see that it is independent of the corresponding event for any other x' in S_i. Thus each node in S_i appears in S_{i+1} with independent probability p, and $S_0, S_1, \ldots$ form a skip list. ∎

For a node x with membership vector $m(x)$, let the skip list $S_{m(x)}$ be called the **skip list restriction** of node x.

4.3.1 Implementation

In an actual implementation of a peer-to-peer system using a skip graph, each node in a skip graph will be a resource. The resources are sorted in increasing lexicographic order of their keys. Mapping these keys to actual physical machines can be done in two ways: In the first approach, we make every machine responsible for the resources that it hosts. Alternatively, we use a DHT approach where we hash node identifiers and resource keys to determine which nodes will be responsible for which keys. The first approach gives security and manageability whereas the second one gives good load balancing. For now, we treat nodes in the skip graph as representing resources, and present our results without committing to how these resources are distributed across machines. Each node in a skip graph stores the address and the key of its successor and predecessor at each of the $O(\log n)$ levels. In addition, each node also needs $O(\log n)$ bits of space for its membership vector.

In both of the above approaches, with n resources in the network, each machine is responsible for maintaining $O(\log n)$ links for *each* resource that it hosts, for a total of $O(n \log n)$ links in the entire network. This is a much higher storage requirement than the $O(m \log m)$ links for DHTs, where m is the number of machines in the system. Further, in our repair mechanism (described in Section 4.8), each machine will periodically check to see that its links are functional. This may result in a flood of messages given the high number of links per machine. It is an open question how to reduce the number of pointers in a skip graph and yet maintain the locality properties.

4.4 Related work

SkipNet is a system very similar to skip graphs that was independently developed by Harvey *et al.*[51]. SkipNet builds a trie of circular, singly-linked skip lists to link the *machines* in the system. The machines names are sorted using the domain in which they are located (for example `www.yale.edu`). In addition to the pointers between all the machines in all the domains that are structured like a skip graph, within each individual domain, the machines are also linked using a DHT, and the resources are uniformly distributed over all the machines using hashing. A search consists of two stages: First, the search locates the domain in which a resource lies by using a search operation similar to a skip graph. Second, once the search reaches some machine inside a particular domain, it uses greedy routing as in DHTs to locate the resource within that domain. SkipNet has been successfully implemented, and this shows that a skip-graph-like structure can be used to build real systems.

The SkipNet design ensures **path locality** i.e., the traffic within a domain traverses other nodes only within the same domain. Further, each domain gets to hosts it own data which provides **content locality** and inherent security. Finally, using the hybrid storage and search scheme provides **constrained load balancing** within a given domain. However, as the name of the data item includes the domain in which it is located, transparent remapping of resources to other domains is not possible, thus giving a very limited form of load balancing. Another drawback of this design is that it does not give full-fledged spatial locality. For example, if the resources are document files, sorting according to the domain on which they are served gives no advantage in searching for related files compared to DHTs.

Zhang *et al.*[124] and Awerbuch *et al.*[9] have both independently suggested designs for peer-to-peer systems using separate data structures for resources and the machines that store them. The main idea is to build a data structure D over the resources, which are distributed uniformly among all the machines using hashing, and to build a separate DHT over all the machines in the system. Each resource maintains the keys of its neighboring resources in D, and each machine maintains the addresses of its neighboring machines as per the DHT network. To access a neighbor b of resource a, a initiates a DHT search for

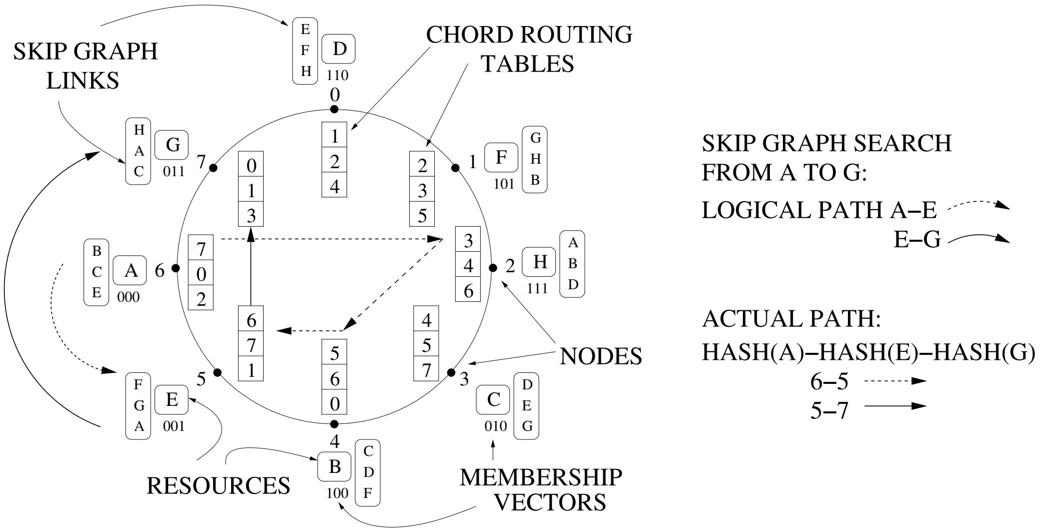


Figure 4.3: An example of the hybrid system design proposed by Awerbuch *et al.*[9]. Here, we use Chord for mapping nodes and a skip graph for mapping resources. This is a simplified version with only one resource per node.

the hash value of b. One pointer access in D is converted to a search operation in the DHT, so if any operation in D takes time t, the same operation takes $O(t \log m)$ time with m machines in this hybrid system. Zhang *et al.*[124] focus on implementing a tree of the resources, in which each node in the tree is responsible for some fixed range of the keyspace that its parent is responsible for. Awerbuch *et al.*[9] propose building a skip graph of the resources on top of the machines in the DHT. A simple example of the latter scheme is shown in Figure 4.3.

This design approach is interesting because it allows building any data structure using the resources, while providing uniform load balancing. In particular, both these systems support complex queries as in skip graphs, and uniform load balancing as in DHTs. We believe that distributing the resources uniformly among all the nodes (as described in Section 4.3.1) will also have the same properties as these two approaches.

However, the Awerbuch *et al.* approach and our uniform resource distribution approach suffer from the same problems of high storage requirements and high volume of repair mechanism message traffic as a skip graph. With m machines and n resources in the system, in the Awerbuch *et al.* approach, each machine has to store $O(\log m)$ pointers (for the DHT links) and $O(\log n)$ keys for *each* resource that it hosts (for the skip graph pointers). Further, the repair mechanism has to repair the broken DHT links as well as inconsistent skip graph keys. Finally, the search performance is degraded to $O(\log^2 m)$ compared to $O(\log m)$ in DHTs and $O(\log n)$ in skip graphs. In comparison, our approach of uniform resource distribution does slightly better as each machine stores $O(\log n)$ pointers for each resource that it hosts, repair is required only for the skip graph links, and the search time is $O(\log n)$ like in skip graphs.

In the Zhang *et al.* approach, each machine has to store k keys for the k children of each tree node that it hosts, and $O(\log m)$ pointers for the DHT links. Repair involves fixing broken tree keys as well as broken DHT links. This scheme suffers from the other problems of tree data structures such as increased traffic on the nodes higher up in the tree, and vulnerability to failures of these nodes. Further, unlike skip graphs, it require a priori knowledge about the keyspace in order to assign specific ranges to the tree nodes. It is still an open problem to design a system that efficiently supports both uniform load balancing and complex queries.

4.5 Algorithms for a skip graph

In this section, we describe the search, insert, and delete operations for a skip graph. For simplicity, we refer to the key of a node (e.g. x.key) with the same notation (e.g. x) as the node itself. It will be clear from the context whether we refer to a node or its key. In the algorithms, we denote the pointer to x's successor and predecessor at level ℓ as x.neighbor$[R][\ell]$ and x.neighbor$[L][\ell]$ respectively. We define xR_ℓ formally to be the value of x.neighbor$[R][\ell]$, if x.neighbor$[R][\ell]$ is a non-nil pointer to a non-faulty node, and $\perp$ otherwise. We define xL_ℓ similarly. We summarize the variables stored at each node in Table 4.1.

In this section, we only give the algorithms and analyze their performance; we defer the proofs of the correctness of the algorithms to Section 4.5.4.

Variable	Type
key	Resource key
neighbor[R]	Array of successor pointers
neighbor[L]	Array of predecessor pointers
m	Membership vector
maxLevel	Integer
deleteFlag	Boolean

Table 4.1: List of all the variables stored at each node.

4.5.1 The search operation

The search operation (Algorithm 1) is identical to the search in a skip list with only minor adaptations to run in a distributed system. The search is started at the topmost level of the node seeking a key and it proceeds along each level without overshooting the key, continuing at a lower level if required, until it reaches level 0. Either the address of the node storing the search key, if it exists, or the address of the node storing the largest key less than the search key is returned.

Algorithm 1: search for node v

upon receiving $\langle$`searchOp`, startNode, searchKey, level$\rangle$:
if *(v.key = searchKey)* **then**
 send $\langle$`foundOp`, $v\rangle$ to startNode
if *(v.key < searchKey)* **then**
 while *level* ≥ 0 **do**
  if *((v.neighbor[R][level].key < searchKey)* **then**
   send $\langle$`searchOp`, startNode, searchKey, level$\rangle$ to v.neighbor[R][level]
   break
  else level←level-1
else
 while *level* ≥ 0 **do**
  if *((v.neighbor[L][level]).key > searchKey)* **then**
   send $\langle$`searchOp`, startNode, searchKey, level$\rangle$ to v.neighbor[L][level]
   break
  else level←level-1
if *(level < 0)* **then**
 send $\langle$`notFoundOp`, $v\rangle$ to startNode

Lemma 4.2 *The search operation in a skip graph S with n nodes takes expected $O(\log n)$ messages and $O(\log n)$ time.*

Proof: Let Σ be the alphabet for the membership vectors of the nodes in the skip graph S, and z be the node at which the search starts. By Lemma 4.1, the sequence $S_{m(z)} = S_0, S_1, S_2, \ldots$, where each $S_i = S_{z\restriction i}$, is a skip list. A search that starts at z in the skip graph will follow the same path in S as in $S_{m(z)}$. So we can directly apply the skip list search analysis given in [93], to analyze the search in S. With n nodes, on an average there will be $O(\log n \frac{1}{\log(1/p)})$ levels, for $p = |\Sigma|^{-1}$. At most $\frac{1}{1-p}$ nodes are searched on average at each level, for a total of $O(\log n \frac{1}{(1-p)\log(1/p)})$ expected messages and $O(\log n \frac{1}{(1-p)\log(1/p)})$ expected time. Thus, with fixed p, the search operation takes expected $O(\log n)$ messages and $O(\log n)$ time. ∎

The network performance depends on the value of $p = |\Sigma|^{-1}$. As p increases, the search time decreases, but the number of levels increase, so each node has to maintain neighbors at more levels. Thus we get a trade-off between the search time and the storage requirements at each node.

The performance shown in Lemma 4.2 is comparable to the performance of distributed hash tables, for example, Chord [112]. With n resources in the system, a skip graph takes $O(\log n)$ time for one search operation. In comparison, Chord takes $O(\log m)$ time, where m is the number of machines in the system. As long as n is polynomial in m, we get the same asymptotic performance from both DHTs and skip graphs for search operations.

Skip graphs can support **range queries** in which one is asked to find a key $\geq x$, a key $\leq x$, the largest key $< x$, the least key $> x$, some key in the interval $[x, y]$, all keys in $[x, y]$, and so forth. For most of these queries, the procedure is an obvious modification of Algorithm 1 and runs in $O(\log n)$ time with $O(\log n)$ messages. For finding all nodes in an interval, we can use a modified Algorithm 1 to find a single element of the interval (which takes $O(\log n)$ time and $O(\log n)$ messages). With r nodes in the interval, we can then broadcast the query through all the nodes (which takes $O(\log r)$ time and $O(r \log n)$ messages). If the originator of the query is capable of processing r simultaneous responses, the entire operation still takes $O(\log n)$ time.

4.5.2 The insert operation

A new node u knows some *introducing* node v in the network that will help it to join the network. Node u inserts itself in one linked list at each level till it finds itself in a singleton list at the topmost level. The insert operation consists of two stages:

1. Node u starts a search for itself from v to find its neighbors at level 0, and links to them.
2. Node u finds the closest nodes s and y at each level $\ell \geq 0$, $s < u < y$, such that $m(u) \restriction (\ell+1) = m(s) \restriction (\ell+1) = m(y) \restriction (\ell+1)$, if they exist, and links to them at level $\ell + 1$.

Because each existing node v does not require $m(v)_{\ell+1}$ unless there exists another node u such that $m(v) \restriction (\ell+1) = m(u) \restriction (\ell+1)$, it can delay determining its value until a new node arrives asking for its value; thus at any given time only a finite prefix of the membership vector of any node needs to be generated. Detailed pseudocode for the insert operation is given in Algorithm 2. Figure 4.4 shows a typical execution of an insert operation in a small skip graph with $\Sigma = \{0, 1\}$, where node $u = 36$ is inserted starting from node $v = 13$.

Lemma 4.3 *The insert operation in a skip graph S with n nodes takes expected $O(\log n)$ messages and $O(\log n)$ time.*

Proof: Let Σ be the alphabet for the membership vectors of the nodes in the skip graph S. With n nodes, there will be average of $O(\log n \frac{1}{\log(1/p)})$ levels in the skip graph, $p = |\Sigma|^{-1}$. To link at level 0, a new node u performs one search operation. From Lemma 4.2, this takes $O(\log n \frac{1}{(1-p)\log(1/p)})$ expected messages and $O(\log n \frac{1}{(1-p)\log(1/p)})$ expected time. At each level ℓ, $\ell \geq 0$, u communicates with an average of $2/p$ nodes, before it finds at most two nodes s and y, with $m(s)_\ell = m(u)_\ell = m(y)_\ell$, $s < u < y$, and connects to them at level $\ell + 1$. The expected number of messages and time for the insert operation at all levels is $O\left(\frac{\log n}{\log(1/p)}\left(\frac{1}{1-p} + \frac{2}{p}\right)\right)$. Thus with fixed p, the insert operation takes expected $O(\log n)$ messages and $O(\log n)$ time. ∎

With m machines and n resources in the system, most DHTs such as CAN, Pastry, and Tapestry take $O(\log m)$ time for insertion; an exception is Chord which takes $O(\log^2 m)$

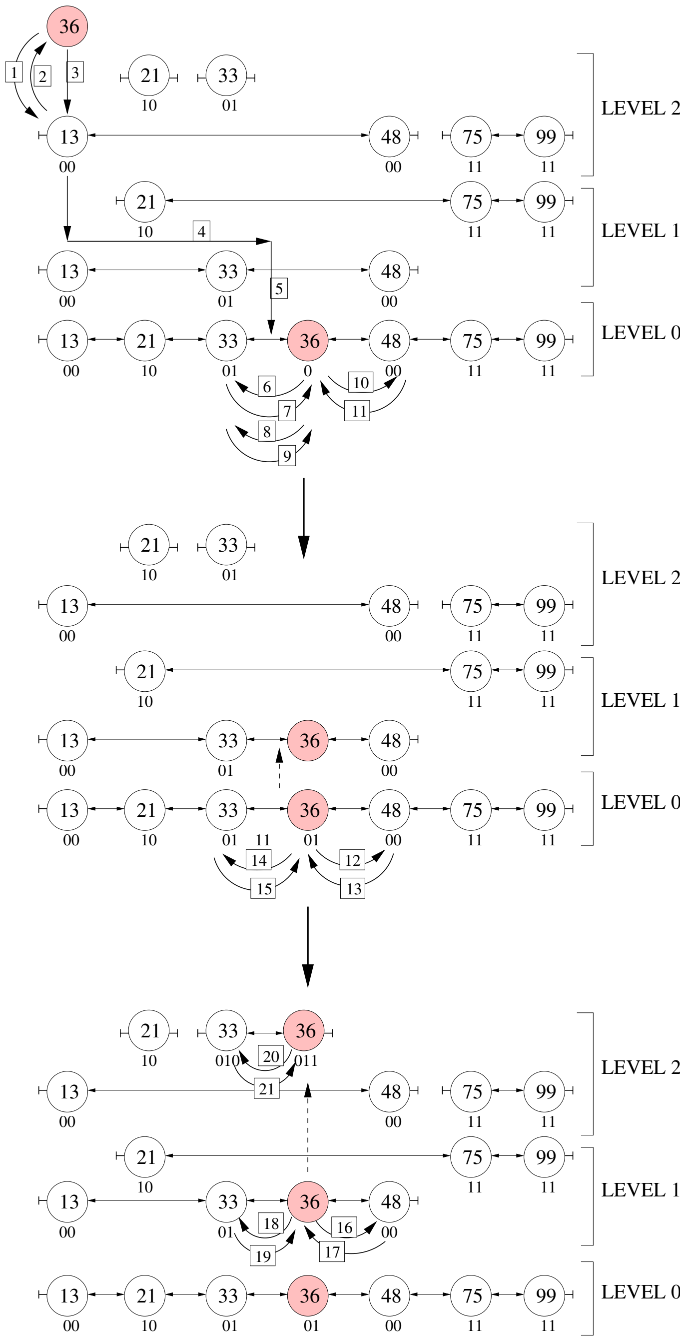


Figure 4.4: Inserting node 36 in a skip graph with $\Sigma = \{0, 1\}$, starting from node 13. Messages are labeled by numbers in boxes in the order in which they are sent. Messages 1–3 implement node 36 determining the maximum level of node 13, and starting the search operation to find its neighbor at level 0. Messages 4–5 implement the search operation, and node 33 informing node 36 that it is node 36's closest neighbor at level 0. Messages 6–11 implement node 36 inserting itself between nodes 33 and 48 at level 0. Messages 12–15 implement node 36 determining its neighbors at level 1, and inserting itself between nodes 33 and 48 at level 1. Messages 16–19 implement node 36 determining its neighbors at level 2, and linking to node 33 at level 2. Messages 20–21 implement node 36 determining its neighbors at level 3, finding that no neighbors exist, and completing its insert operation.

