A Scalable Content-Addressable Network

by

Sylvia Paul Ratnasamy

M.S. (University of California at Berkeley) 1999
B.E. (University of Poona, India) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Dr. Scott Shenker, Co-Chair
Professor Ion Stoica, Co-Chair
Professor John Chuang

Fall 2002

The dissertation of Sylvia Paul Ratnasamy is approved:

Co-Chair Date

Co-Chair Date

Date

University of California at Berkeley

Fall 2002

A Scalable Content-Addressable Network

by

Sylvia Paul Ratnasamy

Abstract

A Scalable Content-Addressable Network

by

Sylvia Paul Ratnasamy

Doctor of Philosophy in Computer Science

University of California at Berkeley

Dr. Scott Shenker, Co-Chair

Prof. Ion Stoica, Co-Chair

In May 1999, Shawn Fanning, then a freshman at Northeastern University, launched the first "peer-to-peer" or P2P file-sharing application – Napster. Napster allowed individual end-users (called peers) to share the MP3-encoded music stored on their local computers directly with one another over the Internet. Within a year, Napster had grown to a user population of over 50 Million users making it the fastest growing Internet application to date. Three years later, despite the closure of Napster, the phenomenon of file-sharing continues its dramatic growth and appears set to remain an important feature of the Internet for the forseeable future. The sheer scale of these file-sharing applications make them important in their own right. And yet, as this thesis will argue, P2P is much more than just a way to trade MP3s over the Internet. The P2P architecture with its use of low-cost, grass-roots resources and its decentralized nature that does not rely on any form of centrally managed

infrastructure, represents a significant departure from the client-server architecture of the Web. These unique characteristics, we believe, allow P2P systems to support the rapid and low-cost deployment of powerful large-scale applications in a manner that would not be possible with the current architecture of the Web.

There are two key pieces to a P2P system: the lookup mechanism used to locate a desired file and the actual file download. The decentralized storage in P2P systems makes the file transfer process inherently scalable; the hard part is finding the peer(s) from which to retrieve the desired file. This thesis addresses this problem of scalable indexing in P2P systems. *I.e.*, given a file identifier, how can we find the IP address of the peer(s) holding the file? Ideally, a solution to this indexing problem must be scalable to millions of users, must find files quickly, and must be resilient to the frequent arrival and departure of participant peers. As a solution, we introduce the concept of a Content-Addressable Network (CAN) as a distributed sytem that provides hash table functionality – mapping "keys" onto "values" – on Internet-like scales. Our CAN design is completely distributed (requiring no form of centralized control, coordination or configuration), scalable (nodes maintain only a small amount of control state that is independent of the number of nodes in the system), and fault-tolerant (nodes can route around failures).

The Distributed Hash Table (DHT) functionality supported by CAN serves as a useful substrate for a range of large distributed systems; for example, Internet-scale facilities such as global file systems, application-layer multicast, event notification, and chat services can all be layered over a DHT system such as CAN.

To Kapil, my husband and best friend

Contents

List of Figures

List of Tables

Acknowledgements

I leave ICSI and school with a stash of memories that I shall always treasure — of Scott reading a delightful poem he wrote to celebrate my wedding, of Mark cheerfully scrubbing my tea-stained laptop under a running tap. Of Steve going over my prelim study list with me, insisting all the while that it is indeed possible to pass a systems prelim without knowing what a multi-threaded program is. Of weekly ICSI meetings that were a treat as much for the sense of humor of those gathered as for their sharp technical comments.

My days as a graduate student have been wonderful. One person, above all, is directly responsible for that – my advisor Scott Shenker. My first instinct in writing this was to say that Scott has been so much more than just an advisor. On second thought, I realise that an “advisor” is exactly what Scott has been – an incredible one – fulfilling each of the many roles required of an advisor to perfection. With his characteristic humor, patience and sheer brilliance, Scott has been teacher, mentor, colleague and friend all rolled into one. Above all, Scott has been, very simply, a *nice* person – equally willing to listen to me whine about a toothache as about some research problem and always giving me complete freedom to deal with my work in the style I was most comfortable with.

Steve McCanne was my advisor during my first two years at Berkeley. Steve’s encouragement meant the world to me at the time and was the main reason I decided to stay on at Berkeley for a Ph.d. (I had joined fully convinced that I was going to leave with a Masters). I shall always be indebted to him for setting me off to a good start down the research track.

From when he first arrived at Berkeley, Ion Stoica has always ungrudgingly taken the time to discuss my research, my thesis, my career plans. He invariably gives me frank and very sound advice for which I am grateful. I've much enjoyed our (sometimes heated) discussions and look forward to continuing them in the years to come.

While working on the CAN problem, some of the most fun discussions I've had started with my sticking my head around Mark Handley's office door. I have troubled Mark with innumerable questions, both high-level ones and very nitty-gritty-detail ones, and he has answered them all with the good sense and humor that is typical of him. I was also lucky to have had the opportunity to work with Dick Karp and learnt a great deal from just watching how he thinks through a problem.

This thesis would not have been possible without the technical help of Paul Francis, Mark Handley, Dick Karp, Scott Shenker and Ion Stoica. This is not one of those theses where the student slaved alone; there is no aspect of the CAN work that did not benefit from the detailed technical input I received from each of them. Paul, together with Steve McCanne, was responsible for first getting us all excited about "P2P", while the core of the CAN algorithm was proposed by Dick Karp.

I am grateful to the members of my quals and thesis committee, John Chuang, Dick Karp, Randy Katz, Scott Shenker and Ion Stoica, for their feedback which helped shape this thesis.

On problems outside of the CAN work, I have had the opportunity to collaborate with several researchers – Yatin Chawathe, Lee Breslau, Deborah Estrin, Sally Floyd, Ramesh Govindan, Brad Karp, Qin Li, Li Yin, Fang Yu – and have greatly enjoyed working with each of them. Randy Katz very generously stepped in to act as my official advisor after Steve McCanne left Berkeley, and was always willing to make the time to meet and discuss my work.

I can imagine no better work environment than that at ICSI and thank the many people – Lila Finhill, Sally Floyd, Atanu Ghosh, Mark Handley, Orion Hodson, Brad Karp, Dick Karp, Eddie Kohler, Vern Paxson, Maria Quintana, Pavlin Radoslavov, Scott Shenker and Diane Starr – that make ICSI the laid-back, cheerful place it is.

I especially thank Lee Breslau who very kindly made an office at AT&T available to me for the two years I was living in Menlo Park and went out of his way to make me feel welcome there.

I'd also like to thank my officemates at school – Gene Cheung, Matt Podolsky, Wison So, Shelley Zhuang, Lakshmi Narayanan and Yan Chen – for making 465 Soda Hall a fun and lively work environment.

Work would not be the same without friends to take my mind off it. With Gabrielle and Sigmund Csicssery, I have enjoyed countless dinners and outings that are always as relaxing as they are interesting. They are warm and affectionate and I am lucky to have them as friends. Sudnya Shroff and Nikhil Jakatdar very kindly took me in when I first arrived at Berkeley and did everything possible to make sure I was not too homesick. Ever since, they have unfailingly been there for me and are friends I shall always cherish. Some of the most enjoyable moments of my day at school have been spent over coffee at Nefeli with Chema Gonzalez talking about everything from the eccentricities of our grandmothers to the finer points of Spanish cuisine. Chema has been a very dear friend, putting up with my pig-headedness when working on projects and always reminding me to not take work too seriously.

My parents-in-law, Nalinee and Vishwas, have been great; always supportive and genuinely interested in my work. My sister Chandra is hilarious. Her determined interest in the length of my hair and complete disinterest in my research is always refreshing and much appreciated. To my parents, I owe more than I could ever put into words. They have encouraged me at every step along the way and yet, have never expected, or asked for, anything from me, always trusting me to make my own decisions.

For eight years now, Kapil is the one I take my thoughts to at the end of each day. Incredibly enough, he still listens patiently and responds with the honesty and good sense I have come to rely on. This thesis might have been possible without Kapil, it just wouldn't have been worth anything.

Chapter 1

Introduction

Over the last decade, the Internet has grown to support a veritable cornucopia of applications; rich in variety as they are in scale. The latest addition to these has been the Peer-to-Peer or "P2P" file sharing applications that allow arbitrary end users, called peers, to share files with one another over the Internet. Despite being only a few years old, P2P file-sharing today accounts for a large fraction of Internet traffic, frequently overtaking even the Web in this respect. For example, an October 2001 trace of the traffic at multiple border routers within a large ISP's backbone [42], revealed that a total volume of approximately 1.2 Tera-Bytes/day could be attributed to just three of the more popular P2P systems. Yet another recent report [15] estimates that between 400,000 and 600,000 films are swapped daily over the Internet using systems such as Gnutella, FastTrack, and IRC, with the trading populations of these sytems soaring to 9 million simultaneous users soon after the release of two highly promoted movies (Spider-Man and Star Wars).

This thesis describes the design of a Content-Addressable Network – a distributed indexing system for P2P systems. Section 1.1 provides a brief overview of the emergence and growth of P2P systems on the Internet. We motivate and define the problem of locating content in P2P systems in Sections 1.2 and 1.3 and provide a brief overview of Content-Addressable Networks – our solution to this problem – in Section 1.4. Section 1.5 provides a roadmap to the remainder of this thesis.

1.1 P2P on the Internet

The Peer-to-Peer phenomenon started in 1999 with the launch of Napster, a service that allowed users to "share" audio files (MP3s). In Napster, individual end users, or peers, stored their collection of MP3 files on local disk while Napster ran a central server storing only the index of files available within the user community. To retrieve a desired song, users issued keyword-based search requests to this central server and obtained the IP address of peers storing matching files. The user could then download the desired file directly from one of these peers. The idea behind Napster was, in retrospect, a very simple one – the decentralized storage of files at end users (rather than servers) together with a central index to locate files. This simple idea proved enormously successful. Within a year, 50 Million users had downloaded the Napster software making it the fastest growing Internet application to date. Following Napster's dramatic growth, similar systems such as Gnutella [7], FreeNet [6], Jungle Monkey [31], MojoNation [30] and others emerged in rapid succession. These systems retained Napster's model of decentralized storage but differed in the lookup mechanism used to locate files. Instead of Napster's centralized lookup mechanism, these systems used decentralized search techniques where a user query is propogated between peers and any peer that has a matching file responds directly to the user. This new generation of file-sharing systems were thus completely decentralized in terms of both file lookup and storage, a change more likely motivated by legal rather than technical reasons since by the time of their introduction, the Recording Industry Association of America (RIAA) had initiated legal proceedings to shut down Napster, claiming that it violated copyright law by facilitating the widespread sharing of MP3s. Because the indexing server was crucial to the file-sharing, the operators of the Napster server (*i.e.,* Napster, the company) were an obvious target for a legal attack; shutting down the central index provided an easy way to bring down the entire system. To avoid this, completely decentralized systems like Gnutella, were designed such that the no single component entity (whether a server, organization or system operator) is indispensible thereby making these systems harder to control or censor. In March 2001, the RIAA won its lawsuit and Napster was shut down. The end of Napster did not however mean the end of P2P file-sharing. On the contrary, the decentralized P2P systems pioneered by Gnutella continued, and still continue, to thrive; new decentralized systems such as KaZaA and Morpheus have emerged and the P2P user population continues to grow. For example, as of July 2002, the KaZaA software has been downloaded by over 100 Million users with an estimated 1.5 Million simultaneous users.[1] Moreover, these systems have evolved to share a variety of content types from audio and video files to software and more. By every indication, P2P systems have grown into a major Internet application. While the deployment of these systems raises a host of interesting legal, social and network policy questions [43], from a technical

[1] Statistics on the number of software downloads for the various file-sharing applications are available at `download.com`; estimates for the number of simultaneous users are available at `zeropaid.com`.

perspective, the (arguably) most remarkable aspect of P2P filesharing has been its *rapid* and *wide-spread* deployment. To what can this successful deployment be attributed? Clearly, the nature of the content being shared is a large contributor. However, the free access to popular copyright material primarily provides a huge user population. That these systems withstood the challenges that accompany such a large user population suggests that there are technical advantages to the architecture adopted by these P2P systems. To expose these advantages, we consider the question of whether file-sharing could have been built using the existing technology of the Web? *I.e.* was the P2P architecture, a new solution to distributing content, really required? We discuss this in the following section.

1.2 The P2P architecture

Since the early 1990s, the Web has served as an effective medium for publishing and accessing content over the Internet. The Web uses a *client-server* architecture – Web documents are stored at servers; to download a particular document, an end-user or client locates the appropriate server, sends it a request for the document and obtains a response (very likely the document itself). Typically, a document is stored at a relatively small number (frequently one) of servers and there are far fewer Web servers than clients. Also, servers tend to be "online", *i.e.,* accessible over the Internet, for relatively long periods of time (on the order of days) while client tend to be more transient (for example, the online duration of dial-up clients may be on the order of minutes). Web documents are uniquely named using Uniform Resource Locators (URLs), for example – `http://www.cs.berkeley.edu/example.html` . URLs consist of three parts: the protocol for communicating with the server (*e.g.,* `http`), the hostname of the server (*e.g.,* `www.cs.berkeley.edu`), and the name of the document at that server (*e.g.,* `example.html`). Given the URL of a desired document, a client uses the Domain Name System (DNS) to "lookup" the IP address of the server corresponding to the hostname embedded within the URL, *i.e.,* the DNS translates from `www.cs.berkeley.edu` to IP address `132.23.45.6`, after which the client can communicate with the server. The DNS is the distributed database used to translate Internet hostnames to IP addresses and vice versa. The DNS infrastructure consists of name servers organized in a hierarchy that reflects the hierarchical structure of hostnames. A hostname is resolved by recursing down the hierarchy to a name server responsible for that hostname; for example, `www.berkeley.edu` would be resolved by first contacting a root name server to learn the address of the name server responsible for `.edu`, which in turn yields the address of the name server for `.berkeley.edu` from which we finally learn the IP address of the machine `www.berkeley.edu`.

For over a decade now, the Web has served to deliver content to many millions of users. In the face of this successful record, an obvious question is whether the architecture of the Web might have served file-sharing applications equally well? We argue not. The architecture of P2P systems is quite different from that of the Web. In P2P systems, end-users (peers) act as both client and server; *i.e.,* peers retrieve, store and serve content. Unlike Web servers, peers are highly transient, joining and leaving the P2P system on short timescales, sometimes even on the order of minutes. Finally, unlike the Web's use of URLs, P2P has no single, well-defined naming scheme used to uniqely name content. Users are free to name the files stored on their local machines as desired. Each architecture lends itself to quite different styles of content distribution:

Compared to the Web, the P2P architecture allows content to be published *easily* and *quickly* thereby extending the ability to publish content to individual end-users On the Web, a first-time publisher must obtain a unique portion of the DNS namespace and set up a name server to resolve this new namespace – neither of these is a simple task. Furthermore, propogating information about the addition or deletion of domain names through the DNS system is a relatively slow process requiring manual configuration. Even updating a mapping from domain name to IP address requires manual configuration. For these reasons, managing a domain name and consequently, controlling the publishing of content, is a non-trivial process not undertaken by typical individual end-users. Hence, although well-suited to larger publishers such as `cnn.com`, the Web's use of the DNS system makes it hard to accomodate the large numbers of transient peers that comprise the P2P publisher population. In P2P systems by contrast, publishing content is merely a matter of copying the desired files to a designated directory on the publisher's local machine.

Publishers/end-users in P2P systems have direct control over making their content available to search algorithms thus allowing newly published content to be rapidly made available to keyword-based searches The primary mode of searching for content, both on the Web and in P2P systems, is via keyword-based searching. Keyword-based search on the Web is effected through centralized search engines such as Google, HotBot and Yahoo, that build massive centralized indices over content discovered by "crawling" – recursively following links from one document to another – the Web. The functioning of these search engines is a process that is entirely independent from that of publishing content and publishers have little control over when their content is indexed, if at all. Thus, although a newly created Web document is immediately available to an end-user who knows the exact URL, the same document is unavailable to a keyword-based search until such time as the search engine's crawler finds its way to the new document; an event over whose occurence and timing, the publisher has no real control.[2] Such solutions are appropriate when content remains in the system for relatively long periods of time (as is the case with the Web) but are ill-suited to systems within which content is frequently being added and removed. In P2P architectures on the other hand, once a peer node joins the system, the files it stores are immediately available to both searches and downloads.[3] This ability for publishers to directly control the availability of their content makes the P2P architecture well-suited to searching in environments where content is published and withdrawn on relatively short timescales by a large number of peers.

Unlike the Web, P2P systems rely entirely on the participation of end-user machines and not on any deployed infrastructure On the Web, storage and bandwidth resources are typically provided by the long term deployment of high performance server machines with high bandwidth connectivity. In addition, the functioning of the Web requires the DNS infrastructure. This reliance on deployed infrastructure makes the performance of the Web largely independent of user behavior and reasonably reliable and consistent in terms of the availability and delivery of content. On the flip side, this infrastructure needs

[2] An exception to this is the "sponsored links" service offered recently by search engines such as Google whereby content providers can pay to have their sites indexed.

[3] This is true even with systems like Napster, that use centralized search techniques; end-user explicitly register the availability of their content with the central indexing server.

to be scaled to meet growing user populations – a process that takes some amount of time and money. P2P systems on the other hand, make no use of deployed infrastucture; the resources available to users are the sum total of the resources available at individual user machines. This allows P2P systems to harness a potentially huge amount of resources without requiring centralized planning or investments in hardware, rack space, or bandwidth. Further, the net system resources available scales naturally and instantaneously with the number of users in the system. The downside is that P2P systems are at the mercy of their user population in terms of both system performance and the extent and variety of available content. Also, for better or for worse, the distinction over the use of deployed infrastructure makes it hard to control or censor the usage of P2P systems.