Algorithm 2: insert for new node u

```
if (introducer = u) then
    u.neighbor[L][0] ← ⊥
    u.neighbor[R][0] ← ⊥
    u.maxLevel ← 0
else
    if (introducer.key < u.key) then
        side ← R
        otherSide ← L
    else
        side ← L
        otherSide ← R
    send ⟨getMaxLevelOp⟩ to introducer
    wait until receipt of ⟨retMaxLevelOp, maxLevel⟩
    send ⟨searchOp, u, u.key, maxLevel-1⟩ to introducer
    wait until foundOp or notFoundOp is received
    upon receiving ⟨foundOp, clone⟩:
        terminate insert
    upon receiving ⟨notFoundOp, otherSideNeighbor⟩:
    send ⟨getNeighborOp, side, 0⟩ to otherSideNeighbor
    wait until receipt of ⟨retNeighborOp, sideNeighbor, 0⟩:
    send ⟨getLinkOp, u, side, 0⟩ to otherSideNeighbor
    wait until receipt of ⟨setLinkOp, newNeighbor, 0⟩:
    u.neighbor[otherSide][0] ← newNeighbor
    send ⟨getLinkOp, u, otherSide, 0⟩ to sideNeighbor
    wait until receipt of ⟨setLinkOp, newNeighbor, 0⟩:
    u.neighbor[side][0] ← newNeighbor
    ℓ ← 0
    while true do
        m(u)_ℓ ← uniformly chosen random element of Σ
        ℓ ← ℓ + 1
        if (u.neighbor[R][ℓ − 1] ≠ ⊥) then
            send ⟨buddyOp, u, ℓ − 1, m(u)_{ℓ−1}, L⟩ to u.neighbor[R][ℓ − 1]
            wait until receipt of ⟨setLinkOp, neighbor, ℓ⟩:
            u.neighbor[R][ℓ] ← neighbor
        else u.neighbor[R][ℓ] = ⊥
        if (u.neighbor[L][ℓ − 1] ≠ ⊥) then
            send ⟨buddyOp, u, ℓ − 1, m(u)_{ℓ−1}, R⟩ to u.neighbor[L][ℓ − 1]
            wait until receipt of ⟨setLinkOp, neighbor, ℓ⟩:
            u.neighbor[L][ℓ] ← neighbor
        else u.neighbor[L][ℓ] = ⊥
        if ((u.neighbor[R][ℓ] = ⊥) and (u.neighbor[L][ℓ] = ⊥)) then
            break
    u.maxLevel ← ℓ
```

Algorithm 3: Node v's message handler for messages received during the insert of new node u.

upon receiving $\langle$`getLinkOp`, u, side, $\ell\rangle$:
change_neighbor(u, side, ℓ)

upon receiving $\langle$`buddyOp`, u, ℓ, val, side$\rangle$:
if *(side = L)* **then** otherSide $\leftarrow R$
else otherSide $\leftarrow L$
if $(m(v)_\ell = \perp)$ **then**
  $m(v)_\ell \leftarrow$ uniformly chosen random element of Σ
  v.neighbor[L][ℓ] $\leftarrow \perp$
  v.neighbor[R][ℓ] $\leftarrow \perp$
if $(m(v)_\ell = val)$ **then**
  change_neighbor(u, side, $\ell + 1$)
else
  if *(v.neighbor[otherSide][ℓ] $\neq \perp$)* **then**
    send $\langle$`buddyOp`, u, val, ℓ, side$\rangle$ to v.neighbor[otherSide][ℓ]
  else
    send $\langle$`setLinkOp`, $\perp$, $\ell\rangle$ to u

Algorithm 4: change_neighbor(u, side, ℓ) for node v

if *(side = R)* **then** cmp $\leftarrow$ <
else cmp $\leftarrow$ >
if *((v.neighbor[side][ℓ]).key cmp u.key)* **then**
  send $\langle$`getLinkOp`, u, side, $\ell\rangle$ to v.neighbor[side][ℓ]
else
  send $\langle$`setLinkOp`, v, $\ell\rangle$ to u
v.neighbor[side][ℓ] $\leftarrow u$

Algorithm 5: Additional messages for node v

upon receiving $\langle$`updateOp`, side, newNeighbor, $\ell\rangle$:
v.neighbor[side][ℓ] $\leftarrow$ newNeighbor

upon receiving $\langle$`getMaxLevelOp`$\rangle$ from u:
send $\langle$`retMaxLevelOp`, $v.maxLevel\rangle$ to u

upon receiving $\langle$`getNeighborOp`, side, $\ell\rangle$ from u:
send $\langle$`retNeighborOp`, $v$side$_\ell\rangle$ to u

time. An $O(\log m)$ time bound improves on the $O(\log n)$ bound for skip graphs when m is much smaller than n. However, the cost of this improvement is losing support for complex queries and spatial locality, and the improvement itself is only a constant factor unless some machines store a superpolynomial number of resources.

Inserts can be trickier when we have to deal with concurrent node joins. Before u links to any neighbor, it verifies that its join will not violate the order of the nodes. So if any new nodes have joined the skip graph between u and its predetermined successor, u will advance over the new nodes if required before linking in the correct location.

4.5.3 The delete operation

The delete operation is very simple. When node u wants to leave the network, it informs its predecessor node at each level to update its successor pointer to point to u's successor. It starts at the topmost level and works its way down to level 0. Node u also informs its successor node at each level to update its predecessor pointer to point to n's predecessor. If u's successor or predecessor are being deleted as well, they pass the message on to their neighbors so that the nodes are correctly linked up. A node does not delete itself from the graph as long as it is waiting for some message as a part of the delete operation of another node.

Algorithm 6: delete for existing node u

u.deleteFlag = true
for $\ell \leftarrow u.max\_levels$ ***downto*** 0 **do**
    if $u.neighbor[R][\ell] \neq \perp$ **then**
        send ⟨`deleteOp`, ℓ, sender⟩ to u.neighbor$[R][\ell]$
        wait until receipt of ⟨`confirmDeleteOp`, ℓ⟩ or ⟨`noNeighborOp`, ℓ⟩:
        upon receiving ⟨`noNeighborOp`, ℓ⟩:
        if $u.neighbor[L][\ell] \neq \perp$ **then**
            send ⟨`setNeighborNilOp`, ℓ, sender⟩ to u.neighbor$[L][\ell]$
            wait until receipt of ⟨`confirmDeleteOp`, ℓ⟩

Algorithm 7: Node v's message handler for messages received during the delete operation.

```
upon receiving ⟨deleteOp, ℓ, sender⟩:
if (v.deleteFlag = true) then
    if (v.neighbor[R][ℓ] ≠ ⊥) then
        send ⟨deleteOp, ℓ, sender⟩ to v.neighbor[R][ℓ]
    else
        send ⟨noNeighborOp, ℓ⟩ to sender
else
    send ⟨findNeighborOp, ℓ, sender⟩ to v.neighbor[L][ℓ]
    wait until receipt of ⟨foundNeighborOp, x, ℓ⟩:
    v.neighbor[L][ℓ] ← x
    send ⟨confirmDeleteOp, ℓ⟩ to sender

upon receiving ⟨findNeighborOp, ℓ, sender⟩:
if (v.deleteFlag = true) then
    if (v.neighbor[L][ℓ] ≠ ⊥) then
        send ⟨findNeighborOp, ℓ, sender⟩ to v.neighbor[L][ℓ]
    else
        send ⟨foundNeighborOp, ⊥, ℓ⟩ to sender
else
    send ⟨foundNeighborOp, v, ℓ⟩ to sender
    v.neighbor[R][ℓ] ← sender

upon receiving ⟨setNeighborNilOp, ℓ, sender⟩:
if (v.deleteFlag = true) then
    if (v.neighbor[L][ℓ] ≠ ⊥) then
        send ⟨setNeighborNilOp, ℓ, sender⟩ to v.neighbor[L][ℓ]
    else
        send ⟨confirmDeleteOp, ℓ⟩ to sender
else
    send ⟨confirmDeleteOp, ℓ⟩ to sender
    v.neighbor[R][ℓ] ← ⊥
```

Lemma 4.4 *The delete operation in a skip graph S with n nodes takes expected $O(\log n)$ messages and $O(1)$ time.*

Proof: Let Σ be the alphabet for the membership vectors of the nodes in the skip graph S. With n nodes, there will be average of $O(\log n \frac{1}{\log(1/p)})$ levels in the skip graph, $p = |\Sigma|^{-1}$. At each level ℓ, $\ell \geq 0$, the node to be deleted communicates with at most two other nodes. It takes an average of $O(\log n \frac{1}{\log(1/p)})$ total messages and $O(1)$ time as the messages at all the levels can be sent in parallel. Thus with fixed p, a delete operation takes $O(\log n)$ messages and $O(1)$ time. ∎

During the delete operation of node u, if u's successor or predecessor at some level are also being deleted, then the number of message at that level is proportional to the number of consecutive nodes being deleted.

4.5.4 Correctness of algorithms

In this section, we prove the correctness of the insert and delete algorithms given in Section 4.5. The definition of a skip graph in Section 4.3 involves global properties of the data structure (such as S_{w1} being a subset of S_w) that are difficult to work with in the correctness proofs. So we start by defining a set of *local* constraints which characterize a skip graph. We first prove that a data structure is a skip graph if and if only if all these constraints are satisfied, and then we prove that these constraints are not violated after an insert and delete operation, thus maintaining the skip graph properties. Further, these constraints will be used for our repair mechanism as we can monitor the state of the graph by checking these constraints locally at each node, and detecting and repairing node failures.

As explained in Section 4.5, we use $\perp$ both to refer to the null pointers at the ends of the doubly-linked lists of the skip graph, and to refer to pointers to failed nodes.

Let x be any node in the skip graph; then for all levels $\ell \geq 0$:

1. If $xR_\ell \neq \perp$, $xR_\ell > x$.

2. If $xL_\ell \neq \perp$, $xL_\ell < x$.

3. If $xR_\ell \neq \perp$, $xR_\ell L_\ell = x$.

4. If $xL_\ell \neq \perp$, $xL_\ell R_\ell = x$.

5. If $m(x) \restriction (\ell+1) = m(xR_\ell^k) \restriction (\ell+1)$ and $\nexists j, j < k, m(x) \restriction (\ell+1) = m(xR_\ell^j) \restriction (\ell+1)$, then $xR_{\ell+1} = xR_\ell^k$. Else, $xR_{\ell+1} = \perp$.

6. If $m(x) \restriction (\ell+1) = m(xL_\ell^k) \restriction (\ell+1)$ and $\nexists j, j < k, m(x) \restriction (\ell+1) = m(xL_\ell^j) \restriction (\ell+1)$, then $xL_{\ell+1} = xL_\ell^k$. Else, $xL_{\ell+1} = \perp$.

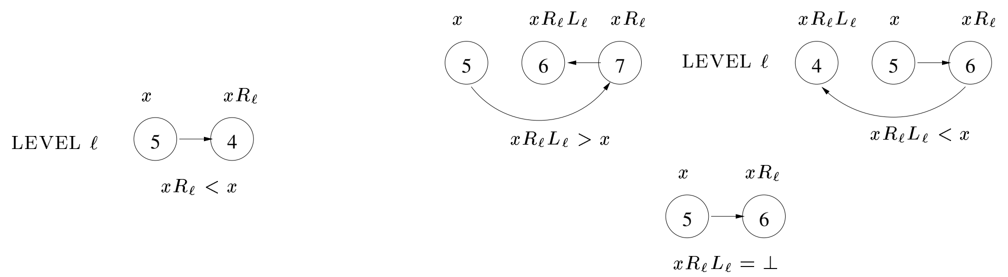
As per the definition of a skip graph given in Section 4.3, all the elements in a doubly linked list S_w (which contains all x for which w is a prefix of $m(x)$ of length ℓ) are in increasing order. Constraints 1 through 4 imply that all non-edge nodes satisfy the increasing order in the linked lists. Constraints 1 and 2 ensure that the order is locally true at every node, whereas Constraints 3 and 4 ensure that the entire list is doubly linked correctly. The constraints that the increasing order is satisfied locally at each node and that the list is doubly-linked correctly, put together ensure that no element is skipped over and that the entire list is sorted.

Constraints 5 and 6 denote how the lists at different levels are related to each other. The successor (predecessor) of node x at level $\ell+1$ is always the first node to its right (left) at level ℓ whose membership vector matches the membership vector of x in one additional position. Node x is connected at level $\ell+1$, on the right side to a node z such that $x, z \in S_w$, and z is the nearest node greater than x in S_w with $m(x)_\ell = m(z)_\ell$. Similarly, x is connected at level $\ell+1$, on the left side to a node u such that $u, x \in S_w$, and u is the nearest node less than x in S_w with $m(u)_\ell = m(x)_\ell$.

Define a *defective* skip graph as a data structure that that contains skip graph elements but does not satisfy the definition of a skip graph; for example, it may contain out-of-order elements, missing links, or worse.

Theorem 4.5 *Every connected component of the data structure is a skip graph if and only if Constraints* $1-6$ *are satisfied.*

Proof: We start with the reverse direction: if the constraints are not satisfied, then some connected component of the data structure is not a skip graph. As Constraints 2, 4 and 6 are mirror images of Constraints 1, 3 and 5 respectively, we will only consider violations of Constraints 1, 3 and 5.



(a) Violation of Constraint 1: $xR_\ell > x$.

(b) Violation of Constraint 3: $xR_\ell L_\ell = x$.

Figure 4.5: Violation of Constraints 1 and 3.

Figure 4.5 shows how Constraints 1 and 3 can be violated. Each violation leads to either an *unsorted* or *inconsistently* linked list at level ℓ, so the data structure is not a skip graph. There are two ways in which Constraint 5 can be violated.

1. For some x and ℓ, $xR_{\ell+1} = xR_\ell^k$ but $\exists j, j < k, m(x) \restriction (\ell+1) = m(xR_\ell^j) \restriction (\ell+1)$.
   Let $y = xR_\ell^j$ and $z = xR_\ell^k$. As the linked list is sorted at level ℓ, $j < k \Rightarrow y < z$, and since $y = xR_\ell^j, x < y$. Let $S_w = \{x | m(x) \restriction (\ell+1) = w\}$. Then in a skip graph, x, y and $z \in S_w$. Since $y \neq xR_{\ell+1}$, either $y \notin S_w$ or the linked list at level $\ell+1$ is not sorted as $x < y < z$. In both cases, the resulting data structure is a defective skip graph.

2. For some x and ℓ, $xR_{\ell+1} \neq xR_\ell^k$.
   As $m(x) \restriction (\ell+1) = m(xR_{\ell+1}) \restriction (\ell+1)$, $m(x) \restriction \ell = m(xR_{\ell+1}) \restriction \ell$. It follows that both x and $xR_{\ell+1}$ are in $S_{m(x)\restriction\ell}$. But then if $xR_{\ell+1} \neq xR_\ell^k$ for any k, some edge in $S_{m(x)\restriction\ell}$ is missing, and the data structure is a defective skip graph.

Now we prove that every connected component of a data structure is a skip graph if all the constraints are satisfied. We first prove that we have sorted, doubly-linked lists at all levels using Constraints 1 through 4, and then prove that each list contains the correct elements as per their membership vectors using Constraints 5 and 6.

Let x be an arbitrary element of the data structure. Let $S_{x,\ell}$ be the maximal sequence of the form $xL_\ell^j, \ldots, x, \ldots, xR_\ell^k$, where $j, k \geq 0$, such that no element of the sequence is $\perp$. We show that this sequence is sorted and doubly-linked using induction. According to Constraint 1, $xR_\ell > x$, and according to Constraint 3, $xR_\ell L_\ell = x$. Thus the sequence x, xR_ℓ is sorted and doubly-linked. Similarly, according to Constraint 2, $xL_\ell < x$, and according to Constraint 4, $xL_\ell R_\ell = x$. Thus the sequence xL_ℓ, x is sorted and doubly-linked. Let the sequence $xL_\ell^{j-1}, \ldots, x, \ldots, xR_\ell^{k-1}$, be a sorted, doubly-linked list. According to Constraints 1 and 3, $xR_\ell^k > xR_\ell^{k-1}$, and $xR_\ell^{k-1}R_\ell L_\ell = xR_\ell^{k-1}$. Similarly, according to constraints 2 and 4, $xL_\ell^j < xL_\ell^{j-1}$, and $xL_\ell^{j-1}R_\ell L_\ell = xL_\ell^{j-1}$. Thus the maximal sequence $S_{x,\ell} = xL_\ell^j, \ldots, x, \ldots, xR_\ell^k$ is sorted and doubly-linked.

We now show that if two nodes are connected at level ℓ, they are also connected at level 0. Suppose that Constraints 5 and 6 hold. Then, we prove that for each level $\ell \geq 0$, $xR_\ell = xR_0^j$ and $xL_\ell = xL_0^j$. Clearly this is true for $\ell = 0$ and $j = 1$. Suppose that it is true for level $\ell - 1$. Let $x = y_0$ and let each $xR_{\ell-1}^i = y_i$, $1 \leq i \leq k$. For each i, $y_i = y_{i-1}R_{\ell-1} = y_{i-1}R_0^{j_i}$. So $y_1 = y_0 R_{\ell-1} = y_0 R_0^{j_0}$, $y_2 = y_1 R_{\ell-1} = y_1 R_0^{j_1} = y_0 R_0^{j_0+j_1}$, and so on. Thus, $y_k = y_0 R_0^{j_0+j_1+j_2+\ldots+j_k} = y_0 R_0^j$, where $j = j_0 + j_1 + \ldots + j_k$. But $y_k = xR_{\ell-1}^k$ and $y_0 = x$. So $xR_{\ell-1} = xR_0^j$. According to Constraint 5, $xR_\ell = xR_{\ell-1}^k$. Thus we get $xR_\ell = xR_0^j$. A similar proof will show that $xL_\ell = x_0 L_0^j$.

We use the proof above to show that any two connected nodes are connected in the same list at level 0. Consider a path $xE_1E_2 \ldots E_k y$ where each E_i is either L_{ℓ_i} or R_{ℓ_j}. As proved above, there exists a path $xE_1'^{j_1}E_2'^{j_2} \ldots E_k'^{j_k} y$ where each E_i' is either L_0 or R_0. Thus it follows that x and y are in the same list at level 0. Also $S_{x,0} = S_{y,0}$ if x and y are in the same connected component. So we get a single list S_ϵ at level 0, which consists of all the elements in the same connected component of the data structures.

As proved above, S_ϵ is also sorted and doubly-linked. With the single list S_ϵ at level 0, according to Constraints 5 and 6, each node $x \in S_\epsilon$, is linked to its right and left at level 1 to the nearest nodes z and u respectively (if they exist), such that $m(x) \restriction 1 = m(z) \restriction 1 = m(u) \restriction 1$, and $u < x < z$. Thus, we get $|\Sigma|$ linked lists at level 1, $S_a = \{y | m(y)_0 = a\}$, one for each $a \in \Sigma$. In general, at level ℓ, we can get up to $|\Sigma|^\ell$ lists, one for each $w \in \Sigma^\ell$. Each list contains all the nodes which have the matching membership vector prefix. As proved above, each of these lists is also sorted and doubly-linked. Thus, if the data structure satisfies all the constraints, it is a skip graph. ∎

Lemma 4.6 *Inserting a new node u in a skip graph S using Algorithm 2 gives a skip graph.*

Proof: Inserting a new node u in S consists of two stages: inserting u in level 0 using a search operation, and inserting u in levels $\ell > 0$ using the neighbors of u at level $\ell - 1$. We consider the case where the introducing node's key is less than u's key; the other case is similar so we omit those details here. Also, we only prove that Constraints 1, 3 and 5 are satisfied as Constraints 2, 4 and 6 are mirror images of Constraints 1, 3 and 5 respectively.

The search operation started by u (line 14 of Algorithm 2) returns the largest node s less than u (line 17 of Algorithm 1), and u sends a `getLinkOp` message to s (line 21 of Algorithm 2). One of the following two cases occur: Either s sets $sR_0' = u > s$ (line 7 of Algorithm 4), maintaining Constraint 1. Or, if additional nodes are inserted between u and s, and $sR_0 < u$ (line 3 of Algorithm 4), then s passes the `getLinkOp` message to sR_0. As the `getLinkOp` message is only passed to nodes whose key is less than that of u, eventually it reaches some node t where the message terminates and $t'R_0 = u > t$, maintaining Constraint 1. Also as u sets $uL_0' = s < u$ or $uL_0' = t < u'$ (line 23 of Algorithm 2), either $sR_0'L_0' = s$ or $tR_0'L_0' = t$ satisfying Constraint 3. In the absence of a suitable s, no pointers are changed and Constraints 1 and 3 are satisfied as the pointer values remain unchanged from before the insert.

Node u also determines the initial right neighbor of s, say z (lines 19 and 20 of Algorithm 2) and sends a `getLinkOp` message to z (line 24 of Algorithm 2). Similar to the earlier `getLinkOp` message, either z sets $z'L_0 = u < z$ (line 7 of Algorithm 4), or passes the message on to zL_0 if it is greater than u (line 3 of Algorithm 4). As the `getLinkOp` message is only passed to nodes whose key is greater than that of u, eventually it reaches some node y where the message terminates and $y'L_0 = u < y$. Node u sets $u'R_0 = z > u$ or $u'R_0 = y > u$ (line 26 of Algorithm 2), maintaining Constraint 1. As z sets $zL_0' = u < z$ or y sets $yL_0' = u < y$ (line 7 of Algorithm 4), $uR_0'L_0' = u$ satisfying Constraint 3. In the absence of a successor, u simply sets $uR_0' = \perp$ (line 2 or 26 of Algorithm 2), thus trivially satisfying Constraints 1 and 3.

Node u uses its neighbors at level ℓ, ($\ell \geq 0$), to find its neighbors at level $\ell+1$. Node u sends a `buddyOp` message to $uR_\ell > u$ (line 32 of Algorithm 2), and this message is passed to the right to successive nodes $uR_\ell^k > u$, $k \geq 1$, until it reaches a node y such that $m(u) \upharpoonright (\ell+1) = m(y) \upharpoonright (\ell+1)$, and u sets $uR'_{\ell+1} = y > u$ (line 34 of Algorithm 2), satisfying Constraint 1. As this message is only sent to the right, u can only connect on its right to nodes greater than itself. As y sets $yL'_{\ell+1} = u < y$, $uR'_{\ell+1}L'_{\ell+1} = u$, satisfying Constraint 3. Similarly, u also sends a `buddyOp` messages to its left to $uL_\ell < u$ (line 37 of Algorithm 2); as this message is only sent to nodes s less that u, it ensures that $sR'_{\ell+1} = u > s$, satisfying Constraint 1. As u sets $uL'_{\ell+1} = s$, $sR'_{\ell+1}L'_{\ell+1} = s$ satisfying Constraint 3.

As u only queries the nodes z in the same list as itself at level ℓ, it is ensured that $m(u) \upharpoonright \ell = m(z) \upharpoonright \ell$. Further, we see that u only links to a node z such that $m(u)_\ell = m(z)_\ell$ (line 10 of Algorithm 3). Thus u can only link to z at level $(\ell+1)$ if $m(u) \upharpoonright (\ell+1) = m(z) \upharpoonright (\ell+1)$. Having seen that u only links to nodes with the correct membership vector prefix, it only remains to show that u links to the nearest such nodes at each level $\ell > 0$. We see that u starts looking for it successor at level $\ell+1$ from uR_ℓ (line 31 of Algorithm 2). Node uR_ℓ is either the successor for node u at level $\ell+1$ (line 10 of Algorithm 3), or it passes the message to its successor (line 14 of Algorithm 3). As the search for $uR_{\ell+1}$ proceeds one node at a time along $S_{m(u)\upharpoonright(\ell)}$, it is guaranteed to find the *nearest* node greater than u, whose membership vector matches $m(u)_\ell$. Thus $uR_{\ell+1} = uR_\ell^k$, for the smallest $k > 0$, satisfying Constraint 5.

We note that with concurrent inserts, additional nodes may get linked at some level between u and its predetermined neighbors, found using either the search operation (for level 0) or the `buddyOp` messages (for levels greater than 0). In each case, we see that when some old node receives a `getlinkOp` messages to link to u, it verifies that pointing to u will maintain the skip graph node order. Otherwise, it passes the message to its appropriate neighbor (line 4 of Algorithm 4). This is explained in detail above, and it ensures that u links to the correct nodes at each level. So the constraints are maintained even with concurrent inserts in the skip graph.

Thus when all the concurrent insert operations are completed, we get a skip graph. ∎

Lemma 4.7 *Deleting node u from a skip graph S using Algorithm 6 gives a skip graph.*

Proof: Deleting a node u from a skip graph S consists of two stages at each level ℓ: finding a node to the right of u that is not being deleted, and then finding a node to the left of u that is not being deleted to link these two nodes together.

Node u sends a `deleteOp` message to its successor uR_ℓ (line 4 of Algorithm 6). As long as this message is received by a node that is itself being deleted (line 2 of Algorithm 7), it is passed on to the right to successive nodes $uR_\ell^2, uR_\ell^3 \dots$ (line 4 of Algorithm 7), until it reaches some node $z > u$ which is not being deleted. Node z sends a `findNeighborOp` message to zL_ℓ to determine its new left neighbor (line 8 of Algorithm 7). As long as this message is received by a node that is itself being deleted (line 13 of Algorithm 7), it is passed on to the left to successive nodes $zL_\ell^2, zL_\ell^3 \dots$ (line 15 of Algorithm 7), until it reaches some node $s < u < z$ which is not being deleted. Node s sends a `foundNeighborOp` message back to z and sets $sR'_\ell = z > s$, satisfying Constraint 1 (lines 19 and 20 of Algorithm 7). Upon receipt of this message (line 9 of Algorithm 7), z sets $zL'_\ell = s < z$ (line 10 of Algorithm 7), satisfying Constraint 2. Further, as $zL'_\ell R'_\ell = z$ and $sR'_\ell L'_\ell = s$, Constraints 3 and 4 are also satisfied. Nodes z and s are in the same list at level ℓ, so $m(z) \upharpoonright \ell = m(s) \upharpoonright \ell$, and they are also in the same list at level $\ell - 1$. As all the nodes between them will be eventually deleted, they are nearest nodes with matching membership vectors to be linked at level ℓ, satisfying Constraints 5 and 6.

If no suitable node s exists, the last node on the left of z that is being deleted, sends a `foundNeighborOp` to z (line 17 of Algorithm 7). Node z sets $zL'_\ell = \perp$ (line 10 of Algorithm 7), thus trivially satisfying Constraints 2, 4 and 6. If no suitable node z exists, the last node to the right of u that is being deleted, informs u of that (line 6 of Algorithm 7). Node u sends a `setNeighborNilOp` message to uL_ℓ (line 8 of Algorithm 6). which is passed to the left until it reaches a node q that is not being deleted (line 24 of Algorithm 7). Node q sets $qR'_\ell = \perp$ (line 29 of Algorithm 6), once again trivially satisfying Constraints 1, 3 and 5. If no suitable node q exists, then no link changes are made at all, and all the constraints are satisfied as before.

We note that with concurrent deletes, additional nodes may get deleted at some level between u and its existing neighbors. We see that when some neighboring node receives a message for the delete operation, it verifies that it is not being deleted itself. If so, it passes the message on to its neighbor as explained in detail above. This ensures that only nodes on either side of u that are not being deleted link to each other.

Thus, after all the concurrent delete operations have been completed, we get a skip graph. ∎

4.6 Applications of skip graphs

As skip graphs do not destroy the ordering of the keys, they can be easily used to support applications such as versioning and user-level replication which are explained below.

Versioning One application which is easily implemented using a skip graph is a version control system, for example, maintaining daily news articles as shown in Figure 4.6. The user can request for the latest version and in the absence of that, the system can seamlessly provide the most recent copy available, using the simple search mechanism. In addition, the user can formulate complex queries such as finding the latest article before a particular date or finding all articles within a specified time frame.

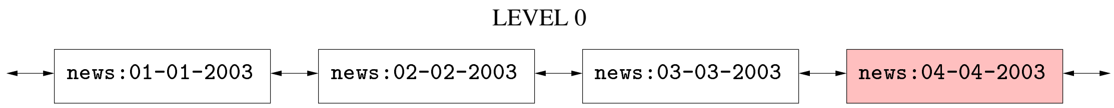


Figure 4.6: Using a skip graph for a versioning application. The user can look for the latest news article, for example, `news:04-05-2003` (where `04-05-2003` is the date on the day of the request and therefore is the most recent date). As that is not present, the system finds the next most recent article dated `04-04-2003`.

User-level replication Skip graphs can be organized to handle *flash crowds*, where the temporary popularity of a resource causes a very large request volume. Recent examples of flash crowds include overwhelming demand for news sites such as CNN and MSNBC during the terrorist attacks in the USA on September 11, 2001. Between 9:26 a.m. and 10:06 a.m., the number of searches for "CNN" on the search engine Google [46] averaged approximately 6000 queries per minute[†]. The flash crowd phenomenon is also popularly known as the "slashdot effect" because of the stirs caused by several stories of the popular web site `http://www.slashdot.com`.

We can utilize user-level replication to alleviate the problem of hot spots caused due to flash crowds. Suppose there is a very popular resource, for example, the web page `http://www.cnn.com`. To avoid a hot spot, the machine that hosts the resource can make several copies of it by appending identifiers at the end of key and adding these new resources as nodes to the data structure. If the resources are uniformly distributed among the machines, it is highly likely that these new resources are assigned to different machines in the system. Any user can run a query of the form `www.cnn.com*` and reach one of the replicas, as shown in Figure 4.7. Depending on where the search starts, different requests will go to different replicas, and effectively provide hot spot management by not overloading any one particular replica. In addition, it also provides survivability, so if a few replicas fail, the others are still available in the network. This mechanism can be used either for replicas of the resource addresses, or for replicas of the resources themselves. The first approach alleviates the load on the peer that maintains the address of the original resource whereas the second one alleviates the load on the peer that actually hosts the resource.

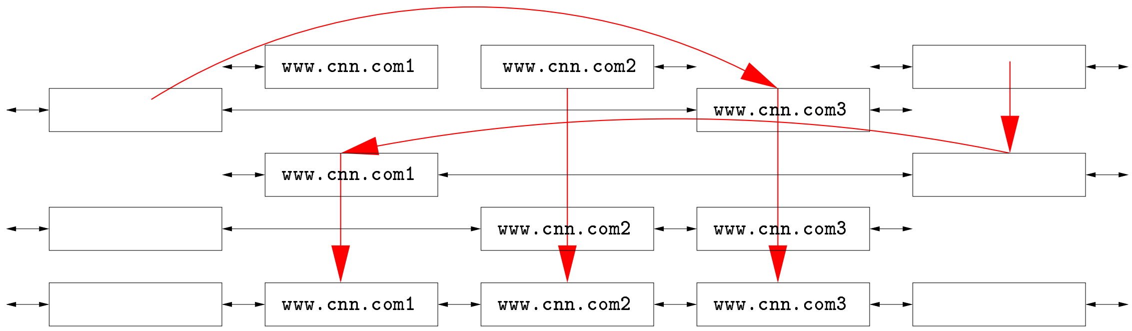


Figure 4.7: Using a skip graph for user-level replication.

We note that these applications can also be implemented using DHTs but they may require an additional application layer and the associated maintenance costs, over the native data structure.

[†]Details of these numbers can be found at: `http://www.google.com/press/zeitgeist/9-11.html`.

4.7 Fault tolerance

In this section, we describe some of the fault tolerance properties of a skip graph with alphabet $\{0,1\}$. Fault tolerance of related data structures, such as augmented versions of linked lists and binary trees, has been well-studied and some results can be seen in [83, 8]. In Section 4.8, we give a repair mechanism that detects node failures and initiates actions to repair these failures. Before we explain the repair mechanism, we are interested in the number of nodes that can be separated from the primary component by the failure of other nodes, as this determines the size of the surviving skip graph after the repair mechanism finishes.

Note that if multiple nodes are stored on a single machine, when that machine crashes, all of its nodes vanish simultaneously. Our results are stated in terms of the fraction of nodes that are lost; if the nodes are roughly balanced across machines, this will be proportional to the fraction of machine failures. Nonetheless, it would be useful to have a better understanding of fault tolerance when the mapping of resources to machines is taken into account; this may in fact dramatically improve fault tolerance, as nodes stored on surviving machines can always find other nodes stored on the same machine, and so need not be lost even if all of their neighbors in the skip graph are lost.

We consider two fault models: a random failure model in which an adversary chooses random nodes to fail, and a worst-case failure model in which an adversary chooses specific nodes to fail after observing the structure of the skip graph For a random failure pattern, experimental results, presented in Section 4.7.1, show that for a reasonably large skip graph nearly all nodes remain in the primary component until about two-thirds of the nodes fail, and that it is possible to make searches highly resilient to failures even without using the repair mechanism, by the use of redundant links. For a worst-case failure pattern, theoretical results, presented in Section 4.7.2, show that even a worst-case choice of failures causes limited damage. With high probability, a skip graph with n nodes has an $\Omega(\frac{1}{\log n})$ expansion ratio, implying that at most $O(f \cdot \log n)$ nodes can be separated from the primary component by f failures. We do not give experimental results for adversarial failures as experiments may not be able to identify the worst-case failure pattern.

4.7.1 Random failures

In our simulations, skip graphs appear to be highly resilient to random failures. We constructed a skip graph of 131072 nodes, where each node was had a unique label from $[1, 131072]$. We progressively increased the probability of node failure and measured the size of largest connected component of the live nodes as well as the number of isolated nodes as a fraction of the total number of nodes in the graph. As shown in Figure 4.8, nearly all nodes remain in the primary component even as the probability of individual node failure exceeds 0.6. We also see that a lot of nodes are isolated as the failure probability increases because all of their immediate neighbors die.

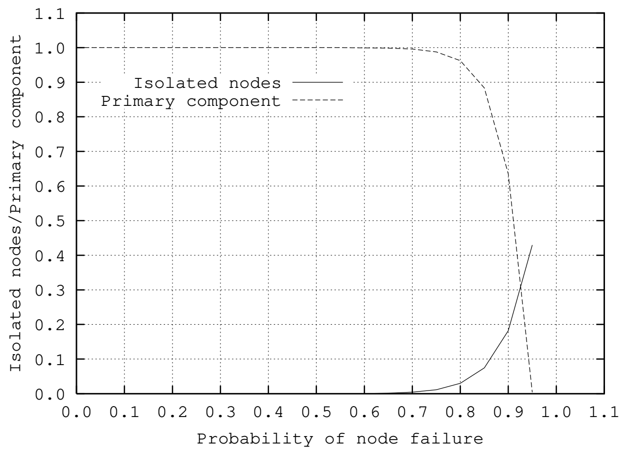


Figure 4.8: The number of isolated nodes and the size of the primary component as a fraction of the surviving nodes in a skip graph with 131072 nodes.

For searches, the fact that the average search involves only $O(\log n)$ nodes establishes trivially that most searches succeed as long as the proportion of failed nodes is substantially less than $O(\frac{1}{\log n})$. By detecting failures locally and using additional redundant edges, we can make searches highly tolerant to small numbers of random faults.

Some further experimental results are shown in Figure 4.9. In these experiments, each node had additional links to up to five nearest successors at every level. A total of 10000 messages were sent between randomly chosen source and destination nodes, and the fraction of failed searches was measured. We see that skip graphs are quite resilient to random failures. This plot appears to contradict the one shown in Figure 4.8, because we would expect

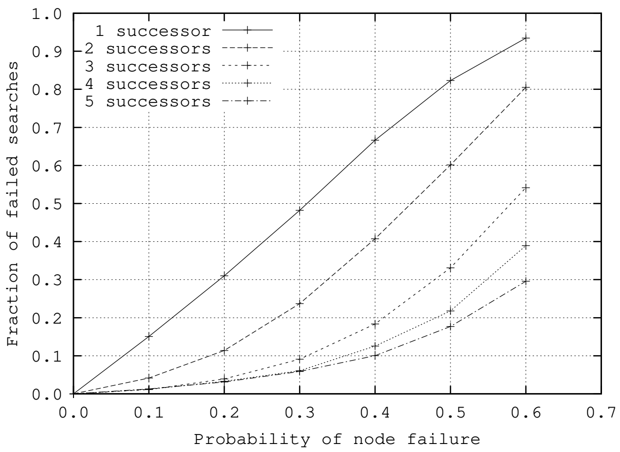


Figure 4.9: Fraction of failed searches in a skip graph with 131072 nodes and 10000 messages. Each node has up to five successors at each level.

all the searches to succeed as long as all live nodes are in the same connected component. However, once the source and target nodes are fixed, there is a fixed, deterministic path along which the search proceeds and if any node on this path fails, the search fails. So there may be *some* path between the source and the destination nodes, putting them in the same connected component, but the path used by the search algorithm may be broken, foiling the search. This suggests that if we use smarter search techniques, such as jumping between the different skip lists that a node belongs to, we can get much better search performance even in the presence of failures.