For the above reasons, we believe that the phenomenal deployment of P2P systems would have been much harder to achieve using an architecture similar to that of the Web. The unique appeal of the P2P architecture, we believe, lies in its potential to enable the rapid and low-cost deployment of powerful, large-scale applications.

Of course, the promise of the P2P architecture can only be realized if these P2P systems are scalable. In the following section, we outline some of the challenges to building scalable P2P systems.

1.3 The P2P lookup problem

There are two key pieces to a P2P system: the lookup mechanism used to locate a desired file and the actual file download. The decentralized storage (and hence decentralized downloads) in P2P systems makes the file transfer process inherently scalable; the hard part is finding the peer(s) from which to retrieve the file. Lookup solutions in deployed systems to date fall into two categories – centralized like Napster and decentralized like Gnutella, KaZaA, FreeNet and others. Centralized solutions are typically cited as being vulnerable due to a single point of failure and being hard to scale to many millions of users. Yet, the deployment of services such as Google and Yahoo have demonstrated that it is possible to engineer a centralized service to scale to very large numbers while maintaining reliability. This however typically requires significant investment in high-end servers, high-bandwidth connectivity, rack space and so forth which implies that a centralized solution is a viable option provided there is an economic incentive or a business model that justifies the expense. It is not clear that such incentives exist for currently deployed P2P applications. Furthermore, strictly requiring a business model of P2P applications might restrict the range of future applications that might be well served by P2P architectures. An additional drawback of centralized solution is that, being easy to shut down, they are vulnerable to censorship and legal attacks as was evinced by the termination of Napster. These financial and legal issues, rather than any fundamental technical issues, led to the adoption of decentralized solutions. Typically, these solutions involve having peer nodes self-organize into an overlay network over which search requests are forwarded. Unfortunately, currently deployed systems have significant scaling problems; for example, in Gnutella, searches are flooded with a certain scope; flooding on every request is clearly not scalable[16] and, because the flooding has to be curtailed at some point, may fail to find the desired file in large systems. FreeNet, another decentralized system, uses a random-walk-based search algorithm that can fail to find files even when they are actually in the system.

Motivated by the scaling problems of deployed P2P systems, we started our investigation with the question: could one devise a scalable solution to the problem of locating files in P2P networks? *I.e.*, given a file identifier, how can we find the IP address of the node(s) holding the file.[4] Our target goals for a solution to this indexing problem were

- scalability: any indexing algorithm for P2P environments must be designed to scale to several million nodes.
- efficiency: files should be located reasonably quickly and with low overhead in terms of the message traffic generated.
- dynamicity: the indexing system should be robust to frequent node arrivals and departures in order to cope with the highly transient user populations characteristic to P2P environments.
- balanced load: in keeping with the decentralized nature of P2P systems, the total indexing load should be roughly balanced across all the nodes in the system.

This thesis proposes a distributed indexing system, which we call a Content-Addressable Network (CAN), as a solution to the above indexing problem.

1.4 Content-Addressable Networks

CAN is a distributed sytem that provides hash table functionality – mapping "keys" onto "values" – on Internet-like scales. In parallel with our work, several research

[4] Notice that the above problem description supports locating files using exact-match lookups rather than the keyword-based approximate-match lookups used by current P2P systems. The intuition behind this is that support for keyword-based search can be layered on top of an exact-match lookup service. A number of ongoing research projects are currently exploring this problem.

projects have proposed systems that support similar hash table functionality; among them are Tapestry [50], Pastry [39] and Chord [47]. We will use the term Distributed Hash Table or DHT to refer to the above systems (including CAN) collectively.

In all these DHT systems, files are associated with a key (produced, for instance, by hashing the file name) and each node in the system is responsible for storing a certain range of keys. There is one basic operation in these DHT systems, `lookup(key)`, which returns the identity (*e.g.*, the IP address) of the node storing the object with that key. This operation allows nodes to `put` and `get` files based on their key, thereby supporting the hash-table-like interface.[5]

Although our work on CAN was initially motivated by the P2P file-sharing systems, we soon realized that the utility of CAN, and DHTs in general, is not limited to peer-to-peer systems. In fact, the DHT functionality in general, is already proving to be a useful substrate for large distributed systems; a number of projects are proposing to build Internet-scale facilities layered above DHTs, including distributed file systems [11, 25, 9], application-layer multicast [36, 51], event notification services [3, 40], and chat services [2]. With so many applications being developed in so short a time, we expect the DHT functionality to become an integral part of the future P2P landscape.[6]

The core of these DHT systems is a name-based routing algorithm. The DHT nodes form an overlay network with each node having several other nodes as neighbors. When a `lookup(key)` is issued, the lookup is routed through the overlay network to the

[5] The interfaces of these systems are not all identical; some reveal only the `put` and `get` interface while others reveal the `lookup(key)` function directly. However, our discussion refers to the underlying functionality and not the details of the API.

[6] While we believe that CANs, and the DHTs in general, will find many applications, our work focusses not on the use of CANs but on their design.

node responsible for that key. Each of the proposed DHT systems – Tapestry, Pastry, Chord, and CAN – employ a different routing algorithm. This thesis describes the CAN routing algorithm that provides the DHT functionality while meeting the previously enumerated design goals of scalability, efficiency, dynamicity and balanced load. In Chapter 5, we discuss the Chord, Pastry and Tapestry routing algorithms.

As we have said, CANs resemble a hash table; the basic operations performed on a CAN are the insertion, lookup and deletion of (key, value) pairs. In our design, the CAN is composed of many individual nodes. Each CAN node stores a chunk (called a *zone*) of the entire hash table. In addition, a node holds information about a small number of "adjacent" zones in the table. Requests (insert, lookup, or delete) for a particular key are routed by intermediate CAN nodes towards the CAN node whose zone contains that key. Our CAN design is completely distributed (requiring no form of centralized control, coordination or configuration), scalable (nodes maintain only a small amount of control state that is independent of the number of nodes in the system), and fault-tolerant (nodes can route around failures). Unlike systems such as the DNS or IP routing, our design does not impose any form of rigid hierarchical naming structure to achieve scalability. Finally, our design can be implemented entirely at the application level.

1.5 Thesis Organization

The central focus of this thesis is on the design and evaluation of CAN. In addition we explore ways in which CANs might be used and/or extended to meet diverse goals ranging from simply improving CAN performance to providing richer services such as multicast communication. Accordingly, the remainder of this thesis is organized as follows:

Chapter 2: CAN – Design and Evaluation

Chapter 2 presents the detailed design of CAN and its evaluation through simulation. The system design we describe in Section 2.1 focusses on the key operations required to support the above DHT interface in a manner that is at once simple and scalable. *I.e.* we focus not on the potential use of CANs, but on their design.

Chapter 3: Low Latency Routing in CAN

CAN is an overlay network; a message is typically routed through a number of intermediate CAN nodes on its way from source to destination. Because CAN nodes may be geographically dispersed, some of these intermediate hops could involve transcontinental links and others merely trips across a LAN leading to inefficient, high latency paths. Thus, an important performance issue for CAN, and in fact for any overlay network, is to reduce the overhead associated with application-level routing. We describe our approach to this problem in Chapter 3. Our techniques greatly improve the performance of CAN routing and also serve as an example of how one might extend the CAN design to achieve performance improvements.

Chapter 4: M-CAN – Can-based Multicast

Our interest in CANs is based on the belief that a hash table-like abstraction would give Internet system developers a powerful design tool that could enable new applications and communication models. As an example of how one might use CAN to support rich higher-level services, we describe M-CAN, an application-level multicast scheme that builds on CAN. The design and evaluation of M-CAN is covered in Chapter 4.

Chapter 5: Discussion

We conclude with a discussion of ongoing research (by us and others) in the space of both, the underlying algorithms used to support a DHT interface and the applications built on top of a DHT interface. As mentioned earlier, several research groups have (independently) proposed scalable P2P systems that support the same DHT functionality as CAN but use different underlying routing algorithms. While a comprehensive comparison of these different systems is beyond the scope of this work, Section 5 reviews this related work, discusses certain issues relevant to these DHT algorithms and raises some open research questions. Finally, Section 5.1 briefly describes some of the ongoing research work on applications and services built over DHTs.

Chapter 2

A Content-Addressable Network

In this chapter, we describe the design of a Content-Addressable Network (CAN). As stated in the previous chapter, CANs resemble a hash table; the basic operations performed on a CAN are the insertion, lookup and deletion of (key,value) pairs. In our design, the CAN is composed of many individual nodes. Each CAN node stores a chunk (called a

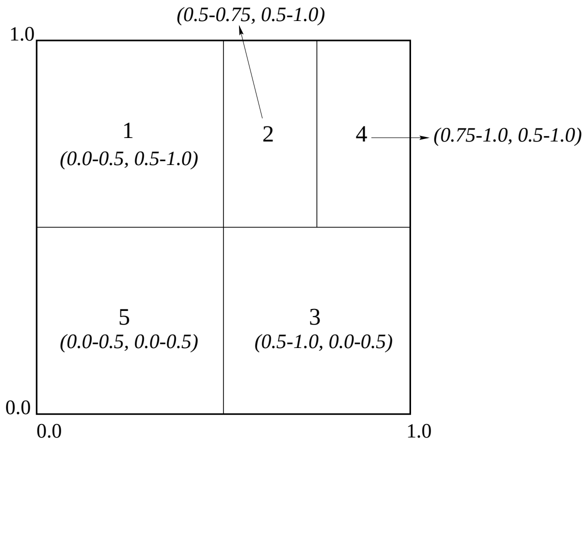


Figure 2.1: *Example 2-d coordinate overlay with 5 nodes*

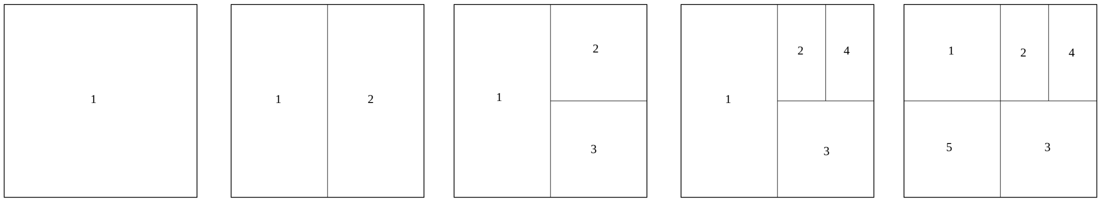


Figure 2.2: *Partitioning of the CAN space as 5 nodes join in succession*

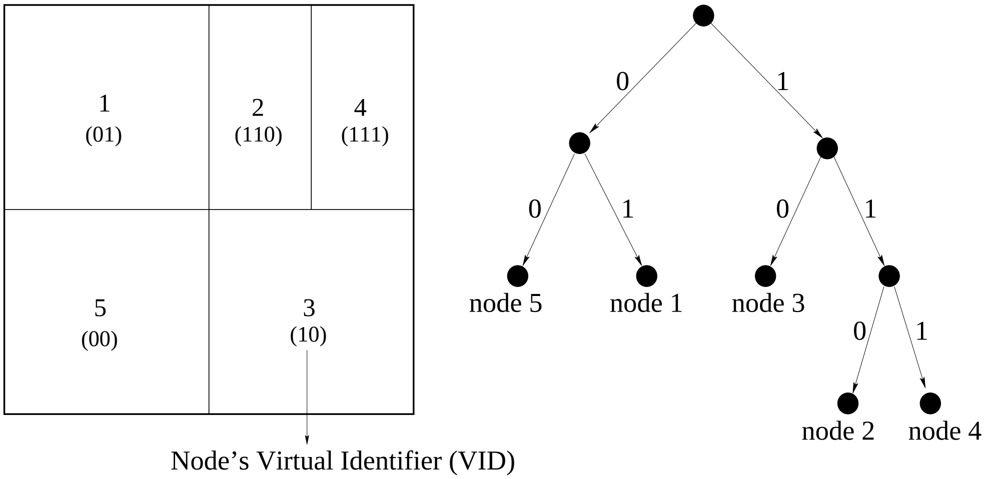


Figure 2.3: *5 node CAN and its corresponding partition tree*

zone) of the entire hash table. Providing this hash-table-like interface then requires every node to support a single operation – given an input *key*, a node must be able to route messages to the node holding *key*. As such, our design primarily addresses the issues related to supporting this name-based routing operation in a manner that is completely distributed (requiring no form of centralized control, coordination or configuration), scalable (nodes maintain only a small amount of control state that is independent of the number of nodes in the system), and robust to node and network failures. Application-specific issues related to how one might use CANs to achieve, for example, some required level of data consistency or availability, while important problems in their own right, are not the focus of this work.

2.1 Design

Our design centers around a virtual d-dimensional Cartesian coordinate space on a d-torus.[1] This coordinate space is completely logical and bears no relation to any physical coordinate system. At any point in time, the *entire* coordinate space is dynamically partitioned among all the nodes in the system such that every node "owns" its individual, distinct zone within the overall space. For example, Figure 2.1 shows the a 2-dimensional $[0, 1] \times [0, 1]$ coordinate space with 5 nodes. Nodes in the CAN self-organize into an overlay network that represents this virtual coordinate space. A node learns and maintains as its set of neighbors the IP addresses of those nodes that hold coordinate zones adjoining its own zone. This set of immediate neighbors serves as a coordinate routing table that enables routing between arbitrary points in the coordinate space.

[1] For simplicity, the illustrations in this paper do not show a torus, so the reader must remember that the coordinate space wraps.

This Cartesian space serves as a level of indirection; one can now talk about storing data at points in the space or routing between points in the space where a point denotes the node that owns the zone within which that point lies. One can thus use the virtual coordinate space to store (key,value) pairs as follows: to store a pair (K,V), key K is deterministically mapped onto a point P in the coordinate space using a uniform hash function. The corresponding key-value pair is then stored at the node that owns the zone within which the point P lies. To retrieve an entry corresponding to key K, any node can apply the same deterministic hash function to map K onto point P and then retrieve the corresponding value from the point P. If the point P is not owned by the requesting node or its immediate neighbors, the request must be routed through the CAN infrastructure until it reaches the node in whose zone P lies. Efficient routing is therefore a critical aspect of our CAN.

In the following sections, we describe the three core pieces of our design: incorporating new nodes into the CAN, CAN routing, and adjusting to the departure of nodes from the CAN overlay.

2.1.1 Node Arrivals

As described above, the entire CAN space is divided among the nodes currently in the system. To obtain such a partitioning, each time a new node joins the CAN, an existing zone is split into two halves, one of which is assigned to the new node. The split is done by following a well-known ordering of the dimensions in deciding along which dimension a zone is to be split, so that zones can be re-merged when nodes leave. For example, for a 2-d space, a zone would first be split along the X dimension, then the Y, then X again followed by Y and so forth. Figure 2.2 depicts the evolution of a 2-d CAN space as 5 nodes join in succession. The first node to join owns the entire CAN space; *i.e.,* its zone is the complete virtual space. When the second node joins, the space is split in two and each node gets one half. The third node to arrive picks one zone and splits it in half, and this process repeats as new nodes arrive.

At a general step we can thus think of each existing zone as a leaf of a binary "partition tree". The internal vertices in the tree represent zones that no longer exist, but were split at some previous time. The children of a tree vertex are the two zones into which it was split. The edges in the partition tree are labelled as follows: an edge connecting a parent and child zone is labelled "0" if the child zone occupies the lower half of the dimension along which the parent zone was split, otherwise (*i.e.,* if the child zone occupies the upper half of the dimension along which the split occured) the edge is labelled with a "1". Figure 2.3 represents a 5 node CAN and its corresponding labelled partition tree. A zone's position (*i.e.,* the zone's coordinate span along each dimension) in the coordinate space is completely defined by the path from the root of the partition tree to the leaf node corresponding to that zone. Consider for example, the path from the root node to the leaf node 2 in Figure 2.3: the first edge label tells us that the 2's zone lies in the range [0.5,1.0] along the X axis, the second edge indicates that 2's zone lies in the range [0.5,1.0] along the Y axis and the third edge indicates that 2's zone lies in [0.5, 0.75] along the X axis (determined as the lower half of its previously determined span of [0.5, 1.0]). Every node in the CAN is addressed with a virtual identifier (VID) – the binary string representing the path from the root in the partition tree to the leaf node corresponding to the node's zone. Thus a node's VID compactly represents its position in the CAN Cartesian space. Of course we don't maintain this partition tree as a data structure, and none of the CAN operation require a node to have knowledge of the entire partition tree, however the tree is a useful conceptual aid to understanding the structure of nodes in a CAN.

To allow the CAN to grow incrementally, a new node that joins the system must (a) be allocated its own portion of the coordinate space (*i.e.,* obtain a unique VID) and (b) discover its neighbors in the space (*i.e.*, discover its neighbors' VIDs and IP addresses). Briefly, this is done by an existing node splitting its allocated zone in half, retaining half and handing the other half to the new node. The process takes three steps:

1. First the new node must find a node already in the CAN.
2. Next, using the CAN routing mechanisms, it must find a node whose zone will be split.
3. Finally, the neighbors of the split zone must be notified so that routing can include the new node.

Bootstrap

A new CAN node first discovers the IP address of any node currently in the system. The functioning of a CAN does not depend on the details of how this is done, but we use the same bootstrap mechanism as YOID [13]. As in [13] we assume that a CAN has an associated DNS domain name, and that this resolves to the IP address of one or more CAN bootstrap nodes. A bootstrap node maintains a partial list of CAN nodes it believes are currently in the system. Simple techniques to keep this list reasonably current are described in [13].