In general, skip graphs do not provide as strong guarantees as those provided by data structures based on explicit use of expanders such as censorship-resistant networks [35, 102, 28]. But we believe that this is compensated for by the simplicity of skip graphs and the existence of good distributed mechanisms for constructing and repairing them.

4.7.2 Adversarial failures

In addition to considering random failures, we are also interested in analyzing the performance of a skip graph when an adversary can observe the data structure, and choose specific nodes to fail. Experimental results may not even be able to identify these worst-case failure patterns. So in this section, we look at the expansion ratio of a skip graph, as that gives us the number of nodes that can be separated from the primary component even with adversarial failures.

Let G be a graph. Recall that the expansion ratio of a set of nodes A in G is $|\delta A|/|A|$, where $|\delta A|$ is the number of nodes that are not in A but are adjacent to some node in A. The expansion ratio of the graph G is the minimum expansion ratio for any set A, for which $1 \leq |A| \leq n/2$. The expansion ratio determines the resilience of a graph in the presence of adversarial failures, because separating a set A from the primary component requires all nodes in δA to fail. We will show that skip graphs have $\Omega(\frac{1}{\log n})$ expansion ratio with high probability, implying that only $O(f \cdot \log n)$ nodes can be separated by f failures, even if the failures are carefully targeted.

Our strategy for showing a lower bound on the expansion ratio of a skip graph will be to show that with high probability, all sets A either have large $\delta_0 A$ (i.e., many neighbors at the bottom level of the skip graph) or have large $\delta_\ell A$ for some particular ℓ chosen based on the size of A. Formally, we define $\delta_\ell A$ as the set of all nodes that are not in A but are joined to a node in A by an edge at level ℓ. Our result is based on the observation that $\delta A = \bigcup_\ell \delta_\ell A$ and $|\delta A| \geq \max_\ell |\delta_\ell A|$. We begin by counting the number of sets A of a given size that have small $\delta_0 A$.

Lemma 4.8 *In a n-node skip graph with alphabet $\{0, 1\}$, the number of sets A, where $|A| = m < n$ and $|\delta_0 A| < s$, is less than $\sum_{r=1}^{s-1} \binom{m+1}{r} \binom{n-m-1}{r-1}$.*

Proof: Without loss of generality, assume that the nodes of the skip graph are numbered from 1 to n. Given a subset A of these nodes, define a corresponding bit-vector x by letting $x_i = 1$ if and only if node i is in A. Then $\delta_0 A$ corresponds to all zeroes in x that are adjacent to a one.

Consider the extended bit-vector $x' = 1x1$ obtained by appending a one to each end of x. Because x' starts and ends with a one, it can be divided into alternating intervals of ones and zeroes, of which $r + 1$ intervals will consist of ones and r will consist of zeroes for some r, where $r > 0$ since x contains at least one zero. Observe that each interval of zeroes contributes at least one and at most two of its endpoints to $\delta_0 A$. It follows that $r \leq |\delta_0 A| \leq 2r$, and thus any A for which $|\delta_0 A| < s$ corresponds to an x for which x' contains $r \leq |\delta_0 A| < s$ intervals of zeroes.

Since there is at least one A with $r < s$ but $|\delta_0 A| \geq s$, the number of sets A with $|\delta_0 A| < s$ is strictly less than the number of sets A with $r < s$. By counting the latter quantity, we get a strict upper bound on the former.

We now count, for each r, the number of bit-vectors x' with $n - m$ zeroes consisting of $r + 1$ intervals of ones and r intervals of zeroes. Observe that we can characterize such a bit-vector completely by specifying the nonzero length of each of the $r + 1$ all-one intervals together with the nonzero length of each of the r all-zero intervals. There are $m + 2$ ones that must be distributed among the $r + 1$ all-one intervals, and there are $\binom{m+2-1}{r+1-1} = \binom{m+1}{r}$ ways to do so. Similarly, there are $n - m$ zeroes to distribute among the r all-zero intervals, and there are $\binom{n-m-1}{r-1}$ ways to do so. Since these two distributions are independent, the total count is exactly $\binom{m+1}{r}\binom{n-m-1}{r-1}$.

Summing over all $r < s$ then gives the upper bound $\sum_{r=1}^{s-1} \binom{m+1}{r}\binom{n-m-1}{r-1}$. ∎

For levels $\ell > 0$, we show with a probabilistic argument, that $|\delta_\ell A|$ is only rarely small

Lemma 4.9 *Let A be a subset of $m \leq n/2$ nodes of a n-node skip graph S with alphabet $\{0, 1\}$. Then for any ℓ,* $\Pr\left[|\delta_h A| \leq \frac{1}{3} \cdot 2^h\right] < 2\binom{2^\ell}{\lfloor \frac{2}{3} \cdot 2^h \rfloor}(2/3)^m$.

Proof: The key observation is that for each b in $\{0, 1\}^\ell$, if A contains a node u with $m(u) \restriction \ell = b$ and A's complement $S - A$ contains a node v with $m(v) \restriction \ell = b$, then there exist nodes $u' \in A$ and $v' \in S - A$ along the path from u to v in S_b, such that u' and v' are adjacent in S_b. Furthermore, since such pairs are distinct for distinct b, we get a lower bound on $\delta_\ell A$ by computing a lower bound on the number of distinct b for which A and $S - A$ both contain at least one node in S_b.

Let $T(A)$ be the set of $b \in \{0, 1\}^\ell$ for which A contains a node of S_b, and similarly for $T(S - A)$. Then

$$\begin{aligned}
\Pr\left[|T(A)| < \frac{2}{3} \cdot 2^\ell\right] &\leq \sum_{B \subset S, |B| = \lfloor \frac{2}{3} \cdot 2^h \rfloor} \Pr\left[T(A) \subseteq B\right] \\
&\leq \binom{2^\ell}{\lfloor \frac{2}{3} \cdot 2^h \rfloor}(2/3)^{|A|},
\end{aligned}$$

and by the same reasoning,

$$\Pr\left[|T(S-A)| < \frac{2}{3}\cdot 2^\ell\right] \;\leq\; \binom{2^\ell}{\lfloor\frac{2}{3}\cdot 2^h\rfloor}(2/3)^{|S-A|}.$$

But if both $T(A)$ and $T(S-A)$ hit at least two-thirds of the b, then their intersection must hit at least one-third, and thus the probability that $T(A)\cap T(S-A) < \frac{1}{3}\cdot 2^\ell$ is at most $\binom{2^\ell}{\lfloor\frac{2}{3}\cdot 2^h\rfloor}\left((2/3)^{|A|}+(2/3)^{|S-A|}\right)$, which is in turn bounded by $2\binom{2^\ell}{\lfloor\frac{2}{3}\cdot 2^h\rfloor}(2/3)^{|A|}$ under the assumption that $|A|\leq|S-A|$. ∎

We can now get the full result by arguing that there are not enough sets A with small $|\delta_0 A|$ (Lemma 4.8) to get a non-negligible probability that any one of them has small $|\delta_\ell A|$ for an appropriately chosen ℓ (Lemma 4.9). Details are given in the proof of Theorem 4.10 below.

Theorem 4.10 *Let $c\geq 6$. Then a skip graph with n nodes and alphabet $\{0,1\}$, has an expansion ratio of at least $\frac{1}{c\log_{3/2} n}$ with probability at least $1-\alpha n^{5-c}$, where the constant factor α does not depend on c.*

Proof: We will show that the probability that a skip graph S with n nodes does *not* have the given expansion ratio is at most αn^{5-c}, where $\alpha = 31$. The particular value of $\alpha = 31$ may be an artifact of our proof; the actual constant may be smaller.

Consider some subset A of S with $|A| = m \leq n/2$. Let $s = \frac{m}{c\log_{3/2} n} = \frac{m\lg(3/2)}{c\lg n}$, and let $s_1 = \lceil s\rceil$. We wish to show that, with high probability, all A of size m have $|\delta A|\geq s_1$. We will do so by counting the expected number of sets A with smaller expansion. Any such set must have both $|\delta_0 A| < s$ and $|\delta_\ell A| < s$ for any ℓ; our strategy will be to show first that there are few sets A in the first category, and that each set that does have small $\delta_0 A$ is very likely to have large $\delta_\ell A$ for a suitable ℓ.

By Lemma 4.8, there are at most $\sum_{r=1}^{s_1-1}\binom{m+1}{r}\binom{n-m-1}{r-1}$ sets A of size m for which $|\delta_0 A| < s_1$. It is not hard to show that the largest term in this sum dominates. Indeed, for $r < s_1$, the ratio between adjacent terms $\binom{m+1}{r}\binom{n-m-1}{r-1}/\binom{m+1}{r+1}\binom{n-m-1}{r}$ equals $\frac{r}{m+1-r}\cdot\frac{r+1}{n-m-r}$, and since $r < \frac{1}{6}m$ and $m \leq \frac{n}{2}$, this product is easily seen to be less than $\frac{1}{2}$. It follows that

$$\begin{aligned}
\sum_{r=1}^{s_1-1} \binom{m+1}{r}\binom{n-m-1}{r-1} &< \binom{m+1}{s_1-1}\binom{n-m-1}{s_1-2}\sum_{i=0}^{\infty} 2^{-i} \\
&= 2\binom{m+1}{s_1-1}\binom{n-m-1}{s_1-2} \\
&< n^{2(s_1-1)} \\
&< n^{2s} = n^{\frac{2m}{c\log_{3/2} n}} = 2^{\frac{2m\lg(3/2)}{c}}.
\end{aligned}$$

Now let $\ell = \lceil \lg s + \lg 3 \rceil$, so that $s \leq \frac{1}{3}2^{\ell} \leq 2s$.

Applying Lemma 4.9, we have

$$\begin{aligned}
\Pr\left[|\delta_\ell A| < s|\right] &\leq \Pr\left[|\delta_h A| \leq \frac{1}{3}\cdot 2^h\right] \\
&< 2\binom{2^\ell}{\lfloor \frac{2}{3}\cdot 2^h \rfloor}(2/3)^m \\
&\leq 2\binom{6s}{2s+1}(2/3)^m \\
&< 2\cdot(6s)^{2s+1}(2/3)^m,
\end{aligned}$$

and thus

$$\sum_{A\subset S, |A|=m, |\delta_0 A|<s} \Pr\left[|\delta_\ell A| < s|\right] < 2^{\frac{2m\lg(3/2)}{c}}\cdot 2\cdot(6s)^{2s+1}(2/3)^m. \tag{4.1}$$

Taking the base-2 logarithm of the right-hand side gives

$$\begin{aligned}
&\frac{2m\lg(3/2)}{c} + 1 + (2s+1)\lg(6s) - m\lg(3/2) \\
&< \frac{2m\lg(3/2)}{c} + 1 + \left(\frac{3m\lg(3/2)}{c\lg n}\right)\lg n - m\lg(3/2) \\
&= 1 + m\left(\frac{5}{c} - 1\right)\lg(3/2).
\end{aligned}$$

It follows that the probability that there exists an A of size m, for which both $|\delta_0 A|$ and $|\delta_\ell A|$ are less than s, is at most 2 to the above quantity, which we can write as $2b^m$ where $b = (3/2)^{(5/c-1)}$.

To compute the probability that any set has a small neighborhood, we sum over m. By definition of the expansion ratio, we need only consider values of m less than or equal to

$n/2$; however, because every proper subset A of S has at least one neighbor, we need to consider only $m > c \log_{3/2} n$. So we have

$$\begin{aligned}
\Pr\left[S \text{ has expansion ratio less than } \tfrac{1}{c \log_{3/2} n}\right] &< \sum_{m=\lceil c \log_{3/2} n \rceil}^{n/2} 2b^m \\
&< 2\left(\sum_{i=0}^{\infty} b^i\right)\left(b^{\lceil c \log_{3/2} n \rceil}\right) \\
&\le \frac{2}{1-b} \cdot b^{c \log_{3/2} n} \\
&= \frac{2}{1-b} \cdot n^{c \log_{3/2} b} \\
&= \frac{2}{1-b} \cdot n^{c\left(\frac{5}{c}-1\right)} \\
&= \frac{2}{1-b} \cdot n^{5-c} \\
&< 31 \cdot n^{5-c},
\end{aligned}$$

where the last inequality follows from the assumption that $c \geq 6$. ∎

4.8 Repair mechanism

Although a skip graph can survive a few disruptions, it is desirable to avoid accumulating errors. A large number of unrepaired failures will degrade the ideal search performance of a skip graph. Replication alone may not guarantee robustness, and we need a repair mechanism that automatically heals disruptions. Further, as failures occur continuously, the repair mechanism needs to continuously monitor the state of the skip graph to detect and repair these failures. The goal of the repair mechanism is to take a defective skip graph and repair all the defects.

We describe the repair mechanism as follows: In Section 4.8.1, we show that the first two skip graph constraints, given in section 4.5.4, are always preserved in any execution of the skip graph. In Section 4.8.2, we show how the remaining constraints can be checked locally by every node, and give algorithms to repair errors which may exist in the data structure. In Section 4.8.3, we prove that the repair mechanism given in Section 4.8.2 repairs a defective skip graph to give a defectless one. We note that the repair mechanism is not a **self-stabilization** mechanism in the strong sense because it will not repair an arbitrarily linked skip graph and restore it to its valid state. Instead, we see that certain defective configurations are impossible given the particular types of failures we consider. Thus the repair mechanism only repairs those failures that occur starting from a defectless skip graph.

4.8.1 Maintaining the invariant

We again list the constraints that describe a skip graph, as given in Section 4.5.4. Let x be any node in the skip graph; then for all levels $\ell \geq 0$:

1. If $xR_\ell \neq \perp$, $xR_\ell > x$.
2. If $xL_\ell \neq \perp$, $xL_\ell < x$.
3. If $xR_\ell \neq \perp$, $xR_\ell L_\ell = x$.
4. If $xL_\ell \neq \perp$, $xL_\ell R_\ell = x$.
5. If $m(x) \restriction (\ell+1) = m(xR_\ell^k) \restriction (\ell+1)$ and $\nexists j, j < k, m(x) \restriction (\ell+1) = m(xR_\ell^j) \restriction (\ell+1)$, then $xR_{\ell+1} = xR_\ell^k$. Else, $xR_{\ell+1} = \perp$.
6. If $m(x) \restriction (\ell+1) = m(xL_\ell^k) \restriction (\ell+1)$ and $\nexists j, j < k, m(x) \restriction (\ell+1) = m(xL_\ell^j) \restriction (\ell+1)$, then $xL_{\ell+1} = xL_\ell^k$. Else, $xL_{\ell+1} = \perp$.

We define Constraints 1 and 2 as an **invariant** for a skip graph as they hold in all states even in the presence of failures. Constraints 3 to 6 may fail to hold with failures, but they can be restored by the repair mechanism. We call Constraints 3 and 4 the R and L **backpointer** constraints respectively, and Constraints 5 and 6 the R and L **inter-level** constraints respectively. Each node periodically checks to see if its backpointer or inter-level constraints have been violated. If it discovers an inconsistent constraint, it initiates the repair mechanism explained in Section 4.8.2.

We consider the failure of some node x as an atomic action which eliminates x from the skip graph, and effectively sets the corresponding pointers to it to $\perp$. If there are any pending messages to other nodes for them to change their pointers to point to x, when the messages are delivered, the corresponding pointers are set to $\perp$. In an actual implementation, each node y will periodically check to see if its neighbors at each level ℓ are alive, and in absence of a response, it will set $yR_\ell = \perp$ or $yL_\ell = \perp$. For the purposes of our proofs however, we will consider that when a node fails, the pointers to it are atomically set to $\perp$. It is possible that a node will detect its failed neighbors before it has to initiate some action, thus setting the pointers to $\perp$ anyway. If it does not, we can ensure that a node checks that its neighbors at level ℓ are alive before it processes a message that it receives for level ℓ.

We use xL'_ℓ and xR'_ℓ etc to denote the value of node x's predecessor and successor respectively *after* some operation has occurred.

We prove that the invariant is maintained for the insert and delete operations in the presence of node failures. Then we give a repair mechanism that uses the invariant constraints to repair any violated backpointer or inter-level constraints due to node failures.

Lemma 4.11 *Failures preserve the invariant when no operation is in progress.*

Proof: Suppose that the invariant holds in the absence of any failures. If a node x has a failed successor or predecessor at level i, we consider $xR'_i = \perp$ or $xL'_i = \perp$ respectively, which trivially satisfies Constraints 1 and 2. Thus, for all nodes y and all levels ℓ, yR'_ℓ and yL'_ℓ are either equal to their previous values or they are set to $\perp$, and the invariant is maintained. ∎

Lemma 4.12 *The invariant is maintained during an insert operation even in the presence of failures.*

Proof: Suppose that the invariant holds prior to the insert operation. We consider each link change during an insert operation and prove that this does not violate Constraint 1; we omit the details for Constraint 2 as it is a mirror image of Constraint 2. We also consider only the case where the introducing node s is less than the new node u that is being added as the other case is similar. The successor link changes in Algorithm 2 (and its subroutine Algorithm 4) during the insert operation are as follows:

- Line 23: Node u sends a `getLinkOp` message to s (line 21 of Algorithm 2). Node s either returns a `setLinkOp` message to u (line 6 of Algorithm 4), or passes the message to sL_0 (line 4 of Algorithm 4) if $u < s$. The latter case occurs when a new node has been inserted between s and sL_0 during concurrent inserts. Thus u's original `getLinkOp` messages is only passed to nodes smaller than u, and u receives the corresponding `setLinkOp` from a node p smaller than itself. Thus $pR'_0 = u > p$ (line 7 of Algorithm 4). If some node fails in this process or no suitable predecessor exists, all R links remain unchanged, thus satisfying Constraint 1.

- Line 26: Node u sends a `getLinkOp` message to $sR_0 > u$ (line 21 of Algorithm 2). Node sR_0 either returns a `setLinkOp` message to u (line 6 of Algorithm 4), or passes the message to sR_0^2 (line 4 of Algorithm 4) if $sR_0 < u$. The latter case occurs when a new node has been inserted between s and sR_0 during concurrent inserts. Thus u's original `getLinkOp` messages is only passed to nodes greater than itself, and it receives the corresponding `setLinkOp` from a node v greater than itself. Thus $uR'_0 = v > u$. If some node fails in this process or no suitable successor exists, $uR'_0 = \perp$, thus satisfying Constraint 1.

- Line 34: For level $\ell \geq 0$, u sends a `buddyOp` message to $uR_\ell > u$ (line 32 of Algorithm 2). If $m(uR_\ell)_\ell = m(u)_\ell$, then uR_ℓ sends a `setLinkOp` message to u (line 6 of Algorithm 4), and $uR'_{\ell+1} = uR_\ell > u$. Else, uR_ℓ sends the `buddyOp` message to uR_ℓ^2 (line 4 of Algorithm 4). Node u's original `buddyOp` message is only sent to nodes greater than itself, and it receives the corresponding `setLinkOp` message only from a node z greater than itself. Thus $uR'_{\ell+1} = z > u$. If some node fails in this process or no suitable successor exists, $uR'_{\ell+1} = \perp$, thus satisfying Constraint 1.

- Line 39: For level $\ell \geq 0$, u sends a `buddyOp` message to $uL_\ell < u$ (line 32 of Algorithm 2). If $m(uL_\ell)_\ell = m(u)_\ell$, then uL_ℓ sends a `setLinkOp` message to u (line 6 of Algorithm 4), and $uL_\ell R'_{\ell+1} = u > uL_\ell$. Else, uL_ℓ sends the `buddyOp` message to uL_ℓ^2 (line 4 of Algorithm 4). Node u's original `buddyOp` message is only sent to nodes smaller than itself, and it receives the corresponding `setLinkOp` message only from a node s smaller than itself. Thus $sR'_{\ell+1} = u < s$ (line 7 of Algorithm 4). If some node fails in this process or no suitable predecessor exists, all R links remain unchanged, thus satisfying Constraint 1.

Thus the invariant is maintained during an insertion, even in the presence of failures. ∎

Lemma 4.13 *The invariant is maintained during a delete operation even in the presence of failures.*

Proof: Suppose that the invariant holds prior to the delete operation. We consider each link change during a delete operation and prove that this does not violate Constraint 1; we omit the details for Constraint 2 as it is a mirror image of Constraint 2. The successor links changes in Algorithm 7 during the delete operation of node u are as follows:

- Line 20: At each level $\ell \geq 0$, u sends a `deleteOp` message to uR_ℓ (line 4 of Algorithm 6). If uR_ℓ is being deleted, it passes the message to uR_ℓ^2 (line 4 of Algorithm 7). This message is only passed to nodes greater than u until it reaches a node $v > u$ which is not being deleted. Node v sends a `findNeighborOp` message to vL_ℓ (line 8 of Algorithm 7), which is passed to the left (line 15) of Algorithm 7 until it reaches a node $s < u$ which is not being deleted. Node s sends a `foundNeighborOp` to v (line 19 of Algorithm 7), and it sets $sR'_\ell = v > s$. If no suitable s is found (line 17 of Algorithm 7), all the R links remain unchanged, thus satisfying Constraint 1.

- Line 29: If no suitable v, which is not being deleted, is found (line 6 of Algorithm 7), u sends a `setNeighborNilOp` to $uL_\ell < u$ (line 8 of Algorithm 7). This message is passed to the left (line 24 of Algorithm 7) until it reaches some node $s < u$ which is not being deleted. Node s sets $sR'_\ell = \perp$ (line 29 of Algorithm 7). If no suitable s is found (line 17 of Algorithm 7), all the R links remain unchanged, thus satisfying Constraint 1.

Thus the invariant is maintained during a deletion, even in the presence of failures. ∎

Combining Lemmas 4.11, 4.12 and 4.13 directly gives Theorem 4.14.

Theorem 4.14 *The invariant is maintained throughout any execution of a skip graph, even with failures.*

4.8.2 Restoring invalid constraints

The backpointer and inter-level constraints are violated during insert and delete operations as well as when a node fails. However, we will see that the repair mechanism needs to be triggered only for constraint violations caused due to failures, and not during the insert and delete operations. We give a repair mechanism in which each node periodically checks Constraints 3 to 6 and initiates actions to fix invalid constraints due to node failures.

Although Constraints 3 to 6 may be violated midway during an insert or a delete operation, once all the pending operations are completed, these constraints are satisfied. Thus we observe that the repair mechanism is required to restore these constraints only in case of node failures. When a node fails during an insert or a delete, it leads to violations of the backpointer and inter-level constraints of its neighbors. Each node also periodically checks it backpointer and inter-level constraints. In Algorithm 8, node x checks its backpointer constraints by sending `checkNeighborOp` messages to its neighbors at all levels (lines 3 and 5). Similarly, each node checks its inter-level constraints as explained in Section 4.8.2.2.

As shown in Figure 4.10, when node y fails during an insert at level 2 (after having successfully inserted itself at levels 0 and 1) or during a delete at level 1 (after having successfully deleted itself from Level 2), its neighbors x and z detect this failure. With the failure of y, x and z will detect inconsistencies in their constraints and initiate the mechanism to repair them. We prove that it is sufficient to detect and repair the violated constraints to restore the skip graph to its defectless state.

The repair mechanism is divided into two parts: the first part is used to repair the invalid backpointer constraints, and the second part is used to repair invalid inter-level constraints.

4.8.2.1 Restoring backpointer constraints

Each node x periodically checks that $xR_\ell L_\ell = x$ when $xR_\ell \neq \perp$, and that $xL_\ell R_\ell = x$ when $xL_\ell \neq \perp$ for all levels $0 \leq \ell \leq x.\text{maxLevel}$. It triggers the backpointer constraint repair mechanism (Algorithm 8) if it detects an inconsistency.

Lemma 4.15 *In the absence of new failures, inserts, and deletes, the repair mechanism*

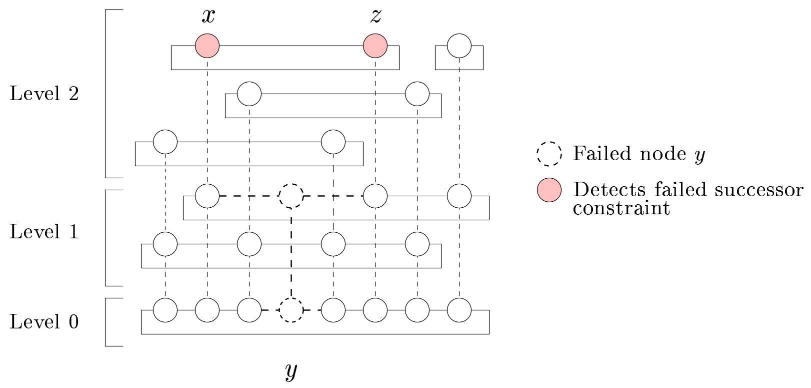


Figure 4.10: Violation of backpointer and inter-level constraints when a node fails half-way through an insert or delete operation. Observe that $xR_2 \neq x_1^j$, for any $j \geq 1$, and $zL_2 \neq z_1^k$, for any $k \geq 1$.

Algorithm 8: Algorithm for repairing invalid backpointer constraints for node v.

---

for $i \leftarrow v.max\_levels$ ***downto*** 0 **do**
    if ($v.neighbor[L][i] \neq \perp$) **then**
        send $\langle$`checkNeighborOp`, R, v, $i\rangle$ to v.neighbor$[L][i]$
    if ($v.neighbor[R][i] \neq \perp$) **then**
        send $\langle$`checkNeighborOp`, L, v, $i\rangle$ to v.neighbor$[R][i]$

upon receiving $\langle$`checkNeighborOp`, side, newNeighbor, $\ell\rangle$:
if *(side = R)* **then**
    cmp $\leftarrow <$
    otherSide $\leftarrow L$
else
    cmp $\leftarrow >$
    otherSide $\leftarrow R$
if *(v.neighbor[side][ℓ] $\neq$ newNeighbor)* **then**
    if *((v.neighbor[side][ℓ] = $\perp$)* **and** *(v cmp newNeighbor))* **then**
        v.neighbor[side][ℓ] $\leftarrow$ newNeighbor
        send $\langle$`checkNeighborOp`, otherSide, v, $\ell\rangle$ to newNeighbor
    else if *((v.neighbor[side][ℓ] $\neq \perp$)* **and** *(v.neighbor[side][ℓ] cmp newNeighbor))* **then**
        send $\langle$`checkNeighborOp`, side, newNeighbor, $\ell\rangle$ to
    else
        send $\langle$`checkNeighborOp`, otherSide, newNeighbor, $\ell\rangle$ to v.neighbor[side][ℓ]
        send $\langle$`checkNeighborOp`, otherSide, v, $\ell\rangle$ to newNeighbor
        send $\langle$`checkNeighborOp`, side, v.neighbor[side][ℓ], $\ell\rangle$ to newNeighbor
        v.neighbor[side][ℓ] $\leftarrow$ newNeighbor

---

described in Algorithm 8 repairs any violated backpointer constraint without losing existing connectivity.

Proof: We prove that Algorithm 8 repairs the violated backpointer constraints for a single node without losing existing connectivity. We concentrate on the repair of the R links as the case for L links is symmetric.

The violations of Constraint 3 for node v at level ℓ are as follows:

1. $vR_\ell = z > v$ but $zL_\ell = \perp$: Node v sends $\langle$`checkNeighborOp`, L, v, $\ell\rangle$ to z (line 5 of Algorithm 8). As $zL_\ell = \perp$ and $z > v$, z sets $zL'_\ell = v$ (line 15 of Algorithm 8). Thus after Algorithm 8 finishes, $vR'_\ell L'_\ell = v$, restoring Constraint 3.

2. $vR_\ell = z > v$ but $zL_\ell = y > v$: Node v sends $\langle$`checkNeighborOp`, L, v, $\ell\rangle$ to z (line 5 of Algorithm 8). As $zL_\ell = y > v$, z passes the message on to y (line 18 of Algorithm 8). As long as this message reaches some node $y > v$ such that $yL_\ell > v$, y will pass it on to yL_ℓ, until it reaches a node $x > v$ such that $xL_\ell < v < x$ or $xL_\ell = \perp$. Then x sets $xL'_\ell = v < x$ (lines 15 or 23 of Algorithm 8). Node x also sends $\langle$`checkNeighborOp`, R, x, $\ell\rangle$ to v (line 16 or 21 of Algorithm 8). Upon receiving that message, v sets $vR'_\ell = x$. Thus after Algorithm 8 finishes, $vR'_\ell L'_\ell = v$, restoring Constraint 3.

3. $vR_\ell = z > v$ but $zL_\ell = u < v$: Node v sends $\langle$`checkNeighborOp`, L, v, $\ell\rangle$ to z (line 5 of Algorithm 8). As $zL_\ell = u < v$, z sets $zL'_\ell = v < z$ (line 20 of Algorithm 8). Thus after Algorithm 8 finishes, $vR'_\ell L'_\ell = v$, restoring Constraint 3.

We prove that the backpointer constraint repair mechanism does not lose any existing connectivity in the skip graph, i.e, if a path between nodes v and z used to exist before the repair mechanism was initiated, a path will still exist after the repair mechanism operations finish. We consider the case where a node v changes its successor pointer to a node z and points to another node y; we omit the case where v changes its predecessor pointer as it is similar to this case. Node v updates its successor pointer to some node z at level ℓ, to point to some other node y, only when $vR_\ell = z > y > v$ (line 23 of Algorithm 8). Node v also sends messages to (i) z to update its predecessor pointer to point to y (line 20 of Algorithm 8), (ii) y to update its predecessor pointer to point to v (line 21 of Algorithm 8) and, (iii) y to update its successor pointer to point to z (line 22 of Algorithm 8). When these messages are delivered, $vR'_\ell = y$ and $yR'_\ell = z$. Thus the path v–z is now replaced by a longer path v–y–z, and no existing connectivity is lost. ∎

It is possible that a node x detects a failed backpointer constraint if it checks it while some node y is in the middle of its insert operation. Suppose that $xR_\ell = z$ but $zL_\ell = y$ because y is yet to connect to x. When x sends a `checkNeighborOp` message to z, it gets passed to y, which then links to x and asks x to link to it (both through the repair mechanism and the insert operation). Thus, the repair mechanism generates additional messages but does not affect the insert operation. In case of a delete operation, a node does not delete itself until it has repaired the links at all the levels so an inconsistent backpointer constraint will not be detected during the delete operation.

4.8.2.2 Restoring inter-level constraints

We see how each node periodically checks Constraint 5; we omit the details for Constraint 6 as it is a mirror image of Constraint 5. For each level $\ell > 0$, each node x sends a message to $xR_{\ell-1}$ to check if $xR_\ell = xR^k_{\ell-1}$, for some $k > 0$. Each node that receives the message passes it to the right until one of the four following cases occur:

1. The message reaches node a, $a < xR_\ell$ and $m(a) \upharpoonright \ell = m(x) \upharpoonright \ell$.

2. The message reaches node a, $a < xR_\ell$ and $aR_{\ell-1} = \perp$.

3. The message reaches node a, $a = xR_\ell$.

4. The message reaches node a, $a > xR_\ell$.

In case 3, Constraint 5 is not violated and no repair action is violated. We provide a repair mechanism for the each of the remaining three cases. The repair mechanism for fixing violations of Constraint 6 is symmetric. It may be possible to combine the two mechanisms to improve the performance but we will treat them separately for simplicity.

In each case, we assume that the link is present at level ℓ but absent at level $\ell - 1$. Note that if a node x is linked at level $\ell - 1$ but not at level ℓ, it can easily traverse the list at level $\ell - 1$ to determine which node to link to at level ℓ. This process is identical to the insertion process where a new node uses its neighbors at lower levels to insert itself at higher levels in the skip graph.

The violations of Constraint 5 are as follows:

1. $xR_\ell = xR^k_{\ell-1}$, but $\exists a = xR^j_{\ell-1}$, $j < k$, $m(x) \upharpoonright \ell = m(a) \upharpoonright \ell$.

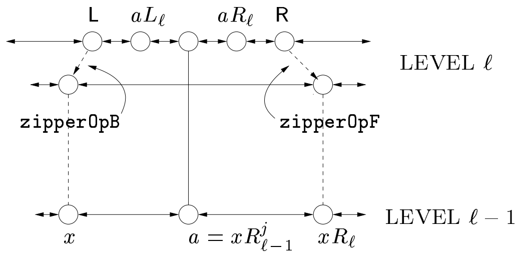


Figure 4.11: Two-way merge to repair a violated inter-level constraint.

The nodes connected to a and x at level ℓ have to be merged together into one list by sending the following messages:

- Probe level ℓ (in list containing a) to find largest $aR^{k'}_\ell = \mathsf{R} < xR_\ell$. Node a starts the probe by sending a message to aR_ℓ. Upon reaching node y, if $y < xR_\ell < yR_\ell$, the probe ends with $y = \mathsf{R}$. Otherwise the message is passed to yR_ℓ.
- Send $\langle$`zipperOpF`, xR_ℓ, $\ell\rangle$ to R.
- Probe level ℓ (in list containing a) to find smallest $aL^{k''}_\ell = \mathsf{L} > x$. Node a starts the probe by sending a message to aL_ℓ. Upon reaching node y, if $yL_\ell < x < y$, the probe ends with $y = \mathsf{L}$. Otherwise the message is passed to yL_ℓ.
- Send $\langle$`zipperOpB`, x, $\ell\rangle$ to L.

2. $xR_\ell \neq xR^k_{\ell-1}$ for any $k > 0$, and $\exists a < xR_\ell, aR_\ell = \perp$.

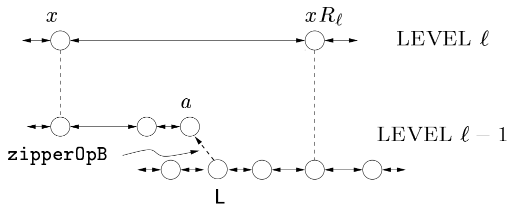


Figure 4.12: One-way merge to repair a violated inter-level constraint.

The nodes connected to a and xR_ℓ at level $\ell-1$ have to be merged together into one list by sending the following messages:

- Probe level $\ell-1$ (in list containing xR_ℓ) to find smallest $xR_\ell L^{k'}_{\ell-1} = \mathsf{L} > a$. Node xR_ℓ starts the probe by sending a message to $xR_\ell L_{\ell-1}$. Upon reaching node y, such that $y > a > yL_{\ell-1}$, the probe stops with $y = \mathsf{L}$. Otherwise the message is passed on to $yL_{\ell-1}$.
- Send $\langle$`zipperOpB`, a, $\ell-1\rangle$ to L.

3. $xR_\ell \neq xR^j_{\ell-1}$ for any $j > 0$, and $\exists a = xR^k_{\ell-1} > xR_\ell$.

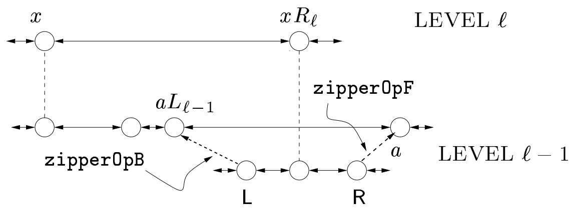


Figure 4.13: Two-way merge to repair a violated inter-level constraint.

The nodes connected to a and xR_ℓ at level $\ell-1$ have to be merged together into one list by sending the following messages[‡]:

- Probe level $\ell-1$ (in list containing xR_ℓ) to find largest $xR_\ell R^{k'}_{\ell-1} = \mathsf{R} < a$. Node xR_ℓ starts the probe by sending a message to $xR_\ell R_{\ell-1}$. Upon reaching node y, if $y < a < yR_{\ell-1}$, the probe ends with $y = \mathsf{R}$. Otherwise the message is passed to $yR_{\ell-1}$.
- Send $\langle$`zipperOpF`, a, $\ell-1\rangle$ to R.
- Probe level $\ell-1$ (in list containing xR_ℓ) to find smallest $xR_\ell L^{k"}_{\ell-1} = \mathsf{L} > aL_{\ell-1}$. In this case, the probe proceeds along the predecessors of xR_ℓ at level $\ell-1$ till it reaches node y such that $y = \mathsf{L} > aL_{\ell-1} > yL_{\ell-1}$.
- Send $\langle$`zipperOpB`, $aL_{\ell-1}$, $\ell-1\rangle$ to L.