To join a CAN, a new node looks up the CAN domain name in DNS to retrieve a bootstrap node's IP address. The bootstrap node then supplies the IP addresses of several randomly chosen nodes currently in the system.

Finding a Zone

The new node then randomly chooses a point P in the space and sends a JOIN request destined for point P. This message is sent into the CAN via any existing CAN node. Each CAN node then uses the CAN routing mechanism (desribed later) to forward the message, until it reaches the node in whose zone P lies.

On receiving the JOIN message, the owner of this zone could directly split its own zone with the new node. However, the owner node knows not only its own zone coordinates, but also those of its neighbors. Therefore, instead of directly splitting its own zone, the existing occupant node first compares the volume of its zone with those of its immediate neighbors in the coordinate space. The zone that is split to accommodate the new node is then the one with the largest volume. While not strictly required, the effect of this *1-hop volume check* is to achieve a more uniform partitioning of the space over all nodes.[2]

The selected occupant node then splits its zone in half; the occupant retains the half occupying the lower end of the dimension along which the zone is split and assigns the

[2] Since (key,value) pairs are spread across the coordinate space using a uniform hash function, the volume of a node's zone is indicative of the size of the (key,value) database the node will have to store, and hence indicative of the load placed on the node. A uniform partitioning of the space is thus desirable to achieve load balancing. Note that this is not sufficient for true load balancing because some (key,value) pairs will be more popular than others thus putting higher load on the nodes hosting those pairs. This is similar to the "hot spot" problem on the Web and can be addressed using caching and replication schemes as discussed in [35, 45].

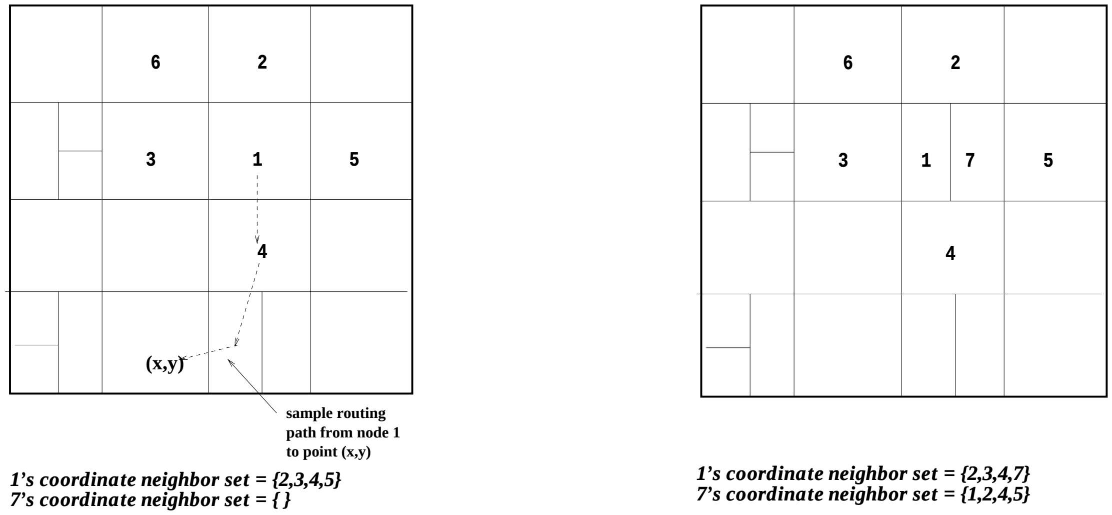

6
2
3
1
5
4
(x,y)
sample routing path from node 1 to point (x,y)
1's coordinate neighbor set = {2,3,4,5}
7's coordinate neighbor set = { }
6
2
3
1
7
5
4
1's coordinate neighbor set = {2,3,4,7}
7's coordinate neighbor set = {1,2,4,5}

Figure 2.4: *Example 2-d space before and after node 7 joins*

other (higher-end) half to the new node.[3] The occupant node then appends a "0" to its original VID to reflect this shrinking of its zone; the new node acquires its VID by simply appending a "1" to the occupant's original VID. Finally, the (key, value) pairs from the half zone to be handed over are also transfered from the occupant node to the new node.

Joining the Routing

Having obtained its zone, the new node must learns the IP addresses of its coordinate neighbor set. In a d-dimensional coordinate space, two nodes are neighbors if their coordinate spans overlap along $d - 1$ dimensions and abut along one dimension. For example, in Figure 2.4, node 6 is a neighbor of node 3 because its coordinate zone overlaps with 6's along the X axis and abuts along the Y-axis. On the other hand, node 2 is not a neighbor of 3 because their coordinate zones abut along both the X and Y axes.

[3] Our choice of which node (new or occupant) acquires which half is entirely arbitrary – any rule whereby the two nodes occupy separate halves is acceptable.

A new node's zone is derived by splitting the previous occupant's zone; consequently, the new node's neighbor set is a subset of the the previous occupant's neighbors, plus that occupant itself. Similarly, the previous occupant updates its neighbor set to eliminate those nodes that are no longer neighbors. Finally, both the new and old nodes' neighbors must be informed of this reallocation of space. Every node in the system sends an immediate update message, followed by periodic refreshes, with its currently assigned zone to all its neighbors. These soft-state style updates ensure that all of their neighbors will quickly learn about the change and will update their own neighbor sets accordingly. Figure 2.4 shows an example of a new node (node 7) joining a 2-dimensional CAN.

As can be inferred, the addition of a new node affects only a small number of existing nodes in a very small locality of the coordinate space. The number of neighbors a node maintains depends only on the dimensionality of the coordinate space and is independent of the total number of nodes in the system. Thus, for a d-dimensional space, node insertion affects only $O(d)$ existing nodes which is important for CANs with huge numbers of nodes.

2.1.2 Routing

Intuitively, routing in a Content Addressable Network works by following the straight line path through the Cartesian space from source to destination coordinates.

A CAN node maintains a coordinate routing table that holds the IP address and VIDs of each of its neighbors in the coordinate space. This purely local neighbor state is sufficient to route between two arbitrary points in the space: A CAN message includes the destination coordinates. Using its neighbor coordinate set, a node routes a message towards its destination by simple greedy forwarding to the neighbor with coordinates closest to the destination coordinates. Figure 2.4 shows a sample routing path.

For a d dimensional space partitioned into n equal zones, individual nodes maintain $2d$ neighbors (two neighbors per dimension, one to advance and one to retreat along each dimension) and the average routing path length is $(d/4)(n^{1/d})$ (each dimension has $n^{1/d}$ nodes; on a torus, a destination will, on average be $(1/4)(n^{1/d})$ nodes away along each of the d dimensions, thus yielding the above result).[4] These scaling results mean that for a d dimensional space, we can grow the number of nodes (and hence zones) without increasing per node state while the path length grows as $O(n^{1/d})$.

Note that many different paths exist between two points in the Cartesian space and so, even if one or more of a node's neighbors were to crash, a node would automatically route along the next best available path. If however, a node loses all its neighbors in a certain direction, and the repair mechanisms described in Section 2.1.3 have not yet rebuilt the void in the coordinate space, then greedy forwarding may temporarily fail. In this case, the forwarding node first checks with its neigbhbors to see whether any of them can make progress towards the destination and if so, greedy routing is resumed through a node two hops away from the current forwarding node. As the example in Figure 2.5 shows, this *1-hop route check* is useful in circumventing certain voids, particularly at lower dimensions when a node has fewer options in finding neighbors that make progress to a destination. If, despite the 1-hop route check, greedy routing fails then the message is forwarded using the rules used to route recovery messages (described in the next section) until it reaches a node

[4] Several recently proposed routing algorithms for location services [33, 8] route in $O(\log n)$ hops with each node maintaining $O(\log n)$ neighbors. Notice that were we to select the number of dimensions $d \geq (\log_2 n)/2$, we could achieve the same scaling properties. We choose to hold d fixed independent of n, since we envision applying CANs to very large systems with frequent topology changes. In such systems, it is important to keep the number of neighbors independent of the system size.

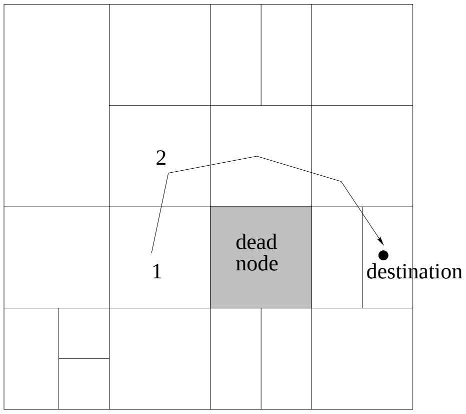


Figure 2.5: *1-hop route check allows node 1 to reach its destination via neighboring node 2 even though node 2 is further from the destination than node 1*

from which greedy forwarding can resume.

2.1.3 Node Departures

When nodes leave a CAN, we need to ensure that the zones they occupied are taken over by the remaining nodes. The normal procedure for doing this is for a node to explicitly hand over its zone state (*i.e.,* its own VID and its list of neighbor VIDs and IP addresses) and the associated (key,value) database to a specific node called the *takeover* node. If the takeover's zone can be merged with the departing node's zone to produce a valid single zone, then this is done. If not, then the takeover node can temporarily handle both zones.

The CAN also needs to be robust to node or network failures, where one or more nodes simply become unreachable. This is handled through a recovery algorithm (described below) that ensures that the takeover node and the failed node's neighbors independently work to reconstruct the routing structure at the failed node's zone. However in this case

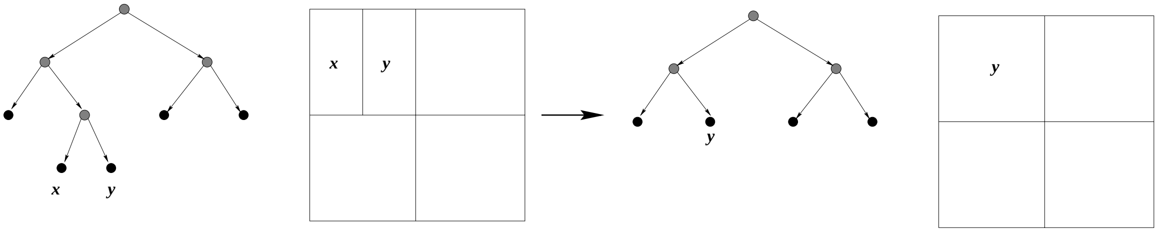


Figure 2.6: *CAN before and after the departure of node x: Case #1*

the (key,value) pairs held by the departing node is lost and needs to be rebuilt. This can be achieved in a number of ways. For example, the state can be refreshed by the holders of the data.[5] Alternately, each (key,value) pair might be replicated at multiple points in the CAN and a lost (key,value) pair can be rebuilt from its replicas. The appropriate solution to reconstructing the (key,value) database is largely dependent on application-level issues such as data consistency and availability requirements and hence we do not address this issue beyond noting that the problem appears tractable.

We now describe the detailed recovery process by which routing state is rebuilt when a node fails. CAN recovery comprises two key pieces: the identification of a unique node, called the *takeover* node, that occupies the departed node's zone and the process by which the departed node's neighbors discover the takeover node and vice versa. We expand on these two pieces in the following sections.

[5] To prevent stale entries as well as to refresh lost entries, nodes that insert (key,value) pairs into the CAN might periodically refresh these entries.

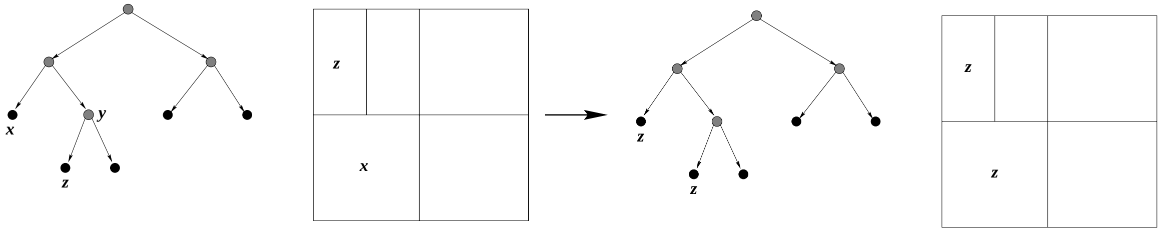


Figure 2.7: *CAN before and after the departure of node x: Case #2*

Identification of Takeover Nodes

The unique, well-defined node that takes over for a given departed node is called the departed node's *takeover*. Conceptually, a given node's takeover can be easily defined using the partition tree. Recall that in a partition tree, the internal vertices represent zones that no longer exist but were split at some time while the children of a tree vertex are the two zones into which it was split. By an abuse of notation, we use the same name for a leaf vertex, for the zone corresponding to that leaf vertex, and for the node responsible for that zone.

Now suppose a leaf vertex x leaves the CAN. If the sibling of this leaf is also a leaf (call it y) the departure is easy: y is the takeover node for x and we simply coalesce leaves x and y, making their former parent vertex a leaf, and assign node y to that leaf. Thus zones x and y merge into a single zone which is owned by node y. Figure 2.6 shows an example of such a takeover.

If x's sibling y is not a leaf (because the sibling zone has been further split), perform a depth-first in the subtree rooted at y until a leaf node is found.[6] This leaf, call

[6] If x is the left sibling of y, the the DFS traversal must visit left siblings before right ones, otherwise (*i.e.,* x is the right sibling of y) the traversal is biased to first visit right siblings.

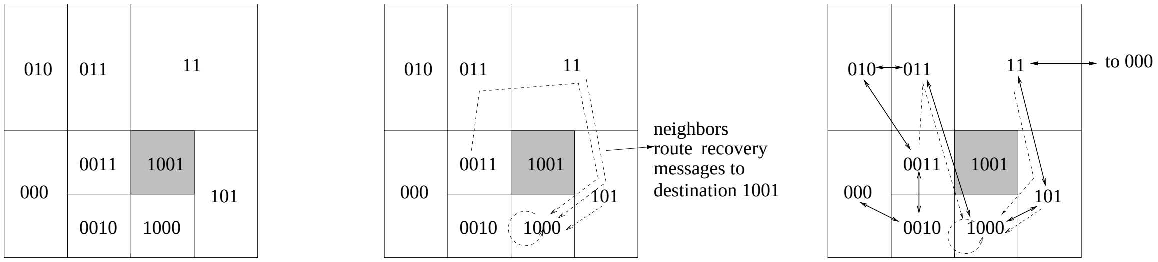
it z, acts as x's sibling and takes over for x. The zones of x and z cannot be simply merged into a single zone and hence z temporarily owns two distinct zones. Node z retains both zones until contacted by a new node at which point, it simply hands off one zone to the new node (rather than split one of its zones).[7] Figure 2.7 shows an example of such a takeover.

The above description uses the partition tree to identify a departed node's takeover. However, nodes do not explicitly maintain the partition tree structure; instead each node maintains only its own VID that summarizes its location in the tree. Nonetheless, as the following description reveals, a node's VID is enough information to identify its takeover. As stated earlier, a node's VID is a binary string that denotes the path from the root to the node in the partition tree. Thus a VID can be regarded as the *prefix* of a k-bit string where k is selected to be greater than the depth of any partition tree expected in practice.[8] A VID of length l, thus specifies the first l of the k bits; the remaining $k - l$ bits are simply set to zero. Regarded in this manner, every VID has an associated numerical value and examination of the partition tree reveals that for a given node, its takeover (as defined earlier) is simply the node with VID numerically closest to its own VID.

Note, that our system now uses two different notions of "distance"; the first is the Cartesian distance between two points in the CAN coordinate space and the second is the absolute value of the difference between two VIDs as defined above. In the remainder of this document, we use the terms "Cartesian distance" and "VID distance" to disambiguate between the two distance metrics.

[7] Alternatively, z can simply depart from its original zone (since it is the smaller of the two zones); the original zone will then in turn be taken over by a node lower in the tree than x causing a cascade of departures that terminates with the merging of the first pair of sibling leaf vertices encountered by a depth first search in the subtree rooted at y.

[8] For example, k could be set to a constant times $\log N$ where N is greater than the expected maximum number of CAN nodes.



node 1001 dies

node 1001's neighbors route recovery messages towards VID=1001; recovery messages terminate at takeover node 1000

ordered link list of VIDs maintained in the face of node failures using Chord's stabilization algorithm.

Figure 2.8: *CAN Recovery: discovering the takeover node*

To summarize, when a node departs the CAN, its zone is taken over by the node with VID numerically closest to the departed node's VID. We now describe the actual process by which the departed node's neighbors and the takeover node discover each other.

Recovery algorithm

The preceeding discussion defined the takeover node that occupies a departed node's zone but did not specify how the departed node's neighbors discover the takeover node and vice versa. We now addresss this issue of restoring neighbor links – we describe a distributed recovery algorithm by which each of the departed node's neighbors independently discovers the takeover node.

Under normal conditions a node sends periodic update messages to each of its neighbors. The prolonged absence of an update message from a neighbor signals its failure. When a node detects a failed neighbor, it deletes the dead node from its neighbor set and attempts to contact the dead neighbor's takeover. It does so by forwarding a *recovery* message to its neighbor closest to the dead node in terms of VID-distance.[9] In this manner, the recovery message is routed incrementally closer to the dead node's VID. Ideally, this recovery message arrives at the takeover node from where it cannot be forwarded any closer to the dead node's VID. The takeover node thus infers that it is to occupy the dead node's zone and that the source of the recovery message is a neighbor of its newly acquired zone. Since each of the dead node's neighbors will independently initiate such a recovery message, the takeover node discovers all its new neighbors and vice versa. An example of this process is shown in Figure 2.8 where the node with VID 1001 dies and is replaced by node 1000. Using only the above algorithm, if multiple nodes fail simultaneously, it is possible that a recovery message might deadend at a node other than the takeover node. For example, in Figure 2.8, if node 11 were to also fail then, at the node 011, the recovery message from node 0011 would no longer be able to make progress towards VID 1001. For resilience to such failures, we borrow an idea first proposed in the Chord routing algorithm [47]. In Chord[10], nodes are assigned unique binary identifiers and each node maintains a *successor* pointer to the first node with identifer greater than its own, effectively maintaining an ordered linked list of all the nodes in the system. This linked list, provided it can be maintained in the face of node dynamics, guarantees connectivity between any two nodes. For this, Chord also specifies a *stabilization* algorithm guaranteed, with high probability, to maintain this ordered linked list in the face of any sequence of node arrivals and departures.