[‡]Details of the `zipperOp` messages are given in Algorithms 9 and 10.

Algorithm 9: `zipperOpB` for node v

upon receiving $\langle$`zipperOpB`, x, $\ell\rangle$:
if v.*neighbor[L][ℓ]* $> x$ **then**
  send $\langle$`zipperOpB`, x, $\ell\rangle$ to v.neighbor$[L][\ell]$
else
  tmp $= v$.neighbor$[L][\ell]$
  v.neighbor$[L][\ell] = x$
  send $\langle$`updateOp`, R, v, $\ell\rangle$ to x
  if *tmp* $\neq \perp$ **then**
    send $\langle$`zipperOpB`, tmp, $\ell\rangle$ to x

Algorithm 10: `zipperOpF` for node v

upon receiving $\langle$`zipperOpF`, x, $\ell\rangle$:
if v.*neighbor[R][ℓ]* $< x$ **then**
  send $\langle$`zipperOpF`, x, $\ell\rangle$ to v.neighbor$[R][\ell]$
else
  tmp $= v$.neighbor$[R][\ell]$
  v.neighbor$[R][\ell]= x$
  send $\langle$`updateOp`, L, v, $\ell\rangle$ to x
  if *tmp* $\neq \perp$ **then**
    send $\langle$`zipperOpF`, tmp, $\ell\rangle$ to x

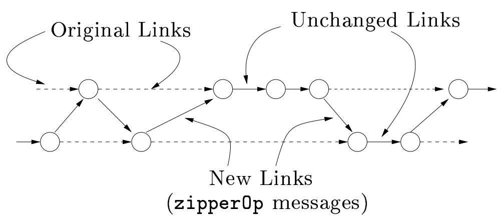


Figure 4.14: `zipperOp` operation to merge nodes on the same level.

Lemma 4.16 *In the absence of new failures, inserts, and deletes, the repair mechanism described in Section 4.8.2.2 repairs any violated inter-level constraint without losing existing connectivity.*

Proof: The algorithm initiates repair for all the possible violations of the inter-level constraints of a node as given above. It only remains to be proved that the `zipperOp` messages merge two sorted lists at a given level into a single sorted list, without losing existing connectivity. We concentrate on the `zipperOpF` messages as the `zipperOpB` messages are symmetric.

To prove that the repair mechanism merges two sorted lists into a single sorted one, we first see that a node v always receives a `zipperOpF` message to link to a node greater than itself. The initial `zipperOpF` messages sent are as follows: (i) In Case 3, R receives a `zipperOpF` message to link to $a > \mathsf{R}$, and (ii) in Case 1, R receives a `zipperOpF` message to link to $xR_\ell > \mathsf{R}$. When a node v receives a `zipperOpF` message to link to x, if $vR_\ell < x$, it sends the message to vR_ℓ to link to x (line 3 of Algorithm 10). Otherwise, it updates $vR'_\ell = x$ (line 6 of Algorithm 10), and it sends a `zipperOpF` message to $x < vR_\ell$ to link to vR_ℓ (line 9 of Algorithm 10). In both cases, the `zipperOpF` message reaches a node that has to link to a node greater than itself. Also, each node v only links to a new node x if it is smaller than the current successor of v, so $v < vR_\ell = x < vR_\ell$. Thus the two sorted lists get merged into a single sorted list, until one of the lists terminates.

We also prove that the inter-level constraint repair mechanism does not lose any existing connectivity in the skip graph, i.e, if a path between nodes v and z used to exist before the repair mechanism was initiated, a path will still exist after the repair mechanism operations finish. A link change occurs only when v receives a `zipperOpF` message to link to $x < vR_\ell = z$. Node v sends a `zipperOpF` message to x to link to z. Upon receipt of that message, x either sets to $xR'_\ell = z$, or it sends passes the message to $xR_\ell < z$. In the first case, the path between v and z is replaced by a new longer path v-x-z. In the latter case, the `zipperOp` message passes through several nodes $xR_\ell, xR^2_\ell, \ldots$, until it reaches a node y such that $y < z < yR_\ell$ and y sets $yR'_\ell = z$. Then the path v–z is replaced by a longer path

v–x–xR_ℓ–xR_ℓ^2–…–y–z, and no existing connectivity is lost. ∎

It is possible that a node x detects a failed inter-level constraint if it checks it while some node y is in the middle of its insert or delete operation. Node x will detect a failed constraint at level ℓ if y has inserted itself at level $\ell - 1$ and not at level ℓ. The probe message along level $\ell - 1$ will reach y which can then inform x that it is yet to complete its insert operation, and thus terminate the repair mechanism.

4.8.3 Proof of correctness

In this section we prove that the repair mechanism given in Algorithm 8 and Section 4.8.2.2 repairs a defective a defective skip graph.

We prove that the repair mechanism repairs a defective skip graph by showing that it repairs level 0 after some finite interval of time, and then uses the links at level 0 to restore the links at higher levels. Lemma 4.17 and Corollary 4.18 show that if there exists a path between two nodes that consists entirely of pointers in any one direction (L or R), then the repair mechanism ensures that after some finite interval of time, there is a path between those two nodes in the same direction at level 0. For their proofs, we consider only the case for the R links as the case for the L links is symmetric. This result is further extended in Lemma 4.19 which shows that as long as there is path between two nodes, irrespective of the directions of the edges in the path, there will be a path in both directions between the two nodes at level 0. Corollary 4.20 shows that this leads to a single, sorted, doubly-linked list at level 0 as in a defectless skip graph. Finally, Lemma 4.21 and Theorem 4.22 show how the list at level 0 is used to create lists at higher levels, to eventually give a defectless skip graph.

Lemma 4.17

1. *Suppose we have* $y_0 R_{\ell_1} R_{\ell_2} \dots R_{\ell_r} = y_r$, $y_0 < y_r$, *for some* r, *and for each* $1 \le i \le r$, $\ell \le \ell_i \le \ell + 1$. *Then after some finite interval of time,* $y_0 R_\ell^k = y_r$, *for some* k.

2. *Suppose we have* $y_0 L_{\ell_1} L_{\ell_2} \dots L_{\ell_r} = y_r$, $y_0 > y_r$, *for some* r, *and for each* $1 \le i \le r$, $\ell \le \ell_i \le \ell + 1$. *Then after some finite interval of time,* $y_0 L_\ell^k = y_r$, *for some* k.

Proof: Let $y_0 R_{\ell_1} R_{\ell_2} \ldots R_{\ell_i} = y_i$. Then there exists a link between each y_{i-1} and y_i at level ℓ or $\ell+1$. For each $R_{\ell_i} = R_{\ell+1}$, as $y_{i-1} R_{\ell+1} = y_i$, the inter-level repair mechanism given in Section 4.8.2.2 ensures that after some finite interval of time, $y_{i-1} R_\ell^{k_i} = y_i$, for some k_i (Lemma 4.16). We then have $y_0 R_{\ell_1} R_{\ell_2} \ldots R_{\ell_r} = y_0 R_\ell^{k_1} R_\ell^{k_2} \ldots R_\ell^{k_r} = y_r$, where $k_i = 1$ if $R_{\ell_i} = R_\ell$. Thus we get $y_0 R_\ell^k = y_r$, where $k = k_1 + k_2 + \ldots + k_r$. ∎

Corollary 4.18

1. *Suppose we have $y_0 R_{\ell_1} R_{\ell_2} \ldots R_{\ell_r} = y_r$, $y_0 < y_r$, for some r, and for each i, $1 \le i \le r$, $\ell_i \ge 0$. Then after some finite interval of time, $y_0 R_0^k = y_r$, for some k.*

2. *Suppose we have $y_0 L_{\ell_1} L_{\ell_2} \ldots L_{\ell_r} = y_r$, $y_0 > y_r$, for some r, and for each i, $1 \le i \le r$, $\ell_i \ge 0$. Then after some finite interval of time, $y_0 L_0^k = y_r$, for some k.*

Proof: Let $y_0 R_{\ell_1} R_{\ell_2} \ldots R_{\ell_i} = y_i$. Then for each y_{i-1}, $y_{i-1} R_{\ell_i} = y_i$. By Lemma 4.17, after some finite interval of time, $y_{i-1} R_{\ell_i - 1}^{k_{i,\ell_i-1}} = y_i$, for some k_{i,ℓ_i-1}. By repeatedly applying Lemma 4.17, after some finite interval of time, we get $y_{i-1} R_0^{k_{i,0}} = y_i$, for some $k_{i,0}$. So we get $y_0 R_0^{k_{1,0}} R_0^{k_{2,0}} \ldots R_0^{k_{r,0}} = y_r$. Thus, $y_0 R_0^k = y_r$, where $k = k_{1,0} + k_{2,0} + \ldots + k_{r,0}$. ∎

Lemma 4.19 *Suppose we have $y_0 E_{\ell_1} E_{\ell_2} \ldots E_{\ell_r} = y_r$, for some r, $y_0 < y_r$, $E \in \{L, R\}$, and for each $1 \le i \le r$, $\ell_i \ge 0$. Then after some finite interval of time, $y_0 R_0^k = y_r$ and $y_r L_0^k = y_0$ for some k.*

Proof: Let $y_0 E_{\ell_1} E_{\ell_2} \ldots E_{\ell_i} = y_i$. For each y_i, $y_{i-1} E_{\ell_i} = y_i$. By Corollary 4.18, after some finite interval of time, $y_{i-1} E_0^{k_i} = y_i$ for some k_i. So we have $y_0 E_0^{k_1} E_0^{k_2} \ldots E_0^{k_r} = y_r$. If any of the y_i's are not distinct, then we can eliminate the path between two consecutive occurrences of y_i. So we can replace a path of the form $y_i E_0^{k_i} \ldots y_j E_0^{k_j}$, where $y_i = y_j$, with $y_i E_0^{k_j}$. Thus we have a path consisting of L_0 and R_0 edges starting at y_0 and terminating at y_r which consists only of unique nodes.

For each node, after some finite interval of time, the backpointer constraint repair mechanism given in Algorithm 8 will repair any violated backpointer constraints without losing existing connectivity (Lemma 4.15). So Constraints 3 and 4 are repaired for all the nodes in the path, and as proved in Theorem 4.14, Constraints 1 and 2 are always maintained.

As proved in Theorem 4.5, with Constraints 1 through 4 satisfied for all the nodes in the path, we get a sorted, doubly-linked list of the nodes. Thus, $y_0 R_0^k = y_r$ and $y_r L_0^k = y_0$, for some k. ∎

Corollary 4.20 *After some finite interval of time, all nodes in the same connected component of a skip graph are linked together in a single, sorted, doubly-linked list at level 0.*

Proof: Lemma 4.19 shows that any two connected nodes x and y are in the same sorted, doubly-linked list at level 0 after some finite interval of time. Any other node z in the same connected component is also connected to both x and y, so it has to be in the same list at level 0. Thus, all the nodes in the same connected component are in a single list at level 0. ∎

Given a set S, let $M_\ell(S)$ be the set of all membership vector prefixes of length ℓ represented by the nodes in S, i.e., $M_\ell(S) = \{w \mid \exists x \in S, m(x) \upharpoonright \ell = w\}$.

Lemma 4.21 *Suppose we have all nodes in the same connected component C of a skip graph linked together in $|M_\ell(C)|$ sorted, doubly-linked lists at level ℓ, one for each $w \in M_\ell(C)$. Then after some finite interval of time, we get $|M_{\ell+1}(C)|$ sorted, doubly-linked lists at level $\ell+1$, one for each $w \in M_{\ell+1}(C)$.*

Proof: Consider a single list $\mathbb{L}$ at level ℓ, and let node $x \in \mathbb{L}$. As explained in the successor constraint repair mechanism given in Section 4.8.2.2, if $xL_\ell, xR_\ell \notin \{\perp\}$, x uses its neighbors at level ℓ, to find its neighbors at level $\ell+1$. As all the lists at level ℓ are sorted and doubly-linked, x can find $xR_{\ell+1}$ and $xL_{\ell+1}$ in $O(2|\Sigma|)$ time, just like in an insert operation (Algorithm 2). When all the nodes of list $\mathbb{L}$ determine their neighbors at level $\ell+1$ after some finite interval of time, we get the lists from the nodes of $\mathbb{L}$ at level $\ell+1$, one for each $w \in M_{\ell+1}(\mathbb{L})$. This is identical to the insert operation and as proved in Lemma 4.6, all the lists thus created are sorted and doubly-linked, and only consist of nodes that have the matching membership vector prefix of length $\ell+1$. Thus, considering all the $|M_\ell(C)|$ lists at level ℓ, after some finite interval of time, we get $|M_{\ell+1}(C)|$ sorted, doubly-linked lists at level $\ell+1$, one for each $w \in M_{\ell+1}(C)$. ∎

Theorem 4.22 *In the absence of new failures, inserts and deletes, the repair mechanism given in Section 4.8 repairs a defective skip graph to give a defectless skip graph after some finite interval of time.*

Proof: Corollary 4.20 shows that after some finite interval of time, the repair mechanism links all the nodes in the same connected component in a single, sorted, doubly-linked list at level 0. Further, there are only finitely many levels in a skip graph. Using Lemma 4.21 inductively, with Corollary 4.20 as the base case, we can show that we get sorted, doubly-linked lists, which contain all the nodes with the matching non-empty membership vector prefixes, at all levels of the skip graph. Thus, the repair mechanism repairs a defective skip graph to give a defectless skip graph after some finite interval of time. ∎

4.9 Congestion

In addition to fault tolerance, a skip graph provides a limited form of congestion control, by smoothing out hot spots caused by popular search targets. The guarantees that a skip graph makes in this case are similar to the guarantees made for survivability. Just as a node's continued connectivity depends on the survival of its neighbors, its message load depends on the popularity of its neighbors as search targets. However, we can show that this effect drops off rapidly with distance; nodes that are far away from a popular target in the bottom-level list of a skip graph get little increased message load on average.

We give two versions of this result. The first version, given in Section 4.9.1, shows that the probability that a particular search uses a node between the source and target drops off inversely with the distance from the node to the target. This fact is not necessarily reassuring to heavily-loaded nodes. Since the probability averages over all choices of membership vectors, it may be that some particularly unlucky node finds itself with a membership vector that puts it on nearly every search path to some very popular target. The second version, given in Section 4.9.2, shows that our average-case bounds hold with high probability. While it is still possible that a spectacularly unlucky node is hit by most searches, such a situation only occurs for very low-probability choices of membership vectors. It follows that most skip graphs alleviate congestion well. For our results, we consider skip graphs with alphabet $\{0, 1\}$.

4.9.1 Average congestion for a single search

Our argument that the average congestion is inversely proportional to distance is based on the observation that a node only appears on a search path in a skip list S if it is among the tallest nodes between itself and the target. We will need a small technical lemma that counts the expected number of such tallest nodes. Consider a set-valued Markov process $A_0 \supseteq A_1 \supseteq A_2 \ldots$ where A_0 is some nonempty initial set and each element of A_t appears in A_{t+1} with independent probability $\frac{1}{2}$. Let τ be the largest index for which A_τ is not empty. We will now show that $\mathrm{E}[|A_\tau|]$ is small regardless of the size of the initial set A_0.

Lemma 4.23 *Let $A_0, A_1, \ldots, A_\tau$ be defined as above. Then $\mathrm{E}[|A_\tau|] < 2$.*

Proof: The bound on $\mathrm{E}[|A_\tau|]$ will follow from a surprising connection between $\mathrm{E}[|A_\tau|]$ and $\Pr[|A_\tau| = 1]$. We begin by obtaining a recurrence for $\Pr[|A_\tau|] = 1$.

Let $P(n) = \Pr[|A_\tau| = 1 : |A_0| = n]$. Clearly $P(0) = 0$ and $P(1) = 1 \geq \frac{2}{3}$. For larger n, summing over all $k = |A_1|$ gives $P(n) = 2^{-n} \sum_{k=0}^{n} \binom{n}{k} P(k)$, which can be rewritten as $P(n) = \frac{1}{2^n - 1} \sum_{k=1}^{n-1} \binom{n}{k} P(k)$. The solution to this recurrence goes asymptotically to $\frac{1}{2 \ln 2} \pm 10^{-5} \approx 0.7213\cdots$ [106, Theorem 7.9]; however, we will use the much simpler property that when $n \geq 2$, $P(n) = \Pr[|A_\tau| = 1]$ is the probability of an event that does not always occur, and is thus less than 1.

Let $E(n) = \mathrm{E}[|A_\tau| : |A_0| = n]$. Then $E(n) = 2^{-n} \left(n + \sum_{k=1}^{n} \binom{n}{k} E(k) \right)$, and eliminating the $E(n)$ term on the right-hand side gives $E(n) = \frac{1}{2^n - 1} \left(n + \sum_{k=1}^{n-1} \binom{n}{k} E(k) \right)$. For $n = 1$, $E(1) = 1$ by definition. Recall that $P(1)$ is also equal to 1. We will now show that $E(n) = 2P(n)$ for all $n > 1$. Suppose that $E(k) = 2P(k)$ for $1 < k < n$. Let $n = 2$, then

$E(k) = 2P(k)$ for all $1 < k < n$ (an empty set of k). Then,

$$
\begin{aligned}
E(n) &= \frac{1}{2^n - 1}\left(n + \sum_{k=1}^{n-1}\binom{n}{k}E(k)\right) \\
&= \frac{1}{2^n - 1}\left(n + \binom{n}{1}E(1) + 2\sum_{k=2}^{n-1}\binom{n}{k}P(k)\right) \\
&= \frac{1}{2^n - 1}\left(2\binom{n}{1}P(1) + 2\sum_{k=2}^{n-1}\binom{n}{k}P(k)\right) \\
&= 2 \cdot \frac{1}{2^n - 1}\sum_{k=1}^{n-1}\binom{n}{k}P(k) \\
&= 2P(n).
\end{aligned}
$$

Since $P(n) < 1$ for $n > 1$, we immediately get $E(n) < 2$ for all n. For large n this is an overestimate: given the asymptotic behavior of $P(n)$, $E(n)$ approaches $\frac{1}{\ln 2} \pm 2 \times 10^{-5} \approx 1.4427\cdots$. But it is close enough for our purposes. ∎

Theorem 4.24 *Let S be a skip graph with alphabet $\{0, 1\}$, and consider a search from s to t in S. Let u be a node with $s < u < t$ in the key ordering (the case $s > u > t$ is symmetric), and let d be the distance from u to t, defined as the number of nodes v with $u < v \leq t$. Then the probability that a search from s to t passes through u is less than $\frac{2}{d+1}$.*

Proof: Let $S_{m(s)}$ be the skip list restriction of s whose existence is shown by Lemma 4.1. From Lemma 4.2, we know that searches in S follow searches in $S_{m(s)}$. Observe that for u to appear in the search path from s to t in $S_{m(s)}$ there must be no node v with $u < v \leq t$ whose height in $S_{m(s)}$ is higher than u's. It follows that u can appear in the search path only if it is among the tallest nodes in the interval $[u, t]$, i.e., if $|m(s) \wedge m(u)| \geq |m(s) \wedge m(v)|$ for all v with $u < v \leq t$. Recall that $m(s) \wedge m(v)$ is the common prefix (possibly empty) of $m(s)$ and $m(v)$.

There are $d + 1$ nodes in this interval. By symmetry, if there are k tallest nodes then the probability that u is among them is $\frac{k}{d+1}$. Let T be the random variable representing the set of tallest nodes in the interval. Then:

$$\begin{aligned}\Pr[u \in T] &= \sum_{k=1}^{d+1} \Pr[|T| = k] \frac{k}{d+1} \\ &= \frac{\mathrm{E}[|T|]}{d+1}.\end{aligned}$$

What is the expected size of T? All $d+1$ nodes have height at least 0, and in general each node with height at least k has height at least $k+1$ with independent probability $\frac{1}{2}$. The set T consists of the nodes that are left at the last level before all nodes vanish. It is thus equal to A_τ in the process defined in Lemma 4.23, and we have $\mathrm{E}[|T|] < 2$ and thus $\Pr[u \in T] < \frac{2}{d+1}$. ∎

For comparison, experimental data for the congestion in a skip graph with 131072 nodes, together with the theoretical average predicted by Theorem 4.24, is shown in Figure 4.15.

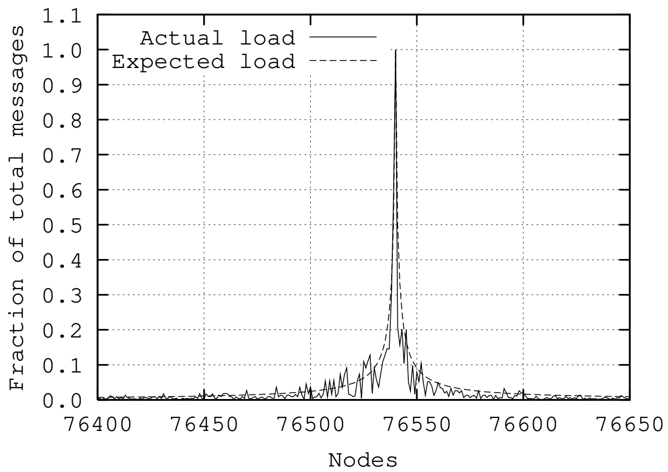


Figure 4.15: Actual and expected congestion in a skip graph with 131072 nodes with the target=76539. Messages were delivered from each node to the target and the actual number of messages through each node was measured. The bound on the expected congestion is computed using Theorem 4.24. Note that this bound may overestimate the actual expected congestion.

4.9.2 Distribution of the average congestion

Theorem 4.24 is of small consolation to some node that draws a probability $\frac{2}{d+1}$ straw and participates in every search. Fortunately, such disasters do not happen often. Define the

average congestion L_{tu} imposed by a search for t on a node u as the probability that an $s-t$ search hits u conditioned on the membership vectors of all nodes in the interval $[u, t]$, where $s < u < t$, or, equivalently, $s > u > t$.[§] Note that since the conditioning does not include the membership vector of s, the definition in effect assumes that $m(s)$ is chosen randomly. This approximates the situation in a fixed skip graph where a particular target t is used for many searches that may hit u, but the sources of these searches are chosen randomly from the other nodes in the graph.

Theorem 4.24 implies that the expected value of L_{tu} is no more than $\frac{2}{d+1}$. In the following theorem, we show the distribution of L_{tu} declines exponentially beyond this point.

Theorem 4.25 *Let S be a skip graph with alphabet $\{0,1\}$. Fix nodes t and u, where $u < t$ and $|\{v : u < v \leq t\}| = d$. Then for any integer $\ell \geq 0$, $\Pr[L_{tu} > 2^{-\ell}] \leq 2e^{-2^{-\ell}d}$.*

Proof: Let $V = \{v : u < v \leq t\}$ and let $m(V) = \{m(v) : v \in V\}$. As in the proof of Theorem 4.24, we will use the fact that u is on the path from s to t if and only if u's height in $S_{m(s)}$ is not exceeded by the height of any node v in V.

To simplify the notation, let us assume without loss of generality that $m(u)$ is the all-zero vector. Then the height of u in $S_{m(s)}$ is equal to the length of the initial prefix of zeroes in $m(s)$, and u has height at least ℓ with probability $2^{-\ell}$. Whether this is enough to raise it to the level of the tallest nodes in V will depend on what membership vectors appear in $m(V)$.

Let $m(s) = 0^i1\cdots$. Then u has height exactly i, and is hit by an $s-t$ search unless there is some $v \in V$ with $m(v) = 0^i1\cdots$. We will argue that when $d = |V|$ is sufficiently large, then there is a high probability that all initial prefixes 0^i1 appear in $m(V)$ for $i < \ell$. In this case, u can only appear in the $s-t$ path if its height is at least ℓ, which occurs with probability only $2^{-\ell}$. So if 0^i1 appears as a prefix of some $m(v)$ for all $i < \ell$, then $L_{tu} \leq 2^{-\ell}$. Conversely, if $L_{tu} > 2^{-\ell}$, then 0^i1 does *not* appear as a prefix of some $m(v)$ for some $i < \ell$.

Now let us calculate the probability that not all such prefixes appear in $m(V)$. We are going to show that this probability is at most $2e^{-2^{-\ell}d}$, and so we need to consider only the

[§]It is immediate from the proof of Theorem 4.24 that L_{tu} does not depend on the choice of s.

case where $e^{-2^{-\ell}d} \le \frac{1}{2}$; this bound is used in steps (4.2) and (4.3) below. We have:

$$
\begin{aligned}
\Pr[L_{tu} > 2^{-\ell}] &\le \Pr[\neg\left(\forall i < \ell : \exists v \in V : 0^i 1 \preceq m(v)\right)] \\
&= \Pr[\exists i < \ell : \forall v \in V : 0^i 1 \not\preceq m(v)] \\
&\le \sum_{i=0}^{\ell-1} \Pr[\forall v \in V : 0^i 1 \not\preceq m(v)] \\
&= \sum_{i=0}^{\ell-1} \left(1 - 2^{-i-1}\right)^d \\
&\le \sum_{i=0}^{\ell-1} e^{-2^{-i-1}d} \\
&= \sum_{j=0}^{\ell-1} e^{-2^{-\ell+j}d} \\
&= \sum_{j=0}^{\ell-1} \left(e^{-2^{-\ell}d}\right)^{2^j} \\
&\le \sum_{j=0}^{\ell-1} \left(e^{-2^{-\ell}d}\right)^{j+1} && (4.2) \\
&\le \sum_{j=0}^{\infty} \left(e^{-2^{-\ell}d}\right)^{j+1} \\
&= \frac{e^{-2^{-\ell}d}}{1 - e^{-2^{-\ell}d}} \\
&\le 2e^{-2^{-\ell}d}. && (4.3)
\end{aligned}
$$

∎

4.10 Conclusions and future work

We have defined a new data structure, the skip graph, for distributed data stores that has several desirable properties. Constructing, inserting new nodes into, and searching in a skip graph can be done using simple and straightforward algorithms. Skip graphs are highly resilient, tolerating a large fraction of failed nodes without losing connectivity. Using the repair mechanism, disruptions to the data structure can be repaired in the absence of additional faults. Skip graphs also support range queries which allows, for example, searching for a copy of a resource near a particular location by using the location as low-order field in the key and clustering of nodes with similar keys.

As explained in Section 4.4, one issue that remains to be addressed is the large number of pointers per machine in the system. It would be interesting to design a peer-to-peer system that maintains fewer pointers per machine and yet supports spatial locality. Also, skip graphs do not exploit network locality (such as or latency along transmission paths) in location of resources and it would be interesting to study performance benefits in that direction, perhaps by using multi-dimensional skip graphs. As with other overlay networks, it would be interesting to see how the network performs in the presence of Byzantine failures. Finally, it would be useful to develop a more efficient repair mechanism and a self-stabilization mechanism to repair defective skip graphs.

Chapter 5

Conclusions and Open Problems

We conclude in this chapter by summarizing our contributions, comparing our approaches with other contemporary peer-to-peer systems, and discussing some open problems as well as future directions for research.

5.1 Conclusions

The thesis of this research is that efficient, fault-tolerant, decentralized peer-to-peer systems can be built using distributed data structures that consist of individual, unreliable components. We presented two different data structures that efficiently manage resources while providing fault tolerance, maintaining a dynamic network, and supporting complex queries. Although there are still several open problems in the area, we believe that this research will provide new insights which will lead to the development of more sophisticated systems and algorithms.

We compare the performance of our approaches with some other contemporary peer-to-peer systems in Table 5.1. The lookup time for all the DHT systems is logarithmic and the insert time is at most polylogarithmic in the number of machines m. For our abstract DHT model as well as skip graphs, similar bounds apply but in the number of resources n. As long as n is polynomial in m, we get the same asymptotic performance for all the systems. The space requirements for maintaining state vary greatly from system to system. With m machines in the network, Chord, Pastry, and Tapestry require $O(\log m)$ space at each

Scheme	Search Time	Insert Time	Space per machine	Spatial locality	Topology awareness
Chord [112]	$O(\log m)$	$O(\log^2 m)$	$O(\log m)$	No	No
CAN [95] (With $d = \log m$)	$O(dm^{1/d})$ $O(\log m)$	$O(d)$ $O(\log m)$	$O(d)$ $O(\log m)$	No	Partially
Pastry [99] Tapestry [125]	$O(\log m)$	$O(\log m)$	$O(\log m)$	No	Partially
Viceroy [78]	$O(\log m)$	$O(\log m)$	$O(1)$	No	No
Abstract DHT model [4]	$O(\log n)$	$O(\log^2 n)$	$O(\log n)$	No	No
Skip graph [5]	$O(\log n)$	$O(\log n)$	$O(\log n)$	Yes	No

Table 5.1: Comparison of peer-to-peer systems. m is the number of machines in the system whereas n is the number of resources.

machine in the network; Viceroy takes it one step further by reducing this requirement to $O(1)$. CAN requires $O(d)$ space at each machine. In comparison, skip graphs require much more space; with n resources in the systems, each machine requires $O(\log n)$ space for *each* resource that it hosts. We discuss possible directions for reduction in space requirements while retaining spatial locality in Section 5.2.1

The primary differences between the various systems are sensitivity to topology and support for spatial locality. As explained in Chapter 2, both CAN and Pastry/Tapestry are partially sensitive to topology. The topologically-sensitive construction of the CAN overlay network supports proximity routing, but it has the disadvantage of non-uniform population of the coordinate space, and it is not completely self-configuring. Both Pastry and Tapestry exclusively provide the proximity neighbor selection in all the systems developed so far, although they too weigh progress in the identifier space against the benefits in latency or physical distance. These systems can choose the best neighbor as per the proximity metric if the entire m^2 latency/proximity matrix for all the m machines is available (or by using some efficient heuristics in the absence of this matrix). It is unclear whether this is the best approach to optimize proximity or if it can be easily applied to other systems.

To the best of our knowledge, skip graphs are the first data structure that maintain spatial locality. This naturally enhances the impoverished query language provided by

DHTs by supporting complex searches such as range queries and near matches to a key. SkipNet is another system which partially supports spatial locality, that was developed independently by Harvey *et al.*[51]. As the primary goal of SkipNet is to provide path and content locality, it does not concentrate on full-fledged spatial locality.

5.2 Open Problems

Even though peer-to-peer systems were introduced only a few years ago, they are now one of the most popular Internet applications and have become a major source of Internet traffic. Various designs have been proposed to implement such systems but a lot still remains to be done. We end this thesis with a list of open problems and note that it merely meant to be illustrative, not exhaustive. We also note that we focus on issues directly related to the results in this dissertation; this is not intended to imply in any way that related topics such as security, anonymity, and replica management are any less significant.

5.2.1 State-Locality trade-off

In the spectrum of contemporary peer-to-peer systems, DHTs are at one end where they minimize the state requirement at each node, and skip graphs are at the other end with support for spatial locality. Can we design a system that provides support for spatial locality and uniform load balancing without excessive storage space at each node? We outline some possible directions to solve this problem of maintaining locality but reducing the state information at each node.

(i) Range splitting: One way to reduce the state is to build a skip graph by *splitting ranges* of resources and distributing them to machines physically close to one another. For example, if the range of keys is [a-z], then we could assign [a-e] to one machine, [f-i] to another machine, and so on. The links to neighbors at lower levels need not be maintained as they will point to the same machine as the one on which the resource itself is stored. When a particular machine gets overloaded with too many resources in its allocated range, it can spill over excess load onto its neighbors. Splitting ranges over machines that are close together with respect to network locality, can reduce the overhead of shifting resources or their address entailed by this method, and can even help in more efficient network-locality routing. It would be interesting to see if this idea could be further developed, potentially using some of the techniques used in B-tree node splits [13] to maintain load balance.

(ii) Locality-sensitive hashing: The earlier approach maintains the basic skip graph structure and tries to minimize space usage. Another prospective approach would be to retain the structure of a DHT for load balancing, and then add locality to this. We saw one technique used for answering approximate range queries in Chapter 2 by Gupta *et al.*[47], which replaced consistent hashing, typically used in DHTs, with locality-sensitive hashing. This method compromises on both load balancing as well as the ability to answer queries *exactly*. It would be interesting to see if there is a hash function that could give both good load balancing as well as good locality properties.

5.2.2 Richer query language

DHTs only support exact matches to a key to locate resources. We discussed some extensions such as the use of bloom filters and locality-sensitive hashing to improve searches in DHTs in Section 2.6.1. However, these improvements are only marginal and not very effective for complex queries.

Skip graphs represent an important step in improving the search capabilities of a peer-to-peer system. With support for spatial locality, skip graphs can support range queries, near matches to a key, and approximate queries. However, a rich query language should also allow for combinations and correlations between data items. For example, it is possible to find all the paintings by Picasso in a skip graph, but it would not be possible to find the ones that have flowers in them as that information may not be contained in the name of the painting. Can we improve the query language to enable answering more complex queries in a peer-to-peer system?

5.2.3 Efficient repair

We gave a repair mechanism for skip graphs in Section 4.8 that in the worst case, takes time quadratic in the number of nodes of the skip graph. This leads to the following question: Is it possible to develop a more efficient repair mechanism for a skip graph? Even more interesting would be a self-stabilization mechanism that could take a disrupted skip graph from any arbitrary starting configuration and restore it efficiently to its defectless state. A good starting point will be to use the merge sort techniques for parallel pointer machines given by Goodrich *et al.*[45]. The methods described therein cannot directly be utilized for our purposes as they presume the existence of some common storage that can be accessed by all the processes, which is absent in the message-passing environment of a peer-to-peer system. A related interesting question is: Can we design a new system which has the benefits of DHTs and skip graphs, and supports quicker and/or cheaper recoveries from failures?

5.2.4 Topologically-sensitive overlay networks

One of the fundamental challenges in peer-to-peer systems today is to build an overlay network that is sensitive to topology and/or latency. In addition to path locality, topologically-sensitive routing would also be beneficial to serve content from replicas located closest in physical distance. In all the overlay networks, a path consists of a certain number of application-level, not IP-level hops. In current systems, little effort is taken to see that the application level hops are congruent to the IP-level hops. As explained earlier, Pastry [99], Tapestry [125], and CAN [95] partially support topologically-sensitive routing. In other systems, a lookup from a node in Yale could potentially go through some distant node in India before it reaches the destination node in Harvard.

Routing schemes for mobile networks that incorporate geographic distances can be seen is systems like grid location service [72] and greedy perimeter stateless routing [59]. Hildrum *et al.*[52] and Karger *et al.*[58] present some methods to solve the nearest neighbor problem in restricted overlay networks. Can these techniques be suitably applied to incorporate full-fledged network locality in peer-to-peer systems? Is it sufficient to use proximity routing and nearest neighbor searches to achieve the benefits of *global* routing sensitive to network locality?

One approach is to generalize our abstract model of a DHT to multiple dimensions. One of the dimensions could be used to represent the actual, physical locations of the machines in the system. Ideally, we would like large jumps in the dimension that represents the keyspace and smaller strides in the dimension for network distances. Another approach would be to use Bartal's probabilistic approximation to the overlay metric space using hierarchically separated trees [11], and obtain an approximate solution for locating nearest neighbors. We could possibly also use multi-dimensional skip graphs to achieve the same purpose.

5.2.5 Handling failures

Our data structures are tolerant to crash failures; random failures for the abstract DHT model, and both random as well as adversarial failures for skip graphs. However, we are yet to study the performance of these networks in the presence of byzantine failures, where nodes can behave arbitrarily. For example, a byzantine node can send different messages to different processes even if it is supposed to send the same message to all processes. If the message is improperly formatted, then the recipient can detect that the sender is faulty but difficulties arise when the message is plausible to the recipient but incorrect. A faulty process can also mimic the behavior of a crashed process by sending no messages beyond a certain point. Typical techniques for dealing with such failures would include redundancy and cryptographic signing to detect and repair broken nodes [77].

A recent survey by Feigenbaum *et al.*[34] shows that there is interest in designing more realistic failure models for peer-to-peer systems where the nodes could collude with one another and deviate from the established protocol for their own selfish interests. The idea is to build computationally-feasible systems that allow nodes to behave *rationally*, while behaving collectively to maximize the common welfare of the system as a whole. This model has been studied for other problems in the field of Algorithmic Mechanism Design [88] which focuses on combining aspects of computational feasibility and game theory such as incentive compatibility. More recently, there has been interest is using this model for distributed systems in the area of Distributed Algorithmic Mechanism Design (DAMD) [33, 34].