Inspired by Chord's idea of maintaining an ordered link list of nodes, we now augment our algorithms to require every node to know their immediate successor and pre-

[9] Note that a node does not, a priori, know the VID or IP address of the takeover node; it only knows that the takeover's VID is closest to the dead node's VID.

[10] We discuss the Chord algorithm in more detail in Chapter 5.

decessor in the numerical ordering of VIDs. Provided this chain of VIDs is maintained, a recovery message is guaranteed to arrive at the appropriate takeover node because every node can always make progress (in terms of VID-distance) to an arbitrary VID. Figure 2.8 depicts this ordered link list and its use in routing recovery messages when node 1001 dies.

To recap: In addition to its regular neighbor set, every node now maintains links to its immediate successor and predecessor in the VID space and uses Chord's stabilization algorithm to actively maintain this ordered link list of nodes. When a node dies its zone is taken over by the node closest (in terms of VID-distance) to the dead node. The dead node's neighbors discover this takeover node by greedy routing towards the dead node's VID.

2.2 Evaluation

The previous section described the design of a Content-Addressable Network; we now evaluate this design through simulation. Our CAN simulator models the underlying physical network as a Transit-Stub (TS) topology using the GT-ITM topology generator[49]. TS topologies model networks using a 2-level hierarchy of routing domains with transit domains that interconnect lower level stub domains.

Our evaluation focusses on four key performance aspects:

1. scalability: the ability of the system to scale to large numbers of nodes

2. robustness: the ability of the system to withstand, and adapt to, node failures

3. load balancing: the extent to which the load is evenly spread across nodes

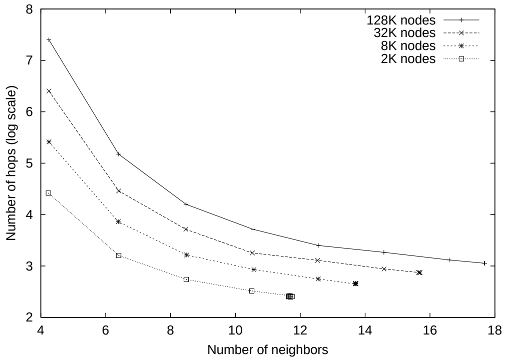


Figure 2.9: *State-efficiency tradeoff*

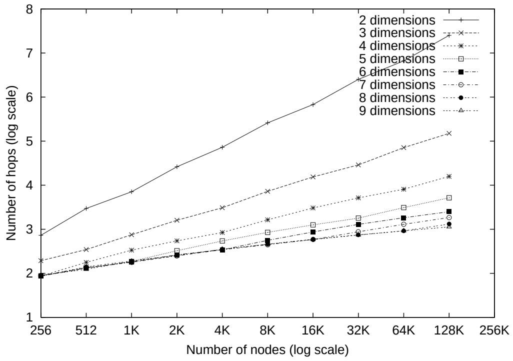


Figure 2.10: *Effect of dimensions on path length*

4. low-latency: the ability of the system to reduce the overhead of overlay routing

Our CAN algorithm has a single design parameter: d, the number of dimensions in the Cartesian space; we look at the effect of varying d on the above performance aspects particularly as n, the number of nodes in the CAN is scaled up. From the above list, we discuss the first three items in this chapter and defer the issue of low-latency routing to Chapter 3.

2.2.1 Scalability

The most obvious measure of the scalability of a routing algorithm is the overhead associated with keeping routing tables (*i.e.,* neighbor sets). Our metric in measuring this overhead is the number of neighbors per node. This isn't just a measure of the state required to do routing but it is also a measure of how much state needs to be adjusted when nodes join or leave. Given the prevalence of inexpensive memory and the highly transient user populations in P2P systems, this second issue is likely to be much more important than the first. To present a more complete picture, when we discuss the per-node state required by a routing algorithm, we will also cite the corresponding routing efficiency as measured by the resulting pathlength which is the number of application-level hops along a path. This is becaues one can always trade-off state for efficiency or vice versa as can be seen from two extreme examples: on one end, a complete mesh of nodes where pathlengths are $O(1)$ but each node maintains $O(n)$ routing state and at the other end, a circular linked list where each node maintains $O(1)$ neighbor state but the resulting pathlength is $O(n)$. Ideally, one would thus like to achieve a reasonable tradeoff between these two metrics.

For a uniformly partitioned d-dimensional Cartesian space with n nodes, every CAN node has $2d$ neighbors (one in each direction along every dimension) and the resulting average pathlength is $(dn^{1/d})/4$. Since the Cartesian space is unlikely to be perfectly partitioned in practice, we measured the per-node neighbor state and resulting pathlength through simulation. Figure 2.9 plots the pathlength-neighbor tradeoff for CANs with different system sizes while Figure 2.10 uses the same data to plot the path length for increasing numbers of CAN nodes for coordinate spaces with different dimensions. For a system with n nodes and d dimensions, we see that the path length scales as $O(dn^{1/d})$ in keeping with the analytical results for perfectly partitioned coordinate spaces. Also, from the clustering of data points at the tail end of each plotted line in Figure 2.9, we see that for a system with n nodes, increasing the number of dimensions beyond $\log n/2$ dimensions does not serve to further reduce the pathlength. This is because, n nodes are insufficient to fill the space at these higher dimensions; *i.e.,* zones are unlikely to be split along the higher order dimensions.

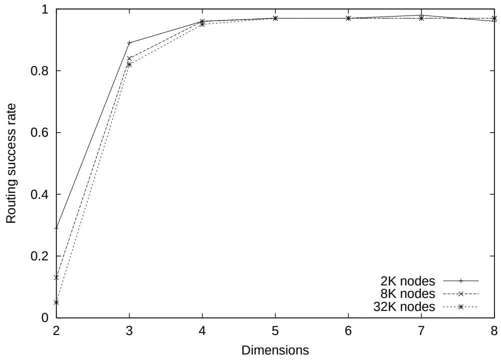


Figure 2.11: *Routing resilience improves with increasing dimensions*

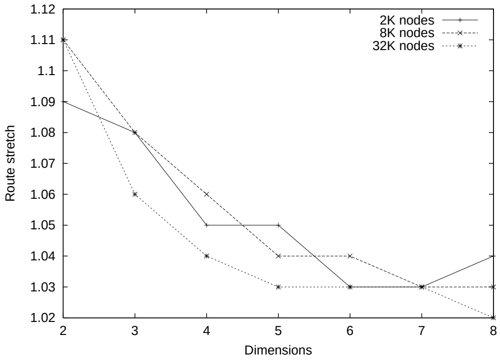


Figure 2.12: *Routing stretch decreases with increasing dimensions*

2.2.2 Routing Resilience

The above routing results refer to a perfectly functioning system with all nodes operational. However, P2P nodes are notoriously transient and the resilience of routing to failures is a very important consideration. Characterizing the performance of CAN in a realistic failure-prone setting depends on many details such as the node failure model (random, byzantine, etc.) and the time constants in the recovery algorithm. Here, we focus on the *routing* issues in achieving resilience. In other words, we focus our evaluation on understanding the extent to which the routing is capable of reaching a node that is alive. We do not directly measure the availability of data stored in the CAN merely noting that if a piece of data can be kept alive (*i.e.,* replicated/cached at a live node) in the CAN then the routing resilience is indicative of the likelihood that this data will be available.

As mentioned earlier, there are multiple paths between two points in the CAN space and hence nodes can route around trouble provided at least one neighbor node (or a neighbor's neighbor using the 1-hop route check) can make progress towards the destination.

Thus the structure of the overlay network itself provides a certain level of robustness due to the redundancy in routing paths. To capture this, we evaluate the extent to which routing can continue to function (and with what efficiency) as nodes fail without any time for other nodes to establish other neighbors to compensate; that is the neighboring nodes know that a node has failed, but they don't use the recovery scheme to establish any new neighbor relations with other nodes. We will call this *static resilience* and measure it in terms of:

- the percentage of available key locations that are still reachable. *I.e.* if a destination node is still alive, can it be reached by other live nodes?
- routing stretch: the resulting *increase* in the pathlength of a successful route. Note that it is important to compute the increase in the pathlength of a particular path (*i.e.,* with particular source and destination coordinates) because as the node failure rate increases, successful routes are more likely to be those routes with fewer hops to begin with; simply measuring the absolute pathlength of successful routes might thus falsely hide any degradation in pathlengths under node failures.

Figures 2.11and 2.12 plot the percentage of reachable key locations and the increase in pathlength respectively for increasing CAN dimensions. The node failure rate here is held fixed at 25%. As expected, performance improves with increasing dimensions because of the greater number of alternate paths at higher dimensions. Also, performance degrades slightly with increasing n because pathlengths are longer and the chance of a route failure increases accordingly. Overall, we see that routing is extremely resilient even at reasonably low dimensions (*e.g.,* 95% of route attempts are successful for a 32,768 node CAN with only 4 dimensions) and the penalty in terms of increase in pathlength is very low (at most 1.11

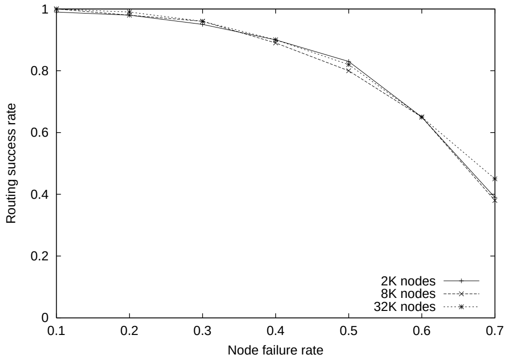


Figure 2.13: *Routing resilience with increasing failure rate*

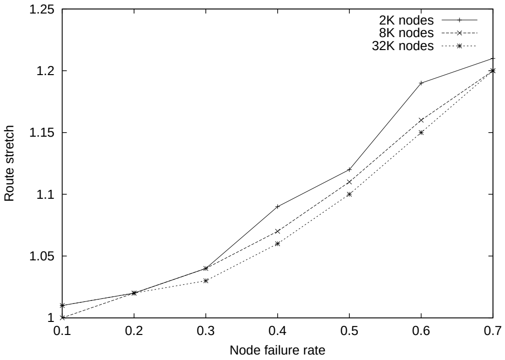


Figure 2.14: *Routing stretch increases with increasing failure rate*

times the optimal pathlength for the above test cases).

Figures 2.13 and 2.14 plot the percentage of reachable key locations and the increase in pathlength respectively with increasing node failure rates for CANs with 8 dimensions. We see that routing remains resilient upto large failure rates – close to 80% of route attempts succeed even with a 50% node failure rate.

To identify the usefulness of the 1-hop route check described in Section 2.1.3, Figure 2.15 plots the percentage of reachable key locations both with and without the 1-hop check for both test cases – increasing dimensions and node failure rate. As can be seen, the 1-hop check can improve routing performance by upto 20%.

Static resilience represents the worst-case scenario in that no attempt is made to recover from node failures. The above results show that, even in this scenario, CAN routing is extremely resilient. A conclusion one might draw from this is that routing is unlikely to be the bottleneck in achieving high data availability; if a piece of data is stored at a live node, that node is very likely reachable through the routing. Rather, the challenge to

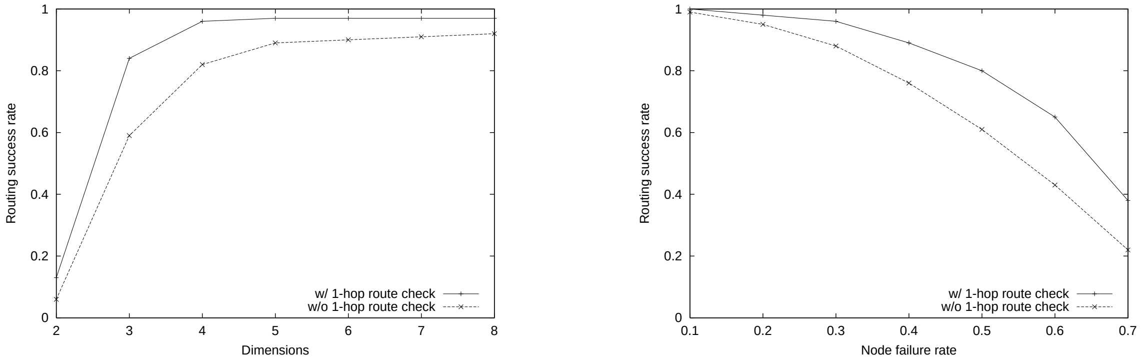


Figure 2.15: *Performance gain with the 1-hop route check for increasing dimensions and node failures*

applications layered over CAN will lie in devising replication schemes that can maintain a live replica in failure-prone environments.

The recovery algorithms described in Section 2.1.3 serve to repair the CAN overlay structure in the face of node failures and can only improve the resiliency.

2.2.3 Load Balancing

The volume of a node's zone is indicative of the size of the (key,value) database it will have to store and the amount of message forwarding it is likely to perform. Hence, while we do not require a perfect partitioning of the space to achieve reasonable load balancing, we would, at the very least, want to avoid any significantly non-uniform partitionings. Recall the join procedure for new CAN nodes: a new node, picks a random point in space and discovers its current occupant; the occupant performs a *1-hop volume check*, selecting the largest of its own and its immediate neighbor's zones which is then split with the new node.

To measure the extent to which the CAN space is uniformly partitioned and also the usefulness of this 1-hop volume check we ran simulations with 32,768 nodes both with and without this 1-hop volume check. At the end of each run, we compute the volume of the zone assigned to each node. If the total volume of the entire coordinate space were V_T and n the the total number of nodes in the system then a perfect partitioning of the space among the n nodes would assign a zone of volume V_T/n to each node. We use V to denote V_T/n. Fig 2.16 plots different possible volumes in terms of V on the X axis and shows the percentage of the total number of nodes (Y axis) that were assigned zones of a particular volume. From the plot, we can see that without the 1-hop volume check, a little over 40% of the nodes are assigned to zones with volume V as compared to almost 82.0% with this feature; the largest zone volume drops from $8V$ to $2V$.

Figure 2.17 plots the percentage of nodes with the ideal zone volume of V for increasing dimensions. We see, not surprisingly, that the partitioning of the space using the 1-hop volume check further improves with increasing dimensions. For this simulation, with dimensions of 3 and higher, all the zone volumes lay between $V/2$ and $2V$ (for 2 dimensions, the spread was from $V/4$ to $4V$) confirming that the 1-hop volume is indeed very useful in achieving a close to perfect partitioning of the space.

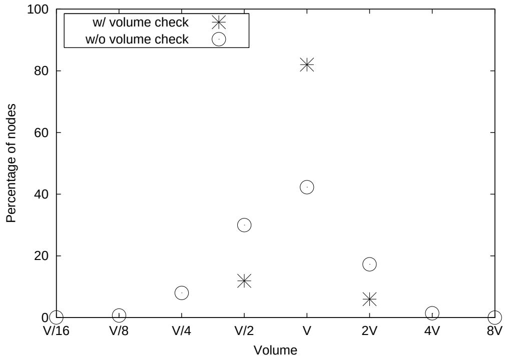


Figure 2.16: *Distribution of zone volumes with and without 1-hop volume check for a CAN with 32,768 nodes and 3 dimensions*

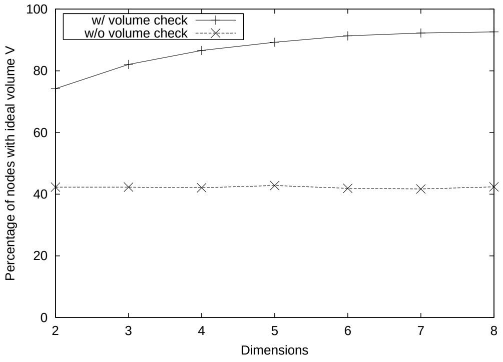


Figure 2.17: *Effect of increasing dimensions on the efficacy of the 1-hop volume check for a CAN with 32,768 nodes*

Chapter 3

Low-latency routing in CAN

The efficiency measure used in the previous chapter was the pathlength in number of application-level hops and the corresponding amount of routing state. By this metric, our CAN algorithm achieves a balance between low per-node state of $O(d)$ for a d-dimensional space and short pathlengths of $O(dn^{1/d})$ hops for d dimensions and n nodes. However, the true efficiency measure is the end-to-end *latency* of the path. The average lookup latency is the average number of CAN hops times the average latency of each CAN hop. The above bounds refer only to the number of application-level hops and the latency of each such hop might be substantial; recall that nodes that are adjacent in the CAN might be many miles (and many IP hops) away from each other. Ideally, we would like to achieve a lookup latency that is comparable (within a small factor) to the underlying IP path latencies (between the requester and the CAN node holding the key). In this chapter, we describe various schemes whose primary goal is to reduce the latency of CAN routing. Not unintentionally, many of these techniques offer the additional advantage of improved CAN robustness both in terms of routing and data availability. These heuristics also serve to illustrate the many ways in which the CAN design can be adapted to meet performance requirements.

In a nutshell, our strategy in attempting to reduce path latency is to reduce either the path length or the per-hop latency because the overall CAN path latency depends directly on the path length (*i.e.* number of hops) and the latency of each hop along the path. A key difference between the two approaches is that reducing the number of application-level hops per lookup still does not require any knowledge of the geographic (*i.e.,* latency) properties of the underlying IP paths; reducing the per-hop latency on the other hand does require incorporating some level of knowledge of the geographic properties of paths with more fine-grained knowledge presumably offering greater latency savings.

The heuristics described in this chapter yield significant improvements but come at the cost of increased per-node state (although per-node state still remains independent of the number of nodes in the system) and somewhat increased complexity. We stress that the heuristics presented in this chapter are *not* part of the core CAN algorithm and are not required for correct CAN operation. Rather, the heuristics presented here can be viewed as different (largely independent) approaches to optimizing overlay routing performance. The extent to which the following heuristics are applied (if at all) involves a trade-off between improved routing performance and system robustness on the one hand and increased per-node state and system complexity on the other. Until we have greater deployment experience, and know the application requirements better, we are not prepared to decide on these tradeoffs.

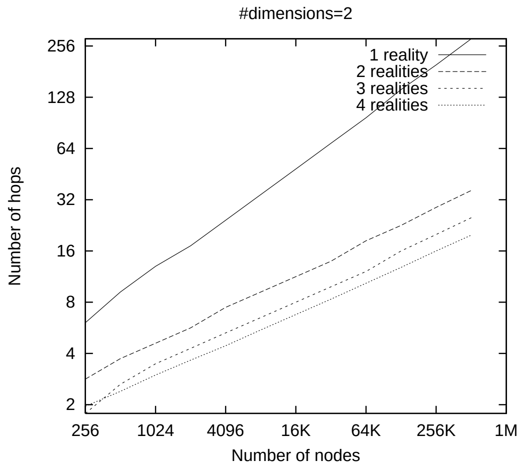


Figure 3.1: *Pathlength decreases with increasing number of realities*

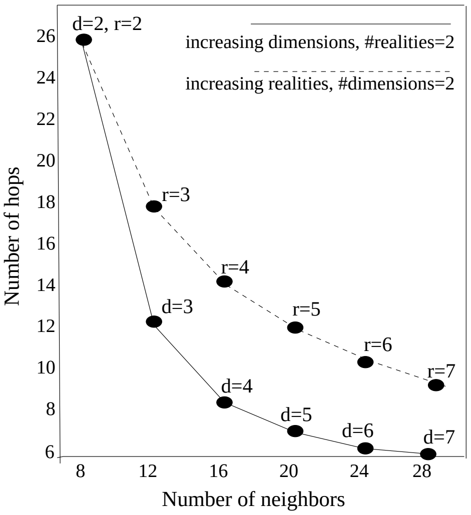


Figure 3.2: *Path length with increasing neighbor state: multiple realities versus multiple dimensions*

3.1 Reducing Pathlengths

We consider two alternate approaches to reducing the average CAN pathlength:

- keeping additional routing state that provides a node with "shortcut" links to distant portions of the Cartesian space thus reducing the average pathlengths.
- storing multiple copies of a (key, value) pair so that a node can fetch the replica closest to itself in terms of Cartesian distance, once again leading to fewer overlay hops.

In what follows, we discuss the particular schemes we have experimented with under each approach.

Realities: Multiple Coordinate Spaces

While there are likely many ways to reduce pathlengths by increasing routing state[1] we select here a scheme that not only results in more efficient routing but also improves data availability. Our strategy is to maintain multiple, independent coordinate spaces with each node in the system being assigned a different zone in each coordinate space. We call each such coordinate space a “reality”. Hence, for a CAN with r realities, a single node is assigned r coordinate zones, one on every reality and holds r independent neighbor sets.

The contents of the hash table are replicated on every reality. This replication improves data availability. For example, say a pointer to a particular file is to be stored at the coordinate location (x,y,z). With three independent realities, this pointer would be stored at 3 different nodes corresponding to the coordinates (x,y,z) on each reality and hence is unavailable only when all three nodes are unavailable. Further, because the contents of the hash table are replicated on every reality, routing to location (x,y,z) translates to reaching (x,y,z) on *any* reality. A given node owns one zone per reality each of which is at a distinct, and possibly distant, location in the coordinate space. Thus, an individual node has the ability to, in a single hop, reach distant portions of the coordinate space thereby reducing the average path length. To forward a message, a node now checks all its neighbors on each reality and forwards the message to that neighbor with coordinates closest to the destination. Figure 3.1 plots the path length for increasing numbers of nodes for different numbers of realities. From the graph, we see that realities significantly reduce path length and consequently the overall CAN path latency. Finally, multiple realities improve routing

[1] An obvious way is to increase the dimensionality of the CAN space. Increasing the dimensions of the CAN coordinate space reduces the routing path length, and hence the path latency, for a small increase in the size of the coordinate routing table.

fault tolerance, because in the case of a routing breakdown on one reality, messages can continue to be routed using the remaining realities.

A CAN system could thus make use of multiple, multi-dimensional coordinate spaces with every additional reality improving robustness and routing efficiency at the cost of increased control and data state per node.

Multiple dimensions versus multiple realities Increasing either the number of dimensions or realities results in shorter path lengths and higher per-node state. Here we compare the relative improvements caused by each of these features. Figure 3.2 plots the path length versus the average number of neighbors maintained per node for increasing dimensions and realities. We see that for the same number of neighbors, increasing the dimensions of the space yields shorter path lengths than increasing the number of realities. One should not, however, conclude from these tests that multiple dimensions are more valuable than multiple realities because multiple realities offer other benefits such as improved data availability and fault-tolerance. Rather, the point to take away is that if one were willing to incur an increase in the average per-node neighbor state for the sole purpose of improving routing efficiency, then the right way to do so would be to increase the dimensionality d of the coordinate space, rather than the number of realities r.

Multiple hash functions

For shorter pathlengths and improved data availability, one could use k different hash functions to map a single key onto k points in the coordinate space and accordingly replicate a single (key,value) pair at k distinct nodes in the system. A (key,value) pair is

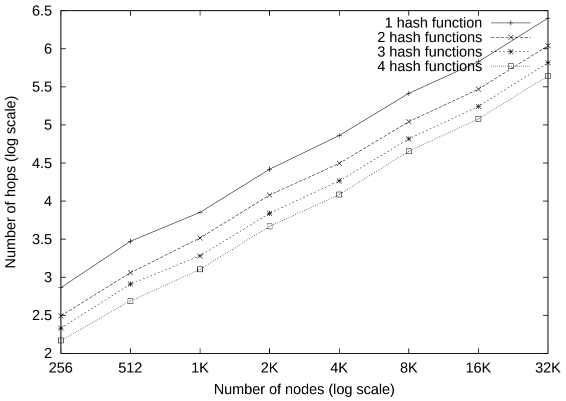


Figure 3.3: *Reduction in pathlength with the use of multiple hash functions*

then unavailable only when all k replicas are simultaneously unavailable. Note the difference between multiple hash functions that store a (key,value) pair at different points on a single coordinate space and multiple realities that store an entry at the same point on multiple coordinate spaces. With k available replicas, a node now retrieves an key-value entry from that replica which is closest to it in the coordinate space.

Figure 3.3 plots this query latency (*i.e.* the time to fetch a (key,value) pair) for increasing number of nodes for different numbers of hash functions. Of course, the reduction in latency come at the cost of increasing the size of the (key,value) database.

3.2 Incorporating Geography

In our discussion so far, neighboring nodes in a CAN could be geographically dispersed and hence some application-level hops could involve transcontinental links, and others merely trips across a LAN; routing algorithms that ignore the latencies of individual hops are likely to result in paths with some high latency hops. In this section, we discuss different approaches to extending the CAN algorithms to make some attempt at incorporating the geographic proximity of nodes. There are (at least) three ways of coping with geography.

Proximity Routing: Proximity routing is when the routing choice is based not just on which neighboring node makes the "most" progress towards the destination key, but is also based on which neighboring node is "closest" in the sense of latency. Our routing metric, as described in Section 2.1.2, is the progress in terms of Cartesian distance made towards the destination. One can improve this metric to better reflect the underlying IP topology by having each node measure the network-level round-trip-time (RTT) to each of its neighbors. For a given destination, a message is now forwarded to the neighbor with the maximum ratio of progress to RTT. This favors lower latency paths, and helps the application level CAN routing avoid unnecessarily long hops.

Unlike increasing the number of hash functions or realities, proximity routing aims at reducing the latency of individual hops along the path and not at reducing the path length. Thus, our metric for evaluating the efficacy of proximity routing is the per-hop latency, obtained by dividing the overall path latency by the path length.

To quantify the effect of this routing metric, we used Transit-Stub topologies with link latencies of 100ms for intra-transit domain links, 10ms for stub-transit links and 1ms for intra-stub domain links. With our simulated topology, the average end-to-end latency of the underlying IP network path between randomly selected source-destination nodes is approximately 115ms. Table 3.1 compares the average per-hop latency with and without

Number of dimensions	Non-RTT weighted routing (ms)	RTT weighted routing (ms)
2	116.8	88.3
3	116.7	76.1
4	115.8	71.2
5	115.4	70.9

Table 3.1: *Per-hop latency using proximity routing*

Nodes per zone	per-hop latency (ms)
1	116.4
2	92.8
3	72.9
4	64.4

Table 3.2: *Per-hop latencies using multiple nodes per zone*

RTT weighting. These latencies were averaged over test runs with n, the number of nodes in the CAN, ranging from 2^8 to 2^{18}.

As can be seen, while the per-hop latency without proximity routing matches the underlying average IP network latency, RTT weighted routing lowers the per-hop latency by between 24 - 40% depending on the number of dimensions. Higher dimensions give more next-hop forwarding choices and hence even greater improvements.

Proximity Neighbor Selection: This is a variant of the idea above, but now the proximity criterion is applied when choosing neighbors, not just when choosing the next hop. Unfortunately, the CAN algorithm does not easily lend itself to proximity neighbour selection because given its zone, a node has no flexibility in selecting which nodes (*i.e.,* zones) are to be its neighbors. A way around this constraint is to *overload* zones to allow multiple nodes to share the same zone. Nodes that share the same zone are termed peers. We define a system parameter MAXPEERS, which is the maximum number of allowable peers per zone (we imagine that this value would typically be rather low, 3 or 4 for example).

With zone overloading, a node maintains a list of its peers in addition to its neighbor list. While a node must know all the peers in its own zone, it need not track all the peers in its neighboring zones. Rather, a node selects one node from each of its neighboring zones. Thus, zone overloading does not increase the amount of *neighbor* information an individual node must hold, but does require it to hold additional state for up to MAXPEERS peer nodes.

Overloading a zone is achieved as follows: When a new node A joins the system, it discovers, as before, an existent node B whose zone it is meant to occupy. Rather than directly splitting its zone as described earlier, node B first checks whether it has fewer than MAXPEERS peer nodes. If so, the new node A merely joins B's zone without any space splitting. Node A obtains both its peer list and its list of coordinate neighbors from B. Periodic soft-state updates from A serve to inform A's peers and neighbors about its entry into the system.

If the zone is full (already has MAXPEERS nodes), then the zone is split into half as before. Node B informs each of the nodes on it's peer-list that the space is to be split. Using a deterministic rule (for example the ordering of IP addresses), the nodes on the peer list together with the new node A divide themselves equally between the two halves of the now split zone. As before, A obtains its initial list of peers and neighbors from B.

Periodically, a node sends its coordinate neighbor a request for its list of peers, then measures the RTT to all the nodes in that neighboring zone and retains the node with the lowest RTT as its neighbor in that zone. Thus a node will, over time, measure the round-trip-time to all the nodes in each neighboring zone and retain the closest (*i.e.* lowest latency) nodes in its coordinate neighbor set. After its initial bootstrap into the system, a node can perform this RTT measurement operation at very infrequent intervals so as to not unnecessarily generate large amounts of control traffic.

The contents of the hash table itself may be either divided or replicated across the nodes in a zone. Replication provides higher availability but increases the size of the data store at every node by a factor of MAXPEERS (because the overall space is now partitioned into fewer, and hence larger, zones) and data consistency must be maintained across peer nodes. On the other hand, partitioning data among a set of peer nodes does not require consistency mechanisms or increased data storage but does not improve availability either.

Overloading zones reduces the per-hop latency because a node now has multiple choices in its selection of neighboring nodes and can select neighbors that are closer (in terms of latency). Table 3.2 lists the average per-hop latency for increasing MAXPEERS for system sizes ranging from 2^8 to 2^{18} nodes with the same Transit-Stub simulation topologies as in Chapter 2.1. We see that placing 4 nodes per zone can reduce the per-hop latency by about 45%.

Overloading zones also leads to shorter routing pathlengths because placing multiple nodes per zone has the same effect as reducing the number of nodes in the system; *i.e.,* the path length is still $O(dn^{1/d})$ but n, is now approximately n/MAXPEERS. Overloading zones also improves fault tolerance because a zone is vacant only when *all* the nodes in a zone crash simultaneously (in which case the repair process of Section 2.1.3 is still required). On the negative side, overloading zones adds somewhat to system complexity because nodes must additionally track a set of peers.

Geographic Layout: So far, in our CAN algorithm, the node identifiers are chosen randomly (*i.e.,* a new node joins the CAN at a randomly selected point in space) and the neighbor relations are established based solely on these node identifiers and so a node's neighbors on the CAN need not be topologically nearby on the underlying IP network. This can lead to seemingly strange routing scenarios where, for example, a CAN node in Berkeley has its neighbor nodes in Europe and hence its path to a node in Stanford may traverse distant nodes in Europe. While the design mechanisms described so far try to improve the selection of paths on an existing overlay network they do not try to improve the overlay network structure itself. One could also attempt to choose node identifiers in a geographically informed manner. Note that geographic layout differs from the two above proximity methods in that here there is an attempt to affect the global layout of the node identifiers, whereas the proximity methods merely affect the local choices of neighbors and forwarding nodes.

Achieving geographic layout requires some level of knowledge of the relative proximity between nodes. Here, we present a *distributed binning* scheme that uses network latency measurements to infer coarse-grained topological information. Our binning strategy is simple (requiring minimal support from any measurement infrastructure), scalable (requiring no form of global knowledge, each node only needs knowledge of a small number of well-known landmark nodes) and completely distributed (requiring no communication or cooperation between the nodes being binned). Distributed binning is however, just one of several conceivable approaches to acquiring topological information (see for example, [14, 32, 5, 24]). We use distributed binning because it is simple and lightweight enough to be of practical use in very large scale distributed applications; however any similarly scalable technique that offers reasonably accurate topological hints is likely to serve our purpose equally well.

Our binning scheme assumes the existence of a well known set of machines (for example, the DNS root name servers) that act as landmarks on the Internet. We achieve a form of "distributed binning" of CAN nodes based on their relative distances from this set of landmarks. Every CAN node measures its round-trip-time to each of these landmarks and orders the landmarks in order of increasing RTT. Thus, based on its delay measurements to the different landmarks, every CAN node has an associated ordering. With m landmarks, $m!$ such orderings are possible. Accordingly we partition the coordinate space into $m!$ equal sized portions, each corresponding to a single ordering. Our current (somewhat naive) scheme to partition the space into $m!$ portions works as follows: assuming a fixed cyclical ordering of the dimensions (*e.g.* xyzxyzx...), we first divide the space, along the first dimension, into m portions, each portion is then sub-divided along the second dimension into $m-1$ portions each of which is further divided into $m-2$ portions and so on. Previously, a new node joined the CAN at a random point in the entire coordinate space. Now, a new node joins the CAN at a random point in that portion of the coordinate space associated with its landmark ordering.

The rationale behind this scheme is that topologically close nodes are likely to have the same ordering and consequently, will reside in the same portion of the coordinate space and hence neighbors in the coordinate space are likely to be topologically close on the Internet.

A consequence of the above binning strategy is that the coordinate space is no longer uniformly populated. Because some orderings (bins) are more likely to occur than others their corresponding portions of the coordinate space are also more densely occupied than others leading to an uneven distribution of load amongst the nodes. While we believe the use of background load balancing techniques (described in [35]) where an overloaded node hands off a portion of its space to a more lightly loaded one might be used to alleviate this problem, we do not explore this question further here and defer it to future work.

The metric we use to evaluate the above binning scheme is the ratio of the latency on the CAN network to the average latency on the IP network. We call this the latency *stretch*. A subtle side-effect of the uneven partitioning of the space is that the average number of hops on the path between two points in the CAN space decreases. This is because a single node might own a disproportionately large zone. Such a node thus has the ability to cross a large portion of the coordinate space in a single hop leading to shorter paths than would be the case if the space were uniformly partitioned. This reduced path length in turn leads to lower average CAN path latencies. In order to not take advantage of this reduced CAN latency caused by an uneven partitioning of the space, we calculate the average CAN path latency using binning-based construction as follows: for a CAN with binning-based construction, we divide the path latency by the number of hops on the path to get the per-hop latency. We then multiply this per-hop latency by the average number of hops on the randomly constructed CAN (for which the space is evenly partitioned). This gives us the path latency for a CAN with binning based construction without taking advantage of the uneven space distribution.[2] Finally, we divide this by the path latency on the underlying IP-level network to obtain the latency stretch.

We tested our binning scheme and its application to CAN construction on both

[2] We wish to stress that this adjustment actually makes our results look worse.

simulated physical topologies and Internet measurement data. The test topologies we use are as follows:

1. TS-1K: Transit-Stub topology [49] with 1,000 nodes. We assign link latencies of 20ms for intra-transit domain links, 5ms for stub-transit links and 2ms for intra-stub domain links (we also experimented with a delay distribution of 100, 10 and 1ms instead of 20, 5 and 2ms respectively with no real change in our results).