One such architecture for fair sharing of storage resources (allowing a node to consume only as much storage as it provides), which is robust against collusions of nodes is presented by Ngan *et al.*[87]. MojoNation [81], which provides a distributed RAID network, also tries to ensure fair sharing of resources by introducing an intermediate form of currency between peers that is called "mojo". Cooper *et al.*[23, 22] give techniques, such as deed trading, advertising, and bidding for trading storage space between two nodes in the system. A list of interesting open questions related to peer-to-peer systems with selfish participants can be found in [108].

5.2.6 New designs

Detailed measurement analysis [104, 103, 71] has been done on peer-to-peer traffic in commercial systems such as Napster [85], Gnutella [43], and KaZaA [61]. These studies reveal that the hosts that participate in these networks are highly heterogeneous with respect to factors such as latencies, lifetimes and shared data. Further, the peers are not always willing to cooperate, and do not behave equally in contributing and consuming resources. Contemporary designs do not take this heterogeneity into account and attempt to distribute the load of maintaining the network as uniformly as possible among all the participants. It would be interesting to see if new designs can be developed that take advantage of this heterogeneity to built a more efficient peer-to-peer system.

Peer-to-peer systems have come a long way in a short span of time since the early days of Napster. Several elegant approaches have been proposed and implemented for the design of such systems, each of which have their own benefits and limitations. We believe that it would be instructional to combine the insights of these different approaches to develop more sophisticated systems in the future.

Bibliography

[1] FIPS 180-1. Secure Hash Standard. In *U. S. Department of Commerce/NIST, National Technical Information Service*, Springfield, VA, USA, April 1995. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`.

[2] Sameer Ajmani, Dwaine E. Clarke, Chuang-Hue Moh, and Steven Richman. ConChord: Cooperative SDSI Certificate Storage and Name Resolution. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 141–154, Cambridge, MA, USA, March 2002.

[3] Akamai. `http://www.akamai.com`.

[4] James Aspnes, Zoë Diamadi, and Gauri Shah. Fault-tolerant Routing in Peer-to-peer Systems. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC)*, pages 223–232, Monterey, CA, USA, July 2002. Submitted to *Distributed Computing*. Available at `http://arXiv.org/abs/cs/0302022`.

[5] James Aspnes and Gauri Shah. Skip Graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 384–393, Baltimore, MD, USA, January 2003. Submitted to a special issue of *Journal of Algorithms* dedicated to select papers of SODA 2003.

[6] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill Publishing Company, Cambridge, UK, 1998.

[7] AudioGalaxy. `http://www.audiogalaxy.com`.

[8] Yonatan Aumann and Michael A. Bender. Fault Tolerant Data Structures. In *Proceedings of the Thirty-Seventh Annual Symposium on Foundations of Computer Science (FOCS)*, pages 580–589, Burlington, VT, USA, October 1996.

[9] Baruch Awerbuch and Christian Scheideler. Peer-to-peer Systems for Prefix Search. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing (PODC)*, Boston, MA, USA, July 2003. To Appear.

[10] Hari Balakrishnan, Scott Shenker, and Michael Walfish. Semantic-Free Referencing in Linked Distributed Systems. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[11] Yair Bartal. Probabilistic Approximation of Metric Spaces and its Algorithmic Applications. In *Proceedings of the Thirty-Seventh Annual Symposium on Foundations of Computer Science (FOCS)*, pages 184–193, Burlington, VT, USA, October 1996.

[12] Mayank Bawa, Roberto J. Bayardo Jr., Sridhar Rajagopalan, and Eugene J. Shekita. Make it Fresh, Make it Quick - Searching a Network of Personal Webservers. In *Proceedings of the Twelfth International World Wide Web Conference (WWW)*, pages 129–140, Budapest, Hungary, May 2003. To Appear.

[13] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1(3):173–189, 1972.

[14] Bobby Bhattacharjee, Pete Keleher, and Bujor Silaghi. The Design of TerraDir. Technical Report CS-TR-4299, University of Maryland, College Park, College Park, MD, USA, October 2001.

[15] Burton H. Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[16] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 34–43, Santa Clara, CA, USA, June 2000.

[17] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[18] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[19] 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 Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, 20(8):1489–1499, October 2002.

[20] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer Overlays. In *Proceedings of the Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, San Francisco, CA, USA, March 2003.

[21] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66, Berkeley, CA, USA, July 2000. `http://www.freenet.sourceforge.net`.

[22] Brian F. Cooper and Hector Garcia-Molina. Bidding for Storage Space in a Peer-to-peer Data Preservation System. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems (ICDCS)*, pages 372–381, Vienna, Austria, July 2002.

[23] Brian F. Cooper and Hector Garcia-Molina. Peer-to-Peer Resource Trading in a Reliable Distributed System. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 319–327, Cambridge, MA, USA, March 2002.

[24] Russ Cox, Athicha Muthitacharoen, and Robert T. Morris. Serving DNS using a Peer-to-peer Lookup Service. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 155–165, Cambridge, MA, USA, March 2002.

[25] Arturo Crespo and Hector Garcia-Molina. Routing Indices for Peer-to-peer Systems. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems (ICDCS)*, pages 23–32, Vienna, Austria, July 2002.

[26] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, December 2001.

[27] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[28] Mayur Datar. Butterflies and Peer-to-Peer Networks. In Rolf Möhring and Rajeev Raman, editors, *Proceedings of the Tenth European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 310–322, Rome, Italy, September 2002.

[29] Rene de la Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, volume 15, pages 295–298, Montvale, NJ, USA, 1959.

[30] Luc Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, 2(3):597–609, 1992.

[31] John R. Douceur. The Sybil Attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260, Cambridge, MA, USA, March 2002.

[32] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient Broadcast in Structured P2P Networks. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[33] Joan Feigenbaum, Christos Papadimitriou, and Scott Shenker. Sharing the Cost of Multicast Transmissions. *Journal of Computer and System Sciences*, 63(1):21–41, August 2001.

[34] Joan Feigenbaum and Scott Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Proceedings of the Sixth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, Atlanta, GA, USA, September 2002.

[35] Amos Fiat and Jared Saia. Censorship Resistant Peer-to-Peer Content Addressable Networks. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 94–103, San Francisco, CA, USA, January 2002. Submitted to a special issue of *Journal of Algorithms* dedicated to select papers of SODA 2002.

[36] Edward Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[37] Michael J. Freedman and David Maziéres. Sloppy hashing and self-organizing clusters. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[38] Michael J. Freedman and Radek Vingralek. Efficient Peer-to-Peer Lookup Based on a Distributed Trie. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 66–75, Cambridge, MA, USA, March 2002.

[39] Christine Frey, P. J. Huffstutter, and Dave Wilson. News Web Sites Clogged in Aftermath; Internet: Breakdown highlights weakness of medium not ready to compete with radio, TV. (Sept. 12, 2001). The Los Angeles Times.

[40] Joaquim Gabarró, Conrado Martínez, and Xavier Messeguer. A Top-Down Design of a Parallel Dictionary using Skip Lists. *Theoretical Computer Science*, 158(1–2):1–33, May 1996.

[41] Joaquim Gabarró and Xavier Messeguer. A Unified Approach to Concurrent and Parallel Algorithms on Balanced Data Structures. In *Proceedings of the Seventeenth International Conference of the Chilean Computer Society (SCCC)*, pages 78–92, Valparáiso, Chile, November 1997.

[42] Prasanna Ganesan, Qixiang Sun, and Hector Garcia-Molina. YAPPERS: A Peer-to-Peer Lookup Service over Arbitrary Topology. In *Proceedings of the Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, San Francisco, CA, USA, March 2003.

[43] Gnutella. `http://gnutella.wego.com`.

[44] Gnutella protocol, Version 0.4.
`http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf`.

[45] Michael T. Goodrich and S. Rao Kosaraju. Sorting on a Parallel Pointer Machine with Applications to Set Expression Evaluation. *Journal of the ACM*, 43(2):331–361, March 1996.

[46] Google. `http://www.google.com`.

[47] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate Range Selection Queries In Peer-to-Peer Systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 141–151, Asilomar, CA, USA, January 2003.

[48] Indranil Gupta, Kenneth Birman, Prakash Linga, Al Demers, and Robbert Van Renesse. Kelips*: building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[49] Steven Hand and Timothy Roscoe. Mnemosyne: Peer-to-peer Steganographic Storage. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 130–140, Cambridge, MA, USA, March 2002.

[50] 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 Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 242–250, Cambridge, MA, USA, March 2002.

[51] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 113–126, Seattle, WA, USA, March 2003.

[52] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed Object Location in a Dynamic Network. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 41–52, Winnipeg, Manitoba, Canada, August 2002.

[53] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-Preserving Hashing in Multidimensional Spaces. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC)*, pages 618–625, El Paso, TX, USA, May 1997.

[54] Sitaram Iyer, Antony Rowstron, and Peter Druschel. SQUIRREL: A decentralized, peer-to-peer web cache. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC)*, pages 213–222, Monterey, CA, USA, July 2002.

[55] Roberto J. Bayardo Jr., Rakesh Agrawal, Daniel Gruhl, and Amit Somani. YouServ: A Web-Hosting and Content Sharing Tool for the Masses. In *Proceedings of the Eleventh International World Wide Web Conference (WWW)*, pages 345–354, Honolulu, HI, USA, May 2002.

[56] M. Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-optimal Distributed Hash Table. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[57] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth ACM Symposium on Theory of Computing (STOC)*, pages 654–663, El Paso, TX, USA, May 1997.

[58] David Karger and Matthias Ruhl. Finding Nearest Neighbors in Growth-restricted Metrics. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 741–750, Montreal, Canada, May 2002.

[59] Brad Karp and H. T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MOBICOM)*, pages 243–254, Boston, MA, USA, August 2000.

[60] Richard M. Karp, Eli Upfal, and Avi Wigderson. The Complexity of Parallel Search. *Journal of Computer and System Sciences*, 36(2):225–253, April 1988.

[61] KaZaA. `http://www.kazaa.com`.

[62] Pete Keleher, Bobby Bhattacharjee, and Bujor Silaghi. Are Virtualized Overlay Networks Too Much of a Good Thing? In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 225–231, Cambridge, MA, USA, March 2002.

[63] David Kempe, Jon M. Kleinberg, and Alan J. Demers. Spatial Gossip and Resource Location Protocols. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 163–172, Crete, Greece, July 2001.

[64] Peter Kirschenhofer, Conrado Martínez, and Helmut Prodinger. Analysis of an Optimized Search Algorithm for Skip Lists. *Theoretical Computer Science*, 144(1–2):119–220, June 1995.

[65] Peter Kirschenhofer and Helmut Prodinger. The Path Length of Random Skip Lists. *Acta Informatica*, 31(8):775–792, 1994.

[66] Jon Kleinberg. The Small-World Phenomenon:An Algorithmic Perspective. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pages 163–170, Portland, OR, USA, May 2000.

[67] Jon Kleinberg. Small-World Phenomena and the Dynamics of Information. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 431–438, Cambridge, MA, USA, December 2001.

[68] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, USA, 1973.

[69] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. *ACM SIGPLAN Notices*, 31(1):190–201, November 2000.

[70] Per-Åke Larson. Dynamic Hash Tables. *Communications of the ACM*, 31(4):446–457, April 1988.

[71] Nathaniel Leibowitz, Aviv Bergman, Roy Ben-Shaul, and Aviv Shavit. Are File Swapping Networks Cacheable? Characterizing P2P Traffic. In *Proceedings of the Seventh International Workshop on Web Content Caching and Distribution*, Boulder, CO, USA, August 2002.

[72] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A Scalable Location Service for Geographic Ad Hoc Routing. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MOBICOM)*, pages 120–130, Boston, MA, USA, August 2000.

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

[74] Nathan Linial and Ori Sasson. Non-Expansive Hashing. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing (STOC)*, pages 509–518, Philadelphia, PA, USA, May 1996.

[75] Witold Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the Sixth International Conference on Very Large Data Bases (VLDB)*, pages 212–223, Montreal, Quebec, Canada, October 1980. Reprinted in *Reading in Database Systems*, M. Stonebraker Ed., 2nd ed., 1995.

[76] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. LH* — a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, December 1996.

[77] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishing, Burlington, MA, USA, 1997.

[78] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC)*, pages 183–192, Monterey, CA, USA, July 2002.

[79] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed Hashing in a Small World. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 127–140, Seattle, WA, USA, March 2003.

[80] Petar Maymounkov and David Maziéres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65, Cambridge, MA, USA, March 2002.

[81] MojoNation. `http://mojonation.net`.

[82] Morpheus. http://www.musiccity.com.

[83] J. Ian Munro and Patricio V. Poblete. Fault Tolerance and Storage Reduction In Binary Search Trees. *Information and Control*, 62(2/3):210–218, August 1984.

[84] Moni Naor and Udi Weider. A Simple Fault-Tolerant Distributed Hash Table. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[85] Napster. http://www.napster.com.

[86] T. S. Eugene Ng and Hui Zhang. Towards Global Network Positioning. In *Proceedings of the First ACM SIGCOMM Workshop on Internet Measurement*, pages 25–29, San Francisco, CA, USA, November 2001.

[87] Tsuen-Wan "Johnny" Ngan, Dan S. Wallach, and Peter Druschel. Enforcing Fair Sharing of Peer-to-Peer Resources. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[88] Noam Nisan and Amir Ronen. Algorithmic Mechanism Design. *Games and Economic Behavior*, 35(1–2):166–196, April 2001.

[89] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The Case for Cooperative Networking. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 178–190, Cambridge, MA, USA, March 2002.

[90] Gopal Pandurangan, Prabhakar Raghavan, and Eli Upfal. Building Low-Diameter P2P Networks. In *Proceedings of the Forty-Second Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 492–499, Las Vegas, NV, USA, October 2001. To appear in *IEEE Journal on Selected Areas in Communications (JSAC)*.

[91] Thomas Papadakis, J. Ian Munro, and Patricio V. Poblete. Analysis of the Expected Search Cost in Skip Lists. In J. R. Gilbert and R. G. Karlsson, editors, *Proceedings of the Second Scandinavian Workshop on Algorithm Theory (SWAT 90)*, volume 447 of *Lecture Notes in Computer Science*, pages 160–172, Bergen, Norway, July 1990.

[92] C. Greg Plaxton, Rajamohan Rajaraman, and Andrea W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. *Theory of Computing Systems*, 32(3):241–280, 1999.

[93] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[94] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Scott Shenker. Load Balancing in Structured P2P Systems. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[95] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 161–172, San Diego, CA, USA, August 2001.

[96] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Proceedings of the Third International Workshop on Networked Group Communication (NGC)*, volume 2233 of *Lecture Notes in Computer Science*, pages 14–29, UCL, London, UK, November 2001.

[97] Jordan Ritter. Why Gnutella Can't Scale. No, Really. `http://www.darkridge.com/~jpr5/doc/gnutella.html`.

[98] Antony Rowstron and Peter Druschel. PAST: A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 75–80, Schoss Elmau, Germany, May 2001.

[99] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[100] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. *ACM SIGOPS Operating Systems Review*, 35(5):188–201, December 2001.

[101] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International Workshop on Networked Group Communication (NGC)*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43, UCL, London, UK, November 2001.

[102] Jared Saia, Amos Fiat, Steven Gribble, Anna Karlin, and Stefan Saroiu. Dynamically Fault-Tolerant Content Addressable Networks. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 270–279, Cambridge, MA, USA, March 2002.

[103] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An Analysis of Internet Content Delivery Systems. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 315–327, Boston, MA, USA, December 2002.

[104] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In Martin G. Kienzle and Prashant J. Shenoy, editors, *Multimedia Computing and Networking (MMCN)*, volume 4673 of *SPIE*, pages 156–170, January 2002.

[105] Nima Sarshar and Vwani Roychowdhury. A Random Structure for Optimum Cache Size Distributed Hash Table (DHT) Peer-to-Peer Design. `http://arxiv.org/abs/cs.NI/0210010`, October 2002.

[106] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, Reading, MA, USA, 1996.

[107] SETI@home. `http://setiathome.ssl.berkeley.edu`.

[108] Jeffrey Shneidman and David C. Parkes. Rationality and Self-Interest in Peer to Peer Networks. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[109] Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Query Routing in the TerraDir Distributed Directory. In Victor Firoiu and Zhi-Li Zhang, editors, *Proceedings of the SPIE ITCOM 2002*, volume 4868 of *SPIE*, pages 299–309, Boston, MA, USA, August 2002.

[110] Sridhar Srinivasan and Ellen Zegura. Network Measurement as a Cooperative Enterprise. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 166–177, Cambridge, MA, USA, March 2002.

[111] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-peer Caching Schemes to Address Flash Crowds. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 203–213, Cambridge, MA, USA, March 2002.

[112] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

[113] Marvin Theimer and Michael B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems (ICDCS)*, pages 52–64, Vienna, Austria, July 2002.

[114] ThreeDegrees. `http://www.threedegrees.com`.

[115] Paul F. Tsuchiya. The Landmark Hierarchy: A New Hierarchy for Routing in Very Large Networks. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 35–42, Stanford, CA, USA, August 1988.

[116] Leslie Valiant. A Scheme for Fast Parallel Communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[117] Marc Waldman and David Maziéres. Tangler: A Censorship-Resistant Publishing System Based on Document Entanglements. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 126–135, Philadelphia, PA, USA, November 2001.

[118] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A Robust, Tamper-Evident, Censorship-Resistant Web Publishing System. In *Proceedings of the Ninth USENIX Security Symposium*, pages 59–72, Berkeley, CA, USA, August 2000.

[119] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishing, Burlington, MA, USA, May 1999.

[120] Yahoo. `http://www.yahoo.com`.

[121] Beverley Yang and Hector Garcia-Molina. Improving Search in Peer-to-Peer Systems. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems (ICDCS)*, pages 5–14, Vienna, Austria, July 2002.

[122] Beverly Yang and Hector Garcia-Molina. Designing a Super-peer Network. In *Proceedings of the Nineteenth International Conference on Data Engineering (ICDE)*, pages 49–60, Bangalore, India, March 2003.

[123] Hui Zhang, Ashish Goel, and Ramesh Govindan. Using the Small-World Model to Improve Freenet Performance. In *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM) (Vol. 3)*, pages 1228–1237, New York, NY, USA, June 2002.

[124] Zheng Zhang, Shuming Shi, and Jing Zhu. SOMO: self-organized metadata overlay for resource management in P2P DHT. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, USA, February 2003.

[125] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, Berkeley, CA, USA, April 2001.