2. PLRG: Recent studies [12, 48] have indicated that the Internet's degree distribution follows a power-law. Motivated by these observations, degree-based generators have been proposed [1] which appear to better model the measured Internet topology. We make use of the same power-law random graph generator as used by [48, 1]. PLRG is a Power-Law Random Graph with 1,779 nodes respectively. To each link in the topology, we assign a random delay between 5 and 100ms.[3]

3. NLANR: The Active Measurement Project (AMP) at the National Laboratory for Applied Network Research (NLANR) uses a distributed network of over 100 active monitors to systematically perform scheduled measurements between each other. Amongst other things, monitors measure the round trip times (RTT) between the different pairs of monitors. We use an NLANR data set with the round-trip-times between 103 such monitors. Our data set is from measurements taken in April 2001. The NLANR sites are primarily located at universities in North America. The details of the NLANR measurement methodology and sites is described in [17].

[3] These delay assignments are probably quite misleading, since the true Internet latencies are not random; at the very least, they usually obey the triangle inequality. However, we do not yet know how to realistically model the latencies on a PLRG.

A detailed evaluation of distributed binning and its applications is presented in [37]. Here, we restrict ourselves to evaluating the performance of CAN with binning-based construction.

Figure 3.4 plots the CAN latency stretch (defined above) for increasing CAN sizes, *i.e.* for increasing numbers of CAN nodes. We use topology TS-1K and scale the CAN size by adding CAN nodes to the stub (leaf) nodes in the underlying topology. The delay of the link from the end-host node to the stub node is set to 1ms. Thus, in scaling the CAN size from 512 to 16K nodes, we're scaling the density of the graph without scaling the backbone (transit) domain. Figure 3.4 compares the latency stretch for randomly constructed CANs (where nodes join the CAN at a random point in the space, as described in [35]) to the stretch using the binning-based CAN construction scheme outline above. For randomly constructed CANs, we see that the latency stretch simply grows as the pathlength since no effort is made to reduce the per-hop latency. We see that binning-based construction greatly lowers the stretch. Also, as expected, with more landmarks, the binning is more accurate and the stretch decreases further.[4]

Figure 3.5 repeats the above test for topology PLRG. As before, the CAN size is scaled by scaling the density of CAN nodes attached to underlying topology nodes. We are not sure to what extent the random assignment of link delays affects our results for PLRG. Table 3.3 lists the stretch for a 100 node CAN using the NLANR data set. Because the NLANR data set has only 103 nodes, we cannot experiment with increasing CAN sizes as we did for TS and PLRG.

[4] While the absolute value of the stretch appears high, this is primarily because we are using a CAN with only two dimensions. Increasing the dimensionality of the CAN space would greatly reduce the stretch for all construction schemes.

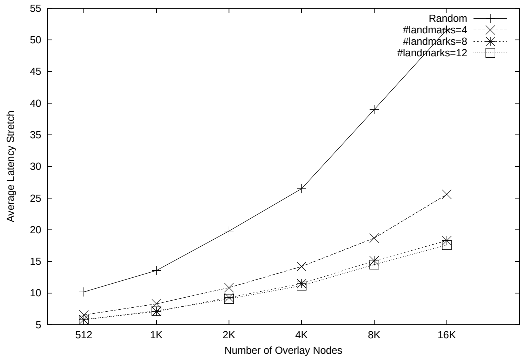


Figure 3.4: *Stretch for a 2-dimensional CAN; topology TS-1K*

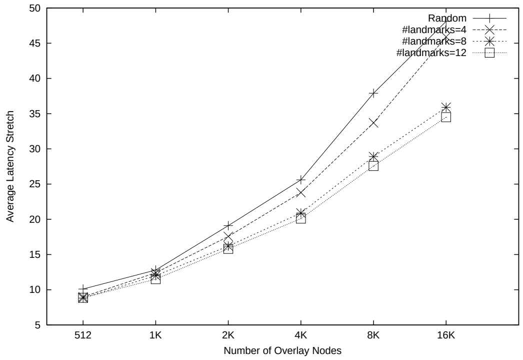


Figure 3.5: *Stretch for a 2-dimensional CAN; topology PLRG*

Construction	Latency Stretch
Random	4.44
Bin, 2 landmarks	4.33
Bin, 3 landmarks	3.9
Bin, 4 landmarks	3.2
Bin, 5 landmarks	3.1

Table 3.3: *Stretch on a 2-d CAN using NLANR*

Chapter 4

M-CAN: CAN-based Multicast

Our interest in CANs is based on the belief that a hash table-like abstraction would give Internet system developers a powerful design tool that could enable new applications and communication models. As an example of how one might use CAN to support rich higher-level services, we describe M-CAN an application-level multicast scheme that builds on CAN.

A number of applications such as software distribution, Internet TV, video conferencing and shared white-boards [29, 20, 27, 23] require one-to-many message transmission to enable efficient many-to-many communication. The IP Multicast service [10] was proposed as an extension to the Internet architecture to support efficient multi-point packet delivery at the network level. With IP Multicast, a single packet transmitted at the source is delivered to an arbitrary number of receivers by replicating the packet within the network at fan-out points (routers) along a distribution tree rooted at the traffic's source. IP Multicast has been studied for many years now. Yet, IP multicast deployment has been slowed by difficult issues related to scalable inter-domain routing protocols, charging models, robust congestion control schemes and so forth [13, 19, 28]. Because of the problems facing the deployment of a network-level multicast service, many recent research proposals have argued for an *application-level* multicast service [4, 13, 19] as a more tractable alternative to network-level multicast and have described designs and applications for such a service.

This chapter looks into the question of how the deployment of a CAN-like distributed infrastructures might be utilized to support multicast services and applications. We outline the design of an application-level multicast scheme built using a CAN. Our design shows that extending the CAN framework to support multicast comes at trivial additional cost in terms of complexity and added protocol mechanism. A key feature of our scheme is that because we exploit the well-defined structured nature of CAN topologies (i.e. the virtual coordinate space) we can eliminate the need for a multicast routing algorithm to construct distribution trees. This allows our CAN-based multicast scheme to scale to large group sizes. While our design is in the context of CANs in particular, we believe our technique of exploiting the structure of these systems should be applicable to other DHT routing algorithms such as Chord [47], Pastry [39] and Tapestry [50].

The majority of previously proposed solutions to application-level (for example [4, 19]) involve having the members of a multicast group self-organize into an essentially random application-level mesh topology over which a traditional multicast routing algorithm such as DVMRP [10] is used to construct distribution trees rooted at each possible traffic source. Such routing algorithms require every node to maintain state for every other node in the topology. Hence, although these proposed solutions are well suited to their targeted applications,[1] their use of a global routing algorithm limits their ability to scale to large (many thousands of nodes) group sizes and to operate under conditions of dynamic group membership.

Bayeux [51] is an application-level multicast scheme that scales to large group sizes but restricts the service model to a single source. In contrast to the above schemes, CAN-based multicast can scale to large group sizes without restricting the service model to a single source.

In summary, M-CAN serves as an example of how one might extend CAN to support richer services such as multicast. Furthermore, M-CAN offers two key advantages relative to prior work on application-level multicast:

- M-CAN is able to scale to very large (*i.e.* many thousands of nodes and higher) group sizes without restricting the service model to a single-source. To the best of our knowledge, no currently proposed application-level multicast scheme can operate in this regime.

- Assuming the deployment of a CAN-like infrastructure, CAN-based multicast is trivially simple to achieve. This is not to suggest that CAN-based multicast by itself is either simpler or more complex than other proposed solutions to application-level multicast. Rather, our point is that CANs can serve as a building block in a range of Internet applications and services and that one such, easily achievable, service is application-level multicast.

[1] The authors in [19], state that End System Multicast is more appropriate for small, sparse groups as in audio-video conferencing and virtual classrooms, while the authors in [4] apply their algorithm, Gossamer, to the self-organization of infrastructure proxies.

The remainder of this chapter is organized as follows: we describe the design of a CAN-based multicast service in Section 4.1 and evaluate this design through simulation in Section 4.2. Finally, we discuss related work on application-level multicast in Section 4.3.

4.1 Design

In this section, we describe our solution for an application-level multicast using CAN. If all the nodes in a CAN are members of a given multicast group, then multicasting a message only requires flooding the message over the entire CAN. As we shall describe in Section 4.1.2, we can exploit the existence of a well defined coordinate space to provide simple, efficient flooding algorithms from arbitrary sources without having to compute distribution trees for every potential source.

If only a subset of the CAN nodes are members of a particular group, then multicasting involves two pieces:

- the members of the group first form a group-specific ”mini” CAN and then,
- multicasting is achieved by flooding over this mini CAN

In what follows, we describe the two key components of our scheme: group formation and multicast by flooding over the CAN.

4.1.1 Multicast Group Formation

To assist in our explanation, we assume the existence of a CAN C within which a subset of the nodes wish to form a multicast group G. We achieve this by forming an additional mini CAN, call it C_g, made up of only the members of G. The underlying CAN

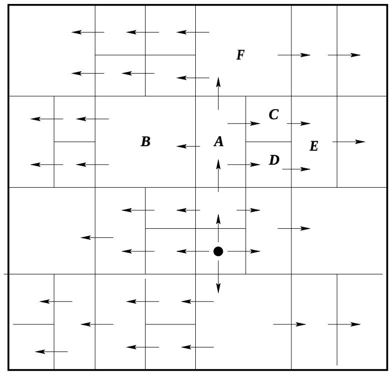


Figure 4.1: *Directed flooding over the CAN*

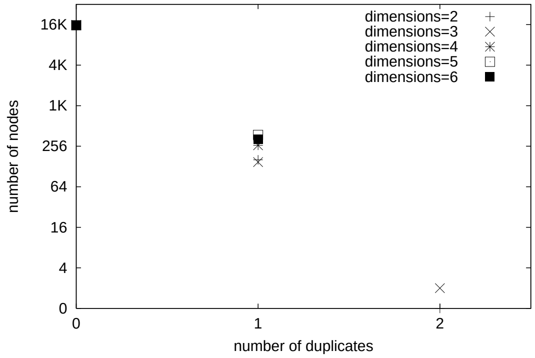


Figure 4.2: *Duplicate messages using CAN-based multicast*

C itself is used as the bootstrap for the formation of C_g as follows: using a well-known hash function, the group address G is deterministically mapped onto a point, say (x, y), and the node on C that owns the point (x, y) serves as the bootstrap node in the construction of C_g. Joining group G thus reduces to joining the CAN C_g. This is done by repeating the usual CAN construction process with (x, y) as the bootstrap node. Because of the light-weight nature of the CAN bootstrap mechanisms, we do not expect the CAN bootstrap node to be overloaded by join requests. If this becomes a possibility however, one could use multiple bootstrap nodes to share the load by using multiple hash functions to deterministically map the group name G onto multiple points in the CAN C; the nodes corresponding to each of these points would then serve as a bootstrap node for the group G. As with the CAN bootstrap process, the failure of the bootstrap node(s) does not affect the operation of the multicast group itself; it only prevents new nodes from joining the group during the period of failure.

Thus, every group has a corresponding CAN made up of all the group members.

Note that with this group formation process a node only maintains state for those groups for which it is itself a member or for which it serves as the bootstrap node. For a d-dimensional CAN, a member node maintains state for $2d$ additional nodes (its neighbors in the CAN), independent of the number of traffic sources in the multicast group.

4.1.2 Multicast forwarding

Because all the members of group G (and no other node) belong to the associated CAN C_g, multicasting to G is achieved by flooding on the CAN C_g. Different flooding algorithms are conceivable; for example, one might consider a naive flooding algorithm wherein a node caches the sequence numbers of messages it has recently received. On receiving a new message, a node forwards the message to all its neighbors (except of course, the neighbor from which it received the message) only if that message is not already in its cache. With this type of flood-cache-suppress algorithm a source can reach every group member without requiring a routing algorithm to discover the network topology. Such an algorithm does not make any special use of the CAN structure and could in fact be run over any application-level topology including a random mesh topology as generated in [19, 4]. The problem with this type of naive flooding algorithm is that it can result in a large amount of duplication of messages; in the worst case, a node could receive a single message from each of its neighbors.

A more efficient flooding solution would be to exploit the coordinate space structure of the CAN as follows:

Assume that our CAN is a d-dimensional CAN with dimensions $1 \ldots d$. Individual nodes thus have at least $2d$ neighbors; 2 per dimension with one to move forward and another to move in reverse along each dimension. *i.e.* for every dimension i a node has at least one neighbor whose zone abuts its own own in the forward direction along i and another neighbor whose zone abuts its own in the reverse direction along i. For example, consider node A in Figure 4.1: node B abuts A in the reverse direction along dimension 1 while nodes C and D abut A in the forward direction along dimension 1.

Messages are then forwarded as follows:

1. The source node (*i.e.* node that generates a new message) forwards a message to all its neighbors

2. Let i be the first dimension along which a node does not overlap with the source. This node then forwards a message to all its neighbors with which it abuts along dimension $1 \ldots (i-1)$ and the neighbors with which it abuts along dimension i in the direction going away from the source node.[2] Figure 4.1 depicts this directed flooding algorithm for a 2-dimensional CAN.

3. a node does not forward a message along a particular dimension if that message has already traversed at least half-way across the space from the source coordinates along that dimension. This rule prevents the flooding from looping round the back of the space.

4. a node caches the sequence numbers of messages it has received and does not forward a message that it has already previously received

[2] In an earlier version of this rule (published in [36]) nodes used the dimension along which it received an incoming message to make forwarding decisions for that message. The resultant algorithm was vulnerable to race conditions under certain dynamics; the above rule avoids this problem by using only the relative positions of the source and the forwarding node to make static forwarding decisions. In both cases however, messages follow the same path. We thank Alec Wolman for pointing out the race condition and discussing the above fix.

For a perfectly partitioned (*i.e.* where nodes have equal sized zones) coordinate space, the above algorithm ensures that every node receives a message exactly once. For imperfectly partitioned spaces however, a node might receive the same message from more than one neighbor. For example, in Figure 4.1, node E would receive a message from both neighbors C and D.

Certain duplicates can be easily avoided because, under normal CAN operation, every node knows the zone coordinates for each of its neighbors. For example, consider once more Figure 4.1; nodes C and D both know each others' and node E's zone coordinates and could hence use a deterministic rule such that only one of them forwards messages to E. Such a rule, however, only eliminates duplicates that arise by flooding along the first dimension. The rule works along the first dimension because, *all* nodes forward along the first dimension. Hence even if a node, by applying some deterministic rule, does not forward a message to its neighbor along the first dimension, we know that some other node that does satisfy the deterministic rule will do so. But this need not be the case when forwarding along higher dimensions. Consider a 3-dimensional CAN; if a node by the application of a deterministic rule decides not to forward to a neighbor along the second dimension, there is no guarantee that any node will eventually forward it up along the second dimension because the node that does satisfy the deterministic rule might receive the packet along the first dimension and hence will not forward the message along the second dimension.[3] For example, in Figure 4.1 let us assume that node A decides (by the use of some deterministic rule) not to forward to node F. Because node C receives the message (from A) along the first dimension, it will not forward the message along the second dimension either and

[3] By the second rule in the flooding algorithm.

hence node F and the other nodes with Y-axis coordinates in the same range as F, will never receive the message. While the above strategy does not eliminate all duplicates, it does eliminate a large fraction of it because most of the flooding occurs along the first dimension. Hence, we augment the above flooding algorithm with the following optimization rule used to eliminate duplicates that arise from forwarding along the first dimension:

- let us assume that a node, P, is to forward a message to its neighboring node Q that abuts P along dimension 1 Consider the corner C_q of Q's zone that abuts P along dimension 1 and has the lowest coordinates along dimensions $2 \ldots d$. Then, P only forwards the message on to Q, if P is in contact with the corner C_q.

So, for example, in Figure 4.1, with respect to nodes C and D, the corner under consideration for node E would be the lower, leftmost corner of E's zone. Hence only D (and not C) would forward messages E in the forward direction along the first dimension.

For the above flooding algorithm, we measured through simulation the percentage of nodes that experienced different degrees of message duplication caused by imperfectly partitioned spaces. Figure 4.2 plots the number of nodes that received a particular number of duplicate messages for a system with 16,384 nodes using CANs with dimensions ranging from 2 to 6. In all cases, over 97% of the nodes receive no duplicate messages and amongst those nodes that do, virtually all of them receive only a single duplicate message. This is a considerable improvement over the naive flooding algorithm wherein *every* node might receive a number of duplicates up to the degree (number of neighbors) of the node.

It is worth noting that the naive flooding algorithm is very robust to message loss because a node can receive a message via any of its neighbors. However, the efficient flooding algorithm is less robust because the loss of a single message results in the breakdown of message delivery to several subsequent nodes thus requiring additional loss recovery techniques. This problem is however, no different than in the case of traditional IP multicast or other application-level schemes where the loss of a packet along a single link results in the packet being lost by all downstream nodes in the distribution tree. With both flooding algorithms, the duplication of messages arises because we do not (unlike most other solutions to multicast delivery) construct a single spanning tree rooted at the source of traffic. However, we believe that the simplicity and scalability gained by not having to run routing algorithms to construct and maintain such delivery trees is well worth the slight inefficiencies that may arise from the duplication of messages.

Using the above flooding algorithm, any group member can multicast a message to the entire group. Nodes that are not group members can also multicast to the entire group by first discovering a random group member and relaying the transmission through this random group member.[4] This random member node can be discovered by contacting the bootstrap node associated with the group name.

4.2 Evaluation

In this section, we evaluate, through simulation, the performance of our CAN-based multicast scheme. We adopt the performance metrics and evaluation strategy used in [19]. As with previous evalution studies of application-level multicasti schemes[19, 4, 51] we compare the performance of CAN-based multicast to native IP multicast and naive unicast-

[4] Note that relaying in our case is different from relayed transmissions as done in source specific multicast [51] because only transmissions from non-member nodes are relayed and even these can be relayed through any member node.

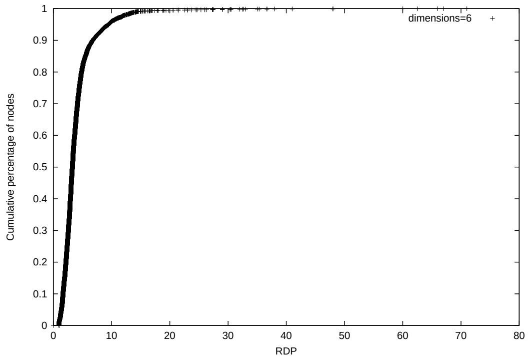


Figure 4.3: *Cumulative distribution of RDP*

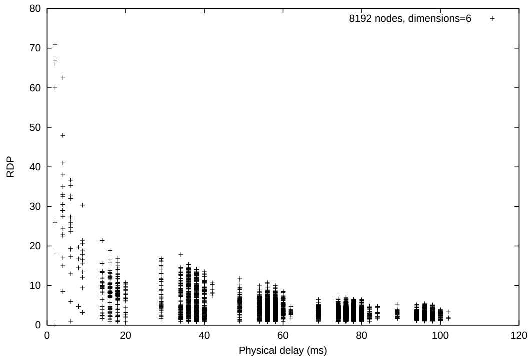


Figure 4.4: *RDP versus physical delay for every group member*

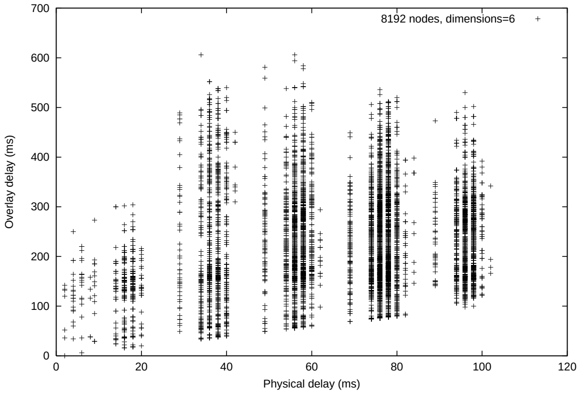


Figure 4.5: *Delay on the overhead versus physical network delay*

based multicast where the source simply unicasts a message to every receiver in succession. Our evaluation metrics are:

- **Relative Delay Penalty (RDP)**: the ratio of the delay between two nodes (in this case, the source node and a receiver) using CAN-based multicast to the unicast delay between them on the underlying physical network
- **Link Stress**: the number of identical copies of a packet carried by a physical link

As before, our simulations were performed on Transit-Stub (TS) topologies.

4.2.1 Relative Delay Penalty

We first present results from a multicast transmission using a single source as this represents the performance typically seen across the different receiver nodes for a transmission from a single source. These simulations were performed using a CAN with 6 dimensions and a group size of 8192 nodes. The source node was selected at random. We used Transit-Stub topologies with link latencies of 20ms for intra-transit domain links, 5ms for stub-transit links and 2ms for intra-stub domain links.

Both IP multicast and Unicast-based multicast achieve an RDP value of one for all group members because messages are transmitted along the direct physical (IP-level) path between the source and receivers. Routing on an overlay network however, fundamentally results in higher delays. Figure 4.3 plots the cumulative distribution of RDP over the group members. While the majority of receivers see an RDP of less than about 5 or 6, a few group members have a high RDP. This can be explained[5] from the scatter-plot in Figure 4.4. The

[5] The authors in [19] make the same observation and explanation.

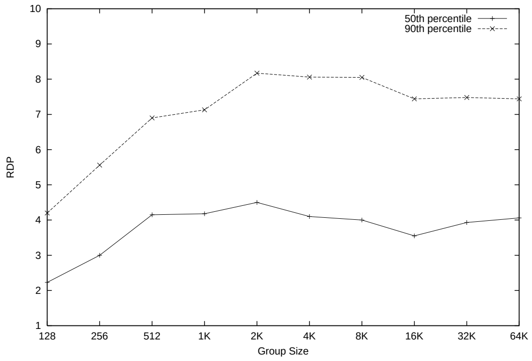


Figure 4.6: *RDP versus increasing group size*

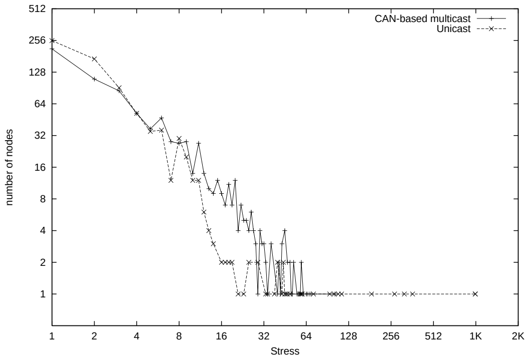


Figure 4.7: *Number of physical links with a given stress*

figure plots the relation between the RDP observed by a receiver and its distance from the source on the underlying IP-level, physical network. Each point in Figure 4.4 indicates the existence of a receiver with the corresponding RDP and IP-level delay. As can be seen, all the nodes with high values of RDP have a low physical delay to the source, i.e. the very low delay from these receivers to the source inflates their RDP. However, the absolute value of their delay from the source on the CAN overlay is not really very high. This can be seen from Figure 4.5, which plots, for every receiver, its delay from the source using CAN multicast versus its physical network delay. The plot shows that while the maximum physical delay can be about 100ms, the maximum delay using CAN-multicast is about 600ms and the receivers on the left hand side of the graph, which had the high RDP, experience delays of not more than 300ms.

Finally, Figure 4.6 plots the 50 and 90 percentile RDP values for group sizes ranging from 128 to 65,000 for a single source. We scale the group size as follows: we take a 1,000 node Transit-Stub topology as before and to this topology, we add end-host (source and receiver) nodes to the stub (leaf) nodes in the topology. The delay of the link from the end-host node to the stub node is set to 1ms. Thus in scaling the group size from a 128 to 65K nodes, we're scaling the density of the graph without scaling the backbone (transit) domain. So, for example, a group size of 128 nodes implies that approximately one in ten stub nodes has an associated group member while a group size of 65K implies that every stub node has approximately 65 attached end-host nodes. This method of scaling the graph causes the flat trend in the growth of RDP with group size because for a given source the relative number of close-by and distant nodes stays pretty much constant. Further, at high density, every CAN node has increasingly many close-by nodes and hence the CAN binning technique used to cluster co-located nodes yields higher gains. Different methods for scaling topologies could yield different scaling trends.

While the significant differences between End-System Multicast and CAN-based multicast makes it hard to draw any direct comparison between the two systems; Figure 4.6 indicates that the performance of CAN-based multicast even for small group sizes is competitive with End-System multicast.

4.2.2 Link Stress

Ideally, one would like the stress on the different physical links to be somewhat evenly distributed. Using native IP multicast, every link in the network has a stress of exactly one. In the case of unicasting from the source directly to all the receivers, links close to the source node have very high stress (equal to the group size at the first hop link from the source). Figure 4.7 plots the number of nodes that experienced a particular stress value for a group size of 1024 for a 6-dimensional CAN. Unlike naive unicast where a small number of links see extremely high stress, CAN-based multicast distributes the stress much more evenly over all the links.

Figure 4.8 plots the worst-case stress for group sizes ranging from 128 to 65,000 nodes. The high stress in the case of large group sizes is because, as described earlier, we scale the group size without scaling the size of the backbone topology. For the above simulation, we used a transit-stub topology with a 1,000 nodes. Hence for a group size of 65,000 nodes, all 65,000 nodes are interconnected by a backbone topology of less than 1,000 nodes thus putting high stress on some backbone links. We repeated the above simulation for a transit-stub topology with 10,000 nodes, thus decreasing the density of the graph by a factor of 10. Figure 4.9 plots the worst-case stress for group sizes up to 2,048 nodes for all three cases (*i.e.* CAN-based multicast using Transit-Stub topologies with 1,000 and 10,000 nodes and naive unicast-based multicast). As can be seen, at lower density the worst-case stress drops sharply. For example, at 2,048 nodes the worst case stress drops from 169 (for TS1000) to 37 (for TS10000). Because, in practice, we do not expect very high densities of group member nodes relative to the Internet topology itself, worst-case stress using CAN-based multicast should be at a reasonable level. In future work, we intend looking into techniques that might further lower this stress value.

4.3 Related Work

The case for application-level multicast as a more tractable alternative to a network-level multicast service was first put forth in [19, 13, 4].

The End-system multicast [19] work proposes an architecture for multicast over

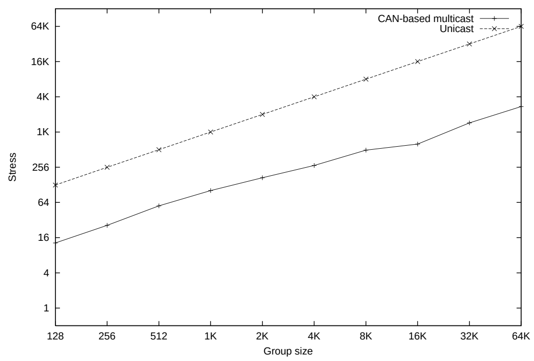


Figure 4.8: *Stress versus increasing group size*

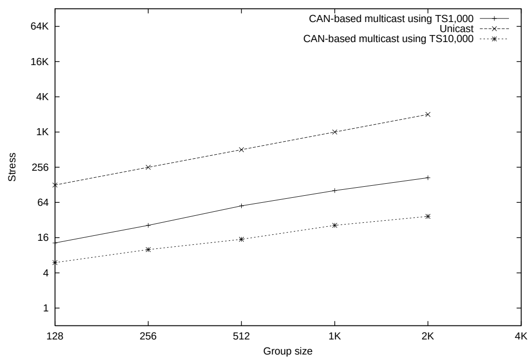


Figure 4.9: *Effect of topology density on stress*

small and sparse groups. End-system multicast builds a mesh structure across participating end-hosts and then constructs source-rooted trees by running a routing protocol over this mesh. The authors also study the fundamental performance penalty associated with such an application-level model. The authors in [4] argue for infrastructure support to tackle the problem of content distribution over the Internet. The Scattercast architecture relies on proxies deployed within the network infrastructure. These proxies self-organize into an application-level mesh over which a global routing algorithm is used to construct distribution trees. In terms of being a solution to application-level multicast, the key difference between our work and the End-System multicast and Scattercast work is the potential for CAN-based multicast to scale to large group sizes.

Yoid [13] proposes a solution to application-level multicast wherein a spanning tree is directly constructed across the participating nodes without first constructing a mesh structure. The resultant protocols are more complex because the tree-first approach results in expensive loop detection and avoidance techniques and must be made resilient to partitions.

Overcast[21] is a scheme for source-specific, reliable multicast using an overlay network. Overcast constructs efficient dissemination trees rooted at the single source of traffic. The overlay network in Overcast is composed of nodes that reside within the network infrastructure. This assumption of the existence of permanent storage within the network distinguishes Overcast from CANs and indeed, from most of the other systems described above. Unlike Overcast, CANs can be composed entirely from end-user machines with no form of central authority.

Tapestry [50] is a wide-area overlay routing and location infrastructure that, like CANs, embeds nodes in a well-defined virtual address space to provide a DHT interface. Bayeux [51] is a source-specific, application-level multicast scheme that leverages the Tapestry routing infrastructure. To join a multicast session, Bayeux nodes send *JOIN* messages to the source node. The source replies to a *JOIN* request by routing a *TREE* message, on the Tapestry overlay, to the requesting node. This *TREE* message is used to set up state at intermediate nodes along the path from the source node to the new member. Similarly, a *LEAVE* message from an existing member triggers a *PRUNE* message from the root, which removes the appropriate forwarding state along the distribution tree.

Similar to Bayeux, is the Scribe [40] multicast scheme, built on top of Pastry. Like Bayeux, Scribe explicitly constructs dissemination trees on top of the Pastry overlay the key difference being that not all *JOIN* messages are forwarded all the way to the root of the tree thus improving the scalability of the tree construction algorithm.

Bayeux, Scribe and M-CAN are similar in that they achieve scalability by leveraging the scalable routing infrastructure provided by systems like CAN, Tapestry and Pastry. In terms of service model, Bayeux supports only source-specific multicast while Scribe supports multiple sources by tunneling all transmissions to the root of the tree from which the message is multicast; M-CAN however allows any group member to act as a traffic source without funneling all transmissions through a single point thus avoid the creation of potential bottleneck nodes. In terms of design, Bayeux and Scribe use an explicit protocol to set-up and tear down distribution trees. M-CAN by contrast, fully exploits the CAN structure because of which messages can be forwarded without requiring a routing protocol to explicitly construct distribution trees.

Finally, M-CAN chooses to *extend* the basic CAN routing algorithm. An alternate approach, recently proposed in [46], is to leave the underlying routing algorithm unchanged and provide multicast functionality using only the DHT *lookup* operation. Understanding the relative advantages and disadvantages of the two approaches is an area of future research that is currently beyond the scope of our work.

Chapter 5

DHT Routing: Related Algorithms And Some Open Questions

Even though they were introduced only a few years ago, peer-to-peer (P2P) file-sharing systems are now one of the most popular Internet applications and have become a major source of Internet traffic. Further, several recent research projects are proposing large-scale distributed applications layered over DHTs. Thus, it is extremely important that these systems be scalable. As mentioned in Chapter 1, several research groups, motivated by the scaling problems of currently deployed systems, have (independently) proposed a new generation of scalable indexing systems that support a *distributed hash table* (DHT) functionality; among them are Tapestry [50], Pastry [39], Chord [47], and Content-Addressable Networks (CAN) [35].

The core of these DHT systems is the routing algorithm. The DHT nodes form an overlay network with each node having several other nodes as neighbors. When a `lookup(key)` is issued, the lookup is routed through the overlay network to the node responsible for that key. The scalability of these DHT algorithms is tied directly to the efficiency of their routing algorithms. Each of the proposed DHT systems listed above – Tapestry, Pastry, Chord, and CAN – employ a different routing algorithm, and each algorithm embodies some insights about routing in overlay networks. Thus, we believe an appropriate goal would be to combine these insights, and seek new insights, to produce even better algorithms. In that spirit we end this thesis with a discussion of issues relevant to routing algorithms and identify some open research questions[38]. Of course, our list of questions is not intended to be exhaustive, merely illustrative.

As should be clear by our description, this section is not about finished work, but instead is about a research agenda for future work (by us and others). We should also note that there are many other interesting issues that remain to be resolved in these DHT systems, such as security and robustness to attacks, system monitoring and maintenance, and indexing and keyword searching. Our focus on routing algorithms is not intended to imply that these other issues are of secondary importance.

We start this Chapter with a brief overview, in Section 5.1, of ongoing work on distributed applications that build on top of DHTs. We then (very) briefly review the routing algorithms used in the various DHT systems in Section 5.2 and, in the concluding sections, discuss various issues relevant to routing: state-efficiency tradeoff, resilience to failures, routing hotspots, geography, and heterogeneity.

5.1 Applications

The DHT systems – Tapestry, Pastry, Chord and CAN – were introduced only a little over a year ago but the DHT functionality is already proving to be a useful substrate for large distributed systems and a number of projects are proposing to build Internet-scale facilities layered above DHTs. We end with a brief overview of the ongoing work in the space of DHT-based applications.

DHTs are a hash table; the most obvious use for them is as indexing systems and many ongoing projects are doing just that:

- Global file systems such as PAST [11], CFS [9] and OceanStore [25] use DHTs to locate data within the file store.

- BackSlash [45], a collaborative web mirroring system run by a collective of web sites that wish to protect themselves from flash crowds, uses DHTs to locate mirrored content in a timely manner.

- The JXTA group at SUN Microsystems is building a CAN-based Chat service [2] that uses DHTs to locate resources such as person identifiers or group names.

- The Geographic Hash Table (GHT) [44, 34] is a DHT for large-scale networks of sensors. Although functionally equivalent to DHTs, GHT does not directly adopt the routing algorithms of the DHTs (CAN, Chord, *etc.*) but instead uses GPSR [22], an ad-hoc geographic routing algorithm better suited to wireless environments.

In addition to their use as indexing systems, a number of projects are looking to use the DHT technology to enable richer higher-level services:

- The Herald [3] and Scribe [40] projects build event notification services that use DHTs for scalable multicasting to large numbers of subscribers [51, 36, 40].
- DHTs provide scalable "exact-match" lookups but lack the rich functionality (*e.g.,* finding correlations over stored data) of traditional database query languages. The PIER project is exploring the design and implementation of complex database query facilities over DHTs [18].
- The *i3* project [46] proposes the deployment of a general-purpose *Internet Indirection Infrastructure* – a DHT-based infrastructure that provides a rendezvous-based communication abstraction. The authors show that a range of services such as multicast, anycast and mobility can be provided using this single rendezvous-based communication primitive thus offering a unified solution to providing rich communication services.

5.2 DHT Routing

This section reviews some of the existing routing algorithms (for completeness, we also include a description of CAN). All of them take, as input, a `key` and, in response, route a message to the node responsible for that key. The keys are strings of digits of some length. Nodes have identifiers, taken from the same space as the keys (*i.e.,* , same number of digits). Each node maintains a routing table consisting of a small subset of nodes in the system. When a node receives a query for a key for which it is not responsible, the node routes the query to the neighbor node that makes the most "progress" towards resolving the query. The notion of progress differs from algorithm to algorithm, but in general is defined in terms of some distance between the identifier of the current node and the identifier of the queried key.

Plaxton *et al.*: Plaxton *et al.* [33] developed perhaps the first routing algorithm that could be scalably used by DHTs. While not intended for use in P2P systems, because it assumes a relatively static node population, it does provide very efficient routing of lookups. The routing algorithm works by "correcting" a single digit at a time: if node number 36278 received a lookup query with key 36912, which matches the first two digits, then the routing algorithm forwards the query to a node which matches the first three digits (*e.g.*, node 36955). To do this, a node needs to have, as neighbors, nodes that match each prefix of its own identifier but differ in the next digit. For example, node 36278 would have the following as neighbors:

- Nodes with prefixes 31, 32, 33, ... to allow it to correct the second digit
- Nodes with prefixes 361, 362, 363, ... to allow it to correct the third digit
- ...

For a system of n nodes, each node has on the order of $O(\log n)$ neighbors. Since one digit is corrected each time the query is forwarded, the routing path is at most $O(\log n)$ overlay (or application-level) hops.

This algorithm has the additional property that if the n^2 node-node latencies (or "distances" according to some metric) are known, the routing tables can be chosen to minimize the expected path latency and, moreover, the latency of the overlay path between two nodes is within a constant factor of the latency of the direct underlying network path between them.

Tapestry: Tapestry [50] uses a variant of the Plaxton *et al.* algorithm. The modifications are to ensure that the design, originally intended for static environments, can adapt to a dynamic node population. The modifications are too involved to describe in this short review. However, the algorithm maintains the properties of having $O(\log n)$ neighbors and routing with path lengths of $O(\log n)$ hops.

Pastry: In Pastry [39], nodes are responsible for keys that are the closest numerically (with the keyspace considered as a circle). The neighbors consist of a *Leaf Set* L which is the set of $|L|$ closest nodes (half larger, half smaller). Correct, not necessarily efficient, routing can be achieved with this leaf set. To achieve more efficient routing, Pastry has another set of neighbors spread out in the key space (in a manner we don't describe here). Routing consists of forwarding the query to the neighboring node that has the longest shared prefix with the key (and, in the case of ties, to the node with identifier closest numerically to the key). Pastry has $O(\log n)$ neighbors and routes within $O(\log n)$ hops.

Chord: Chord [47] also uses a one-dimensional circular key space. The node responsible for the key is the node whose identifier most closely follows the key (numerically); that node is called the key's *successor*. Chord maintains two sets of neighbors. Each node has a *successor list* of k nodes that immediately follow it in the key space. Routing correctness is achieved with these lists. Routing efficiency is achieved with the *finger list* of $O(\log n)$ nodes spaced exponentially around the key space. Routing consists of forwarding to the node closest, but not past, the key; pathlengths are $O(\log n)$ hops.

CAN: CAN chooses its keys from a d-dimensional toroidal space. Each node is associated with a hypercubal region of this key space, and its neighbors are the nodes that "own" the contiguous hypercubes. Routing consists of forwarding to a neighbor that is closer to the key. CAN has a different performance profile than the other algorithms; nodes have $O(d)$ neighbors and pathlengths are $O(dn^{\frac{1}{d}})$ hops. Note, however, that when $d = \log n$, CAN has $O(\log n)$ neighbors and $O(\log n)$ pathlengths like the other algorithms.

5.3 State-Efficiency Tradeoff

The most obvious measure of the efficiency of these routing algorithms is the resulting pathlength. Most of the algorithms have pathlengths of $O(\log n)$ hops, while CAN has longer paths of $O(dn^{\frac{1}{d}})$. The most obvious measure of the overhead associated with keeping routing tables is the number of neighbors which is primarily a measure of how much state needs to be adjusted when nodes join or leave. Most of the algorithms require $O(\log n)$ neighbors, while CAN requires only $O(d)$ neighbors.

Ideally, one would like to combine the best of these two classes of algorithms in hybrid algorithms that achieve short pathlengths with a fixed number of neighbors.

Question 1 *Can one achieve $O(\log n)$ pathlengths (or better) with $O(1)$ neighbors? If so, are there other properties (such as those described in the following sections) that are made worse in these hybrid routing algorithms?*

The Viceroy routing algorithm recently proposed by Malkhi *et al.* [26], achieves $O(\log n)$ pathlengths with $O(1)$ neighbors, thus answering the first of the above questions; the latter question – whether some other aspect of routing gets worse – is still, to the best of our knowledge, an open question.

5.4 Resilience to Failures

Because P2P nodes are notoriously transient, the resilience of routing to failures is a very important consideration. There are (at least) three different aspects to resilience.

First, one needs to evaluate whether routing can continue to function (and with what efficiency) as nodes fail without any time for other nodes to establish other neighbors to compensate. In Chapter 2.1 we termed this *static resilience* and measured it in terms of the percentage of reachable key locations and of the resulting increase in path length.

Question 2 *Can one characterize the static resilience of the various algorithms? What aspects of these algorithms lead to good resilience?*

Second, one can investigate the resilience when nodes have a chance to establish some neighbors, but not all. That is, when nodes have certain “special” neighbors, such as the *successor list* or the *Leaf Set*, and these are re-established after a failure, but no other neighbors are re-established (such as the *finger set*). The presence of these special neighbors allow one to prove the correctness of routing, but the following question remains:

Question 3 *To what extent are the observed path lengths better than the rather pessimistic bounds provided by the presence of these special neighbors?*

Finally, one can ask how long it takes various algorithms to fully recover their routing state, and at what cost (measured, for example, by the number of nodes participating in the recovery or the number of control messages generated for recovery).

Question 4 *How long does it take, on average, to recover complete routing state? And what is the cost of doing so?*

A related question is:

Question 5 *Can one identify design rules that lead to shorter and/or cheaper recoveries?*

For instance, is symmetry (where the node neighbor relation is symmetric) important in restoring state easily? One could also argue that in the face of node failure, having the routing automatically send messages to the correct alternate node (*i.e.* the node that takes over the range of the identifier space that was previously held by the failed node) leads to quicker recovery.

5.5 Routing Hot Spots

When there is a hotspot in the query pattern, with a certain key being requested extremely often, then the node holding that key may become overloaded. Various caching and replication schemes have been proposed to overcome this *query hotspot* problem; the effectiveness of these schemes may vary between algorithms based on the fan-in at the node and other factors, but this seems to be a manageable problem. More problematic, however, is if a node is overloaded with too much routing traffic. These *routing hotspots* are harder to deal with since there is no local action the node can take to redirect the routing load. Some of the *proximity* techniques we describe below might be used to help here, but otherwise this remains an open problem.

Question 6 *Do routing hotspots exist and, if so, how can one deal with them?*

5.6 Incorporating Geography

As stated in Chapter 3, the true efficiency measure is the end-to-end latency of the path. While the original "vanilla" versions of some of these routing algorithms did not take these hop latencies into account, almost all of the "full" versions of the algorithms make some attempt to deal with the geographic proximity of nodes.

Proximity Routing: Proximity routing is when the routing choice is based not just which neighboring node makes the "most" progress towards the key, but is also based on which neighboring node is "closest" in the sense of latency. Various algorithms implement proximity routing differently, but they all adopt the same basic approach of weighing progress in identifier space against cost in latency (or geography). Simulations have shown this to be a very effective tool in reducing the average path latency.

Question 7 *Can one formally characterize the effectiveness of these proximity routing approaches?*

Proximity Neighbor Selection: Here the proximity criterion is applied when choosing neighbors, not just when choosing the next hop.

Question 8 *Can one show that proximity neighbor selection is always better than proximity routing? Is this difference significant?*

As mentioned earlier, if the n^2 node-pair distances (as measured by latency) are known, the Plaxton/Tapestry algorithm can choose the neighbors so as to minimize the expected overlay path latency. This is an extremely important property, that is (so far) the exclusive domain of the Plaxton/Tapestry algorithms. We don't know whether other algorithms can adopt similar approaches.

Question 9 *If one had the full n^2 distance matrix, could one do optimal neighbor selection in algorithms other than Plaxton/Tapestry?*

Geographic Layout: In most of the algorithms, the node identifiers are chosen randomly (*e.g.* hash functions of the IP address, etc.) and the neighbor relations are established based solely on these node identifiers. One could instead attempt to choose node identifiers in a geographically informed manner.[1] Our initial attempt to do so in the context of CAN was reported on in [37]; this approach was quite successful in reducing the latency of paths. There was little in the layout method specific to CAN, but the high-dimensionality of the key space may have played an important role; recent work [32] suggests that latencies in the Internet can be reasonably modeled by a d-dimension geometric space with $d \geq 2$. This raises the question of whether systems that use a one-dimensional key set can adequately mimic the geographic layout of the nodes.

Question 10 *Can one choose identifiers in a one-dimensional key space that will adequately capture the geographic layout of nodes?*

However, this may not matter because the geographic layout may not offer significant advantages over the two proximity methods.

Question 11 *Can the two* local *techniques of proximity routing and proximity neighbor selection achieve most of the benefit of* global *geographic layout?*

[1] Note that geographic layout differs from the two above *proximity* methods in that here there is an attempt to affect the global layout of the node identifiers, whereas the proximity methods merely affect the local choices of neighbors and forwarding nodes.

Moreover, these geographically-informed layout methods may interfere with the robustness, hotspot, and other properties mentioned in previous sections.

Question 12 *Does geographic layout have an impact on resilience, hotspots, and other aspects of performance?*

5.7 Extreme Heterogeneity

All of the algorithms start by assuming that all nodes have the same capacity to process messages and then, only later, add on techniques for coping with heterogeneity.[2] However, the heterogeneity observed in current P2P populations [41] is quite extreme, with differences of several orders of magnitude in bandwidth. One can ask whether the routing algorithms, rather than merely *coping* with heterogeneity, should instead use it to their *advantage*. At the extreme, a star topology with all queries passing through a single hub node and then routed to their destination would be extremely efficient, but would require a very highly capable nub node (and would have a single point of failure). But perhaps one could use the very highly capable nodes as mini-hubs to improve routing. In another position paper here, some of us argue that heterogeneity can be used to make Gnutella-like systems more scalable. The question is whether one could similarly modify the current DHT routing algorithms to exploit heterogeneity:

Question 13 *Can one redesign these routing algorithms to exploit heterogeneity?*

It may be that no sophisticated modifications are needed to leverage heterogeneity. Perhaps the simplest technique to cope with heterogeneity, one that has already been

[2] The authors of [41] deserve credit for bringing the issue of heterogeneity to our attention.

mentioned in the literature, is to *clone* highly capable nodes so that they could serve as multiple nodes; *i.e.*, a node that was 10 times more powerful than other nodes could function as 10 virtual nodes.[3] When combined with proximity routing and neighbor selection, cloning would allow nodes to route to themselves and thereby "jump" in key space without any forwarding hops.

Question 14 *Does cloning plus proximity routing and neighbor selection lead to significantly improved performance when the node capabilities are extremely heterogeneous?*

[3] This technique has already been suggested for some of the algorithms, and could easily be applied to the others. However, in some algorithms it would require alteration in the way the node identifiers were chosen so that they weren't tied to the IP address of the node.

Bibliography

[1] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *32nd Annual ACM Symposium on Theory of Computing*. ACM, 2000.

[2] CAN based Chat. http://di.jxta.org/, 2001.

[3] Luis Felipe Cabrera, Michael B. Jones, and Marvin Theimer. Herald: Achieving a global event notification service. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau/Oberbayern, Germany, May 2001.

[4] Yatin Chawathe, Steven McCanne, and Eric Brewer. An architecture for internet content distribution as an infrastructure service. available at http://www.cs.berkeley.edu/ yatin/papers, 2000.

[5] Yan Chen and Randy Katz. On the placement of network monitoring sites. http://www.cs.berkeley.edu/ yanchen/wnms/, 2001.

[6] Open Source Community. The free network project - rewiring the internet. 2001.

[7] Open Source Community. Gnutella. 2001.

[8] Adam Costello. Private communication, December 2000.

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

[10] Stephen E. Deering. *Multicast Routing in a Datagram Internetwork.* PhD thesis, Stanford University, December 1991.

[11] Peter Druschel and Antony Rowstron. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS 2001)*, pages 65–70, Elmau/Oberbayern, Germany, May 2001.

[12] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-law Relationships of the Internet Topology. In *Proceedings of SIGCOMM '99*, Cambridge, MA, September 1999. ACM.

[13] Paul Francis. Yoid: Extending the internet multicast architecture. Unpublished paper, available at http://www.aciri.org/yoid/docs/index.html, April 2000.

[14] Paul Francis, Sugih Jamin, Vern Paxson, Lixia Zhang, Daniel Gryniewicz, and Yixin Jin. An Architecture for a Global Internet Host Distance Estimation Service. In *Proceedings IEEE Infocom '99*, New York, NY, March 1999.

[15] Andrew Frank. The copyright crusade ii. June 2002.

[16] Jimmy Guterman. Gnutella to the Rescue ? Not so Fast, Napster fiends. link to article at http://gnutella.wego.com, September 2000.

[17] T. Hansen, J. Otero, T. McGregor, and H. Braun. Active measurement data analysis techniques. htpp://amp.nlanr.net.

[18] Matthew Harren, J. Hellerstein, Ryan Huebsch, B. Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in dht-based peero-to-peer networks. In *Proceedings of First International Peer-to-Peer Systems Workshop IPTPS 2002*, March 2002.

[19] Yang hua Chu, Sanjay Rao, and Hui Zhang. A case for end system multicast. In *Proceedings of SIGMETRICS 2000*, Santa Clara, CA, June 2000.

[20] Van Jacobson and Steven McCanne. *Visual Audio Tool.* Lawrence Berkeley Laboratory.

[21] John Jannotti, David Gifford, Kirk Johnson, Frans Kaashoek, and James O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[22] Brad Karp and H. Kung. Greedy Perimeter Stateless Routing. In *Proceedings of ACM Conf. on Mobile Computing and Networking (MOBICOM)*, Boston, MA, 2000. ACM.

[23] Isidor Kouvelas, Vicky Hardman, and Jon Crowcroft. Network adaptive continuous-media applications through self organised transcoding. In *Proceedings of the Network and Operating Systems Support for Digital Audio and Video*, Cambridge, U.K., July 1998.

[24] Balachander Krishnamurthy and Jia Wang. On network-aware clustering of web clients. In *Proceedings of SIGCOMM '00*, Stockholm, Sweden, August 2000.

[25] 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. In *Proceedings of ASPLOS 2000*, Cambridge, Massachusetts, November 2000.

[26] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the Twenty-First Symposium on Principles of Distributed Computing (PODC)*, Monterey, CA, July 2002.

[27] Steven McCanne. A distributed whiteboard for network conferencing, May 1992. U.C. Berkeley CS268 Computer Networks term project and paper.

[28] Steven McCanne. Symposium on the acceleration of rich media on the Internet: Internet Broadcast Networks: Mass Media Distribution for the 21st Century, May 1999.

[29] Steven McCanne and Van Jacobson. *VIC: video conference.* Lawrence Berkeley Laboratory and University of California, Berkeley.

[30] MojoNation. http://www.mojonation.com.

[31] Jungle Monkey. http://www.junglemonkey.net.

[32] Eugene Ng and Hui Zhang. Towards Global Network Positioning. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, November 2001.

[33] C.Greg Plaxton, Rajmohan Rajaram, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

[34] S. Ratnasamy, B. Karp, Li Yin, Fang Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash-table for data-centric storage in sensornets. In *First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, September 2002.

[35] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of SIGCOMM 2001*, August 2001.

[36] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Proceedings of NGC*, London, UK, November 2001.

[37] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Topologically-aware overlay construction and server selection. In *Proceedings of Infocom 2002*, June 2002.

[38] Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. Routing algorithms for DHTs: Some open questions. In *Proceedings of First International Peer-to-Peer Systems Workshop IPTPS 2002*, March 2002.

[39] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelburg, Germany, 2001.

[40] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: A large-scale

and decentralized application-level multicast infrastructure. In *Proceedings of NGC*, London, UK, November 2001.

[41] Stefan Saroiu, Krishna Gummadi, and Steve Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Conferencing and Networking*, San Jose, January 2002.

[42] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, November 2002.

[43] Scott Shenker. Peer-to-peer computing: From exciting social revolution to boring academic research. Presentation at The Jon Postel Distinguished Lecture Series, UCLA, 2001.

[44] Scott Shenker, S. Ratnasamy, B. Karp, Ramesh Govindan, and Deborah Estrin. Data-centric storage in sensornets. In *First Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[45] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-peer caching schemes to address flash crowds. In *Proceedings of First International Peer-to-Peer Systems Workshop IPTPS 2002*, March 2002.

[46] Ion Stoica, Dan Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of SIGCOMM 2002*, August 2002.

[47] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM 2001*, August 2001.

[48] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Network topologies, power laws and hierarchy. Technical Report TR01-746, Technical Report, University of Southern California, 2001.

[49] Ellen Zegura, Ken Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings IEEE Infocom '96*, San Francisco, CA, May 1996.

[50] Ben Y. Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. available at http://www.cs.berkeley.edu/ ravenben/tapestry/, 2001.

[51] Shelley Q. Zhuang, Ben Zhao, Anthony Joseph, Randy Katz, and John Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the Eleventh International Workshop on Network and OS Support for Digital Audio and Video*, New York, July 2001. ACM